Chapter 4

Algorithm Analysis

The term "algorithm analysis" refers to mathematical analysis of algorithms for the purposes of determining their consumption of resources such as the amount of total work they perform, the energy they consume, the time to execute, and the memory or storage space that they require. When analyzing algorithms, it is important to be precise enough so that we can compare different algorithms to assess for example their suitability for our purposes or to select the better one, *and* to be abstract enough so that we don't have to look at minute details of compilers and computer architectures.

To find the right balance between precision and abstraction, we rely on two levels of abstraction: asymptotic analysis and cost models. Asymptotic analysis enables abstracting over small factors contributing to the resource consumption of an algorithm such as the exact time a particular operation may require. Cost models make precise the cost of operations performed by the algorithm but usually only up to the precision of the asymptotic analysis. Of the two forms of cost models, machine-based models and language-based models, in this course , we use a language-based cost model. Perhaps the most important reason for this is that when using a machine-based cost model, the complexity of both the analysis and the specification of the algorithm increases because of the need to reason about the mapping parallel algorithm to actual parallel hardware, which usually involves scheduling of parallel computations over multiple processors.

In the rest of this chapter, we present a brief overview of asymptotic notation, and then discuss cost models and define the cost models used in this course . We finish with recurrence relations and how to solve them.

4.1 Asymptotic Complexity

If we analyze an algorithm precisely, we usually end up with an equation in terms of a variable characterizing the input. For example, by analyzing the work of the algorithm A for problem P in terms of its input size, we may obtain the equation: $W_A(n) = 2n \log n + 3n + 4 \log n + 5$. By

applying the analysis method to another algorithm, algorithm B, we may derive the equation: $W_B(n) = 6n + 7\log^2 n + 8\log n + 9$.

When given such equations, how should we interpret them? For example, which one of the two algorithm should we prefer? It is not easy to tell by simply looking at the two equations. But what we can do is to calculate the two equations for varying values of n and pick the algorithm that does the least amount of work for the values of n that we are interested in.

In the common case, in computer science, what we care most about is how the cost of an algorithm behaves for large values of n—the input size. Asymptotic analysis offers a technique for comparing algorithms at such large input sizes. For example, for the two algorithms that we considered in our example, via asymptotic analysis, we would derive $W_A(n) = \Theta(n \log n)$ and $W_B(n) = \Theta(n)$. Since the first function $n \log n$ grows faster that the second n, we would prefer the second algorithm (for large inputs). The difference between the exact work expressions and the "asymptotic bounds" written in terms of the "Theta" functions is that the latter ignores so called *constant factors*, which are the constants in front of the variables, and *lower-order terms*, which are the terms such as $3n$ and $4\log n$ that diminish in growth with respect to $n \log n$ as n increases.

In addition to enabling us to compare algorithms, asymptotic analysis also allows us to ignore certain details such as the exact time an operation may require to complete on a particular architecture. Specifically, when designing our cost model, we take advantage of this to assign most operations unit costs even if they require more that unit work.

Question 4.1. *Do you know of an algorithm that compared to other algorithms for the same problem, performs asymptotically better at large inputs but poorly at smaller inputs.*

Compared to other algorithms solving the same problem, some algorithm may perform better on larger inputs than on smaller ones. A classical example is the merge-sort algorithm that performs $\Theta(n \log n)$ work but performs much worse on smaller inputs than the asymptotically less efficient $\Theta(n^2)$-work insertion sort. Note that we may not be able to tell that insertion-sort performs better at small input sizes by just comparing their work asymptotically. To do that, we will need to compare their actual work equations which include the constant factors and lower-order terms that asymptotic notation omits.

We now consider the three most important asymptotic functions, the "Big-Oh", "Theta", and "Omega." We also discuss some important conventions that we will follow when doing analysis and using these notations. All of these asymptotic functions are defined based on the notion of asymptotic dominance, which we define below. Throughout this chapter and more generally in this course , the cost functions that we consider must be defined as functions whose domains are natural numbers and whose range is real numbers. Such functions are sometimes called *numeric functions.*

Definition 4.2 (Asymptotic dominance)**.** *Let $f(\cdot)$ and $g(\cdot)$ be two (numeric) functions, we say that $f(\cdot)$ asymptotically dominates $g(\cdot)$ if there exists positive constants c and n_0 such that for all for all $n \geq n_0$,*

$$|g(n)| \leq c \cdot f(n),$$

When a function $f(\cdot)$ asymptotically dominates another $g(\cdot)$, we say that $f(\cdot)$ grows faster than $g(\cdot)$: the absolute value of $g(\cdot)$ does not exceed a constant multiple of $f(\cdot)$ for sufficiently large values.

Big-Oh: $O(\cdot)$. The asymptotic expression $O(f(n))$ is the set of all functions that are asymptotically dominated by the function $f(n)$. Intuitively this means that the set consists of the functions that grow at the same or slower rate than $f(n)$. 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 think of $f(n)$ being an *upper bound* for $g(n)$ because $f(n)$ grows faster than $f(n)$ as n increases.

Definition 4.3. *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 4.4. *Make sure to become very comfortable with asymptotic analysis. Also its different versions such as the $\Theta(\cdot)$ and $\Omega(\cdot)$.*

Exercise 4.5. *Can you illustrate graphically when $g(n) \in O(f(n))$? Show different cases by considering different functions, to hone your understanding.*

Omega notation $\Omega(\cdot)$. The "big-oh" notation gives us a way to upper bound a function but it says nothing about lower bounds. The asymptotic expression $Omega(f(n))$ is the set of all functions that asymptotically dominate the function $f(n)$. Intuitively this means that the set consists of the functions that grow faster than $f(n)$. We write $g(n) \in \Omega(f(n))$ to refer to a function $g(n)$ that is in the set $\Omega(f(n))$. We often think of $f(n)$ being a *lower bound* for $g(n)$.

Definition 4.6. *For a function $g(n)$, we say that $g(n) \in \Omega(f(n))$ if there exist positive constants n_0 and c such that for all $n \geq n_0$, we have $0 \leq c.f(n) \leq g(n)$.*

Theta notation: $\Theta(\cdot)$. The asymptotic expression $\Theta(f(n))$ is the set of all functions that grow at the same rate as $f(n)$. In other words, the set $\Theta(f(n))$ is the set of functions that are both in $O(f(n))$ and $\Omega(f(n))$. We write $g(n) \in \Theta(f(n))$ to refer to a function $g(n)$ that is in the set $\Theta(f(n))$. We often think of $f(n)$ being a *tight bound* for $g(n)$.

Definition 4.7. *For a function* $g(n)$*, we say that* $g(n) \in \Theta(f(n))$ *if there exist positive constants* n_0*,* c_1*, and* c_2 *such that for all* $n \geq n_0$*, we have* $0 \leq c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$*.*

Important conventions. Even though the asymptotic notations $O(\cdot), \Theta(\cdot), \Omega(\cdot)$ all denote sets, we use the equality relation instead of set membership to state that a function belongs to an asymptotic class, e.g., $g(n) = O(f(n))$ instead of $g(n) \in O(f(n))$. This notation makes it easier to use the asymptotic notation. For example, in an expression such as $4W(n/2)+O(n)$, the $O(n)$ refers to some function $g(n) \in O(n)$ that we care not to specify. Be careful, if there are asymptotics are used multiple times an expression, especially in equalities or other relations. For example, in $4W(n/2) + O(n) + \Theta(n^2)$, the $O(n)$ and $\Theta(n^2)$ refer to functions $g(n) \in O(n)$ and $h(n) \in O(n^2)$ that we care not to specify. But in $4W(n/2) + O(n) = \Theta(n^2)$, we mean to say that the equality is satisfied such that for *any* function $g(n) = O(n)$ and we can find some function $h(n) = \Theta(n^2)$ to satisfy the equality.

4.2 Cost Models: Machine and Language Based

Essentially any analysis must assume a *cost model* that specifies the resource cost of the operations that can be performed by an algorithm. Over time, two ways to define cost models have emerged: machine-based and language-based models.

A machine-based model defines the cost of each (kind of) instruction that can be executed by the machine. When using a machine-based model for analysis, we study the instructions executed by the machine when running an algorithm to bound the resources of interest. A language-based model defines cost as a function from the expressions of the language to cost metric. Such a function is usually defined as a recursive function over the different forms of expressions in the language. When using a language-based model for analysis, we analyze the algorithm by using the cost function provided by the model.

Since we are usually interested in performing asymptotic analysis, we can usually simplify our cost functions in both models by ignoring “constant factors” that depend on the specifics of the actual practical hardware our algorithms may execute on. For example, in a machine model, we can assign unit costs to many different kinds of instructions, even though some may be more expensive than others. Similarly, in a language-based model, we can assign unit costs to all primitive operations on numbers, even though the costs of such operations usually vary.

Question 4.8. *What are the advantages of using a machine based and a language-based model?*

There are certain advantages and disadvantages to both models.

The advantage of using machine models is that it is easier to predict the cost of an algorithm when it is executed on actual hardware that is consistent with the machine model. The disadvantage is the complexity of analysis and expressiveness of the languages that can be used for specifying the algorithms. When using a machine model, we have to reason about how the algorithm compiles and runs on that machine. For sequential programs this can be straightforward if the algorithm is expressed in a language that maps easily to the machine model. For example, if we express our algorithm in a low-level language such as C, cost analysis based on a machine model that represents a von Neumanm machine is straightforward because there is an almost one-to-one mapping of statements in C to the instructions of such a machine. For higher-level languages, this becomes somewhat trickier. There may be uncertainties, for example, about the cost of automatic memory management, or the cost of dispatching in an object-oriented language. For parallel programs, cost analysis based on machine models even more tricky, since we may have to reason about how parallel tasks of the algorithm are scheduled on the processors of the machine. Due to this gap between the level at which algorithms are analyzed (machine level) and the level they are usually implemented (programming-language level), there can be difficulties in implementing an algorithm in a high-level language in such a way that matches the bound given by the analysis.

The advantage of using language-based models is that it is easier to predict analyze the algorithm. The disadvantage is that the predicted cost bounds may not precisely reflect the cost observed when the algorithm is executed on actual hardware. This imprecision of the language model, however, can be minimized and in fact essentially eliminated by defining the model to be consistent with the machine model and the programming-language environment assumed such as the compiler and the run-time system. 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, specifically its syntax and its dynamic semantics that specifies how to evaluate the expressions of the language. We thus simply consider the algorithm as expressed and analyze the cost by applying the cost function provided by the model.

Remark 4.9. *We note that both machine models and language-based models usually abstract over existing architectures and programming languages respectively. This is necessary because we wish to our cost analysis to have broader relevance than just a specific architecture or programming language. For example, machine models are usually defined to be valid over many different architectures such as an Intel Nehalem or AMD Phenom. Similarly, language-based models are defined to be applicable to a range of languages. In this course , we use an abstract language that is essentially lambda calculus with some syntactic sugar. As you may know the lambda calculus can be used to model many languages.*

In the sequential algorithms literature, much work is based on machine models rather than language-based model, partly because the mapping from language constructs to machine cost (time or number of instructions) can be made simple in low-level languages, and partly because much work on algorithm predates or coincides with the development of higher-level languages. For parallel algorithms, however, many years of experience shows that machine based models are difficult to use, especially when considering higher-level languages that are commonly used in practice today. For this reason, in this course we will use a language-based cost model. Our language-based model allows us to use abstract costs, work and span, which have no direct meaning on a physical machine.

4.3 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 ith 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)

4.4 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.

4.5 The Work-Span Model

In this course , we will use a language-based cost model to analyze parallel algorithms. From the discussion in Section 4.2, you may recall that the key point that we have be careful about in defining a cost model is that it is can be realized by implementing the necessary compilation and run-time system support. Indeed, for the cost-model that we describe here, this is the case (see Section 4.6 for more details).

Work and Span. The cost model that we use throughout this course is based on two cost metrics: work and span. Roughly speaking, the *work* corresponds to the total number of operations we perform, and *span* to the longest chain of dependencies in the computation.

Example 4.10.

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

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 $\mathtt{mySort}$ is a good sorting algorithm we would have:

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

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

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

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

Definition 4.11 shows the precise definitions of the work and span of PML, the language that we use in this course , by using compositional rules over expressions in the language. In the definition and throughout this course , we write $W(e)$ for the work of the expression and $S(e)$ for its span. As would be expected from a language-based model, the definition follows the definition of the expression language for PML (Section 1.5). We make one simplifying assumption in the presentation: instead of considering general bindings, we only consider the case where a single variable is bound to the value of the expression.

Definition 4.11 (PML Cost Model)**.** *The work and span of PML expressions (Section 1.5) are defined as follows. The notation Eval(e) evaluates the expression e and returns the result, and the notation $[v/x]\ e$ indicates that all free (unbound) occurrences of the variable x in the expression e are replaced with the value v.*

$$
\begin{array}{lcl}
W(v) & = & 1 \\
W(\textbf{fn}\ p \Rightarrow e) & = & 1 \\
W(e_1\ e_2) & = & W(e_1) + W(e_2) + W([\mathit{Eval}(e_2)/x]\ e_3) + 1 \\
 & & \textit{where } \mathit{Eval}(e_1) = \textbf{fn}\ x \Rightarrow e_3 \\
W(e_1\ \mathit{op}\ e_2) & = & W(e_1) + W(e_2) + 1 \\
W(e_1\ ,\ e_2) & = & W(e_1) + W(e_2) + 1 \\
W(e_1\ ||\ e_2) & = & W(e_1) + W(e_2) + 1 \\
W(\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) & = & \begin{cases} W(e_1) + W(e_2) + 1 & \mathit{Eval}(e_1) = \mathit{True} \\ W(e_1) + W(e_3) + 1 & \textit{otherwise} \end{cases} \\
W(\textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end}) & = & W(e_1) + W([\mathit{Eval}(e_1)/x]\ e_2) + 1 \\
W((e)) & = & W(e) \\
W(v) & = & 1 \\
S(\textbf{fn}\ p \Rightarrow e) & = & 1 \\
S(e_1\ e_2) & = & S(e_1) + S(e_2) + 1 \\
S(e_1\ \mathit{op}\ e_2) & = & S(e_1) + S(e_2) + 1 \\
S(e_1\ ,\ e_2) & = & S(e_1) + S(e_2) + 1 \\
S(e_1\ ||\ e_2) & = & \max\left(S(e_1), S(e_2)\right) + 1 \\
S(\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) & = & \begin{cases} S(e_1) + S(e_2) + 1 & \mathit{Eval}(e_1) = \mathit{True} \\ S(e_1) + S(e_3) + 1 & \textit{otherwise} \end{cases} \\
S(\textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end}) & = & S(e_1) + S([\mathit{Eval}(e_1)/x]\ e_2) + 1 \\
S((e)) & = & S(e)
\end{array}
$$

As an example, consider the expression $e_1 + e_2$ where e_1 and e_2 are themselves other expressions (e.g. function calls). Note that this is an instance of the rule the case e_1 op e_2, where op is a plus operation. In PML, we evaluate this expressions by first evaluating e_1 and then e_2 and then computing the sum. The work of the expressions is therefore

$$W(e_1 + e_2) = W(e_1) + W(e_2) + 1.$$

The additional 1 accounts for computation of the sum.

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.

Example 4.12. *Let expressions compose sequentially.*

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

Question 4.13. *In PML, when are expressions evaluated in parallel?*

In PML, we use the notation $(e_1 \mid\mid e_2)$ to mean that the two expressions are evaluated in parallel. The result is a pair of values containing the two results. As a result, the work and span for all expressions except for the parallel construct || are defined in the same way. As we will see later in the course , in addition to the || construct, we assume the set-like notation such as $\{f(x) : x \in A\}$ to be evaluated in parallel, i.e., all calls to $f(x)$ run in parallel.

Example 4.14. *The expression* $(\mathtt{fib}(6) \mid\mid \mathtt{fib}(7))$ *runs the two calls to* `fib` *in parallel and returns the pair* $(8, 13)$*. It does work*

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

and span

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

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

Remark 4.15. *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. Since in PML, we evaluate* e_1 *and* e_2 *sequentially, the span of the expression is calculated in the same way:*

$$S(e_1 + e_2) = S(e_1) + S(e_2) + 1.$$

Note that this does not mean that the span and the work of the expressions are the same! Since PML is purely functional language, we could have in fact evaluated e_1 *and* e_2 *in parallel, wait for the to complete and perform the summation. In this case the span of would have been*

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

Note that since we have to wait for both of the expressions to complete, we take the maximum of their span. Since the can perform the final summation serially after they both return, we add the 1 *to the final span.*
In this book, however, to make it more clear whether expressions are evaluated sequentially or in parallel we will assume that expressions are evaluated in parallel only when indicated by the syntax, i.e., when they are composed with the explicit parallel form.

Remark 4.16. *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 4.17. *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 4.18. *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 4.19 (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 4.20. *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.

4.6 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 4.21. *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 4.22. *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 4.23. *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 4.24. *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 4.25. *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 effects. By moving a job we might have to move data along with it. Because of these effects 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.*

4.7 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.

4.7.1 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:

```
1 function permutations(S) =
2    if |S| = 1 then {S}
3    else
4       flatten({append(⟨ s ⟩ , p)
5                  : s ∈ S, p ∈ permutations(S \ s)})
```

What is the span of this code?

4.7.2 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.

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

4.8 Cost Analysis with Recurrences

The cost of many of the algorithms considered in this course can be analyzed by using recurrences, which are equality or inequality relations that specify a quantity by reference to itself. Such recurrences are especially common in recursive algorithms, where they usually follows the recursive structure of the algorithm, but are a function of size of the arguments instead of the actual values. While recurrence relations are informative to the trained eye, they are not as useful as closed form solutions, which are immediately available. In this section, we will review the three main methods for solving recurrences.

For example, we can write the work of the merge-sort algorithm with a recurrence of the form $W(n) = 2W(n/2) + O(n)$. This corresponds to the fact that for an input of size n, merge sort makes two recursive calls of size $n/2$, and also performs $O(n)$ other work. In particular the merge itself requires $O(n)$ work. Similarly for span we can write a recurrence of the form $S(n) = \max(S(n/2), S(n/2)) + O(\log n) = S(n/2) + O(\log n)$. Since the two recursive calls are parallel, we take the maximum instead of summing them as in work, and since the merge function has to take place after them and has span $O(\log n)$ we add $O(\log n)$.

In the rest of this section, we discuss methods for solving such recurrences after noting a few conventions commonly employed when setting up and solving recurrences.

Conventions and techniques. When we analyze algorithm using recurrences, we usually ignore several technical details. For example, when stating the recurrence for merge sort, we completely ignored the bases cases, we stated only the recursive case. A more precise statement of the recursion would be

$$W(n) = \begin{cases} O(1) & \text{if} \quad n \leq 1 \\ 2W(n/2) + O(n) & \text{otherwise} \end{cases}$$

> **Question 4.27.** *Why is this justified?*

We justify omitting base cases because by definition any algorithm performs constant work on constant-size input. Considering the base case usually changes the closed-form solution of the recursion only by a constant factor, which don't matter in asymptotic analysis. Note however that an algorithm might have multiple cases depending on the input size, and some of those cases might not be constant. It is thus important when writing the recursive relation to determine constants from non-constants.

> **Question 4.28.** *There is still an imprecision in the recursion stated above for merge sort. Can you see what it is?*

There is one more imprecision in the recursion that we stated for merge sort. Note that the size of the input to merge sort n, and in fact many other algorithms, are natural numbers. But $n/2$ is not always a natural number. In fact, the recursion that we stated is precise only for powers of 2. A more precise statement of the recursion would have been:

$$W(n) = \begin{cases} O(1) & \text{if} \quad n \leq 1 \\ W(\lceil n/2 \rceil) + W(\lfloor n/2 \rfloor) + O(n) & \text{otherwise.} \end{cases}$$

We ignore floors and ceiling because they change the size of the input by at most one, which again does not usually affect the closed form by more than a constant factor.

When stating recursions, we may use asymptotic notation to express certain terms such as the $O(n)$ in our example. How do you perform calculations with such terms? The trouble is that if you add any two $O(n)$ terms what you get is a $O(n)$ but you can't do that addition a non-constant many times and still have the result be $O(n)$. To prevent mistakes in calculations, we often replace such terms with a non-asymptotic term and do our calculations with that term. For example, we may replace $O(n)$ with n, $2n$, $2n + \log n + 3$, $3n + 5$, or with something parametric such as $c_1 n + c_2$ where c_1 and c_2 are constants. Such kinds of replacement may introduce some more imprecision to our calculations but again they usually don't matter as they change the closed-form solution by a constant factor.

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.

The idea of the tree method is to consider the recursion tree of the recurrence and to derive an expression that bounds the cost at each level. We can then calculate the total cost by summing over all levels.

To apply the method, we start by replacing the asymptotic notation in the recursion. By the definition of asymptotic complexity, we can establish that

$$W(n) \quad \leq \quad 2W(n/2) + c_1 \cdot n + c_2,$$

where c_1 and c_2 are constants. We now draw a tree to represent the recursion. Since there are two recursive calls, the tree is a binary tree, where each node has 2 children, whose input is half the size of the size of the parent node. We then annotate each node in the tree with its cost noting that if the problem has size m, then the cost, excluding that of the recursive calls, is at most $c_1 \cdot m + c_2$. Figure 4.1 shows the recursion tree annotated with costs.

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?

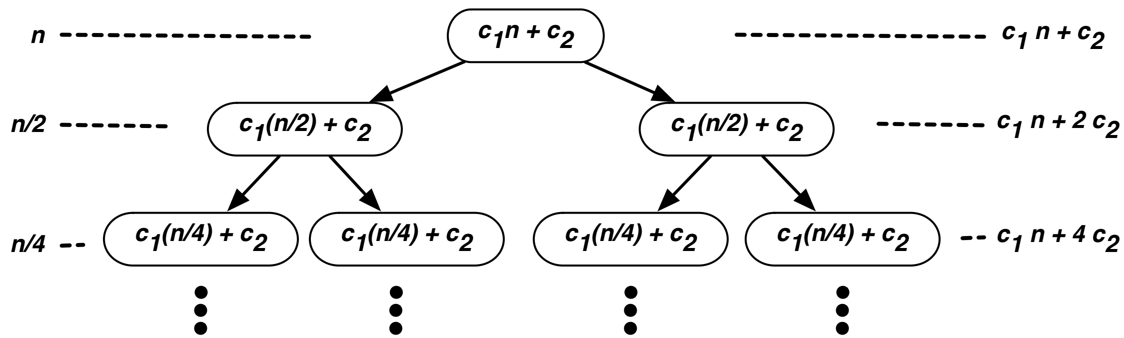

Figure 4.1: Recursion three for the recursion $W(n) \leq 2W(n/2) + c_1 m + c_2$. Each level is annotated with the problem size and the cost at that level.

- 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 $c_1(n/2^i) + c_2$. Thus, the total cost in level i is at most

$$2^i \cdot \left(c_1 \frac{n}{2^i} + c_2\right) \quad = \quad c_1 \cdot n + 2^i \cdot c_2.$$

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

$$\begin{aligned} W(n) &\leq \sum_{i=0}^{\log n} \left(c_1 \cdot n + 2^i \cdot c_2\right) \\ &= c_1 n(1 + \log n) + c_2(n + \tfrac{n}{2} + \tfrac{n}{4} + \cdots + 1) \\ &\leq c_1 n(1 + \log n) + 2c_2 n \\ &\in O(n \log n), \end{aligned}$$

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

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

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: when 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 L_i denote the total cost at level i of the recursion tree. We now check if L_i are consistent with one of the following three cases. For the discussion below let d denote the depth of the tree.

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, $L_{i+1} \geq \rho \cdot L_i$	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, $L_{i+1} \leq \rho \cdot L_i$
++ ++++ ++++++ ++++++++	++++++++ ++++++++ ++++++++ ++++++++	+++++++++ ++++++ ++++ ++
Implication: $O(L_d)$	*Implication:* $O(d \cdot \max_i L_i)$	*Implication:* $O(L_0)$
The house is stable, with a strong foundation.	The house is sort of stable, but don't build too high.	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.

The Substitution Method. Using the definition of big-O, we know that

$$W(n) \leq 2W(n/2) + c_1 \cdot n + c_2,$$

where c_1 and c_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 4.29. *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 $n > 1$. □

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.

.