Lecture 1 — Overview and Sequencing the Genome

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — August 28, 2012

1 Administrivia

Welcome to 15-210 Parallel and Sequential Data Structures and Algorithms. This course will teach you methods for designing, analyzing, and programming sequential and parallel algorithms and data structures. The an emphasis will be on fundamental concepts that will be applicable across a wide variety of problem domains, and transferable across a broad set of programming languages and computer architectures. *There is no textbook for the class.* We will (electronically) distribute lecture notes and supplemental reading materials as we go along. We will try to generate a draft of lecture notes and post them before class..

The course web site is located at

`http://www.cs.cmu.edu/~15210`

Please take time to look around. While you're at it, we *strongly* encourage you to read and understand the collaboration policy.

Instead of spamming you with mass emails, we will post announcements, clarifications, corrections, hints, etc. on the course website and piazza—please check them on a regular basis.

There will be weekly assignments (generally due at 11:59pm on Mondays), 2 exams, and a final. The first assignment is coming out tomorrow and will be due *at 11:59pm on Wednesday Sept 5, 2012*. (Note the unusual date due to Labor Day and it being the first assignment)

Since we are still an early incarnation of 15-210, we would appreciate feedback any time. Please come talk to us if you have suggestions and/or concerns.

2 Course Overview

This is a data structures and algorithms course and centers around the following themes:

- defining precise problems and abstract data types (the interfaces)
- designing and programming correct and efficient algorithms and data structures for the given problems and data types (the implementations)

and can be summarized by the following table:

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

	Interface	**Implementation**
Functions	Problem	Algorithm
Data	Abstract Data Type	Data Structure

A *problem* specifies precisely the intended input/output behavior in an abstract form. It is an abstract (precise) definition of the problem but does not describe how it is solved. Whereas an *algorithm* is what allows us to solve a problem; it is an implementation that meets the intended specification. Typically, a problem will have many algorithmic solutions. For example, sorting is a problem—it specifies what the input is (e.g., a sequence of numbers) and the intended output (e.g., an ordered sequence of numbers). On the other hand `quicksort` and insertion sort are algorithms for solving the sorting problem. The distinction between problems vs. algorithms is standard.

Similarly, an *abstract data type* (ADT) specifies precisely an interface for accessing data in an abstract form without specifying how the data is structured, whereas a *data structure* is a particular way of organizing the data to support the interface. For an ADT the interface is specified in terms of a set of operations on the type. For example, a priority queue is an ADT with operations that might include `insert`, `findMin`, and `isEmpty?`. Various data structures can be used to implement a priority queue, including binary heaps, arrays, and balanced binary trees. The terminology ADTs vs. data structures is not as widely used as problems vs. algorithms. In particular, sometimes the term data structure is used to refer to both the interface and the implementation. We will try to avoid such usage in this class.

There are several reasons it is very important to keep a clean distinction between interface and implementation. One critical reason is to enable proofs of correctness, e.g. to show that an algorithm properly implements an interface. Many software disasters have been caused by badly defined interfaces. Another equally important reason is to enable reuse and layering of components. One of the most common techniques to solve a problem is to reduce it to another problem for which you already know algorithms and perhaps already have code. We will look at such an example today. A third reason is that when we compare the performance of different algorithms or data structures it is important we are not comparing apples with oranges, i.e., we have to make sure the algorithms we compare are solving the same problem and subtle differences in the problem specification can make very large differences in the difficulty of solving a problem efficiently.

For these reasons, in this course we will put a strong emphasis on helping you define precise and concise abstractions and then implementing those abstractions using algorithms and data structures. When discussing algorithmic solutions to problems we are not just going to give a list of algorithms and data structures, but instead will emphasize general techniques that can be used to design them, such as divide-and-conquer, the greedy method, dynamic programming, and balance trees. It is important that in this course you learn how to design your own algorithms/data structures given an interface, and even how to specify your own problems/ADTs given a task at hand.

Parallelism. This course will differ in an important way from traditional algorithm and data structure in that we will be working with thinking about parallelism right from the start. We are doing this since in recent years parallelism has become ubiquitous in computing platforms ranging from cell phones to supercomputers. Due to physical and economical constraints, a typical laptop or desktop machine we can buy today has 4 to 8 computing cores, and soon this number will be 16, 32, and 64.

While the number of cores grows at a rapid pace, the per-core speed has not increased much over the past several years and is unlikely to increase much in the future. Therefore to improve performance it is and will become more important to make use of multiple cores. Furthermore most computers have some form of graphical processing unit (GPU) which are themselves highly parallel. To get good performance on these it is critical to use parallelism. Working with parallel algorithms often requires a somewhat different way of thinking than working with sequential algorithms. Learning how to think parallel is an important goal of this class.

You might ask how much advantage does one get from writing a parallel instead of a sequential algorithm to solve a problem. Here are some example timings to give you a sense of what can be gained. These are on a 32 core commodity server machine (you can order one on the Dell web site).

	Serial	**Parallel**	
		1-core	32-core
Sorting 10 million strings	2.9	2.9	.095
Remove duplicates 10M strings	.66	1.0	.038
Min spanning tree 10M edges	1.6	2.5	.14
Breadth first search 10M edges	.82	1.2	.046

In the table, the Serial timings use sequential algorithms while the Parallel timings use parallel algorithms. Notice that the speedup for the parallel 32 core version relative to the sequential algorithm ranges from approximately 12 (min spanning tree) to approximately 32 (sorting). It is unlikely that you will get similar speedup using Standard ML, but maximizing speedup by highly tuning an implementation is not the goal of this course. That is an aim of 15-213.

Instead the goal of this class is to think about parallelism at a high-level which includes learning general techniques for designing parallel algorithms and data structures, and learning how to approximately analyze their costs. As in 15-150, in this class we will be using *work* and *span* to analyze costs.

The course also differs from most traditional algorithms and data structures courses in that we will be using a purely functional model of computation. The primary reason for this is that the functional model is safe for parallelism. In an imperative setting one needs to worry about race conditions since parallel threads of execution might modify shared states in different orders from one run of the code to the next. This makes it much harder to analyze code. However, all the ideas we will cover are also relevant to imperative languages—one just needs to be much more careful when coding imperatively. The functional style also encourages working at a higher level of abstraction by using higher order functions. This again is useful for parallel algorithms since it moves away from the one-at-a-time loop mentality that is pervasive in sequential programming.

This all being said, most of what is covered in a traditional algorithms course will be covered in this course, but perhaps in a somewhat different way.

3 An Example: Sequencing the Genome

As an example of how to define a problem and develop parallel algorithms that solve it we consider the task of sequencing the genome. Sequencing of a complete human genome represents one of the greatest scientific achievements of the century. The efforts started a few decades ago and includes the following major landmarks:

1996	sequencing of first living species
2001	draft sequence of the human genome
2007	full human genome diploid sequence

Interestingly, efficient parallel algorithms played a crucial role in all these achievements. In this lecture, we will take a look at some algorithms behind the recent results—and the power of problem abstraction which will reveal surprising connections between seemingly unrelated problems.

3.1 What makes sequencing the genome hard?

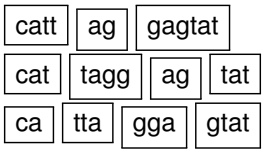
There is currently no way to read long strands with accuracy. Current DNA sequencing machines are only capable of efficiently reading relatively short strands (e.g. 1000 base pairs). Therefore, we resort to cutting strands into shorter fragments and then reassembling the pieces. A technique called "primer walking" can be used to cut the DNA strands into consecutive fragments and sequence each one. But the process is slow because one needs the result of one fragment to "build" in the wet lab the molecule needed to find the following fragment. This is an inherently sequential process with large waiting time between each iteration. Alternatively, there are fast methods to cut the strand at random positions. But this process mixes up the short fragments, so the order of the fragments is unknown. For example, the strand cattaggagtat might turn into, say, ag, gag, catt, tat, destroying the original ordering.

If we could make copies of the original sequence, is there something we could do differently to order the pieces? Let's look at the shotgun method, which according to Wikipedia is the de facto standard for genome sequencing today. It works as follows:

1. Take a DNA sequence and make multiple copies. For example, if we are cattaggagtat, we produce many copies of it:

   cattaggagtat
   cattaggagtat
   cattaggagtat

2. Randomly cut up the sequences using a "shotgun", well, actually using radiation or chemicals. For instance, we could get



3. Sequence each of the short fragments, which can be done in parallel with multiple sequencing machines.

4. Reconstruct the original genome from the fragments.

Steps 1–3 are done in a wet lab, and **Step 4 is where algorithms come in.** In Step 4, it is not always possible to reconstruct the exact original genome and indeed there might be many DNA strings that lead to the same collection of fragments (consider, for example, just repeating the original string). We therefore want to solve a problem that is of the form: *Given a set of overlapping genome subsequences, construct the "best" sequence that includes them all.* However, this is still not a well formed problem. What does it mean to be the "best" sequence? There are many possible candidates. Below is one way to define "best" objectively.

Definition 3.1 (The Shortest Superstring (SS) **Problem**)**.** Given an alphabet set Σ and a set of finite strings $S \subseteq \Sigma^+$, return a shortest string r that contains every $s \in S$ as a substring of r.

In this definition the notation Σ^+ means the set of all possible non-empty strings consisting of characters Σ. Note that in this definition, we require each $s \in S$ to appear as a contiguous block in r. That is, "ag" is a substring of "ggag" but is *not* a substring of "attg".

That is, given sequence fragments, construct the shortest string that contains all the fragments. The idea is that the simplest string is the best. Now that we have a concrete specification of the problem, we are ready to look into algorithms for solving it.

For starters, let's observe that we can ignore strings that are contained in other strings. That is, for example, if we have gagtat, ag, and gt, we can throw out ag and gt. Continuing with the example above, we are left with the following "snippets"

$$S = \Big\{\mathsf{tagg}, \mathsf{catt}, \mathsf{gga}, \mathsf{tta}, \mathsf{gagtat}\Big\}.$$

Following this observation, we can assume that the "snippets" have been preprocessed so that none of the strings are contained in other strings.

The second observation is that each string must start at a distinct position in the result, otherwise there would be a string that is a substring of another. Since they start at distinct positions there must be a total ordering of the strings in S in the solution and indeed any solution can be defined by this ordering.

We will now consider three algorithmic techniques that can be applied to this problem and derive an algorithm from each.

3.2 Algorithm 1: Brute Force

The first algorithm we consider uses the brute first technique. This is the simplest approach we can take, but as we will see it is not efficient.

Definition 3.2 (The Brute Force **Technique**)**.** Enumerate all possible candidate solutions for a problem, score each one (and/or checking that each satisfies the problem statement), and return a best (or feasible) solution.

In our case this involves enumerating all possible permutations of the input strings. For each permutation we can remove the maximum overlap between each adjacent pair of strings and determine the length. For example, the permutation

catt tta tagg gga gagtat

based on our running example will give us cattaggagtat after removing the overlaps (the excised parts are underlined). This has length 12. Note that this result happens to be the original string and is also the shortest superstring.

Exercise 1. *Try a couple other permutations and determine the length after removing overlaps.*

Does trying all permutations always give us the shortest string? As our intuition might suggest, the answer is yes and the proof of it, which we didn't go over in class, hints at an algorithm that we will look at in a moment.

Lemma 3.3. *Given a finite set of finite strings $S \subseteq \Sigma^+$, the brute force method finds the shortest superstring.*

Proof. Let r^* be any shortest superstring of S. We know that each string $s \in S$ appears in r^*. Let i_s denote the beginning position in r^* where s appears. Since we have eliminated duplicates, it must be the case that all i_s's are distinct numbers. Now let's look at all the strings in S, $s_1, s_2, \ldots, s_{|S|}$, where we number them such that $i_{s_1} < i_{s_2} < \cdots < i_{s_{|S|}}$. It is not hard to see that the ordering $s_1, s_2, \ldots, s_{|S|}$ gives us r^* after removing the overlaps. □

The approach of trying all permutations is easy to parallelize. Each permutation can be tested in parallel and it is also easy to generate all permutations in parallel. The problem with this approach, however, is that although highly parallel it has to examine a very large number of combinations, resulting in too much computational work. In particular, there are $n!$ permutations on a collection of n elements. This means that if the input consists of $n = 100$ strings, we'll need to consider $100! \approx 10^{158}$ combinations, which for a sense of scale, is more than the number of atoms in the universe. As such, the algorithm is not going to be feasible for large n.

Can we come up with a smarter algorithm that solves the problem faster? Unfortunately, it is unlikely. As it happens, this problem is NP-hard. But we should not fear NP-hard problems. In general, NP-hardness only suggests that there are families of instances on which the problem is hard in the worst-case. It doesn't rule out the possibility of algorithms that compute near optimal answers or algorithms that perform well on real world instances.

For this particular problem, we know efficient approximation algorithms that (1) give theoretical bounds that guarantee that the answer (i.e., the length) is within a constant factor of the optimal answer, and (2) in practice do even better than the bounds suggest.

Figure 1: A poster from a contest run by Proctor and Gamble in 1962. The goal was to solve a 33 city instance of the TSP. Gerald Thompson, a Carnegie Mellon professor, was one of the winners.

3.3 Algorithm 2: Reducing to Another Problem

Another approach to solving a problem is to reduce it to another problem which we understand better and for which we know algorithms. It is sometimes quite surprising that problems that seem very different can be reduced to each other. Note that reductions are sometimes used to prove that a problem is NP-hard (i.e. if you prove that an NP-complete problem A can be reduced to problem B with polynomial work, then B must also be NP-complete). That is **not** the purpose here. Instead we want the reduction to help us solve our problem.

In particular we consider reducing the shortest superstring problem to another seemingly unrelated problem: the traveling salesperson (TSP) problem. This is a canonical NP-hard problem dating back to the 1930s and has been extensively studied, e.g. see Figure 1. The two major variants of the problem are *symmetric* TSP and *asymmetric* TSP, depending on whether the graph has undirected or directed edges, respectively. The particular variant we're reducing to is the asymmetric version, which can be described as follows.

Definition 3.4 (The Asymmetric Traveling Salesperson (aTSP) Problem)**.** Given a weighted directed graph, find the shortest path that starts at a vertex s and visits all vertices exactly once before

returning to s.

That is, find a Hamiltonian cycle of the graph such that the sum of the edge weights along the cycle is the minimum of all such cycles (a cycle is a path in a graph that starts and ends at the same vertex, and a Hamiltonian cycle is a cycle that visits every vertex exactly once).

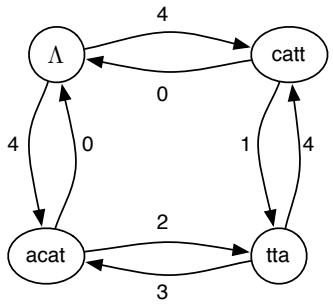
Motivated by the observation that the shortest superstring problem can be solved exactly by trying all permutations, we'll make the TSP problem try all the combinations for us. For this, we will set up a graph so that each valid Hamiltonian cycle corresponds to a permutation. Note that in our case, there is always a Hamiltonian cycle because the graph is a complete graph—there are two directed edges between every pair of vertices.

Let $\texttt{overlap}(s_i, s_j)$ denote the maximum overlap for s_i followed by s_j. This would mean $\texttt{overlap}$ ("tagg","gga") $= 2$.

The Reduction. Now we build a graph $D = (V, A)$.

- The vertex set V has one vertex per string and a special "source" vertex Λ where the cycle starts and ends.
- The arc (directed edge) from s_i to s_j has weight $w_{i,j} = |s_j| - \texttt{overlap}(s_i, s_j)$. This quantity represents the increase in the string's length if s_i is followed by s_j. As an example, if we have "tagg" followed by "gga", then we can generate "tagga" which only adds 1 character giving a weight of 1—indeed, $|\text{"gga"}| - \texttt{overlap}(\text{"tagg"}, \text{"gga"}) = 3 - 2 = 1$.
- The weights for arcs incident to Λ are set as follows: $(\Lambda, s_i) = |s_i|$ and $(s_i, \Lambda) = 0$. That is, if s_i is the first string in the permutation, then the arc (Λ, s_i) pays for the whole length s_i.

To see this reduction in action, the input {catt, acat, tta} results in the following graph (not all edges are shown).



As intended, in this graph, a cycle through the graph that visits each vertex once corresponds to a permutation in the brute force method. Furthermore, the sum of the edge weights in that cycle is equal to the length of the superstring produced by the permutation. Since TSP considers all Hamiltonian cycles, it also corresponds to considering all orderings in the brute force method. Since the TSP finds the min cost cycle, and assuming the brute force method is correct, then TSP finds

```
1  fun greedyApproxSS(S) =
2    if |S| = 1 then S_0
3    else let
4      val O = {(overlap(s_i, s_j), s_i, s_j) : s_i ∈ S, s_j ∈ S, s_i ≠ s_j}
5      val (o, s_i, s_j) = maxval <_#1 O
6      val s_k = join(s_i, s_j)
7      val S' = ({s_k} ∪ S)\{s_i, s_j}
8    in
9      greedyApproxSS(S')
10   end
```

Figure 2: A Greedy Algorithm for the Shortest Superstring (SS) Problem.

the shortest superstring. Therefore, if we could solve TSP, we would be able to solve the shortest superstring problem.

But TSP is also NP-hard. What we have accomplished so far is that we have reduced one NP hard problem to another, but the advantage is that there is a lot known about TSP, so maybe this helps.

3.4 Algorithm 3: Greedy

So far we have considered two techniques and corresponding algorithms for solving our problem: brute force and reduction. We now consider a third technique, the "greedy" technique, and corresponding algorithm.

Definition 3.5 (The Greedy **Technique**)**.** Given a sequence of steps, on each step make a locally optimal decision based on some criteria without ever backtracking on previous decisions.

The greedy technique (or approach) is a heuristic that when applied to many problems does not necessarily return an optimal solution. In our case the greedy algorithm indeed is guaranteed to find the shortest superstring but we can guarantee that it gives a good "approximation", and furthermore it works very well in bounds on how close it is to the optimal, and it works very well in practice. Greedy algorithms are popular because of their simplicity.

To describe the greedy algorithm, we'll define a function `join`(s_i, s_j) that appends s_j to s_i and removes the maximum overlap. For example, `join`("tagg","gga") = "tagga".

Figure 2 shows pseudocode for our greedy algorithm. In this course the pseudocode we use will be purely functional and easy to translate into ML code or code for just about any functional language. In fact it should not be hard to translate it to imperative languages, especially if they support higher-order functions. Primarily, the difference from ML is that we will use standard mathematical notation, such as subscripts, and set notation (e.g. $\{f(x) : x \in S\}$, $\cup$, $|S|$).

Given a set of strings S, the `greedyApproxSS` algorithm finds the pair of strings s_i and s_j in S that are distinct and have the maximum overlap—the `maxval` function takes a comparison operator (in this case comparing the first element of the triple) and returns a maximum element of a set (or

sequence) based on that comparison. The algorithm then replaces s_i and s_j with $s_k = \texttt{join}(s_i, s_j)$ in S. The new set S' is therefore one smaller. It recursively repeats this process on this new set of strings until there is only a single string left. The algorithm is greedy because at every step it takes the pair of strings that when joined will remove the greatest overlap, a locally optimal decision. Upon termination, the algorithm returns a single string that contains all strings in the original S. However, the superstring returned is not necessarily the shortest superstring.

Exercise 2. *In the code we remove s_i, s_j from the set of strings but do not remove any strings from S that are contained within $s_k = \texttt{join}(s_i, s_j)$. Argue why there cannot be any such strings.*

Exercise 3. *Prove that algorithm* `greedyApproxSS` *indeed returns a string that is a superstring of all original strings.*

Exercise 4. *Give an example input S for which* `greedyApproxSS` *does not return the shortest superstring.*

Exercise 5. *Consider the following greedy algorithm for TSP. Start at the source and always go to the nearest unvisited neighbor. When applied to the graph described above, is this the same as the algorithm above? If not what would be the corresponding algorithm for solving the TSP?*

Although the greedy algorithm merges pairs of strings one by one, we note there is still significant parallelism in the algorithm, at least as described. In particular in line **??** we can calculate all the overlaps in parallel, and in line **??** we can can calculate the largest overlap in parallel using a reduction. We will look at the cost analysis in more detail in the next lecture.

Although `greedyApproxSS` does not return the shortest superstring, it returns an "approximation" of the shortest superstring. In particular, it is known that it returns a string that is within a factor of 3.5 of the shortest and conjectured that it returns a string that is within a factor of 2. In practice, it typically *does much better* than the bounds suggest. The algorithm also generalizes to other similar problems.

Of course, given that the SS problem is NP-hard, and `greedyApproxSS` does only polynomial work (see below), we cannot expect it to give an exact answer on all inputs—that would imply **P** = **NP**, which is unlikely. In literature, algorithms such as `greedyApproxSS` that solve an NP-hard problem to within a constant factor of optimal, are called *constant-factor approximation algorithms*.

Truth in advertising. Often when abstracting a problem we can abstract away some key aspects of the underlying application that we want to solve. Indeed this is the case when using the Shortest Superstring problem for sequencing genomes. In actual genome sequencing there are two shortcomings with using the SS problem. The first is that when reading the base pairs using a DNA sequencer there can be errors. This means the overlaps on the strings that are supposed to overlap perfectly might not. Don't fret: this can be dealt with by generalizing the Shortest Superstring problem to deal with approximate matching. Describing such a generalization is beyond the scope of this course, but basically one can give a score to every overlap and then pick the best one for each pair of fragments. The nice thing is that the same algorithmic techniques we discussed for the SS problem still work for this generalization, only the "overlap" scores will be different.

The second shortcoming of using the SS problem by itself is that real genomes have long repeated sections, possibly much longer than the length of the fragments that are sequenced. The SS problem does not deal well with such repeats. In fact when the SS problem is applied to the fragments of an initial string with longer repeats that the fragment sizes, the repeats or parts of them are removed. One method that researchers have used to deal with this problem is the so-called *double-barrel shotgun method*. In this method strands of DNA are cut randomly into lengths that are long enough to span the repeated sections. After cutting it up one can read just the two ends of such a strand and also determine its length (approximately). By using the two ends and knowing how far apart they are it is possible to build a "scaffolding" and recognize repeats. This method can be used in conjunction with the generalization of the SS discussed in the previous paragraph. In particular the SS method allowing for errors can be used to generate strings up to the length of the repeats, and the double barreled method can put them together.

4 What did we review and learn in this lecture?

- It is important to distinguish interfaces (problems and ADTs) from implementations (algorithms and data structures).
- Defining interfaces precisely is key.
- The shortest superstring (SS) problem and its application to genome sequencing.
- The brute force method explores all possibilities and picks the best. This leads to a simple but inefficient solution to the SS problem.
- Often it is possible to reduce one problem to another that appears very different. We reduced the SS problem to the traveling salesperson problem.
- The greedy method greedily makes the best "local" decision in a sequence of steps. It sometimes does not lead to an optimal solution, but when applied to the SS problem can lead to an approximate solution—one that is guaranteed to be within a constant factor of optimal.

Lecture 2 — Algorithmic Cost Models

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — 30 August 2012

Today:

- Continue on the Shortest Superstring Problem
- Overview of algorithmic cost models for analyzing algorithms

1 Cost Models

When we analyze the cost of an algorithm formally, we need to be reasonably precise in what model we are performing the analysis. Typically when analyzing algorithms the purpose of the model is not to calculate exact running times (this is too much to ask), but rather just to analyze asymptotic costs (*i.e.*, big-O). These costs can then be used to compare algorithms in terms of how they scale to large inputs. For example, as you know, some sorting algorithms use $O(n \log n)$ work and others $O(n^2)$. Clearly the $O(n \log n)$ algorithm scales better, but perhaps the $O(n^2)$ is actually faster on small inputs. In this class we are concerned with how algorithms scale, and therefore asymptotic analysis is indeed what we want. Because we are using asymptotic analysis the exact constants in the model do not matter, but what matters is that the asymptotic costs are well defined. Since you have seen big-O, big-Theta, and big-Omega in 15-122, 15-150 and 15-251 (if you have taken it) we will not be covering it here but would be happy to review it in office hours.

There are two important ways to define cost models, one based on machines and the other based more directly on programming constructs. Both types can be applied to analyzing either sequential and parallel computations. Traditionally machine models have been used, but in this course, as in 15-150, we will use a model that abstract to the programming constructs. We first review the traditional machine model.

1.1 The RAM model for sequential computation:

Traditionally, algorithms have been analyzed in the Random Access Machine (RAM)[1] model. This model assumes a single processor accessing unbounded memory indexed by the non-negative integers. The processor interprets sequences of machine instructions (code) that are stored in the memory. Instructions include basic arithmetic and logical operations (e.g. +, -, *, and, or, not), reads from and writes to arbitrary memory locations, and conditional and unconditional jumps to other locations in the code. The cost of a computation is measured in terms of the number of instructions execute by the machine, and is referred to as *time*.

This model has served well for analyzing the asymptotic runtime of sequential algorithms and most work on sequential algorithms to date has used this model. It is therefore important to

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

[1]Not to be confused with Random Access Memory (RAM)

understand what this model is. One reason for its success is that there is an easy mapping from algorithmic pseudocode and sequential languages such as C and C++ to the model and so it is reasonably easy to reason about the cost of algorithms and code. As mentioned earlier, the model should only be used for deriving asymptotic bounds (*i.e.*, using big-O, big-Theta and big-Omega) and not for trying to predict exact runtimes. One reason for this is that on a real machine not all instructions take the same time, and furthermore not all machines have the same instructions.

The problem with the RAM for our purposes is that the model is sequential. There is an extension of the RAM model for parallelism, which is called the parallel random access machine (PRAM). It consists of p processors sharing a memory. All processors execute the same instruction on each step. We will not be using this model since we find it awkward to work with and everything has to be divided onto the p processors. However, most of the ideas presented in this course also work with the PRAM, and many of them were originally developed in the context of the PRAM.

We note that one problem with the RAM model is that it assumes that accessing all memory locations has the same cost. On real machines this is not at all the case. In fact, there can be a factor of over 100 between the time for accessing a word of memory from the first level cache and accessing it from main memory. Various extensions to the RAM model have been developed to account for this cost. For example one variant is to assume that the cost for accessing the the i^{th} memory location is $f(i)$ for some function f, e.g. $f(i) = \log(i)$. Fortunately, however, most of the algorithms that turn out to be the best in these more detailed models are also good in the RAM. Therefore analyzing algorithms in the simpler RAM model is a reasonable approximation to analyzing in the more refined models. Hence the RAM has served quite well despite not fully accounting for memory costs.

The model we use in this course also does not directly account for the variance in memory costs. Towards the end of the course, if time permits, we will discuss how it can be extended to capture memory costs.

1.2 The Parallel Model Used in this Course

Instead of using a machine model, in this course, as with 15-150, we will define a model more directly tied to programming constructs without worrying how it is mapped onto a machine. The goal of the course is to get you to "think parallel" and we believe the the model we use makes it much easier to separate the high-level concepts of parallelism from low-level machine-specific details. As it turns out there is a way to map the costs we derive onto costs for specific machines.

To formally define a cost model in terms of programming constructs requires a so-called "operational semantics". We won't define a complete semantics, but will give a partial semantics to give a sense of how it works. We will measure complexity in terms of two costs: work and span. Roughly speaking the *work* corresponds to the total number of operations we perform, and *span* to the longest chain of dependences. Although you have seen work and span in 15-150, we will review the definition here in and go into some more detail.

We define work and span in terms of simple compositional rules over expressions in the language. For an expression e we will use $W(e)$ to indicate the work needed to evaluate that expression and $S(e)$ to indicate the span. We can then specify rules for composing the costs across sub expressions. Expressions are either composed sequentially (one after the other) or in parallel (they can run at the same time). When composed sequentially we add the work and we add the span, and when

composed in parallel we add the work but take the maximum of the span. Basically that is it! We do, however, have to specify when things are composed sequentially and when in parallel.

In a functional language, as long as the output for one expression is not required for the input of another, it is safe to run the two expressions in parallel. So for example, in the expression $e_1 + e_2$ where e_1 and e_2 are themselves other expressions (e.g. function calls) we could run the two expressions in parallel giving the rule $S(e_1 + e_2) = 1 + \max(S(e_1), S(e_2))$. This rule says the two subexpressions run in parallel so that we take the max of the span of each subexpression. But the addition operation has to wait for both subexpressions to be done. It therefore has to be done sequentially after the two parallel subexpressions and hence the reason why there is a plus 1 in the expression $1 + \max(S(e_1), S(e_2))$.

In an imperative language we have to be much more careful. Indeed it can be very hard to figure out when computations depend on each other and, therefore, when it is safe to put things together in parallel. In particular subexpressions can interact through shared state. e.g. For example in C, in the expression:

```
foo(2) + foo(3)
```

it is not safe to make the two calls to `foo(x)` in parallel since they might interact. Suppose

```
int y = 0;
int foo(int x)  return y = y + x;
```

With `y` starting at 0, the expression `foo(2) + foo(3)` could lead to several different outcomes depending on how the instructions are interleaved (scheduled) when run in parallel. This interaction is often called a race condition and will be covered further in more advanced courses.

In this course, as we will use only purely functional constructs, it is always safe to run expressions in parallel. To make it clear whether expressions are evaluated sequentially or in parallel, in the pseudocode we write we will use the notation (e_1, e_2) to mean that the two expressions run sequentially (even when they could run in parallel), and $e_1 \parallel e_2$ to mean that the two expressions run in parallel. Both constructs return the pair of results of the two expressions. For example `fib`$(6) \parallel$ `fib`(7) would return the pair $(8, 13)$. In addition to the $\parallel$ construct we assume the set-like notation we use in pseudocode $\{f(x) : x \in A\}$ also runs in parallel, *i.e.*, all calls to $f(x)$ run in parallel. These rules for composing work and span are outlined in Figure 1. Note that the rules are the same for work and span except for the two parallel constructs we just mentioned.

As there is no $\parallel$ construct in the ML, in your assignments you will need to specify in comments when two calls run in parallel. We will also supply an ML function `parallel(f1,f2)` with type `(unit -> `α`)` $\times$ `(unit -> `β`) -> `$\alpha \times \beta$. This function executes the two functions that are passed in as arguments in parallel and returns their results as a pair. For example:

```
parallel (fn => fib(6), fn => fib(7))
```

returns the pair $(8, 13)$. We need to wrap the expressions in functions in ML so that we can make the actual implementation run them in parallel. If they were not wrapped both arguments would be evaluated sequentially before they are passed to the function `parallel`. Also in the ML code you do

$$
\begin{aligned}
W(c) &= 1 \\
W(\text{op}\ e) &= 1 \\
W((e_1, e_2)) &= 1 + W(e_1) + W(e_2) \\
W(e_1 \mid\mid e_2) &= 1 + W(e_1) + W(e_2) \\
W(\texttt{let val } x = e_1 \texttt{ in } e_2 \texttt{ end}) &= 1 + W(e_1) + W(e_2[\text{Eval}(e_1)/x])) \\
W(\{f(x) : x \in A\}) &= 1 + \sum_{x \in A} W(f(x))
\end{aligned}
$$

$$
\begin{aligned}
S(c) &= 1 \\
S(\text{op}\ e) &= 1 \\
S((e_1, e_2)) &= 1 + S(e_1) + S(e_2) \\
S((e_1 \mid\mid e_2)) &= 1 + \max(S(e_1), S(e_2)) \\
S(\texttt{let val } x = e_1 \texttt{ in } e_2 \texttt{ end}) &= 1 + S(e_1) + S(e_2[\text{Eval}(e_1)/x])) \\
S(\{f(x) : x \in A\}) &= 1 + \max_{x \in A} S(f(x))
\end{aligned}
$$

Figure 1: Composing work and span costs. In the first rule c is any constant value (e.g. 3). In the second rule op is a primitive operator such as op.+, op.$*$, op. , ... The next rule, the pair $(e_1, e_)2)$ evaluates the two expressions sequentially, whereas the rule $(e_1 \mid\mid e_2)$ evaluates the two expressions in parallel. Both return the results as a pair. In the rule for `let` the notation Eval(e) evaluates the expression e and returns the result, and the notation $e[v/x]$ indicates that all free (unbound) occurrences of the variable x in the expression e are replaced with the value v. These rules are representative of all rules of the language. Notice that all the rules for span are the same as for work except for parallel application indicated by $(e_1 \mid\mid e_2)$ and the parallel map indicated by $\{f(x) : x \in A\}$. The expression e inside $W(e)$ and $S(e)$ have to be closed. Note, however, that in the rules such as for `let` we replace all the free occurrences of x in the expression e_2 with their values before applying W.

not have the set notation $\{f(x) : x \in A\}$, but as mentioned before, it is basically equivalent to a `map`. Therefore, for ML code you can use the rules:

$$W(\texttt{map}\ \ f\ \ \langle s_0, \ldots, s_{n-1} \rangle) = 1 + \sum_{i=0}^{n-1} W(f(s_i))$$

$$S(\texttt{map}\ \ f\ \ \langle s_0, \ldots, s_{n-1} \rangle) = 1 + \max_{i=0}^{n-1} S(f(s_i))$$

Parallelism: An additional notion of cost that is important in comparing algorithms is the *parallelism* of an algorithm. The parallelism is simply defined as the work over the span:

$$\mathbb{P} = \frac{W}{S}$$

For example for a mergesort with work $\theta(n \log n)$ and span $\theta(\log^2 n)$ the parallelism would be $\theta(n/\log n)$. Parallelism represents roughly how many processors we can use efficiently. As you saw in the processor scheduling example in 15-150, $\mathbb{P}$ is the most parallelism you can get. That is, it measures the limit on the performance that can be gained when executed in parallel.

For example, suppose $n = 10,000$ and if $W(n) = \theta(n^3) \approx 10^{12}$ and $S(n) = \theta(n \log n) \approx 10^5$ then $\mathbb{P}(n) \approx 10^7$, which is a lot of parallelism. But, if $W(n) = \theta(n^2) \approx 10^8$ then $\mathbb{P}(n) \approx 10^3$, which is much less parallelism. The decrease in parallelism is not because of the span was large, but because the work was reduced.

Goals: In parallel algorithm design, we would like to keep the parallelism as high as possible but without increasing work. In general the goals in designing efficient algorithms are

1. First priority: to keep work as low as possible
2. Second priority: keep parallelism as high as possible (and hence the span as low as possible).

In this course we will mostly cover algorithms where the work is the same or close to the same as the best sequential time. Indeed this will be our goal throughout the course. Now among the algorithm that have the same work as the best sequential time we will try to achieve the greatest parallelism.

Under the hood: In the parallel model we will be using a program can generate tasks on the fly and can generate a huge amount of parallelism, typically much more than the number of processors available when running. It therefore might not be clear how this maps onto a fixed number of processors. That is the job of a scheduler. The scheduler will take all of these tasks, which are generated dynamically as the program runs, and assign them to processors. If only one processor is available, for example, then all tasks will run on that one processor.

We say that a scheduler is *greedy* if whenever there is a processor available and a task ready to execute, then the task will be scheduled on the processor and start running immediately. Greedy schedulers have a very nice property that is summarized by the following:

Definition 1.1. The *greedy scheduling principle* says that if a computation is run on p processors using a greedy scheduler, then the total time (clock cycles) for running the computation is bounded by

$$T_p < \frac{W}{p} + S \tag{1}$$

where W is the work of the computation, and S is the span of the computation (both measured in units of clock cycles).

This is actually a very powerful statement. The time to execute the computation cannot be any better than $\frac{W}{p}$ clock cycles since we have a total of W clock cycles of work to do and the best we can possibly do is divide it evenly among the processors. Also note that the time to execute the computation cannot be any better than S clock cycles since S represents the longest chain of sequential dependences. Therefore the very best we could do is:

$$T_p \geq \max\left(\frac{W}{p}, S\right)$$

We therefore see that a greedy scheduler does reasonably close to the best possible. In particular $\frac{W}{p} + S$ is never more than twice $\max(\frac{W}{p}, S)$ and when $\frac{W}{p} \gg S$ the difference between the two is very small. Indeed we can rewrite equation 1 above in terms of the parallelism $\mathbb{P} = W/S$ as follows:

$$\begin{aligned} T_p &< \frac{W}{p} + S \\ &= \frac{W}{p} + \frac{W}{\mathbb{P}} \\ &= \frac{W}{p}\left(1 + \frac{p}{\mathbb{P}}\right) \end{aligned}$$

Therefore as long as $P \gg p$ (the parallelism is much greater than the number of processors) then we get near perfect speedup. (Speedup is W/T_p and perfect speedup would be p).

Truth in advertising. No real schedulers are fully greedy. This is because there is overhead in scheduling the job. Therefore there will surely be some delay from when a job becomes ready until when it starts up. In practice, therefore, the efficiency of a scheduler is quite important to achieving good efficiency. Also the bounds we give do not account for memory affects. By moving a job we might have to move data along with it. Because of these affects the greedy scheduling principle should only be viewed as a rough estimate in much the same way that the RAM model or any other computational model should be just viewed as an estimate of real time.

2 Super Costs: Shortest Superstring Revisited

As examples of how to use our cost model we will analyze a couple of the algorithms we described for the shortest superstring problem: the brute force algorithm and the greedy algorithm.

2.1 The Brute Force Shortest Superstring Algorithm

Recall that the idea of the brute force algorithm for the SS problem is to try all permutations of the input strings and for each permutation to determine the maximal overlap between adjacent strings and remove them. We then pick whichever remaining string is shortest, of if there is a tie we pick any of the shortest. We can calculate the overlap between all pairs of strings in a preprocessing phase. Let n be the size of the input S and m be the total number of characters across all strings in S, i.e.,

$$m = \sum_{s \in S} |s|.$$

Note that $n \leq m$. The preprocessing step can be done in $O(m^2)$ work and $O(\log n)$ span (see analysis below). This is a low order term compared to the other work, as we will se, so we can ignore it.

Now to calculate the length of a given permutation of the strings with overlaps removed we can look at adjacent pairs and look up their overlap in the precomputed table. Since there are n strings and each lookup takes constant work, this requires $O(n)$ work. Since all lookups can be done in parallel, it will require only $O(1)$ span. Finally we have to sum up the overlaps and subtract it from m. The summing can be done with a **reduce** in $O(n)$ work and $O(\log n)$ span. Therefore the total cost is $O(n)$ work and $O(\log n)$ span.

As we discussed in the last lecture the total number of permutations is $n!$, each of which we have to check for the length. Therefore the total work is $O(nn!) = O((n+1)!)$. What about the span? Well we can run all the tests in parallel, but we first have to generate the permutations. One simple way is to start by picking in parallel each string as the first string, and then for each of these picking in parallel another string as the second, and so forth. The pseudocode looks something like this:

```
func permutations S =
   if |S| = 1 then {S}
   else
      {append([s], p) :  s in S, p in permutations(S/s)}
```

What is the span of this code?

2.2 The Greedy Shortest Superstring Algorithm

We'll consider a straightforward implementation, although the analysis is a little tricky since the strings can vary in length. First we note that calculating `overlap`(s_1, s_2) and `join`(s_1, s_2) can be done in $O(|s_1||s_2|)$ work and $O(\log(|s_1| + |s_2|)$ span[2]. This is simply by trying all overlap positions

[2]We'll use the terms *span* and *depth* interchangeably in this class.

between the two strings, seeing which ones match, and picking the largest. The logarithmic span is needed for picking the largest matching overlap using a reduce.

Let W_{ov} and S_{ov} be the work and span for calculating all pairs of overlaps (the line {(`overlap` $(s_i, s_j), s_i, s_j) : s_i \in S, s_j \in S, s_i \neq s_j$}).

We have

$$\begin{aligned} W_{ov} &\leq \sum_{i=1}^{n}\sum_{j=1}^{n} W(\texttt{overlap}(s_i, s_j))) \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} O(|s_i||s_j|) \\ &\leq \sum_{i=1}^{n}\sum_{j=1}^{n} (k_1 + k_2|s_i||s_j|) \\ &= k_1 n^2 + k_2 \left(\sum_{i=1}^{n} |s_i|\right)^2 \\ &\in O(m^2) \end{aligned}$$

and since all pairs can be done in parallel,

$$\begin{aligned} S_{ov} &\leq \max_{i=1}^{n} \max_{j=1}^{n} S(\texttt{overlap}(s_i, s_j))) \\ &\in O(\log m) \end{aligned}$$

The `maxval` can be computed in $O(m^2)$ work and $O(\log m)$ span using a simple reduce. The other steps cost no more than computing `maxval`. Therefore, not including the recursive call each call to `greedyApproxSS` costs $O(m^2)$ work and $O(\log m)$ span.

Finally, we observe that each call to `greedyApproxSS` creates S' with one fewer element than S, so there are at most n calls to `greedyApproxSS`. These calls are inherently sequential because one call must complete before the next call can take place. Hence, the total cost for the algorithm is $O(nm^2)$ work and $O(n \log m)$ span, which is highly parallel.

Exercise 1. *Come up with a more efficient way of implementing the greedy method.*

Lecture 3 — Algorithmic Techniques and Divide-and-Conquer

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — September 4, 2012

Material in this lecture:

- Quick review of Cost Models (from last lecture)
- Algorithmic techniques
- Divide and Conquer

1 Algorithmic Techniques

There are many algorithmic techniques/approaches for solving problems. Understanding when and how to use these techniques is one of the most important skills of a good algorithms designer. In the context of the shortest superstring (SS) problem we already mentioned three techniques: brute force, reducing one problem to another, and the greedy approach. In this course we will cover many techniques and we now give a brief overview of the techniques. We will then go into one of the techniques, divide-and-conquer, in more detail. All these techniques are useful for both sequential and parallel algorithms, however some, such as divide-and-conquer, play an even larger role in parallel algorithms than sequential algorithms.

Brute Force: The brute force approach typically involves trying all possibilities. In the SS problem, for example, we argued that every solution has to correspond to a permutation of the inputs with overlaps removed. The brute force approach therefore tried all permutations and picked the best. Since there are $n!$ permutations, this solution is not "tractable" for large problems. In many other problems there are only polynomially many possibilities. For example in your current assignment there should be only $O(n^2)$ possibilities to try. However, even $O(n^2)$ is not good, since as you will work out in the assignment there is another solution that require only $O(n)$ work. One place the brute force approach can be very useful is when writing a test routine to check the correctness of more efficient algorithms. Even if inefficient for large n the brute force approach could work well for testing small inputs. The brute force approach is often the simplest solution to a problem, although not always.

Reducing to another problem: Sometimes the easiest approach to solving a problem is just reduce it to another problem for which known algorithms exist. For the SS problem we reduced it to what would seem superficially to be a very different problem, the Traveling Salesperson (TSP) problem. The reduction was pretty straightforward, but how would anyone even think of doing such a translation. When you get more experienced with algorithms you will start recognizing similarities between ~~problems that on the surface seem very~~ different.

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

Inductive techniques: The idea behind inductive techniques is to solve one or more smaller problems that can be used to solve the original problem. The technique most often uses recursion to solve the sub problems and can be proved correct using (strong) induction. Common techniques that fall in this category include:

- *Divide-and-conquer.* Divide the problem on size n into $k > 1$ subproblems on sizes $n_1, n_2, \ldots n_k$, solve the problem recursively on each, and combine the solutions to get the solution to the original problem.
- *Greedy.* For a problem on size n use some greedy approach to pull out one element leaving a problem of size $n - 1$. Solve the smaller problem.
- *Contraction.* For a problem of size n generate a significantly smaller (contracted) instance (e.g. of size $n/2$), solve the smaller instance, and then use the result to solve the original problem. This only differs from divide and conquer in that we make one recursive call instead of multiple.
- *Dynamic Programming.* Like divide and conquer, dynamic programming divides the problem into smaller problems and then combines the solutions to the subproblems to find a solution to the original problem. The difference, though, is that the solutions to subproblems are reused multiple times. It is therefore important to store the solutions for reuse either using memoization or by building up a table of solutions.

Data Types and Data Structures: Some techniques make heavy use of the operations on abstract data types representing collections of values and their implementation using a variety of data structures. Abstract collection types that we will cover in this course include: Sequences, Sets, Priority Queues, Graphs, and Sparse Matrices.

Randomization: Randomization in is a powerful technique for getting simple solutions to problems. We will cover a couple of examples in this course. Formal cost analysis for many randomized algorithms, however, requires probability theory beyond the level of this course.

Once you have defined the problem, you can look into your bag of techniques and, with practice, you will find a good solutions to the problem. When we say a *good solution* we mean:

1. **Correct:** Clearly correctness is most important.
2. **Low cost:** Out of the correct solutions, we would prefer the one with the lowest cost.

2 Divide and Conquer

Divide and conquer is a highly versatile technique that generally lends itself very well to parallel algorithms design. You probably have seen the divide-and-conquer technique many times before this course, but it is such an important technique that it is worth seeing it over and over again. It is particularly suited for "thinking parallel" because it offers a natural way of creating parallel tasks.

In most divide-and-conquer algorithms you have encountered so far, the subproblems are occurrences of the problem you are solving (e.g. recursive sorting in merge sort). This is not always

the case. Often, you'll need more information from the subproblems to properly combine results of the subproblems. In this case, you'll need to **strengthen** the problem definition, much in the same way that you strengthen an inductive hypothesis when doing an inductive proof. This involves defining a problem that solves more than what you ultimately need, but makes it easier or even possible to use solutions of subproblems to solve the larger problem. In the recitation you looked at how to solve the Parenthesis Matching problem by defining a version of the problem that returns the number of unmatched parentheses on the right and left. This is a stronger problem than the original—the original is the special case when both these values are zero (no unmatched right or left parentheses). Therefore the modified version tells us more than we need but was necessary to make the divide-and-conquer work.

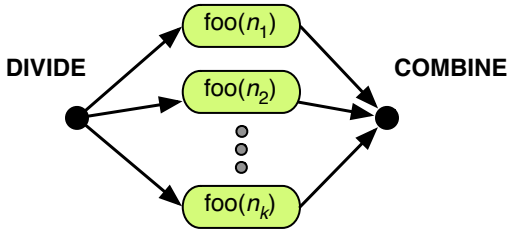
The structure of a divide-and-conquer algorithm follows the structure of a proof by (strong) induction. This makes it easy to show correctness. Often it also makes it easy to determine costs bounds since we can write recurrences that match the inductive structure. The general form of divide-and-conquer looks as follows:

— **Base Case:** When the problem is sufficiently small, we return the trivial answer directly or resort to a different, usually simpler, algorithm, which works great on small instances.

— **Inductive Step:**

1. First, the algorithm *divides* the current instance I into parts, commonly referred to as *subproblems*, each smaller than the original problem.
2. It then recurses on each of the parts to obtain answers for the parts. In proofs, this is where we assume inductively that the answers for these parts are correct.
3. It then *combines* the answers to produce an answer for the original instance I, and in the reasoning about correctness and proof we have to show that this combination gives the correct answer.

This process can be schematically depicted as



On the assumption that the subproblems can be solved independently, the work and span of such an algorithm can be described using simple recurrences. In particular for a problem of size n is broken into k subproblems of size $n_1, \ldots, n_k$, then the work is

$$W(n) = W_{\text{divide}}(n) + \sum_{i=1}^{k} W(n_i) + W_{\text{combine}}(n)$$

and the span is

$$S(n) = S_{\text{divide}}(n) + \max_{i=1}^{k} S(n_i) + S_{\text{combine}}(n)$$

Note that the work recurrence is simply adding up the work across all components. More interesting is the span recurrence. First, note that a divide and conquer algorithm has to split a problem instance into subproblems *before* these subproblems are recursively solved. We therefore have to add the span for the divide step. The algorithm can then execute all the subproblems in parallel. We therefore take the maximum of the span for these subproblems. Finally *after* all the problems complete we can combine the results. We therefore have to add in the span for the combine step.

Applying this formula often results in familiar recurrences such as $W(n) = 2W(n/2) + O(n)$. In the rest of this lecture, we'll get to see other recurrences. For how, let's derive a closed-form for this expression.

The first recurrence we're looking at is $W(n) = 2W(n/2) + O(n)$, which you probably have seen many times already. To derive a closed form for it, we'll review the tree method, which you have seen in 15-122 and 15-251.

But first, let's think through what it means when we write $O(f(n))$ in an expression (e.g., when we write $4W(n/2) + O(n)$ in the recurrence above). In these expression, we write $O(f(n))$ in place of *some* function $g(n) \in O(f(n))$. From the definition of $O(\cdot)$, this means that there exist positive constants N_0 and c such that for all $n \geq N_0$, we have $g(n) \leq c \cdot f(n)$. It follows that *there exist constants k_1 and k_2 such that for all $n \geq 1$,*

$$g(n) \leq k_1 \cdot f(n) + k_2,$$

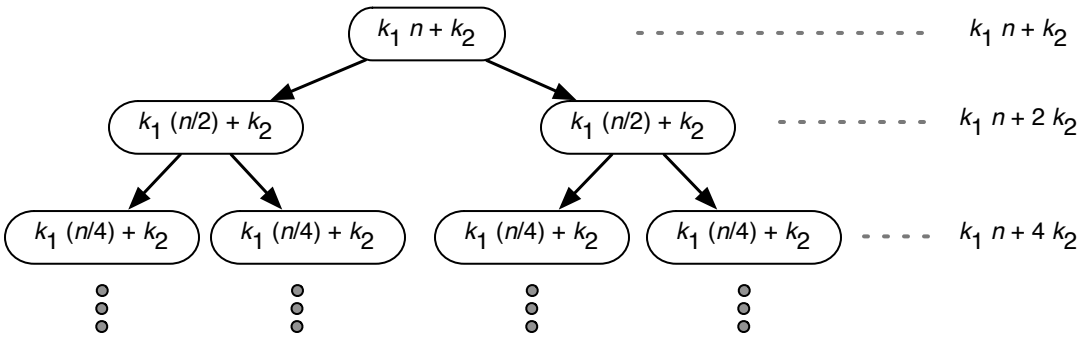
where, for example, we can take $k_1 = c$ and $k_2 = \sum_{i=1}^{N_0} |g(i)|$.

By this argument, we can establish that

$$W(n) \quad \leq \quad 2W(n/2) + k_1 \cdot n + k_2,$$

where k_1 and k_2 are constants. We'll now use the tree method.

The idea of the tree method is to consider the recursion tree of the algorithm in hope to derive an expression for what's happening at each level of the tree. In this particular example, we can see that each node in the tree has 2 children, whose input is half the size of that of the parent node. Moreover, if the input has size n, the recurrence shows that the cost, excluding that of the recursive calls, is at most $k_1 \cdot n + k_2$. Therefore, our recursion tree annotated with cost looks like this:



To apply the tree method, there are some key questions we should ask ourselves to aid drawing out the recursion tree and to understand the cost associated with the nodes:

- How many levels are there in the tree?
- What is the problem size at level i?
- What is the cost of each node in level i?
- How many nodes are there at level i?
- What is the total cost across level i?

Our answers to these questions lead to the following analysis: We know that level i (the root is level $i = 0$) contains 2^i nodes, each costing at most $k_1(n/2^i) + k_2$. Thus, the total cost in level i is at most

$$2^i \cdot \left(k_1 \frac{n}{2^i} + k_2\right) \quad = \quad k_1 \cdot n + 2^i \cdot k_2.$$

Since we keep halving the input size, the number of levels is bounded by $1 + \log n$. Hence, we have

$$\begin{aligned} W(n) &\leq \sum_{i=0}^{\log n} \left(k_1 \cdot n + 2^i \cdot k_2\right) \\ &= k_1 n(1 + \log n) + k_2(n + \tfrac{n}{2} + \tfrac{n}{4} + \cdots + 1) \\ &\leq k_1 n(1 + \log n) + 2k_2 n \\ &\in O(n \log n), \end{aligned}$$

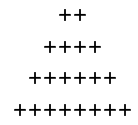
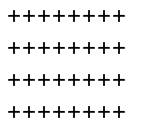
where in the second to last step, we apply the fact that for $a > 1$,

$$1 + a + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} \leq a^{n+1}.$$

2.1 A Useful Trick — The Brick Method

The tree method involves determining the depth of the tree, computing the cost at each level, and summing the cost across the levels. Usually we can easily figure out the depth of the tree and the cost of at each level relatively easily—but then, the hard part is taming the sum to get to the final answer. The following trick can help you derive the final answer quickly. By recognizing which shape a given recurrence is, we can almost immediately determine the asymptotic complexity of that recurrence.

The vital piece of information is *the ratio of the cost between adjacent levels*. Let cost_i denotes the total cost at level i when we draw the recursion tree. This puts recurrences into three broad categories:

Leaves Dominated	***Balanced***	***Root Dominated***
Each level is larger than the level before it by at least a constant factor. That is, there is a constant $\rho > 1$ such that for all level i, $\mathsf{cost}_{i+1} \geq \rho \cdot \mathsf{cost}_i$	All levels have approximately the same cost.	Each level is smaller than the level before it by at least a constant factor. That is, there is a constant $\rho < 1$ such that for all level i, $\mathsf{cost}_{i+1} \leq \rho \cdot \mathsf{cost}_i$
		
Implication: $O(\mathsf{cost}_d)$, where d is the depth The house is stable, with a strong foundation.	*Implication:* $O(d \cdot \max_i \mathsf{cost}_i)$ The house is sort of stable, but don't build too high.	*Implication:* $O(\mathsf{cost}_0)$ The house will tip over.

You might have seen the "master method" for solving recurrences in previous classes. We do not like to use it since it only works for special cases and does not give an intuition of what is going on. However, we will note that the three cases of the master method correspond to special cases of leaves dominated, balanced, and root dominated.

3 An Example : The Maximum Contiguous Subsequence Sum Problem

We now consider an example problem for which we can find a couple divide-and-conquer solutions.

Definition 3.1 (The Maximum Contiguous Subsequence Sum (MCSS) Problem)**.** Given a sequence of numbers $s = \langle s_1, \ldots, s_n \rangle$, the *maximum contiguous subsequence sum* problem is to find

$$\mathsf{mcss}(s) = \max \left\{ \sum_{k=i}^{j} s_k \ : \ 1 \leq i \leq n, i \leq j \leq n \right\}.$$

(i.e., the sum of the contiguous subsequence of s that has the largest value).

Let's look at an example. For $s = \langle 1, -5, 2, -1, 3 \rangle$, we know that $\langle 1 \rangle$, $\langle 2, -1, 3 \rangle$, and $\langle -5, 2 \rangle$ are all *contiguous* subsequences of s—whereas $\langle 1, 2, 3 \rangle$ is not. Among such subsequences, we're interested in finding one that maximizes the sum. In this particular example, we can check that $\mathsf{mcss}(s) = 4$, achieved by taking the subsequence $\langle 2, -1, 3 \rangle$.

We have to be careful about what our MCSS problem returns given an empty sequence as input. Here we assume the max of an empty set is $-\infty$ so it returns $-\infty$. We could alternatively decide it is undefined.

3.1 Algorithm 1: Brute Force

Immediate from the definition is an algorithm with $O(n^3)$ work and $O(\log n)$ span. This algorithm examines all possible contiguous subsequences and for each one of them, it computes the sum. It then takes the maximum of these sums. Note that every subsequence of s can be represented by a starting position i and an ending position j.

The pseudocode for this algorithm using our notation exactly matches the definition of the problem. For each subsequence $i..j$, we can compute its sum by applying a plus `reduce`. This does $O(j-i)$ work and has $O(\log(j-i))$ span. Furthermore, all the subsequences can be examined independently in parallel (using, e.g., `tabulate`). This leads to the following bounds:

$$\begin{aligned} W(n) &= 1 + \sum_{1 \le i \le j \le n} W_{\text{reduce}}(j-i) \le 1 + n^2 \cdot W_{\text{reduce}}(n) = 1 + n^2 \cdot O(n) = O(n^3) \\ S(n) &= 1 + \max_{1 \le i \le j \le n} S_{\text{reduce}}(j-i) \le 1 + S_{\text{reduce}}(n) = O(\log n) \end{aligned}$$

These are cost bounds for enumerating over all possible subsequences and computing their sums. The final step of the brute-force solution is to find the max over these combinations. Since max reduce has $O(n^2)$ work and $O(\log n)$ span[1], the cost of the final step is subsumed by other costs analyzed above. Overall, we have an $O(n^3)$-work $O(\log n)$-span algorithm.

As you might notice already, this algorithm is clearly inefficient. We'll apply divide and conquer to come up with a more efficient solution.

3.2 Algorithm 2: Divide And Conquer — Version 1.0

We'll design a divide-and-conquer algorithm for this problem. In coming up with such an algorithm, an important first step lies in figuring out how to properly divide up the input instance.

What is the simplest way to divide a sequence? Let us divide the sequence in half, recursively solve the problem on both halves and see what we can do. For example, imagine we divide the sequence in the middle and we get the following answers:

$$\langle \text{—— } L \text{ ——} \parallel \text{—— } R \text{ ——} \rangle$$

$$\Downarrow$$

$$L = \underbrace{\langle \cdots \rangle}_{\text{mcss}=56} \qquad R = \underbrace{\langle \ldots \rangle}_{\text{mcss}=17}$$

How would we use this to create a divide-and-conquer algorithm? So far, we have suggested a simple way to divide the input sequence. Assuming that we can recursively solve the problem, by calling the algorithm itself on smaller instances, we need answer the next question, how to combine the answers to the subproblems to obtain the final answer?

Our first (blind) guess might be to return $\max(\text{mcss}(L), \text{mcss}(R))$. This is unfortunately incorrect. We need a better understanding of the problem to devise a correct combine step. Notice that the

[1] Note that it takes the maximum over $\binom{n}{2} \le n^2$ values, but since $\log n^a = a \log n$, this is simply $O(\log n)$

subsequence we're looking for has one of the three forms: (1) the maximum sum lies completely in the left subproblem, (2) the maximum sum lies completely in the right subproblem, and (3) the maximum sum spans across the divide point. The first two cases are easy and have already been taken care of by the recursive calls. The more interesting case is when the largest sum goes between the two subproblems.

How do we tackle this case? It is not hard to see that the maximum sum going across the divide is the largest sum of a suffix on the left and the largest sum of a prefix on the right. Cost aside, this leads to the following algorithm:

```
1  fun mcss(s) =
2    case (showt s)
3      of EMPTY = −∞
4       | ELT(x) = x
5       | NODE(L,R) =
6           let val (m_L, m_R) = (mcss(L) || mcss(R))
7               val m_A = bestAcross(L,R)
8           in max{m_L, m_R, m_A}
9           end
```

We will show you soon how to compute the max prefix and suffix sums in parallel, but for now, we'll take it for granted that they can be done in $O(n)$ work and $O(\log n)$ span. We'll also assume that dividing takes $O(\log n)$ work and span (as you have seen in 15-150). This yields the following recurrences:

$$\begin{aligned} W(n) &= 2W(n/2) + O(n) \\ S(n) &= S(n/2) + O(\log n) \end{aligned}$$

Proof of correctness. Let's switch gear and talk about proof of correctness. So far, as was the case in 15-150, you are familiar with writing detailed proofs that reason about essentially every step down to the most elementary operations. In this class, although we still expect your proof to be rigorous, we are more interested in seeing the critical steps highlighted and less important or standard ones summarized, with the idea being that if probed, you can easily provide detail on demand. The idea is that we want to make key ideas in an algorithm stand out as much as we can. It will be difficult for us to specify exactly how detailed we expect the proof to be, but you will pick it up by example.

We'll now prove that the `mcss` algorithm above computes the maximum contiguous subsequence sum. Specifically, we're proving the following theorem:

Theorem 3.2. *Let s be a sequence. The algorithm mcss(s) returns the maximum contiguous subsequence sum in s—and returns $-\infty$ if s is empty.*

Proof. The proof will be by (strong) induction on length. We have two base cases: one when the sequence is empty and one when it has one element. On the empty sequence, it returns $-\infty$ as we stated. On any singleton sequence $\langle x \rangle$, the MCSS is x, for which

$$\max\left\{\sum_{k=i}^{j} s_k \;:\; 1 \le i \le 1, 1 \le j \le 1\right\} = \sum_{k=1}^{1} s_k = s_1 = x\,.$$

For the inductive step, let s be a sequence of length $n > 1$, and assume inductively that for any sequence s' of length $n' < n$, $\texttt{mcss}(s')$ correctly computes the maximum contiguous subsequence sum. Now consider the sequence s and let L and R denote the left and right subsequences resulted from dividing s in half (i.e., `NODE(L, R) = showt s`). Furthermore, let $s_{i..j}$ be any contiguous subsequence of s that has the largest sum, and this value is v. Note that the proof has to account for the fact there might be many other subsequences with equal sum. Every contiguous subsequence must start somewhere and end after it. We consider the following 3 possibilities corresponding to how the sequence $s_{i..j}$ lies with respect to L and R:

- If the sequence $s_{i..j}$ starts in L and ends R then its sum equals its part in L (a suffix of L) and its part in R (a prefix of R). If we take the maximum of all suffixes in R and prefixes in L and add them this must equal the maximum of all contiguous sequences bridging the two since $\max\{a+b : a \in A, b \in B\}\} = \max\{a \in A\} + \max\{b \in B\}$. By assumption this equals the sum of $s_{i..j}$ which is v. Furthermore by induction m_L and m_R are sums of other subsequences so they cannot be any larger than v and hence $\max\{m_L, m_R, m_A\} = v$.
- If $s_{i..j}$ lies entirely in L, then it follows from our inductive hypothesis that $m_L = v$. Furthermore m_R and m_A correspond to the maximum sum of other subsequences which cannot be larger than v so again $\max\{m_L, m_R, m_A\} = v$.
- Similarly, if $s_{i..j}$ lies entirely in R, then it follows from our inductive hypothesis that $m_r = \max\{m_L, m_R, m_A\} = v$.

We conclude that in all cases, we return $\max\{m_L, m_R, m_A\} = v$, as claimed. □

Solving these recurrences: Using the definition of big-O, we know that

$$W(n) \quad \leq \quad 2W(n/2) + k_1 \cdot n + k_2,$$

where k_1 and k_2 are constants.

We have solved this recurrence using the tree method. We can also arrive at the same answer by mathematical induction. If you want to go via this route (and you don't know the answer a priori), you'll need to guess the answer first and check it. This is often called the "substitution method." Since this technique relies on guessing an answer, you can sometimes fool yourself by giving a false proof. The following are some tips:

1. Spell out the constants. Do not use big-O—we need to be precise about constants, so big-O makes it super easy to fool ourselves.
2. Be careful that the induction goes in the right direction.
3. Add additional lower-order terms, if necessary, to make the induction go through.

Let's now redo the recurrences above using this method. Specifically, we'll prove the following theorem using (strong) induction on n.

Theorem 3.3. *Let a constant $k > 0$ be given. If $W(n) \leq 2W(n/2) + k \cdot n$ for $n > 1$ and $W(1) \leq k$ for $n \leq 1$, then we can find constants κ_1 and κ_2 such that*

$$W(n) \leq \kappa_1 \cdot n \log n + \kappa_2.$$

Proof. Let $\kappa_1 = 2k$ and $\kappa_2 = k$. For the base case ($n = 1$), we check that $W(1) = k \leq \kappa_2$. For the inductive step ($n > 1$), we assume that

$$W(n/2) \leq \kappa_1 \cdot \tfrac{n}{2} \log(\tfrac{n}{2}) + \kappa_2,$$

And we'll show that $W(n) \leq \kappa_1 \cdot n \log n + \kappa_2$. To show this, we substitute an upper bound for $W(n/2)$ from our assumption into the recurrence, yielding

$$\begin{aligned} W(n) &\leq 2W(n/2) + k \cdot n \\ &\leq 2(\kappa_1 \cdot \tfrac{n}{2} \log(\tfrac{n}{2}) + \kappa_2) + k \cdot n \\ &= \kappa_1 n(\log n - 1) + 2\kappa_2 + k \cdot n \\ &= \kappa_1 n \log n + \kappa_2 + (k \cdot n + \kappa_2 - \kappa_1 \cdot n) \\ &\leq \kappa_1 n \log n + \kappa_2, \end{aligned}$$

where the final step follows because $k \cdot n + \kappa_2 - \kappa_1 \cdot n \leq 0$ as long as $n > 1$. □

Is it possible to do better than $O(n \log n)$ work using divide and conquer?

3.3 Algorithm 3: Divide And Conquer — Version 2.0

As it turns out, we can do better than $O(n \log n)$ work. The key is to strengthen the (sub)problem—i.e., solving a problem that is slightly more general—-to get a faster algorithm. Looking back at our previous divide-and-conquer algorithm, the "bottleneck" is that the combine step takes linear work. Is there any useful information from the subproblems we could have used to make the combine step take constant work instead of linear work?

In the design of our previous algorithm, we took advantage of the fact that if we know the max suffix sum and max prefix sums of the subproblems, we can produce the max subsequence sum in constant time. The expensive part was in fact computing these prefix and suffix sums—we had to spend linear work because we didn't know how generate the prefix and suffix sums for the next level up without recomputing these sums. *Can we easily fix this?*

The idea is to return the overall sum together with the max prefix and suffix sums, so we return a total of 4 values: the max subsequence sum, the max prefix sum, the max suffix sum, and the overall sum. Having this information from the subproblems is enough to produce a similar answer tuple for all levels up, in constant work and span per level. More specifically, *we strengthen our problem to return a* 4*-tuple* (`mcss, max-prefix, max-suffix, total`)*, and if the recursive calls return* (m_1, p_1, s_1, t_1) *and* (m_2, p_2, s_2, t_2)*, then we return*

$$(\max(s_1 + p_2, m_1, m_2), \max(p_1, t_1 + p_2), \max(s_1 + t_2, s_2), t_1 + t_2)$$

This gives the following pseudocode:

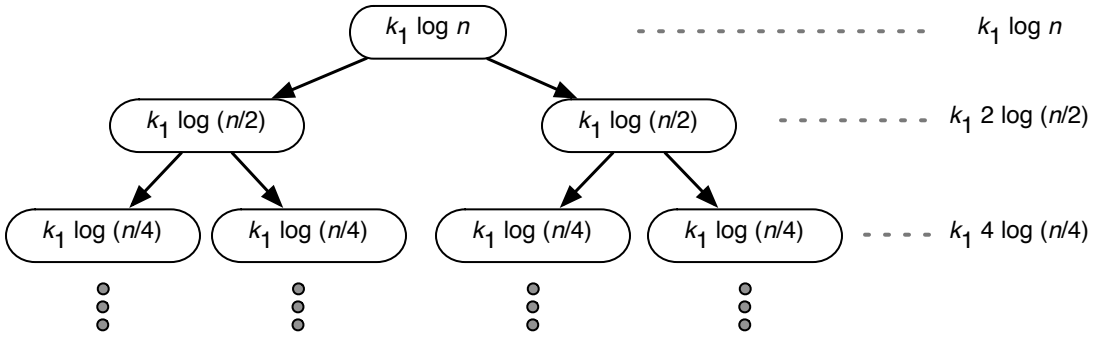
```
1  fun mcss(a) =
2  let
3    fun mcss'(a)
4      case (showt a)
5        of EMPTY = (−∞, −∞, −∞, 0)
6         | ELT(x) = (x, x, x, x)
7         | NODE(L, R) =
8             let
9               val ((m1, p1, s1, t1), (m2, p2, s2, t2)) = (mcss(L) || mcss(R))
10            in
11              (max(s1 + p2, m1, m2), max(p1, t1 + p2), max(s1 + t2, s2), t1 + t2)
12            end
13   val (m, p, s, t) = mcss'(a)
14 in m end
```

You should verify the base cases are doing the right thing. Also you can see the SML code for this algorithm using SML records at the end of these notes.

Cost Analysis. Assuming `showt` takes $O(\log n)$ work and span, we have the recurrences

$$\begin{aligned} W(n) &= 2W(n/2) + O(\log n) \\ S(n) &= S(n/2) + O(\log n) \end{aligned}$$

These are similar recurrences to what you have in Homework 1. Note that the span is the same as before, so we'll focus on analyzing the work. Using the tree method, we have



Therefore, the total work is upper-bounded by

$$W(n) \leq \sum_{i=0}^{\log n} k_1 2^i \log(n/2^i)$$

It is not so obvious what this sum evaluates to. The substitution method seems to be more convenient. We'll make a guess that $W(n) \leq \kappa_1 n - \kappa_2 \log n - k_3$. More formally, we'll prove the following theorem:

Theorem 3.4. *Let $k > 0$ be given. If $W(n) \leq 2W(n/2) + k \cdot \log n$ for $n > 1$ and $W(n) \leq k$ for $n \leq 1$, then we can find constants κ_1, κ_2, and κ_3 such that*

$$W(n) \leq \kappa_1 \cdot n - \kappa_2 \cdot \log n - \kappa_3.$$

Proof. Let $\kappa_1 = 3k$, $\kappa_2 = k$, $\kappa_3 = 2k$. We begin with the base case. Clearly, $W(1) = k \leq \kappa_1 - \kappa_3 = 3k - 2k = k$. For the inductive step, we substitute the inductive hypothesis into the recurrence and obtain

$$\begin{aligned} W(n) &\leq 2W(n/2) + k \cdot \log n \\ &\leq 2(\kappa_1 \tfrac{n}{2} - \kappa_2 \log(n/2) - \kappa_3) + k \cdot \log n \\ &= \kappa_1 n - 2\kappa_2(\log n - 1) - 2\kappa_3 + k \cdot \log n \\ &= (\kappa_1 n - \kappa_2 \log n - \kappa_3) + (k \log n - \kappa_2 \log n + 2\kappa_2 - \kappa_3) \\ &\leq \kappa_1 n - \kappa_2 \log n - \kappa_3, \end{aligned}$$

where the final step uses the fact that $(k \log n - \kappa_2 \log n + 2\kappa_2 - \kappa_3) = (k \log n - k \log n + 2k - 2k) = 0 \leq 0$ by our choice of κ's. □

Finishing the tree method. It is possible to solve the recurrence directly by evaluating the sum we established using the tree method. We didn't cover this in lecture, but for the curious, here's how you can "tame" it.

$$\begin{aligned} W(n) &\leq \sum_{i=0}^{\log n} k_1 2^i \log(n/2^i) \\ &= \sum_{i=0}^{\log n} k_1 \left(2^i \log n - i \cdot 2^i\right) \\ &= k_1 \left(\sum_{i=0}^{\log n} 2^i\right) \log n - k_1 \sum_{i=0}^{\log n} i \cdot 2^i \\ &= k_1(2n-1)\log n - k_1 \sum_{i=0}^{\log n} i \cdot 2^i. \end{aligned}$$

We're left with evaluating $s = \sum_{i=0}^{\log n} i \cdot 2^i$. Observe that if we multiply s by 2, we have

$$2s = \sum_{i=0}^{\log n} i \cdot 2^{i+1} = \sum_{i=1}^{1+\log n} (i-1)2^i,$$

so then

$$\begin{aligned} s &= 2s - s = \sum_{i=1}^{1+\log n} (i-1)2^i - \sum_{i=0}^{\log n} i \cdot 2^i \\ &= \left((1+\log n) - 1\right) 2^{1+\log n} - \sum_{i=1}^{\log n} 2^i \\ &= 2n\log n - (2n-2). \end{aligned}$$

Substituting this back into the expression we derived earlier, we have $W(n) \leq k_1(2n-1)\log n - 2k_1(n\log n - n + 1) \in O(n)$ because the $n\log n$ terms cancel.

4 SML Code

The following code for Algorithm 3 for the MCSS problem uses SML records to give names to each of the four fields. When passing around multiple values such records can make your code more clear and less prone to errors than using unnamed tuples. It is easy to forget the order you store things in tuples, and harder for people who are reading your code to know the order. Fields of a record can be extracted using pattern matching, or a single filed can be accessed using `#fieldname record`, as can be seen at the end of the code.

```
functor mcssDivConq(Seq : SEQUENCE) =
struct
  fun MCSS (A) =
  let
    fun MCSS' A =
      case Seq.showt A
      of Seq.ELT(v) => {mcss = v, prefix = v, suffix = v, total = v}
       | Seq.NODE(A1,A2) =>
         let
            val (B1, B2) = Primitives.par(fn () => MCSS'(A1),
                                          fn () => MCSS'(A2))
            val {mcss = M1, prefix = P1, suffix = S1, total = T1} = B1
            val {mcss = M2, prefix = P2, suffix = S2, total = T2} = B2
         in
            {mcss =   Int.max(S1 + P2, Int.max(M1, M2)),
             prefix = Int.max(P1,      T1 + P2),
             suffix = Int.max(S2,      S1 + T2),
             total = T1 + T2}
         end
  in
    case Seq.showt A
    of Seq.EMPTY => NONE
     |   _       => SOME(#mcss(MCSS' A))
  end
end
```

Lecture 4 — Divide and Conquer Continued

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — September 6

Material in this lecture:

- Maximum Contiguous Subsequence Sum (MCSS) problem (continued from last lecture) with two different divide-and-conquer solutions.
- Solving recurrences using substitution (in the last lecture notes)
- Euclidean Traveling Salesperson Problem using Divide and conquer.

Divide and Conquer Example II: The Euclidean Traveling Salesperson Problem

We'll now turn to another example of divide and conquer. In this example, we will apply it to devise a heuristic method for an **NP**-hard problem. The problem we're concerned with is a variant of the traveling salesperson problem (TSP) from Lecture 1. This variant is known as the Euclidean traveling salesperson (eTSP) problem because in this problem, the points (aka. cities, nodes, and vertices) lie in a Euclidean space and the distance measure is the Euclidean measure. More specifically, we're interested in the planar version of the eTSP problem, defined as follows:

Definition 0.1 (The Planar Euclidean Traveling Salesperson Problem)**.** Given a set of points P in the 2-d plane, the *planar Euclidean traveling salesperson* (eTSP) problem is to find a tour of minimum total distance that visits all points in P exactly once, where the distance between points is the Euclidean (i.e. ℓ_2) distance.

Not counting bridges, this is the problem we would want to solve to find a minimum length route visiting your favorite places in Pittsburgh. As with the TSP, it is **NP**-hard, but this problem is easier[1] to approximate.

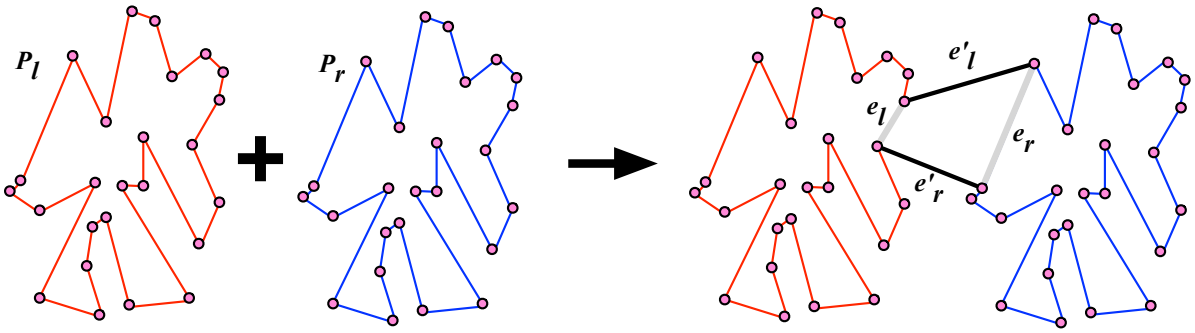
Here is a heuristic divide-and-conquer algorithms that does quite well in practice. In a few weeks, we will see another algorithm based on Minimum Spanning Trees (MST) that gives a constant-approximation guarantee. This divide-and-conquer is more interesting than the ones we have done so far because it does work both before and after the recursive calls. Also, as we will see, the recurrence it generates is root dominated.

The basic idea is to split the points by a cut in the plane, solve the TSP on the two halves, and then somehow merge the solutions. For the cut, we can pick a cut that is orthogonal to the coordinate lines. In particular, we can find in which of the two dimensions the points have a larger spread, and then find the median point along that dimension. We'll split just below that point.

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

[1]Unlike the TSP problem, which only has constant approximations, it is known how to approximate this problem to an arbitrary but fixed constant accuracy ε in polynomial time (the exponent of n has $1/\varepsilon$ dependency). That is, such an algorithm is capable of producing a solution that has length at most $(1+\varepsilon)$ times the length of the best tour.

To merge the solutions we join the two cycles by making an edge swap



To choose which edge swap to make, we consider all pairs of edges of the recursive solutions consisting of one edge $e_\ell = (u_\ell, v_\ell)$ from the left and one edge $e_r = (u_r, v_r)$ from the right and determine which pair minimizes the increase in the following cost:

$$\texttt{swapCost}((u_\ell, v_\ell), (u_r, v_r)) = \|u_\ell - v_r\| + \|u_r - v_\ell\| - \|u_\ell - v_\ell\| - \|u_r - v_r\|$$

where $\|u - v\|$ is the Euclidean distance between points u and v.

Here is the pseudocode for the algorithm

```
1  fun eTSP(P) =
2    case (|P|)
3      of 0,1 ⇒ raise TooSmall
4       | 2 ⇒ {(P[0],P[1]),(P[1],P[0])}
5       | n ⇒ let
6              val (P_ℓ, P_r) = splitLongestDim(P)
7              val (L,R) = ( eTSP(P_ℓ) || eTSP(P_r) )
8              val (c,(e,e')) = minVal_first {(swapCost(e,e'),(e,e')) : e ∈ L, e' ∈ R}
9            in
10             swapEdges(append(L,R),e,e')
11           end
```

The function $\texttt{swapEdges}(E, e, e')$ finds the edges e and e' in E and swaps the endpoints (there are two ways to swap, so the cheaper is picked).

Now let's analyze the cost of this algorithm in terms of work and span. We have

$$\begin{aligned} W(n) &= 2W(n/2) + O(n^2) \\ S(n) &= S(n/2) + O(\log n) \end{aligned}$$

We have already seen the recurrence $S(n) = S(n/2) + O(\log n)$, which solves to $O(\log^2 n)$. Here we'll focus on solving the work recurrence.

In anticipation of recurrences that you'll encounter later in class, we'll attempt to solve a more general form of recurrences. Let $\varepsilon > 0$ be a constant. We'll solve the recurrence

$$W(n) = 2W(n/2) + k \cdot n^{1+\varepsilon}$$

by the substitution method.

Theorem 0.2. *Let $\varepsilon > 0$. If $W(n) \leq 2W(n/2) + k \cdot n^{1+\varepsilon}$ for $n > 1$ and $W(1) \leq k$ for $n \leq 1$, then for some constant κ,*

$$W(n) \leq \kappa \cdot n^{1+\varepsilon}.$$

Proof. Let $\kappa = \frac{1}{1-1/2^\varepsilon} \cdot k$. The base case is easy: $W(1) = k \leq \kappa_1$ as $\frac{1}{1-1/2^\varepsilon} \geq 1$. For the inductive step, we substitute the inductive hypothesis into the recurrence and obtain

$$\begin{aligned} W(n) &\leq 2W(n/2) + k \cdot n^{1+\varepsilon} \\ &\leq 2\kappa \left(\frac{n}{2}\right)^{1+\varepsilon} + k \cdot n^{1+\varepsilon} \\ &= \kappa \cdot n^{1+\varepsilon} + \left(2\kappa \left(\frac{n}{2}\right)^{1+\varepsilon} + k \cdot n^{1+\varepsilon} - \kappa \cdot n^{1+\varepsilon}\right) \\ &\leq \kappa \cdot n^{1+\varepsilon}, \end{aligned}$$

where in the final step, we argued that

$$\begin{aligned} 2\kappa \left(\frac{n}{2}\right)^{1+\varepsilon} + k \cdot n^{1+\varepsilon} - \kappa \cdot n^{1+\varepsilon} &= \kappa \cdot 2^{-\varepsilon} \cdot n^{1+\varepsilon} + k \cdot n^{1+\varepsilon} - \kappa \cdot n^{1+\varepsilon} \\ &= \kappa \cdot 2^{-\varepsilon} \cdot n^{1+\varepsilon} + (1 - 2^{-\varepsilon})\kappa \cdot n^{1+\varepsilon} - \kappa \cdot n^{1+\varepsilon} \\ &\leq 0. \end{aligned}$$

□

Solving the recurrence directly. Alternatively, we could use the tree method and evaluate the sum directly. As argued before, the recursion tree here has depth $\log n$ and at level i (again, the root is at level 0), we have 2^i nodes, each costing $k \cdot (n/2^i)^{1+\varepsilon}$. Thus, the total cost is

$$\begin{aligned} \sum_{i=0}^{\log n} k \cdot 2^i \cdot \left(\frac{n}{2^i}\right)^{1+\varepsilon} &= k \cdot n^{1+\varepsilon} \cdot \sum_{i=0}^{\log n} 2^{-i\cdot\varepsilon} \\ &\leq k \cdot n^{1+\varepsilon} \cdot \sum_{i=0}^{\infty} 2^{-i\cdot\varepsilon}. \end{aligned}$$

But the infinite sum $\sum_{i=0}^{\infty} 2^{-i\cdot\varepsilon}$ is at most $\frac{1}{1-1/2^\varepsilon}$. Hence, we conclude $W(n) \in O(n^{1+\varepsilon})$.

Lecture 5 — Data Abstraction and Sequences I

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — 11 Septermber 2012

Material in this lecture:

- Relationship between ADTs, cost specifications and data structures.
- The sequence ADT
- The scan operation: examples and implementation
- Using contraction : an algorithmic technique

1 Abstract Data Types, Cost Specifications, and Data Structures

So far in class we have defined several "problems" and discussed algorithms for solving them. The idea is that the problem is an abstract definition of what we want in terms of a function specification, and the algorithms are particular ways to solve/implement the problem. In addition to abstract functions we also often need to define abstractions over data. In such an abstraction we define a set of functions (abstractly) over a common data type. As mentioned in the first lecture, we will refer to the abstractions as abstract data types and their implementations as data structures.

An example of an abstract data type, or ADT, you should have seen before is a priority queue. Let's consider a slight extension where in addition to insert, and deleteMin, we will include a function that joins two priority queues into a single one. For historical reasons, we will call such a join a *meld*, and the ADT a "meldable priority queue". As with a problem, we like to have a formal definition of the abstract data type.

Definition 1.1. Given a totally ordered set $\mathbb{S}$, a *Meldable Priority Queue* (MPQ) is a type $\mathbb{T}$ representing subsets of $\mathbb{S}$, along with the following values and functions (partial list):

$$
\begin{array}{lcll}
\texttt{empty} & : & \mathbb{T} & = \{\} \\
\texttt{insert}(S,e) & : & \mathbb{T}\times\mathbb{S}\to\mathbb{T} & = S\cup\{e\} \\
\texttt{deleteMin}(S) & : & \mathbb{T}\to\mathbb{T}\times(\mathbb{S}\cup\{\bot\}) & = \begin{cases} (S,\bot) & S=\{\} \\ (S\setminus\{\min S\},\min S) & otherwise \end{cases} \\
\texttt{meld}(S_1,S_2) & : & \mathbb{T}\times\mathbb{T}\to\mathbb{T} & = S_1\cup S_2
\end{array}
$$

Here we are using standard set notation: empty braces {} denotes an empty set, $A\cup B$ denotes taking the union of the sets A and B, and $A\setminus B$ denotes taking the set difference of A and B (i.e., all the elements in A except those that appear in B). Note that `deleteMin` returns the special element $\bot$ when the queue is empty; it indicates that the function is undefined when the priority queue is empty.

When translated to SML this definition corresponds to a signature of the form:

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

```
signature MPQ
 sig
   struct S : ORD
   type t
   val empty : t
   val insert : t * S.t -> t
   val deleteMin : t -> t * S.t option
   val meld : t * t -> t
 end
```

Note that the type `t` is abstracted (i.e. it is not specified to be sets of elements of type `S.t`) since we don't want to access the queue as a set but only through its interface. Note also that the signature by itself does not specify the semantics but only the types (e.g., it could be `insert` does nothing with its second argument). To be an ADT we have to add the semantics as written on the righthand side of the equations in Definition 2.1.

In general SML signatures for ADTs will look like:

```
signature ADT
 sig
   struct S1 : OtherADT
   ...
   type t
   helper types
   val v1 : ... t ...
   val v2 : ... t ...
   ...
 end
```

1.1 Cost Specifications

Now the operations on a meldable priority queue might have different costs depending on the particular data structures used to implement them. If we are a "client" using a priority queue as part of some algorithm or application we surely care about the costs, but probably don't care about the specific implementation. We therefore would like to have abstract cost associated with the interface. For example we might have for work:

	I1	**I2**	**I3**
`insert`(S,e)	$O(\|S\|)$	$O(\log \|S\|)$	$O(\log \|S\|)$
`deleteMin`(S)	$O(1)$	$O(\log \|S\|)$	$O(\log \|S\|)$
`meld`$(S1,S2)$	$O(\|S_1\| + \|S_2\|)$	$O(\|S_1\| + \|S_2\|)$	$O(\log(\|S_1\| + \|S_2\|))$

You have already seen data structures that match the first two bounds. For the first one maintaining a sorted array will do the job. You have seen a couple that match the second bounds. What are they? We will be covering the third bound, which has a faster `meld` operation than the others, later in the course.

In any case, these cost definitions sit between the ADT and the specific data structures used to implement them. We will refer to them as *cost specifications*. We therefore have three levels:

1. **The abstract data type**: The definition of the interface for the purpose of describing functionality and correctness criteria.
2. **The cost specification**: The definition of the costs for each of the functions in the interface. There can be multiple different cost specifications for an ADT depending on the type of implementation. Although not required to show correctness this specification is required to analyze performance.
3. **The implementation as a data structure**: This is the particular data structure used to implement the ADT. There might be multiple data structures that match the same cost specification. If you are a user of the ADT you don't need to know what this is, although it is good to be curious.

2 The Sequence ADT

The first ADT we will go through in some detail is the sequence ADT. You have used sequences in 15-150. But we will add some new functionality and will go through the cost specifications in more detail. There are two cost specifications for sequences we will consider, one based on an array implementation, and the other based on a tree implementation.

We first do a quick review of set theory so we can more formally define a sequence. A *relation* is a set of ordered pairs. In particular for two sets α and β, ρ is a relation from α to β if $\rho \subseteq \alpha \times \beta$. Here $\alpha \times \beta$ indicates the set of all ordered pairs made from taking the first element from α and the second from β. A *function* is a relation ρ such that for every a in the domain of ρ there is only one b such that $(a, b) \in \rho$. A *sequence* is a function whose domain is $\{0, \ldots, n-1\}$ for some $n \in \mathbb{N}$ (we use $\mathbb{N}$ to indicate the natural numbers, including zero). We define the sequence ADT based on the mathematical definition along with specific operations it supports. We only list a subset. A more complete list can be found in the library documentation.

Definition 2.1. An *Sequence* is a type $\mathbb{S}_\alpha$ representing functions from $\mathbb{N}$ to α with domain $\{0, \ldots, n-1\}$ for some $n \in \mathbb{N}$, and supporting the following values and functions:

`empty`	:	$\mathbb{S}_\alpha$	=	$\{\}$
`length`(A)	:	$\mathbb{S}_\alpha \to \mathbb{N}$	=	$\lvert A \rvert$
`singleton`(v)	:	$\alpha \to \mathbb{S}_\alpha$	=	$\{(0, v)\}$
`nth`(A, i)	:	$\mathbb{S}_\alpha \to \alpha$	=	$A(i)$
`map`(f, A)	:	$(\alpha \to \beta) \times \mathbb{S}_\alpha \to \mathbb{S}_\beta$	=	$\{(i, f(v)) : (i, v) \in A\}$
`tabulate`(f, n)	:	$(\mathbb{N} \to \alpha) \times \mathbb{N} \to \mathbb{S}_\alpha$	=	$\{(i, f(i)) : i \in \{0, \ldots, n-1\}\}$
`take`(A, n)	:	$\mathbb{S}_\alpha \times \mathbb{N} \to \mathbb{S}_\alpha$	=	$\{(i, v) \in A \mid i < n\}$
`drop`(A, n)	:	$\mathbb{S}_\alpha \times \mathbb{N} \to \mathbb{S}_\alpha$	=	$\{(i - n, v) : (i, v) \in A \mid i \geq n\}$
`append`(A, B)	:	$\mathbb{S}_\alpha \times \mathbb{S}_\alpha \to \mathbb{S}_\alpha$	=	$A \cup \{(i + \lvert A \rvert, v) : (i, v) \in B\}$
...				

	ArraySequence		**TreeSequence**	
	Work	*Span*	*Work*	*Span*
`length`(T)	$O(1)$	$O(1)$	$---$	$---$
`nth`(T)	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$
`tabulate` f n	$O\left(\sum_{i=0}^{n} W(f(i))\right)$	$O\left(\max_{i=0}^{n} S(f(i))\right)$	$---$	$O\left(\log n + \max_{i=0}^{n} S(f(i))\right)$
`map` f S	$O\left(\sum_{s\in S} W(f(s))\right)$	$O\left(\max_{s\in S} S(f(s))\right)$	$---$	$O\left(\log \lvert S\rvert + \max_{s\in S} S(f(s))\right)$
`append`(S_1, S_2)	$O(\lvert S_1\rvert + \lvert S_2\rvert)$	$O(1)$	$O(\log(\lvert S_1\rvert + \lvert S_2\rvert))$	$O(\log(\lvert S_1\rvert + \lvert S_2\rvert))$

Figure 1: Sample *cost specifications* for the Array and Tree based implementations of Sequences. The $---$ means it is the same as in ArraySequence. A full list of costs can be found in the "Syntax and Costs" document.

The *arraySequence* and *treeSequence* cost specificatios for the Sequence ADT are given figure 1 for a few of the functions. A more complete list can be found in the "syntax and costs" document.

In the pseudocode used in the course notes we will use a variant on set notation to indicate various operations over sequences. In particular we use triangular brackets $\langle\ \rangle$ instead of curly brackets {}. Here are some examples of the notation we will use:

S_i	The i^{th} element of sequence S
$\lvert S\rvert$	The length of sequence S
$\langle\ \rangle$	The empty sequence
$\langle v \rangle$	A sequence with a single element v
$\langle i, \ldots, j \rangle$	A sequence of integers starting at i and ending at $j \geq i$.
$\langle e : p \in S \rangle$	**Map** the expression e to each element p of sequence S. The same as "`map` (**fn** $p \Rightarrow e$) S" in ML.
$\langle p \in S \mid e \rangle$	**Filter** out the elements p in S that satisfy the predicate e. The same as "`filter` (**fn** $p \Rightarrow e$) S" in ML.

More examples are given in the "Syntax and Costs" document.

We will now cover some of the sequence functions in more detail, including

- `reduce`, `iter`, `scan` and `iterh`
- `tokens`, `fields`
- `collect`

You have most likely seen `reduce` before, but we will cover more applications of it, and how to analyze its cost when the combining function requires more than constant work and span.

3 The Scan Operation

A function closely related to `reduce` is `scan`. We mentioned it during the last lecture and you covered it in recitation. It has the interface:

$$\texttt{scan}\ f\ I\ S : (\alpha \times \alpha \to \alpha) \to \alpha \to \alpha\ \texttt{seq} \to (\alpha\ \texttt{seq} \times \alpha)$$

As with `reduce`, when the function f is associative, the scan function returns the sum with respect to f of each prefix of the input sequence S, as well as the total sum of S. Hence the operation is often called the *prefix sums* operation. For a function f which is associative it can be defined as follows:

```
1  fun scan f I S =
2    (⟨ reduce f I (take(S, i)) : i ∈ ⟨0, ... n − 1⟩ ⟩,
3     reduce f I S)
```

This uses our pseudocode notation and the $\langle\, \texttt{reduce}\ f\ I\ (\texttt{take}(S,i)) : i \in \langle 0,\ldots,n-1\rangle \,\rangle$ indicates that for each i in the range from 0 to $n-1$ apply reduce to the first i elements of S. For example,

$$\begin{aligned}\texttt{scan}\ +\ 0\ \langle 2,1,3\rangle &= (\langle\, \texttt{reduce}\ +\ 0\ \langle\,\rangle, \texttt{reduce}\ +\ 0\ \langle 2\rangle, \texttt{reduce}\ +\ 0\ \langle 2,1\rangle\ \rangle \\ &\qquad \texttt{reduce}\ +\ 0\ \langle 2,1,3\rangle) \\ &= (\langle 0,2,3\rangle, 6)\end{aligned}$$

Using a bunch of reduces, however, is not an efficient way to calculate the partial sums.

Exercise 1. *What is the work and span for the* `scan` *code shown above, assuming f takes constant work.*

We will soon see how to implement a scan with the following bounds:

$$\begin{aligned}W(\texttt{scan}\ f\ I\ S) &= O(|S|) \\ S(\texttt{scan}\ f\ I\ S) &= O(\log|S|)\end{aligned}$$

assuming that the function f takes constant work. For now we will consider some useful applications of scans.

Note that the scan operation takes the "sum" of the elements before the position i. Sometimes it is useful to include the value at position i. We therefore also will use a version of such an *inclusive scan*.

$$\texttt{scanI}\ +\ 0\ \langle 2,1,3\rangle \;=\; \langle 2,3,6\rangle$$

This version does not return a second result since the total sum is already included in the last position.

3.1 The MCSS Problem Algorithm 5: Using Scan

Let's consider how we might use scan operations to solve the Maximum contiguous subsequence (MCSS) problem. Recall, this problem is given a sequence S to find:

$$\max_{0 \le i \le j \le n} (\sum_{k=i}^{j-1} S_k) .$$

As a running example for this section consider the sequence

$$S = \langle 1, -2, 3, -1, 2, -3 \rangle .$$

What if we do an inclusive scan on our input S using addition? i.e.:

$$X = \texttt{scanI + 0}\ S = \langle 1, -1, 2, 1, 3, 0 \rangle$$

Now for any j^{th} position consider all positions $i < j$. To calculate the sum from immediately after i to j all we have to do is return $X_j - X_i$. This difference represents the total sum of the subsequence from $i+1$ to j since we are taking the sum up to j and then subtracting off the sum up to i. For example to calculate the sum between the -2 (location $i+1=1$) and the 2 (location $i=4$) we take $X_4 - X_0 = 3 - 1 = 2$, which is indeed the sum of the subsequence $\langle -2, 3, -1, 2 \rangle$.

Now consider how for each j we might calculate the maximum sum that starts at any $i \le j$ and ends at j. Call it R_j. This can be calculated as follows:

$$\begin{aligned} R_j &= \max_{i=0}^{j} \sum_{k=i}^{j} S_k \\ &= \max_{i=0}^{j} (X_j - X_{i-1}) \\ &= X_j + \max_{i=0}^{j} (-X_{i-1}) \\ &= X_j + \max_{i=0}^{j-1} (-X_i) \\ &= X_j - \min_{i=0}^{j-1} X_i \end{aligned}$$

The last equality is because the maximum of a negative is the minimum of the positive. This indicates that all we need to know is X_j and the minimum previous $X_i, i < j$. This can be calculated with a scan using the minimum operation. Furthermore the result of this scan is the same for everyone, so we need to calculate it just once. The result of the scan is:

$$(M, \_) = \texttt{scan min 0}\ X = (\langle 0, 0, -1, -1, -1, -1 \rangle, -1) ,$$

and now we can calculate R:

$$R = \left\langle X_j - M_j : 0 \le j < |S| \right\rangle = \langle 1, -1, 3, 2, 4, 1 \rangle .$$

You can verify that each of these represents the maximum contiguous subsequence sum ending at position j.

Finally, we want the maximum string ending at any position, which we can do with a reduce using max. This gives 4 in our example.

Putting this all together we get the following very simple algorithm:

```
1  fun MCSS(S) =
2  let
3    val X = scanI + 0 S
4    val (M, _) = scan min 0 X
5  in
6    max ⟨ X_j − M_j : 0 ≤ j < |S| ⟩
7  end
```

Given the costs for scan and the fact that addition and minimum take constant work, this algorithm has $O(n)$ work and $O(\log n)$ span.

3.2 Copy Scan

Previously, we used `scan` to compute partial sums to solve the maximum contiguous subsequence sum problem and to match parentheses. Scan is also useful when you want pass information along the sequence. For example, suppose you have some "marked" elements that you would like to copy across to their right until they reach another marked element. One way to mark the elements is to use options.

That is, suppose you are given a sequence of type α `option seq`. For example

$$\langle \texttt{NONE}, \texttt{SOME}(7), \texttt{NONE}, \texttt{NONE}, \texttt{SOME}(3), \texttt{NONE} \rangle$$

and your goal is to return a sequence of the same length where each element receives the previous SOME value. For the example:

$$\langle \texttt{NONE}, \texttt{NONE}, \texttt{SOME}(7), \texttt{SOME}(7), \texttt{SOME}(7), \texttt{SOME}(3) \rangle$$

Using a sequential loop or `iter` would be easy. How would you do this with `scan`?

If we are going to use a `scan` directly, the combining function f must have type

$$\alpha \; option \times \alpha \; option \to \alpha \; option$$

How about

```
1  fun copy(a, b) =
2    case b of
3      SOME(_) ⇒ b
4    | NONE ⇒ a
```

What this function does is basically pass on its right argument if it is `SOME` and otherwise it passes on the left argument.

There are many other applications of scan in which more involved functions are used. One important case is to simulate a finite state automata.

3.3 Contraction and Implementing Scan

Now let's consider how to implement `scan` efficiently and at the same time apply one of the algorithmic techniques from our toolbox of techniques: *contraction*. Throughout the following discussion we assume the work of the binary operator is $O(1)$. As described earlier a brute force method for calculating scans is to apply a reduce to all prefixes. This requires $O(n^2)$ work and is therefore not efficient (we can do it in $O(n)$ work sequentially).

Beyond the wonders of what it can do, a surprising fact about `scan` is that it can be accomplished efficiently in parallel, although on the surface, the computation it carries out appears to be sequential in nature. At first glance, we might be inclined to believe that any efficient algorithms will have to keep a cumulative "sum," computing each output value by relying on the "sum" of the all values before it. It is this apparent dependency that makes `scan` so powerful. We often use `scan` when it seems we need a function that depends on the results of other elements in the sequence, for example, the copy scan above.

Suppose we are to run `plus_scan` (i.e. `scan (op +)`) on the sequence $\langle 2,1,3,2,2,5,4,1\rangle$. What we should get back is

$$(\langle 0,2,3,6,8,10,15,19\rangle, 20)$$

Thought Experiment I: At some level, this problem seems like it can be solved using the divide-and-conquer approach. Let's try a simple pattern: divide up the input sequence in half, recursively solve each half, and "piece together" the solutions. A moment's thought shows that the two recursive calls are not independent—indeed, the right half depends on the outcome of the left one because it has to know the cumulative sum. So, although the work is $O(n)$, we effectively haven't broken the chain of sequential dependencies. In fact, we can see that any scheme that splits the sequence into left and right parts like this will essentially run into the same problem.

Exercise 2. *Can you see a way to implement a scan in $O(n \log n)$ work and $O(\log n)$ span using divide and conquer? Hint: what can you do with the total sum that is returned on the left?*

Contraction: To compute `scan` in $O(n)$ work in parallel, we introduce a new inductive technique common in algorithms design: contraction. It is inductive in that such an algorithm involves solving a smaller instance of the same problem, much in the same spirit as a divide-and-conquer algorithm. But with contraction, there is only one recursive call. In particular, the contraction technique involves the following steps:

1. Contract the instance of the problem to a (much) smaller instance (of the same sort)
2. Solve the smaller instance recursively
3. Use the solution to help solve the original instance

The contraction approach is a useful technique in algorithm design in general but for various reasons it is more common in parallel algorithms than in sequential algorithms. This is usually because both the contraction and expansion steps can be done in parallel and the recursion only goes logarithmically deep because the problem size is shrunk by a constant fraction each time.

We'll demonstrate this technique first by applying it to a slightly simpler problem, `reduce`. To begin, we have to answer the following question: *How do we make the input instance smaller in a way that the solution on this smaller instance will benefit us in constructing the final solution?*

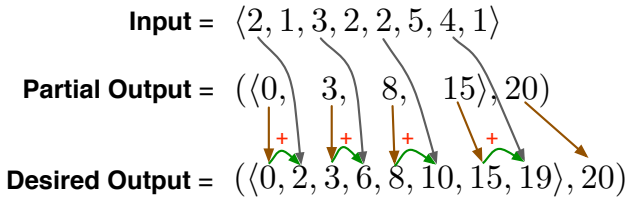
The idea is simple: We apply the combining function pairwise to adjacent elements of the input sequence and recursively run `reduce` on it. In this case, the third step is a "no-op"; it does nothing. For example on input sequence $\langle 2,1,3,2,2,5,4,1\rangle$ with addition, we would contract the sequence to $\langle 3,5,7,5\rangle$. Then we would continue to contract recursively to get the final result. There is no expansion step.

Thought Experiment II: How can we use the same idea to evaluate span? What would be the result after the recursive call? In the example above it would be

$$(\langle 0,3,8,15\rangle, 20).$$

But notice, this sequence is every other element of the final scan sequence, together with the final sum—and this is enough information to produce the desired final output. This time, the third expansion step is needed to fill in the missing elements in the final scan sequence: Apply the combining function element-wise to the even elements of the input sequence and the results of the recursive call to `scan`.

To illustrate, the diagram below shows how to produce the final output sequence from the original sequence and the result of the recursive call:



This leads to the following code. We'll first present `scan` in pseudocode for when n is a power of two, and then an actual implementation of `reduce` and `scan` in Standard ML.

1 ***% implements:*** *the Scan problem on sequences that have a power of 2 length*
2 **fun** `scanPow2` $f\ \ i\ \ s =$
3 **case** $|s|$ **of**
4 $0 \Rightarrow (\langle\rangle, i)$
5 $\mid\ 1 \Rightarrow (\langle i\rangle, s[0])$
6 $\mid\ n \Rightarrow$
7 **let**
8 **val** $s' = \langle f(s[2i], s[2i+1]) : 0 \leq i < n/2\rangle$
9 **val** $(r, t) =$ `scanPow2` $f\ \ i\ \ s'$
10 **in**
11 $(\langle p_i : 0 \leq i < n\rangle, t)$, **where** $p_i = \begin{cases} r[i/2] & \textbf{if } \texttt{even}(i) \\ f(r[i/2], s[i-1]) & \textbf{otherwise.} \end{cases}$
12 **end**

Notice that the reduction tree for `reduce` that we showed in the previous lecture is the same tree that the contract step uses in `scan`. In this way, the final sum in the `scan` output (the second of the pair) is the same as for `reduce`, even when the combining function is non-associative. Unfortunately, the same is not true for the scan sequence (the first of the pair); when the combining function is non-associative, the resulting scan sequence is not necessarily the same as applying `reduce` to prefixes of increasing length.

4 SML Code

```
fun scan f i s =
   case length s
    of 0 => (empty(), i)
     | 1 => (singleton i, f(i, nth s 0))
     | n =>
       let
         val s' = tabulate
                     (fn i => case (2*i = n - 1) of
                                 true =>  (nth s (2*i))
                               | _    => f(nth s (2*i), nth s (2*i + 1)))
                     ((n+1) div 2)
         val (r, t) = scan f i s'
         fun interleave i = case (i mod 2) of
                               0 =>  (nth r (i div 2))
                             | _ => f(nth r (i div 2), nth s (i-1))
       in
         (tabulate interleave n, t)
       end
```

Lecture 6 — Sequences II

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — September 13, 2012

Material in this lecture: Today's lecture is about *reduction*.

- Scan implementation (from the last lecture notes)
- Divide-and-conquer with reduction
- Cost of reduce when f is not constant work

1 Reduce Operation

Recall that `reduce` function has the interface

$$\texttt{reduce}\ f\ I\ S : (\alpha \times \alpha \to \alpha) \to \alpha \to \alpha\ \texttt{seq} \to \alpha$$

When the combining function f is associative—i.e., $f(f(x,y),z) = f(x,f(y,z))$ for all x, y and z of type α—`reduce` returns the sum with respect to f of the input sequence S. It is the same result returned by `iter` $f\ I\ S$. The reason we include `reduce` is that it is parallel, whereas `iter` is strictly sequential. Note, though, `iter` can use a more general combining function with type: $\beta \times \alpha \to \beta$.

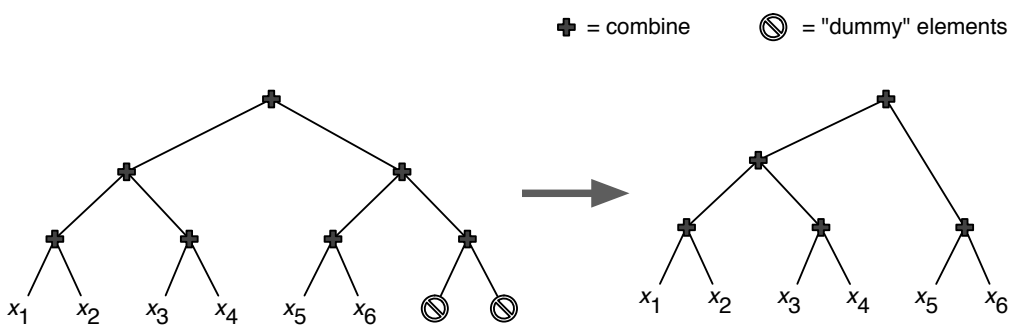
The results of `reduce` and `iter`, however, may differ if the combining function is non-associative. In this case, the order in which the reduction is performed determines the result; because the function is non-associative, different orderings will lead to different answers. While we might try to apply reduce to only associative operations, unfortunately even some functions that seem to be associative are actually not. For instance, floating point addition and multiplication are not associative. In SML/NJ, integer addition is also not associative because of the overflow exception.

To properly deal with combining functions that are non-associative, it is therefore important to specify the order that the combining function is applied to the elements of a sequence. This order is part of the specification of the ADT `Sequence`. In this way, every (correct) implementation returns the same result when applying `reduce`; the results are deterministic regardless of what data structure and algorithm are used.

For this reason, we define a specific combining tree, which is defined quite carefully in the library documentation for `reduce`. This tree is the same as if we rounded up the length of the input sequence to the next power of 2, i.e., $|x| = 2^k$, and then put a perfectly balanced binary tree[1] over the sequence with 2^k leaves. Wherever we are missing children in the tree, we don't apply the combining function. An example is shown in the following figure.

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

[1] A *perfect* binary tree is a tree in which every node other than the leaves have exactly 2 children.



In the next lecture we will offer an explanation why we chose this particular combining order.

1.1 Divide and Conquer with Reduce

Now, let's look back at divide-and-conquer algorithms you have encountered so far. Many of these algorithms have a "divide" step that simply splits the input sequence in half, proceed to solve the subproblems recursively, and continue with a "combine" step. This leads to the following structure where everything except what is in boxes is generic, and what is in boxes is specific to the particular algorithm.

```
1  fun myDandC(S) =
2    case showt(S) of
3      EMPTY ⇒ [emptyVal]
4    | ELT(v) ⇒ [base](v)
5    | NODE(L, R) ⇒ let
6        val (L′,R′) = (myDandC(L) || myDandC(R))
7      in
8        [someMessyCombine](L′,R′)
9      end
```

Algorithms that fit this pattern can be implemented in one line using the sequence `reduce` function. You have seen this in Homework 1 in which we asked for a reduce-based solution for the stock market problem. Turning such a divide-and-conquer algorithm into a reduce-based solution is as simple as invoking `reduce` with the following parameters:

```
reduce [someMessyCombine] [emptyVal] (map [base] S)
```

We will take a look two examples where `reduce` can be used to implement a relatively sophisticated divide-and-conquer algorithm. The first example we will consider today; the second we will considered in the next lecture.

Algorithm 4: MCSS Using Reduce.

The first example is the Maximum Contiguous Subsequence Sum problem from last lecture. Given a sequence S of numbers, find the contiguous subsequence that has the largest sum—more formally:

$$\text{mcss}(s) = \max\left\{\sum_{k=i}^{j} s_k \ : \ 1 \le i \le n, i \le j \le n\right\}.$$

Recall that the divide-and-conquer solution involved returning four values from each recursive call on a sequence S: the desired result `mcss`(S), the maximum prefix sum of S, the maximum suffix sum of S, and the total sum of S. We will denote these as M, P, S, T, respectively. To solve the mcss problem we can then use the following implementations for `combine`, `base`, and `emptyVal`:

fun `combine`$((M_L,P_L,S_L,T_L),(M_R,P_R,S_R,T_L)) =$
$(\max(S_L+P_R,M_L,M_R),\quad \max(P_L,T_L+P_R),\quad \max(S_R,S_L+T_R),\quad T_L+T_R)$

fun `base`$(v) = (v,v,v,v)$

val `emptyVal` $= (-\infty,-\infty,-\infty,0)$

and then solve the problem with:

```
fun mcss(S) =
  reduce combine emptyVal (map base S)
```

Stylistic Notes. We have just seen that we could spell out the divide-and-conquer steps in detail or condense our code into just a few lines that take advantage of the almighty `reduce`. *So which is preferable*, using the divide-and-conquer code or using reduce? We believe this is a matter of taste. Clearly, your reduce code will be (a bit) shorter, and for simple cases easy to write. But when the code is more complicated, the divide-and-conquer code is easier to read, and it exposes more clearly the inductive structure of the code and so is easier to prove correct.

Restriction. You should realize, however, that this pattern does not work in general for divide-and-conquer algorithms. In particular, it does not work for algorithms that do more than a simple split that partitions their input in two parts in the middle. For example, it cannot be used for implementing quick sort as the divide step partitions the data with respect to a pivot. This step requires picking a pivot, and then filtering the data into elements less than, equal, and greater than the pivot.

2 Analyzing the Costs of Higher Order Functions

Last lecture we looked at using `reduce` to solve divide-and-conquer problems. In the example we gave, the maximum increasing subsequence sum, the combining function f had $O(1)$ cost (i.e., both its work and span are constant). In that case the cost specifications of `reduce` on a sequence of

length n is simply $O(n)$ work and $O(\log n)$ span. Does that hold true when the combine function does not have constant cost?

For `map` it is easy to find its costs base on the cost of the function applied:

$$\begin{aligned} W(\texttt{map}\ f\ S) &= 1 + \sum_{s \in S} W(f(s)) \\ S(\texttt{map}\ f\ S) &= 1 + \max_{s \in S} S(f(s)) \end{aligned}$$

Tabulate is similar. But can we do the same for `reduce`?

Merge Sort. As an example, let's consider merge sort. As you have likely seen from previous courses you have taken, merge sort is a popular divide-and-conquer sorting algorithm with optimal work. It is based on a function `merge` that takes two already sorted sequences and returns a sorted sequence containing all elements from both sequences. We can use our reduction technique for implementing divide-and-conquer algorithms to implement merge sort with a `reduce`. In particular, we can write a version of merge sort, which we refer to as reduceSort, as follows:

```
val combine = merge_<
val base = singleton
val emptyVal = empty()
fun reduceSort(S) = reduce combine emptyVal (map base S)
```

where $\texttt{merge}_<$ is a merge function that uses an (abstract) comparison operator $<$. Note that merging is an associative function.

Assuming a constant work comparison function, two sequences S_1 and S_2 with lengths n_1 and n_2 can be merged with the following costs:

$$\begin{aligned} W(\texttt{merge}_<(S_1, S_2)) &= O(n_1 + n_2) \\ S(\texttt{merge}_<(S_1, S_2)) &= O(\log(n_1 + n_2)) \end{aligned}$$

What do you think the cost of `reduceSort` *is?*

2.1 Reduce: Cost Specifications

We want to analyze the cost of `reduceSort`. Does the reduction order matter? As mentioned before, if the combining function is associative, which it is in this case, all reduction orders give the same answer so it seems like it should not matter.

To answer this question, let's consider the reduction order we get by sequentially adding the elements in one after the other. On input $x = \langle x_1, x_2, \ldots, x_n \rangle$, the sequence of $\texttt{merge}_<$ calls looks like the following:

$$\texttt{merge}_<(\ldots\ \texttt{merge}_<(\texttt{merge}_<(\texttt{merge}_<(I, \langle\, x_1 \,\rangle), \langle\, x_2 \,\rangle), \langle\, x_3 \,\rangle), \ldots)$$

i.e. we first merge I and $\langle x_1 \rangle$, then merge in $\langle x_2 \rangle$, then $\langle x_3 \rangle$, etc.

With this order $\texttt{merge}_<$ is called when its left argument is a sequence of varying size between 1 and $n-1$, while its right argument is always a singleton sequence. The final merge combines $(n-1)$-element with 1-element sequences, the second to last merge combines $(n-2)$-element with 1-element sequences, so on so forth. Therefore, the total work for an input sequence S of length n is

$$W(\texttt{reduceSort}\ S) \ \leq \ \sum_{i=1}^{n-1} c \cdot (1+i) \ \in \ O(n^2)$$

since merge on sequences of lengths n_1 and n_2 has $O(n_1+n_2)$ work.

Note that this reduction order is the order that the `iter` function uses, and hence is equivalent to:

```
fun reduceSort'(S) =
  iter merge< (empty()) (map singleton S)
```

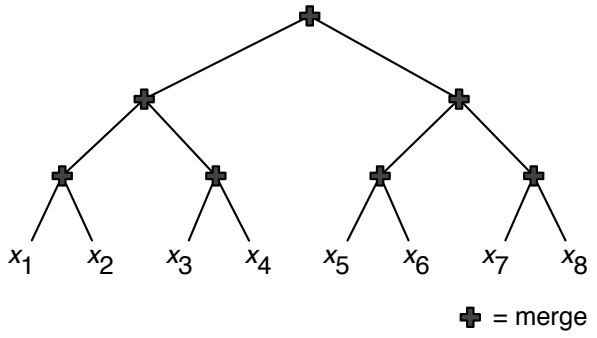
Furthermore, using this reduction order, the algorithm is effectively working from the front to the rear, "inserting" each element into a sorted prefix where it is placed at the correct location to maintain the sorted order. This corresponds to the well-known insertion sort.

Clearly, this reduction order has no parallelism except within each merge, and therefore the span is

$$S(\texttt{reduceSort}\ x) \ \leq \ \sum_{i=1}^{n-1} c \cdot \log(1+i) \ \in \ O(n \log n)$$

since merge on sequences of lengths n_1 and n_2 has $O(\log(n_1+n_2))$ span.

Notice that in the reduction order above, the reduction tree was extremely unbalanced. Would the costs change if the merges are balanced? For ease of exposition, let's suppose that the length of our sequence is a power of 2, i.e., $|x| = 2^k$. Now we lay on top the input sequence a "full" binary tree[2] with 2^k leaves and merge according to the tree structure. As an example, the merge sequence for $|x| = 2^3$ is shown below.



[2]This is simply a binary tree in which every node either has exactly 2 children or is a leaf, and all leaves are at the same depth.

Clearly using this balanced combining tree gives a smaller span than the imbalanced tree. But does it also reduce the work cost? At the bottom level where the leaves are, there are $n = |x|$ nodes with constant cost each (these were generated using a `map`). Stepping up one level, there are $n/2$ nodes, each corresponding to a `merge` call, each costing $c(1+1)$. In general, at level i (with $i = 0$ at the root), we have 2^i nodes where each node is a merge with input two sequences of length $n/2^{i+1}$. Therefore, the work of `reduceSort` using this reduction order is the familiar sum

$$\begin{aligned} W(\texttt{reduceSort}\ x) &\leq \sum_{i=0}^{\log n} 2^i \cdot c\left(\frac{n}{2^{i+1}} + \frac{n}{2^{i+1}}\right) \\ &= \sum_{i=0}^{\log n} 2^i \cdot c\left(\frac{n}{2^i}\right) \end{aligned}$$

This sum, as you have seen before, evaluates to $O(n \log n)$. In fact, this algorithm is essentially the merge sort algorithm. Therefore we see that `mergeSort` and `insertionSort` are just two special cases of `reduceSort` that use different reduction orders.

As discussed earlier, we need to precisely define the reduction order to determine the result of `reduce` when applied to a non-associative combining function. These two examples illustrate, however, that even when the combining function is associative, the particular reduction order we choose can lead to drastically different costs in both work and span. When combining function has $O(1)$ work, using a balanced reduction tree improves the span from $O(n)$ to $O(\log n)$ but does not change the work. But with $\texttt{merge}_<$ as the combining function, we see that the balanced reduction tree also improves the work from $O(n^2)$ to $O(n \log n)$.

In general, how would we go about defining the cost of `reduce` with higher order functions. Given a reduction tree, we'll first define $\mathcal{R}(\texttt{reduce}\ f\ \mathbb{I}\ S)$ as

$$\mathcal{R}(\texttt{reduce}\ f\ \mathbb{I}\ S) = \left\{\text{all function applications } f(a,b) \text{ in the reduction tree}\right\}.$$

Following this definition, we can state the cost of `reduce` as follows:

$$\begin{aligned} W(\texttt{reduce}\ f\ \mathbb{I}\ S) &= O\left(n + \sum_{f(a,b)\in\mathcal{R}(f\ \mathbb{I}\ S)} W(f(a,b))\right) \\ S(\texttt{reduce}\ f\ \mathbb{I}\ S) &= O\left(\log n \max_{f(a,b)\in\mathcal{R}(f\ \mathbb{I}\ S)} S(f(a,b))\right) \end{aligned}$$

The work bound is simply the total work performed, which we obtain by summing across all combine operations. The span bound is more interesting. The $\log n$ term expresses the fact that the tree is at most $O(\log n)$ deep. Since each node in the tree has span at most $\max_{f(a,b)} S(f(a,b)$, any root-to-leaf path, including the "critical path," has at most $O(\log n \max_{f(a,b)} S(f(a,b))$ span.

This can be used, for example, to prove the following lemma:

Lemma 2.1. *For any combine function $f : \alpha \times \alpha \to \alpha$ and a monotone size measure $s : \alpha \to \mathbb{R}_+$, if for any x, y,*

1. *$s(f(x,y)) \leq s(x) + s(y)$ and*
2. *$W(f(x,y)) \leq c_f\ (s(x) + s(y))$ for some universal constant c_f depending on the function f,*

then

$$W(\texttt{reduce}\, f\ \mathbb{I}\ S) = O\left(\log|S| \sum_{x \in S}(1 + s(x))\right).$$

Applying this lemma to the merge sort example, we have

$$W(\texttt{reduce}\ \texttt{merge}_{<}\ \langle\,\rangle\ \langle\langle a\rangle : a \in A\rangle) = O(|A| \log |A|)$$

Lecture 7 — Collect, Sets, and Tables

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — September 18, 2012

Material in this lecture:

- The collect operation.
- Sets and Tables

Challenge: Given sequences of integers A and B of length n, and an initial value X_0, calculate $X_{i+1} = A_i \cdot X_i + B_i$ for all $0 \leq i < n$. You need to do it in $O(n)$ work and $O(\log n)$ span.

1 Collect

In many applications it is useful to collect all items that share a common key. For example we might want to collect students by course, documents by word, or sales by date. More specifically let's say we had a sequence of pairs each consisting of a student's name and a course they are taking, such as

$$\begin{aligned}\textbf{val}\ \texttt{Data} = \langle &(\textit{“jack sprat”}, \textit{“15-210”}),\\ &(\textit{“jack sprat”}, \textit{“15-213”}),\\ &(\textit{“mary contrary”}, \textit{“15-210”}),\\ &(\textit{“mary contrary”}, \textit{“15-251”}),\\ &(\textit{“mary contrary”}, \textit{“15-213”}),\\ &(\textit{“peter piper”}, \textit{“15-150”}),\\ &(\textit{“peter piper”}, \textit{“15-251”}),\\ &\ldots\rangle\end{aligned}$$

and we want to collect all entries by course number so we have a list of everyone taking each course. Collecting values together based on a key is very common in processing databases. In relational database languages such as SQL it is referred to as "Group by". More generally it has many applications and furthermore it is naturally parallel.

We will use the function `collect` for this purpose, and it is part of the sequence library. Its interface is:

$$\texttt{collect} : (\alpha \times \alpha \to \textit{order}) \to (\alpha \times \beta)\ \textit{seq} \to (\alpha \times \beta\ \textit{seq})\ \textit{seq}$$

The first argument is a function for comparing keys of type α, and must define a total order over the keys. The second argument is a sequence of key-value pairs. The `collect` function collects all values that share the same key together into a sequence. If we wanted to collect the entries of `Data` given above by course number we could do the following:

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

val `collectStrings` = `collect String.compare`

val `rosters` = `collectStrings`$(\langle (n,c) : (c,n) \in \texttt{Data} \rangle)$

This would give something like:

val `rosters` = $\langle$(*"15-150"*, $\langle$ *"peter piper"*, ... $\rangle$)
(*"15-210"*, $\langle$ *"jack sprat"*, *"mary contrary"*, ... $\rangle$)
(*"15-213"*, $\langle$ *"jack sprat"*, ... $\rangle$)
(*"15-251"*, $\langle$ *"mary contrary"*, *"peter piper"* $\rangle$)
...$\rangle$

We use a map $(\langle (n,c) : (c,n) \in \texttt{Data} \rangle)$ to put the course number in the first position in the tuple since that is the position used to collect on.

Collect can be implemented by sorting the keys based on the given comparison function, and then partitioning the resulting sequence. In particular, the sort will move all the equal keys so they are adjacent. A partition function can then identify the positions where the keys change values, and pull out all pairs between each change. Doing this partitioning can be done relatively easily by filtering out the indices where the value changes. The dominant cost of `collect` is therefore the cost of the sort. Assuming the comparison has complexity bounded above by W_c work and S_c span then the costs of `collect` are $O(W_c n \log n)$ work and $O(S_c \log^2 n)$ span. It is also possible to implement a version of collect that runs in linear work using hashing. But hashing would require that a hash function is also supplied and would not return the keys in sorted order. Later in the lecture we discuss tables which also have a `collect` function. However tables are specialized to the key type and therefore neither a comparison nor a hash function need to be passed as arguments.

1.1 Using Collect in Map Reduce

Some of you have probably heard of the map-reduce paradigm first developed by Google for programming certain data intensive parallel tasks. It is now widely used within Google as well as by many others to process large data sets on large clusters of machines—sometimes up to tens of thousands of machines in large data centers. The map-reduce paradigm is often used to analyze various statistical data over very large collections of documents, or over large log files that track the activity at web sites. Outside Google the most widely used implementation is the Apache Hadoop implementation, which has a free license (you can install it at home). The paradigm is different from the `mapReduce` function you might have seen in 15-150 which just involved a map then a reduce. The map-reduce paradigm actually involves a map followed by a collect followed by a bunch of reduces, and therefore might be better called the map-collect-reduces.

The map-reduce paradigm processes a collection of documents based on a map function $f_{\mathtt{m}}$ and a reduce function $f_{\mathtt{r}}$ supplied by the user. The $f_{\mathtt{m}}$ function must take a document as input and generate a sequence of key-value pairs as output. This function is mapped over all the documents. All key-value pairs across all documents are then collected based on the key. Finally the $f_{\mathtt{r}}$ function is applied to each of the keys along with its sequence of associated values to reduce to a single value.

In ML the types for map function $f_{\mathtt{m}}$ and reduce function $f_{\mathtt{r}}$ are the following:

$$\begin{aligned} f_{\mathtt{m}} &: (\mathit{document} \to (\mathit{key} \times \alpha)\ \mathit{seq}) \\ f_{\mathtt{r}} &: (\mathit{key} \times (\alpha\ \mathit{seq}) \to \beta) \end{aligned}$$

In most implementations of map-reduce the document is a string (just the contents of a file) and the key is also a string. Typically the α and β types are limited to certain types. Also, in most implementations both the $f_{\mathtt{m}}$ and $f_{\mathtt{r}}$ functions are sequential functions. Parallelism comes about since the $f_{\mathtt{m}}$ function is mapped over the documents in parallel, and the $f_{\mathtt{r}}$ function is mapped over the keys with associated values in parallel.

In ML map reduce can be implemented as follows

```
1  fun mapCollectReduce fm fr docs =
2    let
3      val pairs = flatten ⟨ fm(s) : s ∈ docs ⟩
4      val groups = collect String.compare pairs
5    in
6      ⟨ fr(g) : g ∈ groups ⟩
7    end
```

The function `flatten` simply flattens a nested sequence into a flat sequence, e.g.:

$$\begin{aligned} &\texttt{flatten}\,\langle\langle a,b,c\rangle,\langle d,e\rangle\rangle \\ \Rightarrow\ &\langle a,b,c,d,e\rangle \end{aligned}$$

As an example application of the paradigm, suppose we have a collection of documents, and we want to know how often every word appears across all documents. This can be done with the following $f_{\mathtt{m}}$ and $f_{\mathtt{r}}$ functions.

fun $f_{\mathtt{m}}(\mathtt{doc}) = \langle (w,1) : \mathtt{tokens\ doc} \rangle$

fun $f_{\mathtt{r}}(w,s) = (w, \mathtt{reduce\ +\ 0}\ s)$

Here `tokens` is a function that takes a string and breaks it into tokens by removing whitespace and returning a sequence of strings between whitespace.

Now we can apply `mapCollectReduce` to generate a `countWords` function, and apply this to an example case.

```
val countWords = mapCollectReduce fm fr

countWords ⟨"this is a document",
            "this is is another document",
            "a last document"⟩
```

$\Rightarrow \langle (\text{“a”}, 2), (\text{“another”}, 1), (\text{“document”}, 3), (\text{“is”}, 3), (\text{“last”}, 1), (\text{“this”}, 2) \rangle$

2 An Abstract Data Type for Sets

Sets undoubtedly play an important role in mathematics and are often needed in the implementation of various algorithms. Whereas a sequence is an ordered collection, a set is its *unordered* counterpart.

Most programming languages either support sets directly (e.g., Python and Ruby) or have libraries that support them (e.g., in the C++ STL library and Java collections framework). They sometimes have more than one implementation of sets. For example, Java has sets based on hash tables and balanced trees. Unsurprisingly, the set interface in different libraries and languages differ in subtle ways. So, when using one of these interfaces you should always read the documentation carefully.

We now define an abstract data type for sets. The definition follows the mathematical definition of sets from set theory and is purely functional. In particular, when updating a set (e.g. with insert or delete) it returns a new set rather than modifying the old set.

Definition 2.1. For a universe of elements $\mathbb{U}$ (e.g. the integers or strings), the SET abstract data type is a type $\mathbb{S}$ representing the power set of $\mathbb{U}$ (i.e., all subsets of $\mathbb{U}$) along with the following functions:

`empty`	:	$\mathbb{S}$	=	$\emptyset$
`size`(S)	:	$\mathbb{S} \to \mathbb{Z}_{\geq 0}$	=	$\lvert S \rvert$
`singleton`(e)	:	$\mathbb{U} \to \mathbb{S}$	=	$\{e\}$
`filter`(f, S)	:	$((\mathbb{U} \to \{\mathtt{T}, \mathtt{F}\}) \times \mathbb{S}) \to \mathbb{S}$	=	$\{s \in S \mid f(s)\}$
`find`(S, e)	:	$\mathbb{S} \times \mathbb{U} \to \{\mathtt{T}, \mathtt{F}\}$	=	$\lvert\{s \in S \mid s = e\}\rvert = 1$
`insert`(S, e)	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{S}$	=	$S \cup \{e\}$
`delete`(S, e)	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{S}$	=	$S \setminus \{e\}$
`intersection`(S_1, S_2)	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	=	$S_1 \cap S_2$
`union`(S_1, S_2)	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	=	$S_1 \cup S_2$
`difference`(S_1, S_2)	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	=	$S_1 \setminus S_2$

where $\mathbb{Z}_{\geq 0}$ are the non-negative integers.

We write this definition to be generic and not specific to Standard ML. In our library, the type $\mathbb{S}$ is called `set` and the type $\mathbb{U}$ is called `key`, the arguments are not necessarily in the same order, and some of the functions are curried. For example, the interface for find is `find` : `set` → `key` → `bool`. Please refer to the documents for details. In the pseudocode, we will give in class and in the notes we will use standard set notation as in the right hand column of the table above.

A Note about `map`. You may notice that the interface does not contain a `map` function. If we try to generalize the notion of `map` from sequences, a `map` function does not make sense in the context of a set: if we interpret map to take in a collection, apply some function to each element and return a collection of the same structure. Consider a function that always returns 0. Mapping this over a set would return all zeros, which would then be collapsed into a singleton set, containing exactly 0. Therefore, such a `map` would allow reducing the set of arbitrary size to a singleton, which doesn't match the `map` paradigm (which traditionally preserves the structure and size).

2.1 Cost Model

So far, we have laid out a semantic interface, but before we can put it to use, we need to worry about the cost specification. The most common efficient ways to implement sets are either using hashing or balanced trees. They have various tradeoffs in cost. For simplicity, we'll consider a cost model based on a balanced-tree implementation. We will cover how to implement these set operations when we talk about balanced trees later in the course. For now, a good intuition to have is that we use a comparison function to keep the elements in sorted order in a balanced tree. Since this requires comparisons inside the various set operations, the cost of these comparisons affects the work and span. For this, we'll assume that compare has C_w work and C_s span.

We have the following cost specification:

	Work	*Span*
`size`(S) `singleton`(e)	$O(1)$	$O(1)$
`filter`(f,S)	$O\left(\sum_{e\in S} W(f(e))\right)$	$O\left(\log\lvert S\rvert + \max_{e\in S} S(f(e))\right)$
`find`(S,e) `insert`(S,e) `delete`(S,e)	$O(C_w \cdot \log\lvert S\rvert)$	$O(C_s \cdot \log\lvert S\rvert)$
`intersection`(S_1,S_2) `union`(S_1,S_2) `difference`(S_1,S_2)	$O\big(C_w \cdot m \cdot \log(1+\frac{n}{m})\big)$	$O\big(C_s \cdot \log(n+m)\big)$

where $n = \max(|S_1|,|S_2|)$ and $m = \min(|S_1|,|S_2|)$.

The work bounds for `intersection`, `union`, and `difference` deserve further discussion—at a glance they might seem a bit funky. These turn out to be both the theoretical upper and lower bounds for any comparison-based implementation of sets. We will get to this later.

Notice that when the two sets have the same length ($n = m$), the work is simply

$$O(C_w \cdot m \cdot \log(1+1)) = O(C_w \cdot n).$$

This should not be surprising, because it corresponds to the cost of merging two approximately equal length sequences (effectively what these operations have to do).

Moreover, you should also observe that in the case that one of the sets is a singleton, then the work is $O(\log n)$.

Buy in Bulk and Save. On inspection, the functions `intersection`, `union`, and `difference` are simply the "parallel" counterparts of the functions `find`, `insert`, and `delete`, to wit:

- `intersection` — search for multiple elements instead of one.
- `union` — insert multiple elements.
- `difference` — delete multiple elements.

In fact, it is easy to implement `find`, `insert`, and `delete` in terms of the others.

$$\begin{aligned} \texttt{find}(S,e) &= \texttt{size}(\texttt{intersection}(S,\texttt{singleton}(e))) = 1 \\ \texttt{insert}(S,e) &= \texttt{union}(S,\texttt{singleton}(e)) \\ \texttt{delete}(S,e) &= \texttt{difference}(S,\texttt{singleton}(e)) \end{aligned}$$

Since `intersection`, `union`, and `difference` can operate on multiple elements they are well suited for parallelism, while `find`, `insert`, and `delete` have no parallelism. Consequently, in designing parallel algorithms it is good to think about how to use `intersection`, `union`, and `difference` instead of `find`, `insert`, and `delete` if possible. For example, one way to convert a sequence to a set would be to insert the elements one by one, which can be coded as

```
fun fromSeq(S) = Seq.iter (fn (S',e) => Set.insert e S') Set.empty S
```

However, the above is sequential. To do it in parallel we could instead do

```
fun fromSeq(S) = Seq.reduce Set.union Set.empty (Seq.map Set.singleton S)
```

Exercise 1. *What is the work and span of the first version of* `fromSeq`.

Exercise 2. *Show that on a sequence of length n the second version of* `fromSeq` *does $O(C_w n \log n)$ work and $O(\log^2 n)$ span.*

3 Tables: Associating Each Element With A Value

Suppose we want to extend sets so that each element is associated with a payload. A table is an abstract data type that stores for each key data associated with it. The table ADT supplies operations for finding the value associated with a key, for inserting new key-value pairs, and for deleting keys and their associated value. Tables are also called dictionaries, associative arrays, maps, mappings, and functions (in set theory). Given our focus on parallelism, the interface we will discuss also supplies "parallel" operations that allow the user to insert multiple key-value pairs, to delete multiple keys, and to find the values associated with multiple keys.

In this class, the notation we are going to be using is

$$\{(k_1 \mapsto v_1), (k_2 \mapsto v_2), \ldots, (k_n \mapsto v_n)\},$$

where we have *keys* and *values*—and each key k_i is associated with the value v_i. Mathematically, a table is simply a set of pairs and can therefore be written as a set of ordered pairs $\{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}$. Our notation choice is largely to better identify when tables are being used.

As with sets, tables are commonly used in many applications. Most languages have tables either built in (e.g. dictionaries in Python, Perl, and Ruby), or have libraries to support them (e.g. map in the C++ STL library and the Java collections framework). We note that the interfaces for these languages and libraries have common features but typically differ in some important ways, so be

warned. Most do not support the "parallel" operations we discuss. Again, here we will define tables mathematically in terms of set theory before committing to a particular language.

Formally, a table is set of key-value pairs where each key appears only once in the set. Such sets are called *functions* in set theory since they map each key to a single value. We will avoid this terminology so that we don't confuse it with functions in a programming language. However, note that the `(find T)` in the interface is precisely the "function" defined by the table `T`. In fact it is a *partial function* since the table might not contain all keys and therefore the function might not be defined on all inputs. Here is the definition of a table.

Definition 3.1. For a universe of keys $\mathbb{K}$, and a universe of values $\mathbb{V}$, the TABLE abstract data type is a type $\mathbb{T}$ representing the power set of $\mathbb{K} \times \mathbb{V}$ restricted so that each key appears at most once (i.e., any set of key-value pairs where a key appears just once) along with the following functions:

$$
\begin{array}{lcll}
\texttt{empty} & : & \mathbb{T} & = \emptyset \\
\texttt{size}(T) & : & \mathbb{T} \to \mathbb{Z}_{\geq 0} & = |T| \\
\texttt{singleton}(k,v) & : & \mathbb{K} \times \mathbb{V} \to \mathbb{T} & = \{(k,v)\} \\
\texttt{filter}(f,T) & : & ((\mathbb{V} \to \{\mathtt{T},\mathtt{F}\}) \times \mathbb{T}) \to \mathbb{T} & = \{(k,v) \in T \,|\, f(v)\} \\
\texttt{map}(f,T) & : & ((\mathbb{K} \times \mathbb{V} \to \mathbb{V}) \times \mathbb{T}) \to \mathbb{T} & = \{(k, f(k,v)) | ((k,v) \in T)\} \\
\texttt{insert}(f,T,(k,v)) & : & (\mathbb{V} \times \mathbb{V} \to \mathbb{V}) \times \mathbb{T} \times (\mathbb{K} \times \mathbb{V}) \to \mathbb{T} & = \\
 & & \multicolumn{2}{l}{\quad \forall k \in \mathbb{K}, \begin{cases} (k, f(v,v')) \} & (k,v') \in T \\ (k,v) & (k,v') \notin T \end{cases}} \\
\texttt{delete}(T,k)) & : & \mathbb{T} \times \mathbb{K} \to \mathbb{T} & = \{(k',v) \in T | k \neq k'\} \\
\texttt{find}(T,k) & : & \mathbb{T} \times \mathbb{K} \to (\mathbb{V} \cup \bot) & = \begin{cases} v & (k,v) \in T \\ \bot & otherwise \end{cases} \\
\texttt{merge}(f,T_1,T_2) & : & (\mathbb{V} \times \mathbb{V} \to \mathbb{V}) \times \mathbb{T} \times \mathbb{T} \to \mathbb{T} & = \\
 & & \multicolumn{2}{l}{\quad \forall k \in \mathbb{K}, \begin{cases} (k, f(v_1,v_2)) & (k,v_1) \in T_1 \wedge (k,v_2) \in T_2 \\ (k,v_1) & (k,v_1) \in T_1 \\ (k,v_2) & (k,v_2) \in T_2 \end{cases}} \\
\texttt{extract}(T,S) & : & \mathbb{T} \times \mathbb{S} \to \mathbb{T} & = \{(k,v) \in T | k \in S\} \\
\texttt{erase}(T,S) & : & \mathbb{T} \times \mathbb{S} \to \mathbb{T} & = \{(k,v) \in T | k \notin S\}
\end{array}
$$

where $\mathbb{S}$ is the power set of K (i.e., any set of keys) and $\mathbb{Z}_{\geq 0}$ are the non-negative integers.

Distinct from sets, the `find` function does not return a Boolean, but instead it returns the value associated with the key k. As it may not find the key in the table, its result may be bottom ($\bot$). For this reason, in the Table library, the interface for find is `find : 'a table → key → 'a option`, where `'a` is the type of the values.

Unlike sets, when we insert an element, we can't simply ignore that element if it is already present—their values might be different. For this reason, the `insert` function takes a function $f : \mathbb{V} \times \mathbb{V} \to \mathbb{V}$ as an argument. The purpose of f is to specify what to do if the key being inserted already exists in the table; f is applied to the two values. This function might simply return either its first or second argument, or it can be used, for example, to add the new value to the old one. The parallel counterpart of `find` is the `merge` function, which takes a similar function since it also has to consider the case that an element appears in both tables.

We also introduce new pseudocode notation for `map` and `filter` on tables:

$$\{(k \mapsto f(v) \,|\, (k \mapsto v) \in T\}$$

is equivalent to $\texttt{map}(f, T)$ and

$$\{(k \mapsto v) \in T \mid p(v)\}$$

is equivalent to $\texttt{filter}(p, T)$.

The costs of the table operations are very similar to sets.

	Work	*Span*
$\texttt{size}(T)$ $\texttt{singleton}(k, v)$	$O(1)$	$O(1)$
$\texttt{filter}(f, T)$	$O\left(\sum_{(k,v)\in T} W(f(v))\right)$	$O\left(\log\lvert T\rvert + \max_{(k,v)\in T} S(f(v))\right)$
$\texttt{map}(f, T)$	$O\left(\sum_{(k,v)\in T} W(f(k,v))\right)$	$O\left(\max_{(k,v)\in T} S(f(k,v))\right)$
$\texttt{find}(S, k)$ $\texttt{insert}(T, (k, v))$ $\texttt{delete}(T, k)$	$O(C_w \log\lvert T\rvert)$	$O(C_s \log\lvert T\rvert)$
$\texttt{extract}(T_1, T_2)$ $\texttt{merge}(T_1, T_2)$ $\texttt{erase}(T_1, T_2)$	$O\left(C_w m \log(1 + \frac{n}{m})\right)$	$O\left(C_s \log(n+m)\right)$

where $n = \max(|T_1|, |T_2|)$ and $m = \min(|T_1|, |T_2|)$.

As with sets there is a symmetry between the three operations `extract`, `merge`, and `erase`, and the three operations `find`, `insert`, and `delete`, respectively, where the prior three are effectively "parallel" versions of the earlier three. The `extract` operation can be used to find a set of values in a table, returning just the table entries corresponding to elements in the set. The `merge` operation can add multiple values to a table in parallel by merging two tables. The `erase` operation can delete multiple values from a table in parallel.

We note that, in the SML Table library we supply, the functions are polymorphic (accept any type) over the values but not the keys. In particular the signature starts as:

```
signature TABLE =
sig
  type 'a table
  type 'a t = 'a table
  structure Key : EQKEY
  type key = Key.t
  structure Seq : SEQUENCE
  type a seq = 'a Seq.seq
  type set = unit table
  ...
  val find : 'a table -> key -> 'a option
  ...
```

The `'a` in `'a table` refers to the type of the value. The key type is fixed to be `key`. Therefore there

are separate table structures for different keys (e.g. `IntTable`, `StringTable`). The reason to do this is because all the operations depend on the key type since keys need to be compared for a tree implementation, or hashed for a hash table implementation. Also note that the signature defines `set` to be a `unit table`. Indeed a set is just a special case of a table where there are no values.

In the SML Table library, we supply a `collect` operation that takes a sequence of key-value pairs and produces a table that maps every key in S to all the values associated with it in S, gathering all the values with the same key together in a sequence. This is equivalent to using a sequence collect followed by a `Table.fromSeq`. Alternatively, it can be implemented as

```
1  fun collect(S) =
2  let
3    val S' = ⟨ {k ↦ ⟨ v ⟩} : (k, v) ∈ S ⟩
4  in
5    Seq.reduce (Table.merge Seq.append) {} S'
6  end
```

Exercise 3. *Figure out what this code does.*

Lecture 8 — Sets and Tables II

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — September 20, 2012

Today:

- Finish Tables (from last class)
- Example of Tables for indexing the web
- Single Threaded Sequences

1 Example: Bingle® It

Here we consider an application of sets and tables to searching a corpus of documents. In particular lets say one night, late, while avoiding doing your 210 homework you come up with a great idea: provide a service that indexes all the pages on the web so that people can search them by keywords. You figure a good name for such a service would be **Bingle**®. The idea is to support a function that makes the index from the documents, which is run just once (or once in a while) so it can take a while. On the other hand you want queries to the database to be fast since people will be running them all the time. The type of queries you want to support are logical queries on words involving `And`, `Or`, and `AndNot`. For example a query might look like

> "CMU" `And` "fun" `And` ("courses" `Or` "clubs")

and it would return a list of web pages that match the query (*i.e.*, contain the words "CMU", "fun" and either "courses" or "clubs"). This list would include the 15-210 home page, of course.

OK, so maybe this idea has been thought of before. Indeed these kinds of searchable indexes date back to the 1970s with systems such as Lexis for searching law documents. Today, beyond web searches, searchable indices are an integral part of most mailers and operating systems. The different indices support somewhat different types of queries. For example, by default Google supports queries with `And` and `adjacent to` but with their advanced search you can search with `Or`, `AndNot` as well as other types of searches.

Let's imagine you want to support the following interface

```
signature INDEX = sig
  type word = string
  type docId = string
  type 'a seq
  type index
  type docList

  val makeIndex : (docId * string) seq -> index
```

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

```
  val find : index -> word -> docList
  val And : docList * docList -> docList
  val AndNot : docList * docList -> docList
  val Or : docList * docList -> docList
  val size : docList -> int
  val toSeq : docList -> docId seq
end
```

The input to `makeIndex` is a sequence of pairs each consisting of a document identifier (e.g. the URL) and the contents of the document as a single text string. So for example we might want to index recent tweets, that might include the following "documents":

$T = \langle$ ("jack", "chess club was fun"),
("mary", "I had a fun time in 210 class today"),
("nick", "food at the cafeteria sucks"),
("sue", "In 217 class today I had fun reading my email"),
("peter", "I had fun at nick's party"),
("john", "tiddlywinks club was no fun, but more fun than 218"),
$\rangle$

where the identifiers are the names, and the contents is the tweet.

The interface can be used to make an index of these tweets:

val $f = (\texttt{find}\ (\texttt{makeIndex}(T))) : \texttt{word} \to \texttt{doclist}$

In addition to making the index, this partially applies `find` on it. We can then use this index for various queries. For example:

```
 toSeq(And(f "fun", Or(f "class", f "club")))
⇒ ⟨"jack", "mary", "sue", "john"⟩
```

returns all the documents (tweets) that contain "fun" and either "class" or "club", and

```
 size(AndNot(f "fun", f "tiddlywinks"))
⇒ 4
```

returns the number of documents that contain "fun" and not "tiddlywinks".

We can implement this interface very easily using sets and tables. The `makeIndex` function can be implemented as follows.

```
1  fun makeIndex(docs) =
2  let
3    fun tagWords(id,str) = ⟨(w,id) : w ∈ tokens(str)⟩
4    val Pairs = flatten ⟨ tagWords(d) : d ∈ docs ⟩
5    val Words = Table.collect(Pairs)
6  in
7    {w ↦ Set.fromSeq(d) : (w ↦ d) ∈ Words}
8  end
```

The `tagWords` function takes a document as a pair consisting of the document identifier and contents, breaks the string into tokens (words) and tags each token with the identifier returning a sequence of these pairs. For example, on the document

$$\begin{aligned} & \texttt{tagWords}(\textit{“jack”}, \textit{“chess club was fun”}) \\ \Rightarrow \quad & \langle\, (\textit{“chess”},\textit{“jack”}), (\textit{“club”}, \textit{“jack”}), (\textit{“was”}, \textit{“jack”}), (\textit{“fun”}, \textit{“jack”}) \,\rangle \end{aligned}$$

The function `tagWords` is then applied to all document and the result flattened so it is a single sequence. In our example the result would start as:

$$\begin{aligned} \texttt{Pairs} \;=\; & \langle (\textit{“chess”},\textit{“jack”}), (\textit{“club”}, \textit{“jack”}), (\textit{“was”}, \textit{“jack”}), \\ & (\textit{“fun”}, \textit{“jack”}), (\textit{“I”}, \textit{“mary”}), (\textit{“had”}, \textit{“mary”}), (\textit{“fun”}, \textit{“mary”}), \ldots \end{aligned}$$

The `Table.collect` then collects the entries by word creating a sequence of matching documents. In our example it would start:

$$\begin{aligned} \texttt{Words} \;=\; & \{(\textit{“a”} \mapsto \langle\, \textit{“mary”} \,\rangle), \\ & (\textit{“at”} \mapsto \langle\, \textit{“mary”}, \textit{“peter”} \,\rangle), \\ & \ldots \\ & (\textit{“fun”} \mapsto \langle\, \textit{“jack”}, \textit{“mary”}, \textit{“sue”}, \textit{“peter”}, \textit{“john”} \,\rangle), \\ & \ldots \end{aligned}$$

Finally, for each word the sequences of document identifiers is converted to a set. Note the notation that is used to express a map over the elements of a table.

Assuming that all tokens have a length upper bounded by a constant, the cost of `makeIndex` is dominated by the collect, which is basically a sort. The work is therefore $O(n \log n)$ and the span is $O(\log^2 n)$, assuming the words have constant length. The rest of the interface can be implemented as follows:

fun `find` $T\ v = \texttt{Table.find}\ T\ v$

fun `And`$(s_1, s_2) = s_1 \cap s_2$

fun `Or`$(s_1, s_2) = s_1 \cup s_2$

fun `AndNot`$(s_1, s_2) = s_1 \setminus s_2$

fun `size`$(s) = |s|$

fun `toSeq`$(s) = \texttt{Set.toSeq}(s)$

Note that if we do a `size(f "red")` the cost is only $O(\log n)$ work and span. It just involves a search and then a length.

If we do `And(f "fun", Or(f "courses", f "classes"))` the worst case work and span are at most:

$$\begin{aligned} W &= O(|f(\text{"}\textit{fun}\text{"})| + |f(\text{"}\textit{courses}\text{"})| + |f(\text{"}\textit{classes}\text{"})|) \\ S &= O(\log |\textit{index}|) \end{aligned}$$

The sum of sizes is to account for the cost of the `And` and `Or`. The actual cost could be significantly less especially if one of the sets is very small.

2 Single-Threaded Array Sequences

In this course we will be using purely functional code because it is safe for parallelism, easier to reason about formally, and because its cool. For many algorithms using the purely functional version makes no difference in the asymptotic work bounds—for example quickSort and mergeSort use $\Theta(n \log n)$ work (expected case for quickSort) whether purely functional or imperative. However, in some cases purely functional implementations lead to up to a $O(\log n)$ factor of additional work. To avoid this we will slightly cheat in this class and allow for benign "effect" under the hood in exactly one ADT, described in this section. These effects do not affect the observable values (you can't observe them by looking at results), but they do affect cost analysis—and if you sneak a peak at our implementation, you will see some side effects.

The issue has to do with updating positions in a sequence. In an imperative language updating a single position can be done in "constant time". In the functional setting we are not allowed to change the existing sequence, everything is persistent. This means that for a sequence of length n an update can either be done in $O(n)$ work with an arraySequence (the whole sequence has to be copied before the update) or $O(\log n)$ work with a treeSequence (an update involves traversing the path of a tree to a leaf). In fact you might have noticed that our sequence interface does not even supply a function for updating a single position. The reason is both to discourage sequential computation, but also because it would be expensive.

Consider a function `update` (i, v) S that updates sequence S at location i with value v returning the new sequence. This function would have cost $O(|S|)$ in the arraySequence cost specification. Someone might be tempted to write a sequential loop using this function. For example for a function $f : \alpha -> \alpha$, a `map` function can be implemented as follows:

```
fun map f S =
  iter (fn ((i,S'),v) => (i+1, update (i, f(v)) S'))
       (0,S)
       S
```

This code iterates over S with i going from 0 to $n-1$ and at each position i updates the value S_i with $f(S_i)$. The problem with this code is that even if f has constant work, with an `arraySequence` this will do $O(|S|^2)$ total work since every update will do $O(|S|)$ work. By using a `treeSequence` implementation we can reduce the work to $O(|S| \log |S|)$ but that is still a factor of $O(\log |S|)$ off of what we would like.

In the class we sometimes do need to update either a single element or a small number of elements of a sequence. We therefore introduce an ADT we refer to as a *Single Threaded Sequence* (`stseq`). Although the interface for this ADT is quite straightforward, the cost specification is somewhat tricky. To define the cost specification we need to distinguish between the latest "copy" of an instance of an `stseq`, and earlier copies. Basically whenever we update a sequence we create a new "copy", and the old "copy" is still around due to the persistence in functional languages. The cost specification is going to give different costs for updating the latest copy and old copies. Here we will only define the cost for updating the latest copy, since this is the only way we will be using an `stseq`. The interface and costs is as follows:

	Work	Span
`fromSeq(S) : α seq → α stseq` Converts from a regular sequence to a stseq.	$O(\|S\|)$	$O(1)$
`toSeq(ST) : α stseq → α seq` Converts from a stseq to a regular sequence.	$O(\|S\|)$	$O(1)$
`nth ST` i `: α stseq → int → α` Returns the i^{th} element of ST. Same as for seq.	$O(1)$	$O(1)$
`update` (i,v) S `: (int × α) → α stseq → α stseq` Replaces the i^{th} element of S with v.	$O(1)$	$O(1)$
`inject` I S `: (int × α) seq → α stseq → α stseq` For each $(i,v) \in I$ replaces the i^{th} element of S with v.	$O(\|I\|)$	$O(1)$

An `stseq` is basically a sequence but with very little functionality. Other than converting to and from sequences, the only functions are to read from a position of the sequence (`nth`), update a position of the sequence (`update`) or update multiple positions in the sequence (`inject`). To use other functions from the sequence library, one needs to covert an `stseq` back to a sequence (using `toSeq`).

In the cost specification the work for both `nth` and `update` is $O(1)$, which is about as good as we can get. Again, however, this is only when S is the latest version of a sequence (i.e. noone else has updated it). The work for `inject` is proportional to the number of updates. It can be viewed as a parallel version of `update`.

Now with an `stseq` we can implement our map as follows:

```
1  fun map f S = let
2    val S' = StSeq.fromSeq(S)
3    val R = iter (fn ((i,S''),v) ⇒ (i+1, StSeq.update (i,f(v)) S''))
4                 (0,S')
5                 S
6  in
7    StSeq.toSeq(R)
8  end
```

This implementation first converts the input sequence to an `stseq`, then updates each element of the `stseq`, and finally converts back to a sequence. Since each update takes constant work, and assuming the function f takes constant work, the overall work is $O(n)$. The span is also $O(n)$ since `iter` is completely sequential. This is therefore not a good way to implement `map` but it does illustrate that the work of multiple updates can be reduced from $O(n^2)$ on array sequences or $O(n \log n)$ on tree sequences to $O(n)$ using an `stseq`.

Implementing Single Threaded Sequences. You might be curious about how single threaded sequences can be implemented so they act purely functional but match the cost specification. Here we will just briefly outline the idea.

The trick is to keep two copies of the sequence (the original and the current copy) and additionally to keep a "change log". The change log is a linked list storing all the updates made to the original sequence. When converting from a sequence to an `stseq` the sequence is copied to make a second identical copy (the current copy), and an empty change log is created. A different representation is now used for the latest version and old versions of an `stseq`. In the latest version we keep both copies (original and current) as well as the change log. In the old versions we only keep the original copy and the change log. Lets consider what is needed to update either the current or an old version. To update the current version we modify the current copy in place with a side effect (non functionally), and add the change to the change log. We also take the previous version and mark it as an old version removing its current copy. When updating an old version we just add the update to its change log. Updating the current version requires side effects since it needs to update the current copy in place, and also has to modify the old version to mark it as old and remove its current copy.

Either updating the current version or an old version takes constant work. The problem is the cost of `nth`. When operating on the current version we can just look up the value in the current copy, which is up to date. When operating on an old version, however, we have to go back to the original copy and then check all the changes in the change log to see if any have modified the location we are asking about. This can be expensive. This is why updating and reading the current version is cheap ($O(1)$ work) while working with an old version is expensive.

In this course we will use `stseq`s for some graph algorithms, including breadth-first search (BFS) and depth-first search (DFS), and for hash tables.

3 SML Code

3.1 Indexes

```
functor TableIndex(Table : TABLE where type Key.t = string) : INDEX =
struct

structure Seq = Table.Seq
structure Set = Table.Set

type word = string
type docId = string
type 'a seq = 'a Seq.seq
type docList = Table.set
type index = docList Table.table

fun makeIndex docs =
let
  fun toWords str = Seq.tokens (fn c => not (Char.isAlphaNum c)) str

  fun tagWords(docId,str) = Seq.map (fn t => (t, docId)) (toWords str)

  (* generate all word-documentid pairs *)
  val allPairs = Seq.flatten (Seq.map tagWords docs)
```

```
  (* collect them by word *)
  val wordTable = Table.collect allPairs

in
  (* convert the sequence of documents for each word into a set
     which removes duplicates*)
  Table.map Set.fromSeq wordTable
end

fun find Idx w =
   case (Table.find Idx w) of
      NONE => Set.empty
    | SOME(s) => s

val And = Set.intersection
val AndNot = Set.difference
val Or = Set.union
val size = Set.size
val toSeq = Set.toSeq

end
```

Lecture 9 — Graphs Introduction

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — September 25, 2012

Material in this lecture:

- Graph Introduction
- Graph Representation
- Graph Search

1 Graphs

Graphs (sometimes referred to as networks) offer a way of expressing relationships between pairs of items, and are one of the most important abstractions in the study of algorithms. Graphs can be very important in modeling data. Furthermore a large number of problems can be reduced to known graph problems. A graph consists of a set of vertices with connections between them. Graphs can either be directed, with the directed edges (arcs) pointing from one vertex to another, or undirected, with the edges symmetrically connecting vertices. Graphs can have weights or other values associated with either the vertices and/or the edges.

Here we outline just some of the many applications of graphs.

16 Graph Applications.

1. *Social network graphs: to tweet or not to tweet.* Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. An example is the twitter graph of who follows whom. These can be used to determine how information flows, how topics become hot, how communities develop, or even who might be a good match for who, or is that whom.
2. *Transportation networks.* In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many map programs such as google maps, Bing maps and now Apple IOS 6 maps (well perhaps without the public transport) to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.
3. *Utility graphs.* The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them. Analyzing properties of these graphs is very important in understanding the reliability of such utilities under failure or attack, or in minimizing the costs to build infrastructure that matches required demands.

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

4. *Document link graphs.* The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites.

5. *Protein-protein interactions graphs.* Vertices represent proteins and edges represent interactions between them that carry out some biological function in the cell. These graphs can be used, for example, to study molecular pathways—chains of molecular interactions in a cellular process. Humans have over 120K proteins with millions of interactions among them.

6. *Network packet traffic graphs.* Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them. Such graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity.

7. *Scene graphs.* In graphics and computer games scene graphs represent the logical or spacial relationships between objects in a scene. Such graphs are very important in the computer games industry.

8. *Finite element meshes.* In engineering many simulations of physical systems, such as the flow of air over a car or airplane wing, the spread of earthquakes through the ground, or the structural vibrations of a building, involve partitioning space into discrete elements. The elements along with the connections between adjacent elements forms a graph that is called a finite element mesh.

9. *Robot planning.* Vertices represent states the robot can be in and the edges the possible transitions between the states. This requires approximating continuous motion as a sequence of discrete steps. Such graph plans are used, for example, in planning paths for autonomous vehicles.

10. *Neural networks.* Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 10^{11} neurons and close to 10^{15} synapses.

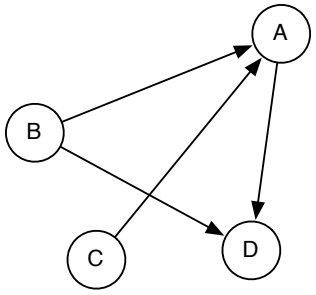
11. *Graphs in quantum field theory.* Vertices represent states of a quantum system and the edges the transitions between them. The graphs can be used to analyze path integrals and summing these up generates a quantum amplitude (yes, I have no idea what that means).

12. *Semantic networks.* Vertices represent words or concepts and edges represent the relationships among the words or concepts. These have been used in various models of how humans organize their knowledge, and how machines might simulate such an organization.

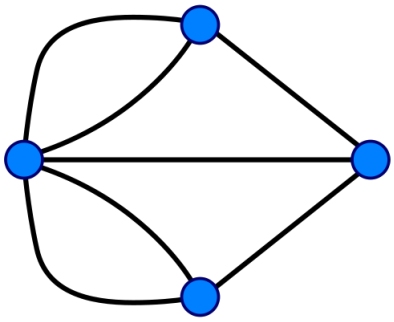
13. *Graphs in epidemiology.* Vertices represent individuals and directed edges the transfer of an infectious disease from one individual to another. Analyzing such graphs has become an important component in understanding and controlling the spread of diseases.

14. *Graphs in compilers.* Graphs are used extensively in compilers. They can be used for type inference, for so called data flow analysis, register allocation and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.

15. *Constraint graphs.* Graphs are often used to represent constraints among items. For example the GSM network for cell phones consists of a collection of overlapping cells. Any pair of cells



An example of a directed graph on 4 vertices.



An undirected graph on 4 vertices[1]

Figure 1: Example Graphs.

that overlap must operate at different frequencies. These constraints can be modeled as a graph where the cells are vertices and edges are placed between cells that overlap.

16. *Dependence graphs.* Graphs can be used to represent dependences or precedences among items. Such graphs are often used in large projects in laying out what components rely on other components and used to minimize the total time or cost to completion while abiding by the dependences.

1.1 Definitions

Formally, a *directed graph* or (*digraph*) is a pair $G = (V, A)$ where

- V is a set of *vertices* (or nodes), and
- $A \subseteq V \times V$ is a set of *directed edges* (or arcs).

Each arc is an ordered pair $e = (u, v)$ and a graph can have *self loops* (u, u). Directed graphs represent asymmetric relationships. An *undirected graph* is a pair $G = (V, E)$ where E is a set of unordered pairs over V (i.e., $E \subseteq \binom{V}{2}$). Each edge can be written as $e = \{u, v\}$, which is the same as $\{v, u\}$, and self loops are generally *not* allowed. Undirected graphs represent symmetric relationships.

Directed graphs are in some sense more general than undirected graphs since we can easily represent an undirected graph by a directed graph by placing an arc in each direction. Indeed, this is often the way we represent undirected graphs in data structures.

Graphs come with a lot of terminology, but fortunately most of it is intuitive once we understand the concept. At this point, we will just talk about graphs that do not have any data associated with edges, such as weights. We will later talk about weighted graphs.

- A vertex u is a *neighbor* of (or equivalently *adjacent* to) a vertex v in a graph $G = (V, E)$ if there is an edge $\{u, v\} \in E$. For a directed graph we use the terms *in-neighbor* if $(u, v) \in E$ and *out-neighbor* if $(v, u) \in E$.

- For an undirected graph $G = (V, E)$, the *neighborhood* $N_G(v)$ of a vertex $v \in V$ is its set of neighbors, i.e. $N_G(v) = \{u \mid \{u, v\} \in E\}$. For a directed graph we use $N_G^+(v)$ to indicate the set of out-neighbors and $N_G^-(v)$ to indicate the set of in-neighbors of v. If we use $N_G(v)$ for a directed graph, we mean the out neighbors. The neighborhood of a set of vertices $U \subseteq V$ is the union of their neighborhoods, e.g. $N_G(V) = \cup_{v \in V} N_G(v)$, or $N_G^+(V) = \cup_{v \in V} N_G^+(v)$.

- The *degree* $d_G(v)$ of a vertex $v \in V$ in a graph $G = (V, E)$ is the size of the neighborhood ($|N_G(v)|$). For directed graphs we use *in-degree* ($d_G^-(v) = |N_G^-(v)|$) and *out-degree* ($d_G^+(v) = |N_G^+(v)|$). We will drop the subscript G when it is clear from the context which graph we're talking about.

- A *path* in a graph is a sequence of adjacent vertices. More formally for a graph $G = (V, E)$, $Paths(G) = \left\{P \in V^+ \mid 1 \leq i < |P|, (P_i, P_{i+1}) \in E\right\}$ is the set of all paths in G, where V^+ indicates all positive length sequences of vertices (allowing for repeats). The length of a path is one less than the number of vertices in the path—i.e., it is the number of edges in the path. A path in a finite graph can have infinite length.

- A *simple path* is a path with no repeated vertices. Often, however, the term simple is dropped, making it sometimes unclear whether path means simple or not (sorry). In this course we will almost exclusively be talking about simple paths and so unless stated otherwise our use of path means simple path.

- A vertex v is *reachable* from a vertex u in G if there is a path starting at v and ending at u in G. We use $R_G(v)$ to indicate the set of all vertices reachable from v in G. An undirected graph is *connected* if all vertices are reachable from all other vertices. A directed graph is *strongly connected* if all vertices are reachable from all other vertices.

- A *cycle* is a path that starts and ends at the same vertex. In a directed graph a cycle can have length 1 (i.e. a *self loop*). In an undirected graph we require that a cycle must have length at least three. In particular going from v to u and back to v does not count. A *simple cycle* is a cycle that has no repeated vertices other than the start vertex being the same as the end. In this course we will exclusively talk about simple cycles and hence, as with paths, we will often drop simple.

- An undirected graph with no cycles is a *forest* and if it is connected it is called a *tree*. A directed graph is a forest (or tree) if when all edges are converted to undirected edges it is undirected forest (or tree). A *rooted tree* is a tree with one vertex designated as the root. For a directed graph the edges are typically all directed toward the root or away from the root.

- A directed graph with no cycles is a *directed acyclic graph* (DAG).

- The *distance* $\delta_G(u, v)$ from a vertex u to a vertex v in a graph G is the shortest path (minimum number of edges) from u to v. It is also referred to as the *shortest path length* from u to v.

- The *diameter* of a graph is the maximum shortest path length over all pairs of vertices: $\mathsf{diam}(G) = \max\left\{\delta_G(u, v) : u, v \in V\right\}$.

Sometimes graphs allow multiple edges between the same pair of vertices, called *multi-edges*. Graphs with multi-edges are called *multi-graphs*. We will allow multi-edges in a couple algorithms just for convenience.

By convention we will use the following definitions:

$$\begin{aligned} n &= |V| \\ m &= |E| \end{aligned}$$

Note that a directed graph can have at most n^2 edges (including self loops) and an undirected graph at most $n(n-1)/2$. We informally say that a graph is *sparse* if $m \ll n^2$ and *dense* otherwise. In most applications graphs are very sparse, often with only a handful of neighbors per vertex when averaged across vertices, although some vertices could have high degree. Therefore, the emphasis in the design of graph algorithms, at least for this class, is typically on algorithms that work well for sparse graphs.

2 How should we represent a graph?

How we want to represent a graph largely depends on the operations we intend to support. For example we might want to do the following on a graph $G = (V, E)$:

(1) Map over the vertices $v \in V$.

(2) Map over the edges $(u, v) \in E$.

(3) Map over the neighbors of a vertex $v \in V$, or in a directed graph the in-neighbors or out-neighbors.

(4) Return the degree of a vertex $v \in V$.

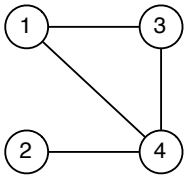
(5) Determine if the edge (u, v) is in E.

(6) Insert or delete vertices.

(7) Insert or delete edges.

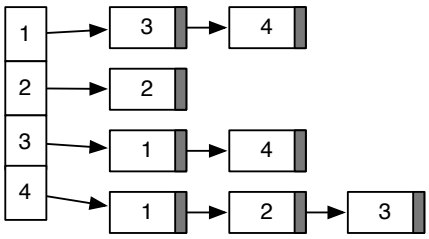
Traditionally, there are four main representations, all of which assume that vertices are numbered from $1, 2, \ldots, n$ (or $0, 1, \ldots, n-1$):

- **Adjacency matrix.** An $n \times n$ matrix of binary values in which location (i, j) is 1 if $(i, j) \in E$ and 0 otherwise. Note that for an undirected graph the matrix is symmetric and 0 along the diagonal.

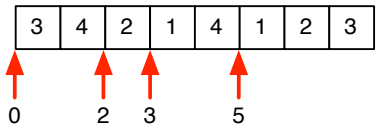


$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

- **Adjacency list.** An array A of length n where each entry $A[i]$ contains a pointer to a linked list of all the out-neighbors of vertex i. In an undirected graph with edge $\{u, v\}$ the edge will appear in the adjacency list for both u and v.



- **Adjacency array.** Similar to an adjacency list, an adjacency array keeps the neighbors of all vertices, one after another, in an array `adj`; and separately, keeps an array of indices that tell us where in the `adj` array to look for the neighbors of each vertex.



- **Edge list.** A list of pairs $(i, j) \in E$.

Since operations on linked lists are inherently sequential, in this course, we are going to raise the level of abstraction so that parallelism is more natural. At the same time, we also want to loosen the restriction that vertices need to be labeled from 1 to n and instead allow for any labels. Conceptually, though, the representations we describe are not much different from adjacency lists and edge lists. We are just going to think parallel and base our view on data types that are parallel.

Here we discuss two representations.

Edge Sets and Tables. Suppose we want a representation that directly follows the mathematical definition of a graph and simply uses a set of edges $E \subseteq \binom{V}{2}$. A moment's thought shows that we can actually support these basic operations rather efficiently using sets. With our ADT for sets, we can implement an *edge set* representation directly. The representation is similar to an edge list representation mentioned before, but it abstracts away from the particular data structure used for the set—the set could be based on a list, an array, a tree, or a hash table. If we want to associate data with each edge, we can use and table instead of a set, which maps each edge to its value.

If we use the balanced-tree cost model for sets, for example, then determining if an edge is in the graph requires much less work than with an edge list— only $O(\log n)$ instead of $\Theta(n)$ (i.e. following the list until the edge is found). The problem with edge sets, as with edge lists, is that they do not allow for an efficient way to access the neighbors of a given vertex v. Selecting the neighbors requires considering all the edges and picking out the ones that have v as an endpoint. Although with an edge set (but not an edge list) this can be done in parallel with $O(\log m)$ span, it requires $\Theta(m)$ work even if the vertex has only a few neighbors.

Adjacency Tables. In our second representation, we aim to get more efficiency in accessing to the neighbors of a vertex. This representation, which we refer to as an *adjacency table*, is a table that maps every vertex to the set of its neighbors. Simply put, it is an edge-set table. If we want to associate data with each edge, we can use an edge-table table, where the table associated with a vertex v maps each neighbor u to the value on the edge (u, v).

In this representation, accessing the neighbors of a vertex v is cheap: it just requires a lookup in the table. Assuming the balanced-tree cost model for tables, this can be done in $O(\log n)$ work and span. Once the neighbor set has been pulled out, mapping a constant work function over the neighbors can be done in $O(d_G(v))$ work and $O(\log d_G(v))$ span. Looking up if an edge is in the graph requires the same work and span as with edge sets: $O(\log n)$. This is because we can first look up one side of the edge in the table and then the second side in the set that is returned. Note that an adjacency list is a special case of adjacency table where the table of vertices is represented as an array and the set of neighbors is represented as a list.

Cost Summary. For these two representations the costs assuming the tree cost model for sets and tables can be summarized in the following table. This assumes the function being mapped uses constant work and span.

	edge set		**adj table**	
	work	*span*	*work*	*span*
`isEdge`$(G,(u,v))$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
`map over all edges`	$O(m)$	$O(\log n)$	$O(m)$	$O(\log n)$
`map over neighbors of` v	$O(m)$	$O(\log n)$	$O(\log n + d_G(v))$	$O(\log n)$
$d_G(v)$	$O(m)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

3 Graph Search

One of the most fundamental tasks on graphs is searching a graph by starting at some source vertex, or set of vertices, and visiting new vertices by crossing (out) edges until there is nothing left to search. In such a search we need to be systematic to make sure that we visit every vertex that is reachable from the source exactly once. This will require recording what vertices have already been visited so they are not visited a second time. Graph searching can be used to determine various properties of graphs, such as whether the graph is connected or whether it is bipartite, as well as various properties relating vertices, such as whether a vertex u is reachable from v, or finding the shortest path between vertices u and v. In the following discussion we use the notation $R_G(u)$ to indicate all the vertices that can be *reached* from u in a graph G (i.e., vertices v for which there is a path from u to v in G).

For all graph search methods we will consider the vertices can be partitioned into three sets at any time during the search:

1. vertices already *visited* (often denoted as X in the notes),
2. the unvisited neighbors of the visited vertices, called the *frontier* (F in the notes),
3. and the rest.

There are three standard graph search methods: breadth first search (BFS), depth first search (DFS), and priority first search (PFS). All these methods visit every vertex that is reachable from a source exactly once, but the order in which they visit the vertices can differ.

Search methods when starting on a single source vertex generate a rooted *search tree*, either implicitly or explicitly.[2] This tree is a subset of the edges from the original graph. In particular a search visits a vertex v by entering from one of its neighbors u via an edge (u, v). This visit to v adds the edge (u, v) to the tree. These edges form a tree (i.e., have no cycles) since no vertex is visited twice and hence there will never be an edge that wraps around and visits a vertex that has already been visited. We refer to the source vertex as the *root* of the tree. Figure 2 gives an example of a graph along with two possible search trees. The first tree happens to correspond to a BFS and the second to a DFS.

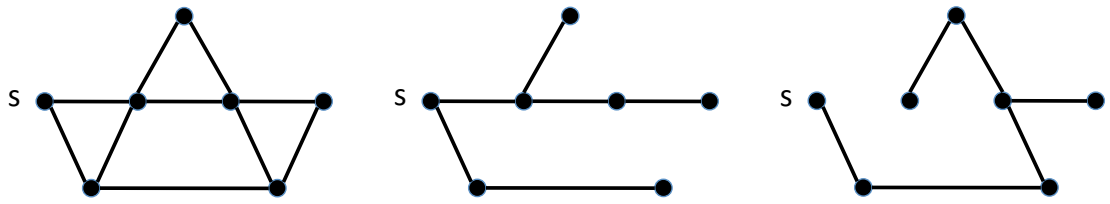


Figure 2: An undirected graph and two possible search trees.

Graph searching has played a very important role in the design of sequential algorithms, but the approach can be problematic when trying to achieve good parallelism. Depth first search (DFS) has a wealth of applications, but it is inherently sequential. Because of this, one often uses other techniques in designing good parallel algorithms. We will cover some of these techniques in upcoming lectures. Breadth first search (BFS), on the other hand, can be parallelized effectively as long as the graph is shallow (the longest shortest path from the source to any vertex is reasonably small). In fact, the depth of the graph will show up in the bounds for span. Fortunately many real-world graphs are shallow. But if we are concerned with worst-case behavior over any graph, then BFS is also sequential.

[2]Note that in assignment 4 you might want to create something other than a tree.

Lecture 10 — BFS

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — September 27, 2012

Material in this lecture:

- Breadth First Search
- Unweighted Shortest Paths

1 Breadth First Search

The first graph search approach we consider is breadth first search (BFS). BFS can be applied to solve a variety of problems including: finding all the vertices reachable from a vertex v, finding if an undirected graph is connected, finding the shortest path from a vertex v to all other vertices, determining if a graph is bipartite, bounding the diameter of an undirected graph, partitioning graphs, and as a subroutine for finding the maximum flow in a flow network (using Ford-Fulkerson's algorithm). BFS, as with the other graph searches, can be applied to both directed and undirected graphs. In the directed case we only consider the outgoing arcs when searching.

The idea of *breadth first search* is to start at a *source* vertex s and explore the graph level by level, first visiting all vertices that are the (out-)neighbors of s (i.e. have distance 1 from s), then vertices that have distance two from s, then distance three, etc. It should be clear that a vertex at distance $i+1$ must have an (in-)neighbor from a vertex a distance i. Therefore, if we know all vertices at distance i, then we can find the vertices at distance $i+1$ by just considering their (out-)neighbors.

As with all the search approaches, the BFS needs to keep track of which vertices have already been visited so that it does not visit them more than once. We will refer to all visited vertices at the start of level i as X_i. Since on level i we visit vertices at a distance i away, the vertices in X_i are exactly those with distance less than i from the source. On each level the search also maintains a frontier. At the start of level i the frontier F_i contains all unvisited neighbors of X_i, which is all vertices in the graph with distance exactly i from s.

In BFS on each level we visit all vertices in the frontier. This differs from DFS which only visits one. What we do when we visit depends on the particular application of BFS, but for now we assume we simply mark the vertices as visited. This is done by simply adding the frontier to the visited vertices, *i.e*, $X_{i+1} = X_i \cup F_i$. To generate the next set of frontier vertices the search simply takes the neighborhood of F and removes any vertices that have already been visited, *i.e.*, $F_{i+1} = N_G(F) \setminus X_{i+1}$. Recall that for a vertex v, $N_G(v)$ are the neighbors of v in the graph G (the out-neighbors for a directed graph) and for a set of vertices F, that $N_G(F) = \cup_{v \in F} N_G(v)$.

Here is pseudocode for a BFS algorithm just described. It returns the set of vertices reachable from a vertex s as well as the shortest distance to the furthest reachable vertex.

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

```
1  fun BFS(G = (V, E), s) =
2  let
3    % requires: X = {u ∈ V | δ_G(s,u) < i} ∧
4                 F = {u ∈ V | δ_G(s,u) = i}
5    % returns: (R_G(s), max{δ_G(s,u) : u ∈ R_G(s)})
6    fun BFS'(X, F, i) =
7      if |F| = 0 then (X, i)
8      else let
9        val X' = X ∪ F          % Visit the Frontier
10       val N = N_G(F)          % Determine the neighbors of the frontier
11       val F' = N \ X'         % Remove vertices that have been visited
12     in BFS'(X', F', i + 1)    % Next level
13     end
14 in BFS'({}, {s}, 0)
15 end
```

Recall that $N_G(F) = \bigcup_{v \in F} N(v)$, where $N(v)$ are the neighbors of v. If we are using a adjacency table representation of the graph, this can be calculated as

$$\textbf{fun } N_G(F) = \texttt{Table.reduce Set.Union \{\} (Table.extract}(G, F))$$

The full SML code for the algorithm is given in the appendix at the end of these notes.

Figure 1 illustrates BFS on an undirected graph where s is the central vertex. Initially, X_0 is empty and F_0 is the single source vertex s, as it is the only vertex that is a distance 0 from s. X_1 is all the vertices that have distance less than 1 from s (just s), and F_1 contains those vertices that are on the inner concentric ring, a distance exactly 1 from s. The outer concentric ring contains vertices in F_2, which are a distance 2 from s. The neighbors $N_G(F_1)$ are the central vertex and those in F_2. Notice that some vertices in F_1 share the same neighbors, which is why $N_G(F)$ is defined as the *union* of neighbors of the vertices in F to avoid duplicate vertices. In general, from which frontiers could the vertices in $N_G(F_i)$ come when the graph is undirected? What if the graph is directed? For the graph in the figure, which vertices are in X_2?

To prove that the algorithm is correct we need to prove the assumptions that are stated in the algorithm. In particular:

Lemma 1.1. *In algorithm BFS when calling BFS'(X, F, i), we have* $X = \{v \in V_G \mid \delta_G(s, v) < i\} \wedge F = \{v \in V_G \mid \delta_G(s, v) = i\}$

Proof. This can be proved by induction on the level i. For the base case (the initial call) we have $X_0 = \{\}$, $F_0 = \{s\}$ and $i = 0$. This is true since only s has distance 0 from s and no vertex has distance less than 0 from s. For the inductive step we assume the claims are correct for i and want to show it for $i + 1$. For X_{i+1} we are simply taking the union of all vertices at distance less than i (X_i) and all vertices at distance exactly i (F_i) so this must include exactly the vertices a distance less than $i + 1$. For F_{i+1} we are taking all neighbors of F_i and removing the X_{i+1}. Since all vertices F_i have distance i from s, by assumption, then a neighbor v of F must have $\delta_G(s, v)$ of no more than $i + 1$. Furthermore

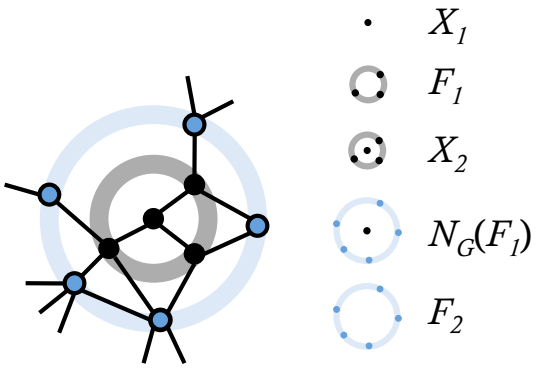


Figure 1: BFS on an undirected graph with the source vertex at the center

all vertices of distance no more than $i+1$ must be reachable from a vertex at distance i. Therefore the neighbors of F_i contain all vertices of distance $i+1$ and only vertices of distance at most $i+1$. When removing X_{i+1} we are left with all vertices of distance i, as needed. □

To argue that the algorithm returns all reachable vertices we note that if a vertex v is reachable from s and has distance $d = \delta(s,v)$ then there must be another u vertex with distance $\delta(s,u) = d-1$. Therefore BFS will not terminate without finding it. Furthermore, for any vertex v, $\delta(s,v) < |V|$ so the algorithm will terminate in at most $|V|$ rounds (levels).

1.1 BFS extensions

So far we have specified a routine that returns the set of vertices reachable from s and the longest length of all shortest paths to these vertices. Often we would like to know more, such as the distance of each vertex from s, or the shortest path from s to some vertex v, i.e. the actual sequence of vertices in the path. It is easy to extend BFS for these purposes. For example the following algorithm returns a table mapping every reachable vertex v to $\delta_G(s,v)$.

```
1  fun BFS(G,s) = let
2    fun BFS'(X, F, i) =
3      if |F| = 0 then X
4      else let
5        val X' = X ∪ {v ↦ i : v ∈ F}
6        val F' = N_G(F) \ domain(X')
7      in BFS'(X', F', i+1) end
8  in BFS'({}, {s}, 0) end
```

To report the actual shortest paths one can generate a shortest path tree, which can be represented as a table mapping each reachable vertex to its parent in the tree. Then one can report the shortest path to a particular vertex by following from that vertex up the tree to the root (see Figure 2). We note that to generate the pointers to parents requires that we not only find the next frontier

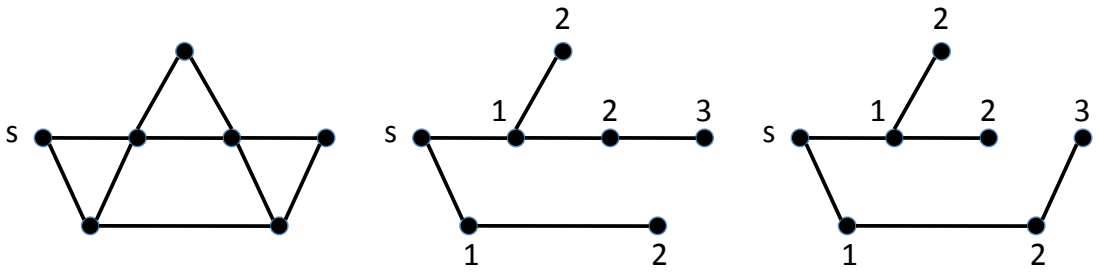


Figure 2: An undirected graph and two possible BFS trees with distances from s

$F' = N(F)/X'$ of the frontier on each level, but that we identify for each vertex $v \in F'$ one vertex $u \in F$ such that $(u, v) \in E$. There could be multiple such edges to the vertex v. Indeed Figure 2 shows two possible trees that differ in what the parent is for the vertex at distance 3.

There are various ways to identify the parent. One is to post-process the result of the BFS that returns the distance. In particular for every vertex we can pick one of its (in-)neighbors with a distance one less than itself. Another way is to identify the parent when generating the neighbors of F. In the appendix we include sample SML code that generates the parents in this way. The ideas is that both the visited vertices X and the frontier F are table that maps the given vertex sets to their parent. Finding the next visited set is easy since it is just a union (or a merge on tables). Finding the next frontier requires tagging each neighbor with where it came from and then merging the results. This merge has decide how to break ties since many vertices in the frontier might have the same neighbor. In the code the first vertex is taken. See the code if you are interested, or need something like this for your homework.

To find $N_G(F)$, the code in the appendix takes the union of the sets of neighbors (one set for each $v \in F$). This time, for each $v \in F$, we generate a table $\{u \mapsto v : u \in N(v)\}$ that maps each neighbor of v back to v. Then instead of union, we use merge to combine all these neighbor tables. In terms of our ML library this would look something like

1.2 BFS Cost

So far in the class we have mostly calculated costs using recurrences. This works well for divide-and-conquer algorithms, but, as we will see, most graph algorithms do not use divide-and-conquer. Instead for many graph algorithms we can calculate costs by counting, i.e., simply adding up the costs across a sequence of rounds of an algorithm.

Since `BFS` works in a sequence of rounds, one per level, we can add up the work and span across the levels. However, the work done on each level varies since it depends on the size of the frontier on that level—in fact it depends on the number of outgoing edges from the frontier for that level. What we do know, however, is that every reachable vertex only appears in the frontier exactly once. Therefore all their out-edges also are only processed exactly once. If we can calculate the cost per edge W_e and per vertex W_v for processing a frontier, then we can simply multiply these by the number of edges and vertices giving $W = W_v n + W_e m$ (recall that $n = |V|$ and $m = |E|$). For the

span we can determine the span per level S_l and multiply it by the number of levels $S = S_l d$, where $d = \max_{v \in V} \delta(s, v)$ is the number of levels.

If we use the tree representations of sets and tables, we can show that the work per edge and per vertex is bounded by $O(\log n)$ and the span per level is bounded by $O(\log^2 n)$. Therefore we have:

$$\begin{aligned} W_{BFS}(n, m, d) &= O(n \log n + m \log n) \\ &= O(m \log n) \\ S_{BFS}(n, m, d) &= O(d \log^2 n) \end{aligned}$$

We drop the $n \log n$ term in the work since for BFS we cannot reach any more vertices than there are edges.

Now lets show that the work per vertex and edge is $O(\log n)$. We can examine the code and consider what is done on each level. In particular the only non-trivial work done on each level is the union $X' = X \cup F$, the calculation of neighbors $N = N_G(F)$ and the set difference $F' = N \setminus F$. The cost of these will depend on the size of the frontier, and in fact in the number of outedges from the frontier. We will use $||F||$ to denote the number of out edges for a frontier plus the size of the frontier, i.e., $||F|| = \sum_{v \in F}(1 + d_G^+(v))$. The costs for each level are as follows

	Work	Span
$X \cup F$	$O(\|F\| \log n)$	$O(\log n)$
$N_G(F)$	$O(\|\|F\|\| \log n)$	$O(\log^2 n)$
$N \setminus F$	$O(\|\|F\|\| \log n)$	$O(\log n)$

The first and last lines fall directly out of the cost spec for the set interface. The second line is a bit more involved. Recall that it is implemented as

fun $N_G(F) =$ `Table.reduce Set.Union {} (Table.extract(`G, F`))`

The work for extract is bounded by $O(|F| \log n)$. For the cost of the union we can use Lemma 2.1 from lecture 6. In particular merge satisfies the conditions of the Lemma, therefore the work is bound by

$$W(\texttt{reduce union \{\}}\ F_{ngh}) = O\left(\log |F_{ngh}| \sum_{\texttt{ngh} \in F_{ngh}} (1 + |\texttt{ngh}|)\right) = O\left(\log n \cdot ||F||\right)$$

where $F_{ngh} =$ `Table.extract(`G, F`)`, and span is bounded by

$$S(\texttt{reduce union \{\}}\ F_{ngh}) = O(\log^2 n)$$

since each union has span $O(\log n)$ and the reduction tree is bounded by $\log n$ depth.

Now we see that work per edge is $O(\log n)$ since on level i we process $||F_i||$ edges and every out edge is only processed once.

Notice that span depends on d. In the worst case $d \in O(n)$ and BFS is sequential. As we mentioned before, many real-world graphs are shallow, and BFS for these graphs has good parallelism.

2 BFS with Single Threaded Sequences

Here we consider a version of BFS that uses sequences to represent the graph and single threaded (ST) sequences to mark the visited vertices. The advantage is that it runs in $O(m)$ total work and $O(d \log n)$ span.

We refer to a graph $G = (V, E)$ where $V = \{0, 1, \ldots, n-1\}$ as an *integer labeled* (IL) graph. For an IL graph we can use the sequences to represent a graph, giving constant work access to the values stored at vertices (if using array sequences). The neighbors of each vertex can also be represented as an array sequence containing the integer indices of their neighbors. An IL graph can therefore be implemented with type:

```
(int seq) seq.
```

We consider the version of BFS that returns a mapping from each vertex to its parent in the BFS tree. We maintain the set of visited vertices using an `(int option) stseq`. This allows us to update which vertices have been visited in constant work per update. In this sequence the option `NONE` indicates the vertex has not been visited, and `SOME`(v) indicates it has been and its parent is v. Each time we visit a vertex, we map it to its parent in the BFS tree; the value v in `SOME`(v) is its parent vertex. As the updates to this sequence are potentially small compared to its length, using an `stseq` is efficient. On the other hand, because the set of frontier vertices is new at each level, we can represent the frontier simply as an integer sequence containing all the vertices in the frontier, allowing for duplicates.

```
1  fun BFS(G : (int seq) seq, s : int) =
2  let
3    fun BFS'(XF : int option stseq, F : int seq) =
4      if |F| = 0 then stSeq.toSeq P
5      else let
6        val N = flatten ⟨ ⟨(u, SOME(v)) : u ∈ G[v]⟩ : v ∈ F ⟩   % neighbors of the frontier
7        val XF' = stSeq.inject(F, XF)                          % new parents added
8        val F' = ⟨ u : (u, v) ∈ N | XF'[u] = v ⟩                % remove duplicates
9      in BFS'(XF', F') end
10   val X_0 = stSeq.toSTSeq(⟨NONE : v ∈ ⟨0, ..., |G| − 1⟩⟩)
11 in
12   BFS'(stSeq.update(s, SOME(s), X_0), ⟨s⟩)
13 end
```

To simplify the algorithm we change the invariant a bit. In particular on entering `BFS'` the sequence `XF` contains parent pointers for both the visited and the frontier vertices instead of just for the visited vertices. F is an integer sequence containing the frontier.

All the work is done in lines 6, 7, and 8. Also note that the `stSeq.inject` on line 7 is always applied to the most recent version. We can write out the following table of costs:

	`XF:stseq`		`XF:seq`	
line	work	span	work	span
6	$O(\|\|F_i\|\|)$	$O(\log n)$	$O(\|\|F_i\|\|)$	$O(\log n)$
7	$O(\|\|F_i\|\|)$	$O(1)$	$O(n)$	$O(1)$
8	$O(\|\|F_i\|\|)$	$O(\log n)$	$O(\|\|F_i\|\|)$	$O(\log n)$
total across all d rounds	$O(m)$	$O(d \log n)$	$O(m + nd)$	$O(d \log n)$

where d is the number of rounds (i.e. the longest path length from s to any other reachable vertex). The last two columns indicate the costs if `XF` was implemented as a regular array sequence instead of an stSeq. The big difference is the cost of `inject`. As before the total work across all rounds is calculated by noting that every out-edge is only processed in one frontier, so $\sum_{i=0}^{d} ||F_i|| = m$.

3 SML Code

Basic BFS. The following SML code for BFS mirrors the pseudo-code in the notes. It uses a table that maps each vertex to a set that contains its (out-)neighbors. The function **fun** `N` G F implements $N_G(F)$ by first using `extract` to get a table with only the vertices in F. That is, the resulting table maps each vertex in F to its neighbors. Next, it combines all the neighbors of F into a single set. Recall that `Table.reduce` f combines the values in the table with the function f.

```
functor TableBFS(Table : TABLE) =
struct
  structure Set = Table.Set
  type vertex = Table.key
  type graph = Set.set Table.table

  fun N (G : graph) (F : Set.set) =
    Table.reduce Set.union Set.empty (Table.extract (G, F))

  fun BFS_reachable (G : graph, s : vertex) =
  let
    (* Require: X = {v in V_G | delta_G(s,v) < i} and
     *          F = {v in V_G | delta_G(s,v) = i}
     * Return: (R_G(s), max {delta_G(s,v) : v in R_G(s)}) *)
    fun BFS' (X : Set.set, F : Set.set, i : int) =
      if  (Set.size F = 0) then (X, i)
      else let
        val X' = Set.union (X, F)
        val F' = Set.difference (N G F, X')
      in
         BFS'(X', F', i+1)
      end
  in
    BFS'(Set.empty, Set.singleton s, 0)
  end
end
```

Generating a BFS Tree. The following code generates a BFS tree. It represents the visited set X and the frontier F as table that map each vertex in the visited set or frontier to their parent in the BFS tree (i.e. who visited them). The function `outEdges` returns the out edges of v or an empty set if v is not found. The $N_G(F)$ function not only returns the neighbors for every vertex $v \in F$, but also tags each neighbor with the vertex v they came from. In particular `tagNeighbors` tags all neighbors with v, and then a `Table.reduce` is used to merge all the tables of tagged neighbors. The `merge` used in the reduce takes the first argument if there are two equal keys, which happens in BFS when a vertex has multiple potential parents.

```
functor TableBFSTree(Table : TABLE) =
struct
  structure Set = Table.Set
  type vertex = Table.key
  type graph = Set.set Table.table
  type 'a table = 'a Table.table
  type set = Set.set
  type parentsTable = vertex table
  fun merge(A,B) = Table.merge (fn (a,b) => a) (A,B)

  fun outEdges (G : graph) (v : vertex) =
      case (Table.find G v) of
        NONE => Set.empty
      | SOME(ngh) => ngh

  (* Return neighbors tagged with where they come from *)
  fun N (G : graph) (F : set) =
      let
        fun tagNeighbors v = Table.tabulate (fn _ => v) (outEdges G v)
        val ngh = Table.tabulate tagNeighbors F
      in
        Table.reduce merge (Table.empty()) ngh
      end

  fun bfsTree (G :graph) (s : vertex) =
      let
        fun BFS(X : parentsTable, F : parentsTable) =
            if (Table.size F = 0) then X
            else let
               val X' = merge(X,F)
               val Ngh = N G (Table.domain F)
               val F' = Table.erase(Ngh, Table.domain X')
            in BFS(X', F') end
      in
         BFS(Table.empty(), Table.singleton(s,s))
      end
end
```

Lecture 11 — Depth-First Search and Applications

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

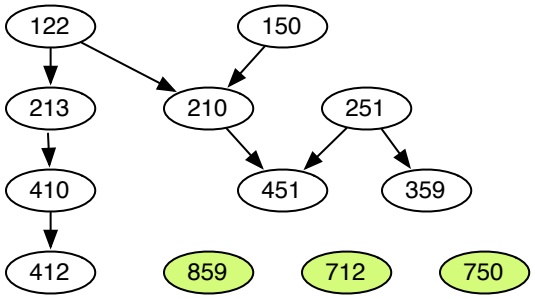
Lectured by Guy Blelloch — October 2, 2012

What was covered in this lecture:

- Depth-first Search
- Using DFS for Cycle Detection (undirected and directed graphs) and Topological Sort

1 A Toy Example

Alice is an ambitious freshman who wants to be well-versed in both theory and systems. She plans to take the following classes during her undergraduate career: 15-122, 15-150, 15-210, 15-213, 15-251, 15-359, 15-451, 15-410, 15-412, 15-750, 15-859, and 15-712. But she knows that she should only take one CS class per semester—and most classes have prerequisites that prevent her from getting in right away. According to her research, the course catalog indicates the following prerequisite structure, depicted as a directed graph:



For example, she cannot take 15-451 Algorithms Design and Analysis until she is done with 15-210 and 15-251—although she could take 15-859 Advanced Algorithms or 15-712 Advanced and Distributed Operating Systems in her first semester (after all, most graduate classes don't have any a formal prerequisite list).

We would like to help her *construct a schedule to take exactly one CS class per semester*. This is known more formally as the *topological sort* problem or simply TopSort. But before we try to generate a schedule for her, we might be interested in finding out whether the graph has a cycle. In a more general context, what we have is a dependency graph and a cycle in a dependency graph indicates a deadlock. For Alice, this would mean such a schedule doesn't exist and she will never graduate unless she drops some of the classes. In this lecture, we will also look at the cycle detection problem for both directed and undirected graphs.

For both problems, we will develop an algorithm using a graph traversal idea called *depth-first search* that looks at an edge at most twice!

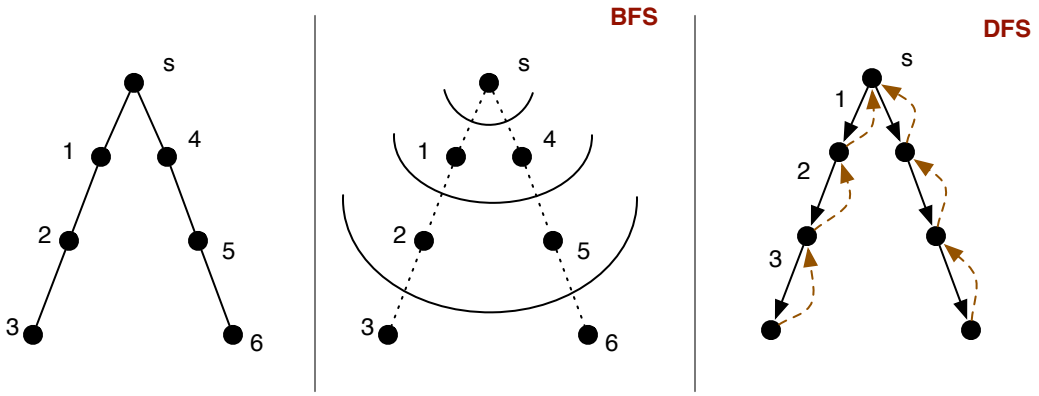
†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

2 DFS: Depth-First Search

Last time, we looked at breadth-first search (BFS), a graph search technique which, as the name suggests, explores a graph in the increasing order of hop count from a source node. We'll spend the bulk of this lecture discussing another equally-common graph search technique, known as depth-first search (DFS). Unlike BFS which explores vertices one level at a time in a breadth first manner, the depth-first search approach proceeds by going as deep as it can until it runs out of unvisited vertices, at which point it backs out until it finds a node with an unvisited neighbor and goes there in the same manner.

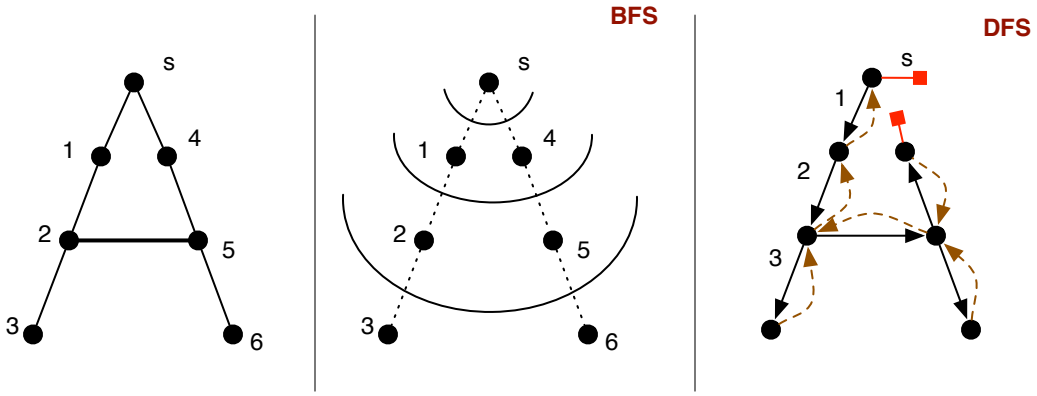
As with BFS, DFS can be used to find all vertices reachable from a start vertex v, to determine if a graph is connected, or to generate a spanning tree. Unlike BFS, it cannot be used to find shortest unweighted paths. But, instead, it is useful in some other applications such as topologically sorting a directed graph (TopSort), cycle detection, or finding the strongly connected components (Scc) of a graph. We will touch on some of these problems briefly.

Example 2.1. *To contrast BFS with DFS, let's consider the following simple V-shaped graph:*



In this example, a BFS starting from s will first visit s, then visit vertices at distance 1 *from s (*1 *and* 4*), then vertices at distance* 2 *from s (*2 *and* 5*), then vertices at distance* 3 *away (*3 *and* 6*)–and we're done. Whereas, a DFS starting form s will go as deep as it can: at s, we could go to* 1 *or* 4 *as the first vertex. Suppose we choose* 1*, then we will proceed to visit* 2 *and* 3*, then after hitting a dead end, we back out to* 2 *then back to* 1*, and proceed down* 4 *to* 5 *to* 6 *and back out.*

Example 2.2. *As another example, we'll look at a slightly more complicated version of the graph above, where we join the nodes* 2 *and* 5 *together with an edge.*



Nothing changes when we run BFS despite that extra edge between 2 *and* 5*; however, the order in which DFS visits vertices changes drastically. Suppose we start at s and choose to go down node* 1 *first. We will visit* 2*,* 3*, then back out to* 2*, then since* 2 *has* 5 *as its neighbor, we'll visit* 5*, which in turn, will take us to* 4 *and to s, but s has been visited before, so we won't visit that—we back out. We'll then go to* 5*,*6*, back to* 5*, to* 2*, to* 1*, to s. We will try to visit* 4 *but quickly realize that* 4 *has been visited, so again, we back out.*

How do we turn this idea into code? We first consider a simple version of depth-first search that simply returns a set of reachable vertices. Notice that in this case, the algorithm returns exactly the same set as BFS—but the crucial difference is that DFS visits the vertices in a different order (depth vs. breadth). Here is the code:

```
        :  fun DFS(G, s) = let
        :    fun DFS'(X, v) =
        :      if (v ∈ X)
TOUCH v :      then X
        :      else let
ENTER v :        val X' = X ∪ {v}
        :        val X'' = iter DFS' X' (N_G(v))
 EXIT v :      in X'' end
        :  in DFS'({}, s) end
```

The helper function $\texttt{DFS}'(X, v)$ does all the work. X is the set of already visited vertices (as in BFS) and v is a single vertex we want to explore from. The code first tests if v has already been visited and returns if so. Otherwise it visits the vertex v by adding it to X (line ENTER v), iterating itself recursively on all neighbors, and finally returning the updated set of visited vertices (line EXIT v). Recall that $(\texttt{iter}\ f\ s_0\ A)$ iterates over the elements of A starting with a state s_0. Each iteration uses the function $f : \alpha \times \beta \to \alpha$ to map a state of type α and element of type β to a new state. It can be thought of as:

$S = s_0$
foreach $a \in A$:
  $S = f(S, a)$
return S

For a sequence `iter` processes the elements in the order of the sequence, but since sets are unordered the ordering of `iter` will depend on the implementation.

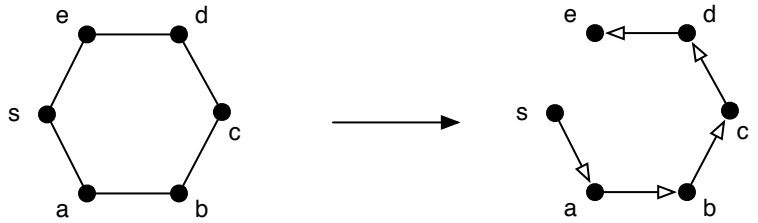
What this means for the DFS algorithm is that when the algorithm visits a vertex v (i.e., reaches the line ENTER v), it picks the first outgoing edge (v, w_1), through `iter`, calls `DFS'`$(X \cup \{v\}, w_1)$ to explore the unvisited vertices reachable from w_1. When the call `DFS'`$(X \cup \{v\}, w_1)$ returns the algorithm has fully explored the graph reachable from w_1 and the vertex set returned (call it X_1) includes all vertices in the input $X \cup v$ plus all vertices reachable from w_1. The algorithm then picks the next edge (v, w_2), again through `iter`, and fully explores the graph reachable from w_2 starting with the the vertex set X_1. The algorithm continues in this manner until it has fully explored all out-edges of v. At this point, `iter` is complete—and X'' includes everything in the original $X' = X \cup \{v\}$ as well as everything reachable from v.

Touching, Entering, and Exiting. There are three points in the code that are particularly important since they play a role in various proofs of correctness and also these are the three points at which we will add code for various applications of `DFS`. The points are labeled on the left of the code. The first point is TOUCH v which is the point at which we try to visit a vertex v but it has already been visited and hence added to X. The second point is ENTER v which is when we first encounter v and before we process its out edges. The third point is EXIT v which is just after we have finished visiting the out-neighbors and are returning from visiting v. At the exit point all vertices reachable from v must be in X.

Exercise 1. *At* ENTER v *can any of the vertices reachable from* v *already be in* X*? Answer this both for directed and separately for undirected graphs.*

Is `DFS` parallel? At first look, we might think this approach can be parallelized by searching the out edges in parallel. This would indeed work if the searches initiated never "meet up" (e.g., the graph is a tree so paths never meet up). However, when the graphs reachable through the outgoing edges are shared, visiting them in parallel creates complications because we don't want to visit a vertex twice and we don't know how to guarantee that the vertices are visited in a depth-first manner.

To understand the situation better, we'll look at a simple example: a cycle graph on 6 nodes (C_6). The left figure shows a 6-cycle C_6 with nodes s, a, b, c, d, e and the right figure shows the order (as indicated by the arrows) in which the vertices are visited as a result of starting at s and first visiting a.



In this example, since the search wraps all the way around, we couldn't know until the end that e would be visited (in fact, it got visited last), so we couldn't start searching s's other neighbor e until we are done searching the graph reachable from a. More generally, in an undirected graph, if two

unvisited neighbors u and v have any reachable vertices in common, then whichever is explored first will always wrap all the way around and visit the other one.

Indeed, depth-first search is known to be **P**-complete, a class of computations that can be done in polynomial work but are widely believed that they do not admit a polylogarithmic span algorithm. A detailed discussion of this topic is beyond the scope of our class. But this provides further evidence that DFS might not be highly parallel.

Cost of DFS The cost of `DFS` will depend on what data structures we use to implement the set, but generally we can bound it by counting how many operations are made and multiplying it by the cost of each operation. In particular we have the following

Lemma 2.3. *For a graph $G = (V, E)$ with m out edges, and n vertices, `DFS'` will be called at most m times and a vertex will be entered for visiting at most $\min(n, m)$ times.*

Proof. Every vertex will be visited at most once since we always add them to X when we enter so the the test $v \in X$ can only fail n times. Every out-edge will only be traversed once invoking a call to `DFS'` since its vertex is only visited once, so at most m calls will be made to `DFS'`. Finally we can only enter a vertex once per call to `DFS'` so the number of enters is bounded by $\min(n, m)$. □

Note that each time we enter `DFS'` we do one check to see if if $v \in X$. For each time we enter a vertex for visiting we do one insertion of v into X. We therefore do at most $\min m, n$ finds and m insertions. This gives:

Corollary 2.4. *The `DFS` algorithm a graph with m out edges, and n vertices, and using the tree-based cost specification for sets runs in $O(m \log n)$ work and span.*

Later we will consider a version based on single threaded sequences that reduces the work and span to $O(m)$.

3 Cycle Detection: Undirected Graphs

We now consider some other applications of `DFS` beyond just reachability. Given a graph $G = (V, E)$ *cycle detection* problem is to determine if there are any cycles in the graph. The problem is different depending on whether the graph is directed or undirected, and here we will consider the undirected case. Later we will also look at the directed case.

How would we modify the generic `DFS` algorithm above to solve this problem? A key observation is that in an undirected graph if `DFS'` ever arrives at a vertex v a second time, and the second visit is coming from another vertex u (via the edge (u, v)), then there must be two paths between u and v: the path from u to v implied by the edge, and a path from v to u followed by the search between when v was first visited and u was visited. Since there are two distinct paths, there is a "cycle". Well not quite! Recall that in an undirected graph a cycle must be of length at least 3, so we need to be careful not to consider the two paths $\langle u, v \rangle$ and $\langle v, u \rangle$ implied by the fact the edge is bidirectional (i.e. a length 2 cycle). It is not hard to avoid these length two cycles. These observations lead to the following code.

```
        :  fun undirectedCycle(G, s) = let
        :    fun DFS' p ((X, C), v) =
        :      if (v ∈ X)
TOUCH v :      then (X, true)
        :      else let
ENTER v :        val X' = X ∪ {v}
        :        val (X'', C') = iter (DFS' v) (X', C) (N_G(v) \ {p})
 EXIT v :      in (X'', C') end
        :  in DFS' s (({}, false), s) end
```

The code returns both the visited set and whether there is a cycle. The key differences from the generic DFS are underlined. The variable C is a boolean variable indicating whether a cycle has been found so far. It is initially set to `false` and set to `true` if we find a vertex that has already been visited. The extra argument p to DFS$'$ is the parent in the DFS tree, i.e. the vertex from which the search came from. It is needed to make sure we do not count the length 2 cycles. In particular we remove p from the neighbors of v so the algorithm does not go directly back to p from v. The parent is passed to all children by "currying" using the partially applied (DFS$'$ v). If the code executes the TOUCH v line then it has found a path of at least length 2 from v to p and the length 1 path (edge) from p to v, and hence a cycle.

Exercise 2. *In the final line of the code the initial "parent" is the source s itself. Why is this OK for correctness?*

4 Topological Sorting

We now return to topological sorting as a second application of DFS.

Directed Acyclic Graphs. A directed graph that has no cycles is called a *directed acyclic graph* or DAG. DAGs have many important applications. They are often used to represent dependence constraints of some type. Indeed, one way to view a parallel computation is as a DAG where the vertices are the jobs that need to be done, and the edges the dependences between them (e.g. a has to finish before b starts). Mapping such a computation DAG onto processors so that all dependences are obeyed is the job of a scheduler. You have seen this briefly in 15-150. The graph of dependencies cannot have a cycle, because if it did then the system would deadlock and not be able to make progress.

The idea of topological sorting is to take a directed graph that has no cycles and order the vertices so the ordering respects reachability. That is to say, if a vertex u is reachable from v, then v must be lower in the ordering. In other words, if we think of the input graph as modeling dependencies (i.e., there is an edge from v to u if u depends on v), then topological sorting finds a partial ordering that puts a vertex *after* all vertices that it depends on.

To make this view more precise, we observe that a DAG defines a so-called *partial order* on the vertices in a natural way:

For vertices $a, b \in V(G)$, $a \leq_p b$ if and only if there is a directed path from a to b[1]

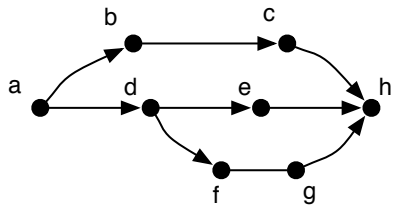
Remember that a partial order is a relation $\leq_p$ that obeys

1. reflexivity — $a \leq_p a$,
2. antisymmetry — if $a \leq_p b$ and $b \leq_p a$, then $b = a$, and
3. transitivity — if $a \leq_p b$ and $b \leq_p c$ then $a \leq_p c$.

In this particular case, the relation is on the vertices. It's not hard to check that the relation based on reachability we defined earlier satisfies these 3 properties. Armed with this, we can define the topological sorting problem formally:

Problem 4.1 (Topological Sorting(TopSort))**.** A *topological sort* of a DAG is a total ordering $\leq_t$ on the vertices of the DAG that respects the partial ordering (i.e. if $a \leq_p b$ then $a \leq_t b$, though the other direction needs not be true).

For instance, consider the following DAG:



We can see, for example, that $a \leq_p c$, $d \leq h$, and $c \leq h$. But it is a partial order: we have no idea how c and g compare. From this partial order, we can create a total order that respects it. One example of this is the ordering $a \leq_t b \leq_t \leq_t c \leq_t d \leq_t e \leq_t f \leq_t g \leq_t h$. Notice that, as this example graph shows, there are many valid topological orderings.

Solving TopSort using `DFS`. To topologically sort a graph, we augment our directed graph $G = (V, D)$ with a new source vertex s and a set of directed edges from the source to every vertex, giving $G' = (V \cup \{s\}, E \cup \{(s, v) : v \in V\})$. We then run the following variant of `DFS` on G' starting at s:

```
        : fun topSort(G = (V, E)) = let
        :   val s = a new vertex
        :   val G' = (V ∪ {s}, E ∪ {(s, v) : v ∈ V})
        :
        :   fun DFS'((X, L), v) =
        :     if (v ∈ X)
TOUCH v :     then (X, L)
        :     else let
ENTER v :       val X' = X ∪ {v}
        :       val (X'', L') = iter DFS' (X', L) (N_G'(v))
 EXIT v :     in (X'', v :: L') end
        :
        : in DFS'(({}, []), s) end
```

[1] We adopt the convention that there is a path from a to a itself, so $a \leq_p a$.

The significant changes from the generic version are marked with underlines. In particular we thread a list L through the search. The only thing we do with this list is cons the vertex v onto the front of it when we exit `DFS` for vertex v (line EXIT v). We claim that at the end, the ordering in the list returned specifies a topological sort of the vertices, with the earliest at the front.

Why is this correct? The correctness crucially follows from the property that `DFS` fully searches any unvisited vertices that are reachable from it before returning. In particular the following theorem is all that is needed.

Theorem 4.2. *On a DAG when exiting a vertex v in* `DFS` *all vertices reachable from v have already exited.*

Proof. This theorem might seem obvious, but we have to be a bit careful. Consider a vertex u that is reachable from v and consider the two possibilities of when u is entered relative to v.

1. u is entered before v is entered. In this case u must also have exited before v is entered otherwise there would be a path from u to v and hence a cycle.
2. u is entered after v is entered. In this case since u is reachable from v it must be visited while searching v and therefore exit before v exits. □

This theorem implies the correctness of the code for topological sort. This is because it places vertices on the front of the list in exit order so all vertices reachable from a vertex v will appear after it in the list, which is the property we want.

5 Cycle Detection: Directed Graphs

We now return to cycle detection but in the directed case. This can be an important preprocessing step for topological sort since topological sort will return garbage for a graph that has cycles. As with topological sort, we augment the input graph $G = (V, E)$ by adding a new source s with an edge to every vertex $v \in V$. Note that this modification cannot add a cycle since the edges are all directed out of s. Here is the code:

```
          :  fun directedCycle(G = (V, E)) = let
          :    val s = a new vertex
          :    val G' = (V ∪ {s}, E ∪ {(s, v) : v ∈ V})
          :
          :    fun DFS'((X, Y, C), v) =
          :      if (v ∈ X)
TOUCH v   :      then (X, Y, v ∈? Y)
          :      else let
ENTER v   :        val X' = X ∪ {v}
          :        val Y' = Y ∪ {v}
          :        val (X'', Y'', C') = iter DFS' (X', Y', C) (N_G'(v))
EXIT v    :      in (X'', Y'' \ {v}, C') end
          :
          :    val (_, _, C) = DFS'(({}, {}, false), s)
          : in C end
```

The differences from the generic version are once again underlined. In addition to threading a boolean value C through the search that keeps track of whether there are any cycles, it threads the set Y through the search. When visiting a vertex v, the set Y contains all vertices that are ancestors of v is the DFS tree. This is because we add a vertex to Y when entering the vertex and remove it when exiting. Therefore, since recursive calls are properly nested, the set will contain exactly the vertices on the recursion path from the root to v, which are also the ancestors in the DFS tree.

To see how this helps we define a *back edge* in a DFS search to be an edge that goes from a vertex v to an ancestor u in the DFS tree.

Theorem 5.1. *A directed graph $G = (V, E)$ has a cycle if and only if for $G' = (V \cup \{s\}, E \cup \{(s, v) : v \in V\})$ a* DFS *from s has a back edge.*

Exercise 3. *Prove this theorem.*

5.1 Generalizing DFS

As already described there is a common structure to all the applications of DFS—they all do their work either when "entering" a vertex, when "exiting" it, or when "touching" it, i.e. attempting to visit when already visited. This suggests that we might be able to derive a generic version of DFS in which we only need to supply functions for these three components. This is indeed possible by having the user define a state of type α that can be threaded throughout search, and then supplying and an initial state and three functions:

$$
\begin{array}{ll}
\Sigma_0 & : \alpha \\
\texttt{touch} & : \alpha \times \texttt{vertex} \times \texttt{vertex} \to \alpha \\
\texttt{enter} & : \alpha \times \texttt{vertex} \times \texttt{vertex} \to \alpha \\
\texttt{exit} & : \alpha \times \texttt{vertex} \times \texttt{vertex} \to \alpha
\end{array}
$$

where each function takes the state, the current vertex v, and the parent vertex p in the DFS tree, and returns an updated state. The code can then be written as:

```
1  fun DFS(G, Σ0, s) = let
2    fun DFS' p ((X, Σ), v) =
3      if (v ∈ X) then (X, touch(Σ, v, p))
4      else let
5        val Σ' = enter(Σ, v, p)
6        val (X', Σ'') = iter (DFS'p) (X ∪ {v}, Σ') N_G^+(v)
7        val Σ''' = exit(Σ, Σ'', v, p)
8      in (X', Σ''') end
9  in DFS' s ((∅, Σ0), s) end
```

At the end, DFS returns an ordered pair $(X, \Sigma) : \texttt{Set} \times \alpha$, which represents the set of vertices visited and the final state Σ.

With this code we can easily define our applications of DFS. For undirected cycle detection we have:

$\Sigma_0 = ([s],\ \texttt{false}) : \texttt{vertex list} \times \texttt{bool}$
fun $\texttt{touch}((h :: T,\ \texttt{fl}),\ v,\ p) = (L,\ h \neq p)$
fun $\texttt{enter}((L,\ \texttt{fl}),\ v,\ p) = (v :: L,\ \texttt{fl})$
fun $\texttt{exit}((h :: T,\ \texttt{fl}),\ v,\ p) = (T,\ \texttt{fl})$

For topological sort we have.

$\Sigma_0 = [\,] : \texttt{vertex list}$
fun $\texttt{touch}(L,\ v,\ p) = L$
fun $\texttt{enter}(L,\ v,\ p) = L$
fun $\texttt{exit}(L,\ v,\ p) = v :: L$

For directed cycle detection we have.

$\Sigma_0 = (\{\},\ \texttt{false}) : \texttt{Set} \times \texttt{bool}$
fun $\texttt{touch}((S,\ \texttt{fl}),\ v,\ p) = (S,\ v \in^? S)$
fun $\texttt{enter}((S,\ \texttt{fl}),\ v,\ p) = (S \cup \{v\},\ \texttt{fl})$
fun $\texttt{exit}((S,\ \texttt{fl}),\ v,\ p) = (S \setminus \{v\},\ \texttt{fl})$

For these last two cases we need to also augment the graph with the vertex s and add the the edges to each vertex $v \in V$.

6 DFS with Single-Threaded Arrays

Here is a version of DFS using adjacency sequences for representing the graph and ST sequences for keeping track of the visited vertices.

```
1  fun DFS(G : (int seq) seq, s : int) =
2  let
3    fun DFS' p ((X : bool stseq, Σ), v : int) =
4      if (X[v]) then (X, touch(Σ, v, p))
5      else let
6        val X' = update(v, true, X)
7        val Σ' = enter(Σ, v, p)
8        val (X'', Σ'') = iter (DFS'v) (X', Σ') (G[v])
9      in (X'', exit(Σ'', v, p))
10   val X_init = stSeq.fromSeq(⟨false : v ∈ ⟨0, ..., |G| − 1⟩⟩)
11 in
12   stSeq.toSeq(DFS'((X_init, Σ_0), s))
13 end
```

If we use an `stseq` for X (as indicated in the code) then this algorithm uses $O(m)$ work and span. However if we use a regular sequence, it requires $O(n^2)$ work and $O(m)$ span.

7 SML Code

Here we give the SML code for the generic version of DFS along with the implementation of directed cycle detection and topological sort.

```
signature DFSops =
sig
  type vertex
  type state
  val start : state
  val enter : state * vertex * vertex -> state
  val exit : state * vertex * vertex -> state
  val touch : state * vertex * vertex -> state
end

functor DFS(structure Table  : TABLE
            structure Ops : DFSops
            sharing type Ops.vertex = Table.Key.t) =
struct
  open Ops
  open Table
  type graph = set table

  fun N(G : graph, v : vertex) =
    case (Table.find G v) of
        NONE => Set.empty
      | SOME(ngh) => ngh

  fun dfs (G : graph, s : vertex) =
    let
      fun dfs' p ((X : set, S : state), v : vertex) =
        if (Set.find X v) then (X, touch(S,p,v))
        else
          let
            val X' = Set.insert v X
            val S' = enter(S, p, v)
            val (X'',S'') = Set.iter (dfs' p) (X',S') (N(G,v))
            val S''' = exit(S'', p, v)
          in (X'',S''')
          end
    in
       dfs' s ((Set.empty, start), s)
    end
end


functor CycleCheck(Table : TABLE) = DFS(
  structure Table = Table
  structure Ops =
  struct
    structure Set = Table.Set
    type vertex = Table.key
```

```
    type state = Set.set * bool
    val start = (Set.empty,false)
    fun enter((S,fl),p,v) = (Set.insert v S, fl)
    fun exit((S,fl),p,v) = (Set.delete v S, fl)
    fun touch((S,fl),p,v) = if (Set.find S v)
                            then (S,true)
                            else (S,fl)
  end)

functor TopSort(Table : TABLE) = DFS(
  structure Table = Table
  structure Ops =
  struct
    type vertex = Table.key
    type state = vertex list
    val start = []
    fun enter(L,_,_) = L
    fun exit(L,p,v) = v::L
    fun touch(L,_,_) = L
  end)
```

Lecture 13 — Shortest Weighted Paths

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — 9 October 2012

What was covered in this lecture:

- Weighted Graphs
- Priority First Search
- Weighted Shortest Paths
- Dijkstra's Algorithm – finished in next classs

1 Weighted Graph Representation

It is often necessary to associate weights or other values with the edges of a graph. Such a "weighted" or "edge-labeled" graph can be defined as a triple $G = (E, V, w)$ where $w : E \to \texttt{eVal}$ is a function mapping edges or directed edges to their values, and `eVal` is the set (type) of possible values. For a weighted graph `eVal` would typically be the real numbers, but for edge-labeled graphs they could be any type.

There are a few ways to represent a weighted or edge-labeled graph. The first representation translates directly from representing the function w. In particular we can use a table that maps each edge (a pair of vertex identifiers) to its value. This would have type

$$\texttt{eVal edgeTable}$$

That is, the keys of the table are edges and the values are of type `eVal`. For example, for a weighted graph we might have:

$$W = \{(0,1) \mapsto 0.7,\ (1,2) \mapsto -2.0,\ (0,2) \mapsto 1.5\}$$

This representation would allow us to look up the weight of an edge e using: $w(e) = \texttt{find}\ W\ e$.

Another way to associate values with edges is to use a structure similar to the adjacency set representation for unweighted graphs and associate a value with each out edge of a vertex. In particular, instead of associating with each vertex a set of out-neighbors, we can associate each vertex with a table that maps each out-neighbor to its value. It would have type:

$$(\texttt{eVal vertexTable})\ \texttt{vertexTable}\ .$$

The graph above would then be represented as:

$$G = \{0 \mapsto \{1 \mapsto 0.7,\ 2 \mapsto 1.5\},\ 1 \mapsto \{2 \mapsto -2.0\},\ 2 \mapsto \{\}\}\ .$$

We will mostly be using this second representation.

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

2 Priority First Search

Generalizing BFS and DFS, *priority first search* (also called best-first search) visits the vertices in some priority order. This priority order can either be static and decided ahead of time, or can be generated on the fly by the algorithm. Recall that a generic graph search maintains two sets: the set of visited vertices X, and the set of frontier vertices $F = N(X) \setminus X$ (in DFS the frontier is maintained implicitly on the recursion stack). The frontier vertices are the unvisited vertices we know about by having visited their in-neighbors. On each step we visit one or more vertices on the frontier, move them from F to X and add their unvisited out-neighbors to F. A priority first search can thus be seen as follows:

```
1  fun pfs(X, F) =
2  if (F = ∅) then X
3  else let
4     val M = highest priority vertices in F
5     val X' = X ∪ M
6     val F' = (F ∪ N(M)) \ X'
7  in pfs(X', F') end
```

This would typically start with $X = \emptyset$ and the frontier F containing a single source.

One can imagine using such a scheme, for example, to explore the web in a way that prioritizes the more interesting or relevant pages. The idea is to keep a ranked list of unvisited outgoing links based on how interesting the tag on the link appears to be. Then, when choosing what page to visit next, follow the link with the highest rank. When visiting the page, scratch it off the list and add all its unvisited outgoing links to the list with ranks.

Several famous graph algorithms are instances of priority first search. For example, Dijkstra's algorithm for finding single-source shortest paths (SSSP), discussed next and Prim's algorithm for finding Minimum Spanning Trees (MST). Priority First Search is a greedy technique since it greedily selects among the choices in front of it (the frontier) based on some cost function (the priorities) and never backs up. Algorithms based on PFS are hence often referred to as greedy algorithms.

2.1 Shortest Weighted Paths

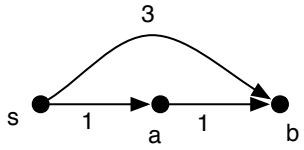
The single-source shortest path (SSSP) problem is to find the shortest (weighted) path from a source vertex s to every other vertex in the graph. Consider a weighted graph $G = (V, E, w)$, $w : E \to \mathbb{R}$. The graph can be either directed or undirected. For convenience we define $w(u, v) = \infty$ if $(u, v) \notin E$. The *weight of a path* is the sum of the weights of the edges along that path.

Definition 2.1 (The Single-Source Shortest Path (SSSP) **Problem**)**.** Given a weighted graph $G = (V, E, w)$ and a source vertex s, the *single-source shortest path (SSSP) problem* is to find the shortest weighted path from s to every other vertex in V.

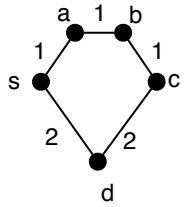
For a weighted graph G we will use $\delta_G(u, v)$ to indicate the weight of the shortest path from u to v. Dijkstra's algorithm solves the SSSP problem when all the weights on the edges are non-negative (i.e. $w : E \to \mathbb{R}_+ \cup \{0\}$). It is a very important algorithm both because shortest paths have many

applications, and also because it is a very elegant example of an efficient greedy algorithm that generates optimal solutions on a nontrivial task.

Before describing Dijkstra's algorithm, we would like to understand why we need a new algorithm for the weighted case. Why, in particular, doesn't BFS work on weighted graphs? Consider the following directed graphs on 3 nodes:



In this example, BFS would visit b then a. This means when we visit b, we assign it an incorrect weight of 3. Since BFS never visit it again, we'll never update it to the correct value. Note that we cannot afford to visit it again as it will be much more expensive. This problem still persists even if the graph is geometric (and satisfies the triangle inequality), as the following example shows:



But why does BFS work in unweighted case? The key idea is that it works outwards from the source. For each frontier F_i, it has the correct unweighted distance from source to each vertex in the frontier. It then can determine the correct distance for unencountered neighbors that are distance one further away (on the next frontier).

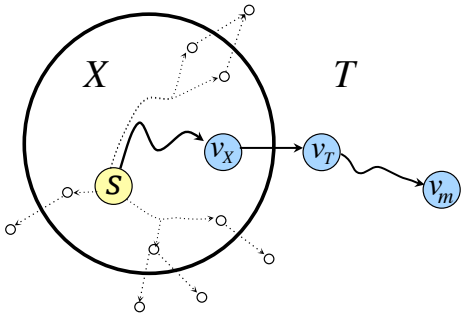
Let's consider using a similar approach when the graphs has non-negative edge weights. Starting from the source vertex s, for which vertex can we safely say we know its shortest path from s? The vertex v that is the closest neighbor of s. There could not be a shorter path to v, since such a path would have to go through one of the neighbors that is further away from s and that path cannot get shorter because none of the edge weights are negative. More generally, if we know the shortest path distances for a set of vertices, how can we determine the shortest path to another vertex? This is the question that Dijkstra's algorithm answers.

3 Dijkstra's Algorithm for SSSP

The key property used by Dijkstra's algorithm is that for a set of vertices $X \subset V$ that include s and the rest of the vertices ($T = V \setminus X$), the closest vertex in T from s based on paths that only go through X is also an overall closest vertex in T. This property will allow us to select the next closest vertex by only considering the vertices we have already visited. Defining $\delta_{G,X}(s,v)$ as the shortest path length from s to v in G that only goes through vertices in X (except for v), the property can be stated more formally and proved as follows:

Lemma 3.1 (Dijkstra's Property)**.** *Consider a (directed) weighted graph $G = (V, E)$, $w : E \to \mathbb{R}_+ \cup \{0\}$, and a source vertex $s \in V$. For any partitioning of the vertices V into X and $T = V \setminus X$ with $s \in X$,*

$$\min_{t \in T} \delta_{G,X}(s,t) = \min_{t \in T} \delta_G(s,t) \,.$$



Proof. Consider a vertex $v_m \in T$ such that $\delta_G(s, v_m) = \min_{t \in T} \delta_G(s,t)$, and a shortest path from s to v_m in G. The path must cross from X to T at some point using some edge (v_X, v_T). Since subpaths of shortest paths are shortest paths, the subpath from s to v_T is the shortest path to v_T and, since edges are non-negative, distances don't decrease along the path implying $\delta_G(s, v_T) \leq \delta_G(s, v_m)$ (it could be that $v_T = v_m$). Furthermore since the shortest path to v_T only goes through X, $\delta_{G,X}(s, v_T) = \delta_G(s, v_T)$. Together we have:

$$\min_{t \in T} \delta_{G,X}(s,t) \leq \delta_{G,X}(s, v_T) = \delta_G(s, v_T) \leq \delta_G(s, v_m) = \min_{t \in T} \delta_G(s,t) \,.$$

Also $\min_{t \in T} \delta_{G,X}(s,t) \geq \min_{t \in T} \delta_G(s,t)$ since we are only restricting paths by requiring them to go through X. Together we have the equality. □

This lemma implies that if we have already visited a set of vertices X we can find a new shortest path to a vertex in $T = V \setminus X$ by just considering the path lengths through X to a neighbor of X. In particular we want to pick a vertex t that minimizes $\delta_{G,X}(s,t)$. This suggests an algorithm for shortest paths based on priority first search using the priority $P(v) = \delta_{G,X}(s,u)$. Also note that $\delta_{G,X}(u) = \min_{v \in V}(\delta_G(v) + w(v,t))$. Indeed this gives us Dijkstra's algorithm, at least in the abstract:

Definition 3.2 (Dijkstra's Algorithm)**.** Given a weighted graph $G = (V, E, w)$ and a source s, Dijkstra's algorithm is priority first search on G starting at s using priority $P(v) = \min_{v \in V}(d(v) + w(v,t))$ (to be minimized) and setting $d(v) = P(v)$ when v is visited and $d(s) = 0$.

Lemma 3.3. *When Dijkstra's algorithm returns, $d(v) = \delta_G(s,v)$ for all reachable v.*

Proof. We prove that $d(x) = \delta_G(s,x)$ for $x \in X$ by induction on the size of X. The base case is true since $d(s) = 0$. Assuming it is true for size i, then the algorithm adds the vertex v that minimizes $P(v) = \delta_{G,X}(s,v)$. By Lemma 3.1 $\delta_{G,X}(s, fv) = \delta_G(s,v)$ giving $d(v) = \delta_G(s,v)$ for $i+1$. □

A curious aspect of Dijkstra's Algorithm. Although Dijkstra's algorithms visits the vertices in increasing (technically non-decreasing) order of distance, this feature does not play any role in the correctness of the algorithm. For example if someone during the algorithm removed an arbitrary set of vertices from X along with their distances and then continued running the algorithm, it would still run correctly. This is because Lemma 3.1 makes no requirement about how X and T are partitioned other than $s \in X$.

We now discuss how to implement this abstract algorithm efficiently using a priority queue to maintain $P(v)$. We use a priority queue that supports `deleteMin` and `insert`. The algorithm is as follows:

```
1  fun dijkstra(G,s) =
2  let
3    fun dijkstra'(X, Q) =
4      case PQ.deleteMin(Q) of
5          (NONE, _) ⇒ X
6        | (SOME(d,v),Q') ⇒
7          if ((v, _) ∈ X) then dijkstra'(X, Q')
8          else let
9            val X' = X ∪ {(v,d)}
10           fun relax (Q,(u,w)) = PQ.insert(d+w,u) Q
11           val Q'' = iter relax Q' (N_G(v))
12         in dijkstra'(X', Q'') end
13 in
14   dijkstra'({}, PQ.insert (0,s) {})
15 end
```

The algorithm maintains the visited set X as a table mapping each visited vertex u to $d(u) = \delta_G(s,u)$. It also maintains a priority queue Q that keeps the frontier prioritized based on the shortest distance from s through vertices in X. On each round, the algorithm selects the vertex x with least distance d in the priority queue (Line 4 in the code) and, if it hasn't already been visited, visits it by adding $(x \mapsto d)$ to the table of visited vertices (Line 9), and then adds all its neighbors v to Q along with the priority $d(x) + w(x,v)$ (i.e. the distance to v through x) (Lines 10 and 11). Note that a neighbor might already be in Q since it could have been added by another of its in-neighbors. Q can therefore contain duplicate entries for a vertex, but what is important is that the minimum distance will always be pulled out first. Line 7 checks to see whether a vertex pulled from the priority queue has already been visited and discard it if it has. This algorithm is just a concrete implementation of the previously described Dijkstra's algorithm.

Dijkstra variants. A variant of Dijkstra's algorithm is to decrease the priority of the neighbors instead of adding duplicates to the priority queue. This requires a more powerful priority queue that supports a `decreaseKey` function. Another variant is to instead of visiting just the closest vertex, to visit all the equally closest vertices on each round, as suggested by the more abstract algorithm. This

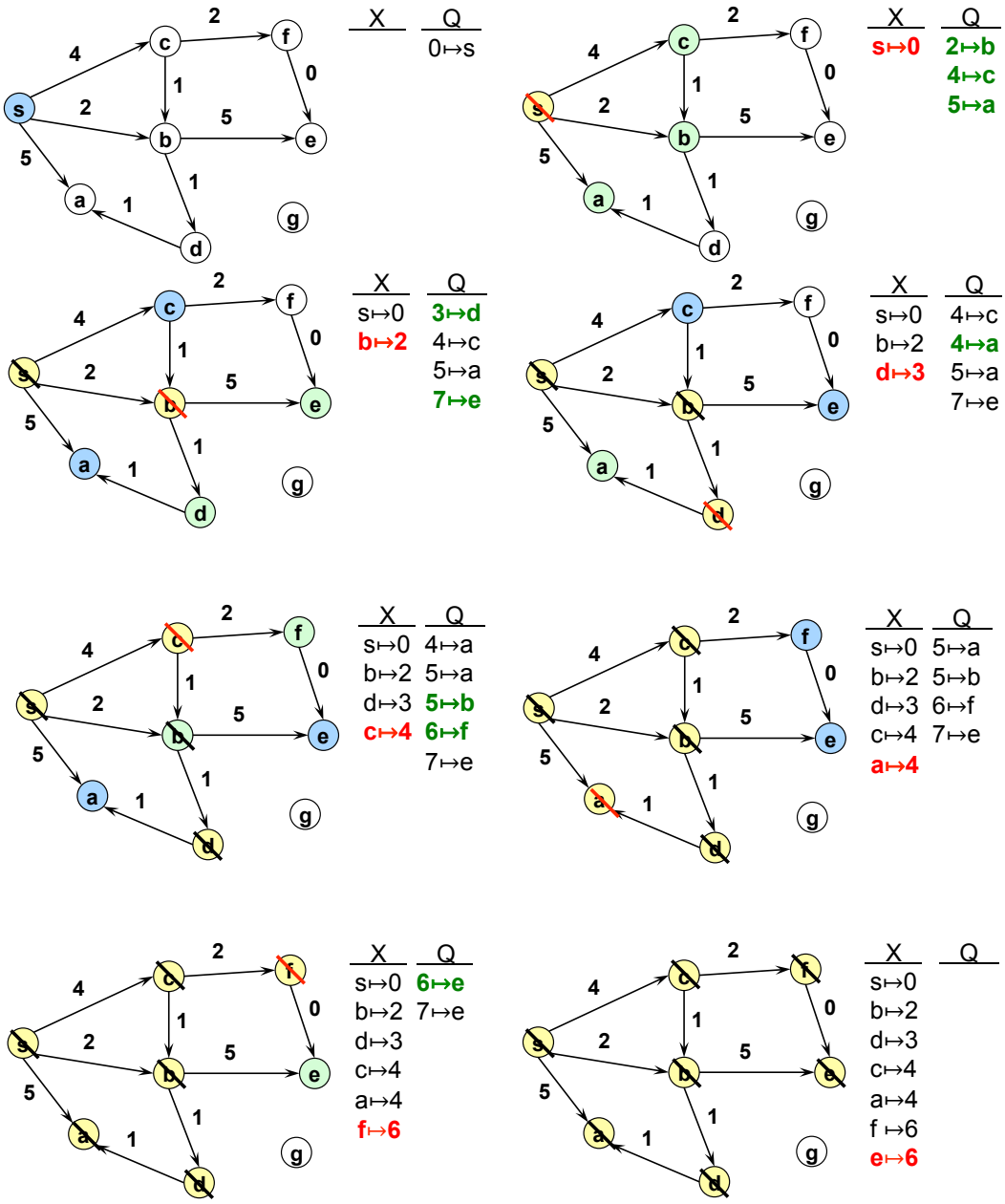


Figure 1: An example of Dijkstra's algorithm initialization and the end of each round. Notice that after visiting s, b, and d there are two distances in Q for a corresponding to the two paths from s to a; only the shortest distance will be added to X. In the next step, both f and b are added to Q, even though b is already visited. After visiting a, the redundant entries for a and b are discarded before visiting f. Finally, e is visited and the remaining element in Q is discarded. The vertex g is never visited as it is unreachable from s. If the priority queue supports a function that can return all minimum value keys, then the rounds for visiting c and a can be done in parallel, since they have the same shortest distance. Finally, notice that the distances in X never decrease.

variant has the advantage that the vertices can be visited in parallel. If all edges have unit weight, for example, this variant does the same as parallel BFS, visiting one level at a time. The amount of parallelism, however, depends on how many vertices have equal distance. This variant requires that the priority queue supports a function that returns all minimum valued keys.

Costs of Dijkstra's Algorithm. We now consider the cost of Dijkstra's algorithm by counting up the number of operations. In the code we have put a box around each operation. The `PQ.insert` in Line 14 is called only once, so we can ignore it. Of the remaining operations, The `iter` and $N_G(v)$ on line 11 are on the graph, Lines 7 and 9 are on the table of visited vertices, and Lines 4 and 10 are on the priority queue. For the priority queue operations, we have only discussed one cost model which, for a queue of size n, requires $O(\log n)$ work and span for each of `PQ.insert` and `PQ.deleteMin`. We have no need for a `meld` operation here. For the graph, we can either use a tree-based table or an array to access the neighbors[1] There is no need for single threaded array since we are not updating the graph. For the table of distances to visited vertices we can use a tree table, an array sequence, or a single threaded array sequences. The following table summarizes the costs of the operations, along with the number of calls made to each operation. There is no parallelism in the algorithm so we only need to consider the sequential execution of the calls.

Operation	Line	# of calls	PQ	Tree Table	Array	ST Array
`deleteMin`	4	$O(m)$	$O(\log m)$	-	-	-
`insert`	10	$O(m)$	$O(\log m)$	-	-	-
Priority Q total			$O(m \log m)$	-	-	-
`find`	7	$O(m)$	-	$O(\log n)$	$O(1)$	$O(1)$
`insert`	9	$O(n)$	-	$O(\log n)$	$O(n)$	$O(1)$
Distances total			-	$O(m \log n)$	$O(n^2)$	$O(m)$
$N_G(v)$	11	$O(n)$	-	$O(\log n)$	$O(1)$	-
`iter`	11	$O(m)$	-	$O(1)$	$O(1)$	-
Graph access total			-	$O(m + n \log n)$	$O(m)$	-

We can calculate the total number of calls to each operation by noting that the body of the let starting on Line 8 is only run once for each vertex. This means that Line 9 and $N_G(v)$ on Line 11 are only called $O(n)$ times. Everything else is done once for every edge.

Based on the table one should note that when using either tree tables or single threaded arrays the cost is no more than the cost of the priority queue operations. Therefore there is no asymptotic advantage of using one over the other, although there might be a constant factor speedup that is not insignificant. One should also note that using regular purely functional arrays is not a good idea since then the cost is dominated by the insertions and the algorithm runs in $\Theta(n^2)$ work.

The total work for Dijkstra's algorithm using a tree table $O(m \log m + m \log n + m + n \log n) = O(m \log n)$ since $m \leq n^2$.

[1] We could also use a hash table, but we have not yet discussed them.

4 SML code

Here we present the SML code for Dijkstra.

```
functor TableDijkstra(Table : TABLE) =
struct
  structure PQ = Default.RealPQ
  type vertex = Table.key
  type 'a table = 'a Table.table
  type weight = real
  type 'a pq = 'a PQ.pq
  type graph = (weight table) table

  (* Out neighbors of vertex v in graph G *)
  fun N(G : graph, v : vertex) =
    case (Table.find G v) of
      NONE => Table.empty()
    | SOME(ngh) => ngh

  fun Dijkstra(u : vertex, G : graph) =
    let
      val insert = Table.insert (fn (x,_) => x)

      fun Dijkstra'(Distances : weight table,
                    Q         : vertex pq) =
        case (PQ.deleteMin(Q)) of
            (NONE, _)       => Distances
          | (SOME(d, v), Q) =>
            case (Table.find Distances v) of

              (* if distance already set, then skip vertex *)
              SOME(_) => Dijkstra'(Distances, Q)

            | NONE =>
              let
                val Distances' = insert (v, d) Distances
                fun relax (Q,(u,l)) = PQ.insert (d+l, u) Q

                (* relax all the out edges *)
                val Q' = Table.iter relax Q (N(G,v))
              in
                Dijkstra'(Distances', Q')
              end
    in
      Dijkstra'(Table.empty(), PQ.singleton (0.0, u))
    end
end
```

Lecture 14 — Shortest Weighted Paths II

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — 11 October 2012

What was covered in this lecture:

- Continue Dijkstra's algorithm from last class
- Bellman Ford algorithm for graphs with negative weight edges – analysis of costs was covered in the next class.

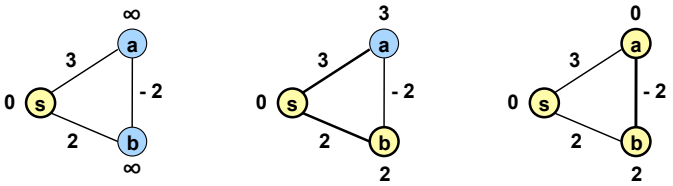
1 The Bellman Ford Algorithm

We now turn to solving the single source shortest path problem in the general case where we allow negative weights in the graph. One might ask how negative weights make sense. If talking about distances on a map, they probably do not, but various other problems reduce to shortest paths, and in these reductions negative weights show up. Before proceeding we note that if there is a negative weight cycle (the sum of weights on the cycle is negative) reachable from the source, then there cannot be a solution. This is because every time we go around the cycle we get a shorter path, so to find a shortest path we would just go around forever. In the case that a negative weight cycle can be reached from the source vertex, we would like solutions to the SSSP problem to return some indicator that such a cycle exists and terminate.

Exercise 1. *Consider the following* currency exchange *problem: given the a set currencies, a set of exchange rates between them, and a source currency s, find for each other currency v the best sequence of exchanges to get from s to v. Hint: how can you convert multiplication to addition.*

Exercise 2. *In your solution to the previous exercise can you get negative weight cycles? If so, what does this mean?*

So why is it that Dijkstra's algorithm does not work with negative edges? What is it in the proof of correctness that fails? Consider the following very simple example:



Dijkstra's algorithm would visit b then a and leave b with a distance of 2 instead of the correct -1. The problem is that it is no longer the case that if we consider the closest vertex v not in the visited

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

set, then its shortest path is through only the visited set and then extended by one edge out of the visited set to v.

So how can we find shortest paths on a graph with negative weights? As with most algorithms, we should think of some inductive hypothesis. In Dijkstra, the hypothesis was that if we have found the i nearest neighbors, then we can add one more to find the $i+1$ nearest neighbors. Unfortunately, as discussed, this does not work with negative weights, at least not in a simple way.

What other things can we try inductively? There are not too many choices. We could think about adding the vertices one by one in an arbitrary order. Perhaps we could show that if we have solved the problem for i vertices then we can add one more along with its edges and fix up the graph cheaply to get a solution for $i+1$ vertices. Unfortunately, this does not seem to work. Similarly, doing induction on the number of edges does not seem to work. You should think through these ideas and figure out why they don't work.

How about induction on the the number of edges in a path (i.e. the unweighted path length)? For this purpose we define the following

$$\delta_G^l(s,t) = \text{the shortest weighted path from } s \text{ to } t \text{ using at most } l \text{ edges.}$$

We can start by determining $\delta_G^0(s,v)$ for all $v \in V$, which is infinite for all vertices except s itself, for which it is 0. Then perhaps we can use this to determine $\delta_G^1(s,v)$ for all $v \in V$. In general we want to determine $\delta_G^{k+1}(s,v)$ based on all $\delta_G^k(s,v)$. The question is how do we calculate this. It turns out to be easy since to determine the shortest path with at most $k+1$ edges to a vertex v all that is needed is the shortest path with k edges to each of its in-neighbors and then to add in the length of the one additional edge. This gives us

$$\delta^{k+1}(v) = \min_{x \in N^-(v)} (\delta^k(v) + w(x,v)) .$$

Remember that $N^-(v)$ indicates the in-neighbors of vertex v.

Here is the Bellman Ford algorithm based on this idea. It assumes we have added a zero weight self loop on the source vertex.

```
1  % implements: the SSSP problem
2  fun BellmanFord(G = (V,E), s) =
3  let
4    % requires: all{D_v = δ^k_G(s,v) : v ∈ V}
5    fun BF(D,k) =
6      let
7        val D' = {v ↦ min_{u∈N⁻_G(v)}(D_u + w(u,v)) : v ∈ V}
8      in
9        if (k = |V|) then ⊥
10       else if (all{D_v = D'_v : v ∈ V}) then D
11       else BF(D', k+1)
12     end
13   val D = {v ↦ if v = s then 0 else ∞ : v ∈ V}
14 in BF(D,0) end
```

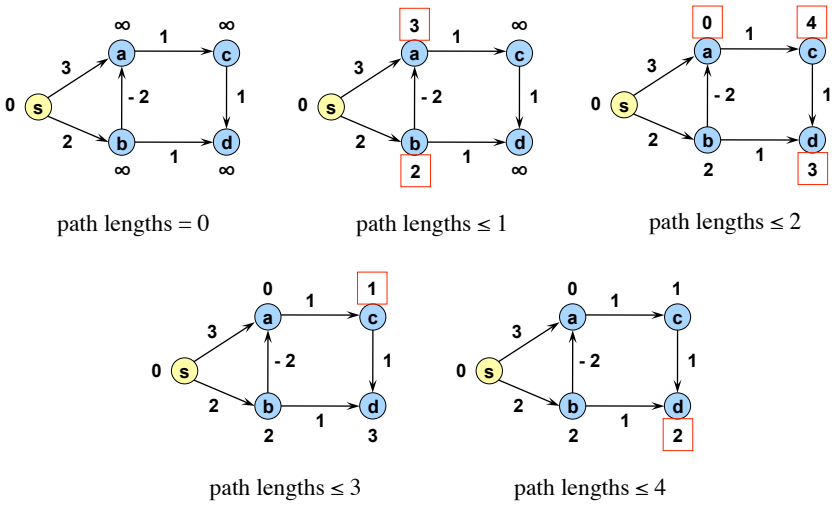


Figure 1: Steps of the Bellman Ford algorithm. The numbers with red squares indicate what changed on each step.

In Line 9 the algorithm returns $\perp$ (undefined) if there is a negative weight cycle reachable from s. In particular since no simple path can be longer than $|V|$, if the distance is still changing after $|V|$ rounds, then there must be a negative weight cycle that was entered by the search. An illustration of the algorithm over several steps is shown in Figure 1.

Theorem 1.1. *Given a directed weighted graph $G = (V, E, w)$, $w : E \to R$, and a source s, the* `BellmanFord` *algorithm returns the shortest path length from s to every vertex or indicates that there is a negative weight cycle in G reachable from s.*

Proof. By induction on the number of edges k in a path. The base case is correct since $D_s = 0$. For all $v \in V \setminus s$, on each step a shortest (s, v) path with up to $k+1$ edges must consist of a shortest (s, u) path of up to k edges followed by a single edge (u, v). Therefore if we take the minimum of these we get the overall shortest path with up to $k+1$ edges. For the source the self edge will maintain $D_s = 0$. The algorithm can only proceed to n rounds if there is a reachable negative-weight cycle. Otherwise a shortest path to every v is simple and can consist of at most n vertices and hence $n-1$ edges. □

Cost of Bellman Ford. We now analyze the cost of the algorithm. First we assume the graph is represented as a table of tables, as we suggested for the implementation of Dijkstra. We then consider representing it as a sequence of sequences.

For a table of tables we assume the graph G is represented as a ($\mathbb{R}$ `vtxTable`) `vtxTable`, where `vtxTable` maps vertices to values. The $\mathbb{R}$ are the real valued weights on the edges. We assume the distances D are represented as a $\mathbb{R}$ `vtxTable`. Let's consider the cost of one call to BF, not including the recursive calls. The only nontrivial computations are on lines 7 and 10. Line 7 consists of a tabulate over the vertices. As the cost specification for tables indicate, to calculated the work we take the sum of the work for each vertex, and for the span we take the maximum of the spans, and add

$O(\log n)$. Now consider what the algorithm does for each vertex. First it has to find the neighbors in the graph (using a `find G v`). This requires $O(\log |V|)$ work and span. Then it involves a map over the neighbors. Each instance of this map requires a find in the distance table to get D_u and an addition of the weight. The find takes $O(\log |V|)$ work and span. Finally there is a reduce that takes $O(1 + |N_G(v)|)$ work and $O(\log |N_G(v)|)$ span. Using $n = |V|$ and $m = |E|$, the overall work and span are therefore

$$\begin{aligned} W &= O\left(\sum_{v \in V}\left(\log n + |N_G(v)| + \sum_{u \in N_G(v)}(1 + \log n)\right)\right) \\ &= O\left((n+m)\log n\right) \\ S &= O\left(\max_{v \in V}\left(\log n + \log |N_G(v)| + \max_{u \in N(v)}(1 + \log n)\right)\right) \\ &= O(\log n) \end{aligned}$$

Line 10 is simpler to analyze since it only involves a tabulate and a reduction. It requires $O(n \log n)$ work and $O(\log n)$ span.

Now the number of calls to BF is bounded by n, as discussed earlier. These calls are done sequentially so we can multiply the work and span for each call by the number of calls giving:

$$\begin{aligned} W(n,m) &= O(nm \log n) \\ S(n,m) &= O(n \log n) \end{aligned}$$

Cost of Bellman Ford using Sequences If we assume the vertices are the integers $\{0, 1, \ldots, |V|-1\}$ then we can use array sequences to implement a `vtxTable`. Instead of using a `find`, which requires $O(\log n)$ work, we can use `nth` requiring only $O(1)$ work. This improvement in costs can be applied for looking up in the graph to find the neighbors of a vertex, and looking up in the distance table to find the current distance. By using the improved costs we get:

$$\begin{aligned} W &= O\left(\sum_{v \in V}\left(1 + |N_G(v)| + \sum_{u \in N_G(v)} 1\right)\right) \\ &= O(m) \\ S &= O\left(\max_{v \in V}\left(1 + \log |N_G(v)| + \max_{u \in N(v)} 1\right)\right) \\ &= O(\log n) \end{aligned}$$

and hence the overall complexity for `BellmanFord` with array sequences is:

$$\begin{aligned} W(n,m) &= O(nm) \\ S(n,m) &= O(n \log n) \end{aligned}$$

By using array sequences we have reduced the work by a $O(\log n)$ factor.

Lecture 15 — Probability and Randomized Algorithms

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — October 16, 2012

What was covered in this lecture:

- Finish Bellman Ford from last class
- Introduction to randomized algorithms
- Probability theory review
- A randomized algorithm for finding the two largest values – continued in next class

1 Randomized Algorithms Introduction

The main theme of this lecture is *randomized algorithms*. These are algorithms that make use of randomness in their computation. We will begin this lecture with a simple question:

> **Question:** How many comparisons do we need to find the second largest number in a sequence of n distinct numbers?

Without the help of randomization, there is a naïve algorithm that requires about $2n-3$ comparisons and there is a divide-and-conquer solution that needs about $3n/2$ comparisons. With the aid of randomization, there is a simple randomized algorithm that uses only $n-1+2\log n$ comparisons on average, under some notion of averaging. In probability speak, this is $n-1+2\log n$ comparisons in expectation. We'll develop the machinery needed to design and analyze this algorithm.

By the end of this lecture, you will be well-equipped to analyze the a randomize d algorithm for finding the median value in a sequence of numbers, or in fact more generally the k^{th} smallest value for any k. This can be done expected $O(n)$ work and $O(\log^2 n)$ span.

2 Discrete Probability: Let's Toss Some Dice

Hopefully you have all seen some probability before. Here we will review some basic definitions of discrete probability before proceeding into randomized algorithms. We begin with an example. Suppose we have two *fair* dice, meaning that each is equally likely to land on any of its six sides. If

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

we toss the dice, what is the chance that their numbers sum to 4? You can probably figure out that the answer is

$$\frac{\#\text{ of outcomes that sum to 4}}{\#\text{ of total possible outcomes}} = \frac{3}{36} = \frac{1}{12}$$

since there are three ways the dice could sum to 4 (1 and 3, 2 and 2, and 3 and 1), out of the $6 \times 6 = 36$ total possibilities. Such throwing of dice is called a *probabilistic experiment* since it is an "experiment" (we can repeat it many time) and the outcome is probabilistic (each experiment might lead to a different outcome).

It is often useful to calculate various properties of a probabilistic experiments, such as averages of outcomes or how often some property is true. For this we use probability theory, which we loosely formalize here.

Sample space. A *sample space* Ω is an arbitrary and possibly infinite (but countable) set of possible outcomes of a probabilistic experiment. In our coin tossing example the Ω is the set of 36 possible outcomes of tossing the two coins: $\{(1,1),(1,2),\ldots,(2,1),\ldots,(6,6)\}$.

Events and Primitive Events. A *primitive event* (or *sample point*) of a sample space Ω is any one of its elements, i.e. any one of the outcomes of the random experiment. For example it could be the coin toss $(1,4)$. An *event* is any subset of Ω and typically represents some property common to multiple primitive events. For example an event could be "the first die is 3" which would correspond to the set $\{(3,1),(3,2),(3,3),(3,4),(3,5),(3,6)\}$, or it could be "the dice sum to 4", which would correspond to the set $\{(1,3),(2,2),(3,1)\}$.

Probability Function. A *probability function* $\mathbf{Pr} : \Omega \to [0,1]$ is associated with the sample space with the condition that $\sum_{e\in\Omega} \mathbf{Pr}[e] = 1$ (i.e. the probabilities of the outcomes of a probabilistic experiment sum to 1, as we would expect). The probability of an event A is simply the sum of the probabilities of its primitive events: $\mathbf{Pr}[A] = \sum_{e\in A} \mathbf{Pr}[e]$. For example in tossing two dice, the probability of the event "the first die is 3" is $\frac{6}{36} = \frac{1}{6}$, and the probability of the event "the dice sum to 4" is $\frac{3}{36} = \frac{1}{12}$ (as above). With fair dice all probabilities are equal so all we have to do is figure out the size of each set and divide it by $|\Omega|$. In general this might not be the case. We will use $(\Omega, \mathbf{Pr})$ to indicate the sample space along with its probability function.

Random variables and Indicator Random Variables. A *random variable* X on a sample space Ω is a real valued function on Ω, thus having type: $X : \Omega \to \mathbb{R}$. For a sample space there can be many random variables each keeping track of some quantity of a probabilistic experiment. For example for the two dice example, we could have a random variable X representing the sum of the two rolls (**fun** $X(d_1,d_2) = d_1 + d_2$) or a random variable Y that is 1 if the values on the dice are the same and 0 otherwise (**fun** $Y(d_1,d_2) =$ **if** $(d_1 = d_2)$ **then** 1 **else** 0).

This second random variable Y is called an *indicator random variable* since it takes on the value 1 when some condition is true and 0 otherwise. In particular, for a predicate $p : \Omega \to$ `bool` a random indicator variable is defined as **fun** $Y(e) =$ **if** $p(e)$ **then** 1 **else** 0.

For a random variable X and a number $a \in \mathbb{R}$, the event "$X = a$" is the set $\{\omega \in \Omega \mid X(\omega) = a\}$. We typically denote random variables by capital letters from the end of the alphabet, e.g. X, Y Z.

Expectation. The *expectation* of a random variable X given a sample space $(\Omega, \mathbf{Pr})$ is the sum of the random variable over the primitive events weighted by their probability, specifically:

$$\mathop{\mathbf{E}}_{\Omega,\mathbf{Pr}}[X] = \sum_{e \in \Omega} X(e) \cdot \mathbf{Pr}[e] \ .$$

The expectation can be though of as a higher order function that takes in the random variable function as an argument:

$$\textbf{fun } \mathbf{E}\ (\Omega, \mathbf{Pr})\ X = \texttt{reduce}\ +\ 0\ (\texttt{map}\ (\textbf{fn}\ e \Rightarrow X(e) \times \mathbf{Pr}[e])\ \Omega)$$

We note that it is often not practical to compute expectations directly since the sample space can be exponential in the size of the input, or even countably infinite. We therefore resort to more clever tricks. In the notes we will most often drop the $(\Omega, \mathbf{Pr})$ subscript on $\mathbf{E}$ since it is clear from the context.

The expectation of a indicator random variable Y is the probability that the associated predicate p is true. This is because

$$\mathbf{E}[Y] = \sum_{e \in \Omega} (\textbf{if } p(e) \textbf{ then } 1 \textbf{ else } 0) \cdot \mathbf{Pr}[e] = \sum_{e \in \Omega, p(e) = \texttt{true}} \mathbf{Pr}[e] = \mathbf{Pr}\left[\{e \in \Omega \mid p(e)\}\right] \ .$$

Independence. Two events A and B are *independent* if the occurrence of one does not affect the probability of the other. This is true if and only if $\mathbf{Pr}[A \cap B] = \mathbf{Pr}[A] \cdot \mathbf{Pr}[B]$. For our dice throwing example, the events $A = \{(d_1, d_2) \in \Omega \mid d_1 = 1\}$ (the first die is 1) and $B = \{(d_1, d_2) \in \Omega \mid d_2 = 1\}$ (the second die is 1) are independent since $\mathbf{Pr}[A] = \mathbf{Pr}[B] = \frac{1}{6}$ and $\mathbf{Pr}[A \cap B] = \frac{1}{36}$. However the event $C = \{(d_1, d_2) \in \Omega \mid d_1 + d_2 = 4\}$ (the dice add to 4) is not independent of A since $\mathbf{Pr}[C] = \frac{1}{12}$ and $\mathbf{Pr}[A \cap C] = \mathbf{Pr}[\{(1,1)\}] = \frac{1}{36} \neq \mathbf{Pr}[A] \cdot \mathbf{Pr}[C] = \frac{1}{6} \cdot \frac{1}{12} = \frac{1}{72}$. They are not independent since the fact that the first die is 1 increases the probability they sum to 4 (from $\frac{1}{12}$ to $\frac{1}{6}$), or the other way around, the fact that they sum to 4 increases the probability the first die is 1 (from $\frac{1}{6}$ to $\frac{1}{3}$).

When we have multiple events, we say that $A_1, \ldots, A_k$ are *mutually independent* if and only if for any non-empty subset $I \subseteq \{1, \ldots, k\}$,

$$\mathbf{Pr}\left[\bigcap_{i \in I} A_i\right] = \prod_{i \in I} \mathbf{Pr}[A_i] \ .$$

Two random variables X and Y are independent if fixing the value of one does not affect the probability distribution of the other. This is true if and only if for every a and b the events $\{X \leq a\}$ and $\{Y \leq b\}$ are independent. In our two dice example, a random variable X representing the value of the first die and a random variable Y representing the value of the second die are independent. However X is not independent of a random variable Z representing the sum of the values of the two dice.

2.1 Linearity of Expectations

One of the most important theorem in probability is *linearity of expectations*. It says that given two random variables X and Y, $\mathbf{E}[X] + \mathbf{E}[Y] = \mathbf{E}[X + Y]$. If we write this out based on the definition of expectations we get:

$$\sum_{e \in \Omega} \mathbf{Pr}[e] X(e) + \sum_{e \in \Omega} \mathbf{Pr}[e] Y(e) = \sum_{e \in \Omega} \mathbf{Pr}[e] (X(e) + Y(e))$$

The algebra to show this is true is straightforward. The linearity of expectations is very powerful often greatly simplifying analysis.

To continue our running example, lets consider analyzing the expected sum of values when throwing two dice. One way to calculate this is to consider all 36 possibilities and take their average. What is this average? A much simpler way is to sum the expectation for each of the two dice. The expectation for either dice is the average of just the six possible values $1, 2, 3, 4, 5, 6$, which is 3.5. Therefore the sum of the expectations is 7.

Note that for a binary function f the equality $f(\mathbf{E}[X], \mathbf{E}[Y]) = \mathbf{E}[f(X, Y)]$ is **not** true in general. For example $\max(\mathbf{E}[X], \mathbf{E}[Y]) \neq \mathbf{E}[\max(X, Y)]$. If it the equality held, the expected maximum value of two rolled dice would be 3.5.

Exercise 1. *What is the expected maximum value of throwing two dice?*

We note that $\mathbf{E}[X] \times \mathbf{E}[Y] = \mathbf{E}[X \times Y]$ is true if X and Y are independent. The expected value of the product of the values on two dice is therefore $3.5 \times 3.5 = 12.25$.

2.2 More Examples

Example 2.1. *Suppose we toss n coins, where each coin has a probability p of coming up heads. What is the expected value of the random variable X denoting the total number of heads?*

> **Solution I:** We'll apply the definition of expectation directly. This will rely on some messy algebra and useful equalities you might or might not know, but don't fret since this not

the way we suggest you do it.

$$\begin{aligned}
\mathbf{E}[X] &= \sum_{k=0}^{n} k \cdot \mathbf{Pr}[X = k] \\
&= \sum_{k=1}^{n} k \cdot p^k (1-p)^{n-k} \binom{n}{k} \\
&= \sum_{k=1}^{n} k \cdot \frac{n}{k} \binom{n-1}{k-1} p^k (1-p)^{n-k} && \text{[because } \binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} \text{]} \\
&= n \sum_{k=1}^{n} \binom{n-1}{k-1} p^k (1-p)^{n-k} \\
&= n \sum_{j=0}^{n-1} \binom{n-1}{j} p^{j+1} (1-p)^{n-(j+1)} && \text{[because } k = j+1 \text{]} \\
&= np \sum_{j=0}^{n-1} \binom{n-1}{j} p^{j} (1-p)^{(n-1)-j} \\
&= np(p + (1-p))^n && \text{[Binomial Theorem]} \\
&= np
\end{aligned}$$

That was pretty tedious :(

Solution II: We'll use linearity of expectations. Let $X_i = \mathbb{I}\{i\text{-th coin turns up heads}\}$. That is, it is 1 if the i-th coin turns up heads and 0 otherwise. Clearly, $X = \sum_{i=1}^{n} X_i$. So then, by linearity of expectations,

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbf{E}[X_i].$$

What is the probability that the i-th coin comes up heads? This is exactly p, so $\mathbf{E}[X] = 0 \cdot (1-p) + 1 \cdot p = p$, which means

$$\mathbf{E}[X] = \sum_{i=1}^{n} \mathbf{E}[X_i] = \sum_{i=1}^{n} p = np.$$

Example 2.2. *A coin has a probability p of coming up heads. What is the expected value of Y representing the number of flips until we see a head? (The flip that comes up heads counts too.)*

Solution I: We'll directly apply the definition of expectation:

$$\begin{aligned}
\mathbf{E}[Y] &= \sum_{k \geq 1} k(1-p)^{k-1} p \\
&= p \sum_{k=0}^{\infty} (k+1)(1-p)^k \\
&= p \cdot \frac{1}{p^2} && \text{[by Wolfram Alpha, though you should be able to do it.]} \\
&= 1/p
\end{aligned}$$

Solution II: Alternatively, we'll write a recurrence for it. As it turns out, we know that with probability p, we'll get a head and we'll be done—and with probability $1-p$, we'll get a tail and we'll go back to square one:

$$\mathbf{E}[Y] = p \cdot 1 + (1-p)\Big(1 + \mathbf{E}[Y]\Big) = 1 + (1-p)\mathbf{E}[Y] \implies \mathbf{E}[Y] = 1/p.$$

by solving for $\mathbf{E}[Y]$ in the above equation.

3 Finding The Top Two

The top-two problem is to find the two largest elements from a sequence of n (unique) numbers. For inspiration, we'll go back and look at the naïve algorithm for the problem:

```
1  fun max2(S) = let
2    fun replace((m1, m2), v) =
3      if v ≤ m2 then (m1, m2)
4      else if v ≤ m1 then (m1, v)
5      else (v, m1)
6    val start = if S1 ≥ S2 then (S1, S2) else (S2, S1)
7  in iter replace start S⟨3, ..., n⟩
8  end
```

We assume S is indexed from 1 to n. In the following analysis, we will be meticulous about constants. The naïve algorithm requires up to $1 + 2(n-2) = 2n-3$ comparisons since there is one comparison to compute `start` each of the $n-2$ `replace`s requires up to two comparisons. On the surface, this may seem like the best one can do. Surprisingly, there is a divide-and-conquer algorithm that uses only about $3n/2$ comparisons (exercise to the reader). More surprisingly still is the fact that it can be done in $n + O(\log n)$ comparisons. But how?

Puzzle: How would you solve this problem using only $n + O(\log n)$ comparisons?

A closer look at the analysis above reveals that we were pessimistic about the number of comparisons; not all element will get past the "if" statement in Line 3; therefore, only some of the elements will need the comparison in Line 4. But we didn't know how many of them, so we analyzed it in the worst possible scenario.

Let's try to understand what's happening better by looking at the worst-case input. Can you come up with an instance that yields the worst-case behavior? It is not difficult to convince yourself that there is a sequence of length n that causes this algorithm to make $2n-3$ comparisons. In fact, the instance is simple: an increasing sequence of length n, e.g., $\langle 1, 2, 3, \ldots, n\rangle$. As we go from left to right, we find a new maximum every time we counter a new element—this new element gets compared in both Lines 3 and 4.

But perhaps it's unlikely to get such a nice—but undesirable—structure if we consider the elements in random order. With only 1 in $n!$ chance, this sequence will be fully sorted. You can work

out the probability that the random order will result in a sequence that looks "approximately" sorted, and it would not be too high. Our hopes are high that *we can save a lot of comparisons in Line 4 by considering elements in random order.*

The algorithm we'll analyze is the following. On input a sequence S of n elements:

1. Let $T = \texttt{permute}(S, \pi)$, where π is a random permutation (i.e., we choose one of the $n!$ permutations).
2. Run the naïve algorithm on T.

Remarks: We don't need to explicitly construct T. We'll simply pick a random element which hasn't been considered and consider that element next until we are done looking at the whole sequence. For the analysis, it is convenient to describe the process in terms of T.

In reality, the performance difference between the $2n-3$ algorithm and the $n-1+2\log n$ algorithm is unlikely to be significant—unless the comparison function is super expensive. For most cases, the $2n-3$ algorithm might in fact be faster to due better cache locality.

The point of this example is to demonstrate the power of randomness in achieving something that otherwise seems impossible—more importantly, the analysis hints at why on a typical "real-world" instance, the $2n-3$ algorithm does much better than what we analyzed in the worst case (real-world instances are usually not adversarial).

3.1 Analysis

Let X_i be an indicator variable denoting whether Line 4 gets executed for this particular value of i (i.e., Did T_i get compared in Line 4?) That is, $X_i = 1$ if T_i is compared in Line 4 and 0 otherwise. Therefore, the total number of comparisons is

$$Y = \underbrace{1}_{\text{Line 6}} + \underbrace{n-2}_{\text{Line 3}} + \underbrace{\sum_{i=3}^{n} X_i}_{\text{Line 4}};$$

This expression in true regardless of the random choice we're making. We're interested in computing the expected value of Y where the random choice is made when we choose a permutation. By linearity of expectation, we have

$$\begin{aligned} \mathbf{E}[Y] &= \mathbf{E}\left[1+(n-2)+\sum_{i=3}^{n} X_i\right] \\ &= 1+(n-2)+\sum_{i=3}^{n} \mathbf{E}\left[X_i\right]. \end{aligned}$$

Our tasks therefore boils down to computing $\mathbf{E}\left[X_i\right]$ for $i = 3, \ldots, n$. To compute this expectation, we ask ourselves: *What is the probability that $T_i > m_2$?* A moment's thought shows that the condition $T_i > m_2$ holds exactly when T_i is either the largest element or the second largest element

in $\{T_1, \ldots, T_i\}$. So ultimately we're asking: what is the probability that T_i is the largest or the second largest element in randomly-permuted sequence of length i?

To compute this probability, we note that each element in the sequence is equally likely to be anywhere in the permuted sequence (we chose a random permutation. In particular, if we look at the k-th largest element, it has $1/i$ chance of being at T_i. (You should also try to work it out using a counting argument.) Therefore, the probability that T_i is the largest or the second largest element in $\{T_1, \ldots, T_i\}$ is $\frac{1}{i} + \frac{1}{i} = \frac{2}{i}$, so

$$\mathbf{E}\left[X_i\right] = 1 \cdot \tfrac{2}{i} = 2/i.$$

Plugging this into the expression for $\mathbf{E}[Y]$, we get

$$\begin{aligned}
\mathbf{E}[Y] &= 1 + (n-2) + \sum_{i=3}^{n} \mathbf{E}\left[X_i\right] \\
&= 1 + (n-2) + \sum_{i=3}^{n} \frac{2}{i} \\
&= 1 + (n-2) + 2\left(\tfrac{1}{3} + \tfrac{1}{4} + \ldots \tfrac{1}{n}\right) \\
&= n - 4 + 2\left(1 + \tfrac{1}{2} + \tfrac{1}{3} + \tfrac{1}{4} + \ldots \tfrac{1}{n}\right) \\
&= n - 4 + 2H_n,
\end{aligned}$$

where H_n is the n-th Harmonic number. But we know that $H_n \leq 1 + \log_2 n$, so we get $\mathbf{E}[Y] \leq n - 2 + 2\log_2 n$. We could also use the following sledgehammer:

As an aside, the Harmonic sum has the following nice property:

$$H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n} = \ln n + \gamma + \varepsilon_n,$$

where γ is the Euler-Mascheroni constant, which is approximately $0.57721\cdots$, and $\varepsilon_n \sim \frac{1}{2n}$, which tends to 0 as n approaches ∞. This shows that the summation and integral of $1/i$ are almost identical (up to a small adative constant and a low-order vanishing term).

4 Finding The k^{th} Smallest Element

Consider the following problem:

> **Input:** S — a sequence of n numbers (not necessarily sorted)
> **Output:** the k^{th} smallest value in S (i.e. `(nth (sort S) k)`).
> *Requirement:* $O(n)$ expected work and $O(\log^2 n)$ span.

Note that the linear-work requirement rules out the possibility of sorting the sequence. Here's where the power of randomization gives a simple algorithm.

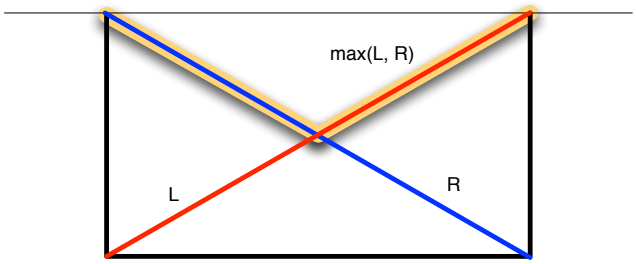
```
1  fun kthSmallest(k, S) = let
2        val p = a value from S picked uniformly at random
3        val L = ⟨ x ∈ S | x < p ⟩
4        val R = ⟨ x ∈ S | x > p ⟩
5    in  if (k < |L|) then kthSmallest(k, L)
6        else if (k < |S| − |R|) then p
7        else kthSmallest(k − (|S| − |R|), R)
```

We'll try to analyze the work and span of this algorithm. Let $X_n = \max\{|L|, |R|\}$, which is the size of the larger side. Notice that X_n is an upper bound on the size of the side the algorithm actually recurses into. Now since Step 3 is simply two `filter` calls, we have the following recurrences:

$$\begin{aligned} W(n) &= W(X_n) + O(n) \\ S(n) &= S(X_n) + O(\log n) \end{aligned}$$

Let's first look at the work recurrence. Specifically, we are interested in $\mathbf{E}[W(n)]$. First, let's try to get a sense of what happens in expectation.

How big is $\mathbf{E}[X_n]$*?* To understand this, let's take a look at a pictorial representation:



The probability that we land on a point on the curve is $1/n$, so

$$\mathbf{E}[X_n] = \sum_{i=1}^{n-1} \max\{i, n-i\} \cdot \tfrac{1}{n} \leq \sum_{j=n/2}^{n-1} \frac{2}{n} \cdot j \leq \frac{3n}{4}$$

(Recall that $\sum_{i=a}^{b} i = \frac{1}{2}(a+b)(b-a+1)$.)

Aside: This is a counterexample showing that $\mathbf{E}[\max\{X, Y\}] \neq \max\{\mathbf{E}[X], \mathbf{E}[Y]\}$.

This computation tells us that in expectation, X_n is a constant fraction smaller than n, so we should have a nice geometrically decreasing sum, which works out to $O(n)$. Let's make this idea concrete: Suppose we want each recursive call to work with a constant fraction fewer elements than before, say at most $\frac{3}{4}n$.

What's the probability that $\mathbf{Pr}\left[X_n \leq \frac{3}{4}n\right]$*?* Since $|R| = n - |L|$, $X_n \leq \frac{3}{4}n$ if and only if $n/4 < |L| \leq 3n/4$. There are $3n/4 - n/4$ values of p that satisfy this condition. As we pick p uniformly at random, this probability is

$$\frac{3n/4 - n/4}{n} = \frac{n/2}{n} = \frac{1}{2}.$$

Notice that given an input sequence of size n, how the algorithm performs in the future is irrespective of what it did in the past. Its cost from that point on only depends on the random choice it makes after that. So, we'll let $\overline{W}(n) = \mathbf{E}[W(n)]$ denote the expected work performed on input of size n.

Now by the definition of expectation, we have

$$\begin{aligned}
\overline{W}(n) &\leq \sum_i \mathbf{Pr}\left[X_n = i\right] \cdot \overline{W}(i) + c \cdot n \\
&\leq \mathbf{Pr}\left[X_n \leq \tfrac{3n}{4}\right] \overline{W}(3n/4) + \mathbf{Pr}\left[X_n > \tfrac{3n}{4}\right] \overline{W}(n) + c \cdot n \\
&= \tfrac{1}{2}\overline{W}(3n/4) + \tfrac{1}{2}\overline{W}(n) + c \cdot n \\
&\Longrightarrow (1 - \tfrac{1}{2})\overline{W}(n) = \tfrac{1}{2}\overline{W}(3n/4) + c \cdot && \text{[collecting similar terms]} \\
&\Longrightarrow \overline{W}(n) \leq \overline{W}(3n/4) + 2c \cdot n. && \text{[multiply by 2]}
\end{aligned}$$

In this derivation, we made use of the fact that with probability $1/2$, the instance size shrinks to at most $3n/4$—and with probability $1/2$, the instance size is still larger than $3n/4$, so we pessimistically upper bound it with n. Note that the real size might be smaller, but we err on the safe side since we don't have a handle on that.

Finally, the recurrence $\overline{W}(n) \leq \overline{W}(3n/4) + 2cn$ is root dominated and therefore solves to $O(n)$.

4.1 Span

Let's now turn to the span analysis. We'll apply the same strategy as what we did for the work recurrence. We have already established the span recurrence:

$$S(n) = S(X_n) + O(\log n)$$

where X_n is the size of the larger side, which is an upper bound on the size of the side the algorithm actually recurses into. Let $\overline{S}(n)$ denote $\mathbf{E}[S(n)]$. As we observed before, how the algorithm performs in the future is irrespective of what it did in the past. Its cost from that point on only depends on the random choice it makes after that. So then, by the definition of expectation, we have

$$\begin{aligned}
\overline{S}(n) =& \leq \sum_i \mathbf{Pr}\left[X_n = i\right] \cdot \overline{W}(i) + c \log n \\
&\leq \mathbf{Pr}\left[X_n \leq \tfrac{3n}{4}\right] \overline{S}(3n/4) + \mathbf{Pr}\left[X_n > \tfrac{3n}{4}\right] \overline{S}(n) + c \cdot \log n \\
&\leq \tfrac{1}{2}\overline{S}(3n/4) + \tfrac{1}{2}\overline{S}(n) + c \cdot \log n \\
&\Longrightarrow (1 - \tfrac{1}{2})\overline{S}(n) \leq \tfrac{1}{2}\overline{S}(3n/4) + c \log n \\
&\Longrightarrow \overline{S}(n) \leq \overline{S}(3n/4) + 2c \log n,
\end{aligned}$$

which we know is balanced and solves to $O(\log^2 n)$.

Lecture 16 — Graph Contraction and Connectivity

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — October 18, 2012

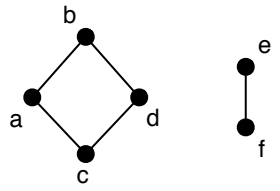
What was covered in this lecture:

- Continued randomize algorithms from last class: finished `max2` and covered `kthSmallest`
- Introduction to graph contraction and connectivity.

1 Graph Contraction

Until recently, we have been talking mostly about techniques for solving problems on graphs that were developed in the context of sequential algorithms. Some of them are easy to parallelize while others are not. For example, we saw there is parallelism in BFS because each level can be explored in parallel, assuming the number of levels is not too large. But there was no parallelism in DFS. There was also no parallelism in the version of Dijkstra's algorithm we discussed, which used priority first search.[1] There was plenty of parallelism in the Bellman-Ford algorithm. We are now going to discuss a technique called "graph contraction" that you can add to your toolbox for parallel algorithms. The technique will use randomization.

Graph Connectivity. To motivate graph contraction consider the graph connectivity problem. Two vertices are connected in an undirected graph if there is a path between them. A graph is connected if every pair if vertices is connected. The *graph connectivity* problem is to partition an undirected graph into its maximal connected subgraphs. By partition we mean that every vertex has to belong to one of the subgraphs and by maximal we mean that no connected vertex can be added to any partition—i.e. there are no edges between the partitions. For example for the following graph:



graph connectivity will return the two subgraphs consisting of the vertices $\{a, b, c, d\}$ and $\{e, f\}$.

How might we solve the graph connectivity problem? We could do it with a bunch of graph searches. In particular we can start at any vertex and search all vertices reachable from it to create

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

[1] In reality, there is some parallelism in both DFS and Dijkstra when graphs are dense—in particular, although vertices need to visited sequentially the edges can be processed in parallel. If we have time, we will get back to this when we cover priority queues in more detail.

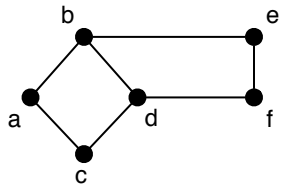
the first component, then move onto the next vertex and if it has not already been searched search from it to create the second component. We then repeat until all vertices have been checked. Either BFS or DFS can be used for the individual searches. This is a perfectly sensible sequential algorithm. However, it has two shortcomings from a parallelism perspective even if we use a parallel BFS. Firstly each parallel BFS takes span proportional to the diameter of the component (the longest distance between two vertices) and in general this could be as large as n. Secondly even if each component is has a small diameter, we have to iterate over the components one by one.

We are therefore interested in an approach that can identify the components in parallel. Furthermore the span should be independent of the diameter, and ideally polylogarithmic in $|V|$. To do this we will give up on the idea of graph search since it seems to be inherently limited by the diameter of a graph. Instead we will borrow some ideas from our algorithm for the `scan` operation. In particular we will resort to *contraction*—on each round of contraction we will shrink the size of the graph by a constant fraction and then solve the connectivity problem on the contracted graph. Contracting a graph, however, is a bit more complicated than just pairing up the odd and even positioned values as we did in the algorithm for `scan`.

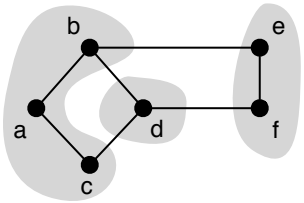
This approach is called *graph contraction*. It is a reasonably simple technique and can be applied to a variety of problems, beyond just connectivity, including spanning trees and minimum spanning trees. We assume the graph is undirected. Lets start by considering a function:

```
contract : graph → partition
```

which takes a undirected graph $G = (V, E)$ and returns a partitioning of the vertices of V into connected subgraphs, but not necessarily maximally connected subgraphs (i.e. there can still be edges between the partitions). For example `contract` on the following graph:

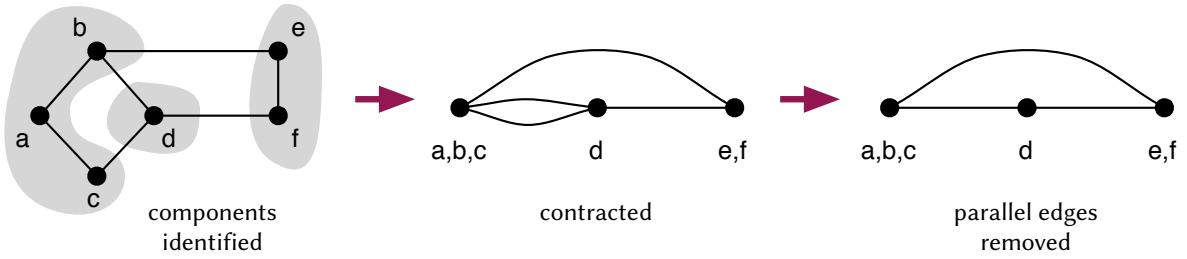


might return the partitioning $\{\{a, b, c\}, \{d\}, \{e, f\}\}$ as indicated by the following shaded regions:



Note that each of the three partitions is connected by edges within the partition. The `contract` function would not return $\{\{a, b, f\}, \{c, d\}, \{e\}\}$, for example, since the subgraph $\{a, b, f\}$ is not connected by edges within the component.

We will return to how to implement `contract`, but given that the partitions returned by `contract` are not maximal (i.e. there are still edges between them), how might we use `contract` to solve the graph connectivity problem. As we already suggested, we can apply it recursively. To do this on each round we replace each partition with a single vertex and then relabel all the edges with the new vertex name. This can be illustrated as follows:

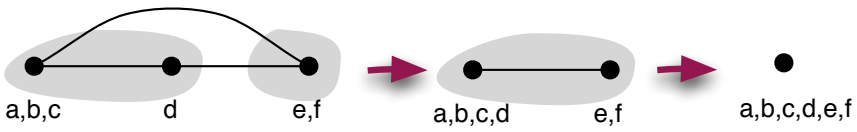


After contracting, we are left with a triangle. Note that in the intermediate step, when we join a, b, c, we create redundant edges to d (both b and c had an original edge to d). We therefore replace these with a single edge. However, depending on the particular use, in some algorithms, it is convenient to allow parallel (redundant) edges rather than going to the work of removing them. Also note that some edge are within a partition. These become self loops, which we drop.

It should be clear that if the graph actually contracts on each round (not all vertices are singleton partitions) then eventually every maximal connected component in the graph will shrink down to a single vertex. This leads to the following high-level algorithm for connectivity:

```
while there are edges left:
  contract the graph
  update the edges to use remaining vertices and remove self loops
```

Continuing with our example, after the first contraction shown earlier we might make the following two contractions in the next two rounds, at which point we have no more edges:



Note that if the input is not connected we would be left with multiple vertices when the algorithm finishes, one per component.

To make this algorithm more concrete we need some representation for the partitions. One way to do this is to use one of the original vertices of each partition as a representative and then keep a mapping from each vertex to this representative. Therefore the partition $\{\{a, b, c\}, \{d\}, \{e, f\}\}$ could be represented as $(\{a, d, e\}, \{a \mapsto a, b \mapsto a, c \mapsto a, d \mapsto d, e \mapsto e, f \mapsto e\})$. With this representation first consider the following more concrete algorithm that counts the number of maximal connected components.

```
1  fun numComponents((V, E), i) =
2  if |E| = 0 then |V|
3  else let
4    val (V', P) = contract((V, E), i)
5    val E' = {(P[u], P[v]) : (u, v) ∈ E | P[u] ≠ P[v]}
6  in
7    numComponents((V', E'), i)
8  end
```

The i is just the round number—as we will see it will be useful in `contract`. As in the high-level description, `numComponents` contracts the graph on each round. Each contraction on Line 4 returns the contracted vertices V' and a table P mapping every $v \in V$ to a $v' \in V'$. Line 5 then does two things. Firstly it updates all edges so that the two endpoints are in V' by looking them up in P: this is what $(P[u], P[v])$ is. Secondly it removes all self edges: this is what the filter $P[u] \neq P[v]$ does. Once the edges are updated, the algorithm recurses on the smaller graph. The termination condition is when there are no edges. At this point each component has shrunk down to a singleton vertex and we just count them with $|V|$.

To make the code return the partitioning of the connected components is not much harder, as can be seen by the following code:

```
1  fun components((V, E), i) =
2  if |E| = 0 then {v ↦ v : v ∈ V}
3  else let
4    val (V', P) = contract((V, E), i)
5    val E' = {(P[u], P[v]) : (u, v) ∈ E | P[u] ≠ P[v]}
6    val P' = components((V', E'), i + 1)
7  in
8    {v ↦ P'[P[v]] : v ∈ V}
9  end
```

This code returns the partitioning in the same format as the partitioning returned by `contract`. The only difference of this code from `numComponents` is that on the way back up the recursion we update the representatives for V based on the representatives from V' returned by the recursive call. Consider our example graph. As before lets say the first `contract` returns

$$\begin{aligned} V' &= \{a, d, e\} \\ P &= \{a \mapsto a, b \mapsto a, c \mapsto a, d \mapsto d, e \mapsto e, f \mapsto e\} \end{aligned}$$

Since the graph is connected the recursive call to `components` will map all vertices in V' to the same vertex. Lets say this vertex is "a" giving:

$$P' = \{a \mapsto a, d \mapsto a, e \mapsto a\}$$

Now what Line 7 in the code does is for each vertex $v \in V$, it looks for v in P to find its representative $v' \in V'$, and then looks for v' in P' to find its representative in the connected component. This is implemented as $P'[P[v]]$. For example vertex f finds e from P and then looks this up in P' to find a. The final result returned by `components` is:

$$\{a \mapsto a, b \mapsto a, c \mapsto a, d \mapsto a, e \mapsto a, f \mapsto a\}$$

The base case of the `components` algorithm is to label every vertex with itself.

This leaves us with the question of how to implement `contract`. There are a three kinds of contraction we will consider:

Edge Contraction: Only pairs of vertices connected by an edge are contracted. One can think of the edges as pulling the two vertices together into one and then disappearing.

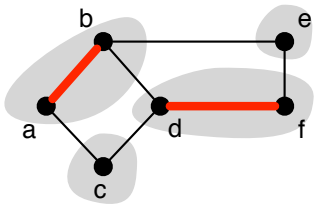
Star Contraction: One vertex of each component is identified as the center of the star and all other vertices are directly connected to it.

Tree Contraction: Generalizing star contraction, disjoint trees within the graph are identified and tree contraction is performed on these trees.

Keep in mind that the goal here is to do the contraction in parallel. Furthermore we want to contract the graph size (number of vertices) by a constant factor on each round. This way we will ensure the algorithm will run with just $O(\log n)$ rounds (contractions).

1.1 Edge Contraction

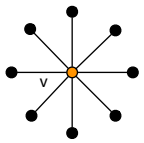
Lets consider contracting edges. How would we find a set of disjoint edges to contract? By disjoint we mean that the no two contraction edges can share a vertex. Here is an example of a set of disjoint edges on which to contract:



The edges to contract are $\{\{a,b\},\{d,f\}\}$. Note that the remaining two vertices, e and c, are left on their own.

Finding such a set of edges to contract is the problem of finding a *vertex matching*, i.e. we are trying to match every vertex with another vertex (monogamously). It turns out this can be done in parallel relatively easily by having every edge pick a random priority in the range $[0,1]$ and then choosing an edge if it has the highest priority on both the vertices it is incident on. Before we delve in further, however, we note that there is a problem with edge contraction. Consider the following kind of graph:

Definition 1.1 (Star)**.** A *star* graph $G=(V,E)$ is an undirected graph with a center vertex $v \in V$, and a set of edges $E = \{\{v,u\} : u \in V \setminus \{v\}\}$.



In words, a star graph is a graph made up of a single vertex v in the middle (called the center) and all the other vertices hanging off of it; these vertices are connected only to the center. Notice that a star graph is in fact a tree. If rooted at the center, it has depth 1, which is a super shallow tree.

If we're given a star graph, will edge contraction work well on this? How many edges can contract on each round. It is not difficult to convince ourselves that on a star graph with $n+1$ vertices—1 center and n "satellites"—any edge contraction algorithm will take $\Omega(n)$ rounds. We therefore move on.

1.2 Star Contraction

We now consider a more aggressive form of contraction that can be used to contract subgraphs which are stars in a single round. The idea is to combine the star center with all its "satellites" all at once. To apply this form of contraction on a general graph, we have to be able to answer the question: *How can we find disjoint stars?* Again, disjoint means that each vertex belongs to at most one partition, i.e. star. As an example, in the graph below (left), we can find 2 disjoint stars (right). The centers are colored red and the neighbors are green.

Finding Stars. One simple idea that has been fruitful so far is the use of randomization. Let's see what we can do with coin flips. At a high level, we will use coin flips to first decide which vertices will be star centers and which ones will be satellites and after that, we'll decide how to pair up each satellite with a center.

As usual, we'll start by flipping a coin for each vertex. If it comes up heads, that vertex is a star center. And if it comes up tails, then it will be a potential satellite—it is only a potential satellite because quite possibly, none of its neighbors flipped a head (it has no center to hook up to).

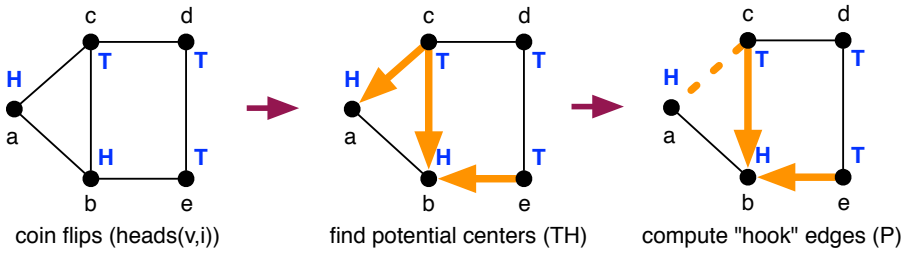
At this point, we have determined every vertex's potential role, but we aren't done: for each satellite vertex, we still need to decide which center it will join. For our purposes, we're only interested in ensuring that the stars are disjoint, so it doesn't matter which center a satellite joins. We will make each satellite choose any center in its set of neighbors arbitrarily.

Before describing the code the code, we need to say a couple words about the source of randomness. What we will assume is for each vertex we have a potentially infinite sequence of random coin flips and that we can access the i^{th} one with the function $\texttt{heads}(v,i) : \texttt{vertex} \times \texttt{int} \to \texttt{bool}$ that returns true if the i^{th} flip on vertex v is heads and false otherwise. You can think of it as having flipped all the coins ahead of time, stored a sequence of flips for each vertex, and now you are using `heads`to access the i^{th} one. Since most machines don't have true sources of randomness, in practice

this can be implemented with a pseudorandom number generator or even with a good hash function. We are now ready for the code for star contraction:

```
1   % requires: an undirected graph G = (V, E) and round number i
2   % returns:  V' = remaining vertices after contraction,
3   %            P = mapping from V to V'
4   fun starContract(G = (V, E), i) =
5   let
6     % select edges that go from a tail to a head
7     val TH = {(u, v) ∈ E | ¬heads(u, i) ∧ heads(v, i)}
8     % make mapping from tails to heads, removing duplicates
9     val P = ∪_{(u,v)∈TH} {u ↦ v}
10    % remove vertices that have been remapped
11    val V' = V \ domain(P)
12    % Map remaining vertices to themselves
13    val P' = {u ↦ u : u ∈ V'} ∪ P
14  in (V', P') end
```

mall example. Consider the following graph, where the coin flips turned up as indicated in the figure.



The function heads(v, i) on round i gives a coin flip for each vertex, which are shown on the left. Line 7 selects the edges that go from a tail to a head, which are shown in the middle as arrows and correspond to the set $TH = \{(c, a), (c, b), (e, b)\}$. Notice that some potential satellites (vertices that flipped tails) are adjacent to multiple centers (vertices that flipped heads). For example, vertex c is adjacent to vertices a and b, both of which got heads. A vertex like this will have to choose which center to join. This is sometimes called "hooking" and is decided on Line 9, which removes duplicates for a tail using union, giving $P = \{c \mapsto b, e \mapsto b\}$. In this example, c is hooked up with b, leaving a a center without any satellite.

Line 11 takes the vertices V and removes from them all the vertices in the domain of P, i.e. those that have been remapped. In our example domain$(P) = \{c, e\}$ so we are left with $V' = \{a, b, d\}$. In general, V' is the set of vertices whose coin flipped heads or whose coin flipped tails but didn't have a neighboring center. Finally we map all vertices in V' to themselves and union this in with the hooks giving $P' = \{a \mapsto a, b \mapsto b, c \mapsto b, d \mapsto d, e \mapsto b\}$.

Analysis of Star Contraction. When we contract these stars found by `starContract`, each star becomes one vertex, so the number of vertices removed is the size of P. In expectation, how big is P? The following lemma shows that on a graph with n non-isolated vertices, the size of P—or the number of vertices removed in one round of star contraction—is at least $n/4$.

Lemma 1.2. *For a graph G with n non-isolated vertices, let X_n be the random variable indicating the number of vertices removed by* `starContract`(G, r). *Then,* $\mathbf{E}\left[X_n\right] \geq n/4$.

Proof. Consider any non-isolated vertex $v \in V(G)$. Let H_v be the event that a vertex v comes up heads, T_v that it comes up tails, and R_v that $v \in \texttt{domain}(P)$ (i.e, it is removed). By definition, we know that a non-isolated vertex v has at least one neighbor u. So, we have that $T_v \wedge H_u$ implies R_v since if v is a tail and u is a head v must either join u's star or some other star. Therefore, $\mathbf{Pr}\left[R_v\right] \geq \mathbf{Pr}\left[T_v\right]\mathbf{Pr}\left[H_u\right] = 1/4$. By the linearity of expectation, we have that the number of removed vertices is

$$\mathbf{E}\left[\sum_{v:v \text{ non-isolated}} \mathbb{I}\{R_v\}\right] = \sum_{v:v \text{ non-isolated}} \mathbf{E}\left[\mathbb{I}\{R_v\}\right] \geq n/4$$

since we have n vertices that are non-isolated. □

Exercise 1. *What is the probability that a vertex with degree d is removed.*

Using `ArraySequence` and `STArraySequence`, we can implement `starContract` reasonably efficiently in $O(n+m)$ work and $O(\log n)$ span for a graph with n vertices and m edges.

1.3 Returning to Connectivity

Now lets analyze the cost of the algorithm for counting the number of connected components we described earlier when using star contraction for `contract`. Let n be the number of non-isolated vertices. Notice that once a vertex becomes isolated (due to contraction), it stays isolated until the final round (contraction only removes edges). Therefore, we have the following span recurrence (we'll look at work later):

$$S(n) = S(n') + O(\log n)$$

where $n' = n - X_n$ and X_n is the number of vertices removed (as defined earlier in the lemma about `starContract`). But $\mathbf{E}\left[X_n\right] = n/4$ so $\mathbf{E}\left[n'\right] = 3n/4$. This is a familiar recurrence, which we know solves to $O(\log^2 n)$.

As for work, ideally, we would like to show that the overall work is linear since we might hope that the size is going down by a constant fraction on each round. Unfortunately, this is not the case. Although we have shown that we can remove a constant fraction of the non-isolated vertices on one star contract round, we have not shown anything about the number of edges. We can argue that the number of edges removed is at least equal to the number of vertices since removing a vertex also removes the edge that attaches it to its star's center. But this does not help asymptotically bound the

number of edges removed. Consider the following sequence of rounds:

round	vertices	edges
1	n	m
2	$n/2$	$m - n/2$
3	$n/4$	$m - 3n/4$
4	$n/8$	$m - 7n/8$

In this example, it is clear that the number of edges does not drop below $m - n$, so if there are $m > 2n$ edges to start with, the overall work will be $O(m \log n)$. Indeed, this is the best bound we can show asymptotically. Hence, we have the following work recurrence:

$$W(n, m) \leq W(n', m) + O(n + m),$$

where n' is the remaining number of non-isolated vertices as defined in the span recurrence. This solves to $\mathbf{E}[W(n, m)] = O(n + m \log n)$. Altogether, this gives us the following theorem:

Theorem 1.3. *For a graph $G = (V, E)$,* `numComponents` *using* `starContract` *graph contraction with an array sequence works in $O(|V| + |E| \log |V|)$ work and $O(\log^2 |V|)$ span.*

1.4 Tree Contraction

Tree contraction takes a set of disjoint trees and contracts them. One way to do this is by repeated star contraction. The advantage of tree contraction over general graph contraction is that the number of edges on each step is at most one less than the number of remaining vertices. Therefore the number of edges must go down geometrically from step to step, and the overall cost of tree contraction is $O(m)$ work and $O(\log^2 n)$ span using an array sequence.

2 Spanning Trees and Forests

Recall that an undirected graph is a forest if it has no cycles and is a tree if it has no cycles and is connected. A tree on n vertices always has exactly $n - 1$ edges, and a forest has at most $n - 1$ edges. A *spanning tree* of an undirected connected graph $G = (V, E)$ is a tree $T = (V, E')$ where $E' \subseteq E$. A *spanning forest* of a graph $G = (V, E)$ is the union of spanning trees on its connected components. We are interested in the spanning forest problem, which is to find a spanning forest for a given undirected graph (the spanning tree is just the special case when the input graph is connected).

It turns out that a spanning forest of a graph G can be generated from our connectivity algorithm. In particular all we need to do is keep track of all the edges that we use to hook, and return the union of these edges. We will see this in more detail as we cover minimum spanning trees, our next topic.

Lecture 17 — Graph Contraction II

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — October 23, 2012

What was covered in this lecture:

- Continued on graph contraction and connectivity from last lecture.

See notes from last lecture.

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

Lecture 18 — Minimum Spanning Tree

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — October 25, 2012

What was covered in this lecture:

- Minimum Spanning Tree

1 Minimum Spanning Tree

The minimum (weight) spanning tree (MST) problem is given an connected undirected graph $G = (V, E)$, where each edge e has weight $w_e \geq 0$, find a spanning tree of minimum weight (i.e., the sum of the weights of the edges). That is to say, we are interested in finding the spanning tree T that minimizes

$$w(T) = \sum_{e \in E(T)} w_e.$$

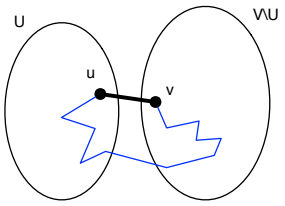
You have seen Minimum Weigh Spanning Trees in both 15-122 and possibly 15-251. In previous classes, you went over Kruskal's and Prim's algorithms. At a glance, Kruskal's and Prim's seem to be two drastically different approaches to solving MST: whereas Kruskal's sorts edges by weight and considers the edges in order, using a union-find data structure to detect when two vertices are in the same component and join them if not, Prim's maintains a tree grown so far and a priority queue of edges incident on the current tree, pulling the minimum edge from it to add to the tree. The two algorithms, in fact, rely on the same underlying principle about "cuts" in a graph, which we'll discuss next.

Light Edge Rule. The main property that underlines many MST algorithms is a simple fact about cuts in a graph. Here, we will assume without any loss of generality that all edges have distinct weights. This is easy to do since we can break ties in a consistent way. For a graph $G = (V, E)$, a *cut* is defined in terms of a subset $U \subsetneq V$. This set U partitions the graph into $(U, V \setminus U)$, and we refer to the edges between the two parts as the cut edges $E(U, \overline{U})$, where as is typical in literature, we write $\overline{U} = V \setminus U$. The subset U might include a single vertex v, in which case the cut edges would be all edges incident on v. But the subset U must be a proper subset of V (i.e., $U \neq \emptyset$ and $U \neq V$).

The following theorem states that the lightest edge across a cut is in the MST of G:

Theorem 1.1. *Let $G = (V, E, w)$ be a connected undirected weighted graph with distinct edge weights. For any nonempty $U \subsetneq V$, the minimum weight edge e between U and $V \setminus U$ is in the minimum spanning tree MST(G) of G.*

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.



Proof. The proof is by contradiction. Assume the minimum-weighted edge $e = (u, v)$ is not in the MST. Since the MST spans the graph, there must be some simple path P connecting u and v in the MST (i.e., consisting of just edges in the MST). The path must cross the cut between U and $V \setminus U$ at least once since u and v are on opposite sides. By attaching P to e, we form a cycle (recall that by assumption $e \notin MST$). If we remove the maximum weight edge from P and replace it with e we will still have a spanning tree, but it will be have less weight. This is a contradiction. □

Note that the last step in the proof uses the facts that (1) adding an edge to a spanning tree creates a cycle, and (2) removing any edge from this cycle creates a tree again.

As mentioned there are three algorithms for solving the MST problem that are all based on Theorem 1.1: Kruskal's algorithm, Prim's algorithm and Borůvka's algorithm. Kruskal's and Prim's are based on selecting a single lightest weight edge on each step and are hence sequential, while Borůvka's selects multiple edges and can be parallelized. We briefly review Kruskal's and Prim's algorithm and will spend most of our time on a parallel variant of Borůvka's algorithm.

Kruskal's Algorithm

The idea of Kruskal's algorithm is to sort the edges and then process them one at a time starting with the lightest edge. We maintain a set of components, where initially each vertex is its own component. When we process an edge we check if the two endpoints are in the same component. If so, we ignore the edge and move on. If not, we join the two components at the endpoints into a single component and add the edge to the MST. This joining is similar to the "edge" contraction we talked about in the last lecture, since it joins two components into one, except that we only do a single edge contraction on each step instead of multiple.

Every edge that we add to the MST is a lightest weight edge because we process the edges in sorted order starting with the minimum—i.e., it is the overall lightest weight remaining edge so it must be the lightest weight edge in the cut between its two endpoints. Efficiently joining components and checking if two vertices are in the same component can be done with a union-find data structure. With an efficient implementation of union-find, the work for the algorithm is dominated by the need to sort the edges. The algorithm therefore runs in $O(m \log n)$ work. It is fully sequential.

Prim's Algorithm

The idea of Prim's algorithm is to do a priority first search. In particular we start at an arbitrary vertex s, maintain a visited set X, and maintain priorities on the frontier vertices $v \in F$ based on the minimum weight edge from X to v. On each step we visit a vertex v with minimum priority via an edge (u, v) and add (u, v) to the MST. Since for MST we assume the graph is connected, this algorithm will visit all vertices. The algorithm is correct since the minimum weight edge leaving X must be in the MST by Theorem 1.1 (it is the minimum weigh edge separating X from $V \setminus X$). Here is the code.

```
1  fun prim(G) =
2  let
3    fun enqueue v (Q,(u,w)) = PQ.insert (w,(v,u)) Q
4    fun proper(X, Q, T) =
5      case PQ.deleteMin(Q) of
6          (NONE, _) ⇒ T                                  % Done
7        | (SOME(d,(u,v)),Q') ⇒
8          if (v ∈? X) then proper(X, Q', T)              % Already visited
9          else let
10           val X' = X ∪ {v}                             % Visit
11           val T' = T ∪ {(u,v)}                         % Add edge to MST
12           val Q'' = iter (enqueue v) Q' N_G(v)         % Enqueue v's neighbors
13         in proper(X', Q'', T') end
14   val s = an arbitrary vertex from G
15   val Q = iter (enqueue s) {} N_G(s)                   % Enqueue s's neighbors
16 in
17   proper({s}, Q, {})
18 end
```

This algorithm returns the set of edges in the MST. The only significant differences from Dijkstra's algorithm is that the weight we use for the priority queue is w instead of $w+d$. There are a few other minor changes such as maintaining the MST T and storing an edge in the priority queue instead of just the target vertex.

The algorithm runs with exactly the same work as Dijkstra's algorithm: $O(m \log n)$.

2 Parallel Minimum Spanning Tree

We now focus on developing an MST algorithm that runs efficiently in parallel. We will be looking at a parallel algorithm based on an approach by Borůvka, which quite surprisingly predates both Kruskal's and Prim's. This oldest and arguably simplest MST algorithm went back to 1926, long before computers were invented. In fact, this algorithm was rediscovered many times.

We'll use graph contraction; our presentation differs slightly from the original description. Here's the key observation:

> The minimum weight edge out of every vertex of a weighted graph G belongs to its MST.

This fact follows from Theorem 1.1 by considering each vertex as our set U. We will refer to these edges as the *minimum weight edges* of the graph. The following example illustrates this situation:

where we have highlighted the minimum weight edges. Note that some edges (e.g. the ones weighted 1 and 4) are minimum for both of their endpoints.

Are the minimum weight edges all the edges of the MST? As this example shows, no—we are still missing some of the edges (in this case just the edge weighted 5). How should we proceed?

Idea #1: Throw all the minimum weight edges into the minimum spanning tree, and contract the subgraphs these edges link together. Repeat until no edges remain. In the example above, after one step we will have added the edges weighted $\{1,2,4,6\}$ and be left with a graph with two vertices connected by the edge weighted 5. On the next step we add this edge and are left with a single vertex and no edges and hence we are done with the MST consisting of the edge weights $\{1,2,4,6\} \cup \{5\} = \{1,2,4,5,6\}$.

This is exactly Borůvka's idea. He did not discuss implementing the contraction in parallel, but we can use the tree contraction from the last lecture notes to do this directly. Furthermore since the minimum weight edges must be a forest (convince yourself of this) we can use tree contraction which only requires $O(m)$ work and $O(\log^2 n)$ span (if using array sequences). Furthermore every step must remove at least half the vertices. Therefore the algorithm will run in at most $\lg n$ rounds and hence will take $O(m \log n)$ work and $O(\log^3 n)$ span.

Exercise 1. *Prove that each Borůvka step must remove at least 1/2 the vertices.*

Idea #2: Instead, we'll explore a slightly different idea: rather than trying to contract all these edges, we'll subselect the edges so that they are made up of disjoint stars, and we'll contract these stars using star contraction. We then repeat this simpler step until there are no edges. More precisely, for a set of minimum weight edges *minE*, let $H = (V, minE)$ be a subgraph of G. We will apply one step of star contraction algorithm on H. To do this we modify our `starContract` routine so that after flipping coins, the tails only hook across their minimum weight edge. The advantage of this second approach is that we will reduce the overall span for finding the MST from $O(\log^3 n)$ to $O(\log^2 n)$.

The modified algorithm for star contraction is as follows:

```
1  fun minStarContract(G = (V, E), i) =
2  let
3     val minE = minEdges(G)
4     val P = {u ↦ (v, w) ∈ minE | ¬heads(u, i) ∧ heads(v, i)}
5     val V' = V \ domain(P)
6  in (V', P) end
```

where `minEdges`(G) finds the minimum edge out of each vertex v.

Before we go into details about how we might keep track of the MST and other information, let's try to understand what effects this change has on the number of vertices contracted away. If we have n non-isolated vertices, the following lemma shows that we're still contracting away $n/4$ vertices in expectation:

Lemma 2.1. *For a graph G with n non-isolated vertices, let X_n be the random variable indicating the number of vertices removed by* `minStarContract`(G, r). *Then,* $\mathbf{E}\left[X_n\right] \geq n/4$.

Proof. The proof is pretty much identical to our proof for `starContract` except here we're not working with the whole edge set, only a restricted one `minE`. Let $v \in V(G)$ be a non-isolated vertex. Like before, let H_v be the event that v comes up heads, T_v that it comes up tails, and R_v that $v \in$ `domain`(P) (i.e, it is removed). Since v is a non-isolated vertex, v has neighbors—and one of them has the minimum weight, so there exists a vertex u such that $(v,u) \in$ `minE`. Then, we have that $T_v \wedge H_u$ implies R_v since if v is a tail and u is a head, then v must join u. Therefore, $\mathbf{Pr}\left[R_v\right] \geq \mathbf{Pr}\left[T_v\right]\mathbf{Pr}\left[H_u\right] = 1/4$. By the linearity of expectation, we have that the number of removed vertices is

$$\mathbf{E}\left[\sum_{v:v \text{ non-isolated}} \mathbb{I}\{R_v\}\right] = \sum_{v:v \text{ non-isolated}} \mathbf{E}\left[\mathbb{I}\{R_v\}\right] \geq n/4$$

since we have n vertices that are non-isolated. □

This means that this MST algorithm will take only $O(\log n)$ rounds, just like our other graph contraction algorithms.

Final Things. There is a little bit of trickiness since as the graph contracts the endpoints of each edge changes. Therefore, if we want to return the edges of the minimum spanning tree, they might not correspond to the original endpoints. To deal with this, we associate a unique label with every edge and return the tree as a set of labels (i.e. the labels of the edges in the spanning tree). We also associate the weight directly with the edge. The type of each edge is therefore (`vertex` × `vertex` × `weight` × `label`), where the two vertex endpoints can change as the graph contracts but the weight and label stays fixed. This leads to the following slightly-updated version of `minStarContract`:

```
1  fun minStarContract(G = (V, E), i) =
2  let
3     val minE = minEdges(G)
4     val P = {(u ↦ (v, w, ℓ)) ∈ minE | ¬heads(u, i) ∧ heads(v, i)}
5     val V′ = V \ domain(P)
6  in (V′, P) end
```

The function `minEdges`(G) in Line 3 finds the minimum edge out of each vertex v and maps v to the pair consisting of the neighbor along the edge and the edge label. By Theorem 1.1, since all these edges are minimum out of the vertex, they are safe to add to the MST. Line 4 then picks from these

edges the edges that go from a tail to a head, and therefore generates a mapping from tails to heads along minimum edges, creating stars. Finally, Line 5 removes all vertices that are in this mapping to star centers.

This is ready to be used in the MST code, similar to the `graphContract` code studied last time, except we return the set of labels for the MST edges instead of the remaining vertices.

```
1  fun MST((V, E), T, i) =
2  if |E| = 0 then T
3  else let
4    val (V', PT) = minStarContract((V, E), i)
5    val P = {u ↦ v : u ↦ (v, w, ℓ) ∈ PT} ∪ {v ↦ v : v ∈ V'}
6    val T' = {ℓ : u ↦ (v, w, ℓ) ∈ PT}
7    val E' = {(P[u], P[v], w, l) : (u, v, w, l) ∈ E | P[u] ≠ P[v]}
8  in
9    MST((V', E'), T ∪ T', i + 1)
10 end
```

The MST algorithm is called by running $\texttt{MST}(G, \emptyset, r)$. As an aside, we know that T is a spanning forest on the contracted nodes.

Finally we have to describe how to implement $\texttt{minEdges}(G)$, which returns for each vertex the minimum edge incident on that vertex. There are various ways to do this. One way is to use merging on tables with an appropriate function to resolve collisions. Here is code that merges edges by taking the one with lighter edge weight.

```
fun joinEdges((v1, w1, l1), (v2, w2, l1)) =
  if (w1 ≤ w2) then (v1, w1, l1) else (v2, w2, l1)

fun minEdges(E) =
let
  val ET = {u ↦ (v, w, l) : (u, v, w, l) ∈ E}
in
  reduce (merge joinEdges) {} ET
end
```

If using sequences for the edges and vertices an even simpler way is to presort the edges by decreasing weight and then use `inject`. Recall that when there are collisions at the same location `inject` will always take the last value, which will be the one with minimum weight.

Lecture 19 — Quicksort and Sorting Lower Bounds

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — 30 Octorber 2012

Today:

- Quicksort
- Work and Span Analysis of Randomized Quicksort
- Lower Bounds (not sure we will get to this)

1 Quicksort

You have surely seen quicksort before. The purpose of this lecture is to analyze quicksort in terms of both its work and its span. As we will seen in upcoming lectures, the analysis of quicksort is effectively identical to the analysis of a certain type of balanced tree called Treaps. It is also the same as the analysis of "unbalanced" binary search trees under random insertion.

Quicksort is one of the earliest and most famous algorithms. It was invented and analyzed by Tony Hoare around 1960. This was before the big-O notation was used to analyze algorithms. Hoare invented the algorithm while an exchange student at Moscow State University while studying probability under Kolmogorov—one of the most famous researchers in probability theory. The analysis we will cover is different from what Hoare used in his original paper, although we will mention how he did the analysis.

Consider the following implementation of quicksort. In this implementation, we intentionally leave the pivot-choosing step unspecified because the property we are discussing holds regardless of the choice of the pivot.

```
1  fun quicksort(S) =
2    if |S| = 0 then S
3    else let
4           val p = pick a pivot from S
5           val S1 = ⟨ s ∈ S | s < p ⟩
6           val S2 = ⟨ s ∈ S | s = p ⟩
7           val S3 = ⟨ s ∈ S | s > p ⟩
8           val (R1, R3) = ( quicksort(S1) || quicksort(S3) )
9         in
10          append(R1, append(S2, R2))
11        end
```

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

There is clearly plenty of parallelism in this version quicksort.[1] There is both parallelism due to the two recursive calls and in the fact that the filters for selecting elements greater, equal, and less than the pivot can be parallel.

The question to ask is: *How does the pivot choice effect the costs of quicksort*? It will be useful to consider the function call tree generated by quicksort. Each call to quicksort either makes no recursive calls (the base case) or two recursive calls. The call tree is therefore binary. We will be interested in analyzing the depth of this tree since this will help us derive the span of the quicksort algorithm.

Let's consider some strategies for picking a pivot:

- *Always pick the first element:* If the sequence is sorted in increasing order, then picking the first element is the same as picking the smallest element. We end up with a lopsided recursion tree of depth n. The total work is $O(n^2)$ since $n-i$ keys will remain at level i and hence we will do $n-i-1$ comparisons at that level for a total of $\sum_{i=0}^{n-1}(n-i-1)$. Similarly, if the sequence is sorted in decreasing order, we will end up with a recursion tree that is lopsided in the other direction. In practice, it is not uncommon for a sort function input to be a sequence that is already sorted or nearly sorted.
- *Pick the median of three elements:* Another strategy is to take the first, middle, and the last elements and pick the median of them. For sorted lists the split is even, so each side contains half of the original size and the depth of the tree is $O(\log n)$. Although this strategy avoids the pitfall with sorted sequences, it is still possible to be unlucky, and in the worst-case the costs and tree depth are the same as the first strategy. This is the strategy used by many library implementations of quicksort.
- *Pick an element randomly:* It is not immediately clear what the depth of this is, but intuitively, when we choose a random pivot, the size of each side is not far from $n/2$ in expectation. This doesn't give us a proof but it gives us hope that this strategy will result in a tree of depth $O(\log n)$ in expectation. Indeed, picking a random pivot gives us expected $O(n \log n)$ work and $O(\log^2 n)$ span for quicksort and an expected $O(\log n)$-depth tree, as we will show.

We are interested in picking elements at random.

2 Expected work for randomized quicksort

As discussed above, if we always pick the first element then the worst-case work is $O(n^2)$, for example when the array is already sorted. The *expected work*, though, is $O(n \log n)$ as we will prove below. That is, the work averaged over all possible input ordering is $O(n \log n)$. In other words, on most input this naive version of quicksort works well on average, but can be slow on some (common) inputs.

[1] This differs from Hoare's original version which sequentially partitioned the input by the pivot using two fingers that moved from each end and swapping two keys whenever a key was found on the left greater than the pivot and on the right less than the pivot.

On the other hand, if we choose an element randomly to be the pivot, the *expected worst-case work* is $O(n \log n)$. That is, for input in **any** order, the expected work is $O(n \log n)$: No input has expected $O(n^2)$ work. But with a very small probability we can be unlucky, and the random pivots result in unbalanced partitions and the work is $O(n^2)$.

For the analysis of randomized quicksort, we'll consider a completely equivalent algorithm that will be slightly easier to analyze. Before the start of the algorithm, we'll pick for each element a random priority uniformly at random from the real interval $[0, 1]$—and in Line 4 in the above algorithm, we'll instead pick the key with the highest priority. Notice that once the priorities are decided, the algorithm is completely deterministic; you should convince yourself that the two presentations of randomized quicksort are fully equivalent (modulo the technical details about how we might store the priority values).

We're interested in counting how many comparisons `quicksort` makes. This immediately bounds the work for the algorithm because this is where the bulk of work is done. That is, if we let

$$X_n \;=\; \#\text{ of comparisons } \texttt{quicksort} \text{ makes on input of size } n,$$

our goal is to find an upper bound on $\mathbf{E}\left[X_n\right]$ for any input sequence S. For this, we'll consider the final sorted order[2] of the keys $T = \texttt{sort}(S)$. In this terminology, we'll also denote by p_i the priority we chose for the element T_i.

We'll derive an expression for X_n by breaking it up into a bunch of random variables and bound them. Consider two positions $i, j \in \{1, \ldots, n\}$ in the sequence T. We use the random indicator variables A_{ij} to indicate whether we compare the elements T_i and T_j during the algorithm—i.e., the variable will take on the value 1 if they are compared and 0 otherwise.

The crux of the matter is in describing the event $A_{ij} = 1$ in terms of a simple event that we have a handle on. Before we prove any concrete result, let's take a closer look at the quicksort algorithm to gather some intuitions. Notice that the top level takes as its pivot p the element with highest priority. Then, it splits the sequence into two parts, one with keys larger than p and the other with keys smaller than p. For each of these parts, we run `quicksort` recursively; therefore, inside it, the algorithm will pick the highest priority element as the pivot, which is then used to split the sequence further.

For any one call to `quicksort` there are three possibilities for A_{ij}, where $i < j$:

- The pivot (highest priority element) is either T_i or T_j, in which case T_i and T_j are compared and $A_{ij} = 1$.
- The pivot is element between T_i and T_j, in which case T_i is in S_1 and T_j is in S_3 and T_i and T_j will never be compared and $A_{ij} = 0$.
- The pivot is less than T_i or greater than T_j. Then T_i and T_j are either both in S_1 or both in S_3, respectively. Whether T_i and T_j are compared will be determined in some later recursive call to `quicksort`.

In the first case above, when two elements are compared, the non-pivot element is part of S_1, S_2, or S_3—but the pivot element is part of S_2, on which we don't recurse. This gives the following observation:

[2] Formally, there's a permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ between the positions of S and T.

Observation 2.1. *If two elements are compared in a* `quicksort` *call, they will never be compared again in other call.*

Also notice in the corresponding BST, when two elements are compared, the pivot element become the root of two subtrees, one of which contains the other element.

Observation 2.2. *In the quicksort algorithm, two elements are compared in a* `quicksort` *call if and only if one element is an ancestor of the other in the corresponding BST.*

Therefore, with these random variables, we can express the total comparison count X_n as follows:

$$X_n \;\leq\; 3\sum_{i=1}^{n}\sum_{j=i+1}^{n} A_{ij}$$

The constant 3 is because our not-so-optimized quicksort compares each element to a pivot 3 times. By linearity of expectation, we have $\mathbf{E}\left[X_n\right] \leq 3\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbf{E}\left[A_{ij}\right]$. Furthermore, since each A_{ij} is an indicator random variable, $\mathbf{E}\left[A_{ij}\right] = \mathbf{Pr}\left[A_{ij}=1\right]$. Our task therefore comes down to computing the probability that T_i and T_j are compared (i.e., $\mathbf{Pr}\left[A_{ij}=1\right]$) and working out the sum.

Computing the probability $\mathbf{Pr}\left[A_{ij}=1\right]$. Let us first consider the first two cases when the pivot is one of $T_i, T_{i+1}, ..., T_j$. With this view, the following observation is not hard to see:

Claim 2.3. *For $i < j$, T_i and T_j are compared if and only if p_i or p_j has the highest priority among $\{p_i, p_{i+1}, \ldots, p_j\}$.*

Proof. We'll show this by contradiction. Assume there is a key T_k, $i < k < j$ with a higher priority between them. In any collection of keys that include T_i and T_j, T_k will become a pivot before either of them. Since T_k "sits" between T_i and T_k (i.e., $T_i \leq T_k \leq T_j$) , it will separate T_i and T_j into different buckets, so they are never compared. □

Therefore, for T_i and T_j to be compared, p_i or p_j has to be bigger than all the priorities in between. Since there are $j-i+1$ possible keys in between (including both i and j) and each has equal probability of being the highest, the probability that either i or j is the greatest is $2/(j-i+1)$. Therefore,

$$\begin{aligned}\mathbf{E}\left[A_{ij}\right] &= \mathbf{Pr}\left[A_{ij}=1\right] \\ &= \mathbf{Pr}\left[p_i \text{ or } p_j \text{ is the maximum among } \{p_i,\ldots,p_j\}\right] \\ &= \frac{2}{j-i+1}.\end{aligned}$$

Notice that element T_i is compared to T_{i+1} with probability 1. It is easy to understand why if we consider the corresponding BST. One of T_i and T_{i+1} must be an ancestor of the other in the BST: There is no element that could be the root of a subtree that has T_i in its left subtree and T_{i+1} in its

right subtree. On the other hand, if we consider T_i and T_{i+2} there could be such an element, namely T_{i+1}, which could have T_i in its left subtree and T_{i+2} in its right subtree. That is, with probability 1/3, T_{i+1} has the highest probability of the three and T_i is not compared to T_{i+2}, and with probability 2/3 one of T_i and T_{i+2} has the highest probability and, the two are compared. In general, the probability of two elements being compared is inversely proportional to the number of elements between them when sorted. The further apart the less likely they will be compared. Analogously, the further apart the less likely one will be the ancestor of the other in the related BST.

Hence, the expected number of comparisons made in randomized `quicksort` is

$$\begin{aligned}
\mathbf{E}\left[X_n\right] &\leq 3\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbf{E}\left[A_{ij}\right] \\
&= 3\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{2}{j-i+1} \\
&= 3\sum_{i=1}^{n}\sum_{k=2}^{n-i+1}\frac{2}{k} \\
&\leq 6\sum_{i=1}^{n}H_n \\
&= 6nH_n \in O(n\log n)
\end{aligned}$$

Indirectly, we have also shown that the average work for the basic deterministic quicksort (always pick the first element) is also $O(n\log n)$. Just shuffle the data randomly and then apply the basic quicksort algorithm. Since shuffling the input randomly results in the same input as picking random priorities and then reordering the data so that the priorities are in decreasing order, the basic quicksort on that shuffled input does the same operations as randomized quicksort on the input in the original order. Thus, if we averaged over all permutations of the input the work for the basic quicksort is $O(n\log n)$ on average.

2.1 An alternative method

Another way to analyze the work of quicksort is to write a recurrence for the expected work (number of comparisons) directly. This is the approach taken by Tony Hoare in his original paper. For simplicity we assume there are no equal keys (equal keys just reduce the cost). The recurrence for the number of comparisons $X(n)$ done by quicksort is then:

$$X(n) \quad = \quad X(Y_n) + X(n - Y_n - 1) + n - 1$$

where the random variable Y_n is the size of the the set S_1 (we use $X(n)$ instead of X_n to avoid double subscrips). We can now write an equation for the expectation of $X(n)$.

$$\begin{aligned}
\mathbf{E}\left[X(n)\right] &= \mathbf{E}\left[X(Y_n) + X(n - Y_n - 1) + n - 1\right] \\
&= \mathbf{E}\left[X(Y_n)\right] + \mathbf{E}\left[X(n - Y_n - 1)\right] + n - 1 \\
&= \frac{1}{n}\sum_{i=0}^{n-1}(\mathbf{E}\left[X(i)\right] + \mathbf{E}\left[X(n-i-1)\right]) + n - 1
\end{aligned}$$

where the last equality arises since all positions of the pivot are equally likely, so we can just take the average over them. This can be by guessing the answer and using substitution. It gives the same result as our previous method. We leave this as exercise.

3 Expected Span of Quicksort

Recall that in randomized quicksort, at each recursive call, we partition the input sequence S of length n into three subsequences L, E, and R, such that elements in the subsequences are less than, equal, and greater than the pivot, respectfully. Let $X_n = \max\{|L|, |R|\}$, which is the size of larger subsequence; The span of quicksort is determined by the sizes of these larger subsequences. For ease of analysis, we will assume that $|E| = 0$, as more equal elements will only decrease the span. As this partitioning uses `filter` we have the following recurrence for span:

$$S(n) = S(X_n) + O(\log n)$$

Let $\overline{S}(n)$ denote $\mathbf{E}[S(n)]$. As we did for `SmallestK` we will bound $\overline{S}(n)$ by considering the $\mathbf{Pr}\left[X_n \leq 3n/4\right]$ and $\mathbf{Pr}\left[X_n > 3n/4\right]$ and use the maximum sizes in the recurrence to upper bound $\mathbf{E}[S(n)]$. Now, the $\mathbf{Pr}\left[X_n \leq 3n/4\right] = 1/2$, since half of the randomly chosen pivots results in the larger partition to be at most $3n/4$ elements: Any pivot in the range $T_{n/4}$ to $T_{3n/4}$ will do, where T is the sorted input sequence.

So then, by the definition of expectation, we have

$$\begin{aligned}
\overline{S}(n) =& \leq \sum_i \mathbf{Pr}\left[X_n = i\right] \cdot \overline{S}(i) + c \log n \\
&\leq \mathbf{Pr}\left[X_n \leq \tfrac{3n}{4}\right] \overline{S}(\tfrac{3n}{4}) + \mathbf{Pr}\left[X_n > \tfrac{3n}{4}\right] \overline{S}(n) + c \cdot \log n \\
&\leq \tfrac{1}{2}\overline{S}(\tfrac{3n}{4}) + \tfrac{1}{2}\overline{S}(n) + c \cdot \log n \\
&\implies (1 - \tfrac{1}{2})\overline{S}(n) \leq \tfrac{1}{2}\overline{S}(\tfrac{3n}{4}) + c \log n \\
&\implies \overline{S}(n) \leq \overline{S}(\tfrac{3n}{4}) + 2c \log n,
\end{aligned}$$

which we know is a balanced cost tree and solves to $O(\log^2 n)$.

That is, with probability $1/2$ we will be lucky and the subproblem size will go down by at least $3n/4$ and with probability $1/2$ we will be unlucky and we have to start again. In the end, the expected span is twice what it would be if we could guarantee partition sizes of $n/4$ and $3n/4$.

4 Lower Bounds

After spending time formulating a concrete problem, we might wonder how hard the problem actually is. In this course thus far, our focus has been on obtaining efficient algorithms for certain problems. For a problem P, we try to design efficient algorithms to solve it. The existence of an algorithm gives an upper bound on the complexity of the problem P. In particular, an algorithm A with work (either

expected or worst-case) $O(f(n))$ is a constructive proof that P can be solved provided $O(f(n))$ work. This is essentially the upper bound part of the question.

In this lecture, we'll turn the tables, showing that certain problems cannot be solved more efficiently than a given bound. This is the lower bound part of the question. In general, this is a harder task: To establish a lower bound, we have to argue that *no algorithm, however smart, can possibly do better than what we claim*; it is no longer sufficient to exhibit an algorithm A and analyze its performance.

4.1 Sorting and Merging Lower Bounds

Before we look at lower bounds for sorting and merging, let us review the (upper) bounds we have for various sorting algorithms we've covered:

Algorithm	Work	Span
Quick Sort	$O(n \log n)$	$O(\log^2 n)$
Merge Sort	$O(n \log n)$	$O(\log^2 n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$
Balanced BST Sort	$O(n \log n)$	$O(\log^2 n)$

Notice that in this table, all algorithms have $O(n \log n)$ work—and except for heap sort, every algorithm is very parallel ($\log^2 n$ span). Can we sort in less than $O(n \log n)$ work? Probably. But we'll show that *any* deterministic *comparison-based* sorting algorithm must use $\Omega(n \log n)$ comparisons to sort n entries in the worst case. In the comparison-based model, we have no domain knowledge about the entries and the only operation we have to determine the relative order of a pair of entries x and y is a comparison operation, which returns whether $x < y$. More precisely, we'll prove the following theorem:

Theorem 4.1. *For a sequence $\langle x_1, \ldots, x_n \rangle$ of n distinct entries, finding the permutation π on $[n]$ such that $x_{\pi(1)} < x_{\pi(2)} < \cdots < x_{\pi(n)}$ requires, in the worst case, at least $\frac{n}{2} \log(\frac{n}{2})$ queries to the $<$ operator.*

Since each comparison takes at least constant work, this implies an $\Omega(n \log n)$ lower bound on the work required to sort a sequence of length n in the comparison model.

What about merging? Can we merge sorted sequences faster than resorting them? As seen in previous lectures, we can actually merge two sorted sequences in $O(m \log(1 + n/m))$ work, where m is the length of the shorter of the two sequences, and n the length of the longer one. We'll show, however, that in the comparison-based model, we cannot hope to do better:

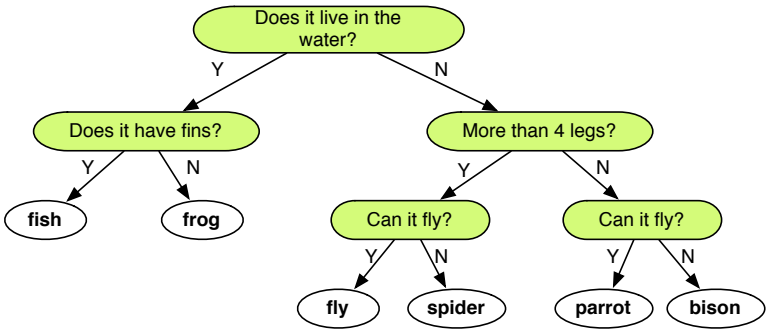
Theorem 4.2. *Merging two sorted sequences of lengths m and n ($m \leq n$) requires at least*

$$m \log_2(1 + \tfrac{n}{m})$$

comparison queries in the worst case.

4.2 Decision Trees or The 20 Questions Game

Let's play game. Suppose I think of an animal but you know for fact it's one of the following: a fish, a frog, a fly, a spider, a parrot, or a bison. You want to find out what animal that is by answering the fewest number of Yes/No questions (you're only allowed to ask Yes/No questions). What strategy would you use? Perhaps, you might try the following reasoning process:



Interestingly, this strategy is optimal: There is no way you could have asked any 2 Yes/No questions to tell apart the 6 possible answers. If we can ask only 2 questions, any strategy that is deterministic and computes the output using only the answers to these Yes/No questions can distinguish between only $2^2 = 4$ possibilities. Thus, using 3 questions is the best one can do.

Determining the minimum number of questions necessary in the worst case in at the crux of many lower-bound arguments. For starters, we describe a way to represent a deterministic strategy for playing such a game in the definition below.

Definition 4.3 (Binary Decision Trees)**.** A *decision tree* is a tree in which

- each leaf node is an answer (i.e. what the algorithm outputs);
- each internal node represents a query—some question about the input instance—and has k children, corresponding to one of the k possible responses $\{0, \ldots, k-1\}$;
- and the answer is computed as follows: we start from the root and follow a path down to a leaf where at each node, we choose which child to follow based on the query response.

The crucial observation is the following: If we're allowed to make at most q queries (i.e., ask at most q Yes/No questions), the number of possible answers we can distinguish is the number of leaves in a binary tree with depth at most q; this is at most 2^q. Taking logs on both sides, we have

If there are N possible outcomes, the number of questions needed is at least $\log_2 N$.

That is, there is *some* outcome, that requires answering at least $\log_2 N$ questions to determine that outcome.

4.3 Warm-up: Guess a Number

As a warm-up question, if I pick a number a between 1 and 2^{20}, how many Yes/No questions you need to ask before you can zero in on a? By the calculation above, since there are $N = 2^{20}$ possible outcomes, you will need *at least*

$$\log_2 N = 20$$

questions in the worst case.

Another way to look at the problem is to suppose I am devious and I don't actually pick a number in advance. Each time you ask a question of the form "is the number greater than x", in effect you are splitting the set of possible numbers into two groups. I always answer so the set of remaining possible numbers has the greater cardinality. That is, each question you ask eliminates at most half of the numbers. Since there are $N = 2^{20}$ possible values, I can force you ask $\log_2 N = 20$ questions before I must concede and pick the last remaining number as my a. This variation of the games shows that no matter what strategy you use to ask questions, there is always *some* a that would cause you to ask a lot of questions.

4.4 A Sorting Lower Bound

Let's turn back to the classical sorting problem. We will prove Theorem 4.1. This theorem follows almost immediately from our observation about k-ary decision trees. There are $n!$ possible permutations, and to narrow it down to one permutation which orders this sequence correctly, we'll need $\log(n!)$ queries, so the number of comparison queries is at least

$$\begin{aligned}\log(n!) &= \log n + \log(n-1) + \cdots + \log(n/2) + \cdots + \log 1 \\ &\geq \log n + \log(n-1) + \cdots + \log(n/2) \\ &\geq \tfrac{n}{2} \cdot \log(n/2).\end{aligned}$$

We can further improve the constants by applying Stirling's formula instead of this crude approximation. Remember that *Stirling's formula* gives the following approximation:

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1 + \Theta(n^{-1})\right) > \left(\frac{n}{e}\right)^n,$$

so $\log_2(n!) > n\log_2(n/e)$.

4.5 A Merging Lower Bound

Closely related to the sorting problem is the merging problem: Given two sorted sequences A and B, the merging problem is to combine these sequences into a sorted one. To apply the argument we used for sorting, we'll need to count how many possible outcomes the comparison operation can produce.

Suppose $n = |A|$, $m = |B|$, and $m \leq n$. We'll also assume that these sequences are made up of unique elements. Now observe that we have not made any comparison between elements of A and

B. This means any interleaving sequence A's and B's elements is possible. Therefore, the number of possible merged outcomes is the number of ways to choose n positions out from $n+m$ positions to put A's elements; this is simply $\binom{n+m}{n}$. Hence, we'll need, in the worst case, at least $\log_2 \binom{n+m}{n}$ comparison queries to merge these sequences.

The following lemma gives a simple lower bound for $\binom{n}{r}$, so that we can simplify $\log_2 \binom{n+m}{n}$ to an expression that we recognize.

Lemma 4.4 (Binomial Lower Bound)**.**

$$\binom{n}{r} \geq \left(\frac{n}{r}\right)^r.$$

Proof. First, we recall that

$$\binom{n}{r} = \frac{n!}{r!(n-r)!} = \frac{n(n-1)(n-2)\ldots(n-r+1)}{r(r-1)(r-2)\ldots 1} = \prod_{i=0}^{r-1} \frac{n-i}{r-i}.$$

We'll argue that for $0 \leq i < \min(r,n)$, $\frac{n-i}{r-i} \geq \frac{n}{r}$. Notice that

$$\frac{n-i}{r-i} \geq \frac{n}{r} \iff r(n-i) \geq n(r-i) \iff rn - ri \geq nr - ni \iff (n-r)i \geq 0.$$

Therefore, we have $\binom{n}{r} \geq \prod_{i=0}^{r-1} \frac{n-i}{r-i} \geq (n/r)^r$. □

With this lemma, we conclude that the number of comparison queries needed to merge sequences of lengths m and n ($m \leq n$) is at least

$$\log_2 \binom{n+m}{m} \geq m \log_2 \left(1 + \tfrac{n}{m}\right),$$

proving Theorem 4.2

Lecture 20 — Search Trees I: BSTs — Split, Join, and Union

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Margaret Reid-Miller — 1 November 2012

This lecture covers *binary search trees* and how to use them to implement sets and tables.

1 Binary Search Trees (BSTs)

Search trees are tree-based data structures that can be used to store and search for items that satisfy a total order. There are many types of search trees designed for a wide variety of purposes. Probably, the most common use is to implement sets and tables (dictionaries, mappings). As shown on right, a *binary tree* is a tree in which every node in the tree has at most two children. A *binary search tree* (BST) is a binary tree satisfying the following "search" property: for each node v, the key of the left child of v is smaller than the key of v and the key of the right child of v is bigger than the key of v; that is, for the tree in the figure on right, we have $k_L < k < k_R$. This ordering is useful navigating the tree.

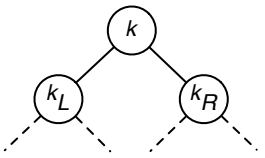


Figure 1: a binary tree

Approximately Balanced Trees. If search trees are kept "balanced" in some way, then they can usually be used to get good bounds on the work and span for accessing and updating them. We refer to such trees as *balanced search trees*. If trees are never updated but only used for searching, then balancing is easy—it needs only be done once—but what makes balanced trees interesting is their ability to efficiently maintain balance even when updated. To allow for efficient updates, balanced search trees do not require that the trees be strictly balanced, but rather that they are approximately balanced in some way. It would be impossible to maintain a perfectly balanced tree while allowing efficient (e.g. $O(\log n)$) updates.

Dozens of balanced search trees have been suggested over the years, dating back to at least AVL trees in 1962. The trees mostly differ in how they maintain balance. Most trees either try to maintain height balance (the two children of a node are about the same height) or weight balance (the two children of a node are about the same size, i.e., the number of elements in the subtrees). Let us list a few balanced trees:

1. *AVL trees.* Binary search trees in which the two children of each node differ in height by at most 1.
2. *Red-Black trees.* Binary search trees with a somewhat looser height balance criteria.
3. *2–3 and 2–3–4 trees.* Trees with perfect height balance but the nodes can have different number of children so they might not be weight balanced. These are isomorphic to red-black trees by grouping each black node with its red children, if any.

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

4. *B-trees.* A generalization of 2–3–4 trees that allow for a large branching factor, sometimes up to 1000s of children. Due to their large branching factor, they are well-suited for storing data on disks.

5. *Weight balanced trees.* Trees in which each node's children have sizes all within a constant factor. These are most typically binary, but can also have other branching factors.

6. *Treaps.* A binary search tree that uses random priorities associated with every element to maintain balance.

7. *Random search trees.* A variant on treaps in which priorities are not used, but random decisions are made with probabilities based on tree sizes.

8. *Skip trees.* A randomized search tree in which nodes are promoted to higher levels based on flipping coins. These are related to skip lists, which are not technically trees but are also used as a search structure.

9. *Splay trees.*[1] Binary search trees that are only balanced in the amortized sense (i.e. on average across multiple operations).

Traditionally, treatments of binary search trees concentrate on three operations: `search`, `insert`, and `delete`. Out of these, search is naturally parallel since any number of searches can proceed in parallel with no conflicts[2]. Insert and delete, however, are inherently sequential, as normally described. For this reason, we'll discuss more general operations that are useful for parallel updates and of which insert and delete are just a special case.

1.1 BST Basic Operations

We'll mostly focus on binary search trees in this class. A BST is defined by structural induction as either a leaf; or a node consisting of a left child, a right child, a key, and optional additional data. That is, we have

```
datatype BST = Leaf | Node of (BST * BST * key * data)
```

The data is for auxiliary information such as the size of the subtree, balance information, and a value associated with the key. The keys stored at the nodes must come from a total ordered set A. For all vertices v of a BST, we require that all values in the left subtree are less than v and all values in the right subtree are greater than v. This is sometimes called the binary search tree (BST) property, or the ordering invariant.

We'll rely on the following two basic building blocks to build up other functions, such as `search`, `insert`, and `delete`, but also many other useful functions such as intersection and union on sets.

`split`(T, k) : `BST` $\times$ `key` $\rightarrow$ `BST` $\times$ (`data option`) $\times$ `BST`

Given a BST T and key k, `split` divides T into two BSTs, one consisting of all the keys from T

[1] Splay trees were invented 1985 by Daniel Sleator and Robert Tarjan. Danny Sleator is a professor of computer science at Carnegie Mellon.

[2] In splay trees and other self-adjusting trees, this is not true since a searches can modify the tree.

less than k and the other all the keys greater than k. Furthermore if k appears in the tree with associated data d then `split` returns `SOME`(d), and otherwise it returns `NONE`.

`join`(L, m, R) : `BST` $\times$ (`key` $\times$ `data`) `option` $\times$ `BST` $\rightarrow$ `BST`
This function takes a left BST L, an optional middle key-data pair m, and a right BST R. It requires that all keys in L are less than all keys in R. Furthermore if the optional middle element is supplied, then its key must be larger than any in L and less than any in R. It creates a new BST which is the union of L, R and the optional m.

For both split and join we assume that the BST taken and returned by the functions obey some balance criteria. For example they might be red black trees. To maintain abstraction over the particular additional data needed to maintain balance (e.g. the color for a red-black tree) we assume the following function to expose the root of a tree without the balance data:

`expose`(T) : `BST` $\rightarrow$ (`BST` $\times$ `BST` $\times$ `key` $\times$ `data`) `option`
Given a BST T, if T is empty it returns `NONE`. Otherwise it returns the left child of the root, the right child of the root, and the key and data stored at the root.

With these functions, we can implement `search`, `insert`, and `delete`. As our first example, here is how we implement `search`:

```
1  fun search T k =
2  let val (_, v, _) = split(T, k)
3  in v
4  end
```

```
1  fun insert T (k, v) =
2  let val (L, v', R) = split(T, k)
3  in join(L, SOME(k, v), R)
4  end
```

```
1  fun delete T k =
2  let val (L, _, R) = split(T, k)
3  in join(L, NONE, R)
4  end
```

Exercise 1. *Write a version of* `insert` *that takes a function* f : `data` $\times$ `data` *and if the insertion key* k *is already in the tree applies* f *to the old and new data.*

As we will show later, implementing search, insert and delete in terms of these other operations is asymptotically no more expensive than a direct implementation. However, there might be some constant factor overhead so in an optimized implementation they could be implemented directly.

2 How to implement split and join on a simple BST?

We now consider a concrete implementation of `split` and `join` for a particular BST. For simplicity, we consider a version with no balance criteria. For the tree, we declare the following data type:

```
datatype BST = Leaf | Node of (BST * BST * key * data)
```

```
1  fun split(T, k) =
2    case T of
3      Leaf  ⇒ (Leaf, NONE, Leaf)
4    | Node(L, R, k′, v) ⇒
5        case compare(k, k′) of
6          LESS ⇒
7            let val (L′, r, R′) = split(L, k)
8            in (L′, r, Node(R′, R, k′, v)) end
9          EQUAL ⇒ (L, SOME(v), R)
10         GREATER ⇒
11           let val (L′, r, R′) = split(R, k)
12           in (Node(L, L′, k′, v), r, R′) end
```

```
1  fun join(T1, m, T2) =
2    case m of
3      SOME(k, v) ⇒ Node(T1, T2, k, v)
4    | NONE ⇒
5        case T1 of
6          Leaf ⇒ T2
7        | Node(L, R, k, v) ⇒ Node(L, join(R, NONE, T2), k, v))
```

3 Union

Let's now consider a more interesting operation: taking the union of two BSTs. Note that this differs from `join` since we do not require that all the keys in one appear after the keys in the other. The code below implements the union function:

```
1  fun union(T1, T2) =
2    case expose(T1) of
3      NONE ⇒ T2
4    | SOME(L1, R1, k1, v1) ⇒
5      let val (L2, v2, R2) = split(T2, k1)
6          val (L, R) = union(L1, L2) || union(R1, R2)
7      in join(L, SOME(k1, v1), R)
8      end
```

For simplicity, this version returns the value from T_1 if a key appears in both BSTs. We'll analyze the cost of `union` next. The code for set intersection and set difference is quite similar.

3.1 Cost of Union

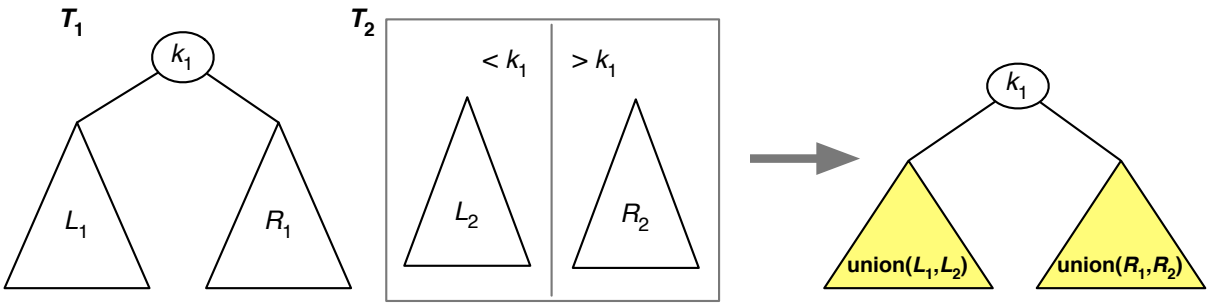
In the 15-210 library, `union` and similar functions (e.g., `intersection` and `difference` on sets and `merge`, `extract` and `erase` on tables) have $O(m \log(1 + \frac{n}{m}))$ work, where m is the size of the smaller input and n the size of the larger one. As you saw in recitation, this bound is the same as the lower bound for merging two sorted sequences. We will see how this bound falls out very naturally from the `union` code.

To analyze this, we'll first assume that the work and span of `split` and `join` is proportional to the depth of the input tree and the output tree, respectively. In a reasonable implementation, these operations traverse a path in the tree (or trees in the case of `join`). Therefore, if the trees are reasonably balanced and have depth $O(\log n)$, then the work and span of both `split` and `join` on a tree of n nodes is $O(\log n)$. Indeed, most balanced trees have $O(\log n)$ depth.

The `union` algorithm we just wrote has the following basic structure. On input T_1 and T_2, the function $\texttt{union}(T_1, T_2)$ performs:

1. For T_1 with key k_1 and children L_1 and R_1 at the root, use k_1 to split T_2 into L_2 and R_2.
2. Recursively find $L_u = \texttt{union}(L_1, L_2)$ and $R_u = \texttt{union}(R_1, R_2)$.
3. Now $\texttt{join}(L_u, k_1, R_u)$.

Pictorially, the process looks like this:



We'll begin the analysis by examining the cost of each `union` call. Notice that each call to `union` makes one call to `split` costing $O(\log |T_2|)$ and one to `join`, each costing $O(\log(|T_1| + |T_2|))$. To ease the analysis, we will make the following assumptions:

1. T_1 it is perfectly balanced (i.e., `expose` returns subtrees of size $n/2$), and
2. Each time a key from T_1 splits T_2, it splits the tree exactly in half.

With these assumptions, we can write a recurrence for the work of `union` as follows:

$$W(|T_1|, |T_2|) = 2W(|T_1|/2, |T_2|/2) + O(\log(|T_1| + |T_2|)),$$

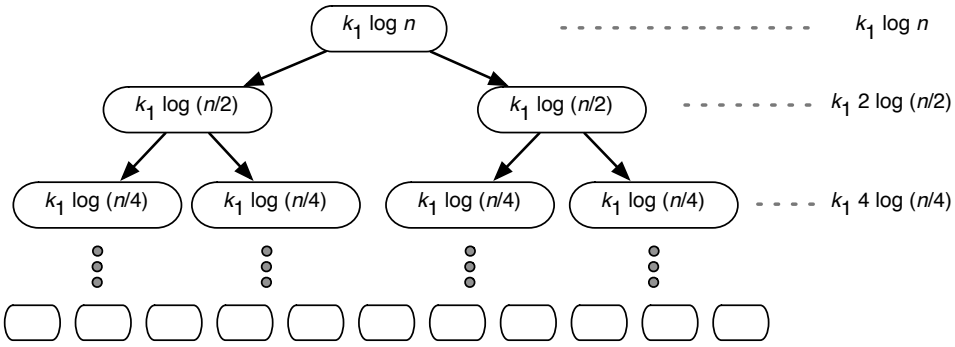
and

$$W(1, |T_2|) = O(\log(1 + |T_2|)).$$

This recurrence deserves more explanation: When $|T_1| > 1$, `expose` gives us a perfect split, resulting in a key k_1 and two subtrees of size $|T_1|/2$ each—and by our assumption (which we'll soon eliminate), k_1 splits T_2 perfectly in half, so the subtrees `split` that produces have size $|T_2|/2$.

When $|T_1| = 1$, we know that `expose` give us two empty subtrees L_1 and R_1, which means that both `union`(L_1, L_2) and `union`(R_1, R_2) will return immediately with values L_2 and R_2, respectively. Joining these together with T_1 costs at most $O(\log(|T_1| + |T_2|))$. Therefore, when $|T_1| = 1$, the cost of `union` (which involves one `split` and one `join`) is $O(\log(1 + |T_2|))$.

Let $m = |T_1|$ and $n = |T_2|$. If we draw the recursion tree that shows the work associated with splitting T_2 and joining the results, we obtain the following:



There are several features of this tree that's worth mentioning: First, ignoring the somewhat-peculiar cost in the base case, we know that this tree is leaf-dominated. Therefore, excluding the cost at the bottom level, the cost of `union` is $O(\#$ of leaves$)$.

But how many leaves are there? And how deep is this tree? To find the number of leaves, we'll take a closer look at the work recurrence. Notice that in the recurrence, the tree bottoms out when $|T_1| = 1$ and before that, T_1 always gets split in half (remember that T_1 is perfectly balanced). Nowhere in there does T_2 affects the shape of the recursion tree or the stopping condition. Therefore, this is yet another recurrence of the form $f(m) = f(m/2) + O(...)$, which means that *it has m leaves and is $1 + \log_2 m$ deep.*

Next, we'll determine the size of T_2 at the leaves. Remember that as we descend down the recursion tree, the size of T_2 gets halved, so the size of T_2 at a node at level i (counting from 0) is $n/2^i$. But we knew already that the depth of the recursion tree is $\log_2 m$, so the size of T_2 at each of the leaves is

$$n/2^{\log_2 m} = \frac{n}{m}.$$

Therefore, each leaf node costs $O(\log(1 + \frac{n}{m}))$. Since there are m leaves, the whole bottom level costs $O(m \log(1 + \frac{n}{m}))$. Hence, if the trees satisfy our assumptions, we have that `union` runs in $O(m \log(1 + \frac{n}{m}))$ work.

Removing An Assumption: Of course, in reality, our keys in T_1 won't split subtrees of T_2 in half every time. But it turns out this only helps. We won't go through a rigorous argument, but if we keep the assumption that T_1 is perfectly balanced, then the shape of the recursion tree stays the same. What is now different is the cost at each level. Let's try to analyze the cost at level i. At this level, there are $k = 2^i$ nodes in the recursion tree. Say the sizes of T_2 at these nodes are $n_1, \ldots, n_k$, where $\sum_j n_j = n$. Then, the total cost for this level is

$$c \cdot \sum_{j=1}^{k} \log(n_j) \;\leq\; c \cdot \sum_{j=1}^{k} \log(n/k) = c \cdot 2^i \cdot \log(n/2^i),$$

where we used the fact that the logarithm function is concave[3]. Thus, the tree remains leaf-dominated and the same reasoning shows that the total work is $O(m \log(1 + \frac{n}{m}))$.

Still, in reality, T_1 doesn't have to be perfectly balanced as we assumed. A similar reasoning can be used to show that T_1 only has to be approximately balanced. We will leave this case as an exercise. We'll end by remarking that as described, the span of `union` is $O(\log^2 n)$, but this can be improved to $O(\log n)$ by changing the the algorithm slightly.

In summary, `union` can be implemented in $O(m \log(1 + \frac{n}{m}))$ work and span $O(\log n)$. The same holds for the other similar operations (e.g. `intersection`).

[3]Technically, we're applying the so-called Jensen's inequality.

Lecture 21 — Treaps

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Margaret Reid-Miller — 6 November 2012

1 Quick Review: Binary Search Trees

For a quick recap, recall that in the last lecture, we were talking about binary search trees (BST). In particular, we looked at the following:

- *Many ways to keep a search tree almost balanced.* Such trees include red-black trees, 2-3 trees, B-trees, AVL trees, Splay trees, Treaps, weight balanced trees, skip trees, among others.

  Some of these are binary, some are not. In general, a node with k children will hold $k-1$ keys. But in this course, we will restrict ourselves to binary search trees.

- *Using* `split` *and* `join` *to implement other operations.* The `split` and `join` operations can be used to implement most other operations on binary search trees, including: `search`, `insert`, `delete`, `union`, `intersection` and `difference`.

- *An implementation of* `split` *and* `join` *on unbalanced trees.* We claim that the same idea can also be easily implemented on just about any of the balanced trees.

- *Cost of* `union` We showed that when both trees are split evenly at every level of the recursion tree, the work for `union` is $O(m \log(1 + \frac{n}{m})$, where $m \leq n$. If they don't split evenly, it only makes less work.

2 Today

Today we introduce a balanced binary search tree that is closely related to randomized quicksort. First we will look at the connection between quicksort and binary search trees, and then we will introduce treaps, which is a binary search tree that uses randomization to maintain balance.

3 Quicksort and BSTs

Can we think of binary search trees in terms of an algorithm we already know? As is turns out, the quicksort algorithm and binary search trees are closely related: if we write out the recursion tree for quicksort and annotate each node with the pivot it picks, what we get is a BST.

Let's try to convince ourselves that the function-call tree for quicksort generates a binary search tree when the keys are distinct. To do this, we'll modify the quicksort code from a earlier lecture

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

to produce the tree as we just described. In this implementation, we intentionally leave the pivot-choosing step unspecified because the property we are discussing holds regardless of the choice of the pivot.

```
1  fun qs_tree(S) =
2    if |S| = 0 then LEAF
3    else let
4           val p = pick a pivot from S
5           val S1 = ⟨ s ∈ S | s < p ⟩
6           val S2 = ⟨ s ∈ S | s > p ⟩
7           val (TL, TR) = (qs_tree(S1) || qs_tree(S3))
8         in
9           NODE(TL, p, TR)
10        end
```

Notice that this is clearly a binary tree. To show that this is a binary *search* tree, we only have to consider the ordering invariant. But this, too, is easy to see: for `qs_tree` call, we compute S_1, whose elements are strictly smaller than p—and S_2, whose elements are strictly bigger than p. So, the tree we construct has the ordering invariant. In fact, this is an algorithm that converts a sequence into a binary search tree.

It clear that, whatever the pivot strategy is, the maximum depth of the binary search tree resulting from `qs_tree` is the same as the maximum depth of the recursion tree for quicksort using that strategy. As shown in lecture, the expected depth of the recursion tree for *randomized* quicksort is $O(\log n)$

Can we maintain a tree data structure that centers on this random pivot-selection idea? If so, we automatically get a nice BST.

4 Treaps

Unlike quicksort, when building a BST we don't necessarily know all the elements that will be in the BST at the start, so we can't randomly pick an element (in the future) to be the root of the BST. So how can we use randomization to help maintain balance in a BST?

A treap (tree + heap) is a randomized BST that maintains balance in a probabilistic way. In a treap, a random priority is assigned to every key. In practice, this is often done by performing a hash on a key, or by assigning a different random number to each key. We will assume that each priority is unique, although it is possible to remove this assumption.

The nodes in a treap must satisfy two properties:

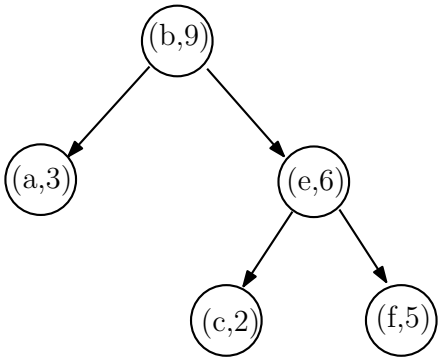
BST Property: Their keys satisfy the BST property (i.e., keys are stored in-order in the tree).

Heap Property: The associated priorities satisfy the *heap* property. The (max) heap property requires for every node that the value at a node is greater than the value of its two children.

Consider the following key-priority pairs:

(a,3), (b,9), (c, 2), (e,6), (f, 5)

These elements would be placed in the following treap.



Theorem 4.1. *For any set S of unique key-priority pairs, there is exactly one treap T containing the key-priority pairs in S which satisfies the treap properties.*

Proof. The key k with the highest priority in S must be the root node, since otherwise the tree would not be in heap order. Only one key has the highest priority. Then, to satisfy the property that the treap is ordered with respect to the nodes' keys, all keys in S less than k must be in the left subtree, and all keys greater than k must be in the right subtree. Inductively, the two subtrees of k must be constructed in the same manner. □

Note, there is a subtle distinction here with respect to randomization. With quicksort the algorithm is randomized. With treaps, none of the functions for treaps are randomized. It is the data structure itself that is randomized[1].

Split and Join on Treaps

As mentioned in the last lecture, for any binary tree all we need to implement is split and join and these can be used to implement the other BST operations. Recall that split takes a BST and a key and splits the BST into two BST and an optional value. One BST only has keys that are less than the given key, the other BST only has keys that are greater than the given key, and the optional value is the value of the given key, if it is the tree. Join takes two BSTs and an optional middle (key,value) pair, where the maximum key on the first tree is less than the minimum key on the second tree. It returns a BST that contains all the keys the given BSTs and middle key.

We claim that the split code given in the last lecture for unbalanced trees does not need to be modified for treaps.

Exercise 1. *Convince yourselves than when doing a split none of the priority orders change (i.e. the code will never put a larger priority below a smaller priority).*

[1] In contrast, for the so called Randomize Binary Search Trees, it is the functions that update the tree that are randomized.

The join code, however, does need to be changed. The new version has to check the priorities of the two roots, and use whichever is greater as the new root. In the algorithm shown below, we assume that the priority of a key can be computed from the key (e.g., priorities are a hash of the key).

```
1  fun join(T1, m, T2) =
2  let
3    fun singleton(k, v) = Node(Leaf, Leaf, k, v)
4    fun join'(T1, T2) =
5      case (T1, T2) of
6        (Leaf, _) ⇒ T2
7      | (_, Leaf) ⇒ T1
8      | (Node(L1, R1, k1, v1), Node(L2, R2, k2, v2)) ⇒
9          if (priority(k1) > priority(k2)) then
10           Node(L1, join'(R1, T2), k1, v1)
11         else
12           Node(join'(T1, L2), R2, k2, v2)
13 in
14   case m of
15     NONE ⇒ join'(T1, T2))
16   | SOME(k, v) ⇒ join'(T1, join'(singleton(k, v), T2))
17 end
```

In the code join′ is a version of join that has no middle element as an argument. Note that line 9 compares the priorities of the two roots and then places the key with the larger priority in the new root causing a recursive call to join on one of the two sides.

We refer to the *left spine* of the tree as the path from the root to the leftmost node in the tree, and the *right spine* as the path from the root to the rightmost node in the tree. What join′(T_1, T_2) does is to interleave pieces of the right spine of T_1 with pieces the left spine of T_2, where the size of each piece depends on the priorities.

Because the keys and priorities determines a treap uniquely, repeated splits and joins on the same key, results in the same treap. This property is not always true of most other kinds of balanced trees; the order that operations are applied can change the shape of the tree.

Because the cost of split and join depends on the depth of the i^{th} element in a treap, we now analyze the expected depth of a key in the tree.

5 Expected Depth of a Key in a Treap

Consider a set of keys K and associated priorities p : key $\rightarrow int$. For this analysis, we assume the priorities are unique. Consider the keys laid out in order, and as with the analysis of quicksort, we use i and j to refer to the i^{th} and j^{th} keys in this ordering. Unlike quicksort analysis, though, when analyzing the depth of a node i, i j can be in any order, since an ancestor in a BST can be either less than or greater than node i.

If we calculate the depth starting with zero at the root, the expected depth of a key is equivalent to the number of ancestors it has in the tree. So we want to know how many ancestors a particular node i has. We use the indicator random variable A_i^j to indicate that j is an ancestor of i. (Note that the superscript here does not mean A_i is raised to the power j; it simply is a reminder that j is the ancestor of i.) By the linearity of expectatons, the expected depth of i can be written as:

$$\mathbf{E}\left[\text{depth of } i \text{ in } T\right] = \mathbf{E}\left[\sum_{j=1}^{n} A_i^j\right] = \sum_{j=1}^{n} \mathbf{E}\left[A_i^j\right].$$

To analyze A_i^j let's just consider the $|j-i|+1$ keys and associated priorities from i to j inclusive of both ends. As with the analysis of quicksort, if an element k has the highest priority and k is less than both i and j or greater than both i and j, it plays no role in whether j is an ancestor of i or not. The following three cases do:

1. The element i has the highest priority.
2. One of the elements k in the middle has the highest priority (i.e., neither i nor j).
3. The element j has the highest priority.

What happens in each case?

1. If i has the highest priority then j cannot be an ancestor of i, and $A_i^j = 0$.
2. If k between i and j has the highest priority, then $A_i^j = 0$, also. Suppose it was not. Then, as j is an ancestor of i, it must also be an ancestor of k. That is, since in a BST every branch covers a contiguous region, if i is in the left (or right) branch of j, then k must also be. But since the priority of k is larger than that of j this cannot be the case, so j is not an ancestor of i.
3. If j has the highest priority, j must be an ancestor of i and $A_i^j = 1$. Otherwise, to separate i from j would require a key in between with a higher priority. We therefore have that j is an ancestor of i exactly when it has a priority greater than all elements from i to j (inclusive on both sides).

Therefore j is an ancestor of i if and only if it has the highest priority of the keys between i and j, inclusive. Because priorities are selected randomly, there a chance of $1/(|j-i|+1)$ that $A_i^j = 1$ and we have $\mathbf{E}\left[A_i^j\right] = \frac{1}{|j-i|+1}$. (Note that if we include the probability of either j being an ancestor of i or i being an ancestor of j then the analysis is identical to quicksort. Think about why.)

Now we have

$$\begin{aligned}
\mathbf{E}\left[\text{depth of } i \text{ in } T\right] &= \sum_{j=1, j\neq i}^{n} \frac{1}{|j-i|+1} \\
&= \sum_{j=1}^{i-1} \frac{1}{i-j+1} + \sum_{j=i+1}^{n} \frac{1}{j-i+1} \\
&= \sum_{k=2}^{i} \frac{1}{k} + \sum_{k=2}^{n-i+1} \frac{1}{k} \\
&= H_i - 1 + H_{n-i+1} - 1 \\
&< \ln i + \ln(n-i+1) \\
&= O(\log n)
\end{aligned}$$

Recall that the harmonic number is $H_n = \sum_{i=1}^{n} \frac{1}{n}$. It has the following bounds: $\ln n < H_n < \ln n + 1$, where $\ln n = \log_e n$. Notice that the expected depth of a key in the treap is determined solely by it relative position in the sorted keys.

Exercise 2. *Including constant factors how does the expected depth for the first key compare to the expected depth of the middle ($i = n/2$) key?*

Theorem 5.1. *For treaps the cost of* `join`(T_1, m, T_2) *returning* T *and of* `split`$(T, (k, v))$ *is* $O(\log |T|)$ *expected work and span.*

Proof. The `split` operation only traverses the path from the root down to the node at which the key lies or to a leaf if it is not in the tree. The work and span are proportional to this path length. Since the expected depth of a node is $O(\log n)$, the expected cost of split is $O(\log n)$.

For `join`(T_1, m, T_2) the code traverses only the right spine of T_1 or the left spine of T_2. Therefore the work is at most proportional to the sum of the depth of the rightmost key in T_1 and the depth of the leftmost key in T_2. The work of `join` is therefore the sum of the expected depth of these nodes. Since the resulting treap T is an interleaving of these spines, the expected depth is bound by $O(\log |T|)$. □

5.1 Expected overall depth of treaps

Even though the expected depth of a node in a treap is $O(\log n)$, it does not tell us what the expected maximum depth of a treap is. As you have saw in lecture 15, $\mathbf{E}\left[\max_i\{A_i\}\right] \neq \max_i\{\mathbf{E}\left[A_i\right]\}$. As you might surmise, the analysis for the expected depth is identical to the analysis of the expected span of randomized quicksort, except the recurrence uses 1 instead of $c \log n$. That is, the depth of the recursion tree for randomized quicksort is $D(n) = D(Y_n) + 1$, where Y_n is the size of the larger partition. Thus, the expected depth is $O(\log n)$.

It turns out that is possible to say something stronger: For a treap with n keys, the probability that any key is deeper than $10 \ln n$ is at most $1/n^2$[2]. That is, for large n a treap with random priorities

[2]The bound base on Chernoff bounds which relies on events being independent.

has depth $O(\log n)$ with *high probability*. It also implies that randomized quicksort $O(n \log n)$ work and $O(\log^2 n)$ span bounds hold with high probability.

Being able to put high probability bounds on the runtime of an algorithm can be critical in some situations. For example, suppose my company DontCrash is selling you a new air traffic control system and I say that in expectation, no two planes will get closer than 500 meters of each other—would you be satisfied? More relevant to this class, let's say you wanted to run 1000 jobs on 1000 processors and I told you that in expectation each finishes in an hour—would you be happy? How long might you have to wait?

There are two problems with expectations, at least on their own. Firstly, they tell us very little if anything about the variance. And secondly, as mentioned in an earlier lecture, the expectation of a maximum can be much higher than the maximum of expectations. The first has implications in real time systems where we need to get things done in time, and the second in getting efficient parallel algorithms (e.g., span is the max span of the two parallel calls). Proving these high probability bounds is beyond the scope of this course.

Summary

Earlier we showed that randomized quicksort has worst-case expected $O(n \log n)$ work, and this expectation was independent of the input. That is, there is no bad input that would cause the work to be worse than $O(n \log n)$ all the time. It is possible, however, (with extremely low probability) we could be unlucky, and the random chosen pivots could result in quicksort taking $O(n^2)$ work.

It turns out the same analysis shows that a deterministic quicksort will on average have $O(n \log n)$ work. Just shuffle the input randomly, and run the algorithm. It behaves the same way as randomized quicksort on that shuffled input. Unfortunately, on some inputs (e.g., almost sorted) the deterministic quicksort is slow, $O(n^2)$, every time on that input.

Treaps take advantage of the same randomization idea. But a binary search tree is a dynamic data structure, and it cannot change the order in which operations are applied to it. So instead of randomizing the input order, it adds randomization to the data structure itself.

Lecture 23 — Fun With Trees

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — Nov 12, 2012

Today:

- Ordered Sets
- Augmenting Balanced Trees

1 Ordered Sets and Tables

The set and table interfaces described so far do not give any operations that make use of the ordering of the elements. This allows it to be defined on types that don't have a natural ordering. It is also well suited for an implementation based on hash tables. In many applications, however, it is useful to take advantage of the order of the keys. For example in a database one might want to find all the customers who spent between \$50 and \$100, all emails in the week of Aug 22, or the last stock transaction before noon on October 11th. Here we assume the data is organized by transaction value, date or any other ordered key.

For these purposes we can extend the operations on sets and tables with some additional operations that take advantage of ordering. Here we will just describe the operations on ordered sets. The operations on ordered tables are completely analogous.

Definition 1.1 (Ordered Set ADT). For a totally ordered universe of elements $\mathbb{U}$ (e.g. the integers or strings), the *Ordered Set* abstract data type is a type $\mathbb{S}$ representing the powerset of $\mathbb{U}$ (i.e., all subsets of $\mathbb{U}$) along with the following functions:

all operations supported by the Set ADT, and

`last`(S)	:	$\mathbb{S} \to \mathbb{U}$	=	$\max S$
`first`(S)	:	$\mathbb{S} \to \mathbb{U}$	=	$\min S$
`split`(S,k)	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{S} \times bool \times \mathbb{S}$	=	$(\{k' \in S \mid k' < k\}, k \stackrel{?}{\in} S, \{k' \in S \mid k' > k\})$
`join`(S_1,S_2)	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	=	$S_1 \cup S_2$, assuming $\max S_1 < \min S_2$
`getRange`(S,k_1,k_2)	:	$\mathbb{S} \times \mathbb{U} \times \mathbb{U} \to \mathbb{S}$	=	$\{k \in S \mid k_1 \leq k \leq k_2\}$

Note that `split` and `join` are the same as the operations we defined for binary search trees. Here, however, we are abstracting the notion to ordered sets.

If we implement using trees, then we can use the tree implementations of `split` and `join` directly. Implementing `first` is straightforward since it only requires traversing the tree down the left branches until a left branch is empty. Similarly `last` need only traverse right branches. The `getRange` operation can easily be implemented with two calls to `split`.

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

Example Application: Bingle® Revisited

Recall that when we first discussed sets and tables we used an example of supporting an index for searching for documents based on the terms that appear in them. The motivation was to generate a search capability similar to what is supported by Bing, Google and other search engines. The interface we described was:

```
signature INDEX = sig
  type word = string
  type docId = string
  type 'a seq
  type index
  type docList

  val makeIndex : (docId * string) seq -> index
  val find : index -> word -> docList
  val And : docList * docList -> docList
  val AndNot : docList * docList -> docList
  val Or : docList * docList -> docList
  val size : docList -> int
  val toSeq : docList -> docId seq
end
```

We implemented this interface by using a table that mapped each word to the set of documents it appeared in. The `docList` type is therefore a `set` and the `index` type is a `set table`, where the sets are indexed by document and the table by words. We could then use `intersection`, `union`, and `difference` to implement `and`, `or` and `andNot` respectively. The interface and implementation make no use of any ordering of the documents.

Now lets say we want to augment the interface to support queries that restrict the search to certain domains, such as `cs.cmu.edu` or `cmu.edu` or even `.edu`. The function we want to add is

```
val inDomain : domain * docList -> docList
```

Given an index `idx` we could then do a search of the form

```
inDomain("cs.cmu.edu", and(find idx "cool", find idx "TAs"))
```

and it would return documents about all the cool TAs in CS.

Lets look at how to implement this with ordered sets. Our interface associates a `docId` with every document and let us assume these IDs are the URL of the document. Since URLs are strings, and strings have a total order, we can use an ordered set to implement the `docList` type. However instead of keeping the sets of URLs ordered "lexicographically", lets keep them ordered in backwards lexicographic order (i.e., the last character is the most significant). Given this order all documents within the same domain will be contiguous.

This suggests a simple implementation of `inDomain`. In particular we can use `getRange` to extract just the URLs in the `docList` that match the domain by "cutting out" the appropriate contiguous range. In particular we have:

```
1  fun inDomain(domain, L) =
2    getRange(L, domain, string.append(domain, "$"))
```

where $ is a character greater than any character appearing in a URL. Note that this extracts precisely the documents that match the URL since strings between `domain` and `string.append(domain, "$")` are precisely the strings that match the domain.

There are many other applications of ordered sets and tables including some which will be discussed in the following section.

2 Augmenting Balanced Trees

Often it is useful to include additional information beyond the key and associated value in a tree. In particular the additional information can help us efficiently implement additional operations. Here we will consider two examples: (1) locating positions within an ordered set or ordered table, and (2) keeping "reduced" values in an ordered or unordered table.

2.1 Tracking Sizes and Locating Positions

Lets say that we are using binary search trees (BSTs) to implement ordered sets and that in addition to the operations already described, we also want to efficiently support the following operations:

$$\begin{array}{lcll} \texttt{rank}(S,k) & : & \mathbb{S}\times\mathbb{U}\to \mathit{int} & = & |\{k' \in S \mid k' < k\}| \\ \texttt{select}(S,i) & : & \mathbb{S}\times \mathit{int}\to\mathbb{U} & = & k \text{ such that } |\{k' \in S \mid k' < k\}| = i \\ \texttt{splitIdx}(S,i) & : & \mathbb{S}\times \mathit{int}\to\mathbb{S}\times\mathbb{S} & = & (\{k \in S \mid k < \texttt{select}(S,i)\}, \\ & & & & \ \{k \in S \mid k \geq \texttt{select}(S,i)\}) \end{array}$$

In the previous lectures the only things we stored at the nodes of a tree were the left and right children, the key and value, and perhaps some balance information. With just this information implementing the `select` and `splitIdx` operations requires visiting all nodes before the i^{th} location to count them up. There is no way to know the size of a subtree without visiting it. Similarly `rank` requires visiting all nodes before k. Therefore all these operations will take $\Theta(|S|)$ work. In fact even implementing `size`(S) requires $\Theta(|S|)$ work.

To fix this problem we can add to each node an additional field that specifies the size of the subtree. Clearly this makes the `size` operation fast, but what about the other operations? Well it turns out it allows us to implement `select`, `rank`, and `splitIdx` all in $O(d)$ work assuming the tree has depth d. Therefore for balanced trees the work will be $O(\log|S|)$. The `select` function can be implemented on a binary search tree as show in Figure 1.

To implement `rank` we could simply do a split and then check the size of the left tree. The implementation of `splitIdx` is similar to split except when deciding which branch to take, be base it on the sizes instead of the keys. In fact with `splitIdx` we don't even need select, we could implement it as a `splitIdx` followed by a `first` on the right tree.

```
1  fun select(T, i) =
2    case expose(T) of
3        NONE ⇒ raise Range
4      | SOME(L, R, k) ⇒
5          case compare(i, |L|) of
6            LESS ⇒ select(L, i)
7          | EQUAL ⇒ k
8          | GREATER ⇒ select(R, i − |L| − 1)
```

Figure 1: Code for `select` based on binary search trees. It assumes you can take the size of a subtree T using $|T|$. The function `expose`(T) returns `NONE` if the tree is empty, or a tripple of the left and right subtrees and key if not.

We can implement sequences using a balanced tree using this approach. In fact you already saw this in 15-150 when you covered the tree implementation of sequences.

2.2 Ordered Tables with Reduced Values

A second application of augmenting trees is to dynamically maintain a sum (using an arbitrary associative operator f) over the values of a table while allowing for arbitrary updates, e.g. insert, delete, merge, extract, split, and join. It turns out that this ability is very useful for several applications. We will get back to the applications but first describe the interface and its implementation using binary search trees. Consider the following abstract data type that extends ordered tables with one additional operation.

Definition 2.1 (**Ordered table with reduced value ADT).** For a specified associative function $f : v \times v \to v$ and identity element I_f, an *ordered table with reduced values* supports all operations on ordered tables, e.g.

- empty, singleton, map, filter, reduce, iter, insert, delete, find, merge, extract, erase
- split, join, first, last, previous, next

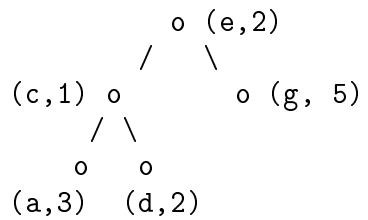
and in addition supports the operation

$$\texttt{reduceVal}(A) : \mathbb{T} \to v = \texttt{reduce}\ f\ I_f\ A$$

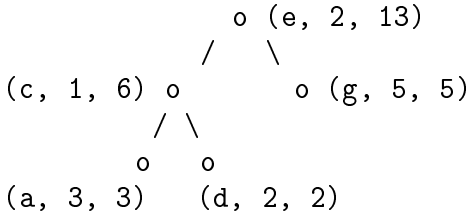
The `reduceVal`(T) function just returns the sum of all values in T using the associative operator f that is part of the data type. It might seem redundant to support this function since it can be implemented by the existing `reduce` function, but the idea is that we will be able to implement it much more efficiently by including it in the interface. In particular our goal is to support all the table operations in the same asymptotic bounds as we have previously used for the binary search tree implementation of ordered tables, but also support the `reduceVal` in $O(1)$ work.

You might ask how can we possibly implement a `reduce` in $O(1)$ work on a table of arbitrary size? The trick is to make use of the fact that the function f over which we are reducing is fixed ahead of time. This means that we can maintain the reduced value and update it whenever we do other operations on the ordered table. That way whenever we ask for the value it has already been computed and we only have to look it up. Using this approach the challenge is not in computing `reduceVal`, it just needs to return a precomputed value, but instead how to maintain this value whenever we do other operations on the table.

We now discuss how to maintain the reduced values on the binary search trees. The basic idea is simply to store with every node v of a binary search tree the reduced value of its subtree (i.e. the sum of all the values that are descendants of v as well as the value at v itself. For example lets assume our function f is addition, and we are given the following BST mapping character keys to integer values:



Now when we associate the reduced values with each node we get



Note that the value at each node can simply be calculated by summing the reduced value in each of the two children and the value in the node. So, for example, the reduced value at the root is the sum of the left reduced value 6, the right reduced value 5, and the node value 2, giving 13. This means that we can maintain these reduced values by simply taking this "sum" of three values whenever creating a node. In turn this means that the only real change we have to make to existing code for implementing ordered tables with binary search trees is to do this sum whenever we make a node. If the work of f is constant, then this sum takes constant work.

Figure 2 gives code for `join` on treaps extended to work with reduced values. Note that the only difference in the `join` code is the use of `makeNode` instead of `Node`. Similar use of `makeNode` can be used in `split` and other operations on treaps. This idea can be used with any binary search tree, not just treaps. In all cases one needs only to replace the function that creates a node with a version that also sums the reduced values from the children and the value from the node to create a reduced value for the new node.

We note that in an imperative implementation of binary search trees in which a child node can be side affected, then the reduced values need to be recomputed on all nodes in the path from the modified node to the root.

We now consider several applications of ordered tables with reduced values.

```
1  datatype Treap = Leaf | Node of (Treap × Treap × key × data × data)

2  fun reduceVal(T) =
3    case T of
4      Leaf ⇒ Reduce.I
5    | Node(_, _, _, _, r) ⇒ r

6  fun makeNode(L, R, k, v) =
7    Node(L, R, k, v, Reduce.f(reduceVal(L), Reduce.f(v, reduceVal(R))))

8  fun join(T1, T2) =
9    case (T1, T2) of
10     (Leaf, _) ⇒ T2
11   | (_, Leaf) ⇒ T1
12   | (Node(L1, R1, k1, v1, s1), Node(L2, R2, k2, v2, s2)) ⇒
13       if (priority(k1) > priority(k2)) then
14         makeNode(L1, join(R1, T2), k1, v1)
15       else
16         makeNode(join(T1, L2), R2, k2, v2)
```

Figure 2: Extending `join` on treaps to work with reduced values stored at every node. In each `Node` the first `data` entry is the data stored at the node and the second is the reduced value.

Example: Analyzing Profits at TRAM⋆LAW®

Lets say that based on your expertise in algorithms you are hired as a consultant by the giant retailer **TRAM⋆LAW®**. Tramlaw sells over 10 billion items per year across its 8000+ stores. As with all major retailers, it keeps track of every sale it makes and analyzes these sales regularly to make business and marketing decisions. Lets say that the sale records it keeps consists of a timestamp when the sale was made, the amount of the sale and various other information associated with the sale.

Tramlaw wants to be able to quickly determine the total amount of sales within any range of time, e.g. between 5pm and 10pm last Friday, or during the whole month of September, or during the halftime break of the last Steeler's football game, or for the week after its latest promo. It uses this sort of information, for example, to decide on staffing levels or marketing strategy. It needs to maintain the database so that it can be updated quickly, including not just adding new sales to the end, but merging in information from all its stores, and possibly modifying old data (e.g. if some item is returned or a record is found to be in error).

How would you do this? Well after thinking a few minutes you remember ordered tables with reduced values from 210. You realize that if you keep the sales information keyed based on timestamps, and maintain the sum of sales amounts as the reduced values then all the operations required are cheap. In particular the function f is simply addition. Now the following will extract the sum in any range:

$$\texttt{reduceVal}(\texttt{getRange}(T, t_1, t_2))$$

This will take $O(\log n)$ work, which is much cheaper than n. Now lets say Tramlaw wanted to

do a somewhat more complicated query where they want to know the total sales between 5 and 7 pm on every day over the past year. You could do this by applying the query above once for each day. These can all be done in parallel and summed in parallel. The total work will be $365 \times O(\log n)$, which is still much cheaper than looking at all data over the past year.

Example: A Jig with QADSAN®

Now in your next consulting job **QADSAN®** hires you to more efficiently support queries on the their stock exchange data. For each stock they keep track of the time and amount of every trade. This data is updated as new trades are made. As with Tramlaw, tables might also need to be merged since they might come from different sources : e.g. the Tokyo stock exchange and the New York stock exchange. Qasdan wants to efficiently support queries that return the maximum price of a trade during any time range (t_1, t_2) .

You tell them that they can use an ordered table with reduced values using max as the combining function f . The query will be exactly the same as with your consulting jig with Tramlaw, `getRange` followed by `reduceVal`, and it will similarly run in $O(\log n)$ work.

Exercise 1. *Now lets say that **QADSAN®** also wants to support queries that given a time range return the maximum increase in stock value during that range (i.e. the largest difference between two trades in which the larger amount comes after the smaller amount). What function would you use for f to support such queries in $O(\log n)$ work.*

Example: Interval Queries

After your two consulting jobs, you are taking 15-451, with Professor Mulb. On a test he asks you to describe a data structure for representing an abstract data type called interval tables. An interval is a region on the real number line starting at x_l and ending at x_r, and an interval table supports the following operations on intervals:

$\texttt{insert}(A, I)$	:	$\mathbb{T} \times (\texttt{real} \times \texttt{real}) \to \mathbb{T}$	*insert interval I into table A*
$\texttt{delete}(A, I)$	:	$\mathbb{T} \times (\texttt{real} \times \texttt{real}) \to \mathbb{T}$	*delete interval I from table A*
$\texttt{count}(A, x)$	:	$\mathbb{T} \times \texttt{real} \to \texttt{int}$	*return the number of intervals crossing x in A*

Exercise 2. *How would you implement this.*

Other Applications of Reduced Values

- By using the parenthesis matching code in recitation 1 we could maintain whether a sequence of parenthesis are matched while allowing updates. Each update will take $O(\log n)$ work. In fact we could append two strings of parenthesis and check whether together they are matched in $O(\log n)$ work.
- By using the maximum contiguous subsequence sum problem described in class you could maintain the sum while updating the sequence by for example inserting new elements, deleting elements, or even merging sequences.

Lecture 24 — Dynamic Programming I

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — 15 November 2012

Today:

- Introduction to Dynamic Programming
- The subset sum problem
- The minimum edit distance problem

1 Dynamic Programming

> "An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. This, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities".
> Richard Bellman ("Eye of the Hurricane: An autobiography", World Scientific, 1984)

The Bellman-Ford shortest path algorithm we have covered is named after Richard Bellman and Lester Ford. In fact that algorithm can be viewed as a dynamic program. Although the quote is an interesting bit of history it does not tell us much about dynamic programming. But perhaps the quote will make you feel better about the fact that the term has little intuitive meaning.

In this course, as commonly used in computer science, we will use the term dynamic programming to mean an algorithmic technique in which (1) one constructs the solution of a larger problem instance

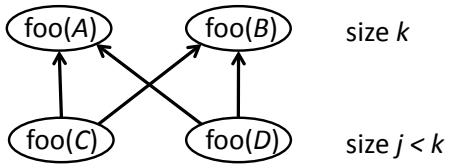
†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

by composing solutions to smaller instances, and (2) the solution to each smaller instance can be used in multiple larger instances. For example, in the Bellman-Ford algorithm, to find the shortest path length from the source to vertex v that uses at most i vertices, depends on finding the shortest path lengths to the in-neighbors of v that use at most $i-1$ vertices. As vertices may share in-neighbors, these smaller instances maybe used more then once. Dynamic programming is thus one of the inductive algorithmic techniques we are covering in this course.

Recall from Lecture 1 that in all the inductive techniques an algorithm relies on putting together smaller parts to create a larger solution. The correctness then follows by induction on problem size. The beauty of such techniques is that the proof of correctness parallels the algorithmic structure.

So far the inductive approaches we have covered are divide-and-conquer, the greedy method, and contraction. In the greedy method and contraction each instance makes use of only a single smaller instance. In the case of greedy algorithms the single instance was one smaller—e.g. Dijkstra's algorithm that removes the vertex closest to the set of vertices with known shortest paths and adds it to this set. In the case of contraction it is typically a constant fraction smaller—e.g. solving the scan problem by solving an instance of half the size, or graph connectivity by contracting the graph by a constant fraction.

In the case of divide-and-conquer, as with dynamic programming, we made use of multiple smaller instances to solve a single larger instance. However in divide-and-conquer we have always assumed the solutions are solved independently and hence we have simply added up the work of each of the recursive calls to get the total work. But what if two instances of size k, for example, both need the solution to the same instance of size $j < k$?



Although sharing the results in this simple example will only make at most a factor of two difference in work, in general sharing the results of subproblems can make an exponential difference in the work performed. The simplest, albeit not particularly useful, example is in calculating the Fibonacci numbers. As you have likely seen, one can easily write the recursive algorithm for Fibonacci:

```
1  fun fib(n) =
2    if (n ≤ 1) then 1
3    else fib(n − 1) + fib(n − 2)
```

But this recursive version will take exponential work in n. If the results from the instances are somehow shared, however, then the algorithm only requires linear work, as illustrated in Figure 1. It turns out there are many quite practical problems where sharing results of subinstances is useful and can make a significant differences in the work used to solve a problem. We will go through several of these examples.

With divide-and-conquer the composition of a problem instance in terms of smaller instances is typically described as a tree, and in particular the so called recursion tree. With dynamic programming

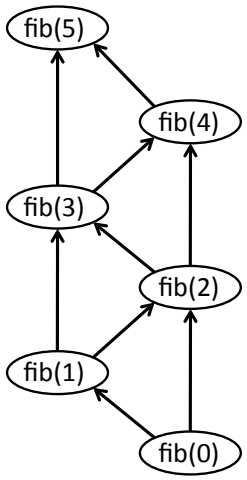


Figure 1: The DAG for calculating the Fibonacci numbers.

the composition can instead be viewed as a Directed Acyclic Graph (DAG). Each vertex in the DAG corresponds to a problem instance and each edge goes from an instance of size j to one of size $k > j$—i.e. we direct the edges (arcs) from smaller instances to the larger ones that use them. We direct them this way since the edges can be viewed as representing dependences between the source and destination (i.e. the source has to be calculated before the destination can be). The leaves of this DAG (i.e. vertices with no in-edges) are the base cases of our induction (instances that can be solved directly), and the root of the DAG (the vertex with no out-edges) is the instance we are trying to solve. More generally we might actually have multiple roots if we want to solve multiple instances.

Abstractly dynamic programming can therefore be best viewed as evaluating a DAG by propagating values from the leaves to the root and performing some calculation at each vertex based on the values of its in-neighbors. Based on this view calculating the work and span of a dynamic program is relatively straightforward. We can associate with each vertex a work and span required for that vertex. The overall work is then simply the sum of the work across the vertices. The overall span is the longest path in the DAG where the path length is the sum of the spans of the vertices along that path. For example consider the DAG shown in Figure 2. This DAG does $5 + 11 + 3 + 2 + 4 + 1 = 26$ units of work and has a span of $1+2+3+1 = 7$. Many dynamic programs have significant parallelism although some do not.

The challenging part of developing an algorithm for a problem based on dynamic programming is figuring out what DAG to use. The best way to do this, of course, is to think inductively—how can I solve an instance of a problem by composing the solutions to smaller instances? Once an inductive solution is formulated you can think about whether the solutions can be shared and how much savings can be achieved by sharing. As with all algorithmic techniques, being able to come up with solutions takes practice.

It turns out that most problems that can be tackled with dynamic programming solutions are optimization or decision problems. An *optimization problem* is one in which we are trying to find a solution that optimizes some criteria (e.g. finding a shortest path, or finding the longest contiguous subsequence sum). Sometimes we want to enumerate (list) all optimal solutions, or count the number

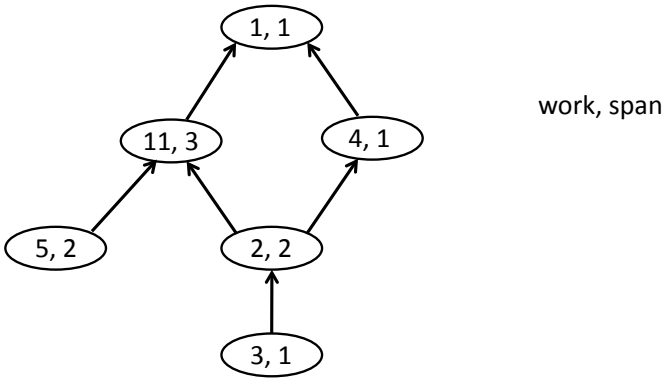


Figure 2: Work and span for a dynamic programming example. The work for each vertex is on the left and the span on the right. The total work is 26 and the span is 7.

of such solutions. A *decision problem* is one in which we are trying to find if a solution to a problem exists. Again we might want to count or enumerate the valid solutions. Therefore when you see an optimization or enumeration problem you should think about possible dynamic programming solutions.

Although dynamic programming can be viewed abstractly as a DAG, in practice we need to implement (code) the dynamic program. There are two common ways to do this, which are referred to as the top-down and bottom-up approaches. The *top-down* approach starts at the root and uses recursion, as in divide-and-conquer, but remembers solutions to subproblems so that when the algorithm needs to solve the same instance many times only the first call does the work and the remaining calls just look up the solution. Storing solutions for reuse is called *memoization*. The *bottom-up* approach starts at the leaves of the DAG and typically processes the DAG in some form of level order traversal—for example, by processing all problems of size 1 and then 2 and then 3, and so on. Each approach has its advantages and disadvantages. Using the top-down approach (recursion with memoization) can be quite elegant and can be more efficient in certain situations by evaluating only those instances actually needed. Using the bottom up approach (level order traversal of the DAG) assumes it will need every instance whether or not is it use in the overall solution, but can be easier to parallelize and can be more space efficient. *It is important, however, to remember to first formulate the problem abstractly in terms of the inductive structure, then think about it in terms of how substructure is shared in a DAG, and only then worry about coding strategies.*

1.1 Subset Sums

The first problem we will cover in this lecture is a decision problem, the subset sum problem:

Definition 1.1. The *subset sum* (SS) problem is, given a multiset[1] of positive integers S and a positive integer value k, determine if there is any $X \subseteq S$ such that $\sum_{x \in X} x = k$.

For example, consider the multiset $S = \{1, 4, 2, 9\}$. There is no subset that sums to 8, where as if the target sum is $k = 7$, the subset $\{1, 4, 2\}$ is a solution.

[1] A *multiset* is like a set, but may include duplicate elements.

In the general case when k is unconstrained this problem is a classic NP-hard problem. However, our goal here are more modest. We are going to consider the case where we include k in the work bounds. We show that as long as k is polynomial in $|S|$ then the work is also polynomial in $|S|$. Solutions of this form are often called *pseudo-polynomial* work (time) solutions.

This problem can be solved by brute force simply considering all possible subsets. This takes exponential work since there are an exponential number of subsets. For a more efficient solution, one should consider an inductive solution to the problem. As greedy algorithms tend to be efficient, you should first consider some form of greedy method that greedily takes elements from S. Unfortunately greedy does not work.

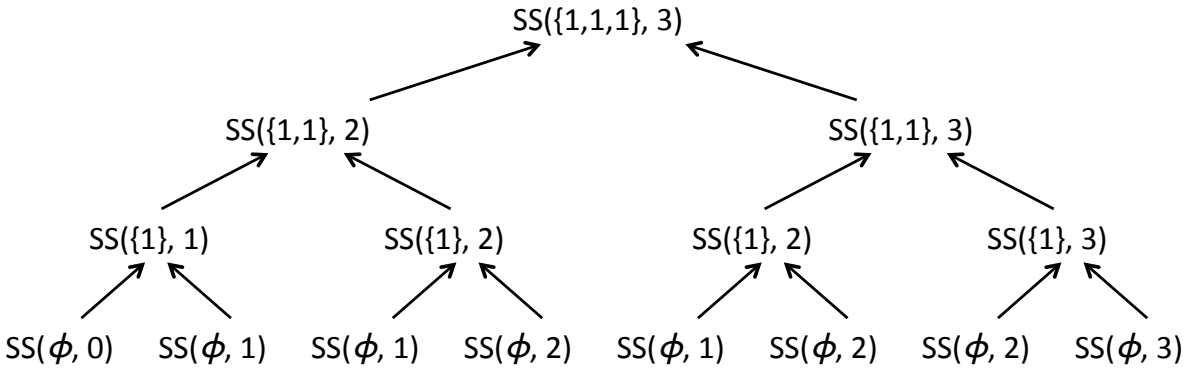
We therefore consider a divide-and-conquer solution. Naively, this will also lead to exponential work, but by reusing subproblems we can show that it results in an algorithm that requires only $O(|S|k)$ work. The idea is to consider one element a out of S (any will do) and consider the two possibilities: either a is included in X or not. For each of these two possibilities we make a recursive call on the subset $S \setminus \{a\}$, and in one case we subtract a from k ($a \in X$) and in the other case we leave k as is ($a \notin X$). Here is an algorithm based on this idea. It assumes the input is given as a list (the order of the elements of S in the list does not matter):

```
1  fun SS(S, k) =
2    case (showl(S), k) of
3      (_, 0) ⇒ true
4    | (NIL, _) ⇒ false
5    | (CONS(a, R), _) ⇒
6        if (a > k) then SS(R, k)
7        else (SS(R, k − a) orelse SS(R, k))
```

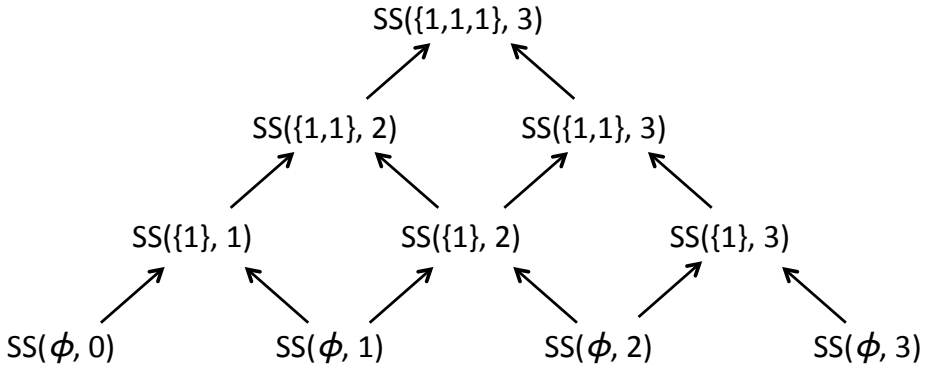
Lines 3 and 4 are the base cases. In particular if $k = 0$ then the result is true since the empty set adds to zero. If $k \neq 0$ and S is empty, then the result is false since there is no way to get k from an empty set. If S is not empty but its first element a is greater than k, then we clearly can not add a to X, and we need only make one recursive call. The last line is the main inductive case where we either include a or not. In both cases we remove a from S in the recursive call R. In the left case we are including a in the set so we have to subtract its value from k. In the right case we are not, so k remains the same.

What is the work of this recursive algorithm? Well, it leads to a binary recursion tree that might be n deep. This would imply something like 2^n work. This is not good. The key observation, however, is that there is a huge overlap in the subproblems. For example Figure **??**(a) shows the recursion tree for the problem instance $\mathtt{SS}(\{1,1,1\},3)$ As you should notice there are many common calls. In the bottom row, for example there are three calls each to $\mathtt{SS}(\emptyset,1)$ and $\mathtt{SS}(\emptyset,2)$. If we coalesce the common calls we get the following DAG where the leaves at the bottom are the base cases and the root at the top is the instance we are solving.

The question is how do we calculate the number of distinct instances of SS, which is also the number of vertices in the DAG?


SS({1,1,1}, 3)
SS({1,1}, 2)
SS({1,1}, 3)
SS({1}, 1)
SS({1}, 2)
SS({1}, 2)
SS({1}, 3)
SS(ϕ, 0)
SS(ϕ, 1)
SS(ϕ, 1)
SS(ϕ, 2)
SS(ϕ, 1)
SS(ϕ, 2)
SS(ϕ, 2)
SS(ϕ, 3)

(a)


SS({1,1,1}, 3)
SS({1,1}, 2)
SS({1,1}, 3)
SS({1}, 1)
SS({1}, 2)
SS({1}, 3)
SS(ϕ, 0)
SS(ϕ, 1)
SS(ϕ, 2)
SS(ϕ, 3)

(b)

For an initial instance $\texttt{SS}(S,k)$ there are are only $|S|$ distinct lists that are ever used (each suffix of S). Furthermore, the value of second argument in the recursive calls only decreases and never goes below 0, so it can take on at most $k+1$ values. Therefore the total number of possible instances of SS (vertices in the DAG) is $|S|(k+1) = O(k|S|)$. Each instance only does constant work to compose its recursive calls. Therefore the total work is $O(k|S|)$. Furthermore it should be clear that the longest path in the DAG is $O(|S|)$ so the total span is $O(|S|)$ and the algorithm has $O(k)$ parallelism.

Why do we say the algorithm is pseudo-polynomial? The size of the subset sum problem is defined to be the number of bits needed to represent the input. Therefore, the input size of k is $\log k$. But the work is $O(2^{\log k}|S|)$, which is exponential in the input size. That is, the complexity of the algorithm is measured with respect to the length of the input (in terms of bits) and not on the numeric value of the input. If the value of k, however, is constrained to be a polynomial in $|S|$ (i.e., $k \leq |S|^c$ for some constant c) then the work is $O(k|S|) = O(|S|^{c+1})$ on input of size $c\log|S| + |S|$, and the algorithm is polynomial in the length of the input.

At this point we have not fully specified the algorithm since we have not explained how to take advantage of the sharing—certainly the recursive code we wrote would not. We will get back to this after a couple more examples. Again we want to emphasize that the first two orders of business are to figure out the inductive structure and figure out what instances can be shared.

1.2 Minimum Edit Distance

The second problem we consider is a optimization problem, the minimum edit distance problem.

Definition 1.2. The minimum edit distance (MED) problem is, given a character set Σ and two sequences of characters $S = \Sigma^*$ and $T = \Sigma^*$, determine the minimum number of insertions and deletions of single characters required to transform S to T.

For example if we started with the sequence

$$S = \langle A, B, C, A, D, A \rangle$$

we could convert it to

$$T = \langle A, B, A, D, C \rangle$$

with 3 edits (delete the C, delete the last A, and insert a C). This is the best that can be done so the fewest edits needed is 3.

The MED problem is an important problem that has many applications. For example in version control systems such as `git` or `svn` when you update a file and commit it, the system does not store the new version but instead only stores the "differences" from the previous version[2]. Storing the differences can be quite space efficient since often the user is only making small changes and it would be wasteful to store the whole file. Variants of the minimum edit distance problem are use to find this difference. Edit distance can also be used to reduce communication costs by only communicating the differences from a previous version. It turns out that edit-distance is also closely related to approximate matching of genome sequences.

[2] Alternatively it might store the new version, but use the differences to encode the old version.

The first observation is that inserting a character into S is equivalent to deleting a character in T. The solution in the above example could be stated as delete C in S, delete the last A in S, delete C in T. Using this formulation, a brute force solution would be to compute all subsequences of S and for any of them that are a subsequences of T return the longest one, clearly an exponential solution. This formulation of the problem is often known as the longest common subsequence problem.

Another possibility would be to consider a greedy method that scans the sequences finding the first difference, fixing it and then moving on. Unfortunately no simple greedy method is known to work. The problem is that there can be multiple ways to fix the error—we can either delete the offending character, or insert a new one. In the example above when we get to the C in S we could either delete C or insert an A. If we greedily pick the wrong way to fix it, we might not end up with an optimal solution. (Recall that in greedy algorithms, once you make a choice, you cannot go back and try an alternative.) Again in the example, if you inserted an A, then more than two more edits will be required.

The first key step in dynamic programming is to recognize the inductive structure of the problem. This step requires precisely defining the subproblems that we will consider. The goal is to keep the number of subproblems small. For example, one possibility would be to find the fewest edits needed between any contiguous subsequences of S and T. Since the number of contiguous subsequences of a sequence of length n is $\binom{n+1}{2}$, there are $O(|S|^2|T|^2)$ possible pairs of subsequences to consider. Although this number is much less the exponential, it is still large. A better choice is to consider all suffixes (or prefixes) of S and T. Now there are at most $O(|S||T|)$ pairs of suffixes to consider.

Next, how do we find the $\texttt{MED}(S,T)$ in terms of the smaller problems? The greedy solution gives a good hint how. Suppose $S = s :: S'$ and $T = t :: T'$. If the first characters of S and T match, then no insertion or deleted is needed, and the we only need to consider edits to the suffixes, S' and T'. But what if the first two characters do not match? In particular when we get to the C in our example there were exactly two possible ways to fix it—deleting C or inserting A. As with the subset sum problem, why not consider both ways. This leads to the following algorithm.

```
1  fun MED(S, T) =
2    case (showl(S), showl(T)) of
3      (_, NIL) ⇒ |S|
4    | (NIL, _) ⇒ |T|
5    | (CONS(s, S'), CONS(t, T')) ⇒
6        if (s = t) then MED(S', T')
7        else 1 + min(MED(S, T'), MED(S', T))
```

In the first base case where T is empty we need to delete all of S to generate an empty string requiring $|S|$ insertions. In the second base case where S is empty we need to insert all of T, requiring $|T|$ insertions. If neither is empty we compare the first character. If they are equal we can just skip them and make a recursive call on the rest of the sequences. If they are different then we need to consider the two cases. The first case ($\texttt{MED}(S,T')$) corresponds to inserting the value t. We pay one edit for the insertion and then need to match up S (which all remains) with the tail of T (we have already matched up the head t with the character we inserted). The second case ($\texttt{MED}(S',T)$) corresponds to deleting the value s. We pay one edit for the deletion and then need to match up the tail of S (the head has been deleted) with all of T.

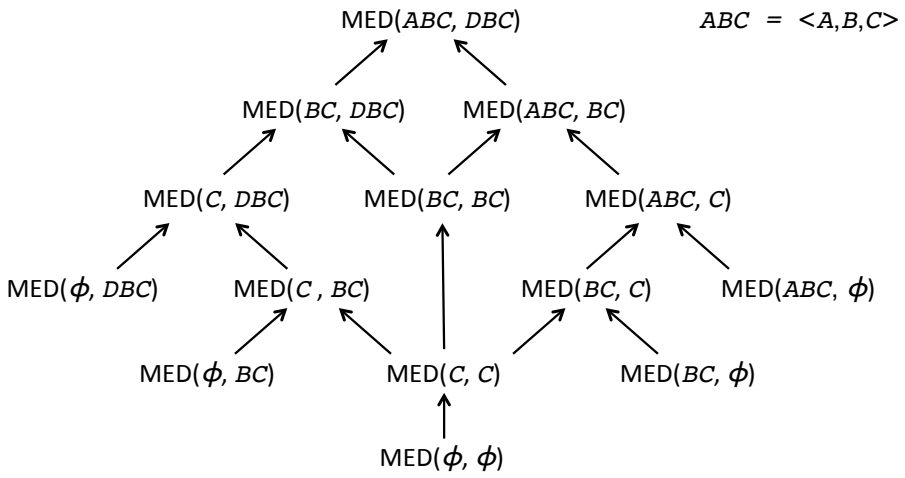


Figure 3: The dynamic programming DAG for an instance of the minimum edit distance (MED) problem.

If we ran the code recursively we would end up with an algorithm that takes exponential work. In particular the recursion tree is binary and has a depth that is linear in the size of S and T. However, as with subset sums, there is significant sharing going on. Again we view the computation as a DAG in which each vertex corresponds to call to `MED` with distinct arguments. An edge is placed from u to v if the call v uses u. For Figure 3 shows an example of the DAG for $\texttt{MED}(\langle A, B, C\rangle, \langle D, B, C\rangle)$.

We can now place an upper bound on the number of vertices in our DAG by bounding the number of distinct arguments. There can be at most $|S|+1$ possible values of the first argument since in recursive calls we only use suffixes of the original S and there are only $|S|+1$ such suffixes (including the empty and complete suffixes). Similarly there can be at most $|T|+1$ possible values for the second argument. Therefore the total number of possible distinct arguments to `MED` on original strings S and T is $(|T|+1)(|S|+1) = O(|S||T|)$. Furthermore the depth of the DAG (longest path) is $O(|S|+|T|)$ since each recursive call either removes an element from S or T so after $|S|+|T|$ calls there cannot be any element left. Finally we note that assuming we have constant work operations for removing the head of a sequence (e.g. using a list) then each vertex of the DAG takes constant work and span.

All together this gives us

$$W(\texttt{MED}(S,T)) = O(|S||T|)$$

and

$$S(\texttt{MED}(S,T)) = O(|S|+|T|).$$

1.3 Problems with Efficient Dynamic Programming Solutions

There are many problems with efficient dynamic programming solutions. Here we list just some of them to give a sense of what these problems are.

1. Fibonacci numbers

2. Using only addition compute (n choose k) in $O(nk)$ work
3. Edit distance between two strings
4. Edit distance between multiple strings
5. Longest common subsequence
6. Maximum weight common subsequence
7. Can two strings S1 and S2 be interleaved into S3
8. Longest palindrome
9. longest increasing subsequence
10. Sequence alignment for genome or protein sequences
11. subset sum
12. knapsack problem (with and without repetitions)
13. weighted interval scheduling
14. line breaking in paragraphs
15. break text into words when all the spaces have been removed
16. chain matrix product
17. maximum value for parenthesizing x1/x2/x3.../xn for positive rational numbers
18. cutting a string at given locations to minimize cost (costs n to make cut)
19. all shortest paths
20. find maximum independent set in trees
21. smallest vertex cover on a tree
22. optimal BST
23. probability of generating exactly k heads with n biased coin tosses
24. triangulate a convex polygon while minimizing the length of the added edges
25. cutting squares of given sizes out of a grid
26. change making
27. box stacking
28. segmented least squares problem
29. counting boolean parenthesization – true, false, or, and, xor, count how many parenthesization return true
30. balanced partition – given a set of integers up to k, determine most balanced two way partition
31. Largest common subtree

Lecture 25 — Dynamic Programming II

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — 20 November 2012

Today:

- Finish minimum edit distance (MED) from last class
- Coding Dynamic Programming
- Optimal Binary Search Trees
- Have a good thankgiving break!!

1 Coding Dynamic Programs

So far we have assumed some sort of magic recognized shared subproblems in our recursive codes and avoided recomputation. As mentioned in the last lecture, there are basically two techniques to code dynamic programming techniques: the top-down approach and the bottom-up approach.

1.1 Top-Down Dynamic Programming

The top-down approach is based on running the recursive code as is, generating implicitly the recursion structure from the root of the DAG down to the leaves. Each time a solution to a smaller instance is found for the first time it generates a mapping from the input argument to its solution. This way when we come across the same argument a second time we can just look up the solution. This process is called *memoization*, and the table used to map the arguments to solutions is called a *memo table*.

The tricky part of memoization is checking for equality of arguments since the arguments might not be simple values such as integers. Indeed in our examples so far the arguments have all involved sequences. We could compare the whole sequence element by element, but that would be too expensive. You might think that we can do it by comparing "pointers" to the values. But this does not work since the sequences can be created separately, and even though the values are equal, there could be two copies in different locations. Comparing pointers would say they are not equal and we would fail to recognize that we have already solved this instance. There is actually a very elegant solution that fixes this issue called *hash consing*. Hash consing guarantees that there is a unique copy of each value by always hashing the contents when allocating a memory location and checking if such a value already exists. This allows equality of values to be done in constant work by . The approach only works in purely functional languages and needs to be implemented as part of the language runtime system (as part of memory allocation). Unfortunately no language does hash consing automatically, so we are left to our own devices.

The most common way to quickly test for equality of arguments is to use a simple type, such as an integer, as a surrogate to represent the input values. The property of these surrogates is that there

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

needs to be a 1-to-1 correspondence between the surrogates and the argument values—therefore if the surrogates match, the arguments match. The user is responsible for guaranteeing this 1-to-1 correspondence.

Consider how we might use memoization in the dynamic program we described for minimum edit distance (MED). You have probably covered memoization before, but you most likely did it with side effects. Here we will do it in a purely functional way, which requires that we "thread" the table that maps arguments to results through the computation. Although this threading requires a few extra characters of code, it is safer for parallelism.

Recall that `MED` takes two sequences and on each recursive call, it uses suffixes of the two original sequences. To create integer surrogates we can simply use the length of each suffix. There is clearly a 1-to-1 mapping from these integers to the suffixes. `MED` can work from either end of the string so instead of working front to back and using suffix lengths, it can work back to front and use prefix lengths—we make this switch since it simplifies the indexing. This leads to the following variant of our `MED` code.

```
1  fun MED(S, T) = let
2    fun MED'(i, 0) = i
3      | MED'(0, j) = j
4      | MED'(i, j) =case (S_i = T_j) of
5                      true ⇒ MED'(i − 1, j − 1)
6                    | false ⇒ 1 + min(MED'(i, j − 1), MED'(i − 1, j))
7  in
8    MED'(|S|, |T|)
9  end
```

You should compare this with the purely recursive code for `MED`. The only real difference is replacing S and T with i and j in the definition of `MED'`. The i and j are therefore the surrogates for S and T respectively. They represent the sequences $S\langle 0,\ldots,i-1\rangle$ and $T\langle 0,\ldots,j-1\rangle$ where S and T are the original input strings.

So far we have not added a memo table, but we can now efficiently store our solutions in a memo table based on the pair of indices (i, j). Each pair represents a unique input. In fact since the arguments range from 0 to the length of the sequence we can actually use a two dimensional array (or array of arrays) to store the solutions.

To implement the memoization we define a memoization function:

```
1  fun memo f (M, a) =
2    case find(M, a) of
3      SOME(v) ⇒ v
4    | NONE ⇒ let
5        val (M', v) = f(M, a)
6      in
7        (update(M', a, v), v)
8      end
```

```
1  fun MED(S, T) = let
2    fun MED'(M, (i, 0)) = (M, i)
3      | MED'(M, (0, j)) = (M, j)
4      | MED'(M, (i, j)) =case (S_i = T_j) of
5                           true ⇒ MED''(M, (i − 1, j − 1))
6                         | false ⇒ let
7                             val (M', v_1) = MED''(M, (i, j − 1))
8                             val (M'', v_2) = MED''(M', (i − 1, j))
9                           in (M'', 1 + min(v_1, v_2)) end
10   and MED''(M, (i, j)) = memo MED' (M, (i, j))
11 in
12   MED'({}, (|S|, |T|))
13 end
```

Figure 1: The memoized version of Minimum Edit Distance (MED).

In this function f is the function that is being memoized, M is the memo table, and a is the argument to f. This function simply looks up the value a in the memo table. If it exists, then it returns the corresponding result. Otherwise it evaluates the function on the argument, and as well as returning the result it stores it in the memo. We can now write `MED` using memoization as shown in Figure 1.

Note that the memo table M is threaded throughout the computation. In particular every call to `MED` not only takes a memo table as an argument, it also returns a memo table as a result (possibly updated). Because of this passing, the code is purely functional. The problem with the top-down approach as described, however, is that it is inherently sequential. By threading the memo state we force a total ordering on all calls to `MED`. It is easy to create a version that uses side effects, as you did in 15-150 or as is typically done in imperative languages. In this case calls to `MED` can be made in parallel. However, then one has to be very careful since there can be race conditions (concurrent threads modifying the same cells). Furthermore if two concurrent threads make a call on `MED` with the same arguments, they can and will often both end up doing the same work. There are ways around this issue which are also fully safe—i.e., from the users point of view all calls look completely functional—but they are beyond the scope of this course.

1.2 Bottom-Up Dynamic Programming

We will now consider the alternate technique for implementing dynamic programs. Instead of simulating the recursive structure, which starts at the root of the DAG, it starts at the leaves of the DAG and fills in the results in some order that is consistent with the DAG–i.e. for all edges (u, v) it always calculates the value at a vertex u before working on v. Because of this careful scheduling, all values will be already calculated when they are needed.

The simplest way to implement bottom-up dynamic programming is to do some form of systematic traversal of a DAG. It is therefore useful to understand the structure of the DAG. For example, consider the structure of the DAG for minimum edit distance. In particular let's consider the two strings $S =$ `tcat` and $T =$ `atc`. We can draw the DAG as follows where all the edges go down and to the right:

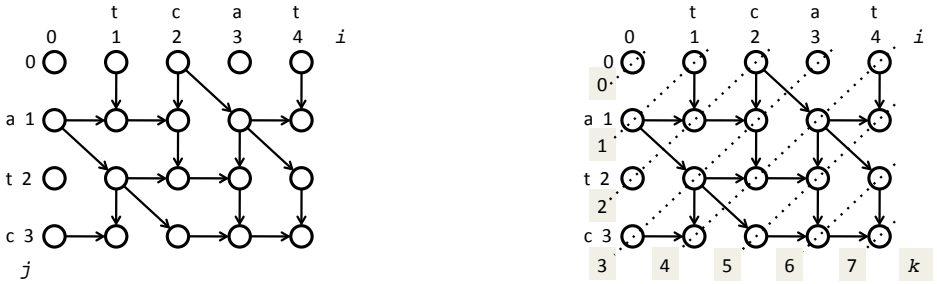


Figure 2: The DAG for `MED` on the strings "tcat" and "atc" (a) and processing it by diagonals (b).

The numbers represent the i and the j for that position in the string. We draw the DAG with the root at the bottom right, so that the vertices are structured the same way we might fill an array indexed by i and j. We Consider $\texttt{MED}(4,3)$. The characters S_4 and T_3 are not equal so the recursive calls are to $\texttt{MED}(3,3)$ and $\texttt{MED}(4,2)$. This corresponds to the vertex to the left and the one above. Now if we consider $\texttt{MED}(4,2)$ the characters S_4 and T_2 are equal so the recursive call is to $\texttt{MED}(3,1)$. This corresponds to the vertex diagonally above and to the left. In fact whenever the characters S_i and T_j are not equal we have edges from directly above and directly to the left, and whenever they are equal we have an edge from the diagonal to the left and above. This tells us quite a bit about the DAG. In particular it tells us that it is safe to process the vertices by first traversing the first row from left to right, and then the second row, and so on. It is also safe to traverse the first column from top to bottom and then the second column and so on. In fact it is safe to process the diagonals in the / direction from top left moving to the bottom right. In this case each diagonal can be processed in parallel.

In general when applying $\texttt{MED}(S,T)$ we can use an $|T| \times |S|$ array to store all the partial results. We can then fill the array either by row, column, or diagonal. Using diagonals can be coded as shown in Figure 3.

The code uses a table M to store the array entries. In practice an array might do better. Each round of `diagonals` processes one diagonal and updates the table M, starting at the leaves at top left. The figure below shows these diagonals indexed by k on the left side and at the bottom. We note that the index calculations are a bit tricky (hopefully we got them right). Notice that the size of the diagonals grows and then shrinks.

2 Optimal Binary Search Trees

We have talked about using BSTs for storing an ordered set or table. The cost of finding an key is proportional to the depth of the key in the tree. In a fully balanced BST of size n the average depth of each key is about $\log n$. Now suppose you have a dictionary where you know probability (or frequency) that each key will be accessed—perhaps the word "of" is accessed much more often than "epistemology". The goal is find a static BST with the lowest overall access cost. That is, make a BST so that the more likely keys are closer to the root and hence the average access cost is reduced. This line of reasoning leads to the following problem:

Definition 2.1. The *optimal binary search tree* (OBST) problem is given an ordered set of keys S and

```
1  fun MED(S, T) = let
2    fun MED′(M, (i, 0)) = i
3      | MED′(M, (0, j)) = j
4      | MED′(M, (i, j)) =case (S_i = T_j) of
5                           true ⇒ M_{i−1,j−1}
6                         | false ⇒ 1 + min(M_{i,j−1}, M_{i−1,j})
7    fun diagonals(M, k) =
8      if (k > |S| + |T|) then M
9      else let
10       val s = max(0, k − |T|)
11       val e = min(k, |S|)
12       val M′ = M ∪ {(i, k − i) ↦ MED(M, (i, k − i)) : i ∈ {s, ..., e}}
13     in
14       diagonals(M′, k + 1)
15     end
16 in
17   diagonals({}, 0)
18 end
```

Figure 3: The dynamic program for MED based on the bottom-up approach using diagonals.

a probability function $p : S \to [0 : 1]$:

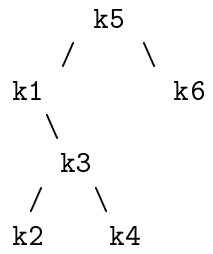
$$\min_{T \in \texttt{Trees}(S)} \left(\sum_{s \in S} d(s, T) \cdot p(s) \right)$$

where $\texttt{Trees}(S)$ is the set of all BSTs on S, and $d(s, T)$ is the depth of the key s in the tree T (the root has depth 1).

For example we might have the following keys and associated probabilities

key	k_1	k_2	k_3	k_4	k_5	k_6
p(key)	1/8	1/32	1/16	1/32	1/4	1/2

Then the tree



has cost 31/16, which is optimal. Creating a tree with these two solutions as the left and right children of S_i, respectively, leads to the optimal solution given S_i as a root.

Exercise 1. *Find another tree with equal cost.*

The brute force solution would be to generate every possible binary search tree, compute their cost, and pick the one with the lowest costs. But the number of such trees is $O(4^n)$ which is prohibitive.

Exercise 2. *Write a recurrence for the total number of distinct binary search trees with n keys.*

Since we are considering binary search trees, one of the keys must be the root of the optimal tree. Suppose S_r is that root. An important observation is that both of its subtrees must be optimal, which is a common property of optimization problems: The optimal solution to a problem contains optimal solutions to subproblems. This *optimal substructure* property is often a clue that either a greedy or dynamic programming algorithm might apply.

Which key should be the root of the optimal tree? A greedy approach might be to pick the key k with highest probability and put it at the root and then recurse on the two sets less and greater than k. You should convince yourself that this does not work. Since we cannot know in advance which key should be the root, let's try all of them, recursively finding their optimal subtrees, and then pick the best of the $|S|$ possibilities.

With this recursive approach, how should we define the subproblems? Let S be all the keys placed in sorted order. Now any subtree of a BST on S must contain the keys of a contiguous subsequence of S. We can therefore define subproblems in terms of a contiguous subsequence of S. We will use $S_{i,j}$ to indicate the subsequence starting at i and going to j (inclusive of both). Not surprisingly we will use the pair (i, j) to be the surrogate for $S_{i,j}$.

Now Let's consider how to calculate the cost give the solution to two subproblems. For subproblem $S_{i,j}$, assume we pick key S_r $(i \leq r \leq j)$ as a the root. We can now solve the OSBT problem on the prefix $S_{i,r-1}$ and suffix $S_{r+1,i}$. We therefore might consider adding these two solutions and the cost of the root $(p(S_r))$ to get the cost of this solution. This, however, is wrong. The problem is that by placing the solutions to the prefix and suffix as children of S_r we have increased the depth of each of their keys by 1. Let T be the tree on the keys $S_{i,j}$ with root S_r, and T_L, T_R be its left and right subtrees. We therefore have:

$$\begin{aligned}
\texttt{Cost}(T) &= \sum_{s \in T} d(s,T) \cdot p(s) \\
&= p(S_r) + \sum_{s \in T_L} (d(s,T_L)+1) \cdot p(s) + \sum_{s \in T_R} (d(s,T_R)+1) \cdot p(s) \\
&= \sum_{s \in T} p(s) + \sum_{s \in T_L} d(s,T_L) \cdot p(s) + \sum_{s \in T_R} d(s,T_R) \cdot p(s) \\
&= \sum_{s \in T} p(s) + \texttt{Cost}(T_L) + \texttt{Cost}(T_R)
\end{aligned}$$

That is, the cost of a subtree T the probability of accessing the root (i.e., the total probability of accessing the keys in the subtree) plus the cost of searching its left subtree and the cost of searching its right subtree. When we add the base case this leads to the following recursive definition:

```
1  fun OBST(S) =
2    if |S| = 0 then 0
3    else ∑_{s∈S} p(s) + min_{i∈⟨1...|S|⟩} (OBST(S_{1,i−1}) + OBST(S_{i+1,|S|}))
```

Exercise 3. *How would you return the optimal tree in addition to the cost of the tree?*

As in the examples of subset sum and minimum edit distance, if we execute the recursive program directly `OBST` it will require exponential work. Again, however, we can take advantage of sharing among the calls to `OBST`. To bound the number of vertices in the corresponding DAG we need to count the number of possible arguments to `OBST`. Note that every argument is a contiguous subsequence from the original sequence S. A sequence of length n has only $n(n+1)/2$ contiguous subsequences since there are n possible ending positions and for the i^{th} end position there are i possible starting positions ($\sum_{i=1}^{n} i = n(n+1)/2$). Therefore the number of possible arguments is at most $O(n^2)$. Furthermore the longest path of vertices in the DAG is at most $O(n)$ since recursion can at most go n levels (each level removes at least one key).

Unlike our previous examples, however, the cost of each vertex in the DAG (each recursive in our code not including the subcalls) is no longer constant. The subsequence computations $S_{i,j}$ can be done in $O(1)$ work each (think about how) but there are $O(|S|)$ of them. Similarly the sum of the $p(s)$ will take $O(|S|)$ work. To determine the span of a vertex we note that the min and sum can be done with a reduce in $O(\log|S|)$ span. Therefore the work of a vertex is $O(|S|) = O(n)$ and the span is $O(\log n)$. Now we simply multiply the number of vertices by the work of each to get the total work, and the longest path of vertices by the span of each vertex to get the span. This give $O(n^3)$ work and $O(n \log n)$ span.

This example of the optimal BST is one of several applications of dynamic programming which effectively based on trying all binary trees and determining an optimal tree given some cost criteria. Another such problem is the matrix chain product problem. In this problem one is given a chain of matrices to be multiplied ($A_1 \times A_2 \times \cdots A_n$) and wants to determine the cheapest order to execute the multiplies. For example given the sequence of matrices $A \times B \times C$ it can either be ordered as $(A \times B) \times C$ or as $A \times (B \times C)$. If the matrices have sizes $2 \times 10, 10 \times 2$, and 2×10, respectively, it is much cheaper to calculate $(A \times B) \times C$ than $a \times (B \times C)$. Since $\times$ is a binary operation any way to evalue our product corresponds to a tree, and hence our goal is to pick the optimal tree. The matrix chain product problem can threrefore be solved in a very similar structure as the OBST algorithm and with the same cost bounds.

2.1 Coding optimal BST

As with the MED problem we first replace the sequences in the arguments with integers. In particular we describe any subsequence of the original sorted sequence of keys S to be put in the BST by its offset from the start (i, 1-based) and its length l. We then get the following recursive routine.

```
1  fun OBST(S) = let
2    fun OBST'(i, l) =
3      if l = 0 then 0
4      else ∑_{k=0}^{l-1} p(S_{i+k}) + min_{k=0}^{l-1} (OBST'(i, k) + OBST'(i + k + 1, l − k − 1))
5  in
6    OBST(1, |S|)
7  end
```

This modified version can now more easily be used for either the top-down solution using memoization or the bottom-up solution. In the bottom-up solution we note that we can build a table with the columns corresponding to the i and the rows corresponding to the l. Each of them range from 1 to n ($n = |S|$). It would as follows:

```
 1 2 ... n
1         /
2       /
.     /
.   /
n /
```

The table is triangular since as l increases the number of subsequences of that length decreases. This table can be filled up row by row since every row only depends on elements in rows above it. Each row can be done in parallel.

Lecture 26 — Hash Tables

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — 27 November 2012

Today:

- Hashing
- Hash Tables

1 Hashing

hash: transitive verb[1]

1. (a) to chop (as meat and potatoes) into small pieces
   (b) confuse, muddle
2. ...

This is the definition of hash from which the computer term was derived. The idea of hashing as originally conceived was to take values and to chop and mix them to the point that the original values are muddled. The term as used in computer science dates back to the early 1950's.

More formally the idea of hashing is to approximate a random function $h : \alpha \to \beta$ from a source (or universe) set U of type α to a destination set of type β. Most often the source set is significantly larger than the destination set, so the function not only chops and mixes but also reduces. In fact the source set might have infinite size, such as all character strings, while the destination set always has finite size. Also the source set might consist of complicated elements, such as the set of all directed graphs, while the destination are typically the integers in some fixed range. Hash functions are therefore many to one functions.

Using an actual randomly selected function from the set of all functions from α to β is typically not practical due to the number of such functions and hence the size (number of bits) needed to represent such a function. Therefore in practice one uses some pseudorandom function.

Exercise 1. *How many hash functions are there that map from a source set of size n to the integers from 1 to m? How many bits does it take to represent them? What if the source set consists of character strings up length up to 20. Assume there are 100 possible characters.*

Why is it useful to have random or pseudo random functions that map from some large set to a smaller set? Generally such functions might be used to hide data, to reorder data in a random order, or to spread data smoothly over a range. Here we consider some applications of each.

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

[1] Merriam Websters

1. We saw how hashing can be used in treaps. In particular we suggested using a hash function to hash the keys to generate the "random" priorities. Here what was important is that the ordering of the priorities is somehow random with respect to the keys. Our analysis assumed the priorities were truly random, but it can be shown that a limited form of randomness that arise out of relatively simple hash functions is sufficient.

2. In cryptography hash functions can be used to hide information. One such applications is in digital signatures where a so-called secure hash is used to describe a large document with a small number of bits.

3. Hashing can be used to approximately match documents, or even parts of documents.

4. Hashing is used to implement hash tables. In hash tables one is given a set of keys $K \subset \alpha$ and needs to map them to a range of integers so they can stored at those locations in an array. The goal is to spread the keys out across the integers as well as possible to minimize the number of keys that collide in the array. You should not confuse the terms hash function and hash table. They are two separate ideas, the later that uses the prior.

There is a deep and interesting theory of hash functions. Depending on the application, the needs of the hash function are very different. We will not cover the theory here but you will likely see it in more advanced algorithms classes.

So what are some reasonable hash functions. Here we consider some simple ones. For hashing integers we can use

$$h(x) = (ax + b) \bmod p$$

where $a \in [1, \ldots, p-1]$, $b \in [0, \ldots, p-1]$, and p is a prime. This is called a linear congruential hash function has some nice properties that you will likely learn about in 15-451.

For strings we can simply use a polynomial

$$h(S) = \left(\sum_{i=1}^{|S|} = s_i a^i \right) \bmod p$$

In our analysis we will assume that we have hash functions with the following idealized property *simple uniform hashing*: The hash function uniformly distributes the n keys over the range $[0 \ldots, m-1]$ and the hash value for any key is independent of the hash value for any other key.

2 Hash Tables

Hash tables are used when an application needs to maintain a dynamic set where the main operations are `insert`, `find` and `delete`. Hash table can implement the abstract data types `Set` and `Table`. Unlike binary search trees, which require the universe of keys has a total ordering, hash tables do not. A total ordering enables the additional ordering operations provided by the Ordered Set abstract data type.

The main issue with hash table is collisions, where two keys hash to the same location. Is it possible to avoid collisions? Not if we don't know the set of keys in advance. Since the size of the table T is much smaller than the universe of keys U, $|T| << |U|$, there must exist at least two keys that map to the same index, by the *Pigeonhole Principle*: If you put more than m items into m bins, then at least one bin contains more than one item. For a particular hash function, the subset of keys $K \subset U$ that we want to hash may or may not cause a collision even when the number of keys is much smaller than the size of the table. Therefore, for general purpose dynamic hash tables we have to assume that collisions will occur.

How likely is there at least one collision? This is the same question as the birthday paradox: When there a n or more people in a room, what is the chance that two people have the same birthday. It turns out that for a table of size 365 you need only 23 keys for a 50% chance of a collision, and as little as 60 keys for a 99% chance. More interesting, when hashing to m locations, you can expect a collision after only $\sqrt{\frac{1}{2}\pi\, m}$ insertions, and can expect every location in the table has an element after $\Theta(m \log m)$ insertions. The former is related to the *birthday paradox*, whereas the later is relate to the *coupon collector* problem.

There are several well-studied collision resolution strategies:

- **Separate chaining**: Store elements not in a table, but in linked lists (containers,bins) hanging off the table.
- **Open addressing**: Put everything into the table, but not necessarily into cell $h(k)$.
- **The perfect hash**: When you know the keys in advance, construct hash functions that avoids collisions entirely.
- **Multiple-choice hashing and Cuckoo hashing**: Consider exactly two locations $h_1(k)$ and $h_2(k)$ only.

We will consider the first two in this lecture, and save the others for another lecture. In our discussion we will assume we have a set of n keys K that we want to store and a hash function $h : \texttt{key} \to [1, \ldots, m]$ for some m.

2.1 Separate Chaining

In 15-122 you covered hash tables using separate chaining. As you may recall, the idea is to maintain an array of linked lists. All keys that hashes to the same location in the sequence are maintained in a linked list. When inserting a key k, it inserts it at the beginning of the linked list at location $h(k)$. But if the application may attempt to insert the same key multiple times, separate chaining needs to search down the list to check whether the key is already in the list, in which case it might just as well add the key to the end of the linked list. To find a key, simply search for key in the linked list at location $h(k)$. To delete a key, remove it from the linked list.

The costs of these operations is related to the average length of a chain, which is n/m when there n keys in a table with m chains. We call $\lambda = n/m$ the *load factor* of the table.

We consider two cases when that can occur when applying these operations: an unsuccessful search when the key is not in the table, and a successful search when the key is in the table. We assume that we can compute the $h(k)$ in $O(1)$ work.

Claim 2.1. *Assuming simple uniform hashing, an* unsuccessful search *takes expected* $\Theta(1+\lambda)$ *work.*

Proof. The average length of a list is λ and an unsuccessful search needs to search the whole list. Including the cost of computing $h(k)$, the total work is $\Theta(1+\lambda)$. □

Claim 2.2. *Assuming simple uniform hashing, a successful* successful search *takes expected* $\Theta(1+\lambda)$ *work.*

Proof. To simplify the analysis, we assume that keys are added at the end of the chain[2]. The the cost of a successful search is the same as an unsuccessful search at the time the key was added to the table. When the table has i keys, an unsuccessful search is expected to be $(1+i/m)$. Averaging over all keys we get

$$\frac{1}{n}\sum_{i=0}^{n-1}(1+i/m) = 1+(n-1)/2m = 1+\lambda/2-\lambda/2m = \Theta(1+\lambda)$$

□

That is, successful searches examine half the list on average and unsuccessful searches full list. If $n = O(m)$ then with simple uniform hashing, all operations have expected $O(1)$ work and span. Even more important, some chains can be long, $O(n)$ in the worst case, but it is extremely unlikely they will more than double λ. The advantage of separate chaining is that it is not particularly sensitive to the size of the table. If the number keys is more than anticipated, the cost of search becomes only somewhat worse. If the number is less, then only a small amount of space in the table is wasted and cost of search is faster.

2.2 Open Address Hash Tables

The next technique does not need any linked lists but instead stores every key directly in the array. Open address hashing using so called linear probing has an important practical advantage over separate chaining: it causes fewer cache misses since typically all locations that are checked are on the same cache line.

The basic idea of open addressing is to maintain an array that is some constant factor larger than the number of keys and to store all keys directly in this array. Every cell in the array is either empty or contains a key.

To decide to which cells to assign a key, open addressing uses an ordered sequence of locations in which the key can be stored. In particular let's assume we have a function $h(k,i)$ that returns the i^{th} location for key k. We refer to the sequence $\langle h(k,0), h(k,1), h(k,2), \ldots\rangle$ as the *probe* sequence. We will get back to how the probe sequence might be assigned, but let's first go through how these sequences are used. When inserting a key the basic idea is to try each of the locations in order until it finds a cell that is empty, and then insert the key at that location. Sequentially insert would look like

[2]The average successful search time is the same whether new keys are added to the front of the end of the chain

```
1  fun insert(T, k) =
2  let
3    fun insert'(T, k, i) =
4      case nth T h(k, i) of
5        NONE ⇒ update(h(k, i), k) T
6      | _ ⇒ insert'(T, k, i + 1)
7  in
8    insert'(T, k, 1)
9  end
```

For example suppose the hash table has the following keys:

	0	1	2	3	4	5	6	7
T =		B			E	A		F

Now if for a key D we had the probe sequence $\langle 1, 5, 3, \cdots \rangle$, then we would find location 1 and 5 full (with B and E) and place D in location 3 giving:

	0	1	2	3	4	5	6	7
T =		B		D	E	A		F

Note that, in order for the update operation to be constant work and span, T must be a single threaded array. Also, the code will loop forever if all locations are full. Such an infinite loop can be prevented by ensuring that $h(k, i)$ tries every location as i is incremented, and checking when the table is full. Also, as given the code will insert the same key multiple times over. This problem is easily corrected by checking if the key is in the table and if so returning immediately.

To search we have the following code:

```
1  fun find(T, k) =
2  let
3    fun find'(T, k, i) =
4      case T[h(k, i)] of
5        NONE ⇒ false
6      | SOME(k') ⇒ if (eq(k, k')) then true
7                   else find'(T, k, i + 1)
8  in
9    find'(T, k, 1)
10 end
```

For example in the table above, if key E has the probe sequence $\langle 7, 4, 2, \cdots \rangle$, `find` would first search location 7, which is full, and then location 4 to find E.

What if we want to delete a key? Let's say we deleted A from the table above, and then searched for D. Will `find` locate it? No, it will stop looking once it finds the empty cell where A was. One solution might be to rehash everything after a delete. But that would be an extremely expensive operation for every delete. An alternative is to use what is called a *lazy delete*. Instead of deleting the key, simply replace the key with a special `HOLD` value. That is, introduce an entry data type:

```
1  datatype α entry = EMPTY | HOLD | FULL of α
```

Of course, we will need to change `find` and `insert` to handle `HOLD` entries. For `find`, simply skip over a `HOLD` entry and move to the next probe. Whereas, `insert`(v) can replace the first `HOLD` entry with `FULL`(v). But if `insert` needs to check for duplicate keys, it first needs to search for the key. If it finds the key it overwrites it with the new value. Otherwise it continues until it finds an empty cell, at which point it can replace the first `HOLD` in the probe sequence.

The main concern with lazy deletes is that they effectively increases the load factor, increasing the cost of the hash table operations. If the load factor becomes large and performance degrades, the solution is to rehash everything to a new larger table. The table should be a constant fraction larger each time the table grow so as to maintain amortized constant costs.

Now let's consider some possible probe sequences we can use. Ideally, we would like a key to use any of the possible $m!$ probe sequences with equal probability. This ideal is called *uniform hashing* But uniform hashing is not practical. Common probe sequences, which we will consider next, are

- linear probing
- quadratic probing
- double hashing

2.2.1 Linear Probing

In linear probing, to insert a key k, it first checks $h(k)$ and then checks each following location in succession, wrapping around as necessary, until it finds an empty location. That is, the i^{th} probe is

$$h(k,i) = (h(k) + i) \mod m.$$

Each position determines a single probe sequence, so there are only m possible probe sequences.

The problem with linear probing is that keys tend to cluster. It suffers from *primary clustering*: Any key that hashes to any position in a cluster (not just collisions), must probe beyond the cluster and adds to the cluster size. Worse yet, primary clustering not only makes the probe sequence longer, it also makes it more likely that it will be lengthen further.

What is the impact of cluster for unsuccessful search? Let's consider two extreme examples when the table is half full, $\lambda = 1/2$. Clustering is minimized when every other location in the table is empty. In this case, the average number of probes needed to insert a new key k is $3/2$: One probe to check cell $h(k)$, and with probability $1/2$ that cell is full and it needs to look at the next location which, by construction, must be empty. In the worst case, all the keys are clustered, let's say at the end of the table. If k hashes to any of the first n locations, only one probe is needed. But hashing to the n^{th} location would require probing all n full locations before finally wrapping around to find an empty location. Similarly, hashing to the second full cell, requires probing $(n-1)$ full cells plus the first empty cell, and so forth. Thus, under uniform hashing the average number of probes needed to insert a key would be

$$1 + [n + (n-1) + (n-2) + ... + 1]/m = 1 + n(n+1)/2m \approx n/4$$

Even though the average cluster length is 2, the cost for an unsuccessful search is $n/4$. In general, each cluster j of length n_j contributes $n_j(n_j+1)/2$ towards the total number of probes for all keys. Its contribution to the average is proportional the *squares* of the length of the cluster, making long cluster costly.

We won't attempt to analyze the cost of successful and unsuccessful searches, as considering cluster formation during linear probing is quite difficult. We make the following claim:

Claim 2.3. *When using linear probing in a hash table of size m that contains $n = \lambda m$ keys, the average number of probes needed for an unsuccessful search or an insert is*

$$\frac{1}{2}\left(1+\frac{1}{1-\lambda^2}\right)$$

and for a successful search is

$$\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right).$$

As you can see from the following table, which show the expected number of probes under uniform hashing, linear probing degrades significantly when the load factor increases:

λ	1/4	1/2	2/3	3/4	9/10
successful	1.2	1.5	2.0	3.0	5.5
unsuccessful	1.4	2.5	5.0	8.5	55.5

Linear probing is quite competitive, though, when the load factors are in the range (30-70%) as clusters tend to stay small. In addition, a few extra probes is mitigated when sequential access is much faster than random access, as is the case of caching. Because of primary clustering, though, it is sensitive to quality of the hash function or the particular mix of keys that result in many collisions or clumping. Therefore, it may not be a good choice for a general purpose hash tables.

2.2.2 Quadratic Probing

Quadratic probe sequences causes probes to move away from cluster, by making increasing larger jumps. The i^{th} probe is

$$h(k,i) = (h(k)+i^2) \mod m.$$

Although, quadratic probing voids primary clustering, it still has *secondary clustering*: When two keys hash to the same location, they have the same probe sequence. Since there are only m locations in the table, there are only m possible probe sequences.

One problem with quadratic probing is that probe sequences do not probe all locations in the table. But since there are $(p+1)/2$ quadratic residues when p is prime, we can make the following guarantee.

Claim 2.4. *: If m is prime and the table is at least half empty, then quadratic probing will always find an empty location. Furthermore, no locations are checked twice.*

Proof. (by contradiction) Consider two probe locations $h(k)+i^2$ and $h(k)+j^2, 0 \leq i,j < \lceil m/2 \rceil$. Suppose the locations are the same but $i \neq j$. Then

$$\begin{aligned} h(k)+i^2 &\equiv (h(k)+j^2) \mod m \\ i^2 &\equiv j^2 \mod m \\ i^2 - j^2 &\equiv 0 \mod m \\ (i-j)(i+j) &\equiv 0 \mod m \end{aligned}$$

Therefore, either $i-j$ or $i+j$ are divisible by m. But since both $i-j$ and $i+j$ are less than m and m is prime, they cannot be divisible by m. Contradiction.

Thus the first $\lceil m/2 \rceil$ probes are distinct and guaranteed to find an empty location. □

Computing the next probe is only slightly more expensive than linear probing as it can be computed without using multiplication:

$$\begin{aligned} h_i - h_{i-1} &\equiv (i^2 - (i-1)^2) \mod m \\ h_i &\equiv (h_{i-1} + 2i - 1) \mod m \end{aligned}$$

Unfortunately, requiring that the table remains less than half full makes quadratic probing space inefficient.

2.2.3 Double Hashing

Double hashing uses two hash functions, one to find the initial location to place the key and a second to determine the size of the jumps in the probe sequence. The i^{th} probe is

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m.$$

Keys that hash to the same location, are likely to hash to a different jump size, and so will have different probe sequences. Thus, double hashing avoids secondary clustering by providing as many as m^2 probe sequences.

How do we ensure every location is checked? Since each successive probe is offset by $h_2(k)$, every cell is probed if $h_2(k)$ is relatively prime to m. Two possible ways to ensure $h_2(k)$ is relatively prime to m are, either make $m = 2^k$ and design $h_2(k)$ so it is always odd, or make m prime and ensure $h_2(k) < m$. Of course, $h_2(k)$ cannot equal zero.

Double hashing behaves quite closely to uniform hashing for careful choices of h_1 and h_2. Under uniform hashing the average number of probes for an unsuccessful search or an insert is at most

$$1 + \lambda + \lambda^2 + \ldots = \left(\frac{1}{1-\lambda}\right)$$

and for a successful search is at most

$$\frac{1}{\lambda}\left(1 + \ln\left(\frac{1}{1-\lambda}\right)\right).$$

The former bound is because the probability of needing more than i probes is at most λ^i. A search always needs one probe, and with probability λ needs a second probe, and with probability λ^2 needs a third probe, and so on. The bound for a successful search for a key k follows the same probe sequences as when it was first inserted. So if k was the $(j+1)^{th}$ key inserted the cost of the for inserting it is at most $1/(1-j/m)$. Therefore the average cost of a successful search is at most

$$\begin{aligned}\frac{1}{n}\sum_{j=0}^{n-1}\frac{1}{1-j/m} &= \frac{m}{n}\sum_{j=0}^{n-1}\frac{1}{m-j}\\ &= \tfrac{1}{\lambda}(H_m - H_{m-n})\\ &\leq \tfrac{1}{\lambda}(\ln m + 1 - \ln(m-n))\\ &= \tfrac{1}{\lambda}\left(1+\ln\left(\tfrac{1}{1-\lambda}\right)\right)\end{aligned}$$

The table below shows the expected number of probes under the assumption of uniform hashing and is the best one can expect by open addressing.

λ	1/4	1/2	2/3	3/4	9/10
successful	1.2	1.4	1.6	1.8	2.6
unsuccessful	1.3	1.5	2.0	3.0	5.5

Comparing these numbers with the numbers in the table for linear probing, the linear probing numbers are remarkable close when the load factor is 50% or below.

The main advantage with double probing is that it allows for smaller tables (higher load factors) than linear or quadratic probing, but at the expense of higher costs to compute the next probe. The higher cost of computing the next probe may be preferable than longer probe sequences, especially when testing two keys equal is expensive.

2.3 Hash Table Summary

Hashing is a classic example of a space-time tradeoff: increase the space so table operations are faster, decrease the space but table operations are slower.

Separate chaining is simple to implement and is less sensitive to the quality of the hash function or load factors, so it is often the choice when it is unknown how many and how frequently keys may be inserted or deleted from the hash table. On the other hand open addressing can be more space efficient as there are no linked lists. Linear probing has the advantage that it has small constants and works well with caches. But it suffers from primary clustering, which means its performance is sensitive to collisions and to high load factors. Quadratic probing, on the other hand, avoids primary clustering, but still suffers from secondary clustering and requires rehashing as soon as load factor reaches 50%. Although double hashing reduces clustering, so high load factors are possible, finding suitable pairs of hash functions is somewhat more difficult and increases the cost of a probe.

2.4 Parallel Hashing

We assume a function $\texttt{injectCond}(IV,S) : (int \times \alpha)\texttt{seq} \times (\alpha\texttt{option})seq \to (\alpha\texttt{option})seq$. It takes a sequence of index-value pairs $\langle (i_1,v_1),\ldots,(i_n,v_n)\rangle$ and a target sequence S and conditionally writes each value v_j into location i_j of S. In particular it only writes the value if the location is set to NONE and there is no previous equal index in IV.

Using this we have:

```
1  fun insert(T,K) =
2  let
3    fun insert'(T,K,i) =
4      if |K| = 0 then T
5      else let
6        val T' = injectCond({(h(k,i),k) : k ∈ K}, T)
7        val K' = {k : k ∈ K | T[h(k,i)] ≠ k}
8      in
9        insert'(T',K',i+1)      end
10 in
11   insert'(T,k,1)
12 end
```

Note that if T is implemented using single threaded arrays, then this basically does the same work as the sequential version adding the keys one by one.

What does it mean for set and table implementations?

- Searches are faster in expectation
- Insertions are faster in expectation
- Map, reduce, and filter remain linear work
- Union/Merge can be slower.

Lecture 27 — Priority Queues and Leftist Heaps

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — 29 November 2012

Today:

- Priority Queues, revisited
- Leftist Heaps

1 Priority Queues

We have already discussed and used priority queues in a few places in this class. We used them as an example of an abstract data type. We also used them in the priority-first graph search to implement Dijkstra's algorithm, and Prim's algorithm for minimum spanning trees.

As you might have seen in other classes, a priority queue can also be used to implement an $O(n \log n)$ work (time) version of selection sort, often referred to as heapsort. The sort can be implemented as:

```
1  fun sort S =
2  let
3    val pq = iter Q.insert Q.empty S
4    fun sort' pq =
5    let
6      case (PQ.deleteMin pq) of
7        NONE ⇒ []
8      | SOME(v, pq') ⇒ v :: sort'(pq')
9  in
10   Seq.fromList(sort'pq)
11 end
```

Priority queues also have applications elsewhere, including

- Huffman Codes
- Clustering algorithms
- Event simulation
- Kinetic algorithms

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

What are some possible implementations of a priority queue?

With sorted and unsorted linked lists (or arrays), one of `deleteMin` and `insert` is fast and the other is slow. On the other hand balanced binary search trees (e.g., treaps) and binary heaps implemented with (mutable) arrays have $O(\log n)$ span for both operations. But why would you choose to use binary heaps over balanced binary search trees? For one, binary heaps provide a `findMin` operation that is $O(1)$ whereas for BSTs it is $O(\log n)$. Let's consider how you would build a priority queue from a sequence.

But first, let's review the heaps and search trees. A *min-heap* is a rooted tree such that the key stored at every node is less or equal to the keys of all its descendants. Similarly a max-heap is one in which the key at a node is greater or equal to all its descendants. A *search-tree* is a rooted tree such that the key sorted at every node is greater than (or equal to) all the keys in its left subtree and less than all the keys in its right subtree. Heaps maintain only a partial ordering, whereas search trees maintain a total ordering.

A binary heap is a particular implementation that maintains two invariants:

- Shape property: A complete binary tree (all the levels of the tree are completely filled except the bottom level, which is filled from the left).
- Heap property

Because of the shape property, a binary heap can be maintained in an array, and the index of the a parent or child node is a simple computation. Recall that operations on a binary heap first restore the shape property, and then the heap property.

To build a priority queue, we can insert one element at a time into the priority queue as we did in heap sort above. With both balanced binary search trees and binary heaps, the cost is $O(n \log n)$. Can we do better? For heaps, yes, build the heap recursively. If the left and right children are already heaps, we can just "shift down" the element at the root:

```
1  fun sequentialFromSeqS =
2  let
3    fun heapify(S, i) =
4      if (i >= |S|/2) then S
5      else
6        val S' = heapify(S, 2*i + 1)
7        val S'' = heapify(S', 2*i + 2)
8        shiftDown(S'', i)
9  in heapify(S, 0) end
```

With ST-sequences, `shiftDown` does $O(\log n)$ work on a subtree of size n. Therefore, `sequentialFromSeq` has work

$$W(n) = 2W(n/2) + O(\log n) = O(n)$$

We can build a binary heap in parallel with ST-sequences. If you consider S as a complete binary tree, the leaves are already heaps. The next level up of this tree, the roots of the subtrees violate

the heap property and need to be shifted down. Since the two children of each root are heaps, the result of shift down is a heap. That is, on each level of the complete tree, fix the heaps at that level by shifting down the elements at that level. The code below, for simplicity, assumes $|S| = 2^k - 1$ for some k:

```
1  fun fromSeq S : 'a seq =
2  let
3    fun heapify (S, d) =
4    let
5      val S' = shiftDown (S, ⟨2^d − 1, ..., 2^(d+1) − 2⟩, d)
6    in
7      if (d = 0) then S'
8      else heapify (S', d − 1)
9  in heapify (S, log2 n − 1) end
```

There is a subtly with this parallel `shiftDown`. It too needs to work one layer of the binary tree at a time. That is, it takes a sequence of indices corresponding to elements at level d and determines if the those elements need to swap with elements at level $d+1$. It does the swaps using `inject`. Then it calls `shiftDown` recursively using the indices to where the elements at d moved to in level $d+1$, if indeed they moved down. When it reaches the leaves it returns the updated ST-sequence.

This parallel version does the same work as the sequential version. But now span is $O(\log n)$ at each of the $O(\log n)$ layers of the tree:

$$S(n) = S(n/2) + O(\log n) = O(\log^2 n).$$

In summary, the table below shows that a binary heap is an improvement over more general purpose structures used for implementing priority queues. The shape property of a binary heap,

Implementation	`findMin`	`deleteMin`	`insert`	`fromSeq`
sorted linked list	$O(1)$	$O(1)$	$O(n)$	$O(n \log n)$
unsorted linked list	$O(n)$	$O(n)$	$O(1)$	$O(n)$
balanced search tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n \log n)$
binary heap	$O(1)$	$O(\log n)$	$O(\log n)$	$O(n)$

though, limits its ability to implement other useful priority queue operations efficiently. Next, we will a more general priority queue, meldable ones.

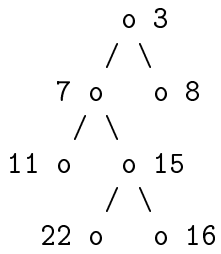
1.1 Meldable Priority Queues

Recall that, much earlier in the course, we introduced a meldable priority queue as an example of an abstract data type. It includes the `meld` operation, which is analogous to `merge` for binary search trees; It takes two meldable priority queues and returns a meldable priority queue that contains all the elements of the two input queues.

Today we will discuss one implementation of a meldable priority queue, which has the same work and span costs as binary heaps, but also has an efficient operation `meld`. This operation has work and span of $O(\log n + \log m)$, where n and m are the sizes of the two priority queues to be merged.

The structure we will consider is a 'leftist heap, which is a binary tree that maintains the heap property, but unlike binary heaps, it not does maintain the complete binary tree property. The goal is to make the `meld` fast, and in particular run in $O(\log n)$ work. First, let's consider how we could use `meld` and what might be an implementation of `meld` on a heap.

Consider the following a min-heap



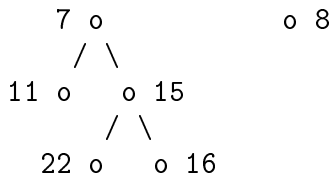
There are two important properties of a min-heap:

1. The minimum is always at the root.
2. The heap only maintains a partial order on the keys (unlike a BST that maintains the keys in a total order).

The first property allows us to access the minimum quickly, and it is the second that gives us more flexibility than available in a BST.

Let's consider how to implement the three operations `deleteMin`, `insert`, and `fromSeq` on a heap. Like `join` for treaps, the `meld` operation, makes the other operations easy to implement.

To implement `deleteMin` we can simply remove the root. This would leave:



This is simply two heaps, which we can use `meld` to join.

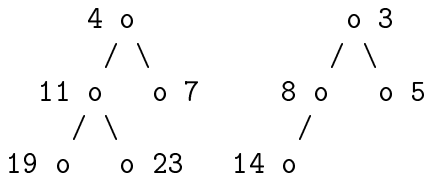
To implement `insert`(Q, v), we can just create a singleton node with the value v and then meld it with the heap for Q.

With `meld`, implementing `fromSeq` in parallel is easy using `reduce`:

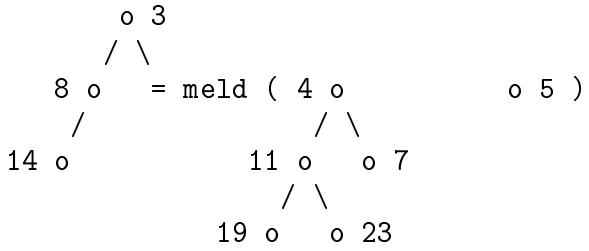
```
(* Insert all keys into priority queue *)
val pq = Seq.reduce Q.meld Q.empty (Seq.map Q.singleton S)
```

In this way, we can insert multiple keys into a heap in parallel: Simply build a heap as above and then meld the two heaps. There is no real way, however, to remove keys in parallel unless we use something more powerful than a heap.

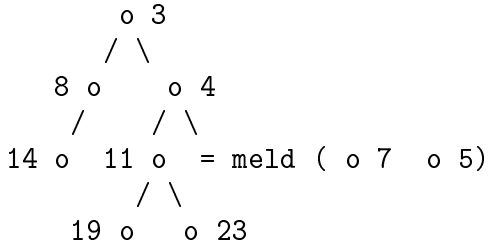
The only operation we need to care about, therefore, is the `meld` operation. Let's consider the following two heaps



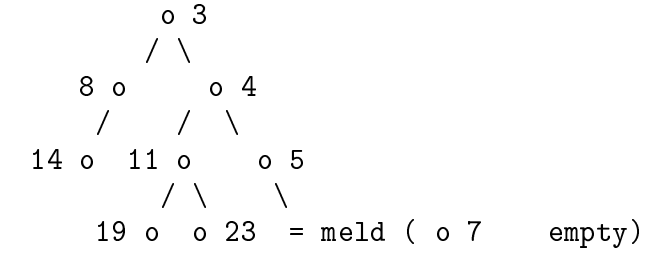
If we meld these two min-heaps, which value should be at the root? Well it has to be 3 since it is the minimum value. So what we can do is select the tree with the smaller root and then recursively meld the other tree with one of its children. In our case let's meld with the right child. So this would give us:



If we apply this again we get



and one more time gives:



Clearly if we are melding a heap A with an empty heap we can just use A. This algorithm leads to the following code:

```
1  datatype PQ = Leaf | Node of (key × PQ × PQ)

2  fun meld(A,B) =
3    case (A,B) of
4      (_,Leaf) ⇒ A
5    | (Leaf, _) ⇒ B
6    | (Node(k_a, L_a, R_a), Node(k_b, L_b, R_b)) ⇒
7        case Key.compare (k_a, k_b) of
8          LESS ⇒ Node(k_a, L_a, meld(R_a, B))
9        | _ ⇒ Node(k_b, L_b, meld(A, R_b))
```

This code traverses the right spine of each tree (recall that the right spine of a binary tree is the path from the root to the rightmost node). The problem is that the tree could be very imbalanced, and in general, we can not put any useful bound on the length of these spines—in the worst case all nodes could be on the right spine. In this case the `meld` function could take $\Theta(|A| + |B|)$ work.

1.2 Leftist Heaps

It turns out there is a relatively easy fix to this imbalance problem. The idea is to keep the trees so that the trees are always deeper on the left than the right. In particular, we define the *rank* of a node x as

$$\mathsf{rank}(x) = \#\text{ of nodes on the right spine of the subtree rooted at } x,$$

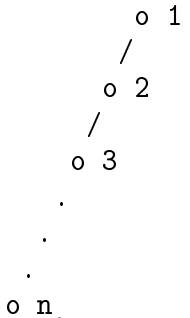
and more formally:

$$\mathsf{rank}(\mathsf{leaf}) = 0$$
$$\mathsf{rank}(\mathsf{node}(\_, \_, R) = 1 + \mathsf{rank}(R)$$

Now we require that all nodes of a leftist heap have the "leftist property". That is, if $L(x)$ and $R(x)$ are the left and child children of x, then we have:

Leftist Property: For all node x in a leftist heap, $\mathsf{rank}(L(x)) \geq \mathsf{rank}(R(x))$

This is why the tree is called leftist: for each node in the heap, the rank of the left child must be at least the rank of the right child. Note that this definition allows the following unbalanced tree.



This is OK since we only ever traverse the right spine of a tree, which in this case has length 1.

At an intuitive level, the leftist property implies that most of entries (mass) will pile up to the left, making the right spine of such a heap relatively short. In this way, all update operations we care about can be supported efficiently. We'll make this idea precise in the following lemma which will prove later; we'll see how we can take advantage of this fact to support fast meld operations.

Lemma 1.1. *In a leftist heap with n entries, the rank of the root node is at most* $\log_2(n+1)$.

In words, this lemma says *leftist heaps have a short right spine,* about $\log n$ in length. To get good efficiency, we should take advantage of it. Notice that unlike the binary search tree property, the heap property gives us a lot of freedom in working with left and right child of a node (in particular, they don't need to be ordered in any specific way). Since the right spine is short, our meld algorithm should, when possible, try to work down the right spine. With this rough idea, if the number of steps required to meld is proportional to the length of the right spine, we have an efficient algorithm that runs in about $O(\log n)$ work.

To make use of ranks we add a rank field to every node and make a small change to our code to maintain the leftist property: the meld algorithm below effectively traverses the right spines of the heaps A and B. (Note how the recursive call to `meld` are only with either (R_a, B) or (A, R_b).)

```
1  datatype PQ = Leaf | Node of (int × key × PQ × PQ)

2  fun rank Leaf = 0
3    | rank (Node(r, _, _, _)) = r

4  fun makeLeftistNode (v, L, R) =
5    if (rank(L) < rank(R))
6    then Node(1 + rank(L), v, R, L)
7    else Node(1 + rank(R), v, L, R)

8  fun meld (A, B) =
9    case (A, B) of
10     (_, Leaf) ⇒ A
11   | (Leaf, _) ⇒ B
12   | (Node(_, k_a, L_a, R_a), Node(_, k_b, L_b, R_b)) ⇒
13       case Key.compare(k_a, k_b) of
14         LESS ⇒ makeLeftistNode (k_a, L_a, meld(R_a, B))
15       | _ ⇒ makeLeftistNode (k_b, L_b, meld(A, R_b))
```

Note that the only real difference is that we now use `makeLeftistNode` to create a node and ensure that the resulting heap satisfies the leftist property assuming the two input heaps L and R did. It makes sure that the rank of the left child is at least as large as the rank of the right child by switching the two children if necessary. It also maintains the rank value on each node.

Theorem 1.2. *If A and B are leftists heaps then the* `meld`(A,B) *algorithm runs in* $O(\log(|A|) + \log(|B|)$ *work and returns a leftist heap containing the union of A and B.*

Proof. The code for `meld` only traverses the right spines of A and B, advancing by one node in one of the heaps. Therefore, the process takes at most $\mathsf{rank}(A) + \mathsf{rank}(B)$ steps, and each step does constant

work. Since both trees are leftist, by Lemma 1.1, the work is bounded by $O(\log(|A|) + \log(|B|))$. To prove that the result is leftist we note that the only way to create a node in the code is with `makeLeftistNode`. This routine guarantees that the rank of the left branch is at least as great as the rank of the right branch. □

Before proving Lemma 1.1 we will first prove a claim that relates the number of nodes in a leftist heap to the rank of the heap.

Claim: If a heap has rank r, it contains at least $2^r - 1$ entries.

To prove this claim, let $n(r)$ denote the number of nodes in the smallest leftist heap with rank r. It is not hard to convince ourselves that $n(r)$ is a monotone function; that is, if $r' \geq r$, then $n(r') \geq n(r)$. With that, we'll establish a recurrence for $n(r)$. By definition, a rank-0 heap has 0 nodes. We can establish a recurrence for $n(r)$ as follows: Consider the heap with root note x that has rank r. It must be the case that the right child of x has rank $r-1$, by the definition of rank. Moreover, by the leftist property, the rank of the left child of x must be at least the rank of the right child of x, which in turn means that $\mathsf{rank}(L(x)) \geq \mathsf{rank}(R(x)) = r-1$. As the size of the tree rooted x is $1 + |L(x)| + |R(x)|$, the smallest size this tree can be is

$$\begin{aligned} n(r) &= 1 + n(\mathsf{rank}(L(x))) + n(\mathsf{rank}(R(x))) \\ &\geq 1 + n(r-1) + n(r-1) = 1 + 2 \cdot n(r-1). \end{aligned}$$

Unfolding the recurrence, we get $n(r) \geq 2^r - 1$, which proves the claim.

Proof of Lemma 1.1. To prove that the rank of the leftist heap with n nodes is at most $\log(n+1)$, we simply apply the claim: Consider a leftist heap with n nodes and suppose it has rank r. By the claim it must be the case that $n \geq n(r)$, because $n(r)$ is the fewest possible number of nodes in a heap with rank r. But then, by the claim above, we know that $n(r) \geq 2^r - 1$, so

$$n \geq n(r) \geq 2^r - 1 \implies 2^r \leq n+1 \implies r \leq \log_2(n+1).$$

This concludes the proof that the rank of a leftist heap is $r \leq \log_2(n+1)$. □

1.3 Summary of Priority Queues

Already, we have seen a handful of data structures that can be used to implement a priority queue. Let's look at the performance guarantees they offer.

Implementation	`insert`	`findMin`	`deleteMin`	`meld`
(Unsorted) Sequence	$O(n)$	$O(n)$	$O(n)$	$O(m+n)$
Sorted Sequence	$O(n)$	$O(1)$	$O(n)$	$O(m+n)$
Balanced Trees (e.g. Treaps)	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(m \log(1 + \frac{n}{m}))$
Leftist Heap	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log m + \log n)$

Indeed, a big win for leftist heap is in the super fast `meld` operation—logarithmic as opposed to roughly linear in other data structures.

Syntax and Costs for Sequences, Sets and Tables

1 Pseudocode Syntax

The pseudocode we use in the class will use the following notation for operations on sequences, sets and tables. In the translations e, e_1, e_2 represent expressions, and p, p_1, p_2, k, k_1, k_2 represent patterns. The syntax described here is not meant to be complete, but hopefully sufficient to figure out any missing rules. Warning: Since we have been refining the notation as we go, this notation might not be completely consistent across the lectures.

Sequences

S_i	`nth` S i
$\|S\|$	`length`(S)
$\langle\,\rangle$	`empty`$()$
$\langle v \rangle$	`singleton`(v)
$\langle i, \ldots, j \rangle$	`tabulate` (**fn** $k \Rightarrow i + k$) $(j - i + 1)$
$\langle e : p \in S \rangle$	`map` (**fn** $p \Rightarrow e$) S
$\langle e : i \in \langle 0, \ldots, n-1 \rangle \rangle$	`tabulate` (**fn** $i \Rightarrow e$) n
$\langle p \in S \mid e \rangle$	`filter` (**fn** $p \Rightarrow e$) S
$\langle e_1 : p \in S \mid e_2 \rangle$	`map` (**fn** $p \Rightarrow e_1$) (`filter` (**fn** $p \Rightarrow e_2$) S)
$\langle e : p_1 \in S_1, p_2 \in S_2 \rangle$	`flatten`(`map` (**fn** $p_1 \Rightarrow$ `map` (**fn** $p_2 \Rightarrow e$) S_2) S_1)
$\langle e_1 : p_1 \in S_1, p_2 \in S_2 \mid e_2 \rangle$	`flatten`(`map` (**fn** $p_1 \Rightarrow \langle e_1 : p_2 \in S_2 \mid e_2 \rangle$) S_1)
$\sum_{p \in S} e$	`reduce add 0` (`map` (**fn** $p \Rightarrow e$) S)
$\sum_{i=k}^{n} e$	`reduce add 0` (`map` (**fn** $i \Rightarrow e$) $\langle k, \ldots, n \rangle$)
$\operatorname{argmax}_{p \in S}(e)$	`argmax compare` (`map` (**fn** $p \Rightarrow e$) S)

The meaning of `add`, 0, and `compare` in the `reduce` and `argmax` will depend on the type. The $\sum$ can be replaced with min, max, $\cup$ and $\cap$ with the presumed meanings. The function `argmax f S` : $(\alpha \times \alpha \to$ `order`$) \to (\alpha$ `seq`$) \to$ `int` returns the index in S which has the maximum value with respect to the order defined by the function f. $\operatorname{argmin}_{p \in S} e$ can be defined by reversing the order of compare.

Sets

$\|S\|$	$\texttt{size}(S)$
$\{\}$	$\texttt{empty}$
$\{v\}$	$\texttt{singleton}(v)$
$\{v \in S \mid e\}$	$\texttt{filter}\ (\textbf{fn}\ v \Rightarrow e)\ S$
$S_1 \cup S_2$	$\texttt{union}(S_1, S_2)$
$S_1 \cap S_2$	$\texttt{intersection}(S_1, S_2)$
$S_1 \setminus S_2$	$\texttt{different}(S_1, S_2)$
$\sum_{k \in S} e$	$\texttt{reduce add 0 (Table.tabulate}\ (\textbf{fn}\ k \Rightarrow e)\ S)$

Tables

$\|T\|$	$\texttt{size}(T)$
$\{\}$	$\texttt{empty}()$
$\{k \mapsto v\}$	$\texttt{singleton}(k, v)$
$\{e : v \in T\}$	$\texttt{map}\ (\textbf{fn}\ v \Rightarrow e)\ T$
$\{k \mapsto e : (k \mapsto v) \in T\}$	$\texttt{mapk}\ (\textbf{fn}\ (k, v) \Rightarrow e)\ T$
$\{k \mapsto e : k \in S\}$	$\texttt{tabulate}\ (\textbf{fn}\ k \Rightarrow e)\ S$
$\{v \in T \mid e\}$	$\texttt{filter}\ (\textbf{fn}\ v \Rightarrow e)\ T$
$\{(k \mapsto v) \in T \mid e\}$	$\texttt{filterk}\ (\textbf{fn}\ (k, v) \Rightarrow e)\ T$
$\{e_1 : v \in T \mid e_2\}$	$\texttt{map}\ (\textbf{fn}\ v \Rightarrow e_1)\ (\texttt{filter}\ (\textbf{fn}\ v \Rightarrow e_2)\ T)$
$\{k : (k \mapsto \_) \in T\}$	$\texttt{domain}(T)$
$\{v : (\_ \mapsto v) \in T\}$	$\texttt{range}(T)$
$T_1 \cup T_2$	$\texttt{merge}\ (\textbf{fn}\ (v_1, v_2) \Rightarrow v_2)\ (T_1, T_2)$
$T \cap S$	$\texttt{extract}(T, S)$
$T \setminus S$	$\texttt{erase}(T, S)$
$\sum_{v \in T} e$	$\texttt{reduce add 0 (map}\ (\textbf{fn}\ v \Rightarrow e)\ T)$
$\sum_{(k \mapsto v) \in T} e$	$\texttt{reduce add 0 (mapk}\ (\textbf{fn}\ (k, v) \Rightarrow e)\ T)$
$\operatorname{argmax}_{(k \mapsto v) \in T}(e)$	$\texttt{argmax max (mapk}\ (\textbf{fn}\ (k, v) \Rightarrow e)\ T)$

2 Function Costs

ArraySequence	*Work*	*Span*
`length`(T) `singleton`(v) `nth` S i `empty`()	1	1
`tabulate` f n	$\sum_{i=0}^{n-1} W(f(i))$	$\max_{i=0}^{n-1} S(f(i))$
`map` f S	$\sum_{e \in S} W(f(e))$	$\max_{e \in S} S(f(e))$
`map2` f S_1 S_2	$\sum_{i=0}^{\min(\lvert S_1\rvert,\lvert S_2\rvert)-1} W(f(S_{1i}, S_{2i}))$	$\max_{i=0}^{\min(\lvert S_1\rvert,\lvert S_2\rvert)-1} S(f(S_{1i}, S_{2i}))$
`filter` f S	$\sum_{s \in S} W(f(s))$	$\log \lvert S\rvert + \max_{s \in S} S(f(s))$
`reduce` f b S	$O\left(\lvert S\rvert + \sum_{f(x,y) \in \mathcal{O}_r(f,b,S)} W(f(x,y))\right)$	$\log \lvert S\rvert \max_{f(x,y) \in \mathcal{O}_r(f,b,S)} S(f(x,y))$
`scan` f b S	$O\left(\lvert S\rvert + \sum_{f(x,y) \in \mathcal{O}_s(f,b,S)} W(f(x,y))\right)$	$\log \lvert S\rvert \max_{f(x,y) \in \mathcal{O}_s(f,b,S)} S(f(x,y))$
`iter` f b_0 S	$O\left(\sum_{i=0}^{\lvert S\rvert-1} W(f(b_i, S_i))\right)$	$\sum_{i=0}^{\lvert S\rvert-1} S(f(b_i, S_i))$
`iterh` f b_0 S	$O\left(\sum_{i=0}^{\lvert S\rvert-1} W(f(b_i, S_i))\right)$	$\sum_{i=0}^{\lvert S\rvert-1} S(f(b_i, S_i))$
`showt` S `showti` S f	$\lvert S\rvert$	1
`showl` S	$\lvert S\rvert$	1
`hidet(NODE`(L,R)`)`	$\lvert L\rvert + \lvert R\rvert$	1
`hidel(CONS`(x,xs)`)`	$\lvert S\rvert$	1

`hidel(NIL)`
`hidet(ELT` e`)`
`hidet(EMPTY)`
	1	1

ArraySequence	*Work*	*Span*
`append`(S_1, S_2)	$\|S_1\| + \|S_2\|$	1
`drop`(S, n)	$\|S\| - n$	1
`take`(S, n) `drop`(S, n) `subseq` S (s, n)	n	1
`rake` S (a, b, s)	$\frac{\|b-a\|}{s}$	1
`splitMid`(S, i)	$\|S\|$	1
`flatten` S	$\|S\| + \sum_{e \in S} \|e\|$	$\log \|S\|$
`inject` I S	$\|I\| + \|S\|$	1
`partition` I S	$\|I\| + \|S\|$	1
`argmax` f S	$\|S\|$	$\log \|S\|$
`merge` f S_1 S_2	$\|S_1\| + \|S_2\|$	$\log(\|S_1\| + \|S_2\|)$
`sort` f S	$\|S\| \log \|S\|$	$\log^2 \|S\|$
`collate` f (S_1, S_2)	$\|S_1\| + \|S_2\|$	$\log(\min(\|S_1\|, \|S_2\|))$
`collect` f S	$\|S\| \log \|S\|$	$\log^2 \|S\|$
`fromList`(S) `%`(S)	$\|S\|$	$\|S\|$
`toString` f S	$\sum_{e \in S} W(f(e))$	$\sum_{e \in S} S(f(e))$
`fields` f S `tokens` f S	$\|S\|$	$\log \|S\|$

For `reduce`, $\mathcal{O}_r(f, i, S)$ represents the set of applications of f as defined in the documentation. For `scan`, $\mathcal{O}_s(f, i, S)$ represents the applications of f defined by the implementation of scan in the lecture notes. For `iter` and `iterh`, $b_i = f(b_{i-1}, S_{i-1})$. For `showti`, `argmax`, `merge`, `sort`, `collate`, `collect`, `fields`, and `tokens` the given costs assume that the work and span of the application of f is constant.

TreeSequence	*Work*	*Span*
`nth` $S\ i$	$\log n$	$\log n$
`tabulate` $f\ n$	$---$	$\log n + \max_{i=0}^{n} S(f(i))$
`map` $f\ S$	$---$	$\log \lvert S\rvert + \max_{s \in S} S(f(s))$
`showt` S	$\log \lvert S\rvert$	$\log \lvert S\rvert$
`hidet`(`NODE`(L,R))	$\log(\lvert L\rvert + \lvert R\rvert)$	$\log(\lvert L\rvert + \lvert R\rvert)$
`append`(S_1, S_2)	$\log(\lvert S_1\rvert + \lvert S_2\rvert)$	$\log(\lvert S_1\rvert + \lvert S_2\rvert)$
`drop`(S,n)	$\log(\lvert S\rvert - n)$	$\log(\lvert S\rvert - n)$
`take`(S,n) `subseq` $S\ (s,n)$	$\log n$	$\log n$
`partition` $I\ S$	$\sum_{p \in S} p$	$\log(\lvert I\rvert + \lvert S\rvert)$
`inject` $I\ S$	$\lvert I\rvert \lg(\lvert I\rvert + \lvert S\rvert)$	$\lg^2 \lvert I\rvert + \log \lvert S\rvert$
`merge` $f\ S_1\ S_2$	$\min(\lvert S_1\rvert, \lvert S_2\rvert) \cdot \lg(\lvert S_1\rvert + \lvert S_2\rvert)$	$\lg(\lvert S_1\rvert + \lvert S_2\rvert)$
`sort` $f\ S$	$\lvert S\rvert \log \lvert S\rvert$	$\log^2 \lvert S\rvert$
`collect` $f\ S$	$\lvert S\rvert \log \lvert S\rvert$	$\log^2 \lvert S\rvert$

For `singleton`, `length`, `filter`, `reduce`, `scan`, `sort` and `collect` the costs are the same as in ArraySequence. All $---$ entries are the same as ArraySequence. For `merge`, `sort`, and `collect` the costs assume that the work and span of the application of f is constant.

Single Threaded ArraySequence	*Work*	*Span*
`nth` $S\ i$ `update` $(i,v)\ S$	$O(1)$	$O(1)$
`inject` $I\ S$	$\lvert I\rvert$	1
`fromSeq` S `toSeq` S	$O(\lvert S\rvert)$	$O(1)$

Tree Sets and Tables	*Work*	*Span*
`size`(T) `singleton`(k, v)	$O(1)$	$O(1)$
`filter` f T	$O\Big(\sum_{(k,v)\in T} W(f(v))\Big)$	$O\Big(\lg\|T\| + \max_{(k,v)\in T} S(f(v))\Big)$
`map` f T	$O\Big(\sum_{(k,v)\in T} W(f(v))\Big)$	$O\Big(\max_{(k,v)\in T} S(f(v))\Big)$
`tabulate` f S	$O\Big(\sum_{k\in S} W(f(k))\Big)$	$O\Big(\max_{k\in S} S(f(k))\Big)$
`find` T k `insert` f (k, v) T `delete` k T	$O(\lg\|T\|)$	$O(\lg\|T\|)$
`merge` f (T_1, T_2) `extract` (T, S) `erase` (T, S)	$O\big(m \lg(\frac{n+m}{m})\big)$	$O\big(\lg(n+m)\big)$
`domain` T `range` T `toSeq` T	$O(\|T\|)$	$O(\lg\|T\|)$
`collect` S `fromSeq` S	$O(\|S\| \lg\|S\|)$	$O(\lg^2\|S\|)$
`union` (S_1, S_2) `intersection` (S_1, S_2) `difference` (S_1, S_2)	$O\big(m \lg(\frac{n+m}{m})\big)$	$O\big(\lg(n+m)\big)$

where $n = \max(|T_1|, |T_2|)$ and $m = \min(|T_1|, |T_2|)$. For `reduce` you can assume the cost is the same as `Seq.reduce f init (range(T))`. In particular `Seq.reduce` defines a balanced tree over the sequence, and `Table.reduce` will also use a balanced tree. For `merge` and `insert` the bounds assume the merging function has constant work.

TreeTables

function	*type*	*Work*	*Span*
`size`(T) `singleton`(k,v)	$\mathbb{T} \to \mathbb{N}$ $\mathbb{K} \times \alpha \to \mathbb{T}_\alpha$	1	1
`filter` f T	$(\alpha \to \mathbb{B}) \to \mathbb{T}_\alpha \to \mathbb{T}_\alpha$	$\sum_{(k,v)\in T} W(f(v))$	$\lg\|T\| + \max_{(k,v)\in T} S(f(v))$
`map` f T	$(\alpha \to \beta) \to \mathbb{T}_\alpha \to \mathbb{T}_\beta$	$\sum_{(k,v)\in T} W(f(v))$	$\max_{(k,v)\in T} S(f(v))$
`tabulate` f T	$(\mathbb{K} \to \alpha) \to \mathbb{S} \to \mathbb{T}_\alpha$	$\sum_{k\in S} W(f(k))$	$\max_{k\in S} S(f(k))$
`find` T k `insert` f (k,v) T `delete` k T	$\mathbb{T}_\alpha \to \mathbb{K} \to (\mathbb{K}$ opt$)$ $(\alpha \times \alpha \to \alpha) \to (\mathbb{K} \times \alpha) \to \mathbb{T}_\alpha \to \mathbb{T}_\alpha$ $\mathbb{K} \to \mathbb{T}_\alpha \to \mathbb{T}_\alpha$	$\lg\|T\|$	$\lg\|T\|$
`merge` f (T_1, T_2) `extract` (T,S) `erase` (T,S)	$(\alpha \times \alpha \to \alpha) \to (\mathbb{T}_\alpha \times \mathbb{T}_\alpha) \to \mathbb{T}_\alpha$ $\mathbb{T}_\alpha \times \mathbb{S} \to \mathbb{T}_\alpha$ $\mathbb{T}_\alpha \times \mathbb{S} \to \mathbb{T}_\alpha$	$m \lg(\frac{n+m}{m})$	$\lg(n+m)$
`domain` T `range` T `toSeq` T	$\mathbb{T}_\alpha \to \mathbb{S}$ $\mathbb{T}_\alpha \to (\alpha$ `Seq`$)$ $\mathbb{T}_\alpha \to ((\mathbb{K} \times \alpha)$ `Seq`$)$	$\|T\|$	$\lg\|T\|$
`collect` S `fromSeq` S	$(\mathbb{K} \times \alpha)$ `Seq` $\to \mathbb{T}_{\alpha\texttt{Seq}}$ $(\mathbb{K} \times \alpha)$ `Seq` $\to \mathbb{T}_\alpha$	$\|S\| \lg\|S\|$	$\lg^2\|S\|$

TreeSets

function	*type*	*Work*	*Span*
`size`(S) `singleton`(k)	$\mathbb{S} \to \mathbb{N}$ $\mathbb{K} \to \mathbb{S}$	1	1
`filter` f S	$(\mathbb{K} \to \mathbb{B}) \to \mathbb{S} \to \mathbb{S}$	$\sum_{k\in S} W(f(k))$	$\lg\|S\| + \max_{k\in S} S(f(k))$
`find` S k `insert` k S `delete` k S	$\mathbb{S} \to \mathbb{K} \to \mathbb{B}$ $\mathbb{K} \to \mathbb{S} \to \mathbb{S}$ $\mathbb{K} \to \mathbb{S} \to \mathbb{S}$	$\lg\|S\|$	$\lg\|S\|$
`fromSeq` S	$\mathbb{K}$ `Seq` $\to \mathbb{S}$	$\|S\| \log\|S\|$	$\lg^2\|S\|$
`union` (S_1, S_2) `intersection`(S_1, S_2) `difference`(S_1, S_2)	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$ $\mathbb{S} \times \mathbb{S} \to \mathbb{S}$ $\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	$m \lg(\frac{n+m}{m})$	$\lg(n+m)$

Where $n = \max(|T_1|, |T_2|)$ and $m = \min(|T_1|, |T_2|)$. For `reduce` you can assume the cost is the same as `Seq.reduce f init (range(T))`. In particular `Seq.reduce` defines a balanced tree over the sequence, and `Table.reduce` will also use a balanced tree. For `merge` and `insert` the bounds assume the merging function has constant work.

`tag` — Version 1.0 ***due:*** *wed, sep 5 @ 11:59pm*

1 Introduction

This assignment is meant to help you familiarize yourself with the hand-in mechanism for 15-210 and to get you thinking about proofs and coding again. To that end, you will answer some questions about the mechanics and infrastructure of the course, then implement two solutions to the parentheses distance problem and perform some analysis of your solutions. Finally, you will do some exercises involving Big-*O* notation and prove an important identity.

1.1 Submission

This assignment is distributed in a number of files in our `git` repository. Instructions on how to access that repository can be found at `http://www.cs.cmu.edu/~15210/resources/git.pdf`. This is how assignments will be distributed in this course.

This assignment requires that you submit both code and written answers.

All of your code for this assignment must be in the files `paren.sml` and `test.sml`. Submit your solutions by placing these files in your handin directory, located at

`/afs/andrew.cmu.edu/course/15/210/handin/<yourandrewid>/assn1/`

Do not submit any other files. Name the files exactly as above. You can run the check script located at

`/afs/andrew.cmu.edu/course/15/210/bin/check/01.sh`

to make sure that you've handed in appropriate files.

Your written answers must be in a file called `hw01.pdf` and must be typeset. You do not have to use LaTeX, but if you do, we have provided the file `defs.tex` with some macros you may find helpful. This file should be included at the top of your `.tex` file with the command `\input{/<file>/<path>/defs.tex}`.

Your `paren.sml` file must contain all the code that you want to have graded for this assignment and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.2 Naming Modules

The questions below ask you to organized your solutions in a number of modules written from scratch. Your modules must be named exactly as stated in the handout. Correct code inside an incorrectly named structure, or a structure that does not ascribe to the specified signatures, *will not receive any credit*.

You may not modify any of the signatures or other library code that we give you. We will test your code against the signatures and libraries that we hand out, so if you modify the signatures your code will not compile and you will not receive credit.

1.3 The SML/NJ Build System

This assignment includes a substantial amount of library code spread over several files. The compilation of this is orchestrated by CM through the file `sources.cm`. Instructions on how to use CM can be found at on the website at `http://www.cs.cmu.edu/~15210/resources/cm.pdf`.

2 Mechanics

The following questions are intended to make sure that you have read and understood the various policies for the course, as well as found the tools that we've set up to communicate with you.

Task 2.1 (1%). Describe the picture posted as an instructor's note on the course's Piazza board.

Task 2.2 (3%). In each of the following situations, have the students followed or broken the collaboration policy? Briefly justify your answers with a sentence or two.

1. Ludmilla, Joaquin, and Alexander have lunch together after 210 lecture. Over lunch, they discuss the homework assignment released earlier in the week, including their progress so far. After lunch, all three go to different classes and don't think about the 210 homework until that evening.

2. While working on 213 homework near his friends from 210, Jon has a moment of insight into the 210 homework assignment. He becomes excited, and tells his friends what he thought of. Ishmael hasn't gotten that far in the homework, so he doesn't quite understand what Jon is talking about. Nonetheless, Ishmael copies down what he thinks are the key bits of what he heard to look at when he gets that far.

3. Yelena has been working on the 210 homework but can't figure out why her solution isn't compiling. She asks Abida to work through a couple of toy examples of functor syntax with together, and they do so with a text editor and the SML REPL. Afterwards, Yelena gets her homework to compile and keeps working towards a solution.

Task 2.3 (1%). If you hand in your homework 25 hours after the posted due date, and you have no late days remaining, what is your maxiumum possible score?

3 The Parentheses Distance Problem

A string, s, is considered 'closed' if and only if it contains only '(' and ')' characters and is one of the following:

- The concatenation of two closed strings, $s_1 s_2$.
- A single closed string surrounded by a pair of 'matched' parentheses, (s_0).
- A single pair of matched parentheses, ().

More colloquially, s is closed if it could be compiled without generating unmatched-parentheses errors.

The distance between a pair of matched parentheses is the number of characters between the parentheses (exclusive). The maximum parentheses distance problem is to find the largest distance between a pair of matched parentheses in a closed string. More formally, for the closed string s, let M_s be the set of index pairs such that for $(i, j) \in M_s$, $i < j$ and (s_i, s_j) is a pair of matched parentheses. The maximum parentheses distance is

$$\max\{j - i - 1 | (i, j) \in M_s\}.$$

For example, for the string '**(**()()**)**()', the maximum parentheses distance would be 4.

When solving this problem, instead of interacting directly with strings, you will work with `paren` sequences, where the type `paren` is defined in a structure that ascribes to `PAREN_PACKAGE` and is given by

```
datatype paren = OPAREN | CPAREN
```

with `OPAREN` corresponding to a left parenthesis and `CPAREN` corresponding to a right parenthesis.

You will implement the function `parenDist : paren seq -> int option` such that `parenDist` S `= NONE` if S is not closed and `parenDist` S `= SOME` max if S is closed, where max is the maximum parentheses distance in S.

You will implement two solutions to this problem in the file `paren.sml`. Each solution will be a functor ascribing to the signature `PAREN`, defined in `PAREN.sig`. Each functor will take a structure ascribing to `PAREN_PACKAGE` as its only argument. The signature `PAREN` is defined in terms of the `SEQUENCE` signature from our library, which is documented in `http://www.cs.cmu.edu/~15210/resources/docs.pdf`. For testing, you should use the structure `ArrayParenPackage`, which uses the `ArraySequence` implementation of `SEQUENCE`.

How to indicate parallel calls? As seen in recitation, you can use the function `par` (inside the structure `Primitives`) to express calls that will be run in parallel. Parallel operations can also be expressed in terms of operations on sequences such as `map` or `reduce`. *In your code, be explicit about what calls are being made in parallel.*

3.1 Brute Force Implementation

Task 3.1 (10%).

Implement a brute-force solution to the maximum parentheses distance problem in a functor called `ParenBF`. You may use `match` from recitation 1 as part of your solution. You might also find `subseq` of the `sequence` library helpful.

Task 3.2 (5%).

What is the work and span for your brute-force solution? You can assume `subseq` has $O(m)$ work and $O(1)$ span, where m is the length of the resulting subsequence, and `match` has $O(n)$ work and $O(\log^2 n)$ span where n is the length of the sequence.

3.2 Divide and Conquer

Task 3.3 (25%).

Implement a solution to the maximum parentheses distance problem by divide-and-conquer recursive programming. If the work of showing any tree view of a sequence is denoted W_{showt}, the work of your solution must be expressed by the recurrence

$$W_{parenDist}(n) = 2\left(W_{parenDist}\left(\frac{n}{2}\right)\right) + W_{showt} + O(1)$$

with

$$W_{parenDist}(0) \in O(1)$$

A solution with correct input-output behavior but with work that is not described by this recurrence will not receive full credit. Write your implementation in a functor called `ParenDivAndConq`. On this assignment we are not requiring you to write a proof of correctness. But we advice that you work out proof by mathematical induction for your solution.

Task 3.4 (10%).

The specification in Task 3.2 stated that the work of your solution must follow a recurrence that was parametric in the work it takes to view a sequence as a tree. Naturally, this changes with the sequence implementation.

1. Solve the recurrence with the assumption that $W_{showt} \in O\left(\lg n\right)$.
2. Solve the recurrence with the assumption that $W_{showt} \in O(n)$ where n is the length of the input sequence.
3. In two or three sentences, describe what data structure you would use to implement the sequence α `seq` so that `showt` would actually have $O(\lg n)$ work.
4. In two or three sentences, describe what data structure you would use to implement the sequence α `seq` so that `showt` would actually have $O(n)$ work.

3.3 Testing Your Code

Task 3.5 (10%).

Write a structure called `ParenTest` ascribing to the signature `TESTS`. Your structure should thoroughly and carefully test both your implementations of the `PAREN` signature. This should include both edge cases and more general test cases on specific sequences.

You should write `ParenTest` in the file `test.sml`. At submission time there should not be any testing code in the same file as your various `PAREN` implementations, as it can make it difficult for us to test your code when you turn it in.

Your brute force implementation will likely be simple enough that it will have few edge cases and require a relatively small number of tests. Once you have confirmed that your brute-force solution works, you can write more exhaustive tests by comparing the faster solution against the slower one on many

sequences including those whose solution would be inconveinent to calculate by hand. **No tests should evaluate at compile time.**

To aid in your tests, we have provided the function `strToParens` which will convert a string containing only the '(' and ')' characters into the corresponding `paren seq`. We have also provided a basic test framework to serve as an example of the type of code we are expecting.

4 Asymptotics

For this problem, let's recall the definition of big-O:

Definition 4.1. A function $f : \mathbb{N} \to \mathbb{R}_+$ is in $O(g)$ if and only if there exist constants $N_0 \in \mathbb{N}$ and $c \in \mathbb{R}_+$ such that for all $n \geq N_0$, $f(n) \leq c \cdot g(n)$.

Task 4.1 (5%). Rearrange the list of functions below so that it is ordered with respect to O—that is, for every index i, all of the functions with index less than i are in Big-O of the function at index i. You can just state the ordering; you don't need to prove anything.

1. $f(n) = n^{\lg(2n)}$
2. $f(n) = 108^n$
3. $f(n) = n^{1.5}$
4. $f(n) = (2n)!$
5. $f(n) = 23n^{42} + 15n^{16} + 4n^8$
6. $f(n) = \lg(n)$
7. $f(n) = n^n$

Task 4.2 (15%). Carefully **prove or disprove** each of the following statements. Remember that verbose proofs are not necessarily careful proofs: your answers for these questions will likely fit on one page.

1. O is a transitive relation on functions. That is to say, for any functions f, g, h, if $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$.
2. O is a symmetric relation on functions. That is to say, for any functions f and g, if $f \in O(g)$, then $g \in O(f)$.
3. O is an anti-symmetric relation on functions. That is to say, for any functions f and g, if $f \in O(g)$ and $g \in O(f)$, then $f = g$.

Task 4.3 (10%). Show that for $a \in \mathbb{R}$, $a \neq \pm 1$, $1 + a^2 + a^4 + \cdots + a^{2n} = \frac{a^{2n+2}-1}{a^2-1}$.

`tag` — Version 1.0 ***due:*** *mon, sep 17 @ 11:59pm*

Disclaimer:
We will not grade non-compiling code.

1 Introduction

This assignment is meant to help you practice implementing, analyzing, and proving the correctness of a divide and conquer algorithm, and familiarize you with the sequence `scan` operation. There are two separate algorithms to code in this assignment; the The Pittsburgh Skyline problem will work towards all of those stated goals, while the Binary Bignum problem will primarily test your command of the `scan` operation.

We will go over `scan` in more detail next week in lecture.

You will likely find this assignment more conceptually challenging than the first assignment. Expect it to take significantly longer than the first assignment to complete.

1.1 Submission

This assignment is distributed in a number of files in our `git` repository. Instructions on how to access that repository can be found at `http://www.cs.cmu.edu/~15210/resources/git.pdf`. This is how assignments will be distributed in this course.

This assignment requires that you submit both code and written answers.

Submit your solutions by placing you solution files in your handin directory, located at

`/afs/andrew.cmu.edu/course/15/210/handin/<yourandrewid>/assn2/`

Name the files exactly as specified below. You can run the check script located at

`/afs/andrew.cmu.edu/course/15/210/bin/check/02/check.pl`

to make sure that you've handed in appropriate files. Do not submit any other files.

Your written answers must be in a file called `hw02.pdf` and must be typeset. You do not have to use LaTeX, but if you do, we have provided the file `defs.tex` with some macros you may find helpful. This file should be included at the top of your `.tex` file with the command `\input{/<file>/<path>/defs.tex}`.

For the programming part, the only files you're handing in are

```
skyline.sml
skyline-test.sml
bignum.sml
bignum-test.sml
```

These files must contain all the code that you want to have graded for this assignment and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.2 Naming Modules

The questions below ask you to organized your solutions in a number of modules written from scratch. Your modules must be named exactly as stated in the handout. Correct code inside an incorrectly named structure, or a structure that does not ascribe to the specified signatures, *will not receive any credit*.

You may not modify any of the signatures or other library code that we give you. We will test your code against the signatures and libraries that we hand out, so if you modify the signatures your code will not compile and you will not receive credit.

1.3 The SML/NJ Build System

This assignment includes a substantial amount of library code spread over several files. The compilation of this is orchestrated by CM through the file `sources.cm`. Instructions on how to use CM can be found at on the website at `http://www.cs.cmu.edu/~15210/resources/cm.pdf`.

2 Function specifications

A style guideline that worked well last semester: We're going to ask that you give every function (and other complex values, at your discretion) something like a *contract* (from 122) or a *purpose* (from 150).

We ask that you annotate each function with its *type*, any *requirements* of its arguments (if it's a partial function), and any *promises* about its return values. A familiar example might look like this:

```
(* match : paren seq -> bool
   [match S] returns true if S is a well-formed parenthesis sequence
    and false otherwise. It does so by invoking the match' helper
    function below
*)
fun match s =
    let
        (* [match' s : paren seq -> (int * int)
           As we showed in recitation, every paren sequence
           can be reduced (by repeatedly deleting the substring "()")
           to a string of the form ")^i(^j".
           [match s] uses a divide and conquer algorithm to return (i,j)
           for s.
        *)
        fun match' s =
            case (showt s) of
                EMPTY => (0,0)
            | ELT OPAREN => (0,1)
            | ELT CPAREN => (1,0)
            | NODE (L,R) =>
                let
                    val ((i,j),(k,l)) = par (fn () => match' L, fn () => match' R)
                in
                    (* s = LR = )^i(^j)^k(^l
                        so we cancel the pairs in the middle
                    *)
```

```
                    case Int.compare(j,k)
                    of GREATER => (i, l + j - k)
                    | _ => (i + k - j, l)
    end
in
    case (match' s)
    (* a sequence is well-formed iff it reduces to (0,0) *)
    of (0,0) => true
    | _ => false
end
```

If the purpose or correctness if a piece of code is not obvious, you should add a comment as well (for example, the code above makes note of the argument that every paren sequence can, for the purposes of `match`, be reduced to "`)^i(^j`"). Ideally, your comments would convince a reader that your code is correct.

3 Writing Proofs

In 15-150, you were encouraged to write correctness proofs by stepping through your code. *Do not do this in 15-210.* For complicated programs, that level of detail quickly becomes tedious for you and your TA. A helpful guideline is that you should prove that your algorithm is correct, not your code. But your proof should highlight the critical steps and describe your algorithm in sufficient detail that your algorithm maps straightforwardly onto your code. Consider Theorem 3.2 of lecture 3, as an example of the level of detail we expect.

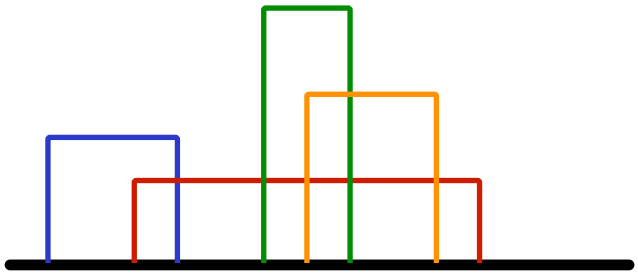
4 Warm up Recurrences

Task 4.1 (17%). Determine the complexity of the following recurrences. Give tight Θ-bounds, and justify your steps to argue that your bound is correct. Recall that $f \in \Theta(g)$ if and only if $f \in O(g)$ and $g \in O(f)$. You may whatever method you choose to show your bound is correct, except you must use the substitution method for problem 3.

1. $T(n) = 3T(n/4) + \Theta(n)$
2. $T(n) = 21T(\frac{n-3}{23}) + \Theta(n)$
3. $T(n) = 2T(n/2) + \Theta(\sqrt{n})$ (Prove by substitution.)
4. $T(n) = \sqrt{n}T(\sqrt{n}) + \Theta(n^2)$.

5 Pittsburgh Skyline

Did you know the Pittsburgh skyline was rated the second most beautiful vista in America by USA Weekend in 2003? However, you won't get to go out and see it any time soon, because you're an overworked Carnegie Mellon student. The 15-210 staff felt sorry for you, so we recorded the location and the silhouette of each tall building in Pittsburgh. Using this data, you can calculate the silhouette of Pittsburgh's skyline.



In this instance of the problem, we will assume that each building silhouette b is a two-dimensional rectangle, represented as the triple (ℓ, h, r). The corners of the rectangle will be at $(\ell, 0)$, (ℓ, h), (r, h), and $(r, 0)$. That is, the base of the building runs along the ground from $x = \ell$ to $x = r$, and the building is h tall. (Contrary to popular belief, the city's ground is flat; this happened last semester, when most of you were too busy with 15-251 to notice).

Definition 5.1 (The Pittsburgh Skyline Problem)**.** Given a non-empty set of buildings $B = \{b_1, b_2, \ldots, b_n\}$, where each $b_i = (\ell_i, h_i, r_i)$, the *Pittsburgh skyline problem* is to find a set of points $S = \{p_1, p_2, ..., p_{2n}\}$, where each $p_j = (x_j, y_j)$ such that

$$S = \left\{ \Big(x, \max(h : (\ell, h, r) \in B \wedge x \in [\ell, r)) \Big) : x \in \bigcup_{(\ell,h,r)\in B} \{\ell, r\} \right\}$$

In other words, the x-coordinates expressed in S are exactly the x-coordinates expressed in B, and the y-coordinate for every element $(x_j, y_j) \in S$ is the height of the tallest building $b_i = (\ell_i, h_i, r_i)$ for which $\ell_i \leq x_j < r_i$. The input set of buildings is represented as an unsorted sequence of tuples, and the output set is represented as a sequence of points sorted by x-coordinate. An example of a skyline instance is given below.

5.1 Logistics

For this problem you will hand in file `skyline.sml`, which should contain a functor ascribing to the signature `SKYLINE` defined in `SKYLINE.sml`. Your functor will take as a parameter a structure ascribing the signature `SEQUENCE`, and implement the function `skyline:(int*int*int) seq -> (int*int) seq`.

5.2 Implementation Instructions

There are many algorithmic approaches to this problem, including brute force, sweep line, and divide and conquer. As you may have guessed, you will implement a work-optimal, low-span, divide-and-conquer

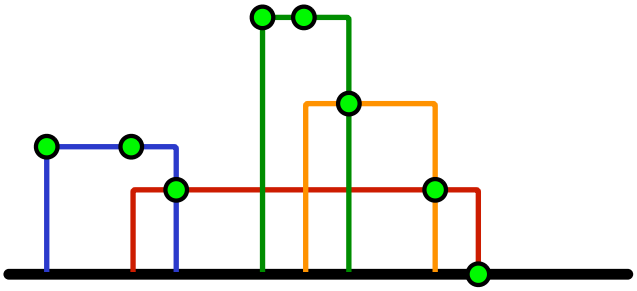
Input sequence:

ℓ_l	h_i	r_i
1	3	4
3	2	11
6	6	8
7	4	10

Output sequence:

x_j	y_j
1	3
3	3
4	2
6	6
7	6
8	4
10	2
11	0

Output points shown on input buildings:



solution to this problem. Even with divide and conquer, there are several choices on how to divide the problem. For example, you can divide along the x-axis, finding the skylines for $x < x'$ and for $x \geq x'$, or you can divide along the y-axis, finding the skylines for tall buildings and for short buildings. Alternately, you can divide the sequence of buildings into two sequences with a nearly equal number of buildings. You will use this last approach.

To simplify your solution, you may assume that no two buildings have the same left or right x-coordinate and that the heights and x-coordinates are positive integers.

You may have noticed that the definition of a skyline output sequence includes redundant points, and that removal of points with the same y-coordinate as the previous point would still result in a fully defined skyline. For example, the two points (3,3) and (7, 6) in the example above are redundant. **Omit these points from the sequence returned by your `skyline` function.**

The work of your divide-and-conquer steps must satisfy the recurrence

$$W_{skyline}(n) = 2W_{skyline}(n/2) + O(n) + W_{combine}(n), \tag{1}$$

where $W_{combine}(n)$ denotes the work for the combine step.

Task 5.1 (30%).

Implement a divide and conquer solution to the Pittsburgh Skyline Problem. The starter code which you will pull from the Git handout system contains the file `skyline.sml` in which to put your `Skyline` functor. Be sure to remove redundant points from the sequence you return, as specified above.

For full credit, **we expect your solution to have $O(n \log n)$ work and $O(\log^2 n)$ span**, where n is the number of input buildings. Carefully explain why your divide-and-conquer steps satisfy the specified recurrence and prove a closed-form solution to the recurrence.

The `copy_scan`, which will be discussed next week in lecture, may be helpful in your combine step.

Task 5.2 (4%). Implement the function `all` in the functor `SkylineTest` in `skyline-test.sml`. `all` should thoroughly and carefully test your implementation of the `SKYLINE` signature, returning true if and only if your tests succeed. Tests should include both edge cases and more general test cases on specific sequences.

At submission time there should not be any testing code in `skyline.sml`, as it can make it difficult for us to read and test your code when you turn it in.

Task 5.3 (15%).

Prove the correctness of your divide-and-conquer algorithm by induction. Be sure to carefully state the theorem that you're proving and to note all the algorithm steps in your proof.

You can use (strong) mathematical induction on the length of the input sequence. That is,

> Let $P(\cdot)$ be a predicate. To prove that P holds for every $n \geq 0$, we prove:
>
> 1. $P(0)$ holds, and
> 2. For all $n \geq 0$, if $P(n')$ holds for all $n' < n$, then $P(n)$.

Or, you can use the following structural induction principle for abstract sequences:

> Let P be a predicate on sequences. To prove that P holds for every sequence, it suffices to show the following:
>
> 1. $P(\langle\,\rangle)$ holds,
> 2. For all x, $P(\langle x \rangle)$ holds, and
> 3. For all sequences S_1 and S_2, if $P(S_1)$ and $P(S_2)$ hold, then $P(S_1 @ S_2)$ holds.

6 Binary Bignum

Native hardware integer representations are typically limited to 32 or 64 bits, which can be insufficient for computations which result in very large numbers. Some cryptography algorithms, for example, utilize large primes that require over 500 bits to represent. This motivates the implementation of an arbitrary-precision integer types and the operations to support them.

In this problem, you will implement addition and subtraction for arbitrary-precision non-negative integers represented as sequences of bits. This is trivial to achieve with a sequential algorithm. Needless to say, in 15-210 you must find a lower span solution to this problem.

6.1 Implementation Instructions

We represent an integer with the type `bignum` which is defined as a `bit seq`, where

```
datatype bit = ZERO | ONE
```

We adopt the convention that `bignums` begin with their least-significant bit. Furthermore, `bignums` never contain trailing zeros—The `bignum` representing 0 is an empty sequence, and all other `bignums` must end in a `ONE`. For example, the value represented by the `bignum` $\langle 0, 1, 1 \rangle$ is 6, and the `bignum` $\langle 1, 0, 0 \rangle$ is invalid (it would represent 1 if the trailing zeros were removed). **You must follow this convention for your solutions.**

Our `bignum` implementation will support addition and restricted subtraction (where the result is undefined if the second operand is larger than the first). You will complete the functor `BigNum` in `bignum.sml`, which ascribes to the signature `BIGNUM`. To help you get started, the starter code already has the `bignum` type declared and the infix operators `++` and `--` defined for you.

6.1.1 Addition

Task 6.1 (20%). Implement the addition function

```
++ :  bignum * bignum -> bignum
```

in the functor `BigNum` in `bignum.sml`. For full credit, on input with m and n bits, your solution must have $O(m+n)$ work and $O(\lg(m+n))$ span. For reference, our solution has 40 lines with comments.

The main challenge in meeting the cost bound lies in propagating the carry bits. For example, try adding 1 to $(11101111111111)_2$ and you will see a "ripple effect." You should use `scan` to get around this, but you need to come up with an associative binary operator. As a hint, we have also provided you with an additional datatype which you may find useful:

```
datatype carry = GEN | PROP | STOP
```

where `GEN` stands for generate, `PROP` for propagation, and `STOP` for stop. You might want to work out a few small examples to understand what is happening. Do you see a pattern in the following example?

```
1000100011
1001101001 +
```

For more inspiration, you should refer to lecture notes on the Sequence `scan` operation.

6.1.2 Subtraction

Task 6.2 (10%). Implement the subtraction function

```
-- :  bignum * bignum -> bignum
```

in the functor `BigNum` in `bignum.sml`, where `x -- y` computes the number obtained by subtracting y from x. You may assume that $x \geq y$. You may also assume for this problem that ++ has been implemented as specified above, even if your ++ is incorrect or does not meet cost bounds. For full credit, if x has n bits, your solution must have $O(n)$ work and $O(\lg n)$ span. Our solution has 20 lines with comments.

Perhaps the easiest way to implement subtraction is to use *two's complement* representation for negation, which you should recall from 15-122 or 15-213. As a quick review: two's complement uses k bits to encode the integers -2^{k-1} to $2^{k-1}-1$. When the most significant bit, or "sign bit", of a two's complement value is zero, the remaining $k-1$ bits are directly interpreted as binary, and thus have a value from 0 to $2^{k-1}-1$. When the sign bit is one, the value represented is the binary interpretation of the $k-1$ bits minus 2^{k-1}, which is a value from -2^{k-1} to -1. A two's complement number can be negated by flipping all the bits and adding 1.

6.1.3 Testing

The `BIGNUM` signature exports `add` and `sub`; we've also provided you with utility functions to convert between `bignum` and SML's `IntInf` (the standard arbitrary precision integer type) which you should use in your tests.

Task 6.3 (4%). Implement the function `all` in the functor `BigNumTest` in `bignum-test.sml`. `all` should thoroughly and carefully test your implementation of the `BIGNUM` signature, returning true if and only if your tests succeed. Tests should include both edge cases and more general test cases on specific sequences.

Although not required, it would be good practice to implement random testing of your functions against `IntInf`'s add and subtract. You can use the `Random210` structure in the 210 library or SML-NJ's `Random` to perform random testing. Be aware that the SML-NJ's `Random` functions are not pure functions.

At submission time there should not be any testing code in `bignum.sml`, as it can make it difficult for us to read and test your code when you turn it in.

`tag` — Version 1.0 ***due:*** *mon, sep 24 @ 11:59pm*

Disclaimer:
We will not grade non-compiling code.

1 Introduction

In this assignment you will program thousands of monkeys on typewriters to generate Shakespeare and pass spam filters.

With evil robots ubiquitous in our lives, it is ever more important to understand their attempts to thwart us. Thus you will play the role of the evil robots trying to pass Turing Tests. A Turing Test is an interaction with an agent designed to determine whether that agent is human or machine. CAPTCHAs are one such example using images, but suppose we limit ourselves to the medium of text. How can a machine fool a text-based Turing Test? It can't just deliver canned responses; there must be some generativity and randomness. But it must "sound like" human language.

Similarly, you could specialize a Turing Test to determine *which* human generated the text; and on the opposite side, you can generate text to sound like someone specific, such as Shakespeare or Guy Blelloch. The same program can imitate a stylistic voice of your choosing by being predicated on the *input corpus*, or large body of text from which it generates its vocabulary.

One way to generate text from a corpus is based on *k-grams*: chunks of text k words long. After selecting k words, you make a random choice for the $k+1$st word weighted by the frequency of that $k+1$-gram in the input corpus.

Your task is to write a library for parsing text and storing k-gram information in a table. Then, you will use this library to write a short application for generating realistic, Turing Test-passing text, and you will test them on real corpora.

1.1 Input data

The handout will include three files for input: `shakespeare.txt`, `kennedy.txt` and `mobydick.txt`.

1.2 Submission

This assignment is distributed in a number of files in our `git` repository. Instructions on how to access that repository can be found at `http://www.cs.cmu.edu/~15210/resources/git.pdf`. This is how assignments will be distributed in this course.

This assignment requires that you submit both code and written answers.

Submit your solutions by placing you solution files in your handin directory, located at

`/afs/andrew.cmu.edu/course/15/210/handin/<yourandrewid>/assn3/`

Name the files exactly as specified below. You can run the check script located at

```
/afs/andrew.cmu.edu/course/15/210/bin/check/03/check.pl
```

to make sure that you've handed in appropriate files. Do not submit any other files.

Your written answers must be in a file called `hw03.pdf` and must be typeset. You do not have to use LaTeX, but if you do, we have provided the file `defs.tex` with some macros you may find helpful. This file should be included at the top of your `.tex` file with the command `\input{<path>/defs.tex}`.

For the programming part, the only files you're handing in are

```
parser.sml
parser-test.sml
kgram-stats.sml
kgram-test.sml
babble.sml
```

These files must contain all the code that you want to have graded for this assignment and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Naming Modules

The questions below ask you to organized your solutions in a number of modules written from scratch. Your modules must be named exactly as stated in the handout. Correct code inside an incorrectly named structure, or a structure that does not ascribe to the specified signatures, *will not receive any credit*.

You may not modify any of the signatures or other library code that we give you. We will test your code against the signatures and libraries that we hand out, so if you modify the signatures your code will not compile and you will not receive credit.

1.4 The SML/NJ Build System

This assignment includes a substantial amount of library code spread over several files. The compilation of this is orchestrated by CM through the file `sources.cm`. Instructions on how to use CM can be found at on the website at `http://www.cs.cmu.edu/~15210/resources/cm.pdf`.

2 Parallel Text Parsing

In this task, you will write a parallel string *tokenizer* using the familiar *sequence* operations (`tabulate`, `filter`, `map2`,..).

Task 2.1 (10%). Implement a functor

```
Parser(Seq : SEQUENCE) : PARSER
```

in the file `parser.sml`. The `PARSER` signature is:

```
signature PARSER =
sig
  structure Seq : SEQUENCE
  val tokens : (char -> bool) -> string -> string Seq.seq
end
```

`tokens f s` should have the following behavior: let those characters `c` for which `f c` returns `true` be called *delimiting characters*. The result should be a sequence of maximal non-empty substrings (tokens) from the input string containing no delimiting characters. By maximal we mean that they cannot be extended on either side without including a delimiter. When the sequence of tokens is concatenated it must form the equivalent of `s` with all delimiting characters removed. It should be identical in behavior to `ArraySequence.tokens`. For our application, we'll use the function `fn c => not (Char.isAlphaNum c)`, but your `tokens` function should work correctly on other choices of delimiters as well.

For full credit your implementation must use $O(n)$ work and $O(l + \log n)$ span, where n is the length of the input string and l is the length of the largest token returned. You can assume that `String.substring(s,i,l)` takes $O(l)$ work and span.

Remark: For parsing you will probably end up evaluating large amount of conditionals. We found in our test, that SML/NJ compiler has a bug when compiling "case x of" statements, leading to very slow running time and even a memory leak. If you experience same problem, you might need to change your "case" to if-statements.

Task 2.2 (5%). Test your implentation in `parser-test.sml`. It should be easy to test your implementation against a benchmark.

Task 2.3 (5%). Explain informally why your implementation has work $O(n)$ and span of $O(l + \log n)$.

3 K-Gram Statistics

The next task is to implement a data structure to support a KGRAM_STATS abstract data type for storing statistics for a corpus. The functions of the data type build the data structure from the corpus (represented as sequence of tokens) and then can answer queries about any given "k-gram" (k consecutive tokens). For a given K, the kgram-stats table needs to store information for all k-grams that appear in the corpus for $k = 0$ (the empty k-gram) up to K. All code from this section should go in the file `kgram-stats.sml`

Assumptions: Throughout this problem, we'll assume that K is constant and that all words (tokens) have length at most a constant L_0.

Task 3.1 (10%). As a preliminary task, you will implement a function that creates a histogram. As an input it will take an ordering operator (such as String.compare) and a sequence of values. As an output it gives a list of tuples (key, n) for each unique key in the input sequence, where n is the number of occurrences of the key in the input.

```
signature HISTOGRAM =
sig
    structure Seq : SEQUENCE
    val histogram :  ('a * 'a -> order) -> 'a Seq.seq -> ('a * int) Seq.seq
end
```

You will probably find this function useful in the main task. The cost of histogram, assuming the ordering operator has constant cost, should be $O(n \log n)$ work and $O(\log^2 n)$ span.

Task 3.2 (30%). Create a structure `KgramStatsTable : KGRAM_STATS`. Use appropriate internal data representation to achieve the cost specifications.

For full credit you need to match the following cost bounds: For an input corpus with n tokens the costs of `make_stats` must be bounded by $O(n \log n)$ work and $O(\log^2 n)$ span, and the cost of both `lookup_freq` and `lookup_extensions` bounded by $O(\log n)$ work and span.

The `KGRAM_STATS` signature follows:

```
(* This is the signature for the kgram-statistics Abstract Data Type *)
signature KGRAM_STATS =
sig
  structure Seq : SEQUENCE
  type kgramstats (* self type *)
  type kgram = token Seq.seq
  type token = string

  (* make_stats tokens K
     Construct the underlying data structure given the sequence tokens
     representing the corpus and a maximum k-gram size K. *)
  val make_stats: token Seq.seq -> int -> kgramstats

  (* lookup_freq corpus prefix token
     For the corpus and a "prefix" of length at most K returns a pair
     consisting of:
      1) the number of times "token" appeared immediately after "prefix"
         for |prefix|=0 this is the total # of times token appears
      2) the total number of tokens that appear after "prefix"
         for |prefix|=0 this is the total size of the corpus *)
  val lookup_freq : kgramstats -> kgram -> token -> (int * int)

  (* lookup_extensions corpus prefix
     For the corpus and a "prefix" of length at most K returns a
     sequence of pairs each consisting of
      1) a token that appears at least once immediately after "prefix", and
      2) a count of how many times that token appears
```

```
     Every token that appears after "prefix" must appear in the seq *)
  val lookup_extensions : kgramstats -> kgram -> ((token * int) Seq.seq)
end
```

Hint: In the library directory the file `defaults.sml` contains a structure definition for Tables with string sequence as a key `StringSeqTable`. You might find it helpful. You can access it as `Default.StringSeqTable`.

Task 3.3 (5%). Write a test structure in `kgram-test.sml` that tests your implementation of `KGRAM_STATS` using $K = 3$ and a hard-coded test corpus (of at least 50 tokens). Test that it returns correct values for prefixes of length $0, 1, 2$ and 3 using prefixes from beginning, middle and end of the input corpus. Test that it works properly with prefixes that do NOT occur in the test corpus.

4 Babble

Using the K-gram statistics data type, it is easy to write an algorithm that generates text that is statistically similar to an input corpus. For example, to write pseudo-Shakespeare, you would compute statistics of Shakespeare's texts and use it to generate new masterpieces. We call this the babble problem. You need to implement the following signature consisting of the babble problem applied to one sentence and a document, and a helper function `choose_random`.

```
signature BABBLE = sig
  structure KS : KGRAM_STATS

  (* Function for selecting a value from a histogram.
     [choose_random f hist] returns a value from the histogram hist
     corresponding to the cumulative distribution at f, where
     f is a (random) number from 0 to 1. *)
   val choose_random : real ->  (KS.token * int) KS.Seq.seq -> KS.token

  (* generate_sentence corpus n seed
     Generates a sentence consisting of n words (tokens) of babble.
     The words are output as a string with spaces between each word
     and ending with a period.
     Each word should be selected on a random basis weighted by its
     likelyhood to follow the previous k tokens and using the
     random seed.   See the description in the text. *)
  val generate_sentence : KS.kgramstats -> int -> int -> string

  (* generate_document corpus n seed
     Generates n sentences (in parallel) with random lengths between
     5 and 10.   Each should be generated with a different seed
     based on the input "seed" (e.g. seed+1, seed+2, ...).
     Sentences should be appended together into a string *)
  val generate_document : KS.kgramstats -> int -> int -> string
end
```

Task 4.1 (10%). Implement `choose_random`. Note that instead of creating a random number itself, the function is passed a random real value from 0 to 1. If this value is not in this range, you can raise an exception.

As an example, consider a histogram: `h = < ("a", 3), ("b", 5), ("c", 2) >`. The cumulative distribution function table is:

a	0.3
b	0.8
c	1.0

Then `choose_random 0.2 h` would return "a", and `choose_random 0.75` should return "b" and `choose_random 0.95` should return "c". That is, it should return the key with least value above f.

Your solution should have linear work in the length of the histogram.

Task 4.2 (3%). Describe how would you implement `choose_random` in linear work and $O(\log n)$ span. You don't need to code it.

Task 4.3 (2%). Describe how you could implement `choose_random` in $O(\log n)$ work and span if you could preprocess the histogram.

Task 4.4 (20%). Implement `generate_sentence` and `generate_document`. We supply a structure `Random210` in the library to generate random numbers. You should use it to generate a sequence of n random numbers from a seed `rseed` as follows:

```
val seed = (Random210.fromInt rseed)
val randomVals = Seq.tabulate (fn i => Random210.randomReal seed i) n
```

Here is how to generate each word in a sentence sequentially with the appropriate probability. Assume you have already generated l words. Let `prefix` be the last (most recent) $\min(l,k)$ words. If this has appeared in the corpus then you should select one of the words that appear after it using `choose_random` (i.e. the selection should be weighted by the number of times each word appears). If `prefix` has not occurred in the input corpus, then try with a prefix that is one shorter (i.e. the previous $\min(l,k)-1$ words). Repeat until you find the kgram in the corpus.

As an example, assume that you have generated words *"dog barked at the cat in the chimney"*, and that you have computed K-gram statistics with $K = 3$. Now to generate the next word, you should first check for any words that followed *"in the chimney"*. If there are none, you should check for any words that followed *"the chimney"*. If that yields no results, then use *"chimney"*. And as a last resort, use an empty-prefix, i.e choose the next word from all words in the corpus (weighted by their frequency).

4.1 Help for testing your code

In the file `babble-test.sml` you will find code for reading an input file (corpus) and for generating an output file of 50 sentences of "babble" for each of three sample files: `kennedy.txt`, `shakespeare.txt`, and `mobydick.txt`.

`tag` — Version 1.0 ***due:*** *Mon, Oct 1st @ 11:59pm*

Disclaimer:
We will not grade non-compiling code.

1 Introduction

In this assignment, you will implement an interface for finding shortest paths in unweighted graphs. Shortest paths are used all over the place, from finding the best route to the nearest Starbucks to numerous less-obvious applications. You will then apply your solution to the thesaurus problem, which is: given any two words, find all of the shortest synonym paths between them in a given thesaurus. Some of these paths are quite unexpected.

1.1 Submission

Submit your solutions by placing your solution files in your handin directory, located at

`/afs/andrew.cmu.edu/course/15/210/handin/<yourandrewid>/assn4/`

Name the files exactly as specified below. You can run the check script located at

`/afs/andrew.cmu.edu/course/15/210/bin/check/04/check.pl`

to make sure that you've handed in appropriate files. Do not submit any other files.

Your written answers must be in a file called `hw04.pdf` and must be typeset. You do not have to use LaTeX, but if you do, we have provided the file `defs.tex` with some macros you may find helpful. This file should be included at the top of your `.tex` file with the command `\input{<path>/defs.tex}`.

For the programming part, the only files you're handing in are

```
AllShortestPaths.sml
AllShortestPathsTest.sml
ThesaurusASP.sml
ThesaurusASPTest.sml
```

These files must contain all the code that you want to have graded for this assignment and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.2 Style

As always, you will be expected to write readable and concise code. If in doubt, you should consult the style guide at `http://www.cs.cmu.edu/~15210/resources/style.pdf` or clarify with course staff. In particular:

1. **Code is Art.** Code *can* and *should* be beautiful. Clean and concise code is self-documenting and demonstrates a clear understanding of its purpose.
2. **If the purpose or correctness of a piece of code is not obvious, document it.** Ideally, your comments should convince a reader that your code is correct.
3. **You will be required to write tests for any code that you write.** In general, you should be in the habit of thoroughly testing your code for correctness, even if we do not explicitly tell you to do so.
4. **Use the check script and verify that your submission compiles.** We are not responsible for fixing your code for errors. If your submission fails to compile, you will lose significant credit.

2 Unweighted Shortest Paths

The first part of this assignment is to implement a general-purpose interface for finding shortest paths. This will be applied to the thesaurus path problem in the next section, but could also be applied to other problems. Your interface should work on directed graphs; to represent an undirected graph you can just include an edge in each direction. Your job is to implement the interface given in `ALL_SHORTEST_PATHS.sig`.

Before you begin, carefully read through the specifications for each function in the `ALL_SHORTEST_PATHS` signature. You will need to come up with your own representations for the `graph` and `asp` types. **You must comment your code and explain why you chose the representations that you did.** Reading the cost specifications beforehand should help you make an informed decision. You may assume that you will be working with simple graphs (i.e., there will be no self-loops and no more than one directed edge in each direction between any two vertices).

2.1 Specification

2.1.1 Graph Construction

Task 2.1 (6%). Implement the function

```
makeGraph :  edge seq -> graph
```

which generates a graph based on an input sequence E of directed edges. The number of vertices in the the resulting graph is equal to the number of vertex labels in the edge sequence. For full credit, `makeGraph` must have $O(|E| \log |E|)$ work and $O(\log^2 |E|)$ span.

2.1.2 Graph Analysis

Task 2.2 (4%). Implement the functions

```
numEdges :  graph -> int
numVertices :  graph -> int
```

which return the number of directed edges and the number of unique vertices in the graph, respectively.

Task 2.3 (4%). Implement the function

```
outNeighbors :  graph -> vertex -> vertex seq
```

which returns a sequence V_{out} containing all out neighbors of the input vertex. In other words, given a graph $G = (V, E)$, `outNeighbors` G v contains all w s.t. $(v, w) \in E$. If the input vertex is not in the graph, `outNeighbors` returns an empty sequence. For full credit, `outNeighbors` must have $O(|V_{\text{out}}| + \log |V|)$ work and $O(\log |V|)$ span, where V is the set of vertices in the graph.

2.1.3 All Shortest Paths Preprocessing

Task 2.4 (14%). Implement the function

```
makeASP : graph -> vertex -> asp
```

to generate an `asp` which contains information about all of the shortest paths from the input vertex v to all other reachable vertices. You must define the type `asp` yourself. If v is not in the graph, the resulting `asp` will be empty. Given a graph $G = (V, E)$, `makeASP` G v must have $O(|E| \log |V|)$ work and $O(D \log^2 |V|)$ span, where D is the longest shortest path (i.e., the shortest distance to the vertex that is the farthest from v).

2.1.4 All Shortest Paths Reporting

Task 2.5 (8%). Implement the function

```
report :  asp -> vertex -> vertex seq seq
```

which, given an `asp` for a source vertex u, returns all shortest paths from u to the input vertex v as a sequence of paths (each path is a sequence of vertices). If no such path exists, `report asp` v evaluates to the empty sequence. For full credit, `report` must have $O(|P||L| \log |V|)$ work and span, where V is the set of vertices in the graph, P is the number of shortest paths from u to v, and L is the length of the longest shortest path from u to v.

Task 2.6 (5%). Briefly give a reasonable worst case bound (in Big-O) for $|P|$, that is, the number of shortest paths in the previous section. Give an example graph where this worst case scenario occurs.

Task 2.7 (5%). Test your `ALL_SHORTEST_PATHS` implementation in the file `AllShortestPathsTest.sml`.

3 Thesaurus Paths

Now that you have a working implementation for finding all shortest paths from a vertex in an unweighted graph, you will use it to solve the Thesaurus problem. You will implement `THESAURUS` in the functor `ThesaurusASP` in `ThesaurusASP.sml`. We have provided you with some utility functions to read and parse from input thesaurus files in `ThesaurusUtils.sml`.

3.1 Specification

3.1.1 Thesaurus Construction

Task 3.1 (4%). Implement the function

```
make :  (string * string seq) seq -> thesaurus
```

which generates a thesaurus given an input sequence of pairs `(w,S)` such that each word `w` is paired with its sequence of synonyms `S`. You must define the type `thesaurus` yourself.

3.1.2 Thesaurus Lookup

Task 3.2 (2%). Implement the functions

```
numWords :  thesaurus -> int
synonyms :  thesaurus -> string -> string seq
```

where `numWords` counts the number of distinct words in the thesaurus while `synonyms` returns a sequence containing the synonyms of the input word in the thesaurus. `synonyms` returns an empty sequence if the input word is not in the thesaurus.

3.1.3 Thesaurus All Shortest Paths

Task 3.3 (8%). Implement the function

```
query :  thesaurus -> string -> string -> string seq seq
```

such that `query th w1 w2` returns all shortest path from `w1` to `w2` as a sequence of strings with `w1` first and `w2` last. If no such path exists, `query` returns the empty sequence. **For full credit, your function `query` must be *staged*.** For example:

```
val earthlyConnection = MyThesaurus.query thesaurus "EARTHLY"
```

should generate the `thesaurus` value with cost proportional to `makeASP`, and then

```
val poisonousPaths = earthlyConnection "POISON"
```

should find the paths with cost proportional to `report`.

Invariably, a good number of students each semester fail to stage `query` appropriately. Don't let this happen to you! Staging is not automatic in SML; ask your TA if you are unsure about SML evaluation semantics or how a properly staged function is implemented.

3.2 Testing

Task 3.4 (5%). Test your implementation of `THESAURUS` in the file `ThesaurusASPTest.sml`. We have provided you with `input/thesaurus.txt` to test your implementation of the thesaurus problem. You should use the helper functions in `ThesaurusUtils` to parse input files.

The file is formatted such that each line is a word followed by its synonyms, separated by spaces. You should use the `parseString` helper in the functor `ThesaurusUtils`, which converts thesaurus strings into a sequence of pairs `(w,S)` such that each word `w` is paired with its sequence of synonyms `S`.

As reference, our implementation returned only find one path of length 10 from "CLEAR" to "VAGUE" as well as from "LOGICAL" to "ILLOGICAL". However, there are two length 8 paths from "GOOD" to "BAD"

Don't try "EARTHLY" to "POISON".

`tag` — Version 1.0 ***due:*** *Wed, Oct 17th @ 11:59pm*

Disclaimer:
We will not grade non-compiling code.

1 Introduction

In this assignment, you will implement an interface for finding shortest paths in *weighted* graphs. Your algorithm will be the Swiss army knife of searches, having many features that standard shortest path algorithms don't have. In particular it will support multiple sources and destinations (finding the shortest path from any of the sources to any of the destinations), it will support the widely used A* heuristic and it will support negative weight edges. It will do this all in one concise piece of code.

1.1 Submission

Submit your solutions by placing your solution files in your handin directory, located at

`/afs/andrew.cmu.edu/course/15/210/handin/<yourandrewid>/assn5/`

Name the files exactly as specified below. You can run the check script located at

`/afs/andrew.cmu.edu/course/15/210/bin/check/05/check.pl`

to make sure that you've handed in appropriate files. Do not submit any other files.

Your written answers must be in a file called `hw05.pdf` and must be typeset. You do not have to use LaTeX, but if you do, we have provided the file `defs.tex` with some macros you may find helpful. This file should be included at the top of your `.tex` file with the command `\input{<path>/defs.tex}`.

For the programming part, the only files you're handing in are

```
distance.sml
distanceTest.sml
```

These files must contain all the code that you want to have graded for this assignment and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.2 Style

As always, you will be expected to write readable and concise code. If in doubt, you should consult the style guide at `http://www.cs.cmu.edu/~15210/resources/style.pdf` or clarify with course staff. In particular:

1. **Code is Art.** Code *can* and *should* be beautiful. Clean and concise code is self-documenting and demonstrates a clear understanding of its purpose.
2. **If the purpose or correctness of a piece of code is not obvious, document it.** Ideally, your comments should convince a reader that your code is correct.
3. **You will be required to write tests for any code that you write.** In general, you should be in the habit of thoroughly testing your code for correctness, even if we do not explicitly tell you to do so.
4. **Use the check script and verify that your submission compiles.** We are not responsible for fixing your code for errors. If your submission fails to compile, you will lose significant credit.

2 Paths"R"Us®

In class we covered Dijkstra's algorithm for finding single source shortest paths in a weighted graph $G = (V, E, W)$. As discussed, it only works with graphs with no negative edge weights. Your goal is to extend this algorithm in various ways as described below. Each of these extensions are worth points. We will run tests on each separately, although the interface is fixed (see the next section). We also have various written questions included below. These must be put in your written solution file.

Multiple Sources. The first extension is to allow multiple source vertices $S \subseteq V$. The goal is to find the shortest shortest path length from any of these sources.

Task 2.1 (10%). Implement multiple sources in your code.

Multiple Targets. Your function will take a set of targets $T \subseteq V$ (think of them as destinations) and return the length of the shortest of the shortest paths to these targets. If multiple sources and destinations are given your algorithm should return the shortest shortest path length between any $s \in S$ and any $t \in T$. The purpose of supplying target vertices is to reduce the part of the graph that needs to be searched, especially if a target is close to the source. Your code therefore needs to take advantage of this.

Task 2.2 (10%). Implement multiple targets in your code.

The A* Heuristic. The A* technique is a very effective heuristic to further reduce the portion of a graph that needs to be searched to find a shortest path to a particular target (or targets). The idea is to narrow the search toward the target vertices. The heuristic is widely used in practice.

The A* technique is a variant of Dijkstra's algorithm. Recall that Dijkstra's idea is to maintain a set of visited vertices $X \subseteq V$ and on each step to add a new vertex $w \in V \setminus X$ to X for which

$$\min_{v \in X, (v,w) \in E} (\delta(s, v) + w(v, w))$$

is minimum. He showed that for such a vertex the given path length is minimum.

In A* we are additionally given a heuristic $h(v) : V \to \mathbb{R}_+ \cup \{0\}$ over the vertices. The A* variant then changes each Dijkstra step so it adds a new vertex $w \in V \setminus X$ for which

$$\min_{v \in X, (v,w) \in E} (\delta(s, v) + w(v, w) + h(w))$$

is minimum. Note that the only difference is the added $h(w)$. For this variant to produce correct path lengths the function $h(w)$ needs to have the following property:

consistency property: for every directed edge $(v, w) \in E$, $h(v) - h(w) \leq w(v, w)$.

If the heuristic satisfies the consistency property, then when the A* approach adds any target $t \in T$ to the visited set, it has found the shortest path length to t.

As an example of heuristic that satisfies the consistency property consider edge weights that represent distances between vertices in euclidean space (e.g. the length of road segments). In this case a heuristic that satisfies the consistency property is the euclidean distance from each vertex (i.e. as the bird flies) to a single target. Multiple targets can be handled by simply taking the minimum euclidean distance to any target.

Task 2.3 (5%). Briefly argue why this heuristic satisfies the consistency property.

Task 2.4 (5%). Give a heuristic that causes A* to perform exactly as Dijkstra's algorithm would.

Task 2.5 (10%). Prove in a paragraph or less that the A* heuristic works: i.e. if the consistency property is true, then when A* visits a vertex in $t \in T$ it has found the shortest path to t.

Task 2.6 (20%). Implement the A* heuristic in your code by accepting a user supplied heuristic $h(v)$. You need not verify that the function satisfies the consistency property.

Negative Edge Weights. In class we will go or have gone over the Bellman Ford algorithm for graphs that have negative edge weights. As it turns out, however, Dijkstra's algorithm can be modified so that it handles negative edge weights. But, this requires allowing a vertex to be visited multiple times and in the worst case ends up requiring the same work as the Bellman Ford algorithm. The modified variant of Dijkstra's has an important advantage—if most edges are non-negative it can do significantly less work than Bellman Ford. Your job is to implement this variant. The idea is to select the next vertex to visit based on the exact same approach as Dijkstra (or A*) but now due to the negative edge weights we are no longer guaranteed the distance is indeed the best path. We therefore allow the vertex to be visited again if the path length decreases.

Task 2.7 (5%). Can you still stop early (i.e. as soon as you visit a target) when using negative edge weights? If not, what is the right stopping condition?

Task 2.8 (5%). If there are no negative edge weights, how many vertices will your modified algorithm visit?

Task 2.9 (20%). Implement this variant that allows negative edge weights in your code. Make sure it terminates even for negative weight cycles.

2.1 Specification

You need to implement an algorithm that supports the single source shortest path problem augmented in the ways described. You will get points for correctness of each of the parts implemented correctly. The interface you need to implement is defined in `DISTANCE.sig`. The main function has the following interface:

```
findPath (c : vertex -> real) (G : graph) (S : Set.set) (T : Set.set)
         : ((vertex * real) option * int)
```

The function returns a pair of values consisting of:

1. The target vertex that is found and the distance to that target. If S or T are empty, if no vertex in T is reachable from S, or if there is a negative weight cycle this value needs to be `NONE`.
2. A count of how many times a vertex is "visited". We say a vertex is visited the first time it is added to the visited set X or if it has already been visited and its distance is decreased (can only happen with negative edges weights).

The reason for counting the number of vertices visited is that the count gives you a sense of the efficiency of your algorithm. After all the goal of specifying target vertices and using the A* heuristic is simply to reduce the number of vertices visited. We will reduce points for algorithms that visit too many vertices or take more than $O(\log n)$ work for each edge they "relax".

2.2 Testing

Task 2.10 (10%). We have given you a file with some tests. You need to augment these.

`tag` — Version 1.0.1 ***due:*** *Mon, Oct 29th @ 11:59pm*

Disclaimer:
We will not grade non-compiling code.

1 Introduction

You have recently been hired as a TA for the course Parallel Cats and Data Structures at the prestigious Carnegie Meow-llon University. There is a lot of work that you need to do in order to keep this course afloat. Obviously, most of this work will come in the form of writing and reasoning with graph algorithms. Fortunately, your time in 15-210 has prepared you well for this.

In this assignment you will identify graph bridges, find maximal independent sets, come up with an approximate solution to the exam-scheduling problem, and read lots of bad cat puns. (Strictly speaking, that last one is optional; you won't receive any point deductions for being a sourpuss.)

1.1 Submission

Submit your solutions by placing your solution files in your handin directory, located at

`/afs/andrew.cmu.edu/course/15/210/handin/<yourandrewid>/assn6/`

Name the files exactly as specified below. You can run the check script located at

`/afs/andrew.cmu.edu/course/15/210/bin/check/06/check.pl`

to make sure that you've handed in appropriate files. Do not submit any other files.

Your written answers must be in a file called `hw06.pdf` and must be typeset. You do not have to use LaTeX, but if you do, we have provided the file `defs.tex` with some macros you may find helpful. This file should be included at the top of your `.tex` file with the command `\input{<path>/defs.tex}`.

For the programming part, the only files you're handing in are

`Bridges.sml`
`BridgesTest.sml`
`SequenceMIS.sml`
`SequenceMISTest.sml`
`Coloring.sml`
`ColoringTest.sml`
`Schedule.sml` (*extra credit*)

These files must contain all the code that you want to have graded for this assignment and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.2 Style

As always, you will be expected to write readable and concise code. If in doubt, you should consult the style guide at `http://www.cs.cmu.edu/~15210/resources/style.pdf` or clarify with course staff. In particular:

1. **Code is Art.** Code *can* and *should* be beautiful. Clean and concise code is self-documenting and demonstrates a clear understanding of its purpose.
2. **If the purpose or correctness of a piece of code is not obvious, document it.** Ideally, your comments should convince a reader that your code is correct.
3. **You will be required to write tests for any code that you write.** In general, you should be in the habit of thoroughly testing your code for correctness, even if we do not explicitly tell you to do so.
4. **Use the check script and verify that your submission compiles.** We are not responsible for fixing your code for errors. If your submission fails to compile, you will lose significant credit.

2 Bridge Finding

On your way to lecture, a professor stops you in the hallway. He introduces himself as a cat-strophysicist and asks for help solving a tricky computational problem that has been purr-plexing his research group.

He is writing code for a cat-body simulation (a much cuter variant of the N-body simulation used by normal astrophysicists). In such a simulation it is not uncommon to track upwards of 3072^3 cats across hundreds of time steps. In order to study large-scale feline structure in the midst of this deluge of data, cat-strophysicists will group large clumps of cats together into objects called "halos".

Your professor friend has attempted to identify halos by forming a graph in which pairs of cats separated than some distance less than δ are connected. The forest of graphs is then reduced so that all connected particles can be identified as a single halo (this approach is somewhat confusingly called the Friends of Friends or FoF algorithm). However, he laments, if two separate, massive halos are connected by a thin filament of closely connected cats, FoF will register both those halos as a single entity. Clearly, this is an inescapable flaw in FoF! "Nonsense," you say, "that thin filament of cats is a graph-bridge and can be identifed relatively quickly."

Let $(u, v) \in E$ for some undirected graph $G = (V, E)$ be given. (u, v) is a *bridge* if it is not contained in any cycles (equivalently, if a bridge is removed there will no longer be a path which connects its endpoints). All code for the first two tasks in this section should go in the functor `Bridges` in the file `bridges.sml`.

Task 2.1 (5%). Define the type `ugraph` representing an undirected graph and write the function `makeGraph (edges: (int * int) seq): ugraph` which takes in a sequence representing the edges of an *undirected* graph and returns that same graph under your `ugraph` representation. You may assume that `edges` is somewhat sane, i.e. that `edges` contains no self-loops or duplicated elements and that no node is labeled by an integer larger than $|V| - 1$ or smaller than 0. Your graph should not contain any vertices which are not explicitly in `edges`. `makeGraph` should have $O(|E| \log |V|)$ work and $O(\log^2 |E|)$ span.

You will receive no points for this task if you do not also include a comment explaining why you chose to represent `ugraph` in the way that you did. This comment does not need to be a novel, but it must be sufficiently detailed that a TA can understand your `ugraph` representation from the comment and type signature alone.

You will need to implement `makeGraph` several times throughout this assignment (possibly with different underlying representations). This will be the only time that you receive points for it.

Task 2.2 (20%). Implement the function

```
findBridges (G: ugraph):  (int * int) seq
```

It should take in an undirected graph and return a sequence containing all the edges of G which are bridges (and only those edges). `findBridges` should run in $O(|E|)$ work and span.

Task 2.3 (5%). Write the test functor `BridgesTest` for a structure subscribing to the `BRIDGES` signature in the file `BridgesTest.sml`.

- `BridgesTest` should not run any tests at compile time.
- `BridgesTest.all()` should return true if all tests pass and false otherwise. Your code should not crash just because the argument structure failed a test.
- `BridgesTest` should test a wide range of edge cases.
- `BridgesTest` should be written in such a way that boiler plate code is kept to a bare minimum. Adding test cases should be easy and repeated work should be dealt with elegantly.

It is possible to get full credit for this task even without completing the previous two tasks.

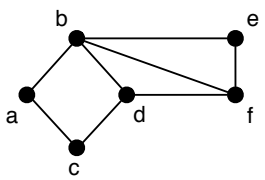
3 Maximal Independent Set

3.1 Problem Definition

In graph theory, an *independent set* is a set of vertices on an undirected graph that do not neighbor one another. More formally, let a graph $G = (V, E)$ be given. We say that a set $I \subseteq V$ is an independent set if and only if $(I \times I) \cap E = \emptyset$.

Unfortunately, the problem of finding the overall largest independent set—known as the Maximum Independent Set problem—is **NP**-hard. Its close cousin Maximal Independent Set, however, admits efficient algorithms and is a useful approximation to the harder problem.

The *Maximal Independent Set* (MIS) problem is: given an undirected graph $G = (V, E)$, find an independent set $I \subseteq V$ such that for all $v \in (V \setminus I)$, $I \cup \{v\}$ is not an independent set. Such a set I is maximal in the sense that we can't add any other vertex and keep it independent, but it easily may not be a maximum—i.e. largest—independent set in the graph.



For example, in the graph above, the set $\{a, d\}$ is an independent set, but not maximal because $\{a, d, e\}$ is also an independent set. On the other hand, the set $\{a, f\}$ is a maximal independent set because there's no vertex that we can add without losing independence. Note that in MIS, we are *not* interested in computing the overall-largest independent set: while maximum independent sets are maximal independent sets, maximal independent set are not necessarily maximum independent sets! Staying with the example above, $\{a, f\}$ is a maximal independent set but not a maximum independent set because $\{a, d, e\}$ is independent and larger.

3.2 Algorithm and Code

You have recently assigned MIS as a homework question for the students in Parallel Cats and Data Structures. You supplied your students with the signature `MIS` in the file `MIS.sig`.

Holding your pre-deadline office hours was like herding cats! Fortunately, the students playing cat-and-mouse with the due date have finally all submitted their code and it is time to begin grading.

Task 3.1 (5%). A student submitted the file `TableMIS.sml` as a solution to this homework. The functions in this file form a valid solution to the MIS problem. As part of the grading process, write down a concise (but careful) description of the algorithm the student uses—a hierarchical enumerated list may help tremendously here.

Task 3.2 (15%). Although the student's table-based implementation is correct, you notice that it runs in a sequence of $O(\log |V|)$ steps in expectation, each of which takes $O((|E| + |V|) \log |V|)$ work and has $O(\log^2 |V|)$ span. As a TA you must one-up your student by writing a more efficient version that uses single-threaded or normal sequences and runs in $O(|E| + |V|)$ work and $O(\log |V|)$ span per step with the same expected total number of steps.

Implement another solution to MIS in a functor `SequenceMIS` in `SequenceMIS.sml` that uses either single-threaded or normal array sequences. Be sure to carefully document the representations of any data structures you use in comments in your SML file.

You must use the same algorithm that the student used in `TableMIS.sml` and obtain the better costs by working with a different representation. You must ensure that if both functions are given the same graph and seed, `SequenceMIS.MIS` will return a seqeunce containing exactly the elements in the set returned by `TableMIS.MIS`.

Task 3.3 (5%). Write a functor `MISTest` in `MISTest.sml` that thoroughly tests its argument structure against the `TableMIS` solution.

Your test battery should include some hard coded small examples, but for full credit, it must also include code to generate larger graphs as stress tests.

You will still need to observe all the restrictions outlined in task 2.2.

3.3 MIS Analysis

The MIS homework was a cat-tastrophe, so you plan to review it during your recitation this week. As part of this review, you want to discuss the cost of your reference solution.

Because you have taken more advanced algorithm courses, you know that there is a beautiful analysis which shows that the code in `SequenceMIS` has $O(|E|+|V|)$ expected total work and $O(\log^2 |V|)$ expected

total span on arbitrary graphs. To whet your students' appetites, you decide to analyze the performance of the algorithm for the special case of graphs where the maximum degree of any vertex is bounded by a constant.

Let $G = (V, E)$ be an input graph. Assume that the maximum degree in G, denoted by Δ, is bounded by a constant positive integer c_Δ.

Task 3.4 (5%). Consider a single step of the MIS algorithm. Calculate the probability p_v that a vertex v is selected as part of the independent set during that step. p_v should be written in terms of $\deg(v)$.

Task 3.5 (10%). Show that the total cost of the MIS algorithm has $O(|V|)$ expected total work and has $O(\log^2 |V|)$ expected total span when implemented as specified.

4 Graph Coloring

The final exam period at Carnegie Meow-llon draws near. You have been tasked with drawing up a schedule for the exams in such a way that no student has two exams scheduled at the same time.

Exams at CM-Mew are structured very similarly to those at CMU: throughout the duration of the exam period, there are n time-slots of uniform length. Any number of exams can take place during a single time slot as long as they do not lead to time conflicts. All exams last exactly as long as a single time slot and must be contained within exactly one time slot.

Formally, given a set of students S, each of which is taking some set of exams X_s, your goal is to create a sequence of time slots T whose elements are sets of exams X_t such that $\forall X_t \in T$, $\forall X_s \in S$, $|X_t \cap X_s| \leq 1$. The *Exam Scheduling* problem is: to find a T such that $|T|$ is as small as possible. This is by no means a trivial problem (in fact it is NP-hard: 3-COLOR can be reduced to it). However during a late-night study session, you realize that the scheduling problem can be reduced to graph-coloring and that graph-coloring can be reduced to MIS. Explaining this revelation will require a bit more formalism.

Let a graph $G = (V, E)$ be given. If C is a set of colors, a *coloring function* for G, $c_G : V \to C$, maps every vertex to a color such that adjacent vertices do not have the same color. That is to say, for all $(u, v) \in E$, $c_G(u) \neq c_G(v)$.

A graph is said to be *k-colorable* if there exists a coloring function for that graph whose range has size at most k. For example, every graph is $|V|$-colorable by mapping each vertex to a unique color. A star graph is 2-colorable by picking one color for the fringe vertices and another for the central vertex. The *Graph Coloring* problem is: to produce a coloring function that uses the smallest possible number of colors.

Task 4.1 (10%). Describe a reduction from the Exam Scheduling problem to the Graph Coloring problem using the tightest cost bounds that you can. State and prove these cost bounds. While this reduction doesn't need to be overly formal, it must be precise enough to be unambiguous. You may include pseudocode if you wish, but just writing a raw SML function that implements this reduction will be insufficient.

$(\Delta + 1)$-coloring

As some would put it, it appears that you have jumped out of the frying pan and into the fire. Although graph coloring is easier to work with, The Graph Coloring problem is also NP-complete. The best we can do at this point is an approximation of the solution. Let Δ denote the maximum degree in a graph. There is an algorithm that reduces finding a $(\Delta + 1)$-coloring to solving MIS.

Task 4.2 (15%). Implement the function

```
graphColor (E : (vertex*vertex) seq):  (vertex*int):  seq,
```

in the structure `MISColoring` in the file `Coloring.sml`. The input sequence contains the graph's *undirected* edges and the output sequence maps every vertex to a color. For your sanity's sake, you will use the SML type `int` to represent colors. Your algorithm should run in $O(\Delta|E|\log|V|)$ work and $(\Delta\log^2|V|)$ span. Be sure to justify why your algorithm meets these bounds in `assn06.pdf`.

It is strongly suggested that you use `TableMIS` as the underlying `MIS` representation in this task.

Task 4.3 (5%). Write a functor `ColoringTest` in `ColoringTest.sml` that thoroughly tests its argument structure. (**Note**: the distributed functor takes in an `MIS` as its argument structure instead of a `COLORING`. You should ignore this and hard-code `MISColoring` in as the tested structure.)

You will still need to observe all the restrictions outlined in task 2.2.

Task 4.4 (5%). **(Extra Credit)** If you ignore the cost of converting the input edge seqeunce into a graph in task 4.2, it is possible to implement this function in $O(\Delta|E|)$ work and $O(\Delta\log^2|V|)$ span. To recieve full credit for this task, your implementation of `graphColor` in `Coloring.sml` should meet these stricter cost bounds, you should justify why it meets these bounds in `assn06.pdf`, and should indicate with a comment in `Coloring.sml` that you are attempting task 4.4.

If you write an $O(\Delta|E|)$ solution, you do not also need to also write an $O(\Delta|E|\log|V|)$ solution.

Task 4.5 (10%). **(Extra Credit)** Implement the function `scheduleExams (S: exam seq seq): exam seq seq` in the structure `Schedule` in the file `schedule.sml`. Each element of `S` represents the collection of exams taken by single student. Each element of the output sequence represents a single time slot, where its elements are the exams scheduled during that time slot. No explicit time bound is given. Document and justify the work and span of your function in `assn06.pdf`. Slower asymptotic times will receive fewer points. You are not required to submit a test structure for this question.

`tag` — Version 1.0 + ε ***due:*** *Mon, Nov 5th @ 11:59pm*

Disclaimer:
We will not grade non-compiling code.

1 Introduction

In this homework, you will implement the parallel Minimum Spanning Tree algorithm known as Borůvka's algorithm, write a grader for it, and solve some written problems on MST and probability.

1.1 Submission

Submit your solutions by placing your solution files in your handin directory, located at

`/afs/andrew.cmu.edu/course/15/210/handin/<yourandrewid>/assn7/`

Name the files exactly as specified below. You can run the check script located at

`/afs/andrew.cmu.edu/course/15/210/bin/check/07/check.pl`

to make sure that you've handed in appropriate files. Do not submit any other files.

Your written answers must be in a file called `hw07.pdf` and must be typeset. You do not have to use LaTeX, but if you do, we have provided the file `defs.tex` with some macros you may find helpful. This file should be included at the top of your `.tex` file with the command `\input{<path>/defs.tex}`.

For the programming part, the only files you're handing in are

```
BoruvkaMST.sml
MSTTest.sml
```

These files must contain all the code that you want to have graded for this assignment and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.2 Style

As always, you will be expected to write readable and concise code. If in doubt, you should consult the style guide at `http://www.cs.cmu.edu/~15210/resources/style.pdf` or clarify with course staff. In particular:

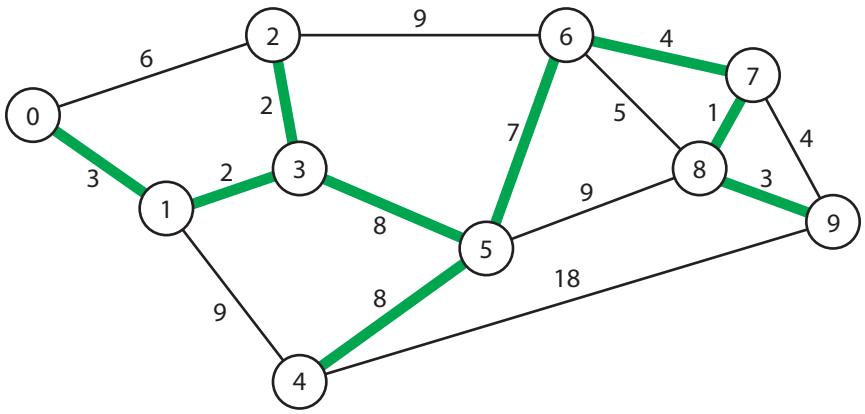
1. **Code is Art.** Code *can* and *should* be beautiful. Clean and concise code is self-documenting and demonstrates a clear understanding of its purpose.
2. **If the purpose or correctness of a piece of code is not obvious, document it.** Ideally, your comments should convince a reader that your code is correct.
3. **You will be required to write tests for any code that you write.** In general, you should be in the habit of thoroughly testing your code for correctness, even if we do not explicitly tell you to do so.
4. **Use the check script and verify that your submission compiles.** We are not responsible for fixing your code for errors. If your submission fails to compile, you will lose significant credit.

2 Borůvka's Algorithm

Recall that the minimum spanning tree (MST) of a connected undirected graph $G = (V, E)$ where each edge e has weight $w : E \to \mathbb{R}^+$ is the spanning tree T that minimizes

$$\sum_{e \in T} w(e)$$

For example, in the graph



the MST shown in green has weight 38, which is minimal.

You should be familiar with Kruskal's and Prim's algorithms for finding minimum spanning trees. However, both algorithms are sequential. For this problem, you will implement the parallel MST algorithm, otherwise known as Borůvka's algorithm.

2.1 Logistics

2.1.1 Representation

Vertices will be labeled from 0 to $|V| - 1$. The input graph is both simple (no self-loops, at most one undirected edge between any two vertices) and connected. We will represent both the input and output graphs as edge sequences, where an edge is defined as

```
type edge = vertex * vertex * weight
```

such that the triple `(u, v, w)` indicates a directed edge from `u` to `v` with edge weight `w`. For the input, since we will be dealing with undirected graphs, for every edge `(u, v, w)` there will be another edge `(v, u, w)` in the sequence. **The MST produced by your solution should only have edges in one direction.** You may find the following function useful:

```
Random210.flip :  Random210.rand -> int -> int seq
```

where `Random210.flip r n` takes a `rand` r and produces a n-length sequence of random 0/1 numbers. You may assume that `Random210.flip r n` has $O(n)$ work and $O(\log n)$ span. Use `Rand.fromInt` and `Rand.next` to generate seed values for each round of your algorithm.

2.2 Specification

Task 2.1 (25%). Implement the function

```
MST : edge seq * int -> edge seq
```

where `MST (E, n)` computes the minimum spanning tree of the graph represented by the input edge sequence E using Borůvka's Algorithm in `BoruvkaMST.sml`. There will be n vertices in the graph, labeled from 0 to $n - 1$. For full credit, your solution must have *expected* $O(m \log n + n)$ work and *expected* $O(\log^k n)$ span for some k, where n is the number of vertices and m is the number of edges.

You may find the pseudocode given in lecture or the component labeling code covered in recitation useful for your implementation. Keep in mind that directly translating pseudocode to SML will not result in the cleanest, nor most efficient solution. For reference, our solution is under 50 lines long and only has one recursive helper. As a hint, recall that the `SEQUENCE` library function

```
inject :  (int * 'a) seq -> 'a seq -> 'a seq
```

takes a sequence of index-value pairs to write into the input sequence. If there are duplicate indices in the sequence, the *last* one is written and the others are ignored. Consider presorting the input sequence of edges `E` from largest to smallest by weight. Then injecting any subsequence of `E` will always give priority to the minimum-weight edges.

2.3 Testing

You will now implement a grader for your `BoruvkaMST` implementation in `MSTTest.sml`. To do this, you must verify that your results satisfy the following conditions:

1. **Spanning**. A simple BFS would work well here.
2. **A Tree**. It suffices to check condition 1 and that you have $|V| - 1$ edges.
3. **Minimum**. Implement a simple MST algorithm and compare the resulting weights.

Task 2.2 (6%). Write the function

```
spanning :  mstresult * int -> bool
```

in `MSTTest.sml` where `spanning (E, n)` evaluates to true if and only if the graph described by the sequence of edges in `E` is spanning on the vertex set $\{0, \ldots, n-1\}$. That is to say, any vertex in $\{0, \ldots, n-1\}$ is reachable from any other vertex. As hinted above, you should use BFS to verify this.

Task 2.3 (4%). Complete the implementation of

```
mstWeight :  mstinput -> int
```

in `MSTTest.sml` where `mstWeight (E, n)` computes the MST weight of a graph with Kruskal's algorithm. Most of the code has already been written for you, as adapted from the `UFLabel` code in recitation 9. You should replace the `raise NotYetImplemented` parts.

Task 2.4 (5%). Complete the test structure `MSTTest` in `MSTTest.sml` by writing the function

```
grader :  mstinput * mstresult -> bool
```

which verifies the above MST conditions using `spanning` and `mstWeight`, as well as the standard function `all : unit -> bool` to run a suite of tests (a combination of hardcoded edge cases and generated graphs) on the argument `MST` structure.

3 Written Problems

For the following problems, typeset your solutions in `hw07.pdf`. You may assume the correctness of any algorithms discussed in lecture.

Task 3.1 (4%). **Second-best is good enough for my MST.** Let $G = (V, E)$ be a simple, connected, undirected graph $G = (V, E)$ with $|E| \geq 2$ and *distinct* edge weights. We know for a fact that the smallest (i.e., least heavy) edge of G must be in the minimum spanning tree (MST) of G. Prove that the 2^{nd} smallest edge of G must also be in the minimum spanning tree of G.

Task 3.2 (8%). **I Prefer Chalk.** There is a very unusual street in your neighborhood. This street forms a perfect circle, and there are $n \geq 3$ houses on this street. As the unusual mayor of this unusual neighborhood, you decide to hold an unusual lottery. Each house is assigned a random number $r \in_R [0, 1]$ (drawn uniformly at random). Any house that receives a larger number than **both** of its two neighbors wins a prize package consisting of a whiteboard marker and two pieces of chalk in celebration of education. What is the expected number of prize packages given? Justify your answer.

Task 3.3 (8%). **It's *Probably* Linear.** Let f be a non-decreasing function satisfying

$$f(n) \leq f\left(X_n\right) + \Theta(n), \qquad \text{where } f(1) = 1.$$

Prove that if for all $n > 1$, $\mathbf{E}\left[X_n\right] = n/3$, then $\overline{f}(n) = \mathbf{E}\left[f(n)\right] \in O(n)$.
Hint: what is $\mathbf{Pr}[X_n \geq 2n/3]$? Use Markov's Inequality, covered in recitation 8.

`tag` — Version 1.0 ***due:*** *Fri, Oct 30th @ 11:59pm*

Disclaimer:
We will not grade non-compiling code.

1 Introduction

In this assignment you will explore applications of dynamic programming in the form of a short programming task and two relatively easy dynamic programming problems. The programming task involves dynamic image resizing using the seam carving technique which was only recently discovered in 2007.

1.1 Submission

Submit your solutions by placing your solution files in your handin directory, located at

`/afs/andrew.cmu.edu/course/15/210/handin/<yourandrewid>/assn9/`

Name the files exactly as specified below. You can run the check script located at

`/afs/andrew.cmu.edu/course/15/210/bin/check/09/check.pl`

to make sure that you've handed in appropriate files. Do not submit any other files.

Your written answers must be in a file called `hw09.pdf` and must be typeset. You do not have to use LaTeX, but if you do, we have provided the file `defs.tex` with some macros you may find helpful. This file should be included at the top of your `.tex` file with the command `\input{<path>/defs.tex}`.

For the programming part, the only files you're handing in are

```
SeamFinder.sml
original.jpg
seamcarved.jpg
```

These files must contain all the code that you want to have graded for this assignment and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

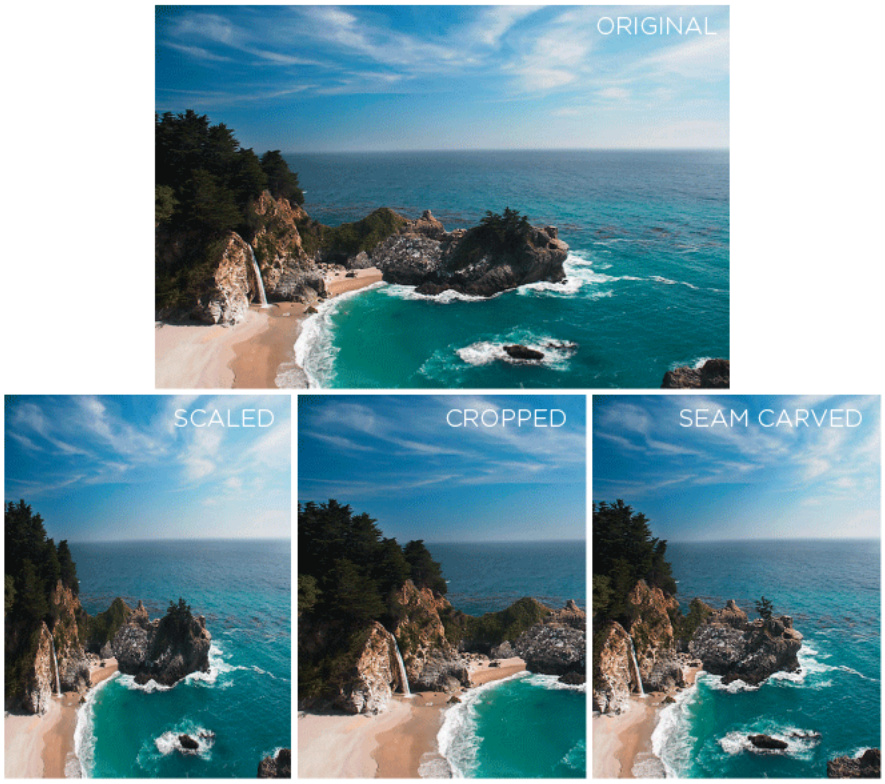
1.2 Style

As always, you will be expected to write readable and concise code. If in doubt, you should consult the style guide at `http://www.cs.cmu.edu/~15210/resources/style.pdf` or clarify with course staff. In particular:

1. **Code is Art.** Code *can* and *should* be beautiful. Clean and concise code is self-documenting and demonstrates a clear understanding of its purpose.
2. **If the purpose or correctness of a piece of code is not obvious, document it.** Ideally, your comments should convince a reader that your code is correct.
3. **You will be required to write tests for any code that you write.** In general, you should be in the habit of thoroughly testing your code for correctness, even if we do not explicitly tell you to do so.
4. **Use the check script and verify that your submission compiles.** We are not responsible for fixing your code for errors. If your submission fails to compile, you will lose significant credit.

2 Seam Carving

Seam Carving is a relatively new technique discovered for "content-aware image resizing" (Avidan & Sheridan, 2007). Traditional image resizing techniques involve either scaling, which results in distortion, or cropping, which results in a very limited field of vision.



The technique involves repeatedly finding 'seams' of least resistance to add or to remove, rather than straight columns of pixels. In this way the salient parts of the image are unaffected. It's a very simple idea, but very powerful. You will work through some simple written problems and then implement the algorithm yourself to use on real images. For simplicity, we will only be dealing with horizontal resizing. To motivate this problem, watch the following SIGGRAPH 2007 presentation video which demonstrates some surprising and almost magical uses of this technique:

```
http://www.youtube.com/watch?v=6NcIJXTlugc
```

2.1 Logistics

2.1.1 Representation

Images will be imported by a Python program that uses image libraries which may not be compatible with your local machine. Make sure to use one of the Andrew machines to run your program. The imported `image` type is defined as

```
type pixel = { r : real, g : real, b : real }
type image = { width : int, height : int, data : pixel seq seq }
```

2.1.2 Gradients

To find a seam of least resistance, we define the following: for any two pixels $p = (r_1, g_1, b_1)$ and $q = (r_2, g_2, b_2)$, the *pixel difference* δ is the sum of the differences squared for each RGB value:

$$\delta(p,q) = (r_2 - r_1)^2 + (g_2 - g_1)^2 + (b_2 - b_1)^2$$

Then, for a given pixel $p_{i,j}$, the *gradient* $g(i,j)$ is defined as the square root of the sum of its difference from the pixel on the right and its difference from the pixel below it.

$$g(i,j) = \sqrt{\delta(p_{i,j}, p_{i,j+1}) + \delta(p_{i,j}, p_{i+1,j})}$$

This leaves the right-most column and bottom row undefined. For an image with height n and width m, we define $g(n-1,j) = 0.0$ for any j, and $g(i,m-1) = \infty$ for any i. Using the gradient as a cost function, we can now develop an algorithm to compute a lowest-cost seam for any image.

2.2 Specification

Task 2.1 (10%). Implement the function

```
generateGradients :  image -> gradient seq seq
```

in `SeamFinder.sml` which converts a raw `image` type into a 2D sequence of gradients $g(i,j)$ as defined above. Your implementation should have work in $O(nm)$ and constant span.

2.2.1 Seam Finding Algorithm

Let's work through a simple example. Suppose we have a 4x4 image with the following table of gradient values already computed (ignoring the right-most column of ∞'s and the bottom row of zeroes):

i ↓ j →	**0**	**1**	**2**	**3**
0	5	3	2	4
1	6	1	7	8
2	2	7	1	2
3	4	7	6	8

Task 2.2 (7%). A valid vertical seam must consist of m pixels if the image has height m, and each pixel must be *adjacent* in the sense that if the seam at row i is at column j, the seam at rows $i-1$ and $i+1$ are limited to columns $j-1$, j, or $j+1$. The cost of a seam is $\sum_{p_{i,j} \in \text{seam}} g(i,j)$. Let $m(i,j)$ be the minimum cost to get from row 0 to $p_{i,j}$. Fill in the table for $m(i,j)$.

i ↓ j →	**0**	**1**	**2**	**3**
0		3		
1	9			
2				
3				

Task 2.3 (3%). What is the lowest cost vertical seam in this image?

Task 2.4 (10%). For the general case, write the mathematical definition of $m(i,j)$ and describe a bottom-up dynamic programming algorithm to find the minimum cost seam. What is the asymptotic work and span of this algorithm? Use a Θ bound.

Task 2.5 (25%). Using this insight, implement the function

```
findSeam :  image -> int seq
```

in `SeamFinder.sml` which finds the lowest-cost seam in the given image, where the seam is represented as an ordered sequence of column indices going from the top row of the image to the bottom row.

2.2.2 Testing

As usual, you will be required to test your code. But it's going to be a little more fun this time! In the handout directory we've given you a few sample images, as well as headshots of the entire course staff to play with (`/hw/09/sml/images/` and `/hw/09/sml/images/staff/`). After completing the above tasks, you should be able to perform the following in the REPL:

```
$ smlnj
Standard ML of New Jersey v110.73 [built: Wed Aug 24 12:35:24 2011]
- CM.make "sources.cm";
[autoloading]
...
[New bindings added.]
val it = true : bool
- SeamCarve.removeSeamsFile ("images/cove.jpg", "images/cove-50.jpg", 50);
val it = () : unit
```

This takes the image `images/cove.jpg`, removes 50 seams from it, and stores the resulting image in `images/cove-50.jpg`. You should compare your results with the given `cove-50.jpg`, `cove-100.jpg`, and `cove-200.jpg` samples. For reference, our solution removes 100 seams from `cove.jpg` in < 8 seconds. Removing more seams from higher resolution images will naturally take longer.

Task 2.6 (5%). Find and submit an image that has an interesting seam-carved result. You should submit the original as `original.jpg` and the altered version as `seamcarved.jpg`. Points will be for awarded for creativity, humor, and self-expression. Dupes will be frowned upon, so be unique!

3 Written Problems

3.1 Subsequence Counting

Suppose you want to count the number of times that a sequence of letters S appears as a subsequence of a string s. The subsequence does not necessarily have to be contiguous. For example, the sequence $\langle k, t, y \rangle$ occurs twice in the string "kitty".

Task 3.1 (2%). How many ways does the string "abbabab" contain the sequence $\langle a, b \rangle$?

Task 3.2 (2%). How many ways does the string "abbabab" contain the sequence $\langle a, b, a \rangle$?

Task 3.3 (8%). Let S_i denote the first i elements of the subsequence S and s_j denote the first j characters of the string s. Consider the subproblem of finding the number of times the prefix S_i appears as a subsequence of the prefix s_j denoted as $C(S_i, s_j)$.

Write a recursive function (in pseudocode or SML) to find $C(S_i, s_j)$ in terms of smaller subsequences of S and smaller substrings of s. Include the base case $C(\emptyset, s_j)$, $0 \leq j < |s|$. That is to say, how many times does the empty sequence appear in the first j characters of s? What is the final answer?

Task 3.4 (5%). Since this is a dynamic programming solution, the recursive call structure is a directed acyclic graph (DAG). Place a bound on the number of vertices (subproblems) in your DAG as well as the longest path in the DAG, in terms of $|S| = n$ and $|s| = m$. What is the work and span of your algorithm? Use Θ notation and explain why.

3.2 Word Breaking

Suppose you are given a sequence of characters $S = \langle c_0, c_1, c_2, \cdots, c_{n-1} \rangle$ that has no spaces or punctuation. You want to determine if you can parse the string into a sequence of words, given a dictionary of valid words. For example, "winterbreakisalmosthere" can be broken into "winter break is almost here". If the sequence has n characters, then are $n-1$ places in which to put a space or not and so there are 2^{n-1} ways to insert spaces. You are to develop a dynamic programming algorithm to break up the sequence into words. You may assume that words are bounded by length k and that you can determine if a word w is in a dictionary in $O(|w|)$ work and span.

Task 3.5 (2%). A greedy algorithm would find a word at the beginning of the sequence, then repeat on the rest of the sequence. Give an example where using a greedy method would not find a valid segmentation of the sequence when one exists.

As is often the case with dynamic programming solutions, instead of finding where to put spaces, you will first consider the decision problem to simplify the problem space: Can you break the character sequence into a sequence of valid words? Later, you will reconstruct where the spaces can go.

Task 3.6 (3%). In the previous problem, we defined for you the subproblems using the notation $C(S_i, s_j)$. For this problem, you will define your own. Clearly state the subproblems and notation that your dynamic programming solution will use.

Task 3.7 (8%). Write a recursive function (in pseudocode or SML) to determine if a sequence of characters can be broken into words based on a dictionary D. Be sure to include all the base cases. Assume a word is itself a sequence of characters, and that for a word w, $D[w]$ returns `true` if w is in the dictionary and `false` otherwise.

Task 3.8 (5%). Place a bound on the number of vertices (subproblems) in your DAG as well as the longest path in the DAG, in terms of $|S| = n$ and the maximum word size k. What is the work and span of your algorithm? Use Θ notation and explain why.

Task 3.9 (5%). Briefly describe how to modify your code to return a list of locations where to insert the spaces. There may be many possible solutions — You need to return just one.

`tag` — Version 1.0 ***due:*** *Fri, Dec 7th @ 11:59pm*

1 Introduction

In this assignment you will explore another dynamic programming application, consider another way to handle deletions when using linear probing for hash tables, and practice with leftist heaps. This assignment is strictly a written homework; there are no programs that you need to hand in. It should be good practice for the final.

1.1 Submission

Submit your solutions by placing your solution files in your handin directory, located at

`/afs/andrew.cmu.edu/course/15/210/handin/<yourandrewid>/assn10/`

Name the files exactly as specified below. You can run the check script located at

`/afs/andrew.cmu.edu/course/15/210/bin/check/10/check.pl`

to make sure that you've handed in appropriate files. Do not submit any other files.

Your written answers must be in a file called `hw10.pdf` and must be typeset. You do not have to use LaTeX, but if you do, we have provided the file `defs.tex` with some macros you may find helpful. This file should be included at the top of your `.tex` file with the command `\input{<path>/defs.tex}`.

Writeups should be short (excessively long or unclear answers will lose points even if they are correct).

2 Independent Sets in Trees

Suppose you have a rooted tree on n vertices, where each vertex has an integer weight. Your goal is to find an independent set A of vertices such that the sum of the weights of the vertices in your set is maximized. Recall that A is an independent set if no two vertices in A are connected by an edge.

You will develop an $O(n)$ work dynamic programming solution to this problem. Let `root` be the root of weighted tree, and let W_v and C_v represent weight and the children of a vertex v, respectively. Note that a tree only has $n-1$ edges, so $\sum_v |C_v|$ is $O(n)$.

Assumptions:

- Vertex weights may be negative.
- There may be duplicate weights.
- You may *not* assume anything about the structure of the tree.
- The empty set has a sum of 0.

Even though you will need to find an actual independent set of vertices of maximum weight, you will first find the maximum achievable weight of independent sets of vertices. Later you will reconstruct an actual set of vertices

Task 2.1 (5%). Clearly define the subproblems you will solve. Which problem provides the overall maximum weight?

Task 2.2 (15%). Use a recursive definition to describe your dynamic programming solution, including all base cases (you can use either mathematical notation or code). You do not need to show memoization.

Task 2.3 (5%). Give a worst case bound on the number of vertices (distinct subproblems) in your DAG and the longest path in the DAG, in terms of the number of vertices n and the depth d of the tree. What is the work and span of your algorithm. Use Θ notation and explain why?

Task 2.4 (10%). Describe how you can find an actual independent set of vertices that has the maximum weight returned by your algorithm above. You need find just one such independent set.

3 Delete with linear probing

As described in lecture, *open addressing* is a hashing technique that does not need any linked lists but instead stores every key directly in the array. Every cell in the array is either empty or contains a key. A probe sequence is used to find an empty slot to insert new elements. Recall that with *linear probing*, to insert a key k, it first checks the position at $h(k)$, and then checks each following location in succession, wrapping to position 0 as necessary, until it finds an empty location. The advantage of linear probing over separate chaining or other probing sequences is that it causes fewer cache misses since typically all probe locations are on the same cache line.

One issue with open addressing, however, is that you cannot simply delete elements from the table, as you then may not be able to find elements that follow the deleted elements in their probe sequence. The solution suggested in lecture is to use a *lazy delete* where deleted elements are replaced with a special `HOLD` value. The problem with lazy deletes is that they persist to fill the table, and result in slower insert and find operations, and more frequent rehashing of the complete table to clear the `HOLD` locations. Lazy deletes are particularly problematic with linear probing, as the keys tend to cluster. Any key that hashes to any position in a cluster (not just collisions), must probe beyond the cluster and adds to the cluster size. Worse yet, this clustering also makes it more likely that the cluster will be lengthen even further. Once the table fills beyond two-thirds full (either with keys and `HOLD` values), linear probing's performance rapidly degrades.

With linear probing, though, it is possible to take another approach for handling deletes: fill a hole created by deleting a value x with another value y in the table. When filling the hole with y, we need to ensure that the value y, and all other values in the table, can still be found with a standard search. Also note that filling the whole with y could create a new hole that might need to be filled. Your task is to write delete using this approach when using linear probing.

Task 3.1 (2%). Beyond needing to fill the new hole, give one other reason why you might not be able to use the value at the position immediately after the hole when using the above method.

Task 3.2 (1%). Which value(s) could safely fill the deleted value so that it is still possible to find it?

Task 3.3 (2%). How would you fill the new hole made by moving y? When can you stop finding new values to fill a hole?

Task 3.4 (15%). Write a function `delete`(T, k) (in pseudocode or SML) that replaces the key k with another value in the table T.

4 Leftist Heaps

As covered in lecture, a *leftist heap* is a data structure for priority queues that holds the keys in a binary tree that maintains the heap property. Unlike binary heaps, however, it not does maintain the complete binary tree property. The goal of the leftist heap is to make the `meld` operation run in $O(\log n)$ work.

Task 4.1 (5%). Given a sequence $S = \langle 1, 2, 3, 4, 5, 6, 7, 8 \rangle$, what is the resulting leftist heap after applying the following code?

```
fun singleton v = Node(1, v, Leaf, Leaf)
Seq.reduce meld Leaf (Seq.map singleton S)
```

where `meld` for leftist heaps is as given in the lecture 27 notes.