Chapter 4

Algorithm Analysis

In this chapter we look at how to analyze the cost of algorithms in a way that is useful for comparing algorithms to determine which is likely to be better, at least for large input sizes, **but** is abstract enough that we don't have to look at minute details of compilers and machine architectures. There are a several parts of such analysis. Firstly we need to abstract the cost from details of the compiler or machine. Secondly we have to decide on a concrete model that allows us to formally define the cost of an algorithm. Since we are interested in parallel algorithms, the model needs to consider parallelism. Thirdly we need to understand how to analyze costs in this model. These are the topics of this chapter.

4.1 Abstracting Costs

When we analyze the cost of an algorithm formally, we need to be reasonably precise about the model we are performing the analysis in.

Question 4.1. *How precise should this model be? For example, would it help to know the exact running time for each instruction?*

The model can be arbitrarily precise but it is often helpful to abstract away from some details such as the exact running time of each (kind of) instruction. For example, the model can posit that each instruction takes a single step (unit of time) whether it is an addition, a division, or a memory access operation. Some more advanced models, which we will not consider in this class, separate between different classes of instructions, for example, a model may require analyzing separately calculation (e.g., addition or a multiplication) and communication (e.g., memory read).

Question 4.2. *Why would abstracting away from exact running times help?*

There are two ways in which such abstraction helps: simplicity and portability. Abstraction simplifies analysis and makes the analysis applicable to many different actual hardware and machines, rather that just one or a few, that are consistent with the model. A model too precise can hurt portability because machines are all different.

Example 4.3. *Consider the following function:*

```
1 fun add (x:int,y:int) = x+y
```

This ML function, when invoked with two integers performs 1 *unit of work in a model where all operations take unit time. Note that the cost is independent of the input.*

Question 4.4. *What happens when cost depends on the input but we don't know the input?*

Cost can depend on the input. Usually, however, all we need is the input size, which we can use for our analysis. Input size is usually written n, m, etc..

Example 4.5. *Consider the following function:*

```
1 datatype tree = Leaf of int | Node of tree * tree
2 fun addTree (t: tree) =
3   case t of
4     Leaf x ⇒ x
5   | Node(l,r) ⇒ addTree (l) + addTree (r)
```

This ML function, when invoked with a tree of n *internal nodes and* m *leaves, performs* $4n + m$ *work in a model where case statement, function call, and the add operations all take unit time.*
If we also count returning from a function call as unit time, the work is $5n + 2m$*.*
Since in a (full) binary tree $m = n + 1$*, the work can be written just in terms of* n *as* $7n + 1$*.*

Question 4.6. *How do we know whether we need to count the "return" or not?*

At the level of abstraction that we write our algorithms in this course, certain "hidden" instructions such as the return instruction can be executed. The existence of such instructions can be difficult to predict, without using a lower level of abstraction.

Fortunately, for our purposes, asymptotic analysis will be sufficient. Instead of performing an exact analysis, we will instead analyze asymptotic costs, e.g., using big-O notation.

Example 4.7. *Consider the following function*

```
1 fun add (x:int,y:int) = x+y
```

This ML function performs $\Theta(1)$ *work.*

Remark 4.8. *We assume you have seen big-O, big-Theta (*Θ*), and big-Omega (*Ω*) in a previous class. If not there is a quick review in the next section.*

Example 4.9. *Consider again the function in Example 4.5. when invoked with a tree of* n *internal nodes and* m *leaves, performs* $\Theta(n+m)$ *work in a model where case statement, function call, and the add operations all take unit time. Since in a binary tree* $m = n+1$*, the work is* $\Theta(n)$*.*

Asymptotic analysis thus adds another level of abstraction, making our analysis further removed from the actual machine.

The advantages continue to hold. Asymptotic analysis further enables comparing algorithms in terms of how they scale to large inputs.

Example 4.10. *Some sorting algorithms have* $\Theta(n \log n)$ *work and others* $\Theta(n^2)$*. Clearly the* $\Theta(n \log n)$ *algorithm scales better. The* $\Theta(n^2)$ *algorithm, however, can be more efficient on small inputs.*

In this class we are concerned with how algorithms scale, and therefore asymptotic analysis is indeed what we want. Because we are using asymptotic analysis the exact constants in the model do not matter, but what matters is that the asymptotic costs are well defined.

4.2 Asymptotic Complexity

What is $O(f(n))$? Precisely, $O(f(n))$ is the set of all functions that are asymptotically dominated by the function $f(n)$. Intuitively this means that these are functions that grow at the same or slower rate than $f(n)$ as n goes to infinity. We write $g(n) \in O(f(n))$ to refer to a function $g(n)$ that is in the set $O(f(n))$. We often use the abusive notation $g(n) = O(f(n))$ as well.

In an expression such as $4W(n/2) + O(n)$, the $O(n)$ refers to some function $g(n) \in O(n)$.

Question 4.11. *Do you remember the definition of* $O(\cdot)$*?*

For a function $g(n)$, we say that $g(n) \in O(f(n))$ if there exist positive constants n_0 and c such that for all $n \geq n_0$, we have $g(n) \leq c \cdot f(n)$.

If $g(n)$ is a finite function ($g(n)$ in finite for all n), then it follows that *there exist constants k_1 and k_2 such that for all $n \geq 1$,*

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

where, for example, we can take $k_1 = c$ and $k_2 = \sum_{i=1}^{n_0} |g(i)|$.

Remark 4.12. *Make sure to become very comfortable with asymptotic analysis. Also its different versions such as the $\Theta(\cdot)$ and $\Omega(\cdot)$.*

Exercise 4.13. *Can you illustrate graphically when $g(n) \in O(f(n))$? Show different examples, to hone your understanding.*

4.3 Defining a Cost Model

Although we have decided to use asymptotic analysis so we can ignore details about the implementation, we still need to be reasonably precise about the definition of the cost model. This is because differences in the model can make asymptotic differences in performance. There are two standard ways to define cost models: machine based and language based. In a machine-based model, costs is defined by counting the number of instructions taken by a particular machine. In a language-based model, cost is defined by composing costs across various programming language constructs. Both types can be applied to analyzing either sequential or parallel algorithms.

Question 4.14. *What are the advantages of using a machine based and a language-based model?*

When using a machine model, we have to reason about how a program (the algorithm) compiles and runs on that machine. For sequential programs this can be straightforward when using low-level languages such as C since there is an almost one-to-one mapping of statements in the language to machine instructions. For higher-level languages it becomes somewhat trickier—there might be uncertainties, for example, about the cost of automatic memory management, or the cost of dispatching in an object-oriented language. For parallel programs it becomes even trickier since it can require reasoning about how tasks are scheduled on the processors.

When analyzing algorithms in a language-based model we don't need to care about how the language compiles or runs on the machine. Costs are defined directly in the language. On the other hand we probably do want to have a sense that the costs in the model have some relevance when we run the code on a real machine. We get back to this issue in Section 4.4. We note that

in the same way that a machine model does not need to be a specific machine such as an Intel Nehalem or AMD Phenom, the language doesn't need to be a specific language such as C++ or Java. Instead it can be an abstract language such as the lambda calculus. The lamba calculus is effectively what we use in this book, with some added syntactic sugar, but you don't need to know much about the lamba calculus to follow the model.

Traditionally with sequential algorithms machine models have been preferred since the mapping from language constructs to machine cost (time or number of instructions) have been mostly obvious. In this course, however, we will use a language-based cost model since the mapping is not so obvious, and because it allows us to use abstract costs, work and span, which have no direct meaning on a physical machine. We first review the traditional sequential machine model.

4.3.1 The RAM Model for Sequential Computation

Traditionally, algorithms have been analyzed in the Random Access Machine (RAM)[1] model. This model assumes a single processor accessing unbounded memory indexed by the non-negative integers. The processor interprets sequences of machine instructions (code) that are stored in the memory. Instructions include basic arithmetic and logical operations (e.g. $+$, $-$, $\star$, and, or, not), reads from and writes to arbitrary memory locations, and conditional and unconditional jumps to other locations in the code. The cost of a computation is measured in terms of the number of instructions executed by the machine, and is referred to as *time*.

This model is quite adequate for analyzing the asymptotic runtime of sequential algorithms; most work on sequential algorithms to date has used this model. It is therefore important to understand the model, or at least know what it is. One reason for the RAM's success is that it is relatively easy to reason about the cost of algorithms because algorithmic pseudo code and sequential languages such as C and C++ can easily be mapped to the model. The model, however, should only be used for deriving asymptotic bounds (i.e., using big-O, big-Theta and big-Omega) and not for trying to predict exact runtimes. One reason for this is that on a real machine not all instructions take the same time, and furthermore not all machines have the same instructions.

We note that one problem with the RAM model is that it assumes that accessing all memory locations has the same cost. On real machines this is not the case. In fact, there can be a factor of over 100 between the time for accessing a word of memory from the first level cache and accessing it from main memory. Various extensions to the RAM model have been developed to account for this cost. For example one variant is to assume that the cost for accessing the i^{th} memory location is $f(i)$ for some function f, e.g. $f(i) = \log(i)$. Fortunately, however, most of the algorithms that turn out to be good in these more detailed models are also good in the RAM. Therefore analyzing algorithms in the simpler RAM model is often a reasonable approximation to analyzing in the more refined models. Hence the RAM has served quite well despite not fully accounting for the variance in memory costs. The model we use in this course also does not account for the variance in memory costs, but as with the RAM the costs can be refined.

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

4.3.2 The Parallel RAM Model

For our purposes, the more serious problem with the RAM model is that it is sequential. One way to extend the RAM to allow parallelism is simply to use multiple processors which share the same memory. This is referred to as the *Parallel Random Access Machine* (PRAM). In the model all of p processors run the same instruction on each step, although typically on different data. For example if we had an array of length p, each processor could add one to its own element allowing us to increment all elements of the array in constant time.

We will not be using the PRAM model since it is awkward to work with, both because it is overly synchronous and because it requires the user to map computation to processors. For simple parallel loops over n elements we could imagine dividing up the elements evenly among the processors—about n/p each, although there is some annoying rounding required since n is typically not a multiple of p. If the cost of each iteration of the loop is different then we would further have to add some load balancing. In particular simply giving n/p to each processor might be the wrong choice—one processor could get stuck with all the expensive iterations. For computations with nested parallelism, such as divide-and-conquer algorithms the mapping is much more complicated, especially given the highly synchronous nature of the model.

Even though we don't use the PRAM model, most of the ideas presented in this course also work with the PRAM, and many of them were originally developed in the context of the PRAM.

4.3.3 The Work-Span Model

Instead of using a machine model, this book will use a model that is more directly tied to programming constructs. We believe that the model makes it much easier to separate the high-level concepts of parallelism from low-level machine-specific details. It is therefore more amenable to get you to "think parallel", one goal of this course. As it turns out, there is a way to map the costs we derive onto costs for specific machines, which is discussed in Section 4.4.

Work and Span. As mentioned in the introduction, in this book we will measure complexity in terms of two costs: work and span. Roughly speaking the *work* corresponds to the total number of operations we perform, and *span* to the longest chain of dependencies. We define work and span based on simple compositional rules over expressions in the language. For an expression e let $W(e)$ indicate the work needed to evaluate that expression and $S(e)$ indicate the span.

Example 4.15.

$$
\begin{array}{lcl}
W(7+3) & = & \textit{Work for adding } 7 \textit{ and } 3 \\
S(\texttt{fib}(11)) & = & \textit{Span for calculating the } 11^{th} \textit{ Fibonacci number} \\
W(\texttt{mySort}(S)) & = & \textit{Work for } \texttt{mySort} \textit{ applied to the sequence } S
\end{array}
$$

Note that in the third example the sequence S is not defined within the expression. Therefore we cannot say in general what the work is as a fixed value. However, we might be able to use asymptotic analysis to write a cost in terms of the length of s, and in particular if `mySort` is a good sorting algorithm we would have:

$$W(\texttt{mySort}(S)) = O(|S| \log |S|)\,.$$

Often instead of writing $|S|$ to indicate the size of the input, we use n or m as shorthand. Also if the cost is for a particular algorithm we use a subscript to indicate the algorithm. This leads to the following notation

$$W_{\texttt{mySort}}(n) = O(n \log n)\,.$$

where n is the size of the input of `mysort`. When obvious from the context (e.g. when in a section on analyzing `mySort`) we sometimes drop the subscript, giving $W(n) = O(n \log n)$.

Now we care about composing costs across expressions. For example we would like to determine $W(e_1 + e_2)$ given $W(e_1)$ and $W(e_2)$? Here we assume e_1 and e_2 are arbitrary expressions. We can do such composition of costs with some simple rules that correspond to each of the different types of expression we have in our language. As discussed in the Introduction, these rules roughtly say that when composing sequentially we add the work and add the span, but that when composing in parallel, we add the work and take the maximum of the span.

Question 4.16. *When are expressions composed in parallel and when sequentially?*

Since in this book we are assuming purely functional programs, it is always safe to run things in parallel if there is no explicit sequencing.

Example 4.17. *In the expression $e_1 + e_2$ where e_1 and e_2 are themselves other expressions (e.g. function calls) we could run the two expressions in parallel giving the rule*

$$S(e_1 + e_2) = 1 + \max(S(e_1), S(e_2)).$$

This rule says the two subexpressions e_1 and e_2 run in parallel, and therefore the span in combination is the maximum of their individual spans. The addition operation, however, has to wait for both subexpressions to be done. It therefore has to be performed sequentially after the two parallel subexpressions and hence the plus 1 in the expression $1 + \max(S(e_1), S(e_2))$.

In this book, however, to make it more clear whether expressions are evaluated sequentially or in parallel we will assume that expressions are composed in parallel only when using explicit parallel forms. In particular we will use the notation $(e_1 \,||\, e_2)$ to mean that the two expressions run in parallel. The result is a pair of values containing the two results. In addition to the $||$ construct we assume the set-like notation we use in our pseudocode $\{f(x) : x \in A\}$ also runs in parallel, i.e., all calls to $f(x)$ run in parallel.

$$
\begin{aligned}
W(c) &= 1 \\
W(e_1 \text{ op } e_2) = W((e_1, e_2)) = W(e_1 \mid\mid e_2) &= 1 + W(e_1) + W(e_2) \\
W(\texttt{let val } x = e_1 \texttt{ in } e_2 \texttt{ end}) &= 1 + W(e_1) + W(e_2[\text{Eval}(e_1)/x])) \\
W(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) &= 1 + W(e_1) + \begin{cases} W(e_2) & \text{Eval}(e_1) = True \\ W(e_3) & \text{otherwise} \end{cases} \\
W(\{f(x) : x \in A\}) &= 1 + \sum_{x \in A} W(f(x))
\end{aligned}
$$

$$
\begin{aligned}
S(c) &= 1 \\
S(e_1 \text{ op } e_2) = S((e_1, e_2)) &= 1 + S(e_1) + S(e_2) \\
S((e_1 \mid\mid e_2)) &= 1 + \max(S(e_1), S(e_2)) \\
S(\texttt{let val } x = e_1 \texttt{ in } e_2 \texttt{ end}) &= 1 + S(e_1) + S(e_2[\text{Eval}(e_1)/x])) \\
S(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) &= 1 + S(e_1) + \begin{cases} S(e_2) & \text{Eval}(e_1) = True \\ S(e_3) & \text{otherwise} \end{cases} \\
S(\{f(x) : x \in A\}) &= 1 + \max_{x \in A} S(f(x))
\end{aligned}
$$

Figure 4.1: Composing work and span across expressions. In the first rule c is any constant value (e.g. 3). In the second rule **op** is any primitive operator such as $+$ or $\times$. The notation Eval(e) evaluates the expression e and returns the result, and the notation $e[v/x]$ indicates that all free (unbound) occurrences of the variable x in the expression e are replaced with the value v. These rules are representative of all rules of the language.

Given these conventions, the rules for composing work and span are outlined in Figure 6.2. For the `let` expression we need to first evaluate e_1 and assign it to x before we can evaluate e_2. Hence the fact that the span is composed sequentially, i.e., by adding the spans. Note that the rules are the same for work and span except for the two parallel constructs: $||$ and $\{f(x) : x \in A\}$.

Example 4.18. *The expression* $(\texttt{fib}(6) \;||\; \texttt{fib}(7))$ *runs the two calls to* `fib` *in parallel and returns the pair* $(8, 13)$. *It does work*

$$1 + W(\texttt{fib}(6)) + W(\texttt{fib}(7))$$

and span

$$1 + \max(S(\texttt{fib}(6)), S(\texttt{fib}(7))) \; .$$

If we know that the span of `fib` *grows with the input size, then the span can be simplified to* $1 + S(\texttt{fib}(7))$.

Example 4.19. *Let expressions compose sequentially.*

$$\begin{aligned} W(\texttt{let a = f(x) in g(a) end}) &= 1 + W(f(x)) + W(g(a)) \\ S(\texttt{let a = f(x) in g(a) end}) &= 1 + S(f(x)) + S(g(a)) \end{aligned}$$

Remark 4.20. *As there is no* $||$ *construct in the ML, in your assignments you will need to specify in comments when two calls run in parallel. We will also supply an ML function* `par (f1,f2)` *with type* `(unit -> α) × (unit -> β) -> α × β`. *This function executes the two functions that are passed in as arguments in parallel and returns their results as a pair. For example:*
`par (fn => fib(6), fn => fib(7))`
returns the pair $(8, 13)$. *We need to wrap the expressions in functions in ML so that we can make the actual implementation run them in parallel. If they were not wrapped both arguments would be evaluated sequentially before they are passed to the function* `par`.
Also in the ML code you do not have the set notation $\{f(x) : x \in A\}$, *but as mentioned before, it is basically equivalent to a* `map`. *Therefore, for ML code you can use the rules:*

$$W(\texttt{map}\;\; f\;\; \langle s_0, \ldots, s_{n-1} \rangle) = 1 + \sum_{i=0}^{n-1} W(f(s_i))$$

$$S(\texttt{map}\;\; f\;\; \langle s_0, \ldots, s_{n-1} \rangle) = 1 + \max_{i=0}^{n-1} S(f(s_i))$$

Parallelism: An additional notion of cost that is important in comparing algorithms is the *parallelism* of an algorithm. Parallelism, sometimes called *average parallelism*, is simply defined as the work over the span:

$$\mathbb{P} = \frac{W}{S}$$

Parallelism informs us approximately how many processors we can use efficiently.

Example 4.21. *For a mergesort with work $\theta(n \log n)$ and span $\theta(\log^2 n)$ the parallelism would be $\theta(n/\log n)$.*

Suppose $n = 10,000$ and if $W(n) = \theta(n^3) \approx 10^{12}$ and $S(n) = \theta(n \log n) \approx 10^5$ then $\mathbb{P}(n) \approx 10^7$, which is a lot of parallelism. But, if $W(n) = \theta(n^2) \approx 10^8$ then $\mathbb{P}(n) \approx 10^3$, which is much less parallelism. The decrease in parallelism is not because of the span was large, but because the work was reduced.

Question 4.22. *What are ways in which we can increase parallelism?*

We can increase parallelism by decreasing span and/or increasing work. Increasing work, however, is not desirable because it leads to an inefficient algorithm.

Definition 4.23 (Work efficiency)**.** *We say that a parallel algorithm is* work efficient *if it perform asymptotically the same work as the best known sequential algorithm for that problem.*

Example 4.24. *A (comparison-based) parallel sorting algorithm with $\Theta(n \log n)$ work is work efficient; one with $\Theta(n^2)$ is not, because we can sort sequentially with $\Theta(n \log n)$ work.*

Designing parallel algorithms. In parallel-algorithm design, we aim to keep parallelism as high as possible but without increasing work. In general the goals in designing efficient algorithms are

1. first priority: to keep work as low as possible, and
2. second priority: keep parallelism as high as possible (and hence the span as low as possible).

In this course we will mostly cover work-efficient algorithms where the work is the same or close to the same as the best sequential time. Indeed this will be our goal throughout the course. Now among the algorithm that have the same work as the best sequential time we will try to achieve the greatest parallelism.

4.4 Scheduling

An important advantage of the work-depth model is that is allows us to design parallel algorithms without having to worry about the details of how they are executed on an actual parallel machine. In other words, we never have to worry about mapping of the parallel computation to processors, i.e., *scheduling*.

Question 4.25. *Is scheduling a challenging task? Why?*

Scheduling can be challenging because a parallel algorithm generate tasks on the fly as it runs, and it can generate a massive number of them, typically much more than the number of processors available when running.

Example 4.26. *A parallel algorithm with* $\Theta(n/\log n)$ *parallelism can easily generate millions parallel subcomptutations or task at the same time, even when running on a multicore computer with for example* 10 *cores.*

Scheduler. Mapping parallel tasks to available processor so that each processor remains busy as much as possible is the task of a scheduler. The scheduler works by taking all parallel tasks, which are generated dynamically as the algorithm evaluates, and assigning them to processors. If only one processor is available, for example, then all tasks will run on that one processor. If two processor are available, the task will be divided between the two.

Question 4.27. *Can you think of a scheduling algorithm?*

Greedy scheduling. We say that a scheduler is *greedy* if whenever there is a processor available and a task ready to execute, then it assigns the task to the processor and start running it immediately. Greedy schedulers have a very nice property that is summarized by the following:

Definition 4.28. *The* greedy scheduling principle *says that if a computation is run on p processors using a greedy scheduler, then the total time (clock cycles) for running the computation is bounded by*

$$T_p < \frac{W}{p} + S \tag{4.1}$$

where W is the work of the computation, and S is the span of the computation (both measured in units of clock cycles).

This is actually a very powerful statement. The time to execute the computation cannot be any better than $\frac{W}{p}$ clock cycles since we have a total of W clock cycles of work to do and the best we can possibly do is divide it evenly among the processors. Also note that the time to execute the computation cannot be any better than S clock cycles since S represents the longest chain of sequential dependencies. Therefore the very best we could do is:

$$T_p \geq \max\left(\frac{W}{p}, S\right)$$

We therefore see that a greedy scheduler does reasonably close to the best possible. In particular $\frac{W}{p} + S$ is never more than twice $\max(\frac{W}{p}, S)$ and when $\frac{W}{p} \gg S$ the difference between the two is very small. Indeed we can rewrite equation 4.1 above in terms of the parallelism $\mathbb{P} = W/S$ as follows:

$$\begin{aligned} T_p &< \frac{W}{p} + S \\ &= \frac{W}{p} + \frac{W}{\mathbb{P}} \\ &= \frac{W}{p}\left(1 + \frac{p}{\mathbb{P}}\right) \end{aligned}$$

Therefore as long as $\mathbb{P} \gg p$ (the parallelism is much greater than the number of processors) then we get near perfect speedup. (Speedup is W/T_p and perfect speedup would be p).

Remark 4.29. *No real schedulers are fully greedy. This is because there is overhead in scheduling the job. Therefore there will surely be some delay from when a job becomes ready until when it starts up. In practice, therefore, the efficiency of a scheduler is quite important to achieving good efficiency. Also the bounds we give do not account for memory affects. By moving a job we might have to move data along with it. Because of these affects the greedy scheduling principle should only be viewed as a rough estimate in much the same way that the RAM model or any other computational model should be just viewed as an estimate of real time.*

4.5 Analysis of Shortest-Superstring Algorithms

As examples of how to use our cost model we will analyze a couple of the algorithms we described for the shortest superstring problem: the brute force algorithm and the greedy algorithm.

4.5.1 The Brute Force Shortest Superstring Algorithm

Recall that the idea of the brute force algorithm for the SS problem is to try all permutations of the input strings and for each permutation to determine the maximal overlap between adjacent strings and remove them. We then pick whichever remaining string is shortest, if there is a tie we pick any of the shortest. We can calculate the overlap between all pairs of strings in a preprocessing phase. Let n be the size of the input S and m be the total number of characters across all strings in S, i.e.,

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

Note that $n \leq m$. The preprocessing step can be done in $O(m^2)$ work and $O(\log n)$ span (see analysis below). This is a low order term compared to the other work, as we will see, so we can ignore it.

Now to calculate the length of a given permutation of the strings with overlaps removed we can look at adjacent pairs and look up their overlap in the precomputed table. Since there are n strings and each lookup takes constant work, this requires $O(n)$ work. Since all lookups can be done in parallel, it will require only $O(1)$ span. Finally we have to sum up the overlaps and subtract it from m. The summing can be done with a **reduce** in $O(n)$ work and $O(\log n)$ span. Therefore the total cost is $O(n)$ work and $O(\log n)$ span.

As we discussed in the last lecture the total number of permutations is $n!$, each of which we have to check for the length. Therefore the total work is $O(nn!) = O((n+1)!)$. What about the span? Well we can run all the tests in parallel, but we first have to generate the permutations. One simple way is to start by picking in parallel each string as the first string, and then for each of these picking in parallel another string as the second, and so forth. The pseudo code looks something like this:

```
func permutations S =
   if |S| = 1 then {S}
   else
      {append([s], p) :  s in S, p in permutations(S/s)}
```

What is the span of this code?

4.5.2 The Greedy Shortest Superstring Algorithm

We'll consider a straightforward implementation, although the analysis is a little tricky since the strings can vary in length. First we note that calculating $\texttt{overlap}(s_1, s_2)$ and $\texttt{join}(s_1, s_2)$ can be done in $O(|s_1||s_2|)$ work and $O(\log(|s_1| + |s_2|))$ span. This is simply by trying all overlap positions between the two strings, seeing which ones match, and picking the largest. The logarithmic span is needed for picking the largest matching overlap using a reduce.

Let W_{ov} and S_{ov} be the work and span for calculating all pairs of overlaps (the line $\{(\texttt{overlap}(s_i, s_j), s_i, s_j) : s_i \in S, s_j \in S, s_i \neq s_j\}$), and for our set of input snipets S recall that $m = \sum_{x \in S} |x|$.

We have

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

and since all pairs can be done in parallel,

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

The $\arg\max$ for finding the maximum overlap can be computed in $O(m^2)$ work and $O(\log m)$ span using a simple reduce. The other steps have less work and span. Therefore, not including the recursive call each call to `greedyApproxSS` costs $O(m^2)$ work and $O(\log m)$ span.

Figure 4.2: Abstraction is a powerful technique in computer science. One reason why is that it enables us to use our intelligence more effectively allowing us not to worry about all the details or the reality. Paul Cezanne noticed that all reality, as we call it, is constructed by our intellect. Thus he thought, I can paint in different ways, in ways that don't necessarily mimic vision, and the viewer can still create a reality. This allowed him to construct more interesting realities. He used abstract, geometric forms to architect reality. Can you see them in his self-portrait? Do you think that his self-portrait creates a reality that is much more three dimensional, with more volume, more tactile presence than a 2D painting that would mimic vision? Cubists such as Picasso and Braque took his ideas on abstraction to the next level.

Finally, we observe that each call to `greedyApproxSS` creates S' with one fewer element than S, so there are at most n calls to `greedyApproxSS`. These calls are inherently sequential because one call must complete before the next call can take place. Hence, the total cost for the algorithm is $O(nm^2)$ work and $O(n \log m)$ span, which is highly parallel.

Exercise 4.30. *Come up with a more efficient way of implementing the greedy method.*

4.6 Solving Recurrences

The best way to analyze most divide-and-conquer algorithms is to write down a so-called "recurrence" for the cost of the algorithm, and then to solve that recurrence. The recurrence follows the recursive structure of the algorithm, but if a function of size instead of the actual values. For example you might have previously seen that the mergeSort algorithm leads to a recurrence of the form $W(n) = 2W(n/2) + O(n)$. This corresponds to the fact that for an input of size n, mergeSort makes two recursive calls of size $n/2$, and also does $O(n)$ other work. In particular the merge itself requires $O(n)$ work. Similarly for span we can write a recurrence of the form $S(n) = \max(S(n/2), S(n/2)) + O(\log n) = S(n/2) + O(\log n)$. This is because the merge has spand $O(\log n$, and since the two recursive calls are made in parallel we take the maximum instead of summing them. Here we discuss methods for solving such recurrences.

4.6.1 The Tree Method

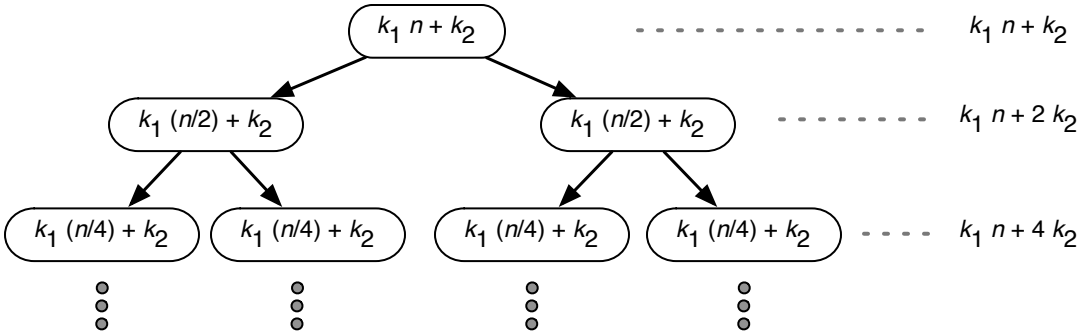
Using the recursion $W(n) = 2W(n/2) + O(n)$, we will review the tree method, which you have seen in 15-122 and 15-251. Our goal is to derive a closed form solution to this recursion.

By the definition of asymptotic complexity, we can establish that

$$W(n) \quad \leq \quad 2W(n/2) + k_1 \cdot n + k_2,$$

where k_1 and k_2 are constants.

The idea of the tree method is to consider the recursion tree of the recurrence to derive an expression for what's happening at each level of the tree. In this particular example, we can see that each node in the tree has 2 children, whose input is half the size of that of the parent node. Moreover, if the input has size n, the recurrence shows that the cost, excluding that of the recursive calls, is at most $k_1 \cdot n + k_2$. Therefore, our recursion tree annotated with cost looks like this:



To apply the tree method, there are some key questions we should ask ourselves to aid drawing out the recursion tree and to understand the cost associated with the nodes:

- How many levels are there in the tree?
- What is the problem size at level i?
- What is the cost of each node in level i?
- How many nodes are there at level i?
- What is the total cost across level i?

Our answers to these questions lead to the following analysis: We know that level i (the root is level $i = 0$) contains 2^i nodes, each costing at most $k_1(n/2^i) + k_2$. Thus, the total cost in level i is at most

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

Since we keep halving the input size, the number of levels is bounded by $1 + \log n$. Hence, we have

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

where in the second to last step, we apply the fact that for $a > 1$,

$$1 + a + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} \leq a^{n+1}.$$

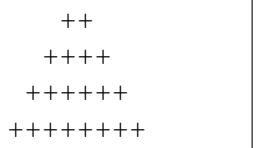
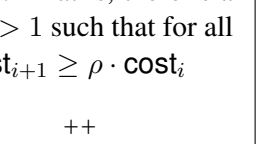
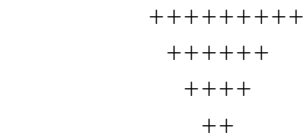
4.6.2 The Brick Method, a Variant of the Tree Method

The tree method involves determining the depth of the tree, computing the cost at each level, and summing the cost across the levels. Usually we can easily figure out the depth of the tree and the cost of at each level relatively easily—but then, the hard part is taming the sum to get to the final answer.

It turns out that there is a special case in which the analysis becomes simpler. This is whet the costs at each level grow geometrically, shrink geometrically, or stay approximately equal.

By recognizing whether the recurrence conforms with one of these cases, we can almost immediately determine the asymptotic complexity of that recurrence.

The vital piece of information is *the ratio of the cost between adjacent levels*. Let cost_i denote the total cost at level i when we draw the recursion tree. This puts recurrences into three broad categories:

Leaves Dominated	***Balanced***	***Root Dominated***
Each level is larger than the level before it by at least a constant factor. That is, there is a constant $\rho > 1$ such that for all level i, $\mathsf{cost}_{i+1} \geq \rho \cdot \mathsf{cost}_i$	All levels have approximately the same cost.	Each level is smaller than the level before it by at least a constant factor. That is, there is a constant $\rho < 1$ such that for all level i, $\mathsf{cost}_{i+1} \leq \rho \cdot \mathsf{cost}_i$
		
Implication: $O(\mathsf{cost}_d)$, where d is the depth The house is stable, with a strong foundation.	*Implication:* $O(d \cdot \max_i \mathsf{cost}_i)$ The house is sort of stable, but don't build too high.	*Implication:* $O(\mathsf{cost}_0)$ The house will tip over.

You might have seen the "master method" for solving recurrences in previous classes. We do not like to use it since it only works for special cases and does not give an intuition of what is going on. However, we will note that the three cases of the master method correspond to special cases of leaves dominated, balanced, and root dominated.

4.6.3 The Substitution Method

Using the definition of big-O, we know that

$$W(n) \leq 2W(n/2) + k_1 \cdot n + k_2,$$

where k_1 and k_2 are constants.

Besides using the recursion tree method, can also arrive at the same answer by mathematical induction. If you want to go via this route (and you don't know the answer a priori), you'll need to guess the answer first and check it. This is often called the "substitution method." Since this technique relies on guessing an answer, you can sometimes fool yourself by giving a false proof. The following are some tips:

1. Spell out the constants. Do not use big-O—we need to be precise about constants, so big-O makes it super easy to fool ourselves.
2. Be careful that the induction goes in the right direction.
3. Add additional lower-order terms, if necessary, to make the induction go through.

Let's now redo the recurrences above using this method. Specifically, we'll prove the following theorem using (strong) induction on n.

Theorem 4.31. *Let a constant $k > 0$ be given. If $W(n) \leq 2W(n/2) + k \cdot n$ for $n > 1$ and $W(1) \leq k$ for $n \leq 1$, then we can find constants κ_1 and κ_2 such that*

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

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

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

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

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

where the final step follows because $k \cdot n + \kappa_2 - \kappa_1 \cdot n \leq 0$ as long as $n > 1$. □