Chapter 1

Introduction

The topic of this course might best be described as **problem solving with computers**. The idea is you have some problem to solve (e.g. finding the shortest path from your dorm to your 15-210 class), and you want to use your computer to solve it for you. Your primarily concern is probably that your answer is correct (e.g. you would likely be unhappy to find yourself at some nursing class at Pitt). However, you also care that you can get the answer reasonably quickly (e.g. it would not be useful if your computer had to think about it until next semester).

As such, this course is about several aspects of such problem solving with computers. It is about cleanly defining the problem you want to solve. It is about learning the different techniques that can be used to computationally solve such a problem, and about designing specific algorithms using these techniques. It is about designing abstract data types that can be used in these algorithms, and data structures that implement these types. And, it is about analyzing the cost of those algorithms and comparing them based on these costs. Unlike most courses on algorithms and data structures, which just consider sequential algorithms, in this course we will focus on how to design and analyze parallel algorithms and data structures.

In the rest of this chapter we will consider why it is important to consider parallelism, the importance of separating interfaces from implementation, and outline some the techniques for algorithm design.

1.1 Parallelism

Question 1.1. *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 1.2. *Can you think of consequences of these developments in hardware?*

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.

Question 1.3. *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.

Parallel Thinking. Maximizing speedup by highly tuning an implementation is not the goal of this course. That is an aim of other more advanced courses. 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.

Work and Span. In this book we analyze the cost of algorithms in terms of two measures: *work* and *span*. Together these measures capture both the sequential time of an algorithm and

the parallelism available. We typically analyze both of these in terms of big-O, which will be described in more detail in Chapter 3.

The *work* of an algorithm corresponds to the total number of primitive operations performed by an algorithm. If running on a sequential machine, it corresponds to the sequential time. However, when running in parallel it might be possible to share work across multiple processors, therefore reducing the time. The question is to what extent can we do such sharing. Ideally we would like to achieve close to linear speedup: with W work *linear speedup* means that the computation will take $\frac{W}{P}$ time on P processors. Getting close to linear speedup would mean we are making almost perfect use of our resources. However, this is not always possible. If our algorithm is fully sequential, for example, we can only take advantage of one processor and the time would not be improved at all by adding more.

The purpose of the second measure, *span*, is to analyze to what extent the work of an algorithm can be shared among processors. The *span* of an algorithm basically corresponds to the longest sequence of dependences in the computation. It can be thought of the time it would take if we had an unlimited number of processors on an ideal machine. However, we do not have an unlimited number of processors, but instead probably care how well it does on some fixed number of processors. What is particularly convenient about analyzing the work and span of an algorithm is that by using just these two measures we can roughly predict the time on any number of processors even without having to schedule the computation onto processors ourselves. Furthermore we can predict the largest number of processors for which we we can get close to linear speedup. This is called the parallelism of the algorithm.

When programming and analyzing algorithms in terms of work and span we assume there is a runtime scheduler that schedules the algorithm onto processors. Consider a scheduler that runs on P processors that has the following greedy property: no processor will sit idle (not work on some part of the computation) if there is work ready to do. Such a scheduler is called a *greedy scheduler*. It can be shown that when using a greedy schedule on a computation with W work and S span, the runtime will be at most

$$T \leq \frac{W}{P} + S \ .$$

If the first term dominates, then we are getting close to linear speedup. We therefore define the *parallelism* $\mathcal{P}$ of an algorithm to be the number of processors at which the two terms are equal, which gives

$$\mathcal{P} = \frac{W}{S} \ .$$

For $P = \mathcal{P}$ we are getting half of linear speedup. For any $P > \mathcal{P}$ the speedup is worse, and for any $P < \mathcal{P}$ it is better. Therefore $\mathcal{P}$ gives us a rough upper bound on the number of processors we can effectively use.

Example 1.4. *As an example, consider the mergeSort algorithm for sorting a sequence of length n. The work is the same as the sequential time, which you might know is*

$$W(n) = O(n \log n) \ .$$

We will show that the span for mergeSort is

$$S(n) = O(\log^2 n) \ .$$

The parallelism is therefore

$$\mathcal{P} = O\left(\frac{n \log n}{\log^2 n}\right) = O(n/\log n) \ .$$

This means that for sorting a million keys, we can effectively make use of quite a few processors: $10^6/(\log_2 10^6) \approx 50,000$. *In practice we might not be able to make good use of this many because of overheads, and constants hidden in the big-O, but we are still likely to make good use of over* 1000 *processors.*

Question 1.5. *How do we calculate the work and span of an algorithm?*

In this book we will calculate the work and span of algorithms in a very simple way that just involves composing costs across subcomputations. Basically we assume that sub-computations are either composed sequentially (one is done after the other) or in parallel (they can be done at the same time). Informally, the rules are simply the following:

	work (W)	**span** (S)
Sequential composition	$1 + W_1 + W_2$	$1 + S_1 + S_2$
Parallel composition	$1 + W_1 + W_2$	$1 + \max(S_1, S_2)$

where W_1 and S_1 are the work and span of the first subcomputation and W_2 and S_2 of the second. The 1 that is added to each rule is the cost of composing the computations.

These rules are hopefully intuitive since whether we do things sequentially or in parallel the total work always adds. Whether I cook 10 eggs and then you cook 15 eggs, or we do them at the same time, the total work we do is cooking 25 eggs. On the other hand the span represents the longest sequence of dependences. For the eggs if I cook my 10 and then when I'm done you cook your 15, then it will take 25 eggs worth of time to finish. On the other hand if we cook them together at the same time, it will only take $\max(10, 15) = 15$ eggs worth of time (I'll have to wait for you to finish). More formal rules are given in Chapter 3.

In summary there are several advantages of analyzing algorithms in terms of work and span.

1. The work tells us the sequential time of an algorithm
2. The span tells us the ideal speedup
3. The work and span give us a rough sense of how many processors we can effectively make use of (i.e. get close to linear speedup).
4. We don't have to schedule the computations onto processors.
5. It is relatively easy to calculate work and depth by composition.

Remark 1.6. Truth in advertising. *Work and span only give a rough estimate of cost and parallelism. There are many costs that are not included in the analysis, such as the overhead for a scheduler, and the cost for communicating among processors. These are topics of a more advanced course.*

1.2 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, it 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 1.7. *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 those numbers), while* 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 1.8. *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.*

Some ADTs we will cover in this course include: sequences, sets, ordered sets, tables, priority queues, and graphs.

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 ambiguous usage in this class.

Question 1.9. *Why is it important 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 in the next chapter. 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.

1.3 Algorithmic Techniques

One of the things that students find most difficult is coming up with their own algorithms. One difficulty with this is that one does not know where to start.

Question 1.10. *Have you designed algorithms before. If so what approaches have you found helpful?*

Given an algorithmic problem, where do you even start? It turns out that most of the algorithms follow several well-known techniques. Here we will outline some such techniques (or approaches), which are key to both sequential and parallel algorithms. We note that the definition of these approaches is not meant to be formal, but rather just a guideline.

Remark 1.11. *To become good at designing algorithms, we would recommend gaining as much experience as possible by both formulating problems and solving them by applying the techniques that we discuss here.*

Brute Force: The brute force approach involves trying all possible (underlying) solutions to a problem. In particular the approach enumerates all candidate solutions, and for each it checks if it is valid, and either returns the best valid solution, or any valid solution. For example, to sort a set of keys, we can try all permutations of those keys and test each one to see if it is sorted. We are guaranteed to find at least one that is sorted.

Question 1.12. *When might we find more than one valid solution (permutation) for the sorting problem? In this case do we care which one?*

However, there are $n!$ permutations and $n!$ is can be very large even for modest n—e.g. $100! \approx 10^{158}$. Therefore this approach to solving the sorting problem is not "tractable" for large problems, but it might be a viable approach for small problems. In some cases the number of candidate solutions is much smaller. In a later chapter, we will see that a brute force method for the maximum contiguous subsequence sum (MCSS) problem only requires trying about n^2 candidate solutions on a string of length n. However, as we will see, this is still not an efficient approach to solving the problem.

Question 1.13. *Does the brute-force approach parallelize well?*

The approach of trying all solutions is typically easy to parallelize. Most often enumerating all solutions (e.g. all permutations) is easy to do in parallel, and testing the solutions is inherently parallel. However, this does not mean it leads to efficient parallel algorithms. In a parallel algorithm we primarily care about the total work and only then about the level of parallelism.

Question 1.14. *Given that the brute force algorithms are often inefficient, why do we care about them?*

One place the brute force approach can be very useful is when writing a test routine to check the correctness of more efficient algorithms. Even if inefficient for large n the brute force approach could work well for testing small inputs. The brute force approach is usually the simplest solution to a problem, but not always.

Note that when solving *optimization problems*, i.e., problems where we care about the optimal value of a solution, the problem definition might only return the optimal value and not the "underlying" solution that gives that value. For example, a shortest path problem on graphs might be defined such that it just returns the shortest distance between two vertices, but not a path that has that distance. Similarly when solving *decision problems*, i.e., problems where we care about whether a solutions exists or not, the problem definition might only return a boolean indicating whether a solution exists and not the valid solution. For both optimization and decision problems of this kind when we say try all "underlying solutions" we do not mean try all possible return values, which might just be a number or a boolean, we mean try all possible solutions that lead to those values, e.g., all paths from a to b.

What is meant by "solution" or "all" is not always clear for a given problem and is open to some interpretation. Indeed in the next chapter we describe two brute force algorithms for a problem called the shortest superstring problem. In one algorithm we consider all possible output strings up to a bounded length and check each one, and in the other we argue that valid solutions must correspond to permutations of the input strings and we therefore consider all possible permutations of the input strings.

Exercise 1.15. *Describe a brute force algorithm for deciding whether an integer n is composite (not prime).*

Reducing to another problem: Sometimes the easiest approach to solving a problem is just reduce it to another problem for which known algorithms exist.

Question 1.16. *You all know about the problem of finding the minimum number in a set. Can you reduce the problem of finding the maximum to this problem?*

Simply invert the signs of all numbers.

Question 1.17. *Can you think of an exercise for becoming better in using this technique?*

When you get more experienced with algorithms you will start recognizing similarities between problems that on the surface seem very different. But one exercise that you might find helpful would be to formulate new algorithmic problems, as this can help in understanding how seemingly different problems may relate.

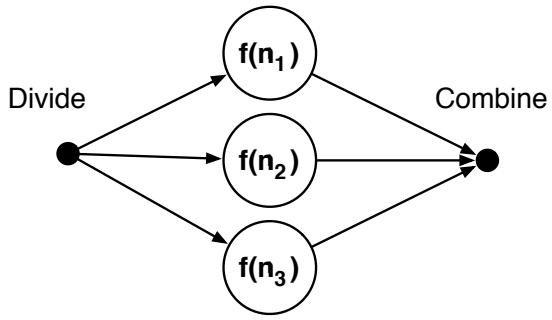


Figure 1.1: Structure of a divide-and-conquer algorithm illustrated ($k = 1$).

Inductive techniques: The idea behind inductive techniques is to solve one or more smaller instances of the same problem, typically referred to as *subproblems*, to solve the original problem. The technique most often uses recursion to solve the subproblems. Common techniques that fall in this category include:

- *Divide and conquer.* Divide the problem on size n into $k > 1$ independent subproblems on sizes $n_1, n_2, \ldots n_k$, solve the problem recursively on each, and combine the solutions to get the solution to the original problem. It is important than the subproblems are independent; this makes the approach amenable to parallelism because independent problems can be solved in parallel.
- *Greedy.* For a problem on size n use some greedy approach to pull out one element leaving a problem of size $n - 1$. Solve the smaller problem.
- *Contraction.* For a problem of size n generate a significantly smaller (contracted) instance (e.g., of size $n/2$), solve the smaller instances recursively, and then use the results to solve the original problem. Contraction only differs from divide and conquer in that it allows there to be only one subproblem.
- *Dynamic Programming.* Like divide and conquer, dynamic programming divides the problem into smaller subproblems and then combines the solutions to the subproblems to find a solution to the original problem. The difference, though, is that the solutions to subproblems are reused multiple times. It is therefore important to store the solutions for reuse either using memoization or by building up a table of solutions.

Randomization: Randomization in is a powerful algorithm design technique that can be applied along with the aforementioned techniques. It often leads to simpler algorithms.

Question 1.18. *Do you know of a randomized divide-conquer algorithm?*

Randomization is also crucial in parallel algorithm design as it helps un "break" the symmetry in a problem without communication.

Question 1.19. *Have you heard of the term "symmetry breaking"? Can you think of a real-life phenomena that is somewhat amusing and can even be uncomfortable where we engage in randomized symmetry breaking?*

We often perform randomized symmetry breaking when walking in a crowded sidewalk with lots of people coming in our direction. With each person we encounter, we may pick a random direction to turn to and the other person responds (or sometimes we do). If we recognize each other late, we may end up in a situation where the randomness becomes more apparent, as we attempt to get past each other's way but make the same (really opposite) decisions. Since we make essentially random decisions, the symmetry is eventually broken.

We will cover a couple of examples of randomized algorithms in this course. Formal cost analysis for many randomized algorithms, however, can (but not always) require knowledge of probability theory beyond the level of this course.

1.4 What makes a good solution?

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

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

Question 1.20. *How do you show that your solution is good?*

Ideally you should prove the correctness and efficiency of your algorithm. For example, in exams, we might ask you do this. Even in real life, we would highly recommend making a habit of this as well. It is often too easy to convince yourself that a poor algorithm is correct and efficient.

Question 1.21. *How can you prove correct an algorithm designed by using an inductive technique? Can you think of a proof technique that might be useful?*

Algorithms designed with inductive techniques can be proved correct using (strong) induction.

Question 1.22. *How can you analyze an algorithm designed by using an inductive technique? Can you think of a technique that might be useful?*

We can often express the complexity of inductive algorithms with recursions and solve them to obtain a closed-form solution.

1.5 Pseudocode

In this course we will use a particular style of pseudocode. It is loosely based on SML, but makes heavy use of mathematical notation. The syntax is given in Figure 1.2 and Figure 1.5 gives some examples. Perhaps most importantly the pseudocode uses a "functional style". Although there is no universal agreement among researchers what functional style means, in our context it means having the following two properties:

1. programs do not have side effects, which is often referred to as being "pure", and
2. programs make use of function composition and higher-order functions.

Not having side effects means that any sub-computation takes an input and returns an output, and leaves the rest of the world, including the input, alone. Since this this is not a course on programming, we will not dive into the advantages of functional programming, but we will mention a few that are particularly useful for parallelism.

Question 1.23. *Can you think of reasons for why purely functional programming can help in designing and implementing parallel algorithms?*

Firstly, functional programs don't have traditional loops. By not having such loops we hope it discourages the use of sequential programming. Of course there are many other ways to express what. For example we will use the following notation

$$\langle\, f(x) : x \in S \,\rangle$$

to mean apply the function f to each x in the sequence S and return a new sequence. This is purely functional (assuming f has no side effects) and is also parallel.

Secondly, and perhaps most importantly 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.

Thirdly, 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

```
var       :=  ['A'-'Z"a'-'z"]+
op        :=  '-'
binop     :=  ',' | '||' | '+' | '*' | '-' | '/' | ...
pattern   :=  var
           |  '(' pattern ',' pattern ')'
           |  var pattern
typec     :=  var 'of' pattern
bind      :=  'val' pattern '=' exp
           |  'fun' var pattern '=' exp
           |  'datatype' type = typec ('|' typec)*
casebind  :=  pattern '=>' exp
setbind   :=  pattern '∈' exp
setcond   :=  '|' exp (',' exp)*
exp       :=  const
           |  var
           |  op exp
           |  exp binop exp
           |  exp exp
           |  '(' exp ')'
           |  'fn' pattern '=>' exp
           |  'let' bind+ 'in' exp 'end'
           |  'case' exp 'of' casebind ('|' casebind)*
           |  'if' exp 'then' exp 'else' exp
           |  '{' (exp ':')? setbind+ setcond? '}'
```

Figure 1.2: Syntax for pseudocode

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.

We note that 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.

We use mathematical set notation as indicated using the last rule in `exp`. We note that set notation is not consistent across different communities. In the notation we use the expression

$$\{e : x \in S \mid p\}$$

```
1  fun fib(n) =
2    if (n < 2) then 1
3    else fib(n - 1)+fib(n - 2)
4
5  fun fib(n) =
6    if (n < 2) then 1
7    else
8      let
9        val x = fib(n-1)
10       val y = fib(n-2)
11     in
12       x + y
13     end
14
15 val (x,y) = (fib(6),45)
16
17 fun examp(x,y) =
18   let
19     fun add a = (fn b ⇒ a + b)
20     val addx = add(x)
21   in
22     addx(y)
23   end
24
25 datatype Tree = Node of Tree * Tree | Leaf of int
26
27 fun sumTree(T) =
28   case T of
29     Leaf(x) ⇒ x
30   | Node(L,R) ⇒ sumTree(L) + sumTree(R)
```

Figure 1.3: Example uses of the syntax.

means apply the expression e to every x taken from the set S such that the predicate p is true. For example

$$\left\{x^2 : x \in \{2, 7, 11\} \mid x > 5\right\}$$

indicates taking each element of the set $\{2, 7, 11\}$, and for those that satisfy $x > 5$, take their squares. The first part can be left off, e.g.,

$$\{x \in \{2, 7, 11\} \mid x > 5\} \Rightarrow \{7, 11\} \ .$$

The last part can also be left off, e.g.,

$$\left\{x^2 : x \in \{2, 7, 11\}\right\} \Rightarrow \{4, 49, 121\} \ ,$$

or can be repeated with different conditions with a comma inbetween conditions, e.g.,

$$\left\{x \in \{2, 7, 11\} \mid x > 5, x^2 < 100\right\} \Rightarrow \{7\} \ .$$

The middle part can be repeated with different assignments, e.g.,

$$\{x + y : x \in \{2, 7\}, y \in \{2, 4\}\} \Rightarrow \{4, 6, 9, 11\} \ .$$

Chapter 2

Example: Genome Sequencing

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

As with many "real world" applications just defining a formal problem that models our application is interesting in itself. We will therefore start with the vague and not well-defined problem of "sequencing the human genome" and convert it into a formal algorithmic problem. To come up with this formal problem we will assume that the sequencing is done using a particular methodology, called the shotgun method.

2.1 The Shotgun Method

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

Figure 2.1: 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.

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 2.1. *Can you think of a way to parallelize primer walking?*

One possible 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.

Example 2.2. *When cut, the strand* cattaggagtat *might turn into,* ag, gag, catt, tat, *destroying the original ordering.*

Question 2.3. *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 about how the fragments are connected. If we had some additional information about how to connect them, we can imagine solving this problem just as we solve a jigsaw puzzle.

Figure 2.2: A jigsaw puzzle.

Figure 2.3: Water lilies by Claude Monet. We think of the patches of color as water lilies even though they are just that, patches of color.

Question 2.4. *Can you think of a way to relate different pieces?*

One way to get additional information about how to join the fragments is to make multiple copies of the original sequence and generate many fragments that overlap. When a fragment overlaps with two others, it can tell us how to connect those two. 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 the sequences using a "shotgun" (actually using radiation or chemicals).

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

4. Reconstruct the original genome from the fragments.

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

Example 2.5. *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 2.6. *In step 4, is it always possible to reconstruct the sequence?*

Unfortunately 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.

2.2 Defining a Formal Problem

Given that there might be an infinite number of solutions, we are interested in finding a way to defining a problem formally that captures the notion of a "best solution".

Question 2.7. *How can we make this intuitive notion of a "best solution" precise?*

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

Question 2.8. *Can you think of a property that the result needs to have in relation to the fragments?*

Note that since the fragments all come from the original genome, the result should at least contain all of them. 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.

Figure 2.4: 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.

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 2.4). This is how we will define the problem.

Problem 2.9 (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 2.10. *In our example, we had the following fragments.*

catt ag gagtat
cat tagg ag tat
ca tta gga gtat

Our snippets are now:

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

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

Since no snippet can be contained in another, snippets cannot start at the same position, and if one starts after another, it must finish after the other. This leads to our second observation.

Observation 1: Ordering. In any superstriong, the starts of the snippets must start in some strict (total) ordering and must finish in the same ordering.

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

2.3 Algorithm-Design Technique 1: Brute Force

As discussed in Section 1.3 the brute-force technique consist of trying all candidate solutions and selecting the best (or any) valid solution. For the SS problem a solution is valid if it is a superstring of all the input strings, and we want to pick the shortest of these strings.

We consider two brute force solutions. The first is simply to consider all strings $r \in \Sigma^+$, and for each r to check if every string $s \in S$ is a substring. It turns out that such a check can be done efficiently, although we won't describe how here. We then can pick the shortest r that is indeed a superstring of all $s \in S$. The problem, however, is that there are an infinite number of strings in Σ^+ so we cannot check them all, but perhaps we can be a little smart about picking the candidate solutions r.

Question 2.12. *Why do we only need to consider strings r with length up to $m = \sum_{s \in S} |s|$? How many different strings are there of length m.*

We only need to consider strings up to length $m = \sum_{s \in S} |s|$ since we can simply put the strings back to back, and the length would be m. There are $|\Sigma|^m$ strings of length m. This number might be extremely large, but not infinite. For the sequencing the genome $\Sigma = 4$ and m is in the order of billions, giving something like $4^{1,000,000,000}$. There are only about $10^{80} \approx 4^{130}$ atoms in the universe so there is no feasible way we could apply the brute force method directly. In fact we couldn't even apply it to two strings each of length 100.

We now consider a second brute force solution. In addition to requiring less computational work, the approach will also help us understand other possible solutions, which we look at next. In the last section we mentioned that snippets have to start in some ordering in any superstring. If this is the case, we can try all possible orderings of the snippets. One of the orderings has to be the right ordering for the shortest superstring. The question remains of how once we pick an ordering we then find the shortest superstring for that ordering. For this purpose we will use the following theorem.

Theorem 2.13 (Removing Overlap)**.** *Given any start ordering of the snippets $s_1, s_2, \ldots, s_n$, removing the maximum overlap between each adjacent pair of snippets (s_i, s_{i+1}) gives the shortest superstring of the snippets for that start ordering.*

Example 2.14. *For our running example, consider the following ordering*

catt tta tagg gga gagtat

When the maximum overlaps are removed (the excised parts are underlined) we get cattaggagtat*, which is indeed the shortest superstring for the given start ordering (indeed it is also the overall shortest).*

Proof. The theorem 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 the shortest superstring of $s_1, \ldots, s_i$ starting in that order. We refer to this superstring as r_i. We now prove that if it is true for r_i then it is true for r_{i+1}. Note that when we add the snippet s_{i+1} after r_i it cannot fully overlap with the previous snippet (s_i) by the definition of snippets. Therefore when we add it on

using the maximum overlap, the string r_{i+1} will be r_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 r_i, plus it will also be a superstring of s_{i+1}, since we are including it at the end. It will also be the shortest since r_i is the shortest for $s_1, \ldots s_i$ (starting in that order) and by picking the maximum overlap we added the least number of additional characters to get a superstring for $s_1, \ldots s_{i+1}$ □

This theorem tells us that we can try all permutations of the snippets, calculate the length of each one by removing the overlaps and pick the best.

Exercise 2.15. *Try a couple other permutations and determine the length after removing overlaps.*

We now look at the work required for this second brute-force algorithm. There are $n!$ permutations on a collection of n elements each of which has to be tried. For $n = 10$ strings this is probably feasible, which is better than our previous technique that did not even work for $n = 2$. However for $n = 100$, we'll need to consider $100! \approx 10^{158}$ combinations, which is still more than the number of atoms in the universe. As such, the algorithm is still not going to be feasible for large n.

Question 2.16. *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 2.17. *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, even on programs with thousands of variables.

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.

2.4 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

Figure 2.5: 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.

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 2.18. *Are you all familiar with the TSP problem?*

The TSP problem is a canonical NP-hard problem dating back to the 1930s and has been extensively studied, e.g. see Figure 2.5. 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.

Problem 2.19 (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 2.20. *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.

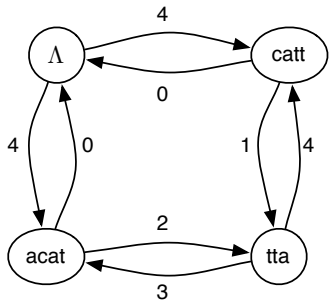
Example 2.21. *For "tagg" and "gga", we have* $\texttt{overlap}$ *("tagg","gga")* $= 2$.

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"|$-\texttt{overlap}$("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. 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 2.22. *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 2.23. *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.

2.5 Algorithm-Design Technique 3: Greedy

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

Definition 2.24 (The Greedy **Technique**). *Give a set of elements, on each step remove at least one element by making a locally optimal decision based on some criteria.*

Pseudo Code 2.27.

```
1  function 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 2.1: Greedy Approximate SS Algorithm

For example, a greedy technique for the TSP could be to always visit the closest unvisited city. Each step makes a locally optimal decision, and each step removes one element, the next city visited.

Question 2.25. *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.

Question 2.26. *Considering that we want to minimize the length of the result, what should our "greedy choice" be?*

To choose an appropriate greedy approach for the SS problem we note that to minimize the length of the superstring we would like to maximimize the overlap among the snippets. Thus one greedy approach is to pick a pair of snippets with the largest overlap and join them by placing one immediately after the other and removing the overlap. This can then be repeated until there is only one string left.

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

The pseudocode for our algorithm is given in Algorithm 2.27. In this book we will be using mathematical set notation in the pseudocode. 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 triples 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 2.28. *Is the algorithm guaranteed to terminate?*

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 2.29. *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 2.30. *Prove that algorithm* `greedyApproxSS` *indeed returns a string that is a superstring of all original strings.*

Exercise 2.31. *Give an example input S for which* `greedyApproxSS` *does not return the shortest superstring.*

Exercise 2.32. *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 $\mathbf{P} = \mathbf{NP}$, 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 2.33. *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.*