Lecture 1 — Overview and Sequencing the Genome

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — August 30, 2011

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, with an emphasis on fundamental concepts that will be applicable across a wide variety of problem domains, and transferable across a reasonably 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. There will be a rough version of the notes posted before each lecture—with a more polished version made available for download later.

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 web site and on the class bboards—please check them on a regular basis. The bboards are `academic.cs.15-210.announce` for announcements from the course staff and `academic.cs.15-210.discuss` for general discussions and clarification questions.

There will be weekly assignments (due at 11:59pm on Mondays), 2 exams, and a final. The first assignment is coming out later today and will be due *at 11:59pm on Tuesday Sept 6, 2011.* (Note the unusual date due to Labor Day)

Since this is the first 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, but it differs from a traditional course in data structures and algorithms in many ways. In particular, this course centers around the following themes:

- defining precise problem and data abstractions
- designing and programming correct and efficient algorithms and data structures for given problems and data abstractions

We will be looking at the following relationship matrix:

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

A *problem* specifies precisely the problem statement and 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)—but `quicksort` is an algorithm that solves the sorting problem and insertion sort is another algorithm. The distinction between problems vs. algorithms is standard in literature.

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.

The crucial differences between this course and a traditional course on data structures and algorithms lie in our focus on *parallelism* and our emphasis on *functional language* implementation. We'll also put heavy emphasis on helping you define precise and concise abstractions.

Due to physical and economical constraints, a typical machine we can buy now 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 hasn't increased much over the past several years. Additionally, graphics processing units (GPUs) are highly parallel platforms with hundreds of cores readily available in commodity machines today. This gives a compelling reason to study parallel algorithms. Here are some example timings from a recent paper:

	Serial	**Parallel**		
		1-core	8-core	32h-core
Sorting 10 million strings	2.9	2.9	.4	.095
Remove duplicates 10M strings	.66	1.0	.14	.038
Min spanning tree 10M edges	1.6	2.5	.42	.14
Breadth first search 10M edges	.82	1.2	.2	.046

32h stands for 32 cores with hyperthreading.

In this table, the sorting algorithm used in sequential is not the same as the algorithm used in parallel. Notice that going from 1 core to 8 core is not quite 8 times faster, as can be expected with overheads associated with parallelization. The magic is going from 8 core to 32 cores, which is more than four times as fast. The reason for this extra speedup is that the 32-core machine uses hyperthreading, which allows for 64 threads and provides the additional speedup. The other algorithms don't show quite the same speedup.

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. Functional languages, however, are great for studying parallel algorithms—they're safe for parallelism because they avoid mutable data. They also generally provide a clear distinction between abstraction and implementations, and are arguably easier to build "interesting" applications quickly.

3 An Example: Sequencing the Genome

Sequencing of a complete human genome represents one of the greatest scientific achievements of the century. It is also one of the most important contributions of algorithms to date. For a brief history, the efforts started a few decades ago with the following major landmarks:

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

Interestingly, all these achievements rely on efficient parallel algorithms. 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. The "primer walking" process cuts the DNA strands into consecutive fragments. But the process is slow because you need the result of one fragment to "build" in the wet lab the molecule needed to find the following fragment (inherently sequential process). 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

   $\boxed{\text{catt}}\ \boxed{\text{ag}}\ \boxed{\text{gagtat}}$

   $\boxed{\text{cat}}\ \boxed{\text{tagg}}\ \boxed{\text{ag}}\ \boxed{\text{tat}}$

   $\boxed{\text{ca}}\ \boxed{\text{tta}}\ \boxed{\text{gga}}\ \boxed{\text{gtat}}$

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, we want to solve the following problem: *Given a set of overlapping genome subsequences, construct the "best" sequence that includes them all.* But what notion of quality are we talking about? 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 finite set of finite, non-empty strings $S \subseteq \Sigma^+$, return a shortest string r that contains every $s \in S$ as a substring of r.

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 a string that contains all the fragments and that is the shortest string. 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"

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

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 string that is a substring of another. That is, we can find a total ordering of the strings.

We will now consider 3 algorithms.

3.2 Algorithm 1: Brute Force

The first algorithm we will look at is a brute force algorithm, because it tries all permutations of the set of strings and for each permutation, we remove the maximum overlap between each adjacent pair of strings. For example, the permutation

$\text{catt}\ \ \underline{\text{tt}}\text{a}\ \ \underline{\text{ta}}\text{gg}\ \ \underline{\text{gg}}\text{a}\ \ \underline{\text{ga}}\text{gtat}$

will give us cattaggagtat after removing the overlaps (the excised parts are underlined). Note that this result happens to be the original string and also the shortest superstring.

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 3.2. *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. $\square$

The problem with this approach is that, although highly parallel, it has to examine a large number of combinations, resulting in a super large work term. 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.

3.3 Algorithm 2: Reducing to Another Problem

We now consider converting the shortest superstring problem to another seemingly unrelated problem: the traveling salesperson[1] (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 3.3 (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.

[1]This is formerly known as the traveling salesman problem.

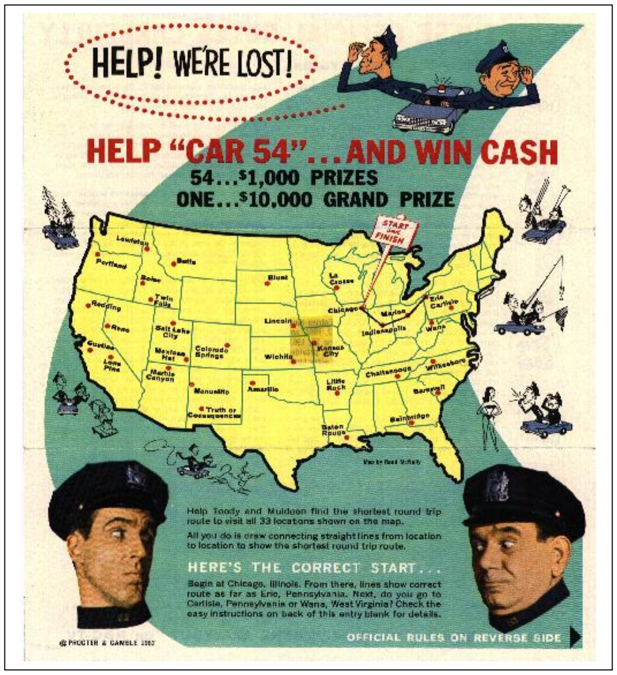

HELP! WE'RE LOST!
HELP "CAR 54"...AND WIN CASH
54...$1,000 PRIZES
ONE...$10,000 GRAND PRIZE
START and FINISH
Help Toody and Muldoon find the shortest round trip route to visit all 33 locations shown on the map.
All you do is draw connecting straight lines from location to location to show the shortest round trip route.
HERE'S THE CORRECT START...
Begin at Chicago, Illinois. From there, lines show correct route as far as Erie, Pennsylvania. Next, do you go to Carlisle, Pennsylvania or Wana, West Virginia? Check the easy instructions on back of this entry blank for details.
OFFICIAL RULES ON REVERSE SIDE
© PROCTER & GAMBLE 1962

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.

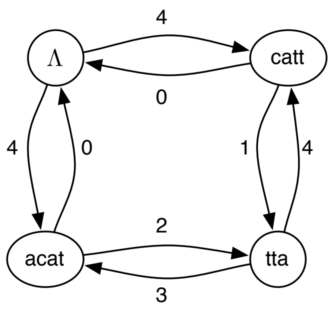
Motivated by the observation that the shortest superstring problem can be solved exactly by trying all permutations, we'll make the TSP problem try all the combinations for us. For this, we will set up a graph so that each valid Hamiltonian cycle corresponds to a permutation. Note that in our case, there 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—indeed, $|$"gga"$| - \texttt{overlap}($"tagg", "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 $\{\mathsf{catt}, \mathsf{acat}, \mathsf{tta}\}$ 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 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

The third algorithm we'll consider is a simple "greedy" algorithm, which finds an "approximate" solution directly. It is not guaranteed to find the shortest superstring but there are theoretical bounds on how close it is to the optimal, and it works very well in practice.

Greedy algorithms are popular because of their simplicity. A greedy algorithm works in steps and at each step it makes a locally optimal choice, in hopes that it will lead to a globally optimal solution, or a solution that is near optimal. Once it makes a choice, it will never reconsider whether to change that choice in a later step. We will cover greedy algorithms in more detail later in this course.

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

Note that the algorithms that we will include in the lecture notes will use pseudocode. The pseudocode will be purely functional and easy to translate in to ML code. Primarily, the difference will be that the pseudocode will freely use standard mathematical notation, such as subscripts, and set notation (e.g. $\{f(x) : x \in S\}$, $\cup$, $|S|$).

The greedy approximation algorithm for the Shortest Superstring Problem is as follows:

```
fun greedyApproxSS($S$) =
  if $|S| = 1$ then $S_0$
  else let
    val $O = \{(\texttt{overlap}(s_i, s_j), s_i, s_j) : s_i \in S, s_j \in S, s_i \neq s_j\}$
    val $(o, s_i, s_j) = maxval <_{\#1} O$
    val $s_k = \texttt{join}(s_i, s_j)$
    val $S' = (\{s_k\} \cup S) \setminus \{s_i, s_j\}$
  in
    greedyApproxSS($S'$)
  end
```

Given a set of strings S, the 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 the 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 1. *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 2. *Prove that algorithm* `greedyApproxSS` *indeed returns a string that is a superstring of all original strings.*

Exercise 3. *Give an example input S for which* `greedyApproxSS` *does not return the shortest superstring.*

Exercise 4. *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 `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, e.g., when we don't require that the overlap be perfect but allow for single errors.

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.

3.5 Cost Analysis

Let n be the size of the input S and N be the total number of characters across all strings in S, i.e.,

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

Note that $n \leq N$. 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 $\texttt{overlap}(s_1, s_2)$ and $\texttt{join}(s_1, s_2)$ 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 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 $\{(\texttt{overlap}\ (s_i, s_j), s_i, s_j) : s_i \in S, s_j \in S, s_i \neq s_j\}$).

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

We have

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

and since all pairs can be done in parallel,

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

The `maxval` can be computed in $O(N^2)$ work and $O(\log N)$ 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 $O(N^2)$ work and $O(\log N)$ 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 $O(nN^2)$ work and $O(n \log N)$ span, which is highly parallel.

Exercise 5. *Come up with a more efficient way of implementing the greedy method.*

4 What did we learn in this lecture?

- Defining a problem precisely is important.
- One problem can have many algorithmic solutions.
- Depending on the input, we can pick the algorithm that works the best for our needs.
- Surprising mappings between problems can be use to reduce one problem to another.
- The ”greedy method” is an important tool in your toolbox
- Many algorithms are naturally parallel but also have sequential dependencies

Lecture 2 — Algorithmic Techniques and Cost Models

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — September 1, 2011

1 Algorithmic Techniques

In this class we are going to cover many algorithmic techniques/approaches for solving problems. 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 the next lecture we will discuss divide-and-conquer and in previous classes I'm sure you have seen other techniques. To give a preview of what else we will be covering in this course, here is a list of techniques we will cover. All these techniques are useful for both sequential and parallel algorithms, however some, such as divide-and-conquer, play a even larger role in parallel algorithms.

Brute Force: The brute force approach typically involves trying all possibilities. In 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 the stock market problem on the first homework you need only try all pairs of elements from the input sequence. There are only $O(n^2)$ such pairs. However, even $O(n^2)$ is not good, since as you will work out in the assignment there are solutions 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.

Reducing to another problem: Sometimes the easiest thing to do is just reduce the problem to another problem for which known algorithms exist. In the case of the SS problem we reduced it to what would seem superficially to be a very different problem, the Traveling Salesperson 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.

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 $n_1, n_2, \dots n_k$, 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.

Collection Types: Some techniques make heavy use of the operations on abstract data types representing collections of values. 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. This brings us to our next topic:

2 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 we will not be covering it here but would be happy to review it in recitation.

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.

2.1 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 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.

2.2 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

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

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 $e_1 \mid\mid e_2$ to mean that the two expressions run in parallel. Both constructs return the pair of results of the two expressions. For example $\texttt{fib}(6) \mid\mid \texttt{fib}(7)$ would return the pair $(8, 13)$. In addition to the $\mid\mid$ 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.

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

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

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. $, \ldots$. 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.

As there is no || 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 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 2.1. 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

$$T_p \quad < \quad \frac{W}{p} + S \qquad (1)$$

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 (perfect speedup would be W/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.

Lecture 3 — More Divide-and-Conquer and Costs

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — September 6, 2011

1 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 class, but this 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.

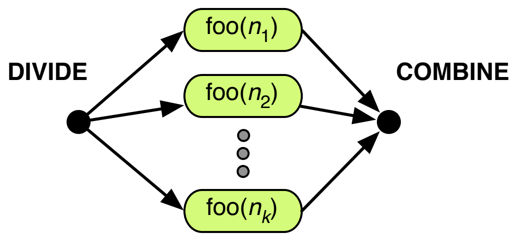
In most divide-and-conquer algorithms you have encountered so far, the subproblems are occurrences of the problem you are solving (e.g. recursive sorting in merge sort). This is not always the case. Often, you'll need more information from the subproblems to properly combine results of the subproblems. In this case, you'll need to strengthen the problem, much in the same way that you strengthen an inductive hypothesis when doing an inductive proof. We will go through some examples in this class where problem strengthening is necessary. But you have seen some such examples already from Recitation 1 and your Homework 1.

The structure of a divide-and-conquer algorithm follows the structure of a proof by (strong) induction. This makes it easy to show correctness and also to figure out cost bounds. The general structure 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:** First, the algorithm divides the current instance I into parts, commonly referred to as *subproblems*, each smaller than the original problem. Then, it 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, and based on this assumption, it combines the answers to produce an answer for the original instance I.

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 as the following simple recurrences: If the 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. Furthermore, it cannot combine the results from these subproblems to generate the ultimate answer *until* the recursive calls on the subproblems are complete. This forms a chain of sequential dependencies, explaining why we add their span together. The parallel execution takes place among the recursive calls since we assume that the subproblems can be solved independently—this is why we take the $\max$ over the subproblems' span.

Applying this formula 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—and learn how to derive a closed-form for them.

2 Example I: Maximum Contiguous Subsequence Sum Problem

The first example we're going to look at in this lecture is the maximum contiguous subsequence sum (MCSS) problem. This can be defined as follows.

Definition 2.1 (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

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

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

2.1 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 combinations of subsequences and for each one of them, it computes the sum and takes the maximum. Note that every subsequence of s can be represented by a starting position i and an ending position j. We will use the shorthand $s_{i..j}$ to denote the subsequence $\langle s_i, s_{i+1}, \ldots, s_j \rangle$.

For each subsequence $i..j$, we can compute its sum by applying a plus `reduce`. This does $O(j-i)$ work and $O(\log(j-i))$ span. Furthermore, all the subsequences can be examined independently in parallel (using, e.g., `tabulate`). This leads the following bounds:

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

Note that these bounds didn't include the cost of the max reduce taken over all these sequences. This max reduce has $O(n^2)$ work and $O(\log n)$ span[1]; therefore, the cost of max reduce is subsumed by the other costs analyzed above. Overall, this is a $O(n^3)$-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.

Exercise 1. *Can you improve the work of the naïve algorithm to $O(n^2)$?*

2.2 Algorithm 2: Divide And Conquer — Version 1.0

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

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

$$
\begin{gathered}
\langle \text{——}\ L\ \text{——} \parallel \text{——}\ R\ \text{——} \rangle\\
\Downarrow\\
L = \underbrace{\langle\quad \cdots \quad\rangle}_{\text{mcss}=56} \qquad R = \underbrace{\langle\quad \ldots \quad\rangle}_{\text{mcss}=17}
\end{gathered}
$$

How would we use this to create a divide-and-conquer algorithm? So far, we have suggested a simple way to split the input sequence, so now the question to answer is, what are the possibilities for how the maximum subsequence sum is formed? There are 3 possibilities: (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 split point. The first two cases are easy. The more interesting case is when the largest sum goes between the two subproblems.

What information do we need to keep track of to handle this case? If we take the largest sum of a suffix on the left and the largest sum of a prefix on the right, then we will get the largest sum that goes between the two. Then we just take a max over the three possibilities, in left (56), in right (17), or between.

We will show next week 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 splitting takes $O(\log n)$ work and span (as you have seen in 15-150). This yields the following recurrences

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

where k is a positive constant.

[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 $O(\log n)$

Solving these recurrences: We'll now focus on solving these recurrences, resorting to the tree method, which you have seen in 15-122 and 15-251.

First, let's think through what it means when we write $O(f(n))$ in an expression (e.g., when we write $2W(n/2) + O(n)$ 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 such that for all $n \geq 1$, and k_2 such that*

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

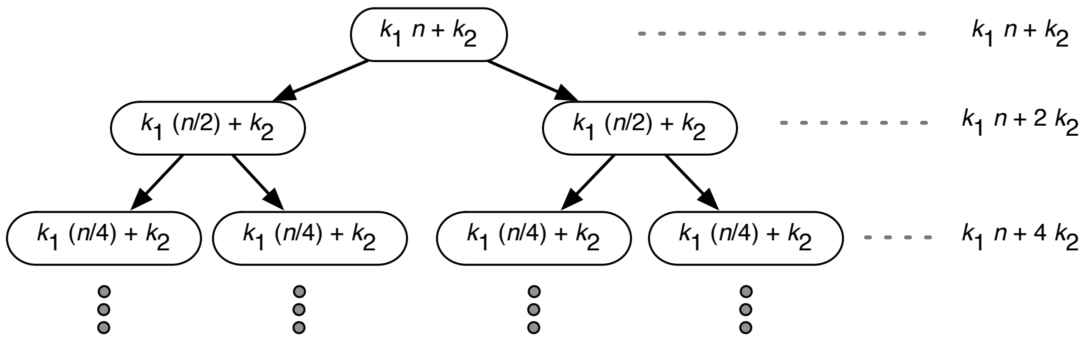
where, for example, we can take $k_1 = c$ and $k_2 = \sum_{i=1}^{N_0} g(i)$, assuming f and g are non-negative functions.

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 go about solving this recurrence using 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 work, 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:



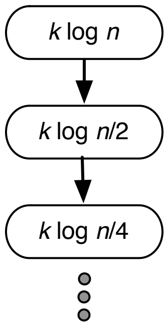
It can be seen that level i (the root is level $i = 0$) contains 2^i nodes, each costing at most $k_1(n/2^i) + k_2$. Thus, the total cost in level i is at most

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

How many levels are there in this tree? Since we keep halving the input size, the number of levels is bounded by $1 + \log n$. Hence, we have

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

As for span, by unfolding the recurrence, we have the following recursion tree:



We conclude that the span is

$$S(n) \;=\; \sum_{i=0}^{\log n} k\log(n/2^i) \;=\; \sum_{i=0}^{\log n} k(\log n - i) \;\in\; O(\log^2 n).$$

Guess and verify by induction: Alternatively, we can also arrive at the same answer by mathematical induction. If you want to go via this route, 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 to avoid common mistakes:

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 2.2. *If $W(n) \le 2W(n/2) + k\cdot n$ for $n > 1$ and $W(1) \le k$ for $n \le 1$, then for some constants κ_1 and κ_2,*

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

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

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

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

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

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?

2.3 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 ML code:

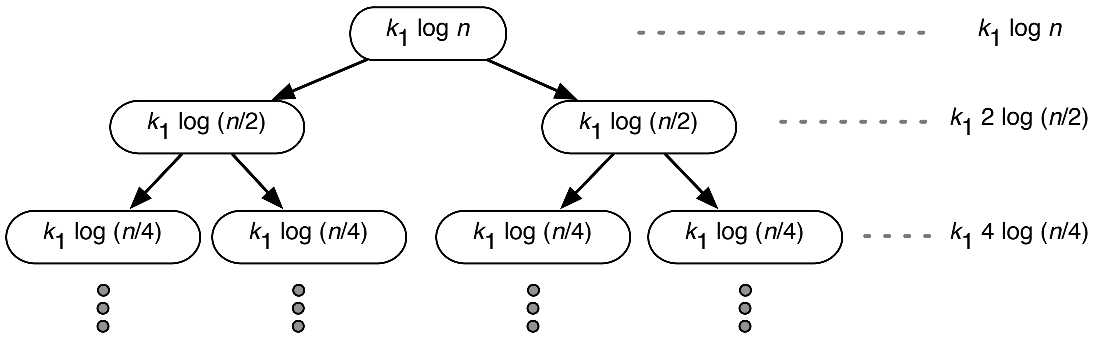
```
fun MCSS (A) =
let
   fun MCSS' (A) =
     case showt(A) of
        EMPTY => {mcss=0, prefix=0, suffix=0, total=0}
      | ELT(v) =>
         {mcss=Int.max(v,0), prefix=Int.max(v,0), suffix=Int.max(v,0), total=v}
      | NODE(A1,A2) =>
        let
           val (B1, B2) = (MCSS'(A1), MCSS'(A2))
           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
```

```
    val {mcss = B, ...} = MCSS′(A)
in
  B
end;
```

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 the following expression:

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

It is not so obvious what this sum equals. 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 2.3. *If $W(n) \leq 2W(n/2) + k \cdot \log n$ for $n > 1$ and $W(1) \leq k$ for $n \leq 1$, then for some constants κ_1 and κ_2,*

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

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

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

where the final step uses the fact that $(k \log n - \kappa_2 \log n + 2(\kappa_2 - \kappa_3)) \leq 0$ 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 mulitply 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

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

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.

3 Example II: The Euclidean Traveling Salesperson Problem

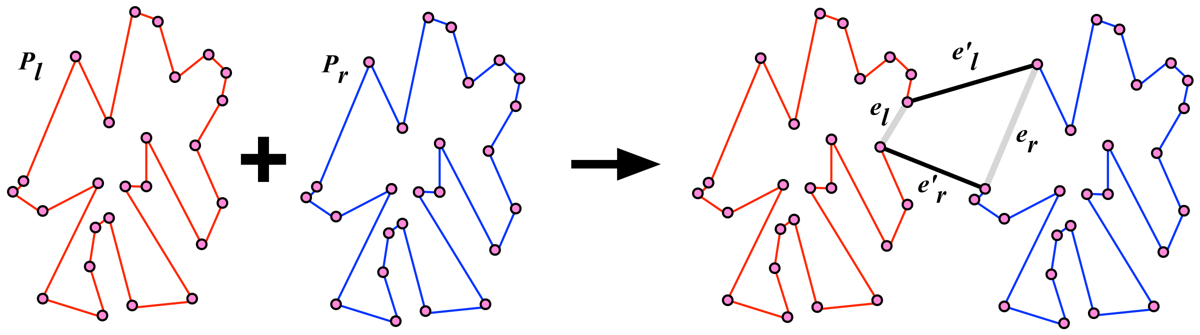
We'll now turn to another example of divide and conquer. In this example, we will apply it to devise a heuristic method for an **NP**-hard problem. The problem we're concerned with is a variant of the traveling salesperson problem (TSP) from Lecture 1. This variant is known as the Euclidean traveling 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 3.1 (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.

As with the TSP, it is **NP**-hard, but this problem is easier[2] 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.

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.

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 each from one side and determine which one minimizes the increase in 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

```
eTSP(P) =
  case (|P|)
    of 0, 1 => raise TooSmall
    | 2 => {(P[0],P[1]),(P[1],P[0])}
    | n => let
      ($P_\ell$,$P_r$) = splitLongestDim(P)
      (L,R) = eTSP($P_\ell$) $\|$ eTSP($P_r$)
      (e,e') = minVal {swapCost(e,e'),(e,e')) : e $\in$ $P_\ell$, e' $\in$ $P_r$}
    in
      swapEdges(append(L,R),e,e')
    end
```

You can find the full code online. Now let's analyze the cost of this algorithm in terms of work and

[2]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.

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

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

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

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

where in the final step, we argued that

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

□

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

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

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 4 — Data Abstraction and Sequences

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — September 8, 2011

1 Abstract Data Types 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 you should have seen before (in 15-122) is a priority queue. Lets consider a slight extension where in addition to insert, and deleteMin, we will add a function that joins two heaps into a single heap. For historical reasons, we will call such a join a meld, and the ADT a "meldable priority queue".

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:

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

Note that `deleteMin` returns the special element $\bot$ when empty. When translated to SML this corresponds to a signature of the form:

```
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 1.1.

In general SML signatures for ADTs will look like:

```
  sig
   struct S1 : ADT1
   ...
   type t
   helper types
   val v1 : ... t ...
   val v2 : ... t ...
   ...
 end
```

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	or I2	or 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 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: the abstract data type (specifying the interface), the cost specification (specifying costs), and the data structure (specifying the implementation).

2 Sequences

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. However in this course we will mostly be using the array implementation. In the lecture we will not go through the full interface, it is available in the documentation, but here are some of functions you should be familiar with:

```
  nth, tabulate, map, reduce, filter, take, drop, showt,
```

and here are some we will discuss in the next couple lectures.

`scan, inject, collect, tokens, fields`

2.1 Scan Operation

We mentioned the scan function during the last lecture and you covered it in recitation. It has the interface:

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

When the function f is associative, i.e., $f(f(x,y),z) = f(x,f(y,z))$, 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 associated f, it can be defined as follows:

```
fun $scan\ f\ I\ S =$
  $(\langle\ reduce\ f\ I\ (take(S,i)) : i \in \langle\, 0, \ldots n-1\,\rangle\,\rangle\,,$
    $reduce\ f\ I\ S)$
```

In this code the notation $\langle\ \mathtt{reduce}\ f\ I\ (\mathtt{take}(S,i)) : i \in \langle\, 0, \ldots, n-1\,\rangle\,\rangle$ indicates that for each i in the range from 0 to $n-1$ apply reduce to the first i elements of S. For example,

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

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. You can assume f takes constant work.*

In the next lecture we will discuss how to implement a scan with the following bounds:

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

assuming that the function f takes constant work. For now we will consider why the operation is useful by giving a set of examples. You should have already seen how to use it for parenthesis matching and the stock market problem in recitation.

The MCSS problem : Algorithm 4 (using scan)

Lets consider how we might use scan operations to solve the MCSS problem. Any ideas? What if we do a scan on our input S using addition starting with 0? Lets say it returns X. Now for a position j lets consider all positions $i < j$. To calculate the sum from i (inclusive) to j (exclusive) all we have to consider is $X_j - X_i$. This represents the total sum between the two. So how do we calculate the maximum sum R_j ending at j (exclusive).

Well this is

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

What is the last term? It is just the minimum value of X up to j (exclusive). Now we want to calculate it for all j, so we can use a scan. This gives the following algorithm

```
fun $MCSS(S) =$
let
   val $X = scan\ +\ 0\ S$
   val $M = scan\ \min\ \infty\ X$
in
   $\max \langle X_j - M_j : 0 \le j < |S| \rangle$
end
```

the work of each of the steps (two scans, map and reduce) is $O(n)$ and the span is $O(logn)$. We therefore have a routine that is even better than any of our divide and conquer routines.

Copy Scan

We will go through one more example that will be helpful in your homework. Lets say you are given a sequence of type α `option seq`. For example

$$\langle\, NONE,\ SOME(7),\ NONE,\ NONE,\ SOME(3),\ NONE\,\rangle$$

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

$$\langle\, NONE,\ NONE,\ SOME(7),\ SOME(7),\ SOME(7),\ SOME(3)\,\rangle$$

Anyone see how they could do this with a 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

```
fun $copy(a, b) =$
   case $b$ of
      $SOME(\_) \Rightarrow b$
    | $NONE \Rightarrow a$
```

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.

2.2 Analyzing the Costs of Higher Order Functions

We already covered map where we have:

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

Tabulate is similar. But what about functions such as `reduce` and `scan` when the combining function does not take constant time. For example for an integer sequence consider the following reduction:

$$\texttt{reduce}\ merge_{int}\ \langle\,\rangle\ \langle\,\langle\,a\,\rangle : a \in A\,\rangle$$

What does this return?

Lecture 5 — More on Sequences

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — September 13, 2011

Today, we'll continue our discussion of sequence operations. In particular, we'll take a closer look at `reduce`, how many divide-and-conquer algorithms can be naturally implemented using `reduce`, and how to implement `Seq.scan` and `Seq.fields`.

1 Divide and Conquer with Reduce

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.

```
fun myDandC(S) =
  case showt(S) of
     EMPTY ⇒ [emptyVal]
   | ELT(v) ⇒ [base](v)
   | NODE(L, R) ⇒ let
       val L′ = myDandC(L)
       val R′ = myDandC(R)
     in
       [someMessyCombine](L′, R′)
     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:

$$\textit{reduce}\ \boxed{\texttt{someMessyCombine}}\ \boxed{\texttt{emptyVal}}\ (\textit{map}\ \boxed{\texttt{base}}\ S)$$

Let's take a look at two examples where `reduce` can be used to implement a relatively sophisticated divide-and-conquer algorithm.

Stock Market Using Reduce. The first example is taken from your Homework 1. Given a sequence S of stock values the stock market problem returns the largest increase in price from a low point to a future high point—more formally:

$$\mathtt{stock}(S) = \max_{i=1}^{|S|}\left(\max_{j=i}^{|S|}(S_j - S_i)\right)$$

Recall that the divide-and-conquer solution involved returning three values from each recursive call on a sequence S: the minimum value of S, the maximum value of S, and the desired result $\mathtt{stock}(S)$. We will denote these as $\bot$, $\top$, and δ, respectively. To solve the stock markert problem we can then use the following implementations for `combine`, `base`, and `emptyVal`:

```
fun  $\mathtt{combine}((\bot_L, \top_L, \delta_L), (\bot_R, \top_R, \delta_R)) =$
     $(\min(\bot_L, \bot_R),\ \ \max(\top_L, \top_R),\ \ \max(\max(\delta_L, \delta_R), \top_R - \bot_L))$
fun  $\mathtt{base}(v) = (v, v, 0)$
val  $\mathtt{emptyVal} = (0, 0, 0)$
```

and then solve the problem with:

```
reduce combine emptyVal (map base S)
```

Merge Sort. As you have seen from previous classes, 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 that combines all inputs from the input. It turns out to be possible to merge two sequences S_1 and S_2 in $O(|S_1| + |S_2|)$ work and $O(\log(|S_1| + |S_2|)$ span. We can use our reduction technique to implement merge sort with a `reduce`. On particular, we use

```
val  combine = merge
val  base = singleton
val  emptyVal = empty
```

Stylistic Notes. We have just seen that we could spell out the divide and conquer steps in details 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 the reduce? We believe this is a matter of taste. Clearly, your reduce code will be (a bit) shorter, but the divide-and-conquer code exposes more clearly the inductive structure of the code.

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 partition their input in two parts in the middle. For example, in the Euclidean Traveling Salesperson algorithm we briefly discussed, each split step splits the space along the larger dimension. This requires checking which is the larger dimension, finding the median line along that dimension, and then filtering the points based on that line. A similar construct can be used for the closest-pair problem from Homework 2. Neither of these algorithms fits the pattern.

2 Reduce: Cost Specifications

Thus far, we made the assumption that the combine function has constant cost (i.e., both its work and span are constant), allowing us to state the cost specifications of `reduce` on a sequence of length n simply as $O(n)$ work and $O(\log n)$. While this bound applies readily to problems such as the one for stock market, we cannot apply it to the merge sort algorithm because the combine step, which we run merge, does more than constant work.

As a running example, we'll consider the following algorithm:

```
fun reduce_sort s = reduce merge< (Seq.empty) (map singleton s)
```

where we use $\texttt{merge}_<$ to denote a merge function that uses an (abstract) comparsion operator $<$ and we further assume that if s_1 and s_2 are sequences of lengths n_1 and n_2, then

$$\begin{aligned} W(\texttt{merge}_<(s_1, s_2)) &= O(n_1 + n_2) \\ S(\texttt{merge}_<(s_1, s_2)) &= O(\log(n_1 + n_2)) \end{aligned}$$

What do you think the cost of `reduce_sort` *is?* We want to analyze the cost of `reduce_sort`. But as we begin to analyze the cost, we quickly realize that the `reduce` function is underspecified: we don't know the reduction order and for that reason, we don't know what `merge` is called with. This begs the question: does the reduction order matter? Or is it in fact doesn't matter because any order will be equally good?

To answer this question, let's consider the following reduction order: the function `reduce` divides the input into 1 and $n - 1$ elements, recurse into the two parts, and run a merge. Thus, on input $x = \langle x_1, x_2, \ldots, x_n \rangle$, the sequence of `merge` calls looks like the following:

```
merge(⟨x1⟩, merge(⟨x2⟩, merge(⟨x3⟩, ...)))
```

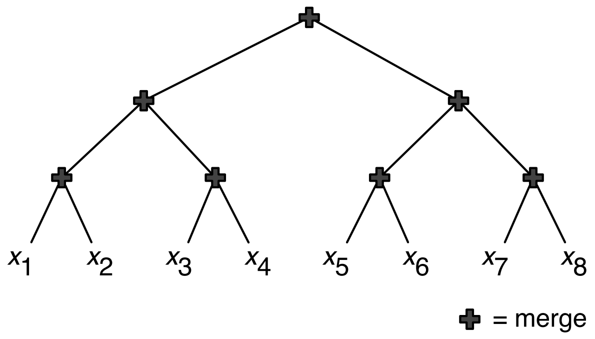
A closer look at this sequence shows that `merge` is always called with a singleton sequence as its left argument but the right argument is of varying sizes between 1 and $n - 1$. The final merge combines 1-element with $(n-1)$-element sequences, the second to last merge combines 1-element with $(n-2)$-element sequences, so on so forth. Therefore, the total work of merge is

$$W(\texttt{reduce\_sort } x) \;\leq\; \sum_{i=1}^{n-1} c \cdot (1 + i) \;\in\; O(n^2)$$

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

As an aside, this reduction order is essentially the order that the `iter` function uses. Furthermore, using this reduction order, the algorithm is effectively working backwards from the rear, "inserting" each element into a sorted suffix where it is kept at the right location to maintain the sorted order. This corresponds roughly to the well-known insertion sort.

Notice that in the reduction order above, the reduction tree was extremely unbalanced. Would the cost 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[1] with 2^k leaves and merge according to the tree structure. As an example, the merge sequence for $|x| = 2^3$ is shown below.



How much does this 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 `reduce_sort` using this reduction order is the familar sum

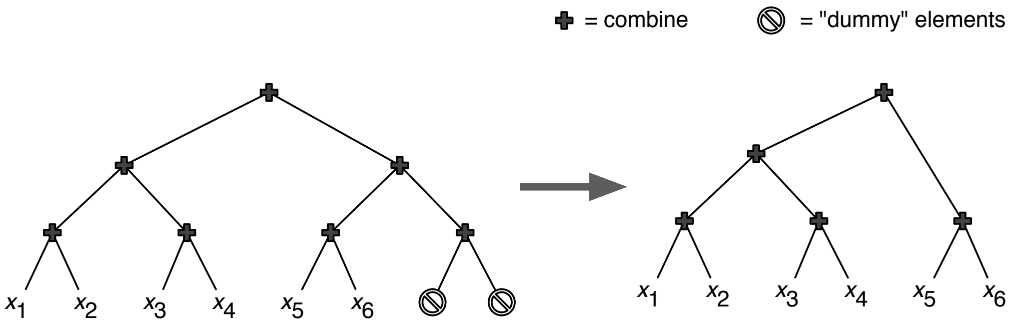
$$\begin{aligned} W(\texttt{reduction\_sort } x) &\leq \sum_{i=0}^{\log n} 2^i \cdot c\left(\frac{n}{2^{i+1}} + \frac{n}{2^{i+1}}\right) \\ &= \sum_{i=0}^{\log n} 2^i \cdot c\left(\frac{n}{2^i}\right) \end{aligned}$$

This sum, as you have seen before, evaluates to $O(n \log n)$. In fact, this algorithm is essentially the merge sort algorithm.

From these two examples, it is clearly important to specify the order of a reduction. These two examples illustrate how the reduction order can lead to drastically different cost. Moreover, there is a second reason to specify the reduction order: to properly deal with combining functions that are non-associative. In this case, the order we perform the reduction determines what result we get; 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 not associative either because of the overflow exception.

For this reason, we define a specific combining tree, which is defined quite carefully in the library documentation for `reduce`. But basically, this tree is the same as if we rounded up the length of the input sequence to the next power of two, and then put a perfectly balanced binary tree over the sequence. Wherever we are missing children in the tree, we don't apply the combining function. An example is shown in the following figure.

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



How would we go about defining the cost of `reduce`? Given a reduction tree, we'll define $\mathcal{R}(\texttt{reduce}\ f\ \mathbb{I}\ S)$ or simply $\mathcal{R}(f,S)$ for brevity,

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

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

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

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 $\max_{f(a,b)} S(f(a,b)$ thus, 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 to prove the following lemma:

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

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

then

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

Applying this lemma to the merge sort example, we have

$$W(\texttt{reduce mergeI}\ \langle\rangle\ \langle\langle a\rangle : a\in A\rangle) = O(|A|\log|A|)$$

3 Contraction and Implementing Scan

Beyond the wonders of what it can do, a surprising fact about `scan` is that it can be accomplished in parallel although on the surface, the computation it carries out appears to be sequential in nature. How can an operation that computes all prefix sums possibly be parallel? At first glance, we might be inclined to believe that any such algorithms will have to keep a cumulative "sum," computing each output value by relying on the "sum" of the all values before it. In this lecture, we'll see a technique that allow us to implement `scan` in parallel.

Let's talk about another algorithmic technique: contraction. This is another common inductive technique in algorithms design. 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. In particular, the contraction technique involves the following steps:

1. Reduce 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

For intuition, we'll look at the following analogy, which might be a stretch but should still get the point across. Imagine designing a new car by building a small and greatly simplified mock up of the car in mind. Then, the mock-up can be used to help build the actual final car, which probably involve a number of refining iterations that add in more and more details. One could even imagine building multiple mock-ups each smaller and simpler than the previous to help build the final car.

The contraction approach is a useful technique in algorithms design, whether it applies to cars or not. For various reasons, it is more common in parallel algorithm than in sequential algorithms, usually because the contraction and expansion 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 by applying it to the scan problem. 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?* Let's look at an example for motivation.

Suppose we're 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

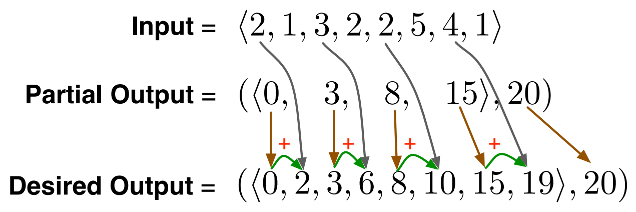
$$(\langle 0, 2, 3, 6, 8, 10, 15, 19\rangle, 20)$$

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.

Thought Experiment II: The crux of this problem is the realization that we can easily generate a sequence consisting of every other element of the final output, together with the final sum—and this is enough information to produce the desired final output with ease. Let's say we are able to somehow generate the sequence

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

Then, the diagram below shows how to produce the final output sequence:


Input = $\langle 2, 1, 3, 2, 2, 5, 4, 1\rangle$
Partial Output = $(\langle 0, 3, 8, 15\rangle, 20)$
+
+
+
+
Desired Output = $(\langle 0, 2, 3, 6, 8, 10, 15, 19\rangle, 20)$

But how do we generate the "partial" output—the sequence with every other element of the desired output? The idea is simple: we pairwise add adjacent elements of the input sequence and recursively run `scan` on it. That is, on input sequence $\langle 2, 1, 3, 2, 2, 5, 4, 1\rangle$, we would be running `scan` on $\langle 3, 5, 7, 5\rangle$, which will generate the desired partial output.

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

```
% implements: the Scan problem on sequences that have a power of 2 lenngth
fun scanPow2 $f$ $i$ $s$ =
  case $|s|$ of
      $0 \Rightarrow (\langle\rangle, i)$
    | $1 \Rightarrow (\langle i\rangle, s[0])$
    | $n \Rightarrow$
        let
          val $s' = \langle f(s[2i], s[2i+1]) : 0 \le i < n/2\rangle$
          val $(r, t) = scanPow2\ f\ i\ s'$
        in
          $(\langle p_i : 0 \le i < n\rangle, t)$,   where $p_i = \begin{cases} r[i/2] & \textbf{if } even(i) \\ f(r[i/2], s[i-1]) & \textbf{otherwise}. \end{cases}$
        end
```

4 How to Parse Text in Parallel 101

One of the most common tasks in your career as a computer scientist is that of parsing text files. For example, you might have some large text file and you want to break it into words or sentences, or you might need to parse a program file, or perhaps you have a web log that lists all accesses to your web pages, one per line, and you want to process it in some way.

The basic idea of parsing is to take a string and break it up into parts; exactly how the string is split up depends on the "grammar" of what're parsing. For this reason, there are many variants of parsing. You probably have come across regular expressions, a class which is reasonably general and powerful enough for most simple text-processing needs. You may have wondered how Standard ML or your favorite compilers are capable of recognizing thousands of lines of code so quickly.

In this lecture, we are only going to talk about two particularly simple, but quite useful, forms of parsing. The first which we call `tokens` is used to break a string into a sequence of tokens separated by one or more whitespaces (or more generally a set of delimiters). A *token* is a maximal nonempty substring of the input string consisting of no white space (delimiter). By maximal, we mean that it cannot be extended on either side without including a white space. There are multiple characters we might regard as a white space, including the space character, a tab, or a carriage return. In general, the user should be able to specify a function that can be applied to each character to determine whether or not it is a space. Specifially, the `tokens` function, as provided by the sequence library, has type

```
val tokens :  (char -> bool) -> string -> string seq
```

where the first argument to `tokens` is a function that checks whether a given character is a white space (delimiter). As an example, we'll parse the string `"this␣is␣␣␣a␣␣short␣string"` (the symbol ␣ denotes a white space),

```
tokens (fn x => (x = #"␣")) "this␣is␣␣␣a␣␣short␣string"
```

which would return the sequence

```
⟨"this", "is", "a", "short", "string"⟩.
```

More formally, we can define the string tokenizer problem as follows:

Definition 4.1 (The String to Token Problem)**.** Given a string (sequence) of characters $S = \Sigma^*$ from some alphabet σ and a function $f : \Sigma \to \{\mathsf{True}, \mathsf{False}\}$, the *string to token* problem is to return a sequence of tokens from S in the order as they appear in S, where a token is a nonempty maximal substring of S such that f evaluates to False for all its characters.

This function appears in the sequence library and is often used to break up text into words. The second useful parsing routine is `fields` and is used to break up text into a sequence of fields based on a delimiter. A field is a maximal possibly empty string consisting of no delimiters. Like in `tokens`, the input string might consist of multiple consecutive delimiters. The key difference here is that a field can be empty so if there are multiple delimiters in a row, each creates a field. For example, we'll parse a line from a comma-separated values (CSV) file.

```
val fields :  (char -> bool) -> string -> string seq
tokens (fn x => (x = #",")) "a,,,line,of,a,csv,,file"
```

which would return

```
⟨"a", "", "", "line", "of", "a", "csv", "", "file"⟩.
```

The `fields` function is useful for separating a file that has fields delimited by certain characters. As mentioned before, a common example is the so-called CSV files that are often used to transfer data between spreadsheet programs.

Traditionally, `tokens` and `fields` have been implemented in a sequential fashion, starting at the front end and processing it one character at a time until reaching the end. During the processing, whenever a whitespace or delimiter is found, the algorithm declares the current token or field finished and starts a new one. Part of the reason for this sequential nature probably has to do with how strings are loaded from external storage (e.g. tape/disk) many years ago. Here, we'll assume that we have random access to the input string's characters, a common assumption now that a program tends to load a big chuck of data into main memory at a time. This makes it possible to implement `fields` and `tokens` in parallel.

In the rest of this lecture, we will discuss a parallel implementation of `fields`. You will think about implementing `tokens` in parallel in your homework.

How do we go about implementing `fields` in parallel? Notice that we can figure out where each field starts by looking at the locations of the delimiters. Further, we know where each field ends—this is necessarily right before the delimiter that starts the next field. Therefore, if there is a delimiter at location i and the next delimiter is at $j \geq i$, we have that the field starting after i contains the substring extracted from locations $(i+1)..(j-1)$, which may be empty. This leads to the following code, in which `delims` contains the starting location of each field. We use the notation $\oplus$ to denote sequence concatenation.

```
fun fields f s = let
  val delims = ⟨0⟩ ⊕ ⟨i + 1 : i ∈ [0, |s|) ∧ f(s[i])⟩ ⊕ ⟨|s| + 1⟩
in
  ⟨s[delims[i], delims[i+1]-1) :  i ∈ [0, |delims|]⟩
end
```

To illustrate the algorithm, let's run it on our familiar example.

```
fields (fn x => (x = #",")) "a,,,line,of,a,csv,,file"

delims = ⟨0, 2, 3, 4, 9, 12, 14, 18, 19, 24⟩

result = ⟨ s[0,1), s[2,2), s[3,3), s[4,8), s[9,11), s[12,13), ... ⟩

result = ⟨"a", "", "", "line", "of","a", ... ⟩.
```

5 SML Code

5.1 Scan

```
functor Scan(Seq : SEQUENCE) =
struct
  open Seq

  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 => if (2*i = n - 1)
                                     then nth s (2*i)
                                     else f(nth s (2*i), nth s (2*i + 1)))
                            (((n-1) div 2)+1)
          val (r, t) = scan f i s'
        in
          (tabulate (fn i => case (i mod 2) of
                                 0 => nth r (i div 2)
                               | _ => f(nth r (i div 2), nth s (i-1)))
                    n, t)
        end
end
```

Lecture 6 — Collect, Sets and Tables

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — September 15, 2011

Today:
- Collect, and example of Google map-reduce
- Sets
- Tables, and example of web searching

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

```
D = <("jack sprat", "15-210"),
     ("jack sprat", "15-213"),
     ("mary contrary", "15-210"),
     ("mary contrary", "15-251"),
     ("mary contrary", "15-213"),
     ("peter piper", "15-150"),
     ("peter piper", "15-251"),
     ... >
```

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, and 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:

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

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 `D` given above by course number we could do the following:

```
fun swap(x,y) = (y,x)
val rosters = collect String.compare (map swap D)
```

This would give something like:

```
rosters = <("15-150", < "peter piper", ... >),
           ("15-210", < "jack sprat", "mary contrary", ... >)
           ("15-213", < "jack sprat", ... >)
           ("15-251", < "mary contrary", "peter piper", ...>)
           ... >
```

The *swap* is used to put the course number in the first position in the tuple. It is often the case that the key needs to be extracted before applying collect.

You might find `collect` useful in your current assignment.

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 is relatively easy and very similar to the `fields` function described at the end of the last class. 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 used in 15-150 which just involved a map then a reduce.[1] 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.But we are stuck with the standard terminology here.

The map-reduce paradigm processes a collection of documents based on *mapF* and *reduceF* functions supplied by the user. The *mapF* 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 *reduceF* 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 and reduce functions are the following:

$$\begin{aligned} \textit{mapF} : &\quad (\textit{document} \to (\textit{key} \times \alpha)\ \textit{seq}) \\ \textit{reduceF} : &\quad (\textit{key} \times (\alpha\ \textit{seq}) \to \beta) \end{aligned}$$

[1]However, there was a question on the 15-150 final about the map-reduce paradigm.

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 *mapF* and *reduceF* functions are sequential functions. Parallelism comes about since the mapF function is mapped over the documents in parallel, and the reduceF function is mapped over the keys with associated values in parallel.

In ML map reduce can be implemented as follows

```
fun mapCollectReduce mapF reduceF S =
  let
    val pairs = flatten (map mapF S)
    val groups = collect String.compare pairs
  in
    map reduceF groups
  end
```

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

```
 flatten < < a, b, c>, < d, e> >
= < a, b, c, d, e >
```

We now consider 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 *mapF* and *reduceF* functions.

```
fun mapF D = map (fn w => (w,1)) (tokens spaceF D)
fun reduceF(w,s) = (w, reduce op+ 0 s)
```

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

```
  val countWords = mapCollectReduce mapF reduceF

  countWords < "this is a document",
               "this is is another document",
               "a last document" >

= < ("a", 2), ("another", 1), ("document" 3), ("is", 3),
    ("last", 1), ("this", 2) >
```

2 Sets

Sets play an important role in mathematics and often needed in the implementation of various algorithms. It is therefore useful to have an abstract data type that supports operations on sets. Indeed most programming languages either support sets directly (e.g., python) or have libraries that support them (e.g., in the C STL library and Java collections framework). Such languages sometimes have more than one implementation of sets. Java, for example, has sets bases on hash tables and balanced trees. We should note, however, that the set interface in different libraries and languages differ in subtle ways. Consequently, 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 powerset of $\mathbb{U}$ (i.e., all subsets of $\mathbb{U}$) along with the following functions:

`empty`	:	$\mathbb{S}$	=	$\emptyset$
`size`(S)	:	$\mathbb{S} \to \mathbb{Z}^*$	=	$\|S\|$
`singleton`(e)	:	$\mathbb{U} \to \mathbb{S}$	=	$\{e\}$
`filter`(f, S)	:	$((\mathbb{U} \to \mathbb{B}) \times \mathbb{S}) \to \mathbb{S}$	=	$\{s \in S \| f(s)\}$
`find`(S, e)	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{B}$	=	$\|\{s \in S \| s = e\}\| = 1$
`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 $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ and $\mathbb{Z}^*$ are the non-negative integers.

This definition is written to be generic and not specific to SML. In the SML Set library we supply, 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` $\to$ `key` $\to$ `set`. 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.

Note that the interface does not contain a `map` function. A `map` function does not make sense in the context of a set, or at least 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 zero. Mapping this over a set would return a bunch of zeros, which would then be collapsed into a set of size one. Therefore such a map would reduce the set of arbitrary size to a singleton, which doesn't match the map paradigm.

In addition to the semantic interface, we need a cost model. The most common efficient ways to implement sets are either using hashing or balanced trees. These have various tradeoffs in cost, but dealing with hash tables in a functional setting where data needs to be persistent is somewhat complicated. Therefore here we will specify a cost model based on a balanced-tree implementation. For now, we won't describe the implementation in detail, but will later in the course. Roughly speaking, however, the idea is to 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. In the table below we assume that the work and span of a comparison on the elements is bounded by C_w and C_s respectively.

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

where $n = \max(|S_1|, |S_2|)$ and $m = \min(|S_1|, |S_2|)$. The work for `intersection`, `union`, and `difference` might seem a bit funky, but the bounds turn out to be both the theoretical upper and lower bounds for any comparison-based implementation of sets. We will get to this later, but for now you should observe that in the special case that the two input lengths are within a constant, the work is simply $O(n)$. This bound corresponds to the cost of merging two approximately equal length sequences, which is effectively what these operations have to do. You should also observe that in the case that one of the sets is a singleton, then the work is $O(\log n)$.

On inspection, the three functions `intersection`, `union`, and `difference` have a certain symmetry with the functions `find`, `insert`, and `delete`, respectively. In particular `intersection` can be viewed as a version of find where we are searching for multiple elements instead of one. Similarly `union` can be viewed as a version of `insert` that inserts multiple elements, and `difference` as a version of `delete` that deletes 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

A table is an abstract data type for storing data associated with keys. They are similar to sets, but along with each element (key) we store some data. 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, or, in set theory, functions. For the purpose of 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.

As with sets, tables are very useful in many applications. Most languages have tables either built in (e.g. dictionaries in python), 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. 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 powerset 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:

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

where S is the powerset of K (i.e., any set of keys), $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ and $\mathbb{Z}^*$ 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 ($\perp$). For this reason, in the Table library, the interface for find is `find` : `'a table` $\to$ `key` $\to$ `'a option`, where `'a` is the type of the values.

Note that 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 `merge` takes a similar function since it also has to consider the case that an element appears in both tables.

Technically a table is a set of pairs and can therefore be written as

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

as long as the keys are distinct. However, to better identify when tables are being used, we will use the notation $k \mapsto v$ in pseudocode to indicate we are using a table, where the key k maps to the value v. For example we will use:

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

The set notation

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

is equivalent to `maps`(f, T) and

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

is equivalent to `filter`(f, T).

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

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

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

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.

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 is analogous to the collect described at the beginning of the class. It takes a sequence S of key-value pairs and generates a table mapping every key that appears in S to all the values that were associated with it in S. It is equivalent to using a sequence collect followed by a `Table.fromSeq`. Alternatively it can be implemented as

```
fun collect(S) =
let
  val S' = $\langle \{k \mapsto \langle v \rangle\} : (k, v) \in S \rangle$
in
  Seq.reduce (Table.merge Seq.append) {} S'
end
```

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

4 Building and searching an index

[This material will be covered in lecture 7]

Here we consider an application of sets and tables. In particular, the goal is to generate an index of the sort that Google or Bing create so that a user can make word queries and find the documents in which those words occur. We will consider 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.

These kinds of searchable indexes predate the web by many years. The Lexis system, for searching law documents, for example, has been around since the early '70s. Now searchable indexes are an integral part of most mailers and many operating systems. By default Google supports queries with *and* and *adjacent to* but with an advanced search you can search with *or* as well as *andnot*.

Let's imagine we 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
  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. So for example we might want to index recent tweets, that might include:

$T = \langle$ ("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", "tidliwinks club was no fun, but more fun than 218"),
$\ldots\rangle$

We can make an index from these tweets:

```
val f = find (makeIndex(T))
```

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"⟩

size(f "fun")
  ⇒ 5
```

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

```
fun makeIndex(docs) =
let
   fun tagWords(name, str) = $\langle\, (w, name) : w \in tokens(str) \,\rangle$
   val Pairs = flatten $\langle\, tagWords(d) : d \in docs \,\rangle$
in
   $\{w \mapsto Set.fromSeq(d) : (w \mapsto d) \in Table.collect(Pairs)\}$
end
```

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

```
fun find(T, v) = 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) = 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 cost is

- work = $O($`size(f "fun")` $+$ `size(f "courses")` $+$ `size(f "classes")`$)$
- span = $O(\log |index|)$

5 SML Code

5.1 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 7 — Introduction to Graphs

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — September 20, 2011

Today:
- Example of using tables and sets for web searching (in the Lecture 6 notes)
- Introduction to Graphs

1 Graphs

Certainly one of the most important abstractions in the study of algorithms is that of a graph, also called a network. Graphs are an abstraction for expressing connections between pairs of items. Graphs can be very important in modeling data, and a large number of problems can be reduced to known graph problems. We already saw this in the case of reducing the shortest superstring problem to the traveling salesperson problem on graphs. 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.

16 Graph Applications. Here we outline just some of the many applications of graphs:

1. *Road networks.* Vertices are intersections and edges are the road segments between them. Such networks are used by google maps, Bing maps and other map programs to find routes between locations. They are also used for studying traffic patterns.
2. *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.
3. *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.
4. *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.
5. *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.

6. *Network 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. *Social network graphs.* 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 tweeted 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.

8. *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).

9. *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.

10. *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.

11. *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.

12. *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.

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 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.

There are many other applications of graphs.

1.1 Formalities

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

- V is a set of *vertices* (or nodes), and
- $E \subseteq V \times V$ 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. Each edge can be written as $e = \{u, v\}$, which is the same as $\{v, u\}$, and self loops are **not** allowed. Undirected graphs represent symmetric relationships.

Note that 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 directed graphs in data structures.

Graphs come with a lot of terminology. Fortunately most of it is intuitive once you 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 if there is an edge between them. For a directed graph we use the terms *in-neighbor* (if the arc points to u) and *out-neighbor* (if the arc points from u).
- The *degree* of a vertex is its number of neighbors and will be denoted as $d_G(v)$. For directed graphs we use *in-degree* ($d_G^-(v)$) and *out-degree* ($d_G^+(v)$) with the presumed meanings.
- 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. $N(v) = \{u \mid \{u, v\} \in E\}$. 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 $N_G^+(V) = \cup_{v \in V} N_G^+(v)$.
- A *path* in a graph is a sequence of adjacent vertices. More formally for a graph $G = (V, E)$, $Paths(G) = \{P \in V^+ \mid 1 \leq i < |P|, (P_i, P_{i+1}) \in E\}$ 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 vertex v is *reachable* from a vertex u in G if there is a path starting at v and ending at u 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 *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 *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.

- For a graph $G = (V, E)$ the (unweighted) *shortest path length* between two vertices is the minimum length of a path between them : $SP_G(u, v) = \min\left\{|P| \mid P \in Paths(G), P_1 = u, P_{|P|} = v\right\}$.

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

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 is typically on algorithms that work well when the graph is sparse.

1.2 Representation

Traditionally when discussing graphs three representations are used. All three assume that vertices are numbered from $1, 2, \ldots, n$ (or $0, 1, \ldots, n-1$). These are:

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

- **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.

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

However, since 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 can also loosen the restriction that vertices need to be labeled from 1 to n and instead allow for any labels. Conceptually, however, the representations we describe are not much different from adjacency lists and edge lists. We are just asking you to think parallel and rid yourself of the idea of lists.

When discussing the implementation of graphs it is important to consider the costs of various basic operations on graphs that are needed in various graph algorithms. In particular common useful operations include: finding the degree of a vertex, finding if an edge is in the graph, mapping or iterating over all edges in the graph, and mapping or iterating over all neighbors of a given vertex. In a dynamically changing graph we might also want to insert and delete edges.

Our first representation directly follows the mathematical definition of a graph and simply uses a set of edges $E \subseteq V \times V$. Such an *edge set* representation can be implemented with the set interface described in lecture 6. The representation is similar to an edge list representation mentioned above, 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 use the balance 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 and 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.

To allow for more efficient access to the neighbors of a vertex our second representation uses a table of sets, which we will refer to as an *adjacency table*. The table simply maps every vertex to the set of its neighbors. Now 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 $O(1)$ 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. We 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.

The costs assuming the tree cost model for sets and tables can be summarized in the following table assuming the function being mapped uses constant work and span:

	edge set		adjacency 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	$O(m)$	$O(\log n)$	$O(\log(n) + d_G(v))$	$O(\log n)$
$d_G(v)$	$O(m)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Lecture 8 — Graph Search and BFS

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — September 22, 2011

Today:
- Graph Search
- Breadth First Search

1 Graph Search

One of the most fundamental tasks on graphs is searching a graph by starting at some 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 all vertices that we can reach and that we do not visit vertices multiple times. This will require recording what vertices we have already visited so we don't visit them again. Graph searching can be use 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 u and v. In the following discussion we use the notation $R_G(v)$ to indicate all the vertices that can be *reached* from v in a graph G (i.e., vertices u for which there is a path from v to u in G).

There are three standard methods for searching graphs: breadth first search (BFS), depth first search (DFS), and priority first search. All these methods visit every vertex that is reachable from a source, but the order in which they visit the vertices can differ. All search methods when starting on a single source vertex generate a rooted *search tree*, either implicitly or explicitly. This tree is a subset of the edges from the original graph. In particular a search always 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 1 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.

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) 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) 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.

```
        o
       / \
S o---o---o---o
   \ /     \ /
    o-------o

        o
       /
S o---o---o---o
   \
    o-------o

        o
       / \
S o   o   o---o
   \       \
    o-------o
```

Figure 1: An undirected graph and two possible search trees.

1.1 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 graph. In the following discussion 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. The idea of *breadth first search* is to start at a *source* vertex v and explore the graph level by level, first visiting all vertices that are the (out) neighbors of v (i.e. have distance 1 from v), then vertices that have distance two from v, 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.

The BFS approach therefore works as follows. As with all the search approaches, the approach needs to keep track of what vertices have already been visited so that it does not visit them more than once. Let's call the set of all visited vertices at the end of step i, X_i. On each step the search also needs to keep the set of new vertices that are exactly distance i from v. We refer to these as the *frontier* vertices $F_i \subset X_i$. 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.*, $N_G(F) \setminus X$. Recall that for a vertex u, $N_G(u)$ are the neighbors of u in the graph G (the out-neighbors for a directed graph) and for a set of vertices U, that $N_G(U) = \cup_{u \in U} N_G(u)$.

Here is pseudocode for a BFS algorithm that returns the set of vertices reachable from a vertex v as well as the furthest distance to any vertex that is reachable.

```
fun $BFS(G, s) =$
let
    % requires: $X = \{u \in V_G \mid \delta_G(s,u) \le i\} \wedge F = \{u \in V_G \mid \delta_G(s,u) = i\}$
    % returns: $(R_G(v), \max\{\delta_G(s,u) : u \in R_G(v)\})$
    fun $BFS'(X, F, i) =$
            if $|F| = 0$ then $(X, i)$
            else $BFS'(X \cup N_G(F),\ \ N_G(F) \setminus X,\ \ i+1)$
in $BFS'(\{s\}, \{s\}, 0)$
end
```

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

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* $X = \{u \in V_G \mid \delta_G(v,u) \le i\} \wedge F = \{u \in V_G \mid \delta_G(v,u) = i\}$

Proof. This can be proved by induction on the step i. For the base case (the initial call) we have $X = F = \{v\}$ and $i = 0$. This is true since only v has distance 0 from v. For the inductive step we note that, if all vertices F at step i have distance i from v, then a neighbor of F must have minimum path of length $d \le i+1$ from v—since we are adding just one more edge to the path. However, if a neighbor of F has a path $d < i+1$ then it must be in X, by the inductive hypothesis so it is not added to F'. Therefore F on step $i+1$ will contain vertices with distance exactly $d = i+1$ from v. Furthermore since the neighbors of F are unioned with X, X at step $i+1$ will contain exactly the vertices with distance $d \le i+1$. □

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

So far we have specified a routine that returns the set of vertices reachable from v 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 v, or the shortest path from v to some vertex u. It is easy to extend BFS for these purposes. For example the following algorithm returns a table mapping every reachable vertex to its shortest path from v.

```
        o
       / \
v o---o---o---o          Original Graph
   \ /     \ /
    o-------o

        2
        o
     1 /  2   3
v o---o---o---o        BFS tree with distances from v
   \
    o-------o
    1       2

        2
        o
     1 /  2   3
v o---o---o   o        anoher BFS tree with distances from v
   \         /
    o-------o
    1       2
```

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

```
fun $BFS(G, s)$ =
let
  fun $BFS'(X, F, i)$ =
    if $|F| = 0$ then $X$
    else let
      val $F' = N_G(F) \setminus \{v : (v, \_) \in X\}$
      val $X' = X \cup \{(v, i+1) : v \in F'\}$
    in $BFS'(X', F', i+1)$end
in $BFS'(\{(v, 0)\}, \{v\}, 0)$
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 neighborhood $F' = N(F)/V$ 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 when generating the neighbors of F to instead of taking the union of the neighbors, to instead for each $v \in F$ generate a table $\{(u, v) : u \in N(v)\}$ and merge all these tables. In our ML library this would look something like

```
1  fun $N(G : \texttt{'a graph},\ F : \texttt{set}) =$
2  let
3    fun $N'(v) = \texttt{Table.map}$ (fn $\Rightarrow w \Rightarrow (v, w)$) $(\texttt{getEdges}\ G\ v)$
4    val $\texttt{nghs} = \texttt{Table.tabulate}\ N'\ F$
5  in
6    $\texttt{Table.reduce merge}\ \{\}\ \texttt{nghs}$
7  end
```

Here `merge = (Table.merge (fn (x,y) => x))`, which merges two tables and when a key appears in both tables, takes the value from the first (left) table. Using either value would work. See the specification of `Table.merge` from Lecture 6.

We can now analyze the cost of $N(G, F)$. For a graph $G = (V, E)$ we assume $n = |V|$ and $m = |E|$. The cost of the function $N'(v)$ in Line 3 is simply $O(|N(v)|)$ work and $O(\log n)$ span. On Line 4 this function is applied across the whole frontier. Its work is therefore $W(F) = O(\sum_{v\in F} |N(v)|)$ and $S(F) = O(\max_{v\in F} \log n) = O(\log n)$. Finally to analyze the reduce in Line 6 we use Lemma 2.1 from lecture 5. In particular merge satisfies the conditions of the Lemma, therefore the work is bound by

$$W(\texttt{reduce merge}\ \{\}\ \texttt{nghs}) = O\left(\log|\texttt{nghs}| \sum_{ngh \in \texttt{nghs}} |ngh|\right) = O\left(\log n \sum_{v\in F} |N(v)|\right).$$

For the span we have $S(\texttt{reduce merge}\ \{\}\ \texttt{nghs}) = O(\log^2 n)$ since each merge has span $O(\log n)$ and the reduction tree is bounded by $\log n$ depth.

Now we note that every vertex will only appear in one frontier, so every (directed) edge will only be counted once. We assume the frontiers are $(F_0, F_1, \ldots, F_d)$ where d is the depth of the shortest path tree. For the overall work done by $N(G, F)$ across the whole algorithm we have:

$$\begin{aligned}
W &= O\left(\sum_{i=[0,\ldots,d]} \log n \sum_{v\in F_i} |N(v)|\right)\\
&= O\left(\log n \sum_{i=[0,\ldots,d]} \sum_{v\in F_i} |N(v)|\right)\\
&= O\left((\log n)|E|\right)\\
&= O(m \log n)
\end{aligned}$$

And for span:

$$\begin{aligned}
S &= O\left(\sum_{i=[0,\ldots,d-1]} \log^2 n\right)\\
&= O(d \log^2 n)
\end{aligned}$$

Similar arguments can be used to bound the cost of the rest of BFS.

2 SML Code

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

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

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

end
```

Lecture 9 — DFS, Weighted Graphs, and Shortest Paths

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — September 27, 2011

Today:
- Depth First Search
- Priority First Search
- Dijkstra's single source shortest path algorithm

1 Depth-First Search (DFS)

Last time, we looked at breadth-first search (BFS), a graph search technique which, as the name suggests, explores a graph an increasing order of hop count from a source node. In this lecture, we'll begin discussing another equally common graph search technique, known as depth-first search (DFS). Instead of exploring 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.

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), or finding the strongly connected components (SCC) of a graph. We will touch on these problems briefly.

For starters, we'll 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). In the algorithm below, X denotes the set of visited vertices—we call them X because they have been "crossed out." Like before, $N_G(v)$ represents the out-neighbors of v in the graph G.

```
fun DFS($G, v$) = let
  fun DFS'($X, v$) =
    if ($v \in X$) then $X$
    else iterate DFS' ($X \cup \{v\}$) ($N_G(v)$)
in DFS'($\{\}, v$) end
```

The line `iterate DFS'` $(X \cup \{v\})$ $(N_G(v))$ deserves more discussion. First, the iterator concept is more or less universal and in broad strokes, `iterate` f `init` A captures the following:

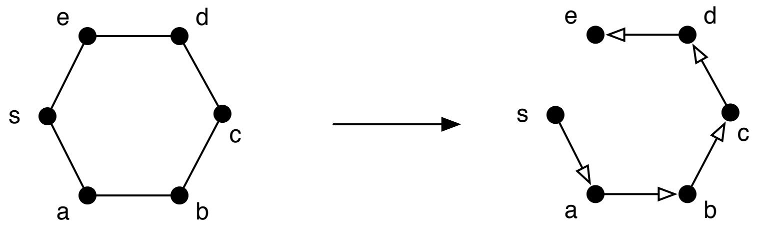
```
$S$ = init
foreach $a \in A$:  $S$ = $f(S, a)$
```

This doesn't specify the order in which the elements of A are considered. All we know is that all of them are going to be considered in some order sequentially. What this means for the DFS algorithm is that when the algorithm visits a vertex v (i.e., `DFS'` (X, v) is called), it picks the first outgoing edge vw_1, through iterate, calls `DFS'` $(X \cup \{v\}, w_1)$ to fully explore the graph reachable through vw_1. We know we have fully explored the graph reachable through vw_1 when the call `DFS'` $(X \cup \{v\}, w_1)$ that we made returns. The algorithm then picks the next edge vw_2, again through iterate, and fully explores the graph reachable from that edge. The algorithm continues in this manner until it has fully explored all out-edges of v. At this point, iterate is complete—and the call `DFS'` (X, v) returns.

As an example, if `DFS'` (X, v) is called on a vertex with $N_G(v) = \{w_1, w_2, w_3\}$ and iterate picks w_1, w_2, and w_3 in this order, then upon called, `DFS'` (X, v) will invoke `DFS'` $(X \cup \{v\}, w_1)$ and after this finishes, it will invoke `DFS'` (X', w_2), and `DFS'` (X'', w_3) in turn. Notice the difference between X, X' and X''—this is because `DFS'` accumulates visited vertices.

Is this parallel? At first look, we might think this approach can be parallelized by searching the outgoing edges in parallel. This would indeed work if the searches initiated never "meet up" (e.g., the graph is a tree, so then what's reachable through each edge would be independent). 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 get a better sense of the situation, we'll consider 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 little example, since the search wraps all the way around, we couldn't know until the end that e would be visited (in fact it is 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 $\mathbf{P}$-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.

1.1 Beefing Up DFS

Thus far, our DFS function is only capable of reporting all vertices reachable from a given starting point. How can we change it so that DFS is a more useful building block? Interestingly, most algorithms based on DFS can be expressed with just two operations associated with each vertex visited: an *enter* operation and an *exit* operation. The *enter* operation is applied when first entering the vertex, and the *exit* operation is applied upon leaving the vertex when all neighbors have been explored.

In other words, depth-first search defines an ordering—the *depth-first ordering*—on the vertices where each vertex is visited twice, and the following code "treads through" the vertices in this order, applying *enter* the first time a particular vertex is visited and *exit* the other time that vertex is seen.

As such, DFS can be expressed as follows.

```
fun DFS $(G, S_0, v)$ = let
  fun DFS' $((X, S), v)$ =
    if $(v \in X)$ then $(X, S)$
    else let
      val $S'$ = enter$(S, v)$
      val $(X', S'')$ = iterate DFS' $(X \cup \{v\}, S')$ $(N_G(v))$
      val $S'''$ = exit$(S'', v)$
    in $(X', S''')$ end
in DFS'$((\{\}, S_0), v)$ end
```

The updated algorithm bears much similarity to our basic DFS algorithm, except now we maintain a state that is effectively treaded through and where state transitions occur by applying the *enter* and *exit* functions in the depth-first order. At the end, DFS returns an ordered pair (X, S), which represents the set of vertices visitd and the final state S.

1.2 Topological Sorting

How can we apply the DFS algorithm to solve a specific problem? We now look at an example that applies this approach to topological sorting (TOPSORT). For this problem, we'll only need the *exit*; thus, the *enter* function simply returns the state as-is.

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]

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

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

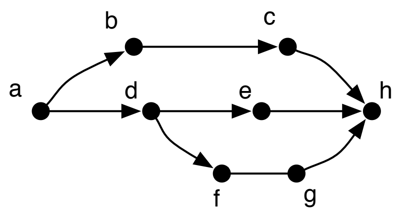
1. reflexivity — $a \leq_p a$,
2. antisymmetry — if $a \leq_p b$ and $b \leq_p a$, then $b = a$, and
3. transitivity — if $a \leq_p b$ and $b \leq_p c$ then $a \leq_p c$.

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 1.1 (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. Let's now go back to DFS. To topologically sort a graph, we create a dummy source vertex o and add an edge from it to all other vertices. We then do run DFS from o. To apply the generic DFS discussed earlier, we need to specify two functions *enter* and *exit* and an initial state. For this, the state maintains a list of visited vertices (initially empty)—and we use the following *enter* and *exit* functions:

```
val $S_0$ = []
fun enter($X$,_) = $X$
fun exit($X, v$) = $v$::$X$
```

We claim that at the end, the ordering in the list returned specifies the total order. 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. Consider any vertex $v \in V$. Suppose we first enter v with an initial list L_v. Now all unvisited vertices reachable from v, denoted by R_v, will be visited before exiting. But then, when exiting, v is placed at the start of the list (i.e., prepended to L_v). Therefore, all vertices reachable from v appear after v (either in L_e or in R_v), as required.

We included a dummy source vertex o to make sure that we actually visit all vertices. This is a useful and common trick in graph algorithms.

2 Priority First Search

Generalizing BFS and DFS, *priority 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. To apply priority first search, we only need to make sure that at every step, we have a priority value for all the unvisited vertices adjacent to the visited vertices. This allows us to pick the best (highest priority) among them. When we visit a vertex, we might update the priorities of the remaining vertices. One could imagine using such a scheme for exploring the web so that the more interesting part can be explored without visiting the whole web. The idea might be to rank the outgoing links based on how interesting the tag on the link appears to be. Then, when choosing what link to visit next, choose the best one. This link might not be from the page you are currently on.

Many famous graph algorithms are instances of priority first search. For example, Dijkstra's algorithm for finding single-source shortest paths (SSSP) from a single source on a weighted graph and Prim's algorithm for finding Minimum Spanning Trees (MST).

3 Shortest Weighted Paths and Dijkstra's Algorithm

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. We'll need a few definitions to describe the problem more formally. Consider a graph (either directed or undirected) graph $G = (V, E)$. A *weighted graph* is a graph $G = (V, E)$ along with a weight function $w: E \to \mathbb{R}$ that associates with every edge a real-valued weight. Thus, the *weight of a path* is the sum of the weights of the edges along that path.

Problem 3.1 (The Single-Source Shortest Path (SSSP) Problem)**.** Given a weighted graph $G = (V, E)$ 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.

We will use $\delta_G(u, v)$ to indicate the weight of the shortest path from u to v in the weighted graph G. Dijkstra's algorithm solves the SSSP problem when all the weights on the edges are non-negative. Dijkstra's is a very important algorithm both because shortest paths have many applications but also because it is a very elegant example of an efficient greedy algorithm that generates optimal solutions on a nontrivial task.

Weighted Graph Representation. There are a number of ways to represent weights in a graph. More generally, we might want to associate any sort of value with the edges (e.g. in Assignment 4). That is, we have a "label" function of type $w: E \to$ `label` where `label` is the type of the label.

The first representation we consider translates directly from viewing edge labels as a function. We keep a table that maps each edge (a pair of vertex identifiers) to its label (or weight). This would have type

(`label` *vertexVertexTable*)

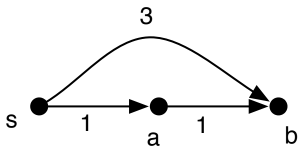
i.e. the keys would be pairs of vertices (hence *vertexVertexTable*), and the values are labels.

Another way to associate values with edges is to use a structure similar to the adjacency set representation for unweighted graphs and try to piggyback labels on top of it. In particular, instead of associating a set of neighbors with each vertex, we can have a table of neighbors that maps each neighbor to its label (or weight). It would have type:

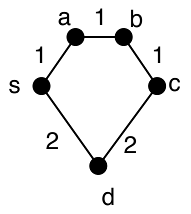
$$(\texttt{label}\ \textit{vertexTable})\textit{vertexTable}.$$

This is the representation we will be using for Dijkstra's algorithm.

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:



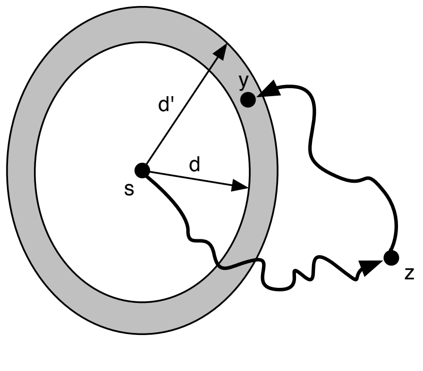
In this example, BFS would visit b then a. This means when we visit b, we assign it an incorrect weight of 3. Since BFS never visit it again, we'll never update it to the correct value. Note that we cannot afford to visit it again as it will be much more expensive. This problem still persists even if the graph is geometric (and satisfies the triangle inequality), as the following example shows:



The crux of Dijkstra's algorithm is the following lemma, which suggests prioritiy values to use and guarantees that such a priority setting will lead to the weighted shortest paths.

Lemma 3.2. *Consider a (directed) weighted graph $G = (V, E)$, $w\colon E \to \mathbb{R}_+ \cup \{0\}$ with no negative edge weights, a source vertex s and an arbitrary distance value $d \in \mathbb{R}_+ \cup \{0\}$. Let $X = \{v \in V\ :\ \delta(s, v) \le d\}$ be the set of vertices that are at most d from s and $d' = \min\{\delta(s, u) : u \in V \setminus X\}$ be the nearest distance strictly greater than d. Then, if $V \setminus X \neq \emptyset$, there must exist a vertex u such that $\delta(s, u) = d'$ and a shortest path to u that only goes through vertices in X.*

Proof. Let $Y = \{v \in V\ :\ \delta(s, v) = d'\}$ be all vertices at distance exactly d'. Note that the set Y is nonempty by definition of d' and since $V \setminus X \neq \emptyset$.



Pick any $y \in Y$. We'll assume for a contradiction that that all shortest paths to y go through some vertex in $Z = V \setminus (X \cup Y)$ (i.e., outside of both X and Y). But for all $z \in Z$, $d(s, z) > d'$. Thus, it must be the case that $d(s, y) \geq d(s, z) > d'$ because all edge weights are non-negative. This is a contradiction. Therefore, there exists a shortest path from s to y that uses only the vertices in $X \cup Y$. Since $s \in X$ and the path ends at $y \in Y$, it must contain an edge $v \in X$ and $u \in Y$. The first such edge has the property that a shortest path to u only uses X's vertices, which proves the lemma. □

This suggests an algorithm that by knowing X, derives d' and one such vertex u. Indeed, X is the set of explored vertices, and we can derive d' and a vertex u attaining it by computing $\min\{d(s, x) + w(xu) : x \in X, u \in N_G(x)\}$. Notice that the vertices we're taking the minimum over is simply $N_G(X)$. The following algorithm uses a priority queue so that finding the minimum can be done fast. Furthermore, the algorithm makes use of a dictionary to store the shortest path distance, allowing for efficient updates and look-ups. We show the algorithm's pseudocode below.

```
fun dijkstra($G, u$) =
  let fun dijkstra'($D, Q$) =
      case (PQ.deleteMin($Q$))
        of (PQ.empty, _) ⇒ $D$
        | (($d, v$), $Q$) ⇒
          if (($v$, _) $\in D$) Dijkstra'($D, Q$)
          else let
            fun relax ($Q$, ($u, w$)) = PQ.insert ($d + w, u$) $Q$
            val $Q'$ = iterate relax $Q$ ($N_G(v)$)
          in dijkstra'($D \cup \{(v, d)\}, Q'$) end

  in dijkstra'({}, PQ.insert(empty,($0.0, u$)))
  end
```

This version of Dijkstra's algorithm differes somewhat from another version that is sometimes used. First, the `relax` function is often implemented as a `decreaseKey` operation. In our algorithm, we simply add in a new value in the priority queue. Although this causes the priority queue to contain more entries, it doesn't affect the asympotic complexity and obviates the need to have the `decreaseKey` operation, which can be tricky to support in many priority queue implementations.

Second, since we keep multiple distances for a vertex, we have to make sure that only the shortest-path distance is registered in our answer. We can show inductively through the lemma we proved already that the first time we see a vertex v (i.e., when `deleteMin` returns that vertex) gives the shortest path to v. Therefore, all subsequence occurences of this particular vertex can be ignored. This is easy to support because we keep the shortest-path distances in a dictionary which has fast lookup.

4 SML code

Here we present the SML code for DFS and Dijkstra.

4.1 DFS with Enter and Exit Functions

```
functor TableDFS(Table : TABLE) =
struct
  open Table
  type vertex = key
  type graph = set table

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

  fun DFS (S0, enter, exit) (G : graph, s : vertex) =
    let
      fun DFS'((X : set, S), v : vertex) =
        if (Set.find X v) then (X, S)
        else
          let
            val S' = enter(S, v)
            val (X',S'') = Set.iter DFS' (Set.insert v X, S') (N(G,v))
            val S''' = exit(S'',v)
          in (X',S''')
          end
    in
      DFS'((Set.empty, S0), s)
    end
end
```

4.2 Topological Sort with DFS

```
functor TableTopSort(Table : TABLE) =
struct
  structure dfs = TableDFS(Table);

  fun topSort(G, v) =
  let
      val S0 = nil
      fun enter(S, v) = S
      fun exit(S, v) = v::S
      val (X, S) = dfs.DFS (S0, enter, exit) (G, v)
  in
      S
  end
end
```

4.3 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 10 — Shortest Paths II (DRAFT)

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — September 29, 2011

Today:
- Continuation of Dijkstra's algorithm (see notes from previous lecture)
- Bellman-Ford's Algorithm, which allows negative weights

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), 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:

```
   a o
    / \
 3 /   \ -2
s o --- o b
     2
```

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 not in the visited set that it needs to have a path through the visited set.

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 unweighted path length (from now on we will refer to path length as the number of edges in the path, and path weight as the sum of the weights on the edges in the path). In particular the idea based on induction is that, given the shortest weighted path of length at most i (i.e. involving at most i edges) from s to all vertices, then we can figure out the shorted weighted path of length at most $i + 1$ to all vertices. It turns out that this idea does pan out, unlike the others. Here is an example:

```
     i  1  i
   a o --- o c
    / \-2   \ 1       Closest distances from s for paths of length 0
 3 /   \ b   \               i indicates infinity
s o --- o --- o d
0    2  i 1   i

     3  1  i
   a o --- o c
    / \-2   \ 1       for paths of length 1
 3 /   \ b   \
s o --- o --- o d
0    2  2 1   i

     0  1  4
   a o --- o c
    / \-2   \ 1       for paths of length 2
 3 /   \ b   \
s o --- o --- o d
0    2  2 1   3

     0  1  1
   a o --- o c
    / \-2   \ 1       for paths of length 3
 3 /   \ b   \
s o --- o --- o d
0    2  2 1   3

     0  1  1
   a o --- o c
    / \-2   \ 1       for paths of length 4
 3 /    \ b    \
s o --- o --- o d
0    2  2 1    2
```

Here is an outline of a proof that this idea works by induction. This proof also leads to an algorithm. We use the convention that a vertex that is not reachable with a path length i has distance infinity (∞) and set the initial distance to all vertices to ∞. For the base case, on step zero no vertices except for the source are reachable with path length 0, and the distance to all such vertices is ∞. The distance to the source is zero. For the inductive case we note that any path of length $i+1$ has to go through a path of length i plus one additional edge. Therefore we can figure out the shortest length $i+1$ path to v by considering all the in-neighbors $u \in N_G^-(v)$ and taking the minimum of $w(u,v) + d(u)$.

Here is the Bellman Ford algorithm based on this idea. The notation $\delta_G^i(s,v)$ indicates the shortest path from s to v in G that uses at most i edges.

```
1   % implements: the SSSP problem
2   fun BellmanFord($G = (V, E), s$) =
3   let
4      % requires: all{$D_v = \delta_G^i(s,v) : v \in V$}
5      fun $BF(D, i)$ =
6         let
7            val $D' = \{v \mapsto \min_{u \in N_G^-(v)}(D_u + w(u,v)) : v \in V\}$
8         in
9            if $(i = |V|)$ then $\perp$
10           else if (all{$D_v = D'_v : v \in V$}) then $D$
11           else $BF(D', i+1)$
12        end
13     val $D = \{v \mapsto$ if $v = s$ then $0$ else $\infty : v \in V\}$
14  in $BF(D, 0)$ end
```

In Line 9 the algorithm returns $\perp$ if there is a negative weight cycle. 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.

We can 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. We use the following cost table. The reduce complexity is assuming the combining function takes constant work.

	Work	*Span*
Array Sequence		
`tabulate` $f\ n$	$O\left(\sum_{i\in[0,n)} W(f(i))\right)$	$O\left(\max_{i\in[0,n)} S(f(i))\right)$
`reduce`* $f\ v\ S$	$O(\|S\|)$	$O(\log \|S\|)$
`nth` $S\ i$	$O(1)$	$O(1)$
Tree Table		
`tabulate` $f\ T$	$O\Big(\sum_{k\in S} W(f(k))\Big)$	$O\Big(\log \|T\| + \max_{k\in S} S(f(k))\Big)$
`reduce`* $f\ v\ T$	$O(\|T\|)$	$O(\log \|T\|)$
`find` $T\ k$	$O(\log \|T\|)$	$O(\log \|T\|)$

Cost of Bellman Ford using a Tables Here we assume the graph G is represented as a $(\mathbb{R}$ `vTable`$)$ `vTable`, where `vTable` maps vertices to values. The $\mathbb{R}$ are the real valued weights on the edges. We assume the distances D are represented as a $\mathbb{R}$ `vTable`. Lets consider the cost of one call to BF, not including the recursive calls. The only non trivial computations are on lines 7 and 10. Line 7 consists of a tabulate over the vertices. As the table indicates, 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 take 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 `vTable`. 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 Bellman Ford 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 11 — Shortest Paths, Graph Representations Revisited

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — October 4, 2011

Today:
- Using Array Sequences to represent graphs
- Continuation of Bellman-Ford's Algorithm (see last lecture notes)
- All pairs shortest paths

1 Representing Graphs With Arrays

Graphs can be represented with an array implementation of sequences instead of tables and sets. The benefit is that we can improve asymptotic performance of certain algorithms, but the cost is that this representation is less general, requiring us to restrict the names of vertices to integers in a fixed range. This can become inconvenient in graphs that change dynamically but is typically OK for static graphs. Furthermore, in the functional setting, we have to be careful since by default, updating an array requires copying the whole array; this is because of the data persistency requirement. This means that if we make an update, we also have to keep the old version. This also means we cannot simply overwrite a value in an existing array. Here, we will describe the interface for a single-threaded sequence that supports efficient updates on the most recent version but works correctly on any version. This is similar to some of the ideas you covered in 15-150.

We refer to the type of this sequence as a `stseq` and it supports the following interface.

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

The costs for `nth`, `update` and `inject` assume the user is using the most recent version. Here we will not define the costs unless using the most recent versions.

Now let's say that we have a graph $G = (V, E)$ where $V = \{0, 1, \ldots, n-1\}$. We will refer to such a graph as an *integer labeled* (IL) graph. For such an IL graph, an α `vertexTable` can be represented as a sequence of length n with the values stored at the appropriate indices. In particular, the table

$$\{(0, a_0), (1, a_1), \cdots, (n-1, a_{n-1})\}$$

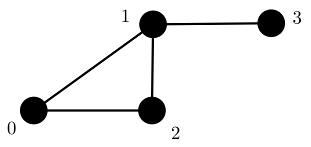
is equivalent to the sequence

$$\langle a_0, a_1, \cdots, a_{n-1} \rangle \ ,$$

using standard reductions between sequences and sets. If we use an array representation of sequences, then this gives us constant work access to the values stored at vertices. We can also represent the set of neighbors of a vertex as an integer sequence containing the indices of those neighbors. Therefore, instead of using an `set table` to represent a graph we can use a

`(int seq) seq.`

For example, the following undirected graph:



would be represented as

$$G = \langle \langle 1, 2 \rangle, \langle 0, 2, 3 \rangle, \langle 0, 1 \rangle, \langle 1 \rangle \rangle.$$

Let's consider how this affects the cost of BFS. We consider the version of BFS that returns a mapping from each vertex to its parent in the BFS tree. This is what was needed for the homework, for example. We can represent the IL graph as a `(int seq) seq`. We can represent the parent mapping P as a `(int option) stseq`. The option is used to indicate whether there is a parent yet. The locations in P with $\mathit{SOME}(v)$ are the visited vertices. We can represent the frontier as an integer sequence containing all the vertices in the frontier. We make P of type `stseq` since it gets updated with changes which are potentially small compared to its length. Then the algorithm is:

```
fun BFS($G$ : (int seq) seq, $s$ : int) =
let
   fun BFS'($P$ : int option stseq, $F$ : int seq) =
      if $|F| = 0$ then toSeq($P$)
      else
         let val $N$ = flatten$\langle\langle(u,v) : u \in G[v]\rangle : v \in F\rangle$   % neighbor edges of frontier
             val $P'$ = inject($N, P$)                                 % new parents added
             val $F' = \langle u : (u,v) \in N \wedge P'[u] = v\rangle$            % remove duplicates
         in BFS'($P', F'$) end
   val $P_{init}$ = $\langle$if ($v = s$) then SOME($s$) else NONE
                  : $v \in \langle 0, \ldots, |G|-1\rangle\rangle$
in BFS'(toSTSeq($P_{init}$), $\langle s \rangle$)
end
```

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

	P : *stseq*		P : *seq*	
line	work	span	work	span
6	$O(\sum_{v \in F} \lvert N_G(v) \rvert)$	$O(\log n)$	$O(\sum_{v \in F} \lvert N_G(v) \rvert)$	$O(\log n)$
7	$O(\sum_{v \in F} \lvert N_G(v) \rvert)$	$O(1)$	$O(n)$	$O(1)$
8	$O(\sum_{v \in F} \lvert N_G(v) \rvert)$	$O(\log n)$	$O(\sum_{v \in F} \lvert N_G(v) \rvert)$	$O(\log n)$
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). Note that the total across rounds is calculated using the fact that every vertex appears in a frontier at most once so that

$$\sum_{i=0}^{d} \sum_{v \in F_i} |N_G(v)| \le |E| = m.$$

We can do a similar transformation to DFS. Here is our previous version.

```
fun $DFS(G : set\ table,\ s : key) =$
let fun $DFS'(X : set,\ v : key) =$
      if $(v \in X)$ then $X$
      else $iterate\ DFS'\ (X \cup \{v\})\ (G_v)$
in $DFS'(\{s\}, s)$ end
```

And the version using sequences.

```
fun $DFS(G : (int\ seq)\ seq,\ s : int) =$
let
   fun $DFS'(X : bool\ stseq,\ v : int) =$
      if $(X[v])$ then $X$
      else $iterate\ DFS'\ (update(X, v, true))\ (G[v])$
   val $X_{init} = \langle (v = s) : v \in \langle 0, \ldots, |G| - 1 \rangle \rangle$
in $DFS'(X_{init}, s)$ 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.

2 All Pairs Shortest Paths

The all pairs shortest path (APSP) problem is: given a weighted directed graph $G = (V, E)$, find the shortest weighted path between ever pair of vertices in V.

It turns out that if the edges all have non-negative weights, then for sparse graphs, the fastest way to solve the APSP problem is simply by solving the (non-negative weight) SSSP problem using each of the vertices as a source. All these can be run in parallel, making the all-pairs problem much more parallel than the single-source problem. If we use the implementation of Dijkstra's algorithm described earlier, and with $n = |V|$ and $m = |E|$, then the overall work and span is:

$$\begin{aligned}
W(m,n) &= \sum_{s \in V} O(m \log n) \\
&= O(mn \log n) \\
S(m,n) &= \max_{s \in V} O(m \log n) \\
&= O(m \log n)
\end{aligned}$$

What about for the general case allowing for negative weights? Later in the course we will see that we can use matrix multiply to solve the APSP problem in $O(n^3 \log n)$ work and $O(\log^2 n)$ span. This is very parallel, but for $m \ll n^2$, it is much more costly than using multiple instances of Dijkstra's algorithm.

For sparse graphs, there is actually another solution that works with negative weights and matches the cost of using multiple instances of Dijkstra's algorithm. The approach boils down to first using Bellman-Ford to convert the graph into one that has no negative weights, and then using Dijkstra's algorithm in parallel across the vertices. The graph is converted in a way such that the shortest path taken between every pair of vertices does not change although the weights of the edges on the path might. Before going further, it is worth thinking about what systematic changes on edges will not change the shortest path taken between two vertices. You might think of simply increasing all the weights on all the edges equally. Unfortunately, this does not work.

Exercise 1. *Come up with a small example that shows that increasing the weight of all edges in the graph equally, changes the shortest paths.*

We could consider multiplying all the weights by any positive number. This will not affect the paths taken, but unfortunately, it does not help us since the negative edges will remain negative. What if we take a vertex and increase every out edge by some constant p_v and decrease every in-edge by the same constant. Does this maintain the shortest paths? Let us consider the possible cases. Firstly, if a path goes through v, then the weight of the path is decreased by v on the edge into v and increased by the same amount on the edge out of v. Therefore, the weight of the path is not changed. Secondly, a path could start at v. In this case all paths out of v increased by the same amount so the shortest path is not affected (although its weight will be). Similarly, for the paths that finishes at v: all paths will be decreased by the same amount. Therefore, this transformation would appear not to change the shortest path between any pair of vertices.

If we can modify the in- and out- weights of one vertex, then we can modify them all. This suggests an idea of assigning values to all vertices and adjusting the edges accordingly. More specifically, we'll assign a real valued "potential" to each vertex, which we indicate as $p : v \to \mathbb{R}$. Now each directed edge (u, v) will be reweighted so that its new weight is

$$w'(u,v) = w(u,v) + p(u) - p(v).$$

(i.e. we add the potential of u going out of u, and subtract the potential of v coming in to v. This leads to the following lemma:

Lemma 2.1. *Given a weighted directed graph $G = (V, E, w)$ with weight function $w : E \to \mathcal{R}$, and "potential" function $p : v \to \mathbb{R}$, then for a graph $G' = (V, E, w')$ with weights*

$$w'(u, v) = w(u, v) + p(u) - p(v),$$

we have that for every path from s to t,

$$W_{G'}(s, t) = W_G(s, t) + p(s) - p(t)$$

where $W_G(s, t)$ is the weight of the path from s to t in graph G.

Proof. Each s-t path is of the form $\langle v_0, v_1, ..., v_k \rangle$, where $v_0 = s$ and $v_l = t$. For every vertex $v_i, 0 < i < k$ in the path the conversion to w' removed p_{v_i} from the weight when entering v_i and added it back in when leaving, so they cancel (the overall sum is a telescoping sum). Therefore, we need only consider the two endpoints, giving the desired result. □

This lemma shows that the potential weight transformation does not affect the shortest paths since all paths between the same two vertices will be affected by the same amount (i.e. $p(s) - p(t)$ for vertices s and t).

Now the question is whether we can pick potentials so that they get rid of all negative weight edges. The answer is yes. The trick is to add a source vertex and link it to all other vertices V with an edge weight of zero. Now we find the shortest path from s to all vertices using the original weights. This can be done with a SSSP algorithm that allows negative weights (e.g. Bellman-Ford). Now we simply use the distance from s as the potential.

Exercise 2. *Show that the this potential guarantees that no edge weight will be negative.*

This together with a parallel application of SSSP with non-negative weights (indicated as SSSP$^+$) gives us our desired algorithm. In particular:

```
fun APSP(G) =
let
   val $G' = G$ with a new source $s$ and 0 weight edges from $s$ to each $v \in V$
   val $D = SSSP(G', s)$
   val $W'' = \{(u, v) \mapsto w(u, v) + D(u) - D(v) : (u, v) \in E_{G'}\}$
   val $G'' = (V_{G'}, E_{G'}, W'')$
in
   $\{v \mapsto SSSP^+(G'', v) : v \in V\}$
end
```

We now consider the work and span of this algorithm. If Bellman Ford is used for SSSP and Dijkstra for SSSP$^+$ then we can just add the work and span of Bellman Ford to the work and span we analyzed earlier for the parallel application of Dijkstra. The cost of the parallel application of Dijkstra dominates giving $O(mn \log n)$ work and $O(m \log n)$ span.

Lecture 13 — Graph Contraction, Connectivity

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — October 11, 2011

Announcements:
- Exam 1 back at end of class
- Assignment 5 due Oct. 18

Today:
- Dijkstra Costs
- Graph Contraction

1 Cost of Dijkstra

Most of the class did not get the short question about the cost of Dijkstra's algorithm when using arrays. We probably did not cover the cost of Dijkstra's algorithm in enough detail, so we will review it here. Dijkstra's algorithm basically operates on three data structures: (1) a structure for the graph itself, (2) a structure to maintain the distance to each vertex that has already been visited, and (3) a priority queue holding distances of vertices that are neighbors of the visited vertices.

Here is Dijkstra's algorithm with the operations on these data types in boxes.

```
1   fun dijkstra$(G, s)$ =
2   let
3      fun dijkstra'$(D, Q)$ =
4         case $\boxed{\texttt{PQ.deleteMin}}(Q)$ of
5            $(\texttt{PQ.empty}, \_) \Rightarrow D$
6          | $((d, v), Q') \Rightarrow$
7            if $(\boxed{(v, \_) \in D})$ then dijkstra'$(D, Q')$
8            else let
9               fun relax $(Q, (u, w)) = \boxed{\texttt{PQ.insert}}(d + w, u)\ Q$
10              val $N = \boxed{N_G(v)}$
11              val $Q'' =$ iterate relax $Q'\ N$
12           in dijkstra'$(\boxed{D \cup \{(v, d)\}}, Q'')$ end
13  in
14     dijkstra'$(\{\}, \texttt{PQ.insert}(\emptyset, (0, s)))$
15  end
```

The `PQ.insert` in Line 14 is called only once, so we can ignore it. Of the remaining operations, Lines 10 and 11 are on the graph, Lines 7 and 12 are on the table of visited vertices, and Lines 4 and

9 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 $O(\log n)$ 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	$O(m)$	$O(\log n)$	-	-	-
`insert`	9	$O(m)$	$O(\log n)$	-	-	-
Priority Q total			$O(m \log n)$	-	-	-
`find`	7	$O(m)$	-	$O(\log n)$	$O(1)$	$O(1)$
`insert`	12	$O(n)$	-	$O(\log n)$	$O(n)$	$O(1)$
Distances total			-	$O(m \log n)$	$O(n^2)$	$O(m)$
$N_G(v)$	10	$O(n)$	-	$O(\log n)$	$O(1)$	-
`iterate`	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 Lines 10 and 12 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.

2 Graph Contraction

So far we have mostly talking about standard techniques for solving problems on graphs that were developed in the context of sequential algorithms. Some of these are easy to parallelize while others are not. For example, we saw there is parallelism in BFS since each level can be explored in parallel, assuming the number of levels is not too large. However, 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.[2] There was plenty of parallelism in the Bellman-Ford algorithm, and also in the all pairs shortest path algorithms since they are based on parallel application of Dijkstra (and perhaps Bellman Ford preprocessing if there are negative weights).

We are now going to discuss some techniques that will add to your toolbox for parallel algorithms.

[1]We could also use a hash table, but we have not yet discussed them.

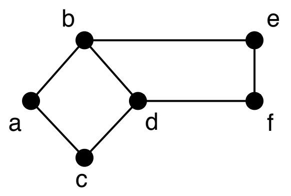
[2]In reality there is some parallelism in both DFS and Dijkstra when graphs are dense—in particular, although vertices need to visited sequentially the edges can be processed in parallel. If we have time, we will get back to this when we cover priority queues in more detail.

The first of these techniques is graph contraction. This is actually a reasonably simple technique and can be applied to a variety of problems including graph connectivity, spanning trees, and minimum spanning trees. In the discussion of graph contraction, we will assume that the graph is undirected unless otherwise stated. The basic outline of the approach is the following:

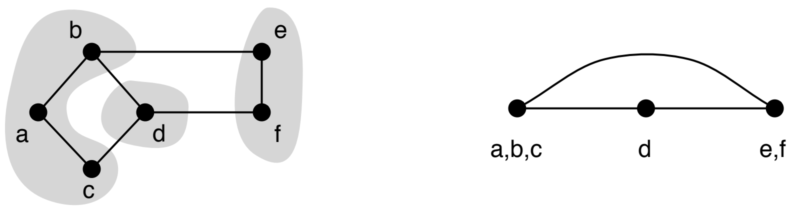
```
**ContractGraph**($G = (V, E)$) =
  1. Identify a set of disjoint connected components in $G$
  2. $V'$ = the set of vertices after contracting each component into a single vertex
  3. $E'$ = after relabeling each edge so its endpoints refer to the new vertex
  4. $E''$ = remove self-loops (and parallel edges)
  5. If $(|E''| > 0)$ then **ContractGraph**($G' = (V', E'')$)
```

We refer to each recursive call as a contraction step.

Now let's go through some examples of how we might contract a graph. Consider the following graph:



In this graph, we could identify the disjoint components $\{a, b, c\}, \{d\}, \{e, f\}$.



After contracting, we would be left with a triangle. Note that in the intermediate step, when we join a, b, c, we create redundant edges to d (each one of them had an original edge to d). We therefore replace these with a single edge. However, in some algorithms, it is convenient to allow parallel (redundant) edges rather than going to the work of removing them. This is sometimes referred to as a multigraph.

Instead of contracting $\{a, b, c\}, \{d\}, \{e, f\}$, we could contract the components $\{a, c\}, \{b, d\}, \{e, f\}$. In this case, we would be left with three vertices connected in a line. In the two limits, we could contract nothing, or contract all vertices.

There are a couple special kinds of contraction that are worth mentioning:

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.

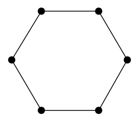
Star Contraction: One vertex of each component is identified as the center of the star and all other vertices are directly connected to it.

Why, you might ask, is contraction useful in parallel algorithms? Well, it is first worth noting that if the size of the graph reduces by a constant factor on each step, then the algorithm will finish after only $O(\log n)$ steps. (This is the familiar recurrence $f(n) = f(n/2) + c$.) Therefore, if we can run each step in parallel, we have a good parallel algorithm. In fact, such algorithms can be theoretically much more parallel than Breadth First Search since the parallelism no longer depends on the diameter of the graph. However, even if we can contract in parallel, how can we use it to do anything useful? One reason is that contraction maintains the connectivity structure of the graph. Therefore, if we start out with k connected components in a graph, we will end up with k components. It also turns out that if we pick the edges to contract on carefully, then the contraction maintains other properties that are useful. For example, we will see how it can be used to find minimum spanning trees.

After answering how contraction is useful, the next question to ask is: how do we select components? Remember that we would like to do this in parallel. Let's start by limiting ourselves to edge contractions. Any ideas?

In particular in parallel we want to select some number of disjoint edges (i.e that don't share an endpoint). Ideally, the edges should cover a constant fraction of the vertices so that the graph contracts sufficiently.

Being able to do this efficiently and deterministically turns out to be quite a difficult problem. The issue is that from the point of view of every vertex the world might look the same. Consider a graph that is simply a cycle of vertices each pair connected by an (undirected) edge. The example below shows a 6-cycle (C_6).



Wherever we are on the cycle, it looks the same, so how do we decide how to hook up? As an example, if each node tries to join the person to the right, we are not going to make any progress. We essentially need a way to break symmetry. It turns out that randomization is a huge help. Any ideas how we might use randomization?

On a cycle, we could flip a coin for each edge. Then, the rule is: *if an edge gets a heads and both its neighbors*[3]*, then select that edge.* This guarantees that no two adjacent edges will be selected so our components are disjoint. Now we can contract each edge and we are left with a smaller cycle.

How much do we expect this graph to contract? Let's assume that all coin flips are unbiased and are independent. What is the expectation that an edge is selected? Notice that the probability that the edge comes up heads and both neighbors come up tails is $\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = \frac{1}{8}$. To calculate the expected number of edges that are removed, the easiest way is to apply linearity of expectations. In particular, the expectation that each edge is selected is $1/8$, so if we have a cycle of length n (which has n vertices and n edges), then the expected number of edges that are removed is $n/8$.

Exercise 1. *Come up with a randomized scheme that on a graph that consists of a single undirected cycle of length n selects an disjoint set of edges of expected size $n/3$.*

[3]of which there are 2 in a cycle

Lecture 14 — Graph Contraction, Min Spanning Tree (DRAFT)

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — October 13, 2011

Today:
- Graph Contraction Continued
- Some Probability Theory

1 Graph Contraction Continued

In the last class we were discussing the idea of graph contraction, which worked in a set of rounds each which contracts the graph. In each round we identify a set of disjoint connected subgraphs and contract each one into a single vertex, and then update the edges. Graph contraction is a powerful technique to generate parallel algorithms for a variety of problems on undirected graphs. The approach can also be applied to generate efficient algorithms on trees, although we won't cover these in this course.

To help understand the key issues in graph contraction we were considering a special case. In particular we were considering just edge contractions—where each connected subgraph we contract is an edge with its two endpoints. Furthermore we were just considering graphs that consists of a set of cycles. This might seem very limited, but actually it exposes most of what is interesting about graph contraction. After understanding this case, we will generalize to arbitrary graphs.

We had discussed how to select a set of independent edges in the graph (i.e. that don't share any endpoints) using randomization. In particular the rule we considered was flipping a coin on every edge and selecting an edge if it is heads and its two neighbors in the cycle are tails. This can easily be done in parallel and gives a probability of $1/8$ that each edge is selected. For example, in the following cycle with the indicated coin flips we will select 2 out of the 12 edges.

```
  a   T   b   H   c   H   d   T   e   H   f
  o --- o --- o --- o --- o --- o
T |                             | T
  o --- o --- o --- o --- o --- o
  l   H   k   H   j   T   i   H   h   T   g
```

After contracting these two edges we will have 10 edges remaining for the next round.

```
  a     b     c     d     ef
  o --- o --- o --- o --- O
T |                       | T
  o --- o --- o --- O --- o
  l     k     j    ih     g
```

Using this scheme we remove an average of $1/8^{th}$ of the remaining edges on each round. Assuming we always remove exactly $1/8^{th}$ we would have the following recurrence for the number of rounds

$$R(n) = R(7n/8) + 1.$$

Since the size is decreasing geometrically this solves to $O(\log n)$ rounds. Furthermore if we use array sequences to represent the graph, each round can be done in $O(\log n)$ span (needed for filtering out the remaining edges), and $O(n)$ work. This would give $O(\log^2 n)$ overall span, and the recurrence for work is

$$W(n) = W(7n/8) + O(n) = O(n).$$

Unfortunately we don't remove exactly $1/8^{th}$ of the vertices, but the number removed varies from round to round depending on the toss of the coins. We therefore need to be more careful about the analysis. For this purpose probability theory is very useful. We therefore will spend some time here reviewing basic definitions as well as introducing some new ideas you probably have not seen. These ideas will also be useful elsewhere in the course.

Probability Theory: Review Recall from probability theory that a *sample space* is the set Ω of all possible outcomes. In the case of flipping coins on n edges, the size of the sample space is 2^n representing all possible outcomes, which are all equally likely if the coins are unbiased. An *event* $\mathcal{E} \subseteq \Omega$ is any subset of the sample space. In our example, it might consist of all outcomes such that a particular edge is selected (it comes up heads and its neighbors tails), or all outcomes where the number of edges selected is at least $7n/8$. We can then ask what the probability of that event is by simply counting the number of outcomes in the event and dividing by the total number of possible outcomes, i.e., $\mathbf{Pr}[\mathcal{E}] = |\mathcal{E}|/|\Omega|$ if each outcome in the sample space is equally likely to happen.

A *random variable* is a function $f : \Omega \to \mathbb{R}$. We typically denote random variables by capital letters $X, Y, \ldots$. That is, a random variable assigns a numerical value to an outcome $\omega \in \Omega$. In our algorithm, one useful random variable is the count of number of selected edges. It maps each outcome to a number between 0 and $n/2$ (why not n?). Another is simply the count on a single edge which maps an edge to either 1 (if selected) or 0 (if not selected). Such a random variable that is zero or 1 is sometimes called an *random indicator variable*. For an event $\mathcal{E}$, the indicator random variable $\mathbb{I}\{\mathcal{E}\}$ takes on the value 1 if $\mathcal{E}$ occurs and 0 otherwise.

The *expectation* (or expected value) of a random variable is simply the weighted average of the value of the function over all outcomes. The weight is the probability of each outcome, giving

$$\mathbf{E}[X] = \sum_{\omega \in \Omega} X(\omega)\,\mathbf{Pr}[\omega]$$

in the discrete case. Applying this definition, we have $\mathbf{E}[\mathbb{I}\{\mathcal{E}\}] = \mathbf{Pr}[\mathcal{E}]$.

In our example, we are interested in the expectation that each single edge is selected and the expectation on the total number of edges selected. The expectation that a single edge is selected is $1/8$ since there are 8 possible outcomes on the coin tosses of three neighbors, and only one of them leads to the edge being selected.

Recall that one of the most important rules of probability is *linearity of expectations*. It says that given two random variables X and Y, $\mathbf{E}[X] + \mathbf{E}[Y] = \mathbf{E}[X + Y]$. This does not require that the events are independent. In particular, if the events are selection of edges, two neighboring edges are certainly not independent (why?), but yet we can still add their expectations. So the expectation that two neighboring edges are selected is $1/8 + 1/8 = 1/4$. And hence the expected total number of edges selected is $n \times 1/8 = n/8$.

The *union bound* (Boole's inequality) states that for a set of events $A_1, A_2, \ldots$, that

$$\mathbf{Pr}\left[\bigcup_i A_i\right] \le \sum_i \mathbf{Pr}\left[A_i\right].$$

If the events do not overlap (their sets of outcomes do not intersect) then the equation is an equality since we just add up the probabilities of every outcome. If they do overlap then the probability of the union is strictly less than the sum of the probabilities since some outcomes are shared. We will find the union bound this useful in showing high-probability bounds, discussed below.

Now we discuss two important techniques we will use. The first is a way to solve for the expectation on recurrences when the size of a subcall is a random variable. This is exactly what we need for the contraction example. The second is the idea of high probability bounds and how they can be used to analyze span.

Recurrences with Random Variables We are interested in recurrences of the form $F(n) = F(n - X_n) + 1$ where X_n is a random variable with a distribution depending on n. In our example it is the number of edges we remove and hence $\mathbf{E}\left[X_n\right] = n/8$. However, we cannot simply plug in the $n/8$ to get $F(n) = F(n - n/8) + 1$ since X_n is a probability distribution and $n/8$ is just the average. But we can use the following useful lemma.

Lemma 1.1 (Karp-Upfal-Widgerson). *Let $T(n) = 1 + T(n - X_n)$ where for each $n \in \mathbb{Z}_+$, X_n is an integer-valued random variable satisfying $0 \le X_n \le n - 1$ and $T(1) = 0$. Let $\mathbf{E}\left[X_n\right] \ge \mu(n)$ for all $n \ge 1$, where μ is a positive non-decreasing function of n. Then,*

$$\mathbf{E}\left[T(n)\right] \quad\le\quad \int_1^n \frac{1}{\mu(t)}dt.$$

The proof is mostly mechanical, so we didn't do it in class. But we'll prove it here for completeness.

Proof of Lemma 1.1. Let

$$h(x) = \int_{t=1}^{x} \frac{dt}{\mu(t)}.$$

We'll show that $\mathbf{E}\left[T(n)\right] \le h(n)$ by induction on n. The base case is trivial. Now assume that this holds for all $m < n$–and show that it holds for n. By the recurrence definition, we have

$$\begin{aligned}
\mathbf{E}\left[T(n)\right] &\le\ 1 + \mathbf{E}\left[T(n - X_n)\right] \ \le\ 1 + \mathbf{E}\left[h(n - X_n)\right] \\
&\le\ 1 + \mathbf{E}\left[\int_{t=1}^{n-X_n} \frac{dt}{\mu(t)}\right] \ \le\ 1 + \mathbf{E}\left[\int_{t=1}^{n} \frac{dt}{\mu(t)} - \int_{t=n-X_n}^{n} \frac{dt}{\mu(t)}\right] \\
&\le 1 + h(n) - \mathbf{E}\left[\int_{t=n-X_n}^{n} \frac{dt}{\mu(n)}\right] \ \le\ 1 + h(n) - \frac{\mathbf{E}\left[X_n\right]}{\mu(n)} \le h(n).
\end{aligned}$$

□

Notice that we could change $h(x)$ to $\sum_{i=1}^{x} \frac{1}{\mu(i)}$ and the proof will still go through. This gives us a slightly friendlier expression:

$$\mathbf{E}\left[T(n)\right] \leq \sum_{i=1}^{n} 1/\mu(i).$$

This lemma can be applied to many problems. In our case, it leads immediately to a solution for the expected number of rounds of contraction:

$$\begin{aligned}\mathbf{E}\left[R(n)\right] &\leq \int_1^n \frac{1}{t/8} dt \\ &= 8 \ln n.\end{aligned}$$

Therefore, the expected number of rounds is indeed logarithmic.

High Probability Bounds Unfortunately, knowing the expectation of components is not always good enough when composing costs. In analyzing the work of an algorithm (equivalent to time in sequential algorithms) we add the work for the various components. Linearity of expectations turns out to be very useful for this purpose since if we know the expected work for each component, we can just add them to get the overall expected work. However when analyzing span over a parallel computation we take the maximum of the span of the components. Unfortunately the following is (in general) not true: $\max(\mathbf{E}\left[X\right], \mathbf{E}\left[Y\right]) = \mathbf{E}\left[\max(X, Y)\right]$. We therefore cannot simply take the maximum of expectations across components to get the overall expected span.

As an example consider two parallel components each which have span either 1 or 5 each with .5 probability. The expected span for each of the components is therefore $(.5 \times 1 + .5 \times 5 = 3$. Now, in combination there are four possible configurations $((1, 1), (1, 5), (5, 1), (5, 5))$. Since we take the max of the two components the overall span for the four are $1, 5, 5, 5$. The expected span is the average of these: $.25 \times (1 + 5 + 5 + 5) = 4$. However if we simply took the maximum of the expected span of each component we would have $\max(3, 3) = 3$, which is wrong. The difference can be much greater than this example.

To deal with this issue instead (or in addition to) figuring out the expectation of a random variable we determine a high probability bound. We say that an event $\mathcal{E}$ takes place *with high probability* if

$$\mathbf{Pr}\left[\mathcal{E}\right] \geq 1 - \frac{1}{n^c}$$

for some constant $c \geq 1$. Here n is the instance size. Typically, in bounding the cost of an algorithm, we're interested in showing that $X_n \leq A$ with high probability. In other words, we want to say

$$\mathbf{Pr}\left[X_n \geq A\right] < \frac{1}{n^k}$$

This is saying that the probability of a "bad" event (the random variable takes on a value greater than some value A) is less than $1/n^k$ for some constant k. This probability is very low. We don't normally say something happens "with low probability", because we are interested in the opposite "good" case. For example, for us the X_n will be some cost measure and we want to say that with high probability the cost will not be greater than some A, i.e., that the probability that it is greater is small.

Combining high-probability bounds when calculating span turns out to be much easier than using expectation. We will get back to this. For now we will just state that it is possible to show that

$$\mathbf{Pr}\left[R(n) \geq 16k \ln n\right] < \frac{1}{n^k}.$$

So, the number of rounds of graph contraction assuming each rounds removes an expected constant fraction ($1/8$ in example) is $O(\log n)$ with high probability.

2 Graph Contraction on General Graphs

Now lets return to graph contraction but now considering general graphs instead of just cycles. Consider a star graph. A star graph is a graph with a single vertex in the middle and all the other vertices hanging off of it with no connections between them.

Draw picture.

Will edge contraction work well on this? How many edges can contract on each step.

Instead we will consider star contraction. The idea is to allow a vertex v to contract with all its neighbors. The vertex is called the star center. The neighbors can be connected with each other, but they are all viewed as contracting with v. We will use a coin to determine if the vertex will be the center of a star. The basic approach is:

```
Find stars:
  1.  Every vertex flips an unbiased coin to decide if it a center
  2.  Every non-center tries to hook up with a neighbor that is a center
  3.  All the hooks identify a set of stars
```

It should be clear that stars are disjoint since every non center only hooks up with one center, and the center does not hook up with anyone.

How many vertices do we expect to remove?

Lecture 15 — Graph Contraction, Min Spanning Tree

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — October 18, 2011

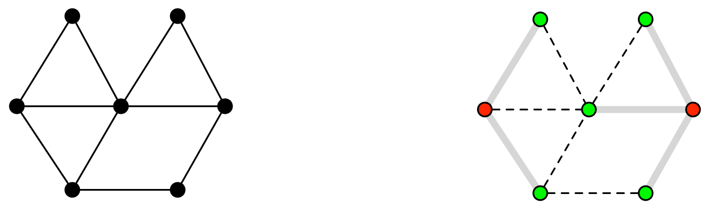
Today:
- Analysis of Graph Contraction
- Minimum Spanning Tree

1 Graph Contraction on General Graphs

Last time, we described a form of graph contraction based on contracting stars. We begin this lecture by recalling the definition of stars.

Definition 1.1 (Star)**.** In an undirected graph $G = (V, E)$, a *star* is a subgraph of G with a center vertex v, a set of neighbors of $U_v \subseteq \{u \mid (v, u) \in E\}$ and the edges between them $E_{v,U} = \{(v, u) \mid u \in U_v\}$.

In fact, the star graph on $n + 1$ vertices is the bipartite graph $K_{1,n}$, which is a tree. The basic idea of star contraction is to identify and contract non-overlapping stars in a graph. For example, the following graph (left) contains 2 non-overlapping stars (right). The centers are colored red and the neighbors, green.



Based on the idea of stars, we can use the following algorithm for graph contraction:

```
% requires:  an undirected graph $G = (V, E)$ and random seed $r$
% returns:  $V'$ = remaining vertices after contraction,
%           $P$ = mapping from removed vertices to $V'$
fun starContract($G = (V, E), r$) =
let
    % flip coin on each vertex
    val $C = \{v \mapsto$ coinFlip$(v, r) : v \in V\}$
    % select edges that go from a tail to a head
    val $TH = \{(u, v) \in E \mid \neg C_u \wedge C_v\}$
    % make mapping from tails to heads, removing duplicates
    val $P = \cup_{(u,v) \in TH} \{u \mapsto v\}$
    % remove vertices that have been remapped
    val $V' = V \setminus$ domain$(P)$
in $(V', P)$ end
```

The `starContract` procedure can be used for `identifyComponents` in the following generic graph-contraction algorithm:

```
fun graphContract$((V, E), r)$ =
if $|E| = 0$ then $V$ else
   let
      val $(V', P) =$ identifyComponents$((V, E), r)$
      val $E' = \left\{(\overline{P_u}, \overline{P_v}) : (u, v) \in E \mid \overline{P_u} \neq \overline{P_v}\right\}$
   in
      graphContract$((V', E'),$ next$(r))$
   end
```

where $\overline{P_u}$ indicates the function that returns p if $(u \mapsto p) \in P$ and u otherwise (i.e., if the vertex has been relabeled it grabs the new label, otherwise it keeps the old one).

This returns one vertex for every connected component in the graph. We can therefore use it to count the number of components. To accomplish more interesting tasks, we have to augment the code similarly to we did with the generic BFS and DFS algorithms. Before looking at how to augment it, let's analyze the cost of the code. We say that a vertex is *attached* if it has at least one neighbor in the graph. We now prove the following lemma:

Lemma 1.2. *For a graph G with n attached vertices, let X_n be the random variable indicating the number of vertices removed by* `starContract`*(G, r). Then,* $\mathbf{E}[X_n] \geq n/4$.

Proof. Let H_v be the event that a vertex v comes up heads, T_v that it comes up tails, and R_v that it is removed. By definition, we know that an attached 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 $\mathbf{Pr}[R_v] \geq \mathbf{Pr}[T_v]\,\mathbf{Pr}[H_u] = 1/4$. By the linearity of expectation, we have the number of removed vertices is

$$\mathbf{E}\left[\sum_{v:v \text{ attached}} \mathbb{I}\{R_v\}\right] = \sum_{v:v \text{ attached}} \mathbf{E}\left[\mathbb{I}\{R_v\}\right] \geq n/4$$

since we have n attached vertices. □

Exercise 1. *What is the probability that a vertex with degree d is removed.*

Now we can write a recurrence for the number of rounds required by the `graphContract` using `starContract` for each step. In particular, let n be the number of attached vertices and $R(n)$ be the number of rounds. When there are no attached vertices, the algorithm terminates since there are no edges and there cannot be one attached vertex, so we have $R(1) = 0$. Although vertices can become detached, they can never become attached again, so we have

$$R(n) \leq 1 + R(n - X_n).$$

Now recall that the Karp-Upfal-Wigderson lemma from last class gave us the solution

$$\mathbf{E}[R(n)] \leq \sum_{i=1}^{n} \frac{1}{\mu(i)}$$

where $\mathbf{E}[X_n] \geq \mu(n)$. This is the summation form; we also did an integral form. In this particular case, we have $\mu(n) = n/4$, giving a similar sum as we saw in the last lecture:

$$\mathbf{E}[R(n)] \leq \sum_{i=1}^{n} \frac{4}{i} = 4H_n = \Theta(\log n),$$

where H_n is the n-th Harmonic number. You should note that this sum will come up many times in the analysis of randomized algorithms.

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 constants and a low-order term).

Using arrays, it is reasonably easy to implement each step of `starContract` using $O(n + m)$ work and $O(\log n)$ span. Therefore, since there are $O(\log n)$ rounds, the overall span of the algorithm is $O(\log^2 n)$.

Ideally, we would like to show that the overall work is linear since we might expect that the size is going down by a constant fraction on each step. However, this is not the case. Although we have shown that we can remove a constant fraction of the attached vertices on one star contract step, 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. However, this does not help asymptotically bound the number of edges removed. Consider the following sequence of steps:

step	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. All together this gives us the following theorem:

Theorem 1.3. *For a graph $G = (V, E)$, graph contraction using* `starContract` *with an array sequence works in $O(|E| \log |V|)$ work and $O(\log^2 |V|)$ span.*

2 Minimum Spanning Tree

The minimum (weight) spanning tree (MST) problem is given an connected undirected graph $G = (V, E)$, find a spanning tree of minimum weight (i.e. sum of the weights of the edges). You have seen Minimum Weigh Spanning Trees in both 15-122 and 15-251.

We believe in both classes, you went over Kruskal's algorithm. The basic structure was

```
sort edges by weight
put each vertex in its own component
for each edge e = (u,v) in order
   if u and v are in the same component skip
   else join the components for u and v and add e to the MST
```

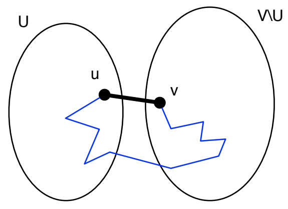
You used a union-find data structure to detect when two vertices are in the same component and join them if not.

The main property that makes Kruskal's algorithm as well as most other algorithms work 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 property that we will use is the following:

Theorem 2.1. *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.*

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.

This property was used in Kruskal's algorithm although we won't review this use here. Another algorithm that uses this property is Prim's algorithm. It basically uses a priority first search to grow the tree starting at an arbitrary source vertex. The algorithm maintains a visited set U, which also corresponds to the set U in the cut. At each step, it selects the minimum weigh edge $e = (u, v), u \in U, v \in V \setminus U$. This is in the MST by the theorem 2.1. It adds the adjoining vertex to U and e to the MST. After $|V|$ steps, it has added all vertices to the tree and it terminates. To select the minimum weight edge leaving U on each step, it stores all edges leaving U in a priority queue. The algorithm is almost identical to Dijkstra's algorithm but instead of storing distances in the priority queue, it stores edge weights.

Both Kruskal's and Prim's algorithms are sequential.

Here we show a parallel algorithm based on an approach by Borůvka that actually predates both Kruskal and Prim. We use graph contraction. The idea is actually quite simple. During graph contraction, we can consider any of the contracted components as our set U. Therefore, the minimum edge out of each component must be in the MST. We can modify our `starContract` routine so that after flipping coins, the tails only attach across their minimum weight edge. The algorithm is as follows.

```
1  fun $\texttt{minStarContract}(G = (V, E), r) =$
2  let
3     val $minE = \texttt{minEdges}\ (G)$
4     val $C = \{v \mapsto \texttt{coinFlip}(v, r) : v \in V\}$
5     val $P = \{(u \mapsto (v, l)) \in minE \mid \neg C_u \wedge C_v\}$
6     val $V' = V \setminus \texttt{domain}(P)$
7  in $(V', P)$ end
```

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 $(\texttt{vertex} \times \texttt{vertex} \times \texttt{weight} \times \texttt{label})$ where the two vertex endpoints can change as the graph contracts but the weight and label stays fixed. The function $\texttt{minEdges}(G)$ 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. Recall that by Theorem 2.1 since all these edges are minimum out of the vertex they are safe to add to the MST. Line 5 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 6 removes all vertices that have been relabeled.

This can now be used in the following MST code, which is similar to the `graphContract` code earlier in the lecture except we return the set of labels for the MST edges instead of the remaining vertices.

```
fun $\texttt{MST}((V, E), r) =$
if $|E| = 0$ then $\{\}$ else
   let
      val $(V', PT) = \texttt{identifyComponents}((V, E), r)$
      val $P = \{u : (u, l) \in PT\}$
      val $T = \{l : (u, l) \in PT\}$
      val $E' = \left\{(\overline{P_u}, \overline{P_v}) : (u, v) \in E \mid \overline{P_u} \neq \overline{P_v}\right\}$
   in
      $\texttt{MST}((V', E'), \texttt{next}(r)) \cup T$
   end
```

Lecture 16 — Bipartite Graphs, Error Correcting Codes, and Set Cover

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — October 20, 2011

Today:
- Bipartite Graphs (bigraphs)
- Bigraphs in error correcting codes
- Bigraphs in set cover

1 Bipartite Graphs

A bipartite graph is a graph with two sets of vertices which are connected to each other, but not within themselves. More formally a bipartite graph $G = (U, V, E)$, consists of a set of vertices U a disjoint set of vertices V and a set of edges $E \subset U \times V$. As with regular graphs we might or might not put weights or other labels on the edges. Bipartite graphs have many applications. They are often used to represent binary relations between two types of objects. A *binary relation* between two sets A and B is a subset of $A \times B$. We can see that this is equivalent to the definition of bipartite graphs as long as A and B are disjoint (i.e. $A \cap B = \emptyset$). In the next lecture we will note that bipartite graphs (and hence binary relations) can also be viewed as matrices. Which view is "best" depends on the context or even on what someone is comfortable with. However, often using the graph view is more intuitive since it gives a pictorial interpretation of what is going on, instead of just a set of pairs.

Here are just a few examples applications of bipartite graphs:

Document/Term Graphs: Here U are documents and V are terms or words, and there is an edge (u, v) if the word v is in the document u. Such graphs are use often to analyze text, for example to cluster the documents.

Movies preferences: In 2009 Netflix gave a $1Million prize to the group that was best able to predict how much someone would enjoy a movie based on their preferences. This can be viewed, and in the submissions often was, as a bipartite graph problem. The viewers are the vertices U and the movies the vertices V and there is an edge from u to v if u viewed v. In this case the edges are weighted by the rating the viewer gave. The winner was algorithm called "BellKor's Pragmatic Chaos". In the real problem they also had temporal information about when a person made a rating, and this turned out to help.

Error correcting codes: In low density parity check (LDPC) codes the vertices U are bits of information that need to be preserved and corrected if corrupted, and the vertices V are parity checks. By using the parity checks errors can be corrected if some of the bits are corrupted. LDPC codes are used in satellite TV transmission, the relatively new 10G Ethernet standard, and part of the WiFi 802.11 standard. We discuss this in this class.

Students and classes: We might create a graph that maps every student to the classes they are taking. Such a graph can be used to determine conflicts, e.g. when classes cannot be scheduled together.

Stable marriage and other matching problems: You've seen this in 251. This is an approach used for matching graduating medical students to resident positions in hospitals. Here U are the students and V the resident slots at hospitals and the edges are the rankings of both the hospitals and residents.

Set Cover: to be covered in this lecture.

2 Error Correcting and LDPC Codes

Here we will only cover a brief introduction to error correcting codes and specifically low density parity check (LDPC) codes. Assume we are given n bits that we want to transmit, but that our transmission media (e.g. wires or radio) is noisy. Such noise will corrupt some of the bits by either making them unreadable, or even worse, flipping them. As industry gets more and more aggressive about how much information they want to transmit over a fixed "sized" channel, the more likely it becomes that signals will be corrupted. In fact theory says that you are better off transmitting at a high enough rate that you will regularly get errors, and then correcting them, than you would be by being conservative and rarely getting errors. Indeed this is part of the reason transmission rates have improved significantly over the years.

The idea of all error correcting codes is to add extra bits of information that can be used to fix certain errors. As long as the number of bits that are corrupted during transmission is small enough then the error can be fixed. Exactly how many errors can be fixed depends on the number of extra bits as well as the specific type of code that is used. We will use c to indicate the number of extra bits. In error correcting codes we imagine the following model:

```
 Input
  |  n-bit message
  v
ENCODER
  |  n + c bit codeword
  v
NOISY CHANNEL
  | possibly corrupted n+c bit codeword
  v
FIX ERRORS
  |  valid n+c bit codeword, hopefully matches
  v       codeword before errors
DECODER
  | n-bit message extracted from codeword
  v          hopefully matches input
 Output
```

In some codes the codeword is simply the original message with some extra bits tagged on (systematic codes) but generally the encoder might arbitrarily transform the message. In the remainder of our discussion we are only going to be interested in the codewords and the step that fixes the errors and not about how to encode and decode.

```
      o                              0 o

      o       o                      1 o       o

      o                              1 o

U     o       o   V                  0 o       o   V

      o                              1 o

      o       o                      0 o       o

      o                              1 o

   n + c      c           A particular setting of code bits
code bits   parity check
            constraints
```

Figure 1: The bipartite graph corresponding to a low density parity check code. The edges between the code bits and parity check constraints are not shown.

LDPC codes. We now consider LDPC codes specifically. An LDPC coding scheme is defined by a bipartite graph. This graph is fixed for all time, i.e. all encoders and decoders use the same graph. In the graph there are $n + c$ vertices on the left (U), c vertices on the right (V), and edges between them (E). Figure 1 gives an example. Typically c will be some factor smaller than n, e.g. $c = n/4$. We now consider the meaning of each of U, V and E.

The vertices U on the left correspond to the bits of a codeword, and hence there are $n + c$ of them. These will be exactly the bits we send over our noisy channel and which might get corrupted. Any particular input message of length n will map to a particular codeword of length $n + c$. The right side of Figure 1 shows a particular setting of the bits. We will not explain how to generate an $(n + c)$-bit codeword from an n-bit message, but will instead just explain how we fix a codeword after it has been corrupted by noise, which is the more interesting part.

Now lets consider the vertices V on the right. Each of these corresponds to a *parity check* constraint. Each parity check on the right puts a constraints on its neighbors. In particular it requires that the neighbors must add to an even number. This is called even parity. This means that only certain code words on the left are valid since each valid word has to satisfy all constraints. You might think that it might be impossible to satisfy all constraints, but note that there are $n + c$ code bits, and only c constraints. This implies that there are n degrees of freedom. Think of this as having $n + c$ variables and c equations. This is why we can encode inputs of n bits.

Finally the edges between the code bits and the parity check constraints are selected so that every code bit has some fixed reasonably small degree k. Furthermore the edges are selected so that any small subset of code bits on the left do not share very many parity check constraints on the right. This notion of not sharing too many parity check constraints can be formalized by using so called *expander graphs*. Expander graphs are beyond the scope of this class, but are basically graphs in which any small enough subset of vertices X must have a neighborhood size that is a constant factor larger than $|X|$. They have significant importance in many areas of theoretical computer science.

We note that the term "low-density parity check" comes from the use of the parity check constraints and the fact that the vertices have low degree.

Now imagine that we start with a valid codeword that satisfy all the parity constraints and we send it across a noisy channel. Some of the bits will be corrupted either being too noisy to read or flipped. Now when we receive the message we can detect errors since we either can't read them or if flipped we will notice that some of the constraints are no longer satisfied. What is interesting is that we will not only be able to determine which bits are bad, but we will be able to fix the errors. In the following discussion we say that a constraint is *violated* if it is not satisfied (i.e. has odd instead of even parity).

We just consider the case when bits are flipped. In this case we don't even immediately know what bits are bad, but we will notice that some constraints are violated. Our goal is to determine which bits to flip back so that all constraints are satisfied again. Here is a simple rule for deciding which bits to flip. Each code bit on the left counts how many of its neighboring parity constraints are violated. We say a code bit is *critical* if more than half of its neighbors are violated. Now simply flip bits that are critical—we can either do this sequentially or in parallel. Repeat this process until no more constraints are violated. It turns out that under certain conditions this techniques is guaranteed to converge on the original code word. In particular the conditions are:

1. the number of errors is less than $c/4$.
2. only flip two bits in parallel if they don't share a constraint
3. the bipartite graph has good expansion (theory beyond the scope of this class)

Exercise 1. *Argue that if we only flip one bit at a time, the number of violated parity constraints goes down on each step.*

There is much more to say about error correcting codes and even LDPC codes. We did not explain how to generate a codeword from the message or the inverse of generating a message from a codeword. This can be done with some matrix operations, and is not that hard, but a bit tedious. We also did not explain what it means to be an expander graph and prove that an expander graph gives us the guarantees about convergence. We encourage you to take more advanced theory courses to figure this one out. The main purpose of the example is to give a widely used real-world example of the use of bipartite graphs.

3 Set Cover

An important problems in both theory and practice is the set cover problem. The problem is given n sets of elements for which the union of all sets is U, determine the smallest number of these sets for which the union is still U. For example we might have the sets:

$$\begin{array}{l}\{a, c, d\}\\ \{a, f\}\\ \{a, b, d, e\}\\ \{b, e, f\}\\ \{d, e\}\end{array}$$

which gives $U = \{a, b, c, d, e, f\}$, and the smallest number of sets that covers U is two. More formally for a set of sets $\mathcal{S}$ for which the *universe* is $\mathcal{U} = \bigcup_{s \in \mathcal{S}} s$, we say that C is a *cover* if $\bigcup_{s \in C} s = \mathcal{U}$. The set cover problem is to find a cover of minimum size. There is also a weighted version in which each element of S has a weight and one wants to find the a cover that minimizes the sum of these weights. Unfortunately, both versions of set cover are NP-complete so there is probably no polynomial work algorithm to solve them exactly. However, set cover and variants are still widely used in practice. Often approximation algorithms are used to solve the problems non-optimally. Other times the problems can be solved optimally, especially if there is some structure to the instance that makes it easy.

Various forms of set cover are widely used by businesses to allocated resources. For example consider the problem of deciding where to put distribution centers. Sunbucks, the well known tea shop, decides it needs a distribution center within 50 miles of every one of its stores. After searching for real estate on the web it comes up with n possible locations where to put a distribution center. Each location covers some number of stores. It wants to select some subset of those locations that cover all its stores. Of course it wants to spend as little as possible so it want to minimize the number of such centers. Set cover is also used for crew scheduling on airlines: the sets correspond to multihop cycles a crew member can take, and the elements are the flight legs that need to be covered.

The set location problem can be viewed as a bipartite graph $G = (V, U, E)$. We can put the sets on the left (V) and the elements on the right (U), and an edge from v to u if set v includes element u. For example the above sets would be.

A cover is then any set of vertices on the left whose neighborhood covers all of U, i.e. $V' \subset V, s.t. N(G, V') = U$. Such a view can be helpful in developing algorithms for the problem.

We now discuss one such algorithm. It is a greedy approximation algorithm. It typically does not find the optimal solution, but is guaranteed to find a solution that is within a factor of $\ln n$ of optimal. It was one of the earliest approximation algorithms for which a provable bound on the quality was shown. Furthermore it has since been shown that unless P = NP we cannot do any better than this in polynomial time. The algorithm is very simple:

```
fun greedySetCover$(V, U, E)$ =
if $|E| = 0$ then $\{\}$
else
  let
    % select a vertex v in V with the largest neighborhood
    val $v = \mathtt{argmax}_{v \in V} |N(v)|$
    % remove v and N(v) from G
    val $(V', U') = (V \setminus \{v\}, U \setminus N(v))$
    % remove unconnected edges
    val $E' = \{(u, v) \in E \mid v \in V' \vee u \in U'\}$
  in
    $\{v\} \cup$ greedySetCover$(V', U', E')$
  end
```

If described in terms of sets, roughly, on each step this algorithm selects the set that covers the most remaining elements, removes this set and all the elements it covers, and repeats until there are no more elements left.

What is the cost of this algorithm as described? If we naively search for the vertex with the largest neighborhood then each step will require $O(m)$ work, where $m = |E|$. There will be as many steps as sets we end up picking. By being smarter and using a priority queue, the algorithm can be implemented to run with $O(m \log n)$ work. The algorithm is inherently sequential, but there are parallel variants of the algorithm that sacrifice some small constant factor in the approximation ratio.

So how good is the solution given by this algorithm. It turns out that it is not very hard to show that it guarantees a solution that is within logarithmic factor of optimal.

Theorem 3.1. *Given a set cover instance $G = (V, U, E)$, let k be the number of sets needed by the optimal set cover solution for G, then $|\texttt{greedySetCover}(V, U, E)| \leq k(1 + \ln |U|)$.*

We give a proof of this theorem below although this wasn't covered in class.

Proof. To bound the number of sets used in our solution, we write a recurrence for the number of recursive calls to `greedySetCover` and then solve it. Since each call adds one element to the result, this bounds the size of the solution. Consider a step of the algorithm in which $|U| = n$. The largest set (neighborhood of a vertex in V) has to be of size at least n/k. This is because there is a solution using at most k sets (the optimal one), so on average, the size of the sets in this solution is n/k and hence, the largest has to be at least n/k. Therefore we can write the following recurrence based on n for the number of rounds.

$$\begin{aligned} R(n) &\leq R\left(n - \frac{n}{k}\right) + 1 \\ R(1) &= 1 \end{aligned}$$

To solve this recurrence, we note that each round reduces the size of the size of the universe by a factor of $(1 - \frac{1}{k})$, so after r rounds, the number of elements in the universe is at most

$$n\left(1 - \frac{1}{k}\right)^r$$

Therefore, the number of rounds r^* to reduce the size to below 1 can be determined by finding the smallest r^* such that

$$n\left(1 - \frac{1}{k}\right)^r < 1$$

To solve this, we apply the super useful inequality $1 + x \leq \exp(x)$, so we have $n\left(1 - \frac{1}{k}\right)^r \leq ne^{-r/k}$. Since $r = k(\ln n + 1)$ gives that $n \exp(-\frac{1}{k} \cdot k(\ln n + 1)) < 1$, we know that $r^* \leq r = k(\ln +1)$. This allows us to conclude that $|\texttt{greedySetCover}(V, U, E)| = R(|U|) \leq k(1 + \ln |U|)$ and we have our proof. □

Lecture 17 — Sparse Matrices

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — October 25, 2011

Today:
- Vector and Matrix Review
- Sparse Matrices and Graphs
- Pagerank

1 Vectors and Matrices

Vectors and matrices are fundamental data types in linear algebra. They have a long and rich history of applications. The first example, which dates back to between 300 BC and AD 200, appears in the Chinese text *Jiu Zhang Suan Shu*, which discusses the use of matrix methods to solve a system of linear equations. Indeed, many believe matrices and vectors were initially invented to express complex sets of equations and operations in a compact form. In Physics, vectors are often used to represent displacements (or velocities) and matrices to represent transformations on these displacements, such as rotations. More generally, however, vectors and matrices can be used to represent a wide variety of physical and virtual data. As we will see, matrices can be used to represent the web graph, social networks, or large scale physical simulations. In such applications, the number of entries in the vectors and matrices can easily be thousands, millions, billions, or more.

In this class, we are most interested in such large vectors and matrices, and not the small 3×3 matrices that are involved in transformations in 3-d space. In such usage, it is typical that most of the entries in a matrix are zero. These matrices with a large number of zero entries are referred to as *sparse matrices*. Sparse matrices are important because with the right representations, they can be much more efficient to manipulate than the equivalently-sized dense matrices. Sparse matrices are also interesting because they are closely related to graphs.

Let's consider the data type specifications for vectors and matrices. Here, we only consider a minimal interface. A full library would include many more functions, but the ones given are sufficient for our purposes. Vectors and matrices are made up of elements that support addition and multiplication. For now, we can assume elements are real numbers and will use $\cdot$ to indicate multiplication, but we will extend this later to allow elements from an arbitrary ring or field. We use $\mathcal{V}$ to indicate the type of a vector, $\mathcal{A}$ the type of a matrix and $\mathcal{E}$ for the type of the elements. By convention, we use bold face upper and lower case letters to denote matrices and vectors, respectively, and lowercase letters without bold face for element. A vector is a length-n sequence of elements

$$\mathbf{x} = \langle\, x_1, x_2, \ldots, x_n \,\rangle\,,\ \text{where } x_i \text{ belongs to an arbitrary ring or field } \mathbb{F}.$$

The vectors support the following functions:[1]

[1]In some contexts it can be important to distinguish between row and column vectors, but we won't make that distinction here.

function	**type**	**semantics**	**requires**
$\mathit{sub}(\mathbf{x}, i)$	$\mathcal{V} \times \mathit{int} \to \mathcal{E}$	$\mathbf{x}_i$	$1 \leq i \leq \|\mathbf{x}\|$
$\mathit{dotProd}(\mathbf{x}, \mathbf{y})$	$\mathcal{V} \times \mathcal{V} \to \mathcal{E}$	$\sum_{i=1}^{\|\mathbf{x}\|} A_i \cdot B_i$	$\|A\| = \|B\|$
$\mathit{scale}(c, \mathbf{x})$	$\mathcal{E} \times \mathcal{V} \to \mathcal{V}$	$\langle\, c \cdot x : x \in \mathbf{x} \,\rangle$	
$\mathit{add}(\mathbf{x}, \mathbf{y})$	$\mathcal{V} \times \mathcal{V} \to \mathcal{V}$	$\langle\, x + y : x \in \mathbf{x}, y \in \mathbf{y} \,\rangle$	$\|\mathbf{x}\| = \|\mathbf{y}\|$

A matrix is an $n \times m$ array of elements

$$\mathbf{A} = \begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix}.$$

Thus, a matrix can be viewed as a collection of n length-m row vectors or a collection of m length-n column vectors. Our matrices support the following functions:

function	**type**	**semantics**	**where**
$\mathit{sub}(\mathbf{A}, i, j)$	$\mathcal{A} \times \mathcal{I} \times \mathcal{I} \to \mathcal{E}$	a_{ij}	$1 \leq i \leq \|\mathbf{A}\|$
$\mathit{mmmult}(\mathbf{A}, \mathbf{B})$	$\mathcal{A} \times \mathcal{A} \to \mathcal{A}$	$\{(i,j) \mapsto \sum_{k=1}^{l} a_{ik} \cdot b_{kj} : (i,j) \in \mathit{Idx}(n,m)\}$	$\|\mathbf{A}\| = n \times l$, $\|\mathbf{B}\| = l \times m$
$\mathit{mvmult}(\mathbf{A}, \mathbf{x})$	$\mathcal{A} \times \mathcal{V} \to \mathcal{V}$	$\left\{i \mapsto \sum_{j=1}^{m} a_{ij} \cdot x_j : i \in \{1, \ldots, n\}\right\}$	$\|\mathbf{A}\| = n \times m$, $\|\mathbf{x}\| = m$
$\mathit{scale}(c, \mathbf{A})$	$\mathcal{E} \times \mathcal{A} \to \mathcal{A}$	$\{(i,j) \mapsto c \cdot a_{ij} : (i,j) \in \mathit{Dom}(\mathbf{A})\}$	
$\mathit{add}(\mathbf{A}, \mathbf{B})$	$\mathcal{A} \times \mathcal{A} \to \mathcal{A}$	$\{(i,j) \mapsto a_{ij} + b_{ij} : (i,j) \in \mathit{Dom}(\mathbf{A})\}$	$\|\mathbf{A}\| = \|\mathbf{B}\|$
$\mathit{transpose}(\mathbf{A})$	$\mathcal{A} \to \mathcal{A}$	$\{(j,i) \mapsto a_{ij} : (i,j) \in \mathit{Dom}(\mathbf{A})\}$	

where $\mathit{Idx}(n,m)$ means the index set $\{(i,j) : i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\}\}$, and $\mathit{Dom}(A)$ for an $n \times m$ matrix means $\mathit{Idx}(n,m)$.

It can sometimes be hard to remember how the indices are combined in matrix multiply. After all, the way it works is completely by convention and therefore arbitrary. One way to remember it is to note that when multiplying $\mathbf{A} \times \mathbf{B}$ ($\mathit{mult}(\mathbf{A}, \mathbf{B})$), one takes a dot-product of every row of $\mathbf{A}$ with every column of $\mathbf{B}$. Think of it as a T on its side, as in ⊣, where the the horizontal part represents a row on the left, and the vertical a column on the right. If you can't remember which way the T faces, then think of driving from left to right until you hit the T intersection. Also note that for two matrices with dimension $n \times \ell$ and $\ell \times m$, the output dimension is $n \times m$, so that the ℓ's cancel. This allows one to figure out how the dot products are placed in the result matrices.

The only operations used on the elements are addition and multiplication. In certain situations, it is useful to generalize these to other operations. For the properties of matrices to be maintained, what we need is that the elements $\mathbb{E}$ and functions $+$ and $\cdot$ with certain properties. Consider the tuple $(\mathbb{E}, +, \cdot)$ with the following properties:

- Closure of $\mathbb{E}$ under addition and multiplication: if $a, b \in \mathbb{E}$, then $a + b \in \mathbb{E}$ and $a \cdot b \in \mathbb{E}$.

- Associativity of addition and multiplication: if $a, b, c \in \mathbb{E}$, then $(a + b) + c = a + (b + c)$ and $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
- Identity for addition and multiplication: there exist an additive identity $I_+ \in \mathbb{E}$ and a multiplicative identity $I_.$ such that for all $a \in \mathbb{E}$, $I_+ + a = a = a + I_+$, $I_. \cdot a = a = a \cdot I_.$.
- Commutativity of addition: $a + b = b + a$
- Inverse for addition: $-a$
- Distributivity of multiplication over addition $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.

These rules together define an a *ring* $(\mathbb{E}, +, \cdot)$. Indeed, a ring can be thought of as another abstract data type.

In algorithms and applications, there are many uses of rings other than addition and multiplication over numbers. One that often comes up is having $+$ be minimum, and $\cdot$ be addition. As will be discussed in recitation, this can be used to implement the Bellman-Ford algorithm.

2 Sparse Matrices

As mentioned before, most often in practice when dealing with large matrices, the matrices are sparse, i.e., only a small fraction of the matrix elements are nonzero (for general rings, zero is I_+). When implementing matrix routines that will operate on sparse matrices, representing all the zero elements is both a waste of time and space. Instead, we can store just the nonzero elements. Such a representation of matrices is called a *sparse matrix representation*. Advances in sparse matrix routines over the past ten to twenty years is possibly the most important advance with regards to efficiency in large scale simulations in scientific, economic, and business applications.

We often use $nz(\mathbf{A})$ to denote the number of nonzero elements in a matrix. For an $n \times m$ matrix, clearly this cannot be larger than $n \times m$.

How do we represent a sparse matrix or sparse vector? One of the most standard ways is the so-called compressed sparse row (CSR) representation. We will consider two forms of this representation, one using sequences and one sets. Consider the matrix

$$\mathbf{A} = \begin{pmatrix} a & 0 & 0 & b & 0 \\ 0 & c & d & 0 & 0 \\ 0 & 0 & e & 0 & 0 \\ f & 0 & g & 0 & h \\ 0 & i & 0 & 0 & j \end{pmatrix}$$

It can be presented as a sequence of sequences, as in:

$$\langle\, \langle\, (0, a), (3, b) \,\rangle\, , \langle\, (1, c), (2, d) \,\rangle\, , \langle\, (2, e) \,\rangle\, , \langle\, (0, f), (2, g), (4, h) \,\rangle\, , \langle\, (1, i), (4, j) \,\rangle\, \rangle$$

In this representation, each subsequence represents a row of the matrix. Each of these rows contains only the non-zero elements each of which is stored as the non-zero value along with the column index in which it belongs. Hence the name compressed sparse row. Now lets consider how we do a vector matrix multiply using this representation.

```
1  fun mvmult(A, x) = $\left\langle \sum_{(j,v)\in \mathbf{r}} \mathbf{x}[j] * v : \mathbf{r} \in \mathbf{A} \right\rangle$
```

For each non-zero element within each row, the algorithm reads the appropriate value from the vector and multiplies it the value of the element. These are then summed up across the rows. For example, let's consider the following matrix-vector product:

$$\underbrace{\begin{pmatrix} 3 & 0 & 0 & -1 \\ 0 & -2 & 2 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 0 & -3 & 1 \end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix} 2 \\ 1 \\ 3 \\ 1 \end{pmatrix}}_{\mathbf{x}}$$

which is represented as

```
A = <<(0,3),(3,-1)>,<(1,-2),(2,2)>,<(2,1)>,<(0,2),(2,-3),(3,1)>

x = < 2, 1, 3, 1 >
```

This is perhaps the most common way to implement sparse matrix vector multiplication in practice. If we assume an array implementation of sequences then this routine will take $O(nz(\mathbf{A}))$ work and $O(\log n)$ span. To calculate the work we note that we only multiply each non-zero element once and each non-zero element is only involved in one reduction. For the span we note that the sum requires a reduction but otherwise everything

It is however not well suited when the vector is sparse as well. Consider the two vectors:

$$\mathbf{x} = \langle\, 0, 1, 0, 0, -2, 0, 0, 0, -1, 0 \,\rangle$$

$$\mathbf{y} = \langle\, -3, 0, 1, 0, 1, 0, 0, 2, 0, 0 \,\rangle$$

Which we would represent as a compressed "row" as

$$\mathbf{x} = \langle (1, 1), (4, -2), (8, -1) \rangle$$

$$\mathbf{y} = \langle (0, -3), (2, 1), (4, 1), (7, 2) \rangle$$

Now lets say we want to implement the vector operations, such as *dotp* and *add*. What do we get? How do we do this.

One easy way to do this is to present sparse vectors as $\mathcal{E}$ tables. We then have, e.g.

$$\mathbf{x} = \{1 \mapsto 1, 4 \mapsto -2, 8 \mapsto -1\}$$

Now we can easily write vector addition and and dot product

```
fun add(x, y) = Table.merge (fn (x, y) ⇒ x + y) (x, y)

fun dotp(x, y) =
  reduce + 0 (Table.merge (fn (x, y) ⇒ x · y) (x, y))
```

3 Relation to Graphs

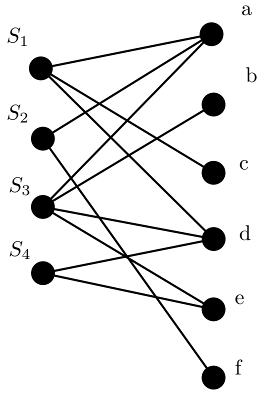
Sparse matrices are closely related to graphs. In particular, an $n \times m$ matrix M can be viewed as a bipartite graph with n vertices V on the left and m on vertices U on the right—i.e. the rows correspond to the vertices U and the columns to the vertices V. We place a weighted edge from vertex v_i on the the left to u_i on the right if $M_{ij} \neq 0$. If $n = m$ (i.e., the matrix is square), then we can view the matrix as a graph from vertices back to themselves—i.e., the rows and columns correspond to the same set of vertices V.

For example, in the last lecture we discussed error correcting codes and how they can be represented as a bipartite graph with the code bits on the left, the check constraints on the right, and edges between the two. Such an error correcting code could equally easily be represented as a matrix. For example here is a matrix for a code corresponding to a codeword of length 6 with 4 parity constraints.

$$\mathbf{B} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

Similarly we could represent the set cover problem as a matrix where the rows correspond to sets and the columns to elements. For example, the sets

{a, c, d}
{a, f}
{a, b, d, e}
{b, e, f}
{d, e}



correspond to the matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

where the rows correspond to the five sets, and the columns correspond to the elements (a, b, c, d, e, f). We can now describe the set cover problem as a matrix problem. In particular for an $n \times m$ binary matrix A, the problem is to

minimize $x \cdot \mathbf{1}$
such that $xA \geq \mathbf{1}$ and $x_i \in \{0, 1\}$

where $\mathbf{1}$ is a vector of all ones. Here the vector x is a binary vector representing which sets are selected. Minimizing the dot product $x \cdot \mathbf{1}$ therefore minimizes the number of sets that are selected. The constraint $xA \geq \mathbf{1}$ makes sure that every element is covered at least once.

The connection between graphs and matrices has been exploited heavily over the past decade in both directions—graph algorithms have been applied to improve matrix computations especially with regards to sparse matrices, and matrix algebra has been used to analyze graphs. For example, the best algorithms for doing Gaussian elimination on sparse matrices are based on graph properties and algorithms. In the other direction, we will see that analyzing random walks on graphs is best dealt with using matrix operations.

We note, however, there are some differences between matrices and graphs. Firstly, the vertices of a graph are not necessarily indexed from 1 to n and 1 to m, they are unordered sets. This means that certain techniques such as graph contractions, are not very naturally described as matrix routines. Secondly, matrices assume specific operations while graphs are really only defined in terms of the vertices and edges.

4 PageRank and Iterative Linear System Solving

We now consider a simple application of matrix-vector multiply for the so-called PageRank problem. PageRank is used by Google to give a score to all pages on the web based on a method for predicting how important the pages are. The PageRank problem was introduced in a paper by Brin, Page, Motwani, and Winograd. Brin and Page were graduate students in CS at Stanford and went on to form Google. Motwani and Winograd were professors at Stanford. Motwani is an author of one of probably the most widely used textbook on Randomized Algorithms (do a search on "randomized algorithms" on google and see how well it is ranked). When Google was started, Stanford received 1.8 million shares of google stock in exchange for patent PageRank patents. It later sold these for $336 million.

To understand PageRank, consider the following process for visiting pages on the web, which we will refer to as WebSurf.

0. Start at some arbitrary page
1. For a very long time,

a. with probability ε jump to a random page
b. otherwise with equal probability visit one of the outlinks of the current page

The question is after a very long time, what is the probability that the process is at a given page. The idea of PageRank is that this probability is a reasonable measure of the importance of a page. In particular, if a page p has many pointers to it, then it will end up with a high rank. Furthermore, if the pages that point to p also have high ranks, then they will effectively improve the rank of p.

It turns out that the probability of being at a page can be calculated reasonably easily and efficiently using matrix-vector multiplication. In particular, we can use a matrix to represent the transition probabilities described by the WebSurf process. In particular, consider a directed web graph G with n vertices (pages), and denote the out degree of page i as d_i. We define an $n \times n$ adjacency matrix $\mathbf{A}$ such that $\mathbf{A}_{ij} = 1/d_i$ if there is an edge from i to j and 0 otherwise. Also let $\mathbf{J}$ be an $n \times n$ matrix of all ones. Now consider the matrix:

$$\mathbf{T} = \frac{\varepsilon}{n}\mathbf{J} + (1 - \varepsilon)\, vA.$$

The claim is that entry $\mathbf{T}_{ij}$ is exactly the probability of going from vertex i to vertex j in process WebSurf. Now to calculate the probability distribution, we use the following simple algorithm.

```
1   fun pageRank($\mathbf{T}$) =
2   let
3      fun pageRank'($p$) =
4         let val $p' = \mathbf{T}p$
5         in
6            if $\|p - p'\| <$ some threshold
7            then $p'$
8            else pageRank'($p'$)
9         end
10  in
11     pageRank'($\frac{1}{n}[1, \ldots, 1]$)
12  end
```

In the code, the vector norm $\|x\|$ stands for the Euclidean norm (or ℓ_2 norm) $\sqrt{x^\top x}$. The vector $\frac{1}{n}[1, \ldots, 1]$ is an initial "guess" at the probabilities. This routine converges on the actual probability distribution we desire and in practice does not take very many rounds (e.g. 10s or 100s). Note that the only real computations are the matrix vector multiply on line 4 and the calculation of $\|p - p'\|$. The cost is dominated by the matrix vector multiply.

Exercise 1. *Note that the matrix T is dense since U is dense. Explain how to computer Tp in $O(|E|)$ work, where E is the number of edges in the web graph.*

Lecture 18 — Binary Search Trees (Split, Join and Treaps)

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — October 27, 2011

Today:

- Binary Search Trees
- Split and Join
- Treaps

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). What's common among search trees is that they store a collection of elements in a tree structure and use values (most often called keys) in the internal nodes to navigate in search of these elements. A *binary search tree* (BST) is a search tree in which every node in the tree has at most two children.

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. You should convince yourself that it would be impossible to maintain a perfectly balanced tree while allowing efficient (e.g. $O(\log n)$) updates.

There are dozens of balanced search trees that have been suggested over the years, dating back at least to AVL trees in 1962. The trees mostly differ in how they maintain approximate 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). Here we 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.
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. *Splay trees.*[1] Binary search trees that are only balanced in the amortized sense (i.e. on average across multiple operations).
6. *Weight balanced trees.* Trees in which the children all have the same size. These are most typically binary, but can also have other branching factors.
7. *Treaps.* A binary search tree that uses random priorities associated with every element to keep balance.
8. *Random search trees.* A variant on treaps in which priorities are not used, but random decisions are made with probabilities based on tree sizes.
9. *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.

Traditional 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] However, insert and delete 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.

2 BST Basic Operations

In this class, we will limit ourselves to binary search trees. We assume 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.

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 now consider just two functions on BSTs that are useful for building up other functions, such as search, insert and delete, but also many other useful functions such as intersection and union on sets.

$\mathtt{split}(T, k) : \mathtt{BST} \times \mathtt{key} \to \mathtt{BST} \times (\mathtt{data\ option}) \times \mathtt{BST}$

Given a BST T and key k, `split` divides T into two BSTs, one consisting of all the keys from T 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 $\mathtt{SOME}(d)$, and otherwise it returns `NONE`.

$\mathtt{join}(L, m, R) : \mathtt{BST} \times (\mathtt{key} \times \mathtt{data})\ \mathtt{option} \times \mathtt{BST} \to \mathtt{BST}$

This takes a left BST L, an optional middle key-data pair m, and a right BST R. It requires that all

[1]Splay trees were invented back in 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.

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 n 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:

$\mathtt{expose}(T) : \mathtt{BST} \to (\mathtt{BST} \times \mathtt{BST} \times \mathtt{key} \times \mathtt{data})\ \mathtt{option}$
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 easily implement search, insert and delete:

```
fun search T k =
let
   val (_, v, _) = split(T, k)
in
   v
end
```

```
fun insert T (k, v) =
let
   val (L, v', R) = split(T, k)
in
   join(L, SOME(k, v), R)
end
```

```
fun delete T k =
let
   val (L, _, R) = split(T, k)
in
   join(L, NONE, R)
end
```

Exercise 1. *Write a version of* `insert` *that takes a function* $f : \mathtt{data} \times \mathtt{data}$ *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.

Now we consider a more interesting operation, taking the union of two BSTs. Note that this is different than `join` since we do not require that all the keys in one appear after the keys in the other.

```
fun $union(T_1, T_2) =$
   case $expose(T_1)$ of
      $NONE \Rightarrow T_2$
    | $SOME(L_1, R_1, k_1, v_1) \Rightarrow$
      let
         val $(L_2, v_2, R_2) = split(T_2, k_1)$
      in
         $join(union(L_1, L_2),\ SOME(k_1, v_1),\ union(R_1, R_2))$
      end
```

This version returns the value from T_1 if a key appears in both BSTs. We will analyze the cost of this algorithm for particular balanced trees later. We note, however, that as written the code only matches our desired bounds if $|T_1| \leq |T_2|$.

The code for intersection is quite similar.

3 Implementing Split and Join on A Plain 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 have:

```
datatype BST = Leaf | Node of (Tree * Tree * key * data)
```

```
fun $split(T, k) =$
   case $(T)$ of
      $Leaf \Rightarrow (Leaf, NONE, Leaf)$
    | $Node(L, R, k', v) \Rightarrow$
         case $compare(k, k')$ of
            $LESS \Rightarrow$
               let val $(L', r, R') = split(L, k)$
               in $(L', r, Node(R', R, k', v))$ end
            $EQUAL \Rightarrow (L, SOME(v), R)$
            $GREATER \Rightarrow$
               let val $(L', r, R') = split(R, k)$
               in $(Node(L, L', k', v), r, R')$ end
```

```
fun $join(T_1, m, T_2) =$
   case $m$ of
      $SOME(k, v) \Rightarrow Node(T_1, T_2, k, v)$
    | $NONE \Rightarrow$
         case $T_2$ of
            $Leaf \Rightarrow T_1$
          | $Node(L, R, k, v) \Rightarrow Node(L, join(R, NONE, T_2), k, v))$
```

4 Treaps

A Treap (tree + heap) is a randomized BST that maintains balance in a probabilistic way. In a few lectures, we will show that with high probability, a treap with n keys will have depth $O(\log n)$. 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.

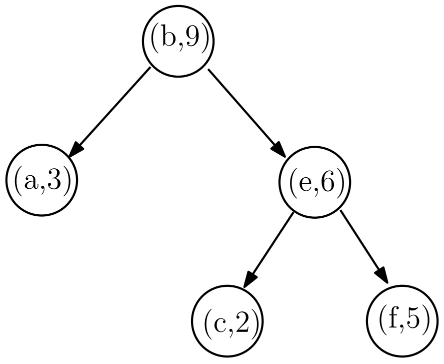
The nodes in a treap must satisfy two properties:

- **BST Property.** Their keys satisfy the BST property (i.e., keys are stored in-order in the tree).
- **Heap Property.** The associated priority 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-value 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 key-value pairs, there is exactly one treap T containing the key-value 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. 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. $\square$

Parting thoughts: Notice that implementing split is identical to the plain BST version. How would you implement join?

Lecture 19 — Union, Quick Sort, and Treaps

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — Nov 1, 2011

Today:
- Quick Sort Revisited
- Split and Join
- Treaps

1 Quick Review: Binary Search Trees

For a quick recap, recall that in the last lecture, we were talking about binary search trees. 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.

We then started describing a particular balanced tree, called Treaps (Trees + Heaps). The idea of a Treap is that we assign to each key a randomized priority and then maintain a tree such that

1. the keys satisfy the binary search tree (BST) property, and
2. the priorities satisfy the heap property (i.e. the priority of every node is greater than the priority of its children).

Today: before getting back to Treaps, we will discuss two other related topics. The first is the analysis of the cost of *`union`* based on the code we gave last time. Notice that the cost for *`intersection`*and set difference is the same. The second is an analysis of the quick sort algorithm. It may seem odd to analyze quick sort in the middle of a lecture on balanced search trees, but as we will see, the analysis of quick sort is the same as the analysis of Treaps. We basically will reduce the analysis of Treaps to the analysis of quick sort.

2 Cost of Union

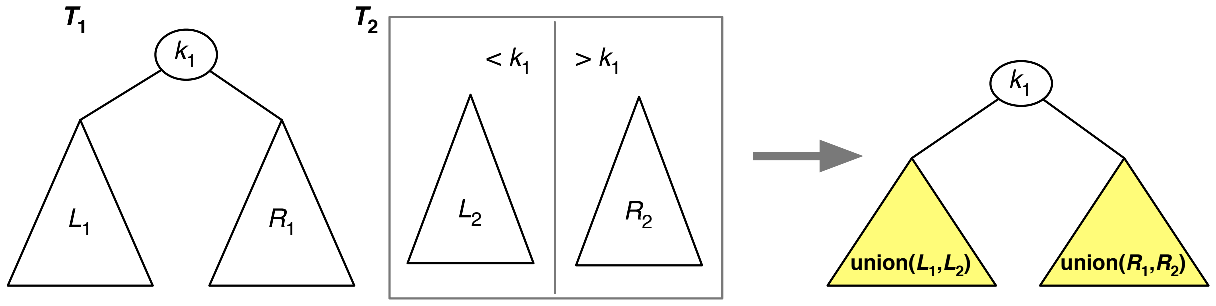
In the 15-210 library, *union* and similar functions (e.g., *intersection* and *difference* on sets and *merge*, *extract* and *erase* on tables) have $O(m \log(n/m))$ work, where m is the length of the shorter input and n the length of the longer one. At first glance, this may seem like a strange bound, but we will see how it falls out very naturally from the union code.

To analyze this, first, we note that the work and span of *split* and *join* is proportional to the depth of the input tree(s). This is simply because the code just traverses a path in the tree (or trees for join). Therefore, if the trees are reasonably balanced and have depth $O(\log n)$, then the work and span of both *split* and *join* is $O(\log n)$. Indeed, most balanced trees have $O(\log n)$ depth. You have already argued this for red-black trees, and we will soon argue it for Treaps.

Let's recall the basic structure of $\texttt{union}(T_1, T_2)$.

- 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.
- Recursively find $L_u = \texttt{union}(L_1, L_2)$ and $R_u = \texttt{union}(R_1, R_2)$.
- Now $\texttt{join}(L_u, k_1, R_u)$.

Pictorially, the process looks like this:

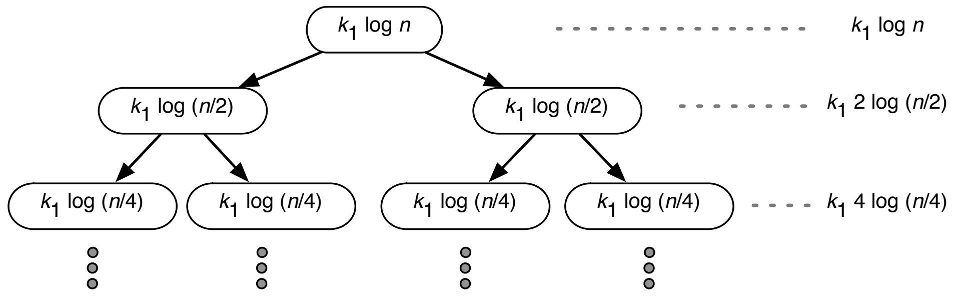


Note that each call to *union* makes one call to *split* and one to *join* each which take $O(\log n)$ work, where n is the size of T_2. We assume that T_1 is the smaller tree with size m. For staters, we'll make the following assumptions to ease the analysis:

1. let's assume that T_1 it is perfectly balanced, and
2. each time a key from T_1 splits T_2, it splits it exactly in half.

Now if we draw the recursion tree[1], we obtain the following:

[1]If we want to write out a work recurrence, it will be $W(m, n) = W(m/2, n/2) + \Theta(\log n)$



Is this tree root or leaf dominated, or evenly sized? And how many levels will it have? It is not hard to see that this tree is dominated at the leaves. In fact, what's happening is that when we get to the bottom level, each leaf in T_1 has to split a subtree of T_2 of size n/m. This takes $O(\log(\frac{n}{m}))$ work. Since there are $O(m)$ such splits, the total work at the leaves is $O(m\log(\frac{n}{m}))$. Furthermore, the work going up the tree decreases geometrically.

To bound this more formally, since T_1 has m keys and it is split exactly in half, it will have $\log_2 m$ levels, so if we start counting from level 0 at the root, we have that the i-th level has 2^i nodes, each costing $\Theta(\log(n/2^i))$. Therefore, for example, the bottom level will have $\frac{m}{2}$ nodes—and the whole bottom level will cost

$$k_1 \cdot \frac{m}{2}\log\left(n/2^{\log m - 1}\right) = k_1 \cdot \frac{m}{2}\left(\log n - \log(m/2)\right).$$

Since the tree is dominated at the leaves, the total work will be $O(m\log(\frac{n}{m}))$, as desired. Hence, if the trees satisfy our assumptions, we have that `union` runs in $O(m\log(\frac{n}{m}))$.

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 it is not difficult to show that the recursion tree remains leaf dominated. So, once again, it suffices to consider the bottom level of recursion. Since T_1 is balanced, there will be $k := m/2$ subtrees of T_2 that need to be split. Suppose these subtrees have sizes $n_1, n_2, \ldots, n_k$, where $\sum_{i=1}^{k} n_i = n$ since the subtrees are a partition of the original tree T_2. Therefore, the total cost of splitting these subtrees is

$$\sum_{i=1}^{k} k_1 \log(n_i) \quad \le \quad \sum_{i=1}^{k} k_1 \log(n/k),$$

where we use the fact that the logarithm function is concave[2] This shows that the total work is $O(m\log(\frac{n}{m}))$.

Still, in actuality, T_1 doesn't have to be perfectly balanced as we assumed. Although this, too, is not too hard to show, 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, this means that `union` can be implemented in $O(m\log(n/m))$ work and span $O(\log n)$. The same holds for the other similar operations (e.g. `intersection`).

3 Quick Sort

We now turn to analyzing quick sort. As mentioned earlier, the motivation for analyzing this now is that this analysis is a nice segue into the Treap analysis. The argument here is essentially the same as the

[2]Technically, we're applying a varaint of the so-called Jensen's inequality.

analysis we needed to show that the expected depth of a node in a Treap is $O(\log n)$. Of course, being able to analyze randomized quick sort is very important on its own, and the analysis gives an elegant example of randomized analysis.

The randomized quick sort algorithm is as follows:

```
fun quickSort(S) =
  if |S| ≤ 1 then S
  else let
      val p = select a random key from S
      val S_1 = ⟨ s ∈ S | s < p ⟩
      val S_2 = ⟨ s ∈ S | s = p ⟩
      val S_3 = ⟨ s ∈ S | s > p ⟩
    in
      quickSort(S_1) @ S_2 @ quickSort(S_3)
    end
```

For the analysis, we'll consider a completely equivalent algorithm which 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 (sound familiar?). Notice that once the priorities are decided, the algorithm is completely deterministic; you should convince yourself that the two presentations of the algorithm 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

$$X_n \quad = \quad \# \text{ of comparisions } \texttt{quickSort} \text{ makes on input of size } n,$$

our goal is to find an upper bound on $\mathbf{E}[X_n]$ for any input sequence S. For this, we'll consider the final sorted order[3] 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.

Looking closely at the algorithm, we have that if some two elements are compared, one of them has to be a pivot in that call. So, then, the other element will be part of S_1, S_2, or S_3—but the pivot element will be part of S_2, which we don't recurse on. This gives the following observation:

Observation 3.1. *In the quick sort algorithm, if some two elements are compared in a* `quickSort` *call, they will never be compared again in other call.*

Therefore, with these random variables, we can express the total comparsion count X_n as follows:

$$X_n \;\leq\; 3\sum_{i=1}^{n}\sum_{j=i+1}^{n} A_{i,j}$$

[3]Formally, there's a permutation $\pi\colon \{1, \ldots, n\} \to \{1, \ldots, n\}$ between the positions of S and T.

This is because our not-so-optimized quick sort compares each element to a pivot 3 times. By linearity of expectation, we have $\mathbf{E}\left[X_n\right] \leq 3 \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbf{E}\left[A_{i,j}\right]$. Furthermore, since each $A_{i,j}$ is an indicator random variable, $\mathbf{E}\left[A_{i,j}\right] = \mathbf{Pr}\left[A_{i,j} = 1\right]$. Our task therefore comes down to computing the probability that T_i and T_j are compared (i.e., $\mathbf{Pr}\left[A_{i,j} = 1\right]$) and working out the sum.

Computing the probability Pr $[A_{i,j} = 1]$. The crux of the matter is in descibing the event $A_{i,j} = 1$ 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 quick sort algorithm to gather some intutions. 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. With this view, the following observation is not hard to see:

Claim 3.2. *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 contradition. Asssume 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 inbetween. Since there are $j - i + 1$ possible keys inbetween (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,

$$\begin{aligned}
\mathbf{E}\left[A_{i,j}\right] &= \mathbf{Pr}\left[A_{i,j} = 1\right] \\
&= \mathbf{Pr}\left[p_i \text{ or } p_j \text{ is the maximum among } \{p_i, \ldots, p_j\}\right] \\
&= \frac{2}{j - i + 1}.
\end{aligned}$$

Hence, the expected number of comparisons made is

$$\begin{aligned}
\mathbf{E}\left[X_n\right] &\leq 3 \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbf{E}\left[A_{i,j}\right] \\
&= 3 \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} \\
&= 3 \sum_{i=1}^{n} \sum_{k=1}^{n-i} \frac{2}{k + 1} \\
&\leq 3 \sum_{i=1}^{n} H_n \\
&= 3nH_n \in O(n \log n)
\end{aligned}$$

4 Treaps

Let's go back to Treaps. We claim that the split code given in the last lecture for unbalanced trees doesn't need to be modified at all 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).*

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. Here is the algorithm:

```
1   fun $join(T_1, m, T_2) =$
2   let
3      fun $singleton(k, v) = Node(Leaf, Leaf, k, v)$
4      fun $join'(T_1, T_2) =$
5         case $(T_1, T_2)$ of
6            $(Leaf, \_) \Rightarrow T_2$
7          | $(\_, Leaf) \Rightarrow T_1$
8          | $(Node(L_1, R_1, k_1, v_1), Node(L_2, R_2, k_2, v_2)) \Rightarrow$
9               if $(priority(k_1) > priority(k_2))$ then
10                 $Node(L_1, join'(R_1, T_2), k_1, v_1))$
11              else
12                 $Node(join'(T_1, L_2), R_2, k_2, v_2))$
13  in
14        case $m$ of
15           $NONE \Rightarrow join'(T_1, T_2))$
16         | $SOME(k, v) \Rightarrow join'(T_1, join'(singleton(k, v), T_2))$
17  end
```

In the code $join'$ is a version of join that does not take a middle element. 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.

The cost of `split` and `join` on treaps is proportional to the depth of nodes in the tree. 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. The $join(T_1, m, T_2)$ code only traverses the right spine of T_1 or the left spine of T_2. The work and span is therefore proportional to the sum of the length of these spines. Similarly, the `split` operation only traverses the path from the root down to the node being split at. The work and span are proportional to this path length. Therefore, if we can show that the depth of all nodes are at $O(\log n)$ then the work and span of `join` and `split` are $O(\log n)$.

Lecture 20 — Balanced Trees Continued

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — Nov 3, 2011

Today:
- Treaps continued
- Law of large numbers and high probability bounds

1 Treaps Continued

Recall that a treap is a binary search tree in which we associate with each key a random priority. The tree is maintained so that the priorities associated with the keys are in (max) heap order, i.e. the priority at a node is larger than the priorities of both of its children.

We will now analyze the expected depth of a key in the tree. This analysis pretty much the same as the analysis we did for quicksort.

Consider a set of keys K and associated priorities $p : key \rightarrow int$. 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 two positions in this order.

```
| | | | | | | | | | | | | | | | | | | |
          i             j
```

If we calculate the depth starting with zero at the root, the expected depth of a key is equivalent to the number of ancestors in the tree. So we want to know how many ancestor. We use the random indicator variable A_{ij} to indicate that j is an ancestor of i. Now the expected depth can be written as:

$$\mathbf{E}\left[\text{depth of } i \text{ in } T\right] = \sum_{j=1}^{n} \mathbf{E}\left[A_{ij}\right].$$

To analyze A_{ij} lets just consider the $|j-i|+1$ keys and associated priorities from i to j inclusive of both ends. We consider 3 cases:

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?

In the first case j cannot be an ancestor of i since i has a higher priority. In the second case note that i can only be a descendant of j if k is also. This is because in a BST every branch covers a contiguous region, so if i is in the left (or right) branch of j, then k must also be.

```
        o j
       /

   o i    o k
```

But since the priority of k is larger than that of j this cannot be the case so j is not an ancestor. Finally in the third case j is an ancestor of i since 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). Since priorities are selected randomly, this has a chance of $1/(j-i+1)$ and we have $\mathbf{E}\left[A_{ij}\right] = \frac{1}{|j-i|+1}$. 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. Recall from last lecture that the recursion tree for quicksort is identical to the structure of the corresponding treap (assuming the same keys and priorities).

Now we have

$$\begin{aligned}
\mathbf{E}\left[\text{depth of } i \text{ in } T\right] &= \sum_{j=1}^{n} \frac{1}{|j-i|+1} \\
&= \sum_{j=1}^{i-1} \frac{1}{i-j+1} + \sum_{j=i+1}^{n} \frac{1}{j-i+1} \\
&= H_{i-1} + H_{n-i-1} \\
&= O(\log n)
\end{aligned}$$

Exercise 1. *Including constant factors how does the expected depth for the first key compare to the expected depth of the middle ($i = n/2$) key.*

Split and Join on Treaps

As mentioned last week for any binary tree all we need to implement is split and join and these can be used to implement the other operations.

We claim that the split code given in lecture 18 for unbalanced trees does not need to be modified at all for Treaps.

Exercise 2. *Prove that the code for split given in lecture 18 need not be modified.*

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. Here is the algorithm.

```
1  fun $join(T_1, m, T_2) =$
2  let
3    fun $singleton(k, v) = Node(Leaf, Leaf, k, v)$
4    fun $join'(T_1, T_2) =$
5      case $(T_1, T_2)$ of
6        $(Leaf, \_) \Rightarrow T_2$
7      | $(\_, Leaf) \Rightarrow T_1$
8      | $(Node(L_1, R_1, k_1, v_1), Node(L_2, R_2, k_2, v_2)) \Rightarrow$
9          if $(priority(k_1) > priority(k_2))$ then
10           $Node(L_1, join'(R_1, T_2), k_1, v_1)$
11         else
12           $Node(join'(T_1, L_2), R_2, k_2, v_2)$
13 in
14     case $m$ of
15       $NONE \Rightarrow join'(T_1, T_2))$
16     | $SOME(k, v) \Rightarrow join'(T_1, join'(singleton(k, v), T_2))$
17 end
```

In the code $join'$ is a version of join that does not take a middle element. 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.

Theorem 1.1. *For treaps the cost of* $join(T_1, m, T_2)$ *returning* T *and of* $split(T)$ *is* $O(\log |T|)$ *expected work and span.*

Proof. The cost of $split$ is proportional to the depth of the node where we are splitting at. Since the expected depth of a node is $O(\log n)$, the expected cost of split is $O(\log n)$. For $join(T_1, m, T_2)$ note that the code only follows T_1 down the right child and terminates when the right child is empty. Similarly it only follows T_2 down the left child and terminates when it is empty. 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 which is expected $O(\log |T|)$. □

We note that these bounds for $split$ and $join$ give us the $O(m \log(n/m))$ work bounds for $union$ and related functions in expectation.

2 Not So Great Expectations

So far, we have argued about the expected work of quick sort and the expected depth of a node in a treap, but *is this good enough?* Let's say 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).

Fortunately, in many cases we can use expectations to prove much stronger bounds at least when we have independent events. These bounds are what we call high probability bounds, again as mentioned in an earlier lecture. Basically, it says that we guarantee some property with probability very close to 1—in fact, so close that the difference is only inverse polynomially small in the problem size. To get some intuitions for high probability bounds, we'll take a detour into the law of large numbers.

3 The Law of Large Numbers

The *law of large numbers* says that the average of the results of a large number of independent trials converges on the expected value of a single trial. The law of large numbers is not just a law of mathematics, but also a hugely practical law that is relevant to everyday life.

Consider, for example, throwing a six sided die. The expected value is 3.5 ((1+2+3+4+5+6)/6). If we throw it just once it is reasonably likely we will not be close to this value. We might roll 1 or 6, for example, and we certainly will not roll a 3.5, at least not with any dice I've used. If we throw it 10 times it is still reasonably likely that the average is, for example, less than 3. However, if we throw it a million times then it becomes very unlikely the average is less than 3. Can anyone guess what the probability is? Well it is about e^{-1000}. That is a very-very-very small number (way smaller than one over the number of particles in the universe). In fact the probability that the average is less than 3.4 or more than 3.6 is less than e^{-40}, which is still a very-very small number. This indicates that as we throw the dice many times the average converges on the mean.

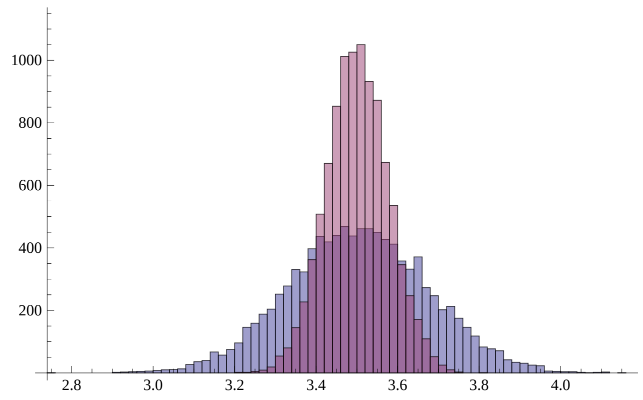


Figure 1: In blue: a $10,000$-trial histogram of the average value of throwing a six-sided die 100 times. In purple: a $10,000$-trial histogram of the average value of throwing a six-sided die 500 times

Another example was the extra credit in assignment 5 where you had to tell us how many trials you had to run to get a reasonable estimate on the expected radius of a random point in a circle.

OK, perhaps throwing dice is not relevant to everyday life, except perhaps for compulsive gamblers, and the radius of random points only to 210 assignments. However, the law of large numbers can be applied to many other things. For example, when investing in stocks, a well-known strategy involves choosing a large number of stocks instead of a single stock to make it much less likely that you will loose big (or gain big). Or, in determining how well a student is doing at Carnegie Mellon: the student might be unlucky and do badly in one class or on one test, but on average across all their classes, their performance will be a good prediction of how well they are understanding or are interested in the material.

The law of large numbers should be intuitive, we hope, but can also be formalized. In particular, for any $\varepsilon > 0$, it[1] states that

$$\lim_{n \to \infty} \mathbf{Pr}\left[|\overline{X}_n - \mu| > \varepsilon\right] = 0\,.$$

where $\mu = E[X]$ is the expected value of X (e.g. 3.5 for a roll of the dice) and $\overline{X}_n$ is the average of the measured value over n independent trials (e.g., the average over n rolls of the dice). This statement says that as n approaches infinity the average over n trials converges exactly to the expected value for a single trial. This statement, however, does not tell us anything about how many trials we need to run before our confidence is high. Who cares what happens at infinity; we will never get there.

Fortunately, there are many theorems applicable in different circumstances that do tell us about the rate of convergence. One that is particularly useful in algorithm analysis is the following bound, known as the Chernoff bound—well, actually, *a* Chernoff bound[2].

Theorem 3.1 (Chernoff Bound)**.** *Given a set of independent random variables $X_1, \ldots, X_n$, each taking a real value between* 0 *and* 1*, with $X = \sum_{i=1}^{n} X_i$ and $\mathbf{E}[X] = \mu$, then for any $\lambda > 0$*

$$\mathbf{Pr}\left[X > (1+\lambda)\mu\right] \;<\; \exp\left(\frac{-\mu\lambda^2}{2+\lambda}\right).$$

The Chernoff bounds are named after Harvard Statistics professor Herman Chernoff—although the very form[3] we're using had been proved in 1924, almost 30 years before Chernoff showed a generalization of it. Proving this is beyond the scope of this course. But we want to make a few remarks about the theorem and tell you how to apply it.

First, let's try to understand what the bound says in words. The factor λ specifies a distance away from the mean. So, for example, by setting $\lambda = .2$, we are asking what the probability is that the actual result is more than 20% larger than the expected result. As we expected, the farther we are away from the mean, the less likely our value will be—this probability, in fact, decays exponentially fast in (roughly) λ^2. Note that this adds to our repertoire of deviation bounds (you have seen Markov's inequality from 15-251).

Second, independence is key in getting this sharp bound. By comparison, Markov's inequality which doesn't require independence can only give $\mathbf{Pr}\left[X > (1+\lambda)\mu\right] < \frac{1}{1+\lambda}$, which is a much weaker bound. As weak as Markov's bound is, it is in certain cases the best we can make of if we can't say anything about independence. Indeed, independence is key here: it makes it unlikely that the random variables will "collude" to pull the mean one way or the other.

[1]This is technically known as the weak law of large numbers; there's another form with a stronger guarantee, known as the strong law of large numbers.

[2]In the literature, Chernoff bounds are more or less a family of inequalities—a generic term for various bounds that deal with the idea that the aggregate of many independent random variables concentrates around the mean.

[3]There is a sharper—and messier—form: for $\lambda > 0$, $\mathbf{Pr}\left[X > (1+\lambda)\mu\right] \le \left(\frac{e^{\lambda}}{(1+\lambda)^{(1+\lambda)}}\right)^{\mu}$.

Third, the Chernoff bound can be seen as a quantitive version of the law of large numbers. At first glance, you might ask: since there is no n in the equation, how this relates to the law of large numbers? The law shows up indirectly since each variable X_i is a number between 0 and 1. Therefore, we have $n \geq \mu$ and hence when μ goes up, n goes up and the probability of being far from the mean goes down, as we would expect from the law of large numbers.

Now let's apply this theorem to determine the probability that if we throw $1,000$ coins that more than 60% come up heads. In this case, $\mu = 500$ and $\lambda = .2$ (since $(1 + .2)500 = 600$). We plug this in and get

$$\begin{aligned} \textbf{Pr}\left[X > 600\right] &< e^{\frac{-500(.2)^2}{2+.2}} \\ &< 0.00012 \end{aligned}$$

If we apply it to $10,000$ coins flips and ask what the probability that more than 60% come up heads we have $\mu = 5,000$ (again, $(1 + 0.2)5000 = 6000$), giving

$$\begin{aligned} \textbf{Pr}\left[X > 6000\right] &< e^{\frac{-5000(.2)^2}{2+0.2}} \\ &< 3.3 \times 10^{-40} \end{aligned}$$

This is a significant decrease in probability, which demonstrates the power of the law of large numbers. Once again, it is important to remember that the Chernoff bounds in particular and the law of large numbers in general are only true if each random variable is independent of the others[4] Consider, for example, the stocks for a set of companies that are all in the same business. These are clearly highly correlated, and hence not independent. Thus, investing in them does not give you the same safety of large numbers as investing in a set of unrelated companies.[5] We also note that the Chernoff bounds are sloppy and the actual bounds are typically much stronger. Since we are interested in upper bounds on the probability, this is OK (our upper bounds are just a bit loose).

4 High Probability Bounds for Quick Sort and Treaps

We now use the Chernoff bounds to get high-probability bounds for the depth of any node in a Treap. Since we said the recursion tree for quick sort has the same structure as a Treap, this will also give us high probability bounds for the depth of the quick sort tree, which can then be used to bound the span of quick sort with high probability.

Recall that the random variable A_{ij} indicates that j is an ancestor of i. It turns out all we have to do is argue that for a given i the random variables A_{ij} are independent. If they are, we can then use the Chernoff bounds. In particular, we want to analyze for any key i, $A_i = \sum_{j=1}^{n} A_{ij}$ (recall that this corresponds to the depth of key i in a treap). As derived earlier the expectation $\mu = \textbf{E}\left[A_i\right]$ is $H_{i-1} + H_{n-i-1}$ which lies between $\ln n$ and $2 \ln n$. If we set $\lambda = 4$, we have

[4] For full disclosure, we can relax this guarantee a bit; indeed, we can tolerate certain amount of dependence at the expense of a weaker bound, but this is beyond the scope of the class.

[5] Of course, no two stocks are truly independent, so the theorem cannot be directly applied to stocks, although one might be able to separate out the common trend.

$$\textbf{Pr}\left[X > (1+4)\mu\right] \;<\; e^{\frac{-\mu(4)^2}{2+4}}$$

Now since $\mu \geq \ln n$ we have

$$\begin{aligned}\textbf{Pr}\left[X > 5\mu\right] &< e^{-2\ln n} \\ &= \frac{1}{n^2}\end{aligned}$$

This means that the probability that a node is deeper than fives times its expectation is at most $\frac{1}{n^2}$, which is very small for reasonably large n (e.g. for $n = 10^6$ it is 10^{-12}). Since the expected depth of a key is at most $2\ln n$, we have

Theorem 4.1. *For a Treap with n keys, the probability that a key i is deeper than $10\ln n$ is at most $1/n^2$.*

Note that this just gives us the probability that any one key is deeper than 5 times its expectation. To figure out the worst case over all keys, we have to multiply the probability that any one of them is deeper than 5 times the expectation by the number of keys. More formally, let $\mathcal{E}_x$ be the "bad" event that the key x is deeper than 5 times its expectation. So then, by applying the union bound, we have

$$\begin{aligned}\textbf{Pr}\left[\text{any key is deeper than } 10\ln n\right] &= \textbf{Pr}\left[\exists x.\mathcal{E}_x\right] \\ &\leq \sum_x \textbf{Pr}\left[\mathcal{E}_x\right] \\ &\leq n \times \frac{1}{n^2} = \frac{1}{n}.\end{aligned}$$

This gives us the following theorem:

Theorem 4.2. *For a Treap with n keys, the probability that any key is deeper than $10\ln n$ is at most $1/n$.*

We now return to why for a given i the variables A_{ij} are independent. This argument is a bit subtle, so be impressed with yourself if you get it the first time. Let's just consider the $j > i$ (the others are true by a symmetrical argument), and scan from i forward. Each time we look at the j whether it is an ancestor only depends on whether its priority is larger than all priorities from i to $j - 1$. It does not depend on the relative ordering of those priorities. It is therefore independent of those priorities.

Conclusions

- The expected depth of each node i in a treap can be analyzed by determining for every other node j the expectation that j is an ancestor of i. These expectations can then be summed across all j giving $O(\log n)$ for the expected depth of each node.
- The analysis of Treaps is very similar to the analysis of quicksort.

- The split operation for treaps is the same as for unbalanced trees. The join operation is slightly more involved. Since nodes have expected $O(\log n)$ depth, they both run in $O(\log n)$ expected work and span.
- Although expectations can sometimes be useful, sometimes we want a stronger guarantee. In particular we might want to know that we will succeed (e.g. complete an operation in an allotted time) with high probability.
- High probability bounds can be generated from the law of large numbers. A particular form of this law are the so-called chernoff bounds, which give us a specific probability of our variable of concern diverging significantly from the mean.
- We can use Chernoff bounds to prove that the depth of a treap with random priorities is $O(\log n)$ with high probability. Since the recursion tree for quicksort has the same distribution as a treap, this also gives the same bounds for the depth of recursion of quicksort.

Lecture 21 — Fun With Trees (DRAFT)

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — Nov 8, 2011

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. 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

$\texttt{last}(S)$	:	$\mathbb{S} \to \mathbb{U}$	=	$\max S$
$\texttt{first}(S)$	:	$\mathbb{S} \to \mathbb{U}$	=	$\min S$
$\texttt{split}(S, k)$	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{S} \times \mathit{bool} \times \mathbb{S}$	=	as with trees
$\texttt{join}(S_1, S_2)$	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	=	as with trees
$\texttt{getRange}(S, k_1, k_2)$	:	$\mathbb{S} \times \mathbb{U} \times \mathbb{U} \to \mathbb{S}$	=	$\{k \in S \mid k_1 \le k \le k_2\}$

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*.

Here we consider a more involved example of the application of ordered sets. 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:

```
fun inDomain(domain, L) =
  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 $\texttt{domain}$ and $\texttt{string.append}(\texttt{domain}, \texttt{"\$"})$ 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:

$$\begin{array}{llll}
\texttt{rank}(S,k) & : \mathbb{S} \times \mathbb{U} \to \mathit{int} & = & |\{k' \in S \mid k' < k\}| \\
\texttt{select}(S,i) & : \mathbb{S} \times \mathit{int} \to \mathbb{U} & = & k \text{ such that } |\{k' \in S \mid k' < k\}| = i \\
\texttt{splitIdx}(S,i) & : \mathbb{S} \times \mathit{int} \to \mathbb{S} \times \mathbb{S} & = & (\{k \in S \mid k < \texttt{select}(S,i)\}, \\
 & & & \ \{k \in S \mid k \geq \texttt{select}(S,i)\})
\end{array}$$

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 $\texttt{select}$ and $\texttt{splitIdx}$ 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 $\texttt{rank}$ requires visiting all nodes before k. Therefore all these operations will take $\Theta(|S|)$ work. In fact even implementing $\texttt{size}(S)$ 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 $\texttt{size}$ operation fast, but what about the other operations? Well it turns out it allows us to implement $\texttt{select}$, $\texttt{rank}$, and $\texttt{splitIdx}$ all in $O(d)$ work assuming the tree has depth d. Therefore for balanced trees the work will be $O(\log|S|)$. Lets consider how to implement $\texttt{select}$:

```
**fun** $\texttt{select}(T,i) =$
   **case** $\texttt{expose}(T)$ **of**
     $\mathit{NONE} \Rightarrow$ **raise** $\texttt{Range}$
    $|$ $\mathit{SOME}(L,R,k) \Rightarrow$
        **case** $\texttt{compare}(i,|L|)$ **of**
           $\mathit{LESS} \Rightarrow \texttt{select}(L,i)$
         $|$ $\mathit{EQUAL} \Rightarrow k$
         $|$ $\mathit{GREATER} \Rightarrow \texttt{select}(R, i - \mathit{size}(L) - 1)$
```

To implement $\mathtt{rank}$ we could simply do a split and then check the size of the left tree. The implementation of $\mathtt{splitIdx}$ is similar to split except when deciding which branch to take, be base it on the sizes instead of the keys. In fact with $\mathtt{splitIdx}$ we don't even need select, we could implement it as a $\mathtt{splitIdx}$ followed by a $\mathtt{first}$ on the right tree.

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 extremely useful for several applications. We will get back to the applications but first describe the interface and its implementation using binary search tres. Consider the following abstract data type that extends ordered tables with one additional operation.

Definition 2.1. Consider an ordered table that maps keys of type k to values of type v, along with a function $f : v \times v \to v$ and identity element I_f. An *orderes table with reduced values* of type $\mathbb{T}$ 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 support the operation

$$\mathtt{reduceVal}(T) : T \to v = \mathtt{reduce}\ f\ I\ T$$

The $\mathtt{reduceVal}(T)$ 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 $\mathtt{reduce}$ 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 $\mathtt{reduceVal}$ in $O(1)$ work.

You might ask how can we possibly implement a $\mathtt{reduce}$ 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 $\mathtt{reduceVal}$, 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 using binary search trees. The basic idea is simply to store with every node n of a binary search tree the reduced value of its subtree (i.e. the sum of all the values that are descendants of n as well as the value at n itself. For example lets assume our function f is addition, and we are given the following BST mapping character keys to integer values:

```
            o (e,2)
          /    \
   (c,1) o        o (g, 5)
        / \
       o   o
   (a,3)   (d,2)
```

Now when we associate the reduced with each node we get

```
               o (e, 2, 13)
             /    \
(c, 1, 6) o         o (g, 5, 5)
         / \
        o   o
(a, 3, 3)    (d, 2, 2)
```

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.

The code to extend treaps with this idea is the following:

```
datatype $\texttt{Treap} = \texttt{Leaf} \mid \texttt{Node}$ of $(\texttt{Treap} \times \texttt{Treap} \times \texttt{key} \times \texttt{data} \times \texttt{data})$
fun $\texttt{reduceVal}(T) =$
  case $T$ of
    $\texttt{Leaf} \Rightarrow \texttt{Reduce}.I$
  | $\texttt{Node}(\_,\_,\_,\_,r) \Rightarrow r$
fun $\texttt{makeNode}(L,R,k,v) =$
  $\texttt{Node}(L,R,k,v,\texttt{Reduce}.f(\texttt{reduceVal}(L),\texttt{Reduce}.f(v,\texttt{reduceVal}(R))))$
fun $\texttt{join}'(T_1,T_2) =$
  case $(T_1,T_2)$ of
    $(\texttt{Leaf},\_) \Rightarrow T_2$
  | $(\_,\texttt{Leaf}) \Rightarrow T_1$
  | $(\texttt{Node}(L_1,R_1,k_1,v_1,s_1),\texttt{Node}(L_2,R_2,k_2,v_2,s_2)) \Rightarrow$
      if $(\texttt{priority}(k_1) > \texttt{priority}(k_2))$ then
        $\texttt{makeNode}(L_1,\texttt{join}'(R_1,T_2),k_1,v_1)$
      else
        $\texttt{makeNode}(\texttt{join}'(T_1,L_2),R_2,k_2,v_2)$
```

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.

2.2.1 Analyzing Profits at Tramlaw

Lets say that based on your expertise in algorithms you are hired as a consultant by the giant retailer Tramlaw$^{\text{TM}}$. Tramlaw sells over 10 billion items per year across its 6000+ stores. As with all 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:

$$\mathtt{reduceVal}(\mathtt{getRange}(T, t_1, t_2))$$

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.

2.2.2 Working for 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. They want to efficiently support queries that return the maximum price of a trade during any time range (t_1, t_2) .

Well you can use an ordered table with reduced values but instead of using addition for f, you would use $\max$. The query now is exactly the same as with your consulting jig with Tramlaw and they 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.*

2.2.3 Interval Queries

After your two consulting jobs, you are taking 15-451, with professor Mulb. On a test he asks you to support an abstract data type iTable that supports the following operations on intervals where an interval is a region on the real number line starting at x_l and ending at x_r.

insert(T, I)	:	iTable * (Real * Real) $\rightarrow$ iTable	insert interval I into table T
delete(T, I)	:	iTable * (Real * Real) $\rightarrow$ iTable	delete interval I from table T
count(T, x)	:	iTable * Real $\rightarrow$ int	return the number of intervals crossing x in T

Exercise 2. *How would you implement this.*

2.2.4 Others

- 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 $O(logn)$ work and verify whether updated.
- 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 23 — Augmenting Balanced Trees and Leftist Heaps (DRAFT)

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — Nov 15, 2011

Today:
- Independence
- Continue on Augmenting Balanced Trees (see notes from class 21)
- Priority Queues and Leftists Heaps

1 Independence

In problem 5c of Exam 2 we asked whether the $O(\log n)$ bounds on the expected size of a subtree in a treap implied $O(\log n)$ bounds with high probability. When discussing the depth of a node in a subtree we argued that $O(\log n)$ expected bounds implied high-probability bounds so the question is whether this carries over to subtree size. Recall that

$$A_{i,j} = \begin{cases} 1 & \text{if } j \text{ is an ancestor of } i \\ 0 & \text{otherwise} \end{cases}$$

where i and j are the locations of keys in the sorted order. Here, we assume a node is an ancestor of itself (i.e., $A_{ii} = 1$). Therefore for the depth of a node (as previously shown) we have

$$D_i = \left(\sum_{j=1}^{n} A_{i,j}\right) - 1$$

where we remove the 1 from the sum since we don't want to include the node itself, and for the size we have

$$S_i = \sum_{j=1}^{n} A_{j,i}$$

where we include the node itself. We also previously analyzed that $\textbf{Pr}\left[A_{i,j}\right] = 1/(|j-i|+1)$ so we can plug this in and get:

$$\textbf{E}\left[D_i\right] = H_i + H_{n-i+1} - 2 = O(\log n)$$

and

$$\textbf{E}\left[S_i\right] == H_i + H_{n-i+1} - 1 = O(\log n)$$

We have already argued that $\textbf{E}\left[D_i\right] = O(\log n)$ implies that with high probability D_i is $O(\log n)$. To do this we argued that the terms in the sum $\sum_{j=1}^{n} A_{i,j}$, the $A_{i,j}$ are independent. This is what allowed us to apply Chernoff bounds.

order	$A_{3,1}$	$A_{2,1}$	$A_{3,1} \| A_{2,1}$	$A_{1,3}$	$A_{1,2}$	$A_{1,3} \| A_{1,2}$
1,2,3	1	1	1	0	0	-
1,3,2	1	1	1	0	0	-
2,1,3	0	0	-	0	1	0
2,3,1	0	0	-	0	1	0
3,1,2	0	1	0	1	0	-
3,2,1	0	0	-	1	1	1
Pr [...]	2/6 = 1/3	3/6 = 1/2	2/3	2/6 = 1/3	3/6 = 1/2	1/3

Figure 1: Considering all 6 possible priority orderings of three keys. The first column represents the ordering: e.g., 1,2,3 indicates that key 1 has the highest priority, key 2 the second highest, and key 3 the lowest. We see that **Pr** $[A_{3,1}] = \frac{1}{3}$ while **Pr** $[A_{3,1}|A_{2,1}] = \frac{2}{3}$ so $A_{2,1}$ and $A_{3,1}$, which are added to determine the subtree size, are not independent. On the other hand **Pr** $[A_{1,3}] =$ **Pr** $[A_{1,3}|A_{1,2}] = \frac{1}{3}$ so $A_{1,2}$ and $A_{1,3}$, which are added to determine node depth, are independent.

The question is whether this is also true for the sum $\sum_{j=1}^{n} A_{j,i}$. By meta-reasoning it cannot be the case that with high probability subtree sizes are $O(\log n)$. This is because in every treap many of the nodes have subtrees larger than $O(\log n)$. In particular the size of the root is n and all but some nodes near the leaves have size greater than $O(\log n)$. This must imply that the terms in the sum are not independent.

To see that they are indeed not independent consider just three keys. We have that $S_1 = 1 + A_{2,1} + A_{3,1}$. So the question is whether $A_{2,1}$ and $A_{3,1}$ are independent, i.e. is the probability of one affected by the outcome of the other. The answer is that they are **not** independent. Informally this is because if 1 wins against 2 ($A_{2,1}$) then it is more likely to win against 2 and 3 ($A_{3,1}$). More formally, it is not hard to verify that **Pr** $[A_{3,1}] = \frac{1}{3}$ but that **Pr** $[A_{3,1}|A_{2,1}] = \frac{2}{3}$ (see Figure 1). Recall that the notation **Pr** $[X|Y]$ means the probability that X is true given that Y is true, and that X and Y are only independent if **Pr** $[X] =$ **Pr** $[X|Y]$. Therefore the probability of $A_{3,1}$ depends on the outcome of $A_{2,1}$ and the two random variables are not independent.

2 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 priority first graph search to implement Dijkstra's algorithm, and the A* variant for shortest paths, 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:

```
fun sort(S) =
let
  (* Insert all keys into priority queue *)
  val pq = Seq.iter Q.insert Q.empty S

  (* remove keys one by one *)
  fun sort' pq =
    case (PQ.deleteMin pq) of
      NONE => S.empyt
    | SOME(v,pq') => Seq.cons(v,sort'(pq'))
in
  sort' pq
end
```

As covered in Exam 1, with a meld operation the insertion into a priority queue can be implemented in parallel using a reduce instead of iter.

```
  (* Insert all keys into priority queue *)
  val pq = Seq.reduce Q.meld Q.empty (Seq.map Q.singleton S)
```

However, there is no real way to implement removing the keys in parallel unless we use something more powerful than a heap.

Priority queues also have applications elsewhere, including

- Huffman Codes
- Clustering algorithms
- Event simulation
- Kinetic algorithms

Today we will discuss how to implement a meldable priority queue that supports insert, deleteMin, and meld all in $O(\log n)$ time, where n is the size of the resulting priority queue. The structure is called a leftist heap. There are several other ways to achieve these bounds, but leftists heaps are particularly simple and elegant.

Before we get into leftist heaps lets consider some other possibilities and what problems they have.

We could use a sorted array.

We could use an unsorted array.

We could use a BST.

2.1 Leftist Heaps

The goal of leftists heaps is to make the `meld` fast, and in particular run in $O(\log n)$ work. As a reminder 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. For example the following is a min-heap

```
          o 3
         / \
      7 o   o 8
       / \
   11 o   o 15
         / \
     22 o   o 14
```

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 the second will give us more flexibility that available in a BST.

Lets consider how to implement the three operations *deleteMin*, *insert* and *meld* on a heap. To implement *deleteMin* we can simply remove the root. This would leave:

```
      7 o       o 8
       / \
   11 o   o 15
         / \
     22 o   o 14
```

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.

This basically means that the only operation we need to care about is the *meld* operation. Lets consider the following two heaps

```
     4 o             o 3
      / \           / \
  11 o   o 7     8 o   o 5
    / \           /
19 o   o 23   14 o
```

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 lets meld with the right child. So this would give us:

```
          o 3
         / \
      8 o   = meld ( 4 o          o 5 )
       /              / \
   14 o           11 o   o 7
                    / \
                19 o   o 23
```

If we apply this again we get

```
           o 3
          / \
      8 o    o 4
       /    / \
  14 o  11 o  = meld ( o 7  o 5)
           / \
      19 o   o 23
```

and one more time gives:

```
           o 3
          / \
     8 o     o 4
      /    /   \
 14 o  11 o     o 5
          / \     \
     19 o  o 23  = meld ( o 7    empty)
```

Clearly if we are melding a heap A with an empty heap we can just use A. This leads to the following code

```
datatype $PQ = \texttt{Leaf} \mid \texttt{Node}$ of $(\texttt{key}, PQ, PQ)$
fun $\texttt{meld}(A, B) =$
  case $(A, B)$ of
    $(\_, \texttt{Leaf}) \Rightarrow A$
  | $(\texttt{Leaf}, \_) \Rightarrow B$
  | $(\texttt{Node}(k_A, L_A, R_A), \texttt{Node}(k_B, L_B, R_B)) \Rightarrow$
      case $\texttt{Key.compare}(k_A, k_B)$ of
        $\texttt{LESS} \Rightarrow \texttt{Node}(k_A, L_A, \texttt{meld}(R_A, B))$
      | $\_ \Rightarrow \texttt{Node}(k_B, L_B, \texttt{meld}(A, R_B))$
```

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't 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. It turns out there is a relatively easy fix to this 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 tree as the length of the right spine. More formally:

```
$\texttt{rank}(\texttt{Leaf}) = 0$
$\texttt{rank}(\texttt{Node}(\_, \_, R) = 1 + \texttt{rank}(R)$
```

Now we require that for all nodes of a leftist heap (v, L, R)

$$\texttt{rank}(L) \geq \texttt{rank}(R)$$

This is why the tree is called leftist: the rank of all left branches are always at least as great as that of the right branch. Note that this definition allows the following unbalanced tree.

```
              o 1
             /
            o 2
           /
          o 3
         .
        .
       .
      o n
```

This is OK since we will only ever traverse the right spine of a tree, which in this case has length 1. We can now bound the rank.

Theorem 2.1. *For any leftist heap* T $\texttt{rank}(T) \leq \log |T|$.

This will be proven in next lecture. 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:

```
datatype $PQ = \texttt{Leaf} \mid \texttt{Node}$ of $(\texttt{int}, \texttt{key}, PQ, PQ)$
fun $\texttt{rank Leaf} = 0$
  | $\texttt{rank}\ (\texttt{Node}(r, \_, \_, \_)) = r$
fun $\texttt{makeLeftistNode}\ (v, L, R) =$
  if $(\texttt{rank}(L) < \texttt{rank}(R))$
  then $\texttt{Node}(1 + \texttt{rank}(L), v, R, L)$
  else $\texttt{Node}(1 + \texttt{rank}(R), v, L, R)$
fun $\texttt{meld}(A, b) =$
  case $(A, B)$ of
     $(\_, \texttt{Leaf}) \Rightarrow A$
   | $(\texttt{Leaf}, \_) \Rightarrow B$
   | $(\texttt{Node}(\_, k_A, L_A, R_A), \texttt{Node}(\_, k_B, L_B, R_B)) \Rightarrow$
       case $\texttt{Key.compare}(k_A, k_B)$ of
          $\texttt{LESS} \Rightarrow \texttt{makeLeftistNode}(k_A, L_A, \texttt{meld}(R_A, B))$
        | $\_ \Rightarrow \texttt{makeLeftistNode}(k_B, L_B, \texttt{meld}(A, R_B))$
```

Note that the only real difference is that we now use `makeLeftistNode` to create a node and 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. We now have

Theorem 2.2. *If A and B are leftists heaps then the* $\texttt{meld}(A, B)$ *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 and does constant work at each step. Therefore since both trees are leftist by Theorem 2.1 the work is bounded by $O(\log(|A|) + \log(|B|))$ work. 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. □

Lecture 24 — More Leftist Heaps and Sorting Lower Bounds (DRAFT)

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Kanat Tangwongsan — November 16, 2011

Today:
- Finishing Leftist Heaps
- Lower Bounds for Sorting and Merging

1 Recap: Leftist Heaps

Let's take another look at leftist min heaps. Last time, we saw that leftist heaps give a priority-queue implementation with fast `meld` operation, which supports combining leftist heaps of sizes n and m in $O(\log n + \log m)$ work.

1.1 How the leftist heap got its name?

Invented by Clark Allen Crane in around 1971, a leftist heap is a binary tree—not a binary search tree—that satisifies the heap property and an additional property known as the leftist property. The heap property means that in a min heap, the value at any node is at least the values at the two children. Note that with heap property alone, we can identify the minimum value very quickly in $O(1)$ since the minimum value is at the root of the tree. But all update operations can take arbitrary long.

This is where the "leftist" idea comes in, with the goal of creating more structure and ensuring that all update operations we care about can be supported efficiently. The leftist property requires that for each node in the heap, the "rank" of the left child must be at least the "rank" of the right child. As we defined last time, the rank of a node x is

$$\mathsf{rank}(x) = \#\text{ of nodes on the right spine of the subtree rooted at } x,$$

and the rank of a leaf is 0. 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, $\mathsf{rank}(L(x)) \geq \mathsf{rank}(R(x))$

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. We'll make this idea precise in the following lemma; following that, 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* $\log_2(n + 1)$.

Proof. We'll 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 node. We can establish a recurrence for $n(r)$ as follows: Let's look at a the smallest-sized heap whose root node x has rank r. First, the right child of x must necessarily have 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 $\mathsf{rank}(L(x)) \geq \mathsf{rank}(R(x)) = r - 1$. Therefore, the size of the tree rooted x is $n(r) = 1 + |L(x)| + |R(x)|$, so then

$$\begin{aligned} n(r) &\geq 1 + n(\mathsf{rank}(L(x))) + n(\mathsf{rank}(R(x))) \\ &\geq 1 + n(r-1) + n(r-1) = 1 + 2 \cdot n(r-1). \end{aligned}$$

Unfolding the recurrence, we get $n(r) \geq 2^r - 1$, which proves the claim.

To conclude the lemma, we'll 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 this heap is $r \leq \log_2(n+1)$. □

In words, this lemma says *leftist heaps have a short right spine,* about $\log n$ in length. To get good effiency, 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. Indeed, this is precisely what the algorithm we saw last time did: the meld algorithm below effectively traverses the right spines of the heaps a and b. (Note how `meld` is called only with either (R_a, b) or (a, R_b).)

```
datatype PQ = Leaf | Node of (int, key, PQ, PQ)
fun rank Leaf = 0
  | rank (Node(r, _, _, _)) = r
fun makeLeftistNode (v, L, R) =
  if (rank(L) < rank(R))
  then Node(1 + rank(L), v, R, L)
  else Node(1 + rank(R), v, L, R)
fun meld(a, b) =
  case (a, b) of
      (_, Leaf) ⇒ a
    | (Leaf, _) ⇒ b
    | (Node(_, k_a, L_a, R_a), Node(_, k_b, L_b, R_b)) ⇒
        case Key.compare(k_a, k_b) of
           LESS ⇒ makeLeftistNode(k_a, L_a, meld(R_a, b))
         | _ ⇒ makeLeftistNode(k_b, L_b, meld(a, R_b))
```

Notice the use of the function `makeLeftistNode`: the role of it is to ensure that the resulting heap satisfies the leftist property assuming the two input heaps L and R did. The function can also be viewed as swapping the left and right children if the original ordering violates the leftist property.

Performance of Meld: As we observed already, the `meld`algorithm only traverses the right spines of the two heaps, advancing by one node in one of the heaps. Therefore, on input heaps A and B, the process takes at most $\mathsf{rank}(A) + \mathsf{rank}(B)$ steps, which by the lemma we just proved, is at most $\log_2(|A| + 1) + \log_2(|B| + 1)$. Since these steps take constant work, we have the following theorem:

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.*

1.2 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.

2 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.

2.1 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 in the *comparison-based model*, we need $\Omega(n \log n)$ comparisons to sort n entries. 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 comparision operation, which returns whether $x < y$. More precisely, we'll prove the following theorem:

Theorem 2.1. *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 comparision-based model, we cannot hope to do better:

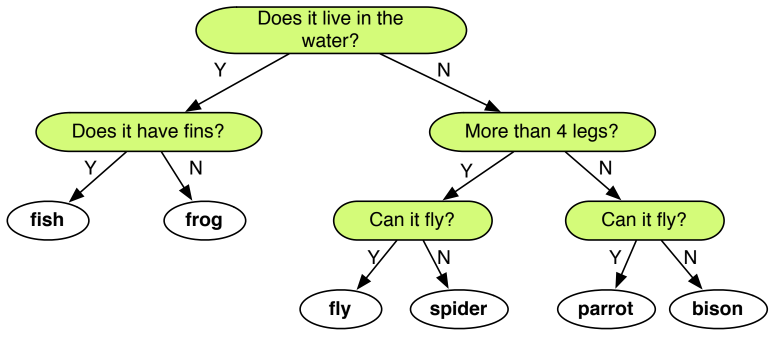
Theorem 2.2. *Merging two sorted sequences of lengths m and n ($m \leq n$) requires at least*

$$m \log_2(1 + \tfrac{n}{m})$$

comparison queries in the worst case.

2.2 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 determistic 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 2.3 (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 $\{0, \dots, k-1\}$;
- 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 cruical observation is the following: if we're allowed to make at most q queries (i.e., ask at most q) 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

$$\boxed{\text{If there are } N \text{ possible outcomes, the number of questions needed is at least } \log_2 N.}$$

2.3 Warm-up: Guess a Number

As a warm-up question, if you pick a number a between 1 and 2^{20}, how many Yes/No questions do I need to ask before I can zero in on a? By the calculation above, since there are $N = 2^{20}$ possible outcomes, I will need *at least*

$$\log_2 N = 20$$

questions in the worst case.

2.4 A Sorting Lower Bound

Let's turn back to the classical sorting problem. We will prove Theorem 2.1. This 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

$$\begin{aligned}\log(n!) &= \log n + \log(n-1) + \dots \log 1\\ &\geq \log n + \log(n-1) + \cdots + \log(n/2)\\ &\geq \tfrac{n}{2} \cdot \log(n/2).\end{aligned}$$

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 $\log_2(n!) > n\log_2(n/e)$.

2.5 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 comparsion 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 comparision 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 2.4 (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 2.2

Lecture 25 — Hashing (DRAFT)

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — Nov 22, 2011

Today:
- Hashing

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 1950s.

More formally the idea of hashing is to approximate a random function $h : \alpha \to \beta$ from a source set α to a destination set 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.

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.

[1]Merriam Websters

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

$$h(S) = \left(\sum_{i=1}^{|S|} = s_i a^i \right) \bmod p$$

1.1 Open Address Hash Tables

In 15-122 you covered hash tables using separate chaining.

We are going to cover a technique that does not need any linked lists but instead stores every key directly in 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.

In our discussion I will assume we have a set of keys we want to store and a hash function $h : \texttt{key} \to [1, \ldots, n]$ for some n.

Basic idea of open address is to maintain an array that is some constant factor larger than the number of keys and store all keys directly in this array. Every cell in the array is either empty or contains a key.

To decide what keys go in what cells we assign every key an ordered sequence of locations in which the key can be stored. In particular lets assume we have a function $h(k, i)$ that returns the i^{th} location for key k. We refer to the sequence $\langle\, h(k, 1), h(k, 2), h(k, 3), \ldots \rangle$ is as the *probe* sequence. We will get back to how the probe sequence might be assigned, but lets first go through how these sequences are used. The basic idea is when inserting for a key to try each of the locations in order until it finds a slot that is empty, and then insert the key at that location. Sequentially this would look like

```
fun $insert(T, k) =$
let
  fun $insert'(T, k, i) =$
    case $T[h(k, i)]$ of
      $NONE \Rightarrow update(i, k, T)$
    | $\_ \Rightarrow insert'(T, k, i + 1)$
in
  $insert'(T, k, 1)$
end
```

For example:

```
T = [ _  B  _   _   E  A _ F ]
```

Now if for a key D we had the probe sequence $\langle 1, 5, 3, \cdots \rangle$ (0 based) then we would find location 1 and 5 full (with B and E) and place D in location 3 giving:

```
T = [ _  B  _   D  E  A _ F ]
```

Note that the code 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 also checking when the table is full. Also note that the code given will insert the same key multiple times over.

To search we then have the following code:

```
fun $find(T, k) =$
let
  fun $find'(T, k, i) =$
    case $T[h(k, i)]$ of
      $NONE \Rightarrow false$
    | $SOME(k') \Rightarrow$ if $(eq(k, k'))$ then $true$
                        else $find'(T, k, i + 1)$
in
  $find'(T, k, 1)$
end
```

Now if for a key E we have the probe sequence $\langle 7, 4, 2, \cdots \rangle$ we would first search location 7 and, which is full, then location 4 and find E.

```
Do example of find
```

```
What about delete
  -- ask if simple delete works
  -- lazy delete -- replace with a special HOLD VALUE
  -- datatype 'a entry = EMPTY | HOLD | FULL of 'a
  -- find will skip over a HOLD and move to next probe
  -- insert v can replace HOLD in probe sequence with FULL(v)
  -- if insert needs to overwrite old value, then must first search to end

What probe sequence can we use.
 - linear probing
 - quadratic probing
 - multiple hash functions
```

1.2 Parallel Hashing

We assume a function $\mathtt{injectCond}(IV, S) : (\mathit{int} \times \alpha)\mathit{seq} \times (\alpha\mathtt{option})\mathit{seq} \rightarrow (\alpha\mathtt{option})\mathit{seq}$. 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:

```
fun $\mathtt{insert}(T, K) =$
let
   fun $\mathtt{insert}'(T, K, i) =$
      if $|K| = 0$ then $T$
      else let
         val $T' = \mathtt{injectCond}(\{(h(k,i), k) : k \in K\}, T)$
         val $K' = \{k \in K \mid T[h(k,i)] \neq k\}$
      in
         $\mathtt{insert}'(T', K', i + 1)$     end
in
   $\mathtt{insert}'(T, k, 1)$
end
```

Note that if T is implemented using single threaded arrays, then this basically does the same work as the sequential version adding the elements one by one.

```
What does it mean for set and table implementations
  - search faster (in expectation)
  - insert faster (in expectation)
  - map, reduce, remain linear
  - union/merge slower or faster depending on particular settings
```

Lecture 26 — Dynamic Programming I (DRAFT)

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — Nov 29, 2011

Today:
- Dynamic Programming

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 name 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.[1]

In this course, as commonly used in computer science, we will use the term to mean an algorithmic technique in which (1) one constructs the solution of a larger problem instance by composing solutions to smaller instances, and (2) the solution to each smaller instance can be used in multiple larger instances. Dynamic programming is thus one of the inductive algorithmic techniques we have are covering in this course. Recall from Lecture 2 that in all the inductive techniques an algorithm relies on putting together

[1]And perhaps it will make you feel more compelled to fight scientific bigotry among our politicians.

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. solving the set cover problem (approximately) by removing the "best" set. In the case of contraction it is typically a constant fractions 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 D&C we have always assumed the solutions are solved independently and hence we have simply added up the work of each of the recursive calls. But what if two instances of size k, for example, both need the solution to the same instance of size $j < k$.

```
foo(A)    foo(B)           size k
   ^   ^   ^   ^

   |    \/    |
   |    /\    |
 foo(C)    foo(D)           size j < k
```

Although in the simple example sharing the results 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:

```
**fun** $fib(n) =$
  **if** $(n \leq 1)$ **then** $1$
  **else** $fib(n-1) + fib(n-2)$
```

but this will take exponential time in n. However, if the results from the instances are somehow shared then the algorithm only requires linear work:

```
   fib(5)
    |   \
    |   fib(4)
    |   /   |
  fib(3)   |
    |   \   |
    |   fib(2)
    |   /   |
  fib(1)   |
        \   |
        fib(0)
```

It turns out there are many quite practical problems where sharing of sub results is useful and can make a significant differences in the work used to solve a problem. In the course 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 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[2] 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 leafs 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, although this can be converted into a DAG with a single root by adding a new vertex and an edge from each of the previous roots to this single new root.

Abstractly dynamic programming can therefore be best viewed as evaluating a DAG by propagating values from the leafs 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. Many dynamic programs have significant parallelism although some do not. For example consider the following DAG in which the work and span for each vertex is given.

```
  (5,2)     (11,3)
    o ------>o----
             ^    \
 (3,1)      /      v
    o -----o---o---o  (1,1)
       (2,2) (4,1)
```

This does $5 + 11 + 3 + 2 + 4 + 1 = 26$ units of work and has a span of $1 + 2 + 3 + 1 = 7$.

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 solution 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. We note however that most problems that can be tackled with dynamic programming solutions are either 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). A *decision problem* is one in which we are trying to find if a solution exists.

Although abstractly dynamic programming can be viewed 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 uses recursion as in divide-and-conquer but remembers solutions so if the algorithm tries to solve the same instance many times only the first call does the work and the rest 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

[2]Note that "size" is used in an abstract sense and does not necessarily mean the size (e.g. number of bytes) of the input but rather any measure that can be used for inductive purposes.

level order traversal—for example,e 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 (divide-and-conquer with memoization) can be quite elegant and can be more efficient in certain situations by evaluating fewer instances. Using the bottom up approach (level order traversal of the DAG) can be easier to parallelize and also 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 class is the subset sum problem:

Definition 1.1. The *subset sum* (SS) problem is given a multiset of integers S and an integer value k to determine if there is any $X \subset S$ such that $\sum_{x \in X} x = k$.

In the general case when k is unconstrained this problem is a classic NP hard problem. However, our goals here are more modest. We are going to consider the case where we include k is the work bounds. We show that as long as k is polynomial in n then the work is also polynomial in n. 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 clearly takes exponential time. For a more efficient solution, one should first consider an inductive solution to the problem. You might consider some form of greedy method greedily takes elements from S. Unfortunately this 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 take one element a out of S and consider the two possibilities: either a is included in X or not. For each these two possibilities we make a recursive call—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))
```

Everything except for the first and last line are 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't add a to X, so we need only make one recursive call. The last line is the main inductive case were we either include a or not. In both cases we remove a from S in the recursive call R.

So what is the work of this 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 here is the recursion tree for the instance $SS(\{1,1,1\}, 3)$:

```
                               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({}, 0) SS({}, 1) SS({}, 1) SS({}, 2) SS({}, 1) SS({}, 2) SS({}, 2) SS({}, 3)
```

As you should notice there are many common calls. In the bottom row, for example there are three calls each to $SS(,1)$ and $SS(,2)$. If we coalesce the common call we get the following DAG where the edges are all going up, the leaves are at the bottom and root is at the top.

```
                              SS({1,1,1}, 3)
                             /              \
                  SS({1,1}, 2)              SS({1,1}, 3)
                /             \             /            \
           SS({1}, 1)           SS({1}, 2)           SS({1}, 3)
            /       \         /           \         /       \
     SS({}, 0)       SS({}, 1)             SS({}, 2)       SS({}, 3)
```

The question is how do we bound the number of distinct instances of SS, which is also the number of vertices in the DAG?

Well for an initial instance $SS(S_s, k_s)$ there are are only $|S_s|$ distinct lists that are ever used (each suffix of S_s). Furthermore k only decreases and never goes below 0, so it can take on at most $k_s + 1$ values. Therefore the total number of possible instances of SS (vertices in the DAG) is $|S_s|(k_s + 1) = O(k|S_s|)$. Each instance only does constant work to compose its recursive calls. Therefore the total work is $O(k|S_s|)$. Furthermore it should be clear that the longest path in the DAG is $O(|S_s|)$ so the total span is $O(|S_s|)$ and the algorithm has $O(k)$ parallelism.

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 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 $T = \Sigma^*$ to 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 minimum edit distance 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 "difference" from the previous version[3]. This is important 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.

One might consider a greedy method that scans the sequence 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. Again in the example, if you inserted an A, then more than two more edits will be required.

However, considering the greedy solution gives a good hint of how to find a correct solution. 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.

```
fun MED(S,T) =
   case (showl(S), showl(T)) of
      (_, NIL) ⇒ |S|
    | (NIL, _) ⇒ |T|
    | (CONS(s,S'), CONS(t,T')) ⇒
         if (s = t) then MED(S',T')
         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 ($\mathit{MED}(S,T')$) 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 ($\mathit{MED}(S',T)$) 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.

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 example here is the DAG for $\mathit{MED}(\langle A, B, C \rangle, \langle D, B, C \rangle)$ (all edges point up):

`MED({A,B,C},{D,B,C})`

[3] Alternatively it might store the new version, but use the difference to encode the old version.

```
                           /                         \
            MED({B,C},{D,B,C})                  MED({A,B,C},{B,C})
                   /            \             /             \
       MED({C},{D,B,C})         MED({B,C},{B,C})          MED({A,B,C},{C})
        /           \                    |                 /         \
MED({},{D,B,C}) MED({C},{B,C})    MED({C},{C})    MED({B,C},{C}) MED({A,B,C},{})
                 /      \                |             /        \
      MED({},{B,C}) MED({C},{C})   MED({},{})   MED({C},{C}) MED({B,C},{})
                          |                          |
                     MED({},{})                 MED({},{})
```

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

$$W(\mathit{MED}(S,T)) = O(|S||T|)$$

and

$$S(\mathit{MED}(S,T)) = O(|S| + |T|).$$

1.3 Problems with Efficient Dynamic Programming Solutions

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
11. Sequence alignment with gaps
12. subset sum

13. knapsack problem (with and without repetitions)
14. weighted interval scheduling
15. line breaking in paragraphs
16. break words in which space has been removed
17. chain matrix product
18. multiplication with a non associative operation – does any parenthesization give a desired result.
19. maximum value for parenthesizing x1/x2/x3.../xn for positive rational numbers
20. cutting a string at given locations to minimize cost (costs n to make cut)
21. all shortest paths
22. find maximum independent set in trees
23. smallest vertex cover on a tree
24. optimal bst
25. probability of generating exactly k heads with n biased coin tosses
26. triangulate a convex polygon while minimizing the length of the added edges
27. cutting squares of given sizes out of a grid
28. card game while picking largest value from one end or the other.
29. change making
30. box stacking
31. segmented least squares problem
32. counting boolean parenthesization – true, false, or, and, xor, count how many parenthesization return true
33. balanced partition – given a set of integers up to k, determine most balanced two way partition
34. RNA secondary structure
35. Largest common subtree

Lecture 27 — Dynamic Programming II (DRAFT)

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — Dec 1, 2011

Today:
- Dynamic Programming

1 Dynamic Programming Continued

1.1 Optimal Binary Search Trees

We have talked about using BSTs for storing an ordered set or table. The cost of finding an element is proportional to the depth of the element in the tree. In a fully balanced BST of size n the average depth of each element is about $\log n$. Now lets say you are given values associated with each element that specify the probability that the element will be accessed—perhaps the word "of" is accessed much more often than "epistemology". The probabilities across the elements must add to 1. The goal is to make it so that the more likely elements are closer to the root and hence the average access cost is reduced. This line of reasoning leads to the following problem:

Definition 1.1. The *optimal binary search tree* problem is given an ordered set of keys S and a probability function $p : S \to [0 : 1]$, determine:

$$\min_{T \in \texttt{Trees}(S)} \left(\sum_{s \in S} d(s, T) \cdot p(s) \right)$$

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.

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

```
        k5
      /    \
  k1         k6
    \
      k3
    /   \
  k2     k4
```

has cost $31/16$, which is optimal.

Exercise 1. *Find another tree with equal cost.*

As usual we are interested in solving the problem efficiently. 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. Instead lets consider a recursive solution. In particular lets try placing every element at the root and then recurse on the subproblems and pick the best of the $|S|$ possibilities. Lets consider how to calculate the cost give the solution to two subproblems. Assume S is a sequence ordered by the keys and we pick location i ($1 \geq i \leq |S|$) as a the root. We can now solve the OSBT problem on the prefix $S_{1,i-1}$ and suffix $S_{i+1,n}$ (the notation $S_{i,j}$ means the locations of S from i to j). 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. We therefore might consider adding these two solutions and the cost of the root ($p(S_i)$) 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_i we have increased the depth of each of their keys by 1. However we can adjust for this. In particular for each key s in each subtree, its cost has gone up by $p(s) \cdot 1$. We therefore have:

$$\begin{aligned}
\mathit{OBST}(S) &= \min_{i \in \langle 1 \ldots |S| \rangle} \left(\mathit{OBST}(S_{1,i-1}) + \mathit{OBST}(S_{i+1,|S|}) + p(S_i) + \sum_{s \in S_{i,i-1}} p(s) + \sum_{s \in S_{i+1,|S|}} p(s) \right) \\
&= \sum_{s \in S} p(s) + \min_{i \in \langle 1 \ldots |S| \rangle} \left(\mathit{OBST}(S_{1,i-1}) + \mathit{OBST}(S_{i+1,|S|}) \right)
\end{aligned}$$

When we add the base case this leads to the following recursive definition:

```
1  fun  $\mathit{OBST}(S) =$
2    if  $|S| = 0$  then  0
3    else $\sum_{s \in S} p(s) + \min_{i \in \langle 1 \ldots |S| \rangle} \left( \mathit{OBST}(S_{1,i-1}) + \mathit{OBST}(S_{i+1,|S|}) \right)$
```

Exercise 2. *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 to what are effectively 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 $a \times (B \times C)$. The matrix chain product problem can be solved in a very similar structure as the OBST algorithm and with the same cost bounds.

2 Coding Dynamic Programs

So far we have assumed some sort of magic recognized shared subproblems in our recursive codes and avoided recomputation. This sort of magic could actually be fully automated in the functional setting using a technique called hash consing [1], but no languages do this so we are left to our own means. As mentioned in the last lecture there are basically two techniques to code up dynamic programming techniques: the top-down approach and the bottom-up approach.

Top-Down Dynamic Programming

The top-down approach is based on running the recursive code basically as is but generating a mapping from input argument to solution as we proceed. This way when we come across the same argument a second time we can just look up the solution. This 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 so even though the values are equal there could be two copies of the same value in different locations and comparing pointers would say they are not equal and we would fail to recognize that we have already solved this instance.

To get around this problem the top-down approach typically requires that the user create integer surrogates that represent the input values[2]. The property of these integers is that there has to be a 1-to-1 correspondence between the integers and the argument values—therefore if the integers match, the arguments match. The user is responsible for guaranteeing this.

We now cover how this can be done for dynamic program we described for minimum edit distance (MED). In 15-150 you covered memoization but you did it using side effects. Here we will do it in a purely functional way. This requires that we "thread" the table that maps arguments to results through the computation. Although this requires a few extra characters of code, it is safer for parallelism.

Recall that MED takes two sequences and on each recursive call only uses suffixes of the two original sequences. To create integer surrogates we can therefore 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

[1]Basically every distinct value is given a unique ID so that testing for equality even for sequences, sets or other complex types can be done very cheaply even if constructed separately.

[2]other simple types would also work.

so instead of working front to back and using suffix lengths it could work back to front and use prefix lengths—we do this since it simplifies the indexing. This leads to the following variant of our MED code.

```
fun MED(S,T) = let
  fun MED'(i,0) = i
    | MED'(0,j) = j
    | MED'(i,j) = case (S_i = T_j) of
                     true ⇒ MED'(i − 1, j − 1)
                   | false ⇒ 1 + min(MED'(i, j − 1), MED'(i − 1, j))
in
  MED'(|S|, |T|)
end
```

We can now find solutions in our memo table quickly since it can be indexed on i and j. In fact since the arguments range from 0 to the length of the sequence we can actually use an array to store the table values.

Now we can actually implement the memoization. To do this we define a memoization function:

```
fun memo f (M,a) =
  case find(T,a) of
    SOME(v) ⇒ v
  | NONE ⇒ let
      val (M',v) = f(M,a)
    in
      (update(M',a,v),v)
    end
```

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 stores the result in the memo table. We can now write MED using memoization.

```
fun MED(S,T) = let
  fun MED'(M,(i,0)) = (M,i)
    | MED'(M,(0,j)) = (M,j)
    | MED'(M,(i,j)) = case (S_i = T_j) of
                         true ⇒ MED''(M,(i − 1, j − 1))
                       | false ⇒ let
                           val (M',v_1) = MED''(M,(i, j − 1))
                           val (M'',v_2) = MED''(M',(i − 1, j))
                         in (M'', 1 + min(v_1,v_2)) end
  and MED''(M,(i,j)) = memo MED' (M,(i,j))
in
  MED'({}, (|S|, |T|))
end
```

Note that the memo table M is threaded throughout the computation. In particular every call toe MED not only takes a memo table as an argument but also returns a possibly different memo table as a result. 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. Now the calls to MED can be made in parallel. However, then one has to be very careful since there can be race conditions (parallel threads modifying the same cells). Furthermore if two parallel threads make a call on MED on the same argument then they can and will often both end up doing the work. There are ways around this 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.

Bottom-Up Dynamic Programming

We will now consider a different 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 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 and it is therefore useful to understand the structure of the DAG. For example lets consider the structure of the DAG for minimum edit distance. In particular lets 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.

```
        t   c   a   t

        1   2   3   4     = i
        |   | \     |
a   1---o---o   o---o
      \     |   | \
t   2   o   o---o   o
        | \     |   |
c   3---o   o---o---o

  = j
```

The numbers represent the i and the j for that position in the string. Consider $\mathit{MED}(4, 3)$. The characters S_4 and T_3 are not equal so the recursive calls are to $\mathit{MED}(3, 3)$ and $\mathit{MED}(4, 2)$. This corresponds to the vertex to the left and the one above. Now if we consider $\mathit{MED}(4, 2)$ the characters S_4 and T_2 are equal so the recursive call is to $\mathit{MED}(3, 1)$. 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 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 done 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 process the array either by row, column, or diagonal. This can be coded as follows.

```
fun MED(S,T) = let
   fun MED'(M, (i, 0)) = i
     | MED'(M, (0, j)) = j
     | MED'(M, (i, j)) = case (S_i = T_j) of
                              true ⇒ M_{i−1,j−1}
                            | false ⇒ 1 + min(M_{i,j−1}, M_{i−1,j})
   fun diagonals(M, k) =
      if (k > |S| + |T|) then M
      else let
         val s = max(0, k − n)
         val e = min(k, m)
         val M' = M ∪ {(i, k − i) ↦ MED(M, (i, k − i)) : i ∈ {s, ..., e}}
      in
         diagonals(M', k + 1)
      end
   in
      diagonals({}, 0)
   end
```

The code uses a table M to store the array entries. In practice an array might do better. Each round of $\texttt{diagonals}$ processes one diagonal and updates the table M. We note that the index calculations are a bit tricky (hopefully we got them right). The size of the diagonal grows and then shrinks.

OBST Revisited. We now return to the OBST problem. 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.

```
fun OBST(S) = let
   fun OBST'(i, l) =
      if l = 0 then 0
      else $\sum_{k=0}^{l-1} p(S_{i+k}) + \min_{k=0}^{l-1} (OBST'(i,k) + OBST'(i+k+1, l-k-1))$
in
   OBST(1, |S|)
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.

15-210: Parallel and Sequential Data Structures and Algorithms
Syntax and Costs for Sequences, Sets and Tables

December 11, 2011

1 Psuedocode Syntax

In the pseudocode in the class we 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 improving the notation as we go, this notation is not completely backward compatible with earlier lectures in the course.

Sequences

S_i	$\texttt{nth}\ S\ i$
$\lvert S \rvert$	$\texttt{length}(S)$
$\langle\, \rangle$	$\texttt{empty}()$
$\langle\, v \,\rangle$	$\texttt{singleton}(v)$
$\langle\, i, \ldots, j \,\rangle$	$\texttt{tabulate}\ (\textbf{fn}\, k \Rightarrow i + k)\ (j - i + 1)$
$\langle\, e : p \in S \,\rangle$	$\texttt{map}\ (\textbf{fn}\, p \Rightarrow e)\ S$
$\langle\, e : i \in \langle 0, \ldots, n-1 \rangle \,\rangle$	$\texttt{tabulate}\ (\textbf{fn}\, i \Rightarrow e)\ n$
$\langle\, p \in S \mid e \,\rangle$	$\texttt{filter}\ (\textbf{fn}\, p \Rightarrow e)\ S$
$\langle\, e_1 : p \in S \mid e_2 \,\rangle$	$\texttt{map}\ (\textbf{fn}\, p \Rightarrow e_1)\ (\texttt{filter}\ (\textbf{fn}\, p \Rightarrow e_2)\ S)$
$\langle\, e : p_1 \in S_1, p_2 \in S_2 \,\rangle$	$\texttt{flatten}(\texttt{map}\ (\textbf{fn}\, p_1 \Rightarrow \texttt{map}\ (\textbf{fn}\, p_2 \Rightarrow e)\ S_2)\ S_1)$
$\langle\, e_1 : p_1 \in S_1, p_2 \in S_2 \mid e_2 \,\rangle$	$\texttt{flatten}(\texttt{map}\ (\textbf{fn}\, p_1 \Rightarrow \langle\, e_1 : p_2 \in S_2 \mid e_2 \,\rangle)\ S_1)$
$\displaystyle\sum_{p \in S} e$	$\texttt{reduce add}\ 0\ (\texttt{map}\ (\textbf{fn}\, p \Rightarrow e)\ S)$
$\displaystyle\sum_{i=k}^{n} e$	$\texttt{reduce add}\ 0\ (\texttt{map}\ (\textbf{fn}\, i \Rightarrow e)\ \langle\, k, \ldots, n \,\rangle)$
$\underset{p \in S}{\operatorname{argmax}}(e)$	$\texttt{argmax compare}\ (\texttt{map}\ (\textbf{fn}\, p \Rightarrow e)\ S)$

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` : $(\alpha \times \alpha \rightarrow \texttt{order}) \rightarrow (\alpha\ \texttt{seq}) \rightarrow \texttt{int}$ 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_v	$\texttt{find}\ S\ v$
$\lvert S\rvert$	$\texttt{size}(S)$
$\{\}$	$\texttt{empty}$
$\{v\}$	$\texttt{singleton}(v)$
$\{p \in S \mid e\}$	$\texttt{filter}\ (\textbf{fn}\ p \Rightarrow e)\ S$
$S_1 \cup S_2$	$\texttt{union}(s_1, s_2)$
$S_1 \cap S_2$	$\texttt{intersection}(s_1, s_2)$
$S_1 \setminus S_2$	$\texttt{different}(s_1, s_2)$
$\sum_{k \in S} e$	$\texttt{reduce add}\ 0\ (\texttt{Table.tabulate}\ (\textbf{fn}\ k \Rightarrow e)\ S)$

Tables

T_k	$\textbf{case}\ (\texttt{find}\ S\ k)\ \textbf{of}\ \texttt{SOME}(v) \Rightarrow v$
$\lvert T\rvert$	$\texttt{size}(T)$
$\{\}$	$\texttt{empty}()$
$\{k \mapsto v\}$	$\texttt{singleton}(k, v)$
$\{e : p \in T\}$	$\texttt{map}\ (\textbf{fn}\ p \Rightarrow e)\ T$
$\{k \mapsto e : (k \mapsto p) \in T\}$	$\texttt{mapk}\ (\textbf{fn}\ (k, p) \Rightarrow e)\ T$
$\{k \mapsto e : k \in S\}$	$\texttt{tabulate}\ (\textbf{fn}\ k \Rightarrow e)\ S$
$\{p \in T \mid e\}$	$\texttt{filter}\ (\textbf{fn}\ p \Rightarrow e)\ T$
$\{(k \mapsto p) \in T \mid e\}$	$\texttt{filterk}\ (\textbf{fn}\ (k, p) \Rightarrow e)\ T$
$\{e_1 : p \in T \mid e_2\}$	$\texttt{map}\ (\textbf{fn}\ p \Rightarrow e_1)\ (\texttt{filter}\ (\textbf{fn}\ p \Rightarrow e_2)\ T)$
$\{k : (k \mapsto \_) \in T\}$	$\texttt{domain}(T)$
$T_1 \cup T_2$	$\texttt{merge}\ (\textbf{fn}\ (v_1, v_2) \Rightarrow v_2)\ (T_1, T_2)$
$T \cap S$	$\texttt{extract}(T, S)$
$T \setminus S$	$\texttt{erase}(T, S)$
$\sum_{p \in T} e$	$\texttt{reduce add}\ 0\ (\texttt{map}\ (\textbf{fn}\ p \Rightarrow e)\ T)$
$\sum_{(k \mapsto p) \in T} e$	$\texttt{reduce add}\ 0\ (\texttt{mapk}\ (\textbf{fn}\ (k, p) \Rightarrow e)\ T)$
$\operatorname{argmax}_{(k \mapsto p) \in T}(e)$	$\texttt{argmax max}\ (\texttt{mapk}\ (\textbf{fn}\ (k, p) \Rightarrow e)\ T)$

2 Function Costs

ArraySequence	*Work*	*Span*
`length`(T) `singleton`(v) `nth` S i	$O(1)$	$O(1)$
`tabulate` f n	$O\left(\sum_{i=0}^{n} W(f(i))\right)$	$O\left(\max_{i=0}^{n} S(f(i))\right)$
`map` f S	$O\left(\sum_{s \in S} W(f(s))\right)$	$O\left(\max_{s \in S} S(f(s))\right)$
`filter` f S	$O\left(\sum_{s \in S} W(f(s))\right)$	$O\left(\log \lvert S\rvert + \max_{s \in S} S(f(s))\right)$
`reduce` f i S	$O\left(\lvert S\rvert + \sum_{f(a,b) \in \mathcal{O}_r(f,i,S)} W(f(a,b))\right)$	$O\left(\log \lvert S\rvert \max_{f(a,b) \in \mathcal{O}_r(f,i,S)} S(f(a,b))\right)$
`scan` f i S	$O\left(\lvert S\rvert + \sum_{f(a,b) \in \mathcal{O}_s(f,i,S)} W(f(a,b))\right)$	$O\left(\log \lvert S\rvert \max_{f(a,b) \in \mathcal{O}_s(f,i,S)} S(f(a,b))\right)$
`showt` S	$O(\lvert S\rvert)$	$O(1)$
`hidet` `NODE`(l, r)	$O(\lvert l\rvert + \lvert r\rvert)$	$O(1)$
`append`(S_1, S_2)	$O(\lvert S_1\rvert + \lvert S_2\rvert)$	$O(1)$
`flatten`(S)	$O\left(\lvert S\rvert + \sum_{s \in S} \lvert s\rvert\right)$	$O(1)$
`partition` I S	$O(\lvert I\rvert + \lvert S\rvert)$	$O(1)$
`inject` I S	$O(\lvert S\rvert)$	$O(1)$
`merge` f $\lvert S_1\rvert$ $\lvert S_2\rvert$	$O(\lvert S_1\rvert + \lvert S_2\rvert)$	$O(\log(\lvert S_1\rvert + \lvert S_2\rvert))$
`sort` f S	$O(\lvert S\rvert \log \lvert S\rvert)$	$O\left(\log^2 \lvert S\rvert\right)$
`collect` f S	$O(\lvert S\rvert \log \lvert S\rvert)$	$O\left(\log^2 \lvert S\rvert\right)$
Single Threaded Array Sequence		
`nth` S i `update` (i, v) S	$O(1)$	$O(1)$
`inject` I S	$O(\lvert I\rvert)$	$O(1)$
`fromSeq` S `toSeq` S	$O(\lvert S\rvert)$	$O(1)$

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 `merge`, `sort`, and `collect` the costs assume that the work and span of the application of f is constant.

Tree Sets and Tables	*Work*	*Span*
`size`(T) `singleton`(k, v)	$O(1)$	$O(1)$
`filter` $f\ T$	$O\Big(\sum_{(k,v)\in T} W(f(v))\Big)$	$O\Big(\lg \lvert T\rvert + \max_{(k,v)\in T} S(f(v))\Big)$
`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` $k\ T$	$O(\lg \lvert T\rvert)$	$O(\lg \lvert T\rvert)$
`extract` (T_1, T_2) `merge` $f\ (T_1, T_2)$ `erase` (T_1, T_2)	$O\big(m \lg(\frac{n+m}{m})\big)$	$O\big(\lg(n+m)\big)$
`domain` T `range` T `toSeq` T	$O(\lvert T\rvert)$	$O(\lg \lvert T\rvert)$
`collect` S `fromSeq` S	$O(\lvert S\rvert \lg \lvert S\rvert)$	$O(\lg^2 \lvert S\rvert)$
`intersection` (S_1, S_2) `union` (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.