Lecture 1 — Overview and Sequencing the Genome

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2013)

Lectured by Guy Blelloch — January 13, 2014

1 Administration and Policies

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."

Textbook. We will make available an online book for your use with the class. We will do our best to cover all the material in book but we might not be able to due to time constraints. You are responsible for all the material in the book, including the parts that are not covered in the lecture.

Communications with the course staff. When you need help from the course staff, please consider using piazza first (see piazza usage etiquette below). If piazza is not appropriate then you can email us at 15210-staff@andrew.cmu.edu. Please refrain emailing professors or individual TA's, except in cases that require privacy.

Web site. The course web site is located at

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

Collaboration Policy. To facilitate cooperative learning, it is permissible to discuss a homework assignment with other students, provided that the following whiteboard policy is respected. A discussion may take place at the whiteboard (or using scrap paper, etc.), but no one is allowed to take notes or record the discussion of what is written on the board, and you must allow two hours to lapse after any discussion before working on the assignment. The fact that you can recreate the solution from memory is taken as proof that you actually understood it.

It is not acceptable to share your solutions or give hints to your friends for a lab after you have already discovered the correct idea. You are not helping your friends by doing so. The right thing to do is to not talk about the lab after you have a solution, and anyone struggling with

the homework should visit office hours to talk to an instructor or TA. This shows the respect deserved by your friends as well as the people who have put a lot of effort into creating the labs.

We may sometimes run automatic code comparison programs (such as MOSS). These programs are very good at detecting similarity between code, even code that has been purposefully obfuscated. Such programs can compare a submitted assignment against all other submitted assignments, against all known previous solutions of a problem, etc. The signal-to-noise ratio of such comparisons is usually very distinctive, making it very clear what code is a student's original creative work and what code is merely transcribed from some other source.

Assignments. There will be weekly assignments, 2 exams, and a final. The first assignment will come out later today and will be due one week from now. The assignments will be handed out, tested, and turned in via autolab. For each test we will have a suite of public tests that will help you in doing the assignment and a suite of private tests that will help us grade.

Late and erroneous submissions. We built in a generous grace period to autolab that prevents any last minute problems from arising when turning in your homework. We therefore will not accept any late submissions sent to us via for example email.

Before submitting your work make sure that it is the most recent version. Check also the submission feedback. We will not accept revisions after the deadline.

Late days. For the duration of the semester, you will have eight late days that you can use for whatever reason. However, you cannot use more that 2 late days per assignment. We will not be able to grant you any additional late days for any reason other than a medical condition that seriously undermines your ability to do your work, by for example hospitalizing you for more than 2 days—common cold, flu, etc are not sufficient. In case of a serious medical condition, please ask your academic adviser to email the professors. Please do not email us personally.

Word of Caution. You will likely find this course to be difficult. There are several reasons for why. First, the material covered in the course, with emphasis on parallelism, will be new to many of you. Second, the way we design our algorithms and implement them, with emphasis on higher-order programming (where functions are first-class values), can be difficult to grasp quickly, though over time you will likely not be able to imagine thinking without them. Third, this course will require generating your own algorithms in addition to understanding existing algorithms. If you don't know Standard ML, then there will also be the additional overhead of learning a new programming language, and you should start learning it immediately. There are many online resources for doing so.

It is thus important for you to mentally prepare yourself for a difficult course. If you do your work, we are confident that you will finish this class with a satisfactory grade. As you will discover throughout the semester, we have an excellent set of teaching assistants that can help great assistance. That said, you should keep in mind three things 1) there is no substitute for

doing your own work, and 2) there is no substitute for doing your own work and 3) the first two things.

One more important point: Albert Camus once said something like this: "an intellectual is one whose mind watches itself." It is important that you think about this. I would say more broadly that to have a interesting life you should learn to watch your mind. How can you watch your mind? For the purposes of this class here, it is important that you understand that your thoughts and beliefs have reasons to be there and that if those reasons change then those thoughts and beliefs will also change. For example, 10 years from now, you are going to find that you worried too much about your grades in college, and that you would have been better off if you worried less. Now, just knowing this, which many us can tell you, will not be sufficient to change your thoughts about the importance of grades. But understanding that such thoughts can be a result of years of conditioning starting from an early age can help you understand why you think that grades are important. Now, those days are over. It is best to adapt as soon as you can by starting to let go off the thought that grades are as important as you (might) think. In college, what is important is for you to find what you love doing and become good at that. In order to find that you have to be ready at trying and failing, and having fun doing so. So bottom line, don't worry about your grades. Worry about learning. The rest will come (even if things at times may appear not as rosy).

Piazza. We will post announcements, clarifications, corrections, hints, etc. on the course web site and piazza—please check them on a regular basis. When using Piazza we ask that you follow the following guidelines:

- **When to ask a question:** Piazza is a great tool but it can create a temptation to ask every question that you may have, especially when you know that you will get a quick response. Resist that temptation as it is not good for your learning. To learn well and to learn how to think like a creative computer scientist, it is important that you give your mind the time it needs to learn. That means thinking about the questions by yourself on your own and doing the needed research on your own before seeking an answer. So in summary resist that temptation to ask as much as possible. This does not apply to certain questions. For example, a bug in the homework distribution that you just discovered.

- **Private versus public questions:** It is important that you make your questions public rather than private because otherwise the course staff have to respont to the same question many times. (Private questions cannot be seen by your classsmates.) Thus except in special circumstances (e.g., you found a bug in the code distribution), avoid using private questions. As a rule of thumb if you don't feel comfortable making your question public, then it might not be an appropriate question for Piazza at all. For example, in general, posting a piece of code that you wrote and asking the TA's to correct it for you is not an appropriate question.

How to learn well. We would recommend: 1) taking notes (the physical action of taking notes will help you learn), 2) as you read the course notes, write down a list of questions and try to solve them, 3) develop a habit of formulating variants of exercises and homeworks that you are given and solving them and coming up with new questions, 4) think about and research the questions that you may have by yourself and try to find the answers on your own (for example for 24 hours), before talking to your friends and course staff. Finally, if you are not used to doing these, I would recommend that you start working on them and being patient.

Academic integrity policy. You will risk failing 15210 and being reported to the academic administration if you violate the academic integrity policy of the university. More specifically, any work submitted as a homework assignment or examination must be entirely your own and may not be derived from the work of others, whether a published or unpublished source, the worldwide web, another student, other textbooks, materials from another course (including prior semesters of this course), or any other person or program. You may not copy, examine, or alter anyone else's homework assignment or computer program, or use a computer program to transcribe or otherwise modify or copy anyone else's files.

If you cheat and you don't hear from us immediately, you may feel that we have missed it. This is unlikely but it will take some time for us to get back to you because of all the checks that we have to go through. Yes we admit dealing with cheaters is a hassle for the course staff but this does not mean we avoid doing the work. We have developed techniques for dealing with this admittedly unpleasant issue and did and will employ them. That said, it is not to your advantage to cheat. As explained above, your focus in college should be to learn and to learn how to learn. This chapter of your life is about learning and discovering new things—it is not about grades or passing courses. Failures are great opportunities to learn, to grow, and to find what you love to do and what you don't. Don't miss out on them by cheating!

2 Book Overview

This course aims to cover the algorithms and data structures, with an emphasis on parallelism, and abstraction (the separation of specification and implementation).

3 Parallelism

Question 3.1. *Why should we care about parallelism?*

There are many reason for why parallelism is important. Fundamentally, parallelism is simply more powerful than sequential or serial computation where there is only one line of computation. In parallel computation, we can perform multiple tasks at the same time. Another reason is

efficiency in terms of energy usage. As it turns out performing a computation twice as fast sequentially requires eight times as much energy. Precisely speaking, energy consumption is a cubic function of frequency (speed). With parallelism, we don't need more energy to speed up a computation, at least in principle. For example, to perform a computation in half the time, we need two lines of computation (instead of one) that ran half the time of the sequential, thus consuming the same amount of energy. In reality, there are some overheads and we will need more energy, usually only a constant fraction more. These two factors—power and energy—has gained importance in the last decade catapulting parallelism to the forefront of computing.

Parallel hardware. Today, it is nearly impossible to avoid parallelism. For example, when you do a simple web search, you are engaging a data center in some part of the world (likely near your geographic location) that houses thousands of computers. Many of these computers (perhaps as many as hundreds, if not thousands) take up your query and sift through data to give you an accurate response as quickly as possible. This form of parallelism may be viewed as large-scale parallelism, as it involves a large number of computers. [1]

Another form of parallelism involves much smaller numbers of processors. For example, portable computers today have chips that have 4, 8, or more processor cores. Such chips, sometimes called *multicore chips*, are predicted to spread and provide increasing amount of parallelism over the years. For example, using current chip technology, it is not difficult to put together several multicore chips in a desktop machine to include 60 cores. While multicore chips were initially used only in laptops and desktops, they are also becoming used common in smaller mobile devices such as phones due to their low-energy consumption (many mobile phones today have 4- or 8- core chips.)

In addition to the aforementioned parallel systems, there has been much interest in developing hardware for specific tasks. For example, GPU's can fit as much as 1000 small cores (processors) onto a single chip. Intel's Mic (or Xeon Phi) architecture can host several hundred processors on a single chip.

Question 3.2. *Can you think of consequences of these developments in hardware?*

Parallel thinking. These developments in hardware 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.

Question 3.3. *What is the advantage of using a parallel algorithm instead of a sequential one+*

[1] Of course, terms such as "large" are relative by definition. What we call large-scale today may be consider small scale in the future.

Parallel software. The most important advantage of using a parallel instead of a sequential algorithm 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 as well as highly tuned implementations. In this course, we will focus on the first challenge.

Challenges of parallel software. The many forms of parallelism, ranging from large to small scale, and from general to special purpose, currently requires many different languages, libraries, and implementation techniques. For example, it is unlikely one can obtain the kinds of speedups that we discussed above from unoptimized software. This diversity of hardware and software makes it a challenge 1) to develop parallel software and 2) to learn and to teach parallelism. For example, we can easily spend weeks talking about how we might optimize a parallel sorting algorithm for a specific hardware.

This course: parallelism. Maximizing speedup by highly tuning an implementation is not the goal of this course. That is an aim of 15-213. In this course, we aim to cover the general design principles for parallel algorithms that can be applied in essentially all parallel systems, from the data center to the multicore chips on mobile phones. We will learn to think about parallelism at a high-level, learning general techniques for designing parallel algorithms and data structures, and learning how to approximately analyze their costs. The focus is on understanding when things can run in parallel, and when not due to dependences. As in 15-150, in this class we will be using *work* and *span* to analyze costs. Together these measures give you a sense of how well an algorithm will work on machines with a range of number of processors.

The algorithms that we design in this course will be purely functional. By functional we mean that they are based on the composition of functions and make use of functions that take

other functions as arguments (higher-order functions). By pure, we mean that our algorithms have no observable side effects. In particular, all our algorithms and their components take an input and return an output, and leave the rest of the world, including the input, alone. This might seem like a radical change from a standard algorithms and data-structures course, but in reality it just means one needs to return new data instead of modifying existing data. For example, if we are inserting a value into a balanced binary tree, we need to return a new tree instead of updating an existing tree. This might seem inefficient, possibly requiring $O(n)$ work for a tree with n nodes. However, all we need to do is create new nodes along the path to where the node is inserted, requiring only $O(\log n)$ work.

Question 3.4. *Can you think of reason for why purely functional programming can help in designing and implementing parallel algorithms?*

One reason for why purely functional programming can help in the design and implementation of parallel algorithms is that purely functional programs are safe for parallelism: any components can be executed in parallel without affecting each other. In an imperative setting, we need to worry about how the parallel components affect each other. In particular if they do affect each other then depending on the exact timing, we might get very different results. Such affects that can change outcomes based on timing are called *race conditions*. Race conditions make it much harder to reason about the correctness and the efficiency of parallel algorithms. They also make it harder to debug parallel code since each time the code is run, it might give a different answer.

Another reason functional languages (even when not pure) help with parallel thinking is that higher-order functions encourage high-level parallel constructs. For example, instead of thinking about a loop that adds the elements of an array together into a sum, which is completely sequential, we think of a general higher order "reduce" function. In addition to taking the array as an argument, the reduce takes a binary associative function as another argument. It then sums the array based on that binary associative function. The advantage for parallelism is that the reduce can use a tree of sums instead of summing one after the other. It also allows for any binary associative function (e.g. maximum, minimum, multiplication, ...). In general, thinking in higher order functions encourages working at a higher level of abstraction, moving us away from the one-at-a-time (loop) way of thinking that is detrimental to parallelism.

Even though the algorithms that we design are purely functional, this does not mean that they cannot be implemented in imperative languages—one just needs to be much more careful when coding imperatively. Some imperative parallel languages, in fact, encourage programming purely functional algorithms. The techniques that we describe thus applicable in the imperative setting as well.

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

4 Specification and Implementation

In this course, we will carefully distinguish between interfaces/specifications and design/implementation.

An *interface* or *specification* defines precisely what we want of a function or a data structure. A *design* or an *implementation* describes how to meet the specification. In other words specifications and designs refer to the *what* and the *how*. What we want a function or data structure to achieve and how to do that.

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

Functions and data. In computing, is is broadly possible to distinguish between **functions** that perform actual computation and **data** which serve as the subject of computation. For each of these two we can distinguish between the **interface** and **implementation** as indicated in the table above.

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.

Example 4.1. *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.*

Similarly an *abstract data type* (ADT) specifies precisely an interface for operating on data in an abstract form. The interface will typically consist of a set of functions for accessing or manipulating the particular data type. The interface, however, does not specify how the data is structured or how the functions are implemented. This is hidden by the ADT.

A *data structure*, on the other hand, implements the interface by organizing the data in a particular form, typically in a way that allows an efficient implementation of the functions.

Example 4.2. *For example, a priority queue is an ADT with functions 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.

Question 4.3. *Why do we need to distinguish between interfaces and implementations.*

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.

5 Introduction and Genome Sequencing

We discuss some of the fundamental concepts such as the notions of problem and algorithm, interface and specification, in the context of the genome sequencing problem, and start discussing the techniques for algorithm design and analysis.

6 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

Figure 1: Wild poppies by Claude Monet. Note the two copies of woman and child. The painting describes a dependency between the two instances of time Monet was interested in showing that painting can be more powerful than a camera because it can illustrate different points in time in the same frame.

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 results—and the power of problem abstraction which will reveal surprising connections between seemingly unrelated problems.

Question 6.1. *Why do you think sequencing the genome is a difficult problem?*

What makes sequencing the genome hard is that 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 over three billion in the whole genome. We therefore resort to cutting strands into shorter fragments and then reassembling the pieces.

Primer walking. A technique called "primer walking" can be used to cut the DNA strands into consecutive fragments and sequence each one. Each step of 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. Note that primer walking is an inherently sequential technique as a step depends on the previous, making it difficult to parallelize and thus speed up.

Question 6.2. *Can you think about a way to parallelize primer walking?*

One way to parallelize primer walking is to divide the genome into a many fragments and sequence them all in parallel. But the problem is that we don't know how to put them together, because we have mixed up the fragments and lost their order.

Figure 2: A jigsaw puzzle.

Figure 3: Water lilies by Claude Monet. We think of the patches of color as water lilies even though they are just that, patches of color.

Exercise 6.3. *When cut, the strand* cattaggagtat *might turn into,* ag, gag, catt, tat, *destroying the original ordering.*

Question 6.4. *The problem of putting together the pieces is a bit like solving a jigsaw puzzle. But it is harder. Can you see why? Can you think of a way of turning this into a jigsaw puzzle that we can solve?*

The shotgun method. When we cut a genome into a fragments, we lose all the information that we have about how to connect them back. If we had some information about how to relate different pieces, we can imagine solving this problem just as we solve a jigsaw puzzle (by relying on our "meta" knowledge about what the whole picture that we are trying to reconstruct is).

Question 6.5. *Can you think of a way to relate different pieces?*

We can relate different pieces if we could make copies of the original sequence and generate many fragments. When a fragment overlaps with two others, it can tell us how to relate them. This is the idea behind the shotgun (sequencing) method, which today seems to be the standard technique for genome sequencing. The shotgun method works as follows:

1. Take a DNA sequence and make multiple copies.
2. Randomly cut up the sequences using a "shotgun" that actually uses radiation or chemicals.
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. This is the interesting algorithmic component.

Steps 1–3 are done in a wet lab, while step 4 is the interesting algorithmic component.

Example 6.6. *For example, for the sequence* cattaggagtat*, we produce three copies:*

cattaggagtat
cattaggagtat
cattaggagtat

We then divide each into fragments

catt | *ag* | *gagtat*
cat | *tagg* | *ag* | *tat*
ca | *tta* | *gga* | *gtat*

Note how each cut is "covered" by an overlapping fragment telling us how to reverse the cut.

Question 6.7. *In step 4, is it always possible to reconstruct the sequence?*

It is not always possible to reconstruct the exact original genome in step 4. For example, we might get unlucky and cut all sequences in the same location. Even if we do cut them in different locations there are many DNA strings that lead to the same collection of fragments. For example, just repeating the original string twice can lead to the same set of fragments if the two sequences are always cut at their seam.

Defining the problem. We will therefore do our best is constructing the original sequence.

Question 6.8. *How can we make this intuitive notion of "doing our best precise"?*

It is not easy to make this notion of "doing our best" precise. This is why, it can be as difficult and important to formulate a problem as it is to solve it.

Figure 4: William of Occam (or Ockham, 1287-1347) posited that among competing hypotheses that explain some data, the one with the fewest assumptions or shortest description is the best one. The term "razor" apparently refers to shaving away unnecessary assumptions, although here is a more modern take on it.

Question 6.9. *Can you think of a property that the result needs to have in relation to the snippets?*

Note that since the fragments all come from the original genome, the result should contain all snippets. In other words, it is a *superstring* of the fragments. As mentioned earlier, however, there will be multiple superstrings for any given set of fragments.

Now of all the superstrings, which one should we pick? We can take one more step in making the problem more precise by constructing the "best" superstring. How about the shortest superstring? This would give us the simplest explanation, which is often desirable. The principle of selecting the simplest explanation is often referred to as Occam's razor (see Figure 4). This is how we will define the problem.

Definition 6.10 (The Shortest Superstring (SS) **Problem**). *Given an alphabet set Σ and a set of finite-length strings $S \subseteq \Sigma^+$, return a shortest string r that contains every $s \in S$ as a substring of r.*

That is, given a set of fragments, construct the shortest string that contains all of them. In the definition the notation Σ^+, the "Kleene plus", means the set of all possible non-empty strings consisting of characters Σ. For the genome sequencing application, we have $\Sigma = \{a, c, g, t\}$. Note that, for a string s to be a *substring* of another string r, s must as a contiguous block in r. That is, "ag" is a substring of "ggag" but is *not* a substring of "attg".

We have now converted a vague problem, sequencing the genome, into a concrete problem, the SS problem. As discussed at the end of this chapter, the SS problem might not be exactly the right abstraction for the application of sequencing the genome, but it is a good start.

Having specified the problem, we are ready to design an algorithm, in fact a few algorithms, for solving it. Let's start with some observations.

Observation 1: Snippets. Note that we can ignore strings that are contained in other strings. For example, if we have gagtat, ag, and gt, we can throw out ag and gt. We will refer to the fragments that are not contained in others as *snippets*.

Example 6.11. *In our example, we had the following fragments.*

catt ag gagtat
cat tagg ag tat
ca tta gga gtat

Our snippets are now:

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

The other strings {cat, ag, tat, ca, gtat} *are all contained within the snippets.*

Observation 2: snippet ordering. By taking any ordering of the snippets and removing the maximum overlap between each adjacent pair of snippets we get a superstring of the snippets.

Example 6.12. *For our running example, consider the following ordering*

catt tta tagg gga gagtat

When we remove the maximum overlaps (the excised parts are underlined) we get cattaggagtat*, which is indeed a superstring.*

This observation can be proven by induction. The base case is true since it is clearly true for a single snippet. Inductively, we assume it is true for the first i snippets, i.e. that removing the maximum overlap between adjacent snippets among these i is a superstring of these i snippets. We refer to this superstring as s_i. We now argue that based on this assumption the property is true for the first $i + 1$ snippets. Note that when we add the $i + 1^{th}$ snippet it cannot fully overlap with the previous snippet, by the definition of snippets. Therefore when we add it on, the string s_{i+1} will be s_i with some new characters added to the end. This will still be a superstring of all the first i snippets since we did not modify s_i, plus it will also be a superstring of the $i + 1^{th}$ snippet, since we are including it at the end.

Observation 3: shortest snippet ordering. Our shortest superstring for a set of snippets will correspond to some ordering of snippets with overlaps removed. This observation is true since in the shortest superstring s each snippet has to start at a different location (otherwise they would overlap). By a similar inductive argument as above, we can show that if it is true for the first i, then it is true for the first $i + 1$.

We are now ready to solve the SS problem by designing algorithms for it. Designing algorithms may appear to be an intimidating task, because it may seem as though we would need brilliant ideas that come out of nowhere. Like the water lilies of Monet, this is just an appearance. In reality, we design algorithms by starting with simple ideas based on several well-known techniques and refine them until we reach the desired result, much like a painter constructing a painting with simple brush strokes.

In the rest of this section, we will consider three algorithmic techniques that can be applied to this problem and derive an algorithm from each.

Question 6.13. *Before we start looking at algorithms let's think about why this problem may be hard. Consider jigsaw puzzles, what makes them "hard"?*

7 Algorithm-Design Technique 1: Brute Force

Brute-force technique simply consist of trying all candidate solutions and selecting the best. It is often not efficient, because there can be many candidates. It can, however, be the only known way to solve a problem.

Definition 7.1. *Enumerate all possible candidate solutions for a problem, score each candidate, and return a best solution.*

Question 7.2. *What are the candidate solutions for the genome sequencing problem?*

Well we already said the optimal solution corresponds to some ordering of the snippets with overlaps removed. We can therefore consider all orderings, or equivalently all permutations. Thus when sequencing the genome with the brute-force technique, candidate solutions consist of such permutations.

Question 7.3. *How can we score each permutation?*

To score each permutation, we can use the length of the permutation after removing the overlaps.

Exercise 7.4. *Try a couple other permutations and determine the length after removing overlaps.*

Question 7.5. *Does trying all permutations always give us the shortest superstring?*

As our intuition might suggest, by trying all permutations, we can indeed find the shortest superstring. The proof of this intuiton hints at an algorithm that we will look at in a moment.

Lemma 7.6. *Given a finite set of finite strings $S \subseteq \Sigma^+$, the brute force method finds the shortest superstring.*

Proof. By Observation 3, one of the orderings will give the shortest superstring. By Observation 2 all other orderings will give a valid superstring. All these have to be at least as long as the shortest superstring so if we score by length, then the algorithm will return the shortest. □

Question 7.7. *Is the brute-force approach a good parallel algorithm?*

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. To understand whether this is a good parallel algorithm or not, we have to consider the work and the span.

Although highly parallel, the brute-force algorithm 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.

Question 7.8. *Can we come up with a smarter algorithm that solves the problem faster?*

Unfortunately the SS problem turns out to be NP-hard, although we will not show this.

Question 7.9. *Is there no way to efficiently solve an instance of an NP-hard problem?*

When a problem is NP hard, it means that there are *instances* of the problem that are difficult to solve. NP-hardness doesn't rule out the possibility of algorithms that quickly compute near optimal answers or algorithms that perform well on real world instances. For example the type-checking problem for the ML language is NP-hard but we use ML type-checking all the time without problems.

Figure 5: 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.

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.

8 Algorithm-Design Technique: Reduction

Another approach to solving a problem is to reduce it to another problem which we understand better and for which we know algorithms, or possibly even have existing code. 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 using polynomial work you can reduce an NP-complete problem A to problem B, 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.

Question 8.1. *Are you all familiar with the TSP problem?*

This is a canonical NP-hard problem dating back to the 1930s and has been extensively studied, e.g. see Figure 5. 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 8.2 (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).

You can think of the TSP problem as the problem of coming up with best possible plan for your annual road trip.

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.

Question 8.3. *Can we set up the TSP problem so that it tries all 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 thus 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.

Example 8.4. *For "tagg" and "gga", we have* $\texttt{overlap}$ *("tagg","gga")* $= 2$.

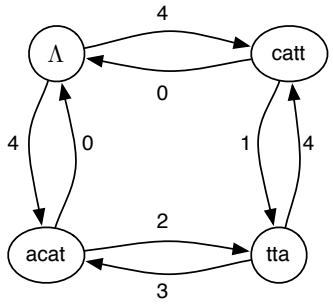
The Reduction. Now we build a graph $D = (V, A)$.

- The vertex set V has one vertex per snippet 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.

For 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"|$-$`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. If s_i is the last string we have already paid for it, so the arc (s_i, Λ) is free.

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



Question 8.5. *What does a Hamiltonian cycle in the graph starting at the source correspond to? What about the total weight of the edges on a cycle?*

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.

Question 8.6. *Is there a cycle in the graph for each permutation?*

Note that since the graph is complete, we can construct a cycle for each permutation by visiting the corresponding vertices in the graph in the specified order.

We have thus established an equivalence between permutations and the Hamiltonian cycles in the graph.

Since TSP considers all Hamiltonian cycles, it considers 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 the TSP problem, we would be able to solve the shortest superstring problem.

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 perhaps this helps.

Figure 6: A view of the Makalu mountain (first climbed in 1955) from mount Everest. Can you identify the good places to start for a successful summit using the greedy approach?

9 Algorithm-Design Technique 3: Greedy

We now consider a third technique, the "greedy" technique, and a corresponding algorithm.

Definition 9.1 (The Greedy **Technique**). *Take a sequence of steps, and on each step make a locally optimal decision based on some criteria without ever backtracking on previous decisions.*

Question 9.2. *Does the greedy technique always return the optimal solution?*

The greedy technique (or approach) is a heuristic that in some cases returns an optimal solution, but in many cases it does not. For a given problem there might be several greedy approaches that depend on the types of steps and what is considered to be locally optimal. The greedy approach for the SS problem we now consider does not guarantee that we will find the optimal solution, but it can guarantee to give a good approximation. furthermore it works very well in practice. Greedy algorithms are popular because of their simplicity.

Remark 9.3. *You can think of a greedy algorithm as taking a walk in a hilly terrain where each step in taken in the direction of the steepest slope in the hope that it will lead to the highest peak. Thus, if you start at a place close to the summit and all peaks are close to each other in terms of elevation, then you will find peak that is guranteed to have an altitude that is close to the summit.*

Question 9.4. *Considering that we want to minimize the length of the result, what should our "greedy choice" be?*

To minimize the length of the string, we should maximize overlap. Thus our algorithm will simply pick a pair of snippets with the largest overlap and join them by placing one immediately after the other and removing the overlap.

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

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

Listing 1: Greedy Approximate SS Algorithm

Listing 1 shows pseudocode for our greedy algorithm. In this course the we will be using mathematical set notation in our pseudocode in the notes and in lecture. The reason for this is that it is concise and to emphasize that the course is not about a particular language (e.g. ML) but about mathematical and algorithmic concepts. We advise that you become familiar with this notation.

Given a set of strings S, the *greedyApproxSS* algorithm checks if the set has only 1 element, and if so returns that element. Otherwise it finds the pair of distinct strings s_i and s_j in S that have the maximum overlap. It does this by first calculating the overlap for all pairs (Line 6) and then picking the one of these that has the maximum overlap (Line 7). Note that O is a set of tripples each corresponding to an overlap and the two strings that overlap. The notation $\arg\max_{(x,\_,\_)\in O} x$ is mathematical notation for selecting the element of O that maximizes the first element of the triple, which is the overlap. After finding the pair (s_i, s_j) with the maximum overlap, the algorithm then replaces s_i and s_j with $s_k = \texttt{join}(s_i, s_j)$ in S.

Question 9.5. *Is the algorithm guaranteed to terminate?*

Note that the new set S' is one smaller than S and that the algorithm recursively repeats this

process on this new set of strings until there is only a single string left. It thus terminates after $|S|$ recursive calls.

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 9.6. *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 = `join`(s_i, s_j). Argue why there cannot be any such strings.*

Exercise 9.7. *Prove that algorithm* `greedyApproxSS` *indeed returns a string that is a superstring of all original strings.*

Exercise 9.8. *Give an example input S for which* `greedyApproxSS` *does not return the shortest superstring.*

Exercise 9.9. *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?*

Parallelizing the greedy algorithm. 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.

Approximation quality. 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 performs 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*.

Remark 9.10. *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.*