Chapter 4

Algorithm Design and Analysis

An essential component of algorithm design is the analysis of the resource usage of algorithms. Resources of interest usually include the amount of total work an algorithm performs, the energy it consumes, the time it requires to execute, and the memory and storage space that it requires. When analyzing algorithms, it is important to be precise so that we can compare different algorithms to assess their suitability for our purposes or to select the better one. It is also equally important to be abstract enough so that we don't have to worry about details of compilers and computer architectures, and our analysis remains valid even as these change over time.

To find the right balance between precision and abstraction, we in this book rely on two levels of abstraction: asymptotic analysis and cost models. Asymptotic analysis enables abstracting from small factors such as the exact time a particular operation may require. Cost models specify the cost of operations available in a computational model, usually only up to the precision of the asymptotic analysis. Of the two forms of cost models, machine-based models and language-based models, we use a language-based cost model.

In the rest of this section we present an overview of asymptotic-complexity notation, define the cost models used in this book, and discuss the four main algorithm design techniques and how they may be analyzed.

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 n, we may obtain the equation: $W_A(n) = 2n \lg n + 3n + 4 \lg n + 5$. By applying the analysis method to another algorithm, algorithm B, we may derive the equation: $W_B(n) = 6n + 7 \lg^2 n + 8 \lg n + 9$.

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 we can 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 the cost of an algorithm at large input sizes. Asymptotic analysis offers a technique for comparing algorithms at large input sizes. For example, for the two algorithms that we considered in our example, via asymptotic analysis, we would derive $W_A(n) = \Theta(n \lg n)$ and $W_B(n) = \Theta(n)$. Since the first function $n \lg n$ 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 $4 \lg n$ that diminish in growth with respect to $n \lg n$ 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 that an operation may require to complete on a particular architecture. When designing our cost model, we take advantage of this to assign most operations unit costs even if they require more than unit work.

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 $\Theta(n \lg n)$ 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.

In algorithm design, the three most important asymptotic functions are the "Big-Oh" $O(\cdot)$, "Omega" $\Omega(\cdot)$ and "Theta" ($\Theta(\cdot)$). 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 book, the cost functions must be mappings from the domain of natural numbers to real numbers. Such functions are sometimes called ***numeric functions***.

Definition 4.1. [Asymptotic dominance] Let $f(\cdot)$ and $g(\cdot)$ be two (numeric) functions, we say that $f(\cdot)$ asymptotically dominates $g(\cdot)$ or that $g(\cdot)$ is asymptotically dominated by $f(\cdot)$ if there exists positive constants c and n_0 such that

$$|g(n)| \leq c \cdot f(n), \text{ for all } n \geq n_0.$$

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.

Upper bounds and the Big-Oh: O$(\cdot)$. The asymptotic expression $O(f(n))$, read "order of $f(n)$", or "big-oh of $f(n)$" denotes the set of all functions that are asymptotically dominated by the function $f(n)$. 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 think of $f(n)$ being an ***upper bound*** for $g(n)$ because $f(n)$ grows faster than $g(n)$

as n increases.

Definition 4.2. 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 ($g(n)$ is finite for all n), then it follows that there exist constants c_1 and c_2 such that for all $n \geq 1$,

$$g(n) \ \leq \ c_1 \cdot f(n) + c_2,$$

where, for example, we can take $c_1 = c$ and $c_2 = \sum_{i=1}^{n_0} |g(i)|$.

Exercise 4.3. Can you illustrate graphically when $g(n) \in O(f(n))$? Show different cases by considering different functions.

Lower bounds and the Omega notation: $\Omega(\cdot)$. The "big-Oh" notation gives us a way to upper bound a function but it says nothing about lower bounds. For lower bounds, we use the "Omega" notation: the expression $\Omega(f(n))$ denotes 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 think of $f(n)$ being a ***lower bound*** for $g(n)$.

Definition 4.4. 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 $c.f(n) \leq |g(n)|$.

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 think of $f(n)$ being a ***tight bound*** for $g(n)$.

Definition 4.5. 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 $c_1 \cdot f(n) \leq |g(n)| \leq c_2 \cdot f(n)$.

Important conventions. When using asymyptotic notations, we follow some standard conventions of convenience. First, 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 $g(n) \in O(f(n))$. Although we use the equality, these equalities should not be thought as symmetric: we never write $O(f(n)) = g(n)$. We use the convention that the right hand side of the equation is more abstract or less precise than the left hand side. This convention is consistent with the view that the equality stands for set membership.

We treat expressions that involve asymptotic notation as sets. For example, in $4W(n/2) + O(n)$, the $O(n)$ refers to some function $g(n) \in O(n)$ that we care not to specify. We can think of the expression as the set of all expression that we would obtain by plugging each possibility for $O(n)$. If we have an equality where the asymptotic notation is used both on the left and the right hand side, then we interpret the left hand side as being a subset of the right hand side. For example, consider the equality $4W(n/2) + O(n) = \Theta(n^2)$. This equation says that the set on the left hand side is contained in the set on the right hand side. In other words, for any function $g(n) = O(n)$, there is some function $h(n) = \Theta(n^2)$ that satisfy the actual equality. Note that the "subset" interpretation of equality is a generalization of its set-membership interpretation.

4.2 Cost Models: Machine and Language Based

Any algorithmic analysis must assume a ***cost model*** that specifies the resource cost of the operations that can be performed by an algorithm. There are two broadly accepted ways of defining cost models: machine-based and language-based cost models.

A ***machine-based (cost) model*** takes a machine model as the starting point and defines the cost of each instruction that can be executed by the machine. When using a machine-based model for analyzing an algorithm, we consider the instructions executed by the algorithm on the machine and calculate their total cost. Similarly, a language-based model takes a programming language as the starting point and defines cost as a function mapping the expressions of the language to their cost. Such a cost function is usually defined as a recursive function over the different forms of expressions in the language. When using a language-based model for analyzing an algorithm, we apply the cost function to the expression describing the algorithm in the language.

In both machine-based and language-based cost models, we usually simplify our cost functions by ignoring "constant factors" that depend on the specifics of the actual practical hardware our algorithms may execute on. For example, in a machine-based 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.

There are certain advantages and disadvantages to both models.

The advantage to machine-based models is that they can better approximate the actual cost of an algorithm, i.e., the cost observed when the algorithm is executed on actual hardware. The disadvantage is the complexity of analysis and the limited 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 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 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-based models even more tricky, since we have to reason about how parallel tasks of the algorithm are scheduled on the processors of the ma-

chine. 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 to language-based models is that they simplify analysis of algorithms. 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 even 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.

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 book we 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.

Remark 4.6. 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 book, 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.

4.2.1 The RAM Model for Sequential Computation

Traditionally, algorithms have been analyzed in the Random Access Machine (RAM)[1] model. In this model, a machine consists of a single processor that can access unbounded memory; the memory is 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

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

locations, and conditional and unconditional jumps to other locations in the code. Each instruction takes unit time. The execution-time, or simply ***time*** of a computation is measured in terms of the number of instructions executed by the machine.

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 is suitable for deriving asymptotic bounds (i.e., using big-O, big-Theta and big-Omega) but not for trying to predict exact runtimes. The 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.

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. $f(i) = \lg(i)$. 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 book also does not account for the variance in memory costs, but as with the RAM the costs can be refined.

4.2.2 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 PRAM model, each processor is assigned a unique index, starting from 0, called a ***processor id***, which the processor has access to. Processors in the PRAM model operate under the control of a common clock. For this reason, the PRAM model is considered to be a *synchronous* model. In PRAM, all processors execute the same program. This sometimes leads to programs where each processor executes the same instruction but possibly on different data. For this reason, some PRAM algorithms fit into ***single instruction multiple data*** or ***SIMD*** programming model. However, not all PRAM programs have to be of SIMD type. In fact, since a processor has access to its processor id, it is technically allowed to make local decisions independently of other processors and thus can execute a different instruction that others. Nevertheless, algorithms for the PRAM model are usually specified by defining a single function for each processor to execute. For example we can specify a PRAM algorithm for adding one to each element of a integer array with p elements using p processors by the following program, which is parameterized by the processor id i.

```
(* Input: integer array A. *)
addone =  A[i] ← A[i]+1
```

Since it is synchronous and it requires the algorithm designer to map computation to processors (perform manual scheduling), the PRAM model is can be awkward to work with. 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 book also work with the PRAM, and many of them were originally developed in the context of the PRAM.

4.2.3 The Work-Span Model

In this book, we use a language-based cost model based on work and span to analyze parallel algorithms. As discussed in Section 4.2, have to be careful about the cost model to make sure that it can be mapped to real hardware 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 4.2.4 for more details).

Work and Span. We use a cost model that is based on two cost metrics: work and span. Roughly speaking, the ***work*** of computation corresponds to the total number of operations it performs, and ***span*** correponds to the longest chain of dependencies in the computation.

For a SPARC expression e, we write $W(e)$ for work of e and $S(e)$ for span of e.

Example 4.7.

$W(7+3)$	$=$	Work of adding 7 and 3
$S(\texttt{fib}\ 11)$	$=$	Span for calculating the 11^{th} Fibonacci number
$W(\texttt{mySort}\ a)$	$=$	Work for `mySort` applied to the sequence a

Note that in the third example the sequence a 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 a, and in particular if `mySort` is a good sorting algorithm we would have:

$$W(\texttt{mySort}(a)) = O(|a| \lg |a|)\,.$$

Often instead of writing $|a|$ 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

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

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

Definition 4.8 shows the precise definitions of the work and span of SPARC, our language for describing algorithms. In the definition and throughout this book, we write $W(e)$ for the work of the expression and $S(e)$ for its span. Both work and span are cost functions that map an expression to a cost measured as in units of time. As common in language-based models, the definition follows the definition expressions for SPARC (Chapter 3). 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.

Eval()

Definition 4.8. [SPARC Cost Model] The work and span of SPARC expressions (Chapter 3) 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.

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

span

The basic idea behind the cost model is quite simple: in SPARC, we computations to be composed sequentially or in parallel. If the composition is sequential then, we sum up their work and span to compute the corresponding quantities. If the composition is parallel, then we sum the work to compute total work but take the maximum of the span to compute the total span.

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 SPARC, 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 4.9. Let expressions compose sequentially.

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

In SPARC, 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 $\mid\mid$ are defined in the same way. As we will see later in the book , in addition to the $\mid\mid$ 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 4.10. The expression $(\texttt{fib}(6) \mid\mid \texttt{fib}(7))$ 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 $1 + S(\texttt{fib}(7))$.

Remark 4.11. In purely functional programs, it is always safe to run things in parallel if there is no explicit sequencing. Since in SPARC, 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 SPARC 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

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

Note that since we have to wait for both of the expressions to complete, we take the maximum of their span. Since we 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.

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:

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

As we will discuss soon in Section 4.2.4, parallelism informs us approximately how many processors we can use efficiently.

Example 4.12. For a mergesort with work $\Theta(n \lg n)$ and span $\Theta(\lg^2 n)$ the parallelism would be $\Theta(n/\lg n)$.

Suppose $n = 10,000$ and if $W(n) = \Theta(n^3) \approx 10^{12}$ and $S(n) = \Theta(n \lg n) \approx 10^5$ 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 the span was large, but because the work was reduced.

We can increase parallelism by decreasing span and/or increasing work. Increasing work, however, is not desirable because it leads to an inefficient algorithm. Recall that we refer to a parallel algorithm as (asymptotically) work-efficient if it performs asymptotically the same work as the best serial algorithm (Definition 1.6).

Example 4.13. A (comparison-based) parallel sorting algorithm with $\Theta(n \lg n)$ work is work efficient; one with $\Theta(n^2)$ is not, because we can sort sequentially with $\Theta(n \lg n)$ 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 book 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.

4.2.4 Scheduling

An important advantage of the work-span model is that it 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***.

The goal of scheduling, or a ***scheduler***, is to run a parallel computation on a given number of processors P to minimize the completion time, T_P. Scheduling can be challenging, because a parallel algorithm can generate a massive number of parallel subcomputations as it runs. For maximum performance and efficiency, these parallel subcomputations must be mapped onto the existing processors as tightly as possible, making sure no processor remains unnecessarily idle.

Example 4.14. A parallel algorithm with $\Theta(n/\lg n)$ parallelism can easily generate millions parallel subcomptutations at the same time, even when running on a multicore computer with 10 cores.

In empirical analysis of parallel algorithms, we measure the effectiveness of a parallel algorithm and a scheduler by measuring the P-processor ***speedup*** of the algorithm, defined as W/T_P for varying number of processors. If the speedup matches P then we call it the ***perfect speedup***.

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 greedy scheduling principle.

Definition 4.15. [Greedy scheduling principle] Using a greedy scheduler, a parallel computation with work W and span S can be executed P processors in time T_P such that

$$T_P < \frac{W}{P} + S. \tag{4.1}$$

The greedy-scheduling principle is a generalization of a result, which was originally due to Brent (1974). It is powerful very statement, because:

- the time to execute the computation cannot be less than $\frac{W}{P}$ since we have a total of W work and the best we can possibly do is divide it evenly among the processors,
- the time to execute the computation cannot be less than S because this is the length of the longest chain of sequential dependencies.

Therefore we have

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

We can thus 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. Therefore $\mathbb{P}$ gives us a rough upper bound on the number of processors we can effectively use.

Example 4.16. 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

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

We will show that the span for mergeSort is

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

The parallelism is therefore

$$\mathbb{P} = O\left(\frac{n \lg n}{\lg^2 n}\right) = O\left(\frac{n}{\lg n}\right) \ .$$

This means that for sorting a million keys, we can effectively make use of quite a few processors: $10^6/(\lg_2 10^6) \approx 50,000$.

This important property of work and span—that they can predict the runtime on parallel hardware—allows us to abstract away from an important detail: the number of processors that we are targeting to use. Depending on the application, this number can be 5, 10, 1000, or in the millions. The number of processors used may also vary from one run to another, and in fact even within a run, as for example the operating system distributes available processors among existing applications. It is therefore ideal to design a parallel algorithm independent of the number of processors. In this book, we therefore design algorithms to minimize work and span and assume that they can be implemented to achieve maximal practical efficiency on any number of processors by using a practical parallel programming language. Such a practical programming language can use a greedy scheduler to map the computation onto processors as the algorithm executes.

Remark 4.17. No real scheduler is fully greedy. This is because there is overhead in scheduling the job and thus there will surely be some delay from when a subcomputation becomes ready until when it starts up. Such overheads can be thought as a form of ***scheduling friction***. In practice, the efficiency of a scheduler is therefore quite important to achieving good efficiency. In addition to friction, scheduling can cause memory effects: moving a subcomputation from one processor to another might lead to additional data movement. Because of friction and memory 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.

4.3 Design Techniques

One of the most difficult and probably most important tasks for any practitioner of computer-science is designing their own algorithms. Beginners often feel that they do not even know

where to start when designing an algorithm. In this section, we will briefly describe some of the key techniques used for algorithm design. These techniques are usually a good starting point when designing an algorithm. Even if a particular technique does not lead to an algorithm with the desired properties, it can still add to our understanding of the problem and lead us to a better algorithm. We note that our review of these techniques is not meant to be formal, but rather presented more as a guideline.

4.3.1 Brute Force

The brute-force technique involves trying all possible (underlying) solutions to a problem. In particular a brute-force algorithm enumerates all candidate solutions; it checks for each solution if it is valid, and returns a valid solution or the the best valid solution, depending on the problem.

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. However, there are $n!$ permutations and $n!$ is very large even for modest n. For example, using Sterling's approximation, we know that $100! \approx \frac{100}{e}^{100} \geq 10^{100}$. There are only about 10^{80} atoms in the universe so there is no feasible way we could apply the brute force method directly. We thus conclude that 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.

For example, suppose that we are given a sequences of numbers and we wish to find the largest number in the sequence. The brute-force technique advises uses to try all possibilities. We can do so by picking each element and testing that it is no less than all the other elements in the sequence. When we find one such element, we know that we have found the maximum. Such an algorithm requires $O(n^2)$ comparisons; such an algorithm can be used feasibly for a small inputs.

The brute-force technique is typically easy to parallelize. Enumerating all solutions (e.g., all permutations, all elements in a sequence) is usually easy to do in parallel. Similarly, testing the solutions is inherently parallel. However, the resulting parallel algorithm may not be (and usually is not) efficient. This is important, as discussed earlier in this section, in a parallel algorithm we first care about the total work and only then span or the amount of parallelism.

Despite its inefficiency, there are two reasons for why brute-force algorithms are often helpful. First, the brute-force technique can be very useful is when testing the correctness of more efficient algorithms. Even if inefficient for large n the brute-force technique could work well for testing small inputs, especially because brute-force algorithms tend to be simple. The second reason is that brute-force algorithms are often easy to design and are a good starting point toward a more efficient algorithm.

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

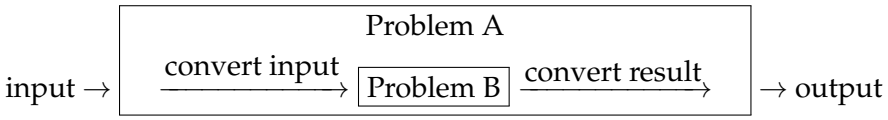


Figure 4.1: An illustration of a reduction from problem A to problem B.

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 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.

Exercise 4.18. Describe a brute-force algorithm for deciding whether an integer n is composite (not prime).

4.3.2 Reduction

Sometimes the easiest approach to solving a problem is to reduce it to another problem for which known algorithms exist. More precisely, to reduce a problem A to a problem B, we use some preprocessing to convert the input of problem A to an input for problem B, then solve problem B on that input, and finally convert the result of problem B back to the result of problem A. Figure 4.1 illustrates a reduction.

When reducing one problem to another it is important to include the cost of converting the input and converting the result. Indeed to calculate the total work for reducing a problem A to B we can just add the work of the two conversions to the cost of an algorithm for solving B. We say a reduction of A to B is ***efficient*** if the conversion takes no more work or span (asymptotically) than B on the same input size as A. Thus an efficient reduction of problem A to problem B tells us that problem A is effectively as easy as problem B (at least within a constant factor).

Example 4.19. We can reduce the problem of finding the largest element in a sequence to the problem of sorting. The idea is to sort the sequence in ascending order and then pick the largest element, which is the at the end of the sequence. Assuming accessing the final element requires less work than sorting, which it usually does, this reduction is efficient. Note, however, that the resulting algorithm is not a work-efficient algorithm, because it would require $O(n \lg n)$ work even though we can find the maximum in $O(n)$ work.

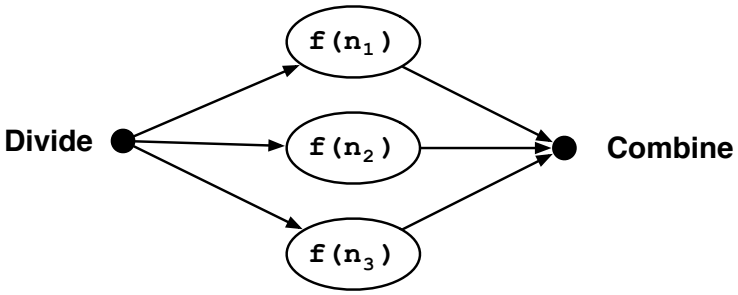


Figure 4.2: Structure of a divide-and-conquer algorithm illustrated ($k = 3$). The independent problems $f(n_1)$, $f(n_2)$, and $f(n_3)$ can be solved in parallel.

Example 4.20. Consider a function `maxVal` that finds the maximum of a set of numbers. We can reduce the problem of finding the minimum of a set of numbers to this problem. More specifically, we can simply invert the sign of all the numbers, use `maxVal` on those numbers, and then invert the sign of the result. This reduction requires asympytotically linear work in the size of the input set. It is therefore efficient if `maxVal` requires same or more work, but is inefficient otherwise.

Reduction is a powerful technique because we can reduce a problem to another that appears very different or even unrelated. In Chapter 5, for example, we reduce a problem on strings of characters to one on graphs. Reduction can also be used "in the other direction" to show that some problem is at least as hard as another or to establish a lower bound. In particular, if we know (or conjecture) that problem A is hard (e.g., requires exponential work), and we can reduce it to problem B (e.g., using polynomial work), then we know that B must also be hard. Indeed the theory of NP-complete problems is based on this idea.

4.3.3 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 the following.

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. Figure 4.2 illustrates the approach for $k = 3$. Since the subproblems are independent, they can be solved in parallel.

Example 4.21. We can find the largest element in a sequence a using divide and conquer as follows. If the sequence has only one element, we return that element, otherwise, we divide the sequence into two equal halves and recursively and in parallel compute the largest element in each half. We then return the largest of the results from the two recursive calls. For a sequence of length n, we can write the work and span for this algorithm as recurrences as follows:

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

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

As discussed later in this chapter, we can solve the recurrences to obtain $W(n) = \Theta(n)$ and $S(n) = \Theta(\lg n)$.

Example 4.22. Generalizing over the algorithm for computing the largest element, leads us to an important parallel-algorithm design concept called reduction. A ***reduction*** refers to a computation that involves applying an associative binary operation op to the elements of a sequence to obtain (reduce the sequence to) a final value. For example, reducing the sequence $\langle 0, 1, 2, 3, 4 \rangle$ with the $+$ operation gives us $0+1+2+3+4 = 10$. If the operation requires constant work (and thus span), then the work and span of a reduction is $\Theta(n)$ and $\Theta(\lg n)$ respectively.

Example 4.23. We can sort a sequence using divide and conquer as follows. If the sequence has only one element, we return the sequence unchanged because it is sorted. Otherwise, we divide the sequence into two equal halves and recursively and in parallel sort each half. We then merge the results from the two recursive calls. Assuming that merging can be done in $\Theta(n)$ work and $\Theta(\lg n)$ span, where n is the sum of the lengths of the two sequences, we can write the work and span for this sorting algorithm as recurrences as follows

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

$$S(n) = \begin{cases} \Theta(1) & \text{if} \quad n \leq 1 \\ S(n/2) + \Theta(\lg n) & \text{otherwise.} \end{cases}$$

As discussed later in this chapter, we can solve the recurrences to obtain $W(n) = \Theta(n \lg n)$ and $S(n) = \Theta(\lg^2 n)$.

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 by combining the results from the smaller instances. Contraction only differs from divide and conquer in that it allows there to be only one independent subproblem to be

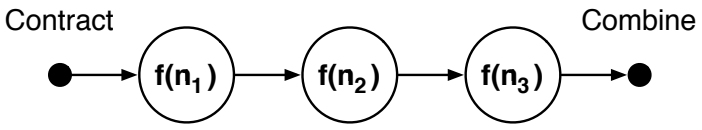


Figure 4.3: Structure of a contraction algorithm illustrated.

solved at a time, though there could be multiple dependent subproblems to be solved one after another (sequentially).

Example 4.24. We can find the largest element in a sequence a using contraction as follows. If the sequence has only one element, we return that element, otherwise, we can map the sequence a into a sequence b which is half the length by comparing the elements of a at consecutive even-odd positions and writing the larger into b. For example, we can map the sequence $\langle 1, 2, 4, 3, 6, 5 \rangle$ to $\langle 2, 4, 6 \rangle$. We then find the largest in b and return this as the result. For a sequence of length n, we can write the work and span for this algorithm as recurrences as follows

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

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

Using the techniques discussed at the end of this chapter, we can solve the recurrences to obtain $WCn) = \Theta(n)$ and $S(n) = \Theta(\lg n)$.

Dynamic programming. Like divide and conquer, dynamic programming divides the problem into smaller subproblems, solves the subproblems, and combines the solutions to the subproblems. The difference, though, is that the solutions to subproblems are used multiple times. It therefore becomes important to store the solutions in a data structure that facilitates quick re-use as needed. We shall see many examples of dynamic programming in Chapter 19.

Greedy. For a problem on size n use some approach to find the "best" element by some greedy metric, remove this element, and solve the problem on the remaining $n - 1$ elements. We shall see an example use of the greedy technique in Chapter 5.

4.3.4 Randomization

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

Example 4.25. We can sort a sequence using divide and conquer and randomization as follows. If the sequence has only one element, we return the sequence unchanged because it is sorted. Otherwise, we select one of the elements as a ***pivot*** and divide the sequence into two sequences consisting of the keys less than, and greater than the pivot. We then sort each recursively and the concatenate the results in order. Assuming that splitting based on the pivot can be done in $\Theta(n)$ work and and assuming that the pivot divides the sequence into two equal halves, we can write the work and span for this sorting algorithm as recurrences as follows

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

$$S(n) = \begin{cases} \Theta(1) & \text{if} \quad n \leq 1 \\ S(n/2) + \Theta(\lg n) & \text{otherwise.} \end{cases}$$

We can solve the recurrences to obtain $WCn) = \Theta(n \lg n)$ and $S(n) = \Theta(\lg^2 n)$.

The assumption we made about pivot dividing the sequence into two equal halves in unrealistic. As discussed in Chapter 11, we can analyze the "expected work" and "expected span" of randomized algorithm more precisely by using probabilistic analysis techniques.

Randomization plays a crucial role in parallel algorithm design because it helps "break" the symmetry in a problem without global communication. We will cover several examples of randomized algorithms that use symmetry breaking in this book. Formal cost analysis for randomized algorithms requires knowledge of probability theory. In Chapter 11, we cover the probability theory required by the material covered in the book.

Example 4.26. [Symmetry breaking] We often perform randomized symmetry breaking when walking on a crowded sidewalk with many people coming in our direction. With each person we encounter, we may pick a random direction to turn and the other person responds, or the other way. 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 but make the same (really opposite) decisions. Since we make essentially random decisions, the symmetry is eventually broken—or we run into each other.

4.4 Cost Analysis with Recurrences

The cost of many algorithms 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 $S(n) = \max(S(n/2), S(n/2)) + O(\lg n) = S(n/2) + O(\lg n)$. 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 $O(\lg n)$ we add $O(\lg n)$.

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 analyzing algorithms 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

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

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.

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, $2n$, $2n+\lg n+3$, $3n+5$, 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 . Our goal is to derive a closed form solution to this recursion.

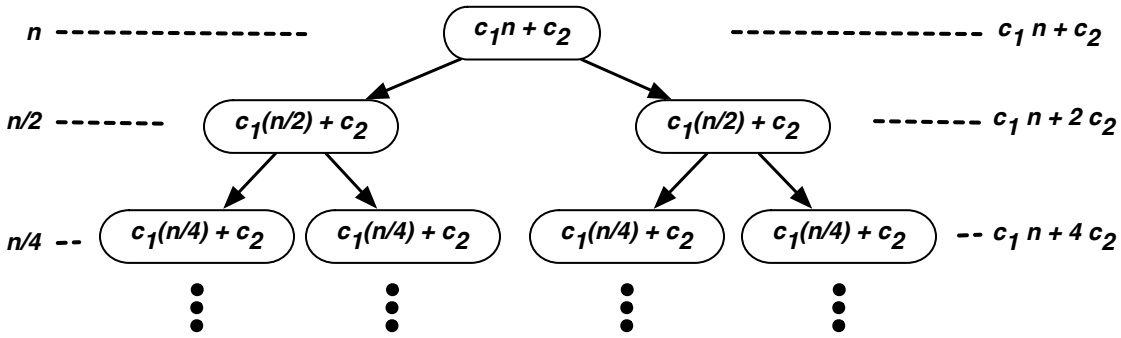


Figure 4.4: Recursion tree for the recursion $W(n) \leq 2W(n/2)+c_1m+c_2$. Each level is annotated with the problem size and the cost at that level.

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 4.4 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?
- 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 $1 + \lg n$. Hence, we have

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

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. Based on this, we check if L_i are consistent with one of the three cases and determine the bound as shown in Definition 4.27.

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.

Definition 4.27. [Brick Method]

For the definition let d denote the depth of the tree as in the tree method.

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, $L_{i+1} \geq \rho \cdot L_i$. The total cost is $O(L_d)$.	All levels have approximately the same cost. The total cost is $O(d \cdot \max_i L_i)$.	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, $L_{i+1} \leq \rho \cdot L_i$. The total cost is $O(L_0)$.
++ ++++ ++++++ ++++++++	++++++++ ++++++++ ++++++++ ++++++++	++++++++ ++++++ ++++ ++

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 4.28. *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*

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

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

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

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

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

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

4.5 What makes a good solution?

When you encounter an algorithms or a data-structures 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. A solution that is correct, and
2. a solution that has the lowest cost possible.

Ideally, the correctness and efficiency of the algorithm or data structure should be proven.

Algorithms designed with inductive techniques can be proved correct using (strong) induction. Similarly, we can often express the complexity of inductive algorithms with recursions and solve them to obtain a closed-form solution.

Figure 4.5: 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.

4.6 Problems

4-1 Closed form solutions

Find the closed from for the following recursions. First use the tree method by writing out the sum and solving it. Then apply the brick method and compare your answers.

- $W(n) = 3W(n/2) + n$
- $S(n) = 2S(n/4) + n$
- $W(n) = 2W(n/4) + n^2$
- $W(n) = 2W(n/2) + n \log n$
- $W(n) = 2W(n/2) + n \log n$
- $S(n) = 2S(\sqrt{n}) + \log n$
- $W(n) = W(n/3) + W(n/4) + n$
- $S(n) = \max(S(n/3), S(n/4)) + \log n$
- $W(n, m) = 2W(n/2, m/4) + nm$
- $W(n, m) = W(n/2, m/4) + n^2 m$
- $W(n, m) = W(n/2, m/4) + \log nm$

4-2 Substitution Method

Show that $W(n) = 2W(n/2) + \log n \in O(n)$ by using tree of substitution method.

4-3 Bricks and trees

Derive the brick method from the tree method.

4-4 2-Optimality of Brent's Theorem

Prove that the bound given by Brent's theorem is within a factor two of optimal.

4-5 Order statistics by reduction to sorting

Suppose that you have an algorithm that can, for given a comparison function, comparison sort a sequence of numbers in $\Theta(n \log n)$ work and $\Theta(\log^2 n)$ span. Using the problem reduction technique:

- Give an algorithm that finds the minimum element in a given sequence of numbers in the same work and time.
- Give an algorithm that finds the maximum element in a given sequence of numbers in the same work and time.
- Give an algorithm that finds the median element in a given sequence of numbers in the same work and time.

- Give an algorithm that finds the element with any given rank in a given sequence of numbers in the same work and time.

4-6 Repeated strings
Describe a brute force algorithm for finding the length of the longest repeated string in a string of characters. For example for the string "`axabaxcabaxa`" the longest repeated string is `abax` and has length 4. You can assume you are given a routine `find`(S, w) that returns how many times the string w appears in the string S. Your algorithm should generate about n^2 candidate solutions, where n is the length of the input string.

4-7 Graph connectivity
Suppose that you know how to solve the problem of determining whether there is a path from any two vertices in a given undirected graph as long as all the vertices in the graph have degree 3 or less. Using reduction, solve the problem of determining whether any two vertices are connected in an arbitrary degree graph. Is your reduction efficient?

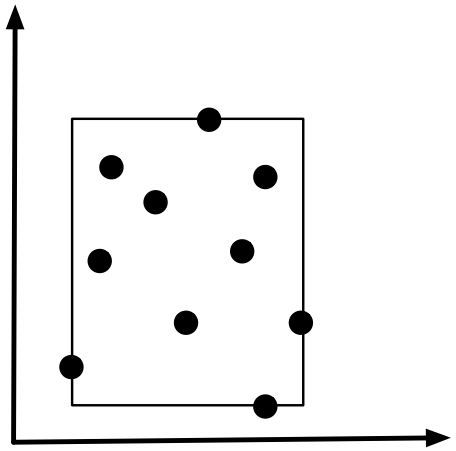
4-8 Multiplication by addition
Suppose that you have an algorithm that can sum up the floating points numbers in a given sequence in $\Theta(n)$ work and $\Theta(\log n)$ span. Using the problem reduction technique: give an algorithm that finds the product of the numbers in a given sequence of numbers in the same work and time. Explain the assumptions that you need to make sure that the result is correct.

4-9 Sorting via reduction to convex hulls
Given a sequence of points in two dimensions P, the planar convex hull problem requires finding the convex hull of the points, which is the smallest polygon containing P. The polygon is defined by the sequence Q, consisting of a subset of the points in P ordered clockwise, such that no three points are collinear (on the same line).

- Design a sorting algorithm by reduction to the convex hull problem.

- State the work and the span of your algorithms in terms of the work and the span of a convex hull algorithm.

4-10 Smallest enclosing rectangle
You are given a set of points in two dimensions and asked to find the smallest axis-aligned rectangle that encloses the points as illustrated in the drawing below. An axis-aligned rectangle is one whose sides are parallel to the "x" and "y" axes. Design a divide-and-conquer algorithm for finding rectangle.



4-11 Academic life
Every morning, a professor wakes up to perform a collection of n tasks $t_1 \ldots t_n$, where each task has a known length l_i. Each day, the professor completes the tasks in some order, performing one task at a time, and thus assigning a finish time f_i to each. Over time, the professor has developed a strategy of minimizing the average completion time of these tasks, that is $\frac{\sum_{i=1}^{n} f_i}{n}$. Exactly why this strategy works continues to be an (unfunded) research project.

- Design a brute-force algorithm that minimizes the average completion time.
- What is the work and span of your brute-force algorithm.
- Design a reduction-based algorithm by sorting that minimizes the average completion time.
- What is the work and span of your reduction-based algorithm.
- Prove that your reduction-based algorithm minimizes the average completion time.

4-12 Greedy dining
It is lunch time and you are very hungry today. But you don't want to spend any more than your usual budget, $5, on lunch. Describe a greedy algorithm for having a big lunch without exceeding your budget.

.