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

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

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.

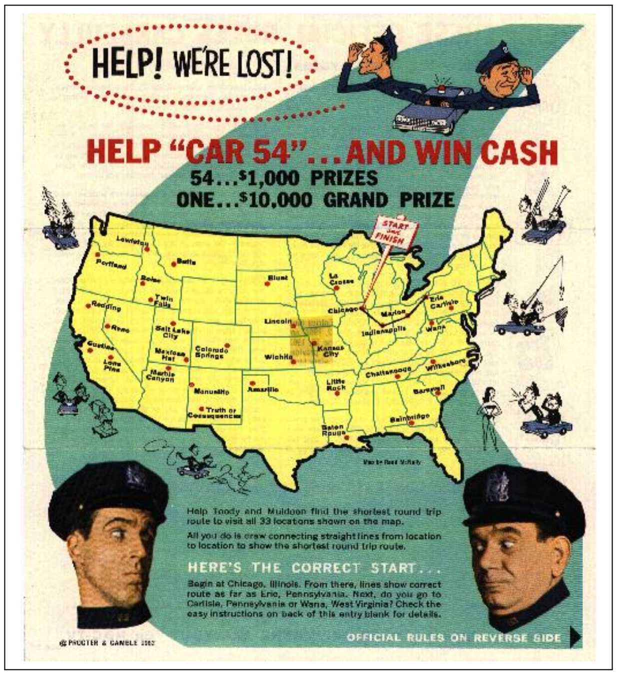
3.3 Algorithm 2: Reducing to Another Problem

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
HERE'S THE CORRECT START...
OFFICIAL RULES ON REVERSE SIDE

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.

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.

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.

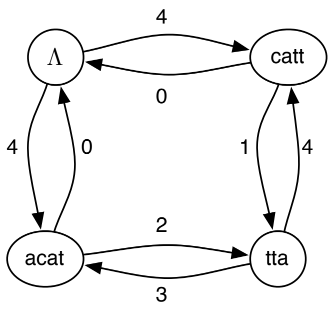
Motivated by the observation that the shortest superstring problem can be solved exactly by trying all permutations, we'll make the TSP problem try all the combinations for us. For this, we will set up a graph so that each valid Hamiltonian cycle corresponds to a permutation. Note that in our case, there 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 {catt, acat, tta} results in the following graph (not all edges are shown).



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