Lecture 1 — Overview and Sequencing the Genome

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

Lectured by Umut Acar — September 15, 2013

1 Administrivia

Welcome to 15-210 Parallel and Sequential Data Structures and Algorithms. This course will teach you methods for designing, analyzing, and programming sequential and parallel algorithms and data structures. The emphasis will be on fundamental concepts that will be applicable across a wide variety of problem domains, and transferable across a broad set of programming languages and computer architectures. One specific goal will be to develop the skills needed for "parallel thinking."

There is no textbook for the class. We will (electronically) distribute lecture notes and supplemental reading materials as we go along. We will try to generate a draft of lecture notes and post them before class.

The course web site is located at

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

Please take time to look around. You should read and understand the collaboration policy.

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

There will be weekly assignments (generally due at 11:59pm on Mondays), 2 exams, and a final. The first assignment is coming out tomorrow and will be due the Monday after next.

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

2 Course Overview

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

- **Interfaces and specifications:** defining precise problems and abstract data types
- **Design and implementation:** designing and programming correct and efficient algorithms and data structures for the given problems and data types (the implementations)

and can be summarized by the following table:

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

	Interface (specification)	**Implementation (design)**
Functions	Problem	Algorithm
Data	Abstract Data Type	Data Structure

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

There is a similar relationship between an abstract data type and its implementation as a data structure. An *abstract data type* (ADT) specifies precisely an interface for operating on data in an abstract form. It does not specify how the data is structured. A *data structure* implements the interface by organizing the data in a particular. For an ADT, the interface is specified in terms of a set of operations on the type. For example, a priority queue is an ADT with operations that might include `insert`, `findMin`, and `isEmpty?`. Various data structures can be used to implement a priority queue, including binary heaps, arrays, and balanced binary trees. The terminology ADTs vs. data structures is not as widely used as problems vs. algorithms. In particular, sometimes the term data structure is used to refer to both the interface and the implementation. We will try to avoid such usage in this class.

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

For these reasons, in this course we will put a strong emphasis on defining precise and concise interfaces and then implementing those abstractions using algorithms and data structures. When discussing solutions to problems we will emphasize general techniques that can be used to design them, such as divide-and-conquer, the greedy method, dynamic programming, and balance trees. It is important that in this course you learn how to design your own algorithms/data structures given an interface, and even how to specify your own problems/ADTs given a task at hand.

Parallelism. This course differs in an important way from traditional algorithms and data structures courses in that we will be thinking about parallelism right from the start. We are doing this since in recent years parallelism has become ubiquitous in computing platforms ranging from cell phones to supercomputers. Due to physical and economical constraints, a typical laptop or desktop machine we can buy today has 4 to 8 computing cores, and soon this number will be 16, 32, and 64. Furthermore, most computers have some form of graphical processing unit (GPU) which are themselves highly parallel. Beyond our homes, data centers and supercomputers today use hundreds and sometimes even thousands of cores to process a single request or job. These developments make the specification, the design, and the implementation of parallel algorithms a very important topic. Parallel computing requires a somewhat different way of thinking than sequential computing. Developing the intellectual skills for *parallel thinking* is an important goal of this class.

You might ask what advantage does one get from writing a parallel instead of a sequential algorithm to solve a problem. The most important advantage is the ability to perform sophisticated computations quickly enough to make them practical. For example without parallelism computations such as Internet searches, realistic graphics, climate simulations would be prohibitively slow. One way to quantify such an advantage is to measure the performance gains that we get from parallelism. Here are some example timings to give you a sense of what can be gained. These are on a 32 core commodity server machine (you can order one on the Dell web site).

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

In the table, the serial timings use sequential algorithms while the parallel timings use parallel algorithms. Notice that the speedup for the parallel 32 core version relative to the sequential algorithm ranges from approximately 12 (min spanning tree) to approximately 32 (sorting). Currently, obtaining such performance requires developing efficient and performant parallel algorithms and highly tuned implementations. In this course, we will focus on the first challenge.

The course also differs from most traditional algorithms and data structures courses in that we will be using a purely functional model of computation. The primary reason for this is that purely functional programs are safe for parallelism, i.e., they can be executed in parallel without any modifications. In an imperative setting one needs to worry about race conditions since parallel threads of execution might modify shared states in different orders from one run of the code to the next. This makes it much harder to think in parallel and reason about the correctness and the efficiency of parallel algorithms. However, all the ideas we will cover are also relevant to imperative languages—one just needs to be much more careful when coding imperatively. The functional style also encourages working at a higher level of abstraction by using higher order functions. This again is useful for parallel algorithms since it moves away from the one-at-a-time (loop) way of thinking that is detrimental to parallelism.

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

3 An Example: Sequencing the Genome

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

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

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

3.1 What makes sequencing the genome hard?

There is currently no way to read long strands with accuracy. Current DNA sequencing machines are only capable of efficiently reading relatively short strands, e.g., 1000 base pairs, compared to the approximately 6 billion in the whole genome. Therefore, we resort to cutting strands into shorter fragments and then reassembling the pieces. A technique called "primer walking" can be used to cut the DNA strands into consecutive fragments and sequence each one. But the process is slow because one needs the result of one fragment to "build" in the wet lab the molecule needed to find the following fragment. This is an inherently sequential process with large waiting time between each iteration.

Alternatively, there are fast methods to cut the strand at random positions. But this process mixes up the short fragments, so the order of the fragments is unknown. For example, the strand cattaggagtat might turn into, say, ag, gag, catt, tat, destroying the original ordering.

If we could make copies of the original sequence, is there something we could do differently to order the pieces? Let's look at the shotgun (sequencing) 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" that actually uses radiation or chemicals. For instance, we could get

catt	ag	gagtat
cat	tagg	ag	tat
ca	tta	gga	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. Algorithms come into play at Step 4. In Step 4, it is not always possible to reconstruct the exact original genome and indeed there might be many DNA strings that lead to the same collection of fragments (consider, for example, just repeating the original string).

We therefore want to solve a problem that is of the form: *Given a set of overlapping genome subsequences, construct the "best" sequence that includes them all.* However, this is still not a well formed problem. What does it mean to be the "best" sequence? There are many possible candidates. Below is one way to define "best" objectively.

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

In this definition the notation Σ^+, the "Kleene plus", means the set of all possible non-empty strings consisting of characters Σ. Note that in this definition, we require each $s \in S$ to appear as a contiguous block in r. That is, "ag" is a substring of "ggag" but is *not* a substring of "attg". That is, given sequence fragments, construct the shortest string that contains all the fragments. The idea is that the simplest string is the best.

Now that we have a concrete specification of the problem, we are ready to look into algorithms for solving it. For starters, let's observe that we can ignore strings that are contained in other strings. That is, for example, if we have gagtat, ag, and gt, we can throw out ag and gt. Continuing with the example above, we are left with the following "snippets"

$$S = \left\{\textsf{tagg}, \textsf{catt}, \textsf{gga}, \textsf{tta}, \textsf{gagtat}\right\}.$$

We can thus 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, because otherwise there would be a string that is a substring of another. Since they start at distinct positions, we can totally order the strings in S. This ordering thus defines a solution.

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

3.2 Algorithm 1: Brute Force

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

Definition 3.2 (The Brute Force **Technique**)**.** Enumerate all possible candidate solutions for a problem, score each one, and return a best solution.

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

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

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

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

Does trying all permutations always give us the shortest string? As our intuition might suggest, the answer is yes and the proof of it hints at an algorithm that we will look at in a moment.

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

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

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

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

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

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.

3.3 Algorithm 2: Reducing to Another Problem

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

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

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

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

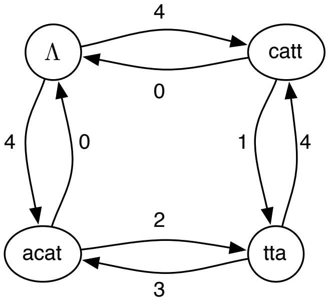
Motivated by the observation that the shortest superstring problem can be solved exactly by trying all permutations, we'll make the TSP problem try all the permutations for us. For this, we will set up a graph so that each valid Hamiltonian cycle corresponds to a permutation. The graph will be complete, containing an edge between any two vertices, and guaranteeing the existence of a Hamiltonian cycle.

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

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

- The vertex set V has one vertex per string and a special “source” vertex Λ where the cycle starts and ends.
- The arc (directed edge) from s_i to s_j has weight $w_{i,j} = |s_j| - \texttt{overlap}(s_i, s_j)$. This quantity represents the increase in the string's length if s_i is followed by s_j. As an example, if we have “tagg” followed by “gga”, then we can generate “tagga” which only adds 1 character giving a weight of 1—indeed, |“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

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

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

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

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

3.4 Algorithm 3: Greedy

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4 What did we review and learn in this lecture?

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