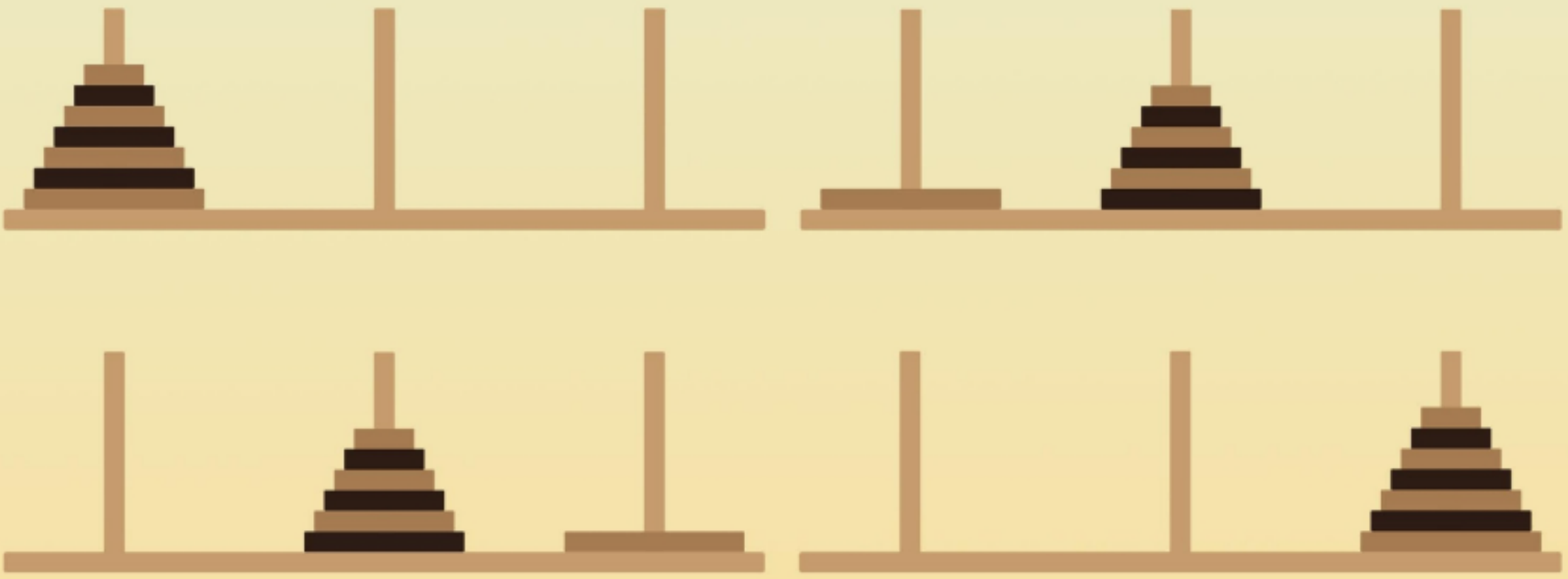
Introduction to the Design and Analysis of Algorithms

A Multi-Paradigm Approach



Arthur Nunes

Introduction to the Design and Analysis of Algorithms

A Multi-Paradigm Approach

Arthur Nunes

Kendall Hunt
publishing company

Kendall Hunt
publishing company

www.kendallhunt.com
Send all inquiries to:
4050 Westmark Drive
Dubuque, IA 52004-1840

ISBN 979-8-7657-2058-5

Published in the United States of America

Contents

Part I

Fundamentals

Chapter 1

Introduction**

1.1 Origin of the Algorithm

The word *algorithm* is derived from the name of the medieval Persian mathematician Muhammad ibn Musa al-Khwarizmi (محمد بن موسى الخوَرزمی) or Mohammed son of Moses the Khwarizmite. His math text introduced Hindü-Arabic numerals, and the methods of arithmetic using these numerals became known as the *algorism*, a variation of al-Khwarizmi. This word evolved into its present form, and came to mean any effective procedure.

The study of algorithms is ancient. In addition to arithmetic, other results have been of interest. For example, an algorithm for approximating the values of square roots was known to Heron of Alexandria in the first century CE; further, it seems this algorithm was even known to the Babylonians in about 1700 BCE (almost 4000 years ago). Another example is Euclid's algorithm for finding greatest common divisors; it was published in Euclid's *Elements* in 300 BCE as a general algorithm rather than mere examples.

It used to be that mathematicians would use these algorithms to compute the desired numerical result by hand. With the creation of computing machinery in the twentieth century, numerical methods and non-numerical methods have become more and more important since their execution has become automated. This book focuses on significant non-numerical methods and discusses approaches to their design and analysis.

1.2 Two Notations for Expressing Algorithms

As we have discussed, algorithms emerge from a long mathematical tradition. It is only appropriate to use mathematical notation to describe algorithms. Specifically, we can describe algorithms by writing mathematical functions; this notation is called *functional*. Yet, when we consider the computer as a machine, we understand that it must perform a sequence of steps. So it is appropriate to use a step-by-step notation to describe algorithms; this one is called *imperative*. One might be tempted to pick only one of these approaches, but we privilege neither and happily use both.

1.2.1 Functional Notation

We make use of the traditional notation for mathematical functions, with some small extensions, to describe algorithms. Thus we write a function definition as the name of the function, the formal parameters (separated by commas or semicolons) in parentheses, an equal sign, and finally an expression. For example, we express the function to square a number as $f(x) = x^2$.

We also write conditionals in the traditional manner where the cases are listed following a single brace. For example, we express the definition of the absolute value function as follows.

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise} \end{cases}$$

There are no looping constructs; only recursion is used. And there is no assignment, but *let*-expressions and *where*-expressions can be used to give names to intermediate values.

For example, the function to compute the larger root of a quadratic equation could be expressed using either of the following definitions.

$$\begin{aligned} \text{root}(a,b,c) &= \text{let } d = \sqrt{b^2 - 4 \times a \times c} \text{ in } (-b+d)/(2 \times a) \\ \text{root}(a,b,c) &= (-b+d)/(2 \times a) \text{ where } d = \sqrt{b^2 - 4 \times a \times c} \end{aligned}$$

The evaluation of a function application occurs in the usual way: evaluate the actual parameters, replace the formal parameters with the actual parameters, and then evaluate the body of the function.

Next, we review what we typically find in a mathematical expression and then describe the extensions needed: arrays, structures, linked lists, and pattern matching.

The Usual Suspects

An expression consists of either a conditional-expression, a naming-expression, a function-expression, an application-expression, a variable, or a value. We have already seen conditional-expressions and naming-expressions.

A function-expression corresponds to the declaration of an anonymous function. It is written starting with the Greek letter λ, the formal parameters in parentheses, a period, and finally an expression. For example, we express the anonymous function to square a number as $\lambda(x).x^2$.

An application-expression is the application of a function. It is often written $e_0(e_1, \cdots, e_n)$, where each e_i is an expression — semicolons may also be used as separators. The first expression e_0 is usually just the name of a function. For example, we can express applying the squaring function f to an expression denoting three as $f(2+1)$; further, we can express applying the anonymous function to square a number to an expression denoting three as $(\lambda(x).x^2)(2+1)$.

An application-expression may also be written using traditional mathematical notation. For example, we have already seen that applying the addition operator to the numbers 2 and 1 is written using the traditional infix notation $2+1$. Similarly, the exponentiation operator is expressed using a superscript so squaring x is written as x^2.

A variable is the name of a value (including the name of a function). Often we follow the mathematical convention of having short names using merely one or two letters. However, following the computing convention, we also use longer names.

A value includes the usual mathematical values: numbers, vectors, and matrices. Regarding numbers, they are written in the usual way. We make use of every variety: the positive integers ($\mathbb{Z}^+$), the natural numbers[1] ($\mathbb{N}$), the integers ($\mathbb{Z}$), the rational numbers ($\mathbb{Q}$), the real numbers ($\mathbb{R}$), the extended real line[2] ($\bar{\mathbb{R}}$), and the complex numbers ($\mathbb{C}$). Further, a value includes the customary computing values: characters, strings, and Booleans ($\mathbb{B}$). Literal characters are written following the computing convention between single quotation marks — for example, 'a'. Also following the computing convention, literal strings are written between double quotation marks — for example, "Hello". Literal Booleans are written following the mathematical convention as $\top$ and $\bot$. In addition to vectors, we allow arbitrary Cartesian products, which are called tuples; for example, an element of $\mathbb{N} \times \Sigma \times \mathbb{B}$ is $(2, \text{'a'}, \top)$, where Σ is the set of characters. Finally, a value includes arrays and structures discussed in the subsequent sections.

Arrays

An *array* is a sequence of elements where any element can be accessed in constant time.[3] We use the notation $\langle 77, 88, 99 \rangle$ to denote an array containing the elements 77, 88, and 99 in that order, and we use square brackets after an array to denote access by index. And so if $a = \langle x_0, x_1, x_2 \rangle$ then $a[0] = x_0$, and so on. We always use zero for the first index. Vertical bars are overloaded and when used on an array denotes its length; if a is defined as before then $|a| = 3$. Plus is overloaded and when used on arrays denotes their concatenation; $\langle 77, 88, 99 \rangle + \langle 100, 200 \rangle = \langle 77, 88, 99, 100, 200 \rangle$.

We allow *array slicing*. An array slice is a sub-array where the beginning and end of the slice are each specified by array indices. Here the indices are inclusive. Note that if the beginning index is greater than the ending index, the slice is empty. If the ending index is not specified, that is understood as the index of the last element of the array. When composing slices or a slice and an access operation, we must increment by the first slice's beginning index. We assume that slicing takes linear time.

$$\begin{array}{lcll} a[i:j] & = & \langle a_i, \cdots, a_j \rangle & \text{if } a = \langle a_0, \cdots, a_i, \cdots, a_j, \cdots \rangle \\ a[i:j] & = & \langle\rangle & \text{if } i > j \\ a[i:] & = & \langle a_i, \cdots \rangle & \text{if } a = \langle a_0, \cdots, a_i, \cdots \rangle \end{array}$$

Structures and Unions

It is common to characterize data as data consisting of parts or data consisting of choices. Mathematically, the parts are put together via the Cartesian product and choices are put together via a union or disjoint union. For example, a point characterized as an ordered pair consists of parts: the x-coordinate and the y-coordinate. Booleans are an example of data consisting of choices. When computing we frequently give names to these Cartesian products to avoid confusion and, in that case, we call the result

[1] The natural numbers include zero.

[2] The extended real line is the set of real numbers together with positive and negative infinity.

[3] Some are less restrictive about the access time for an element in an array, and might, for example, allow logarithmic access time. In this book, we assume constant access time.

a *structure*. Thus we would write a particular point as point(2,3). To select a part from a structure we use either pattern matching or dot-notation (e.g., point(2,3).x = 2 and point(2,3).y = 3). No additional notation is needed for data characterized by choices. It is possible to describe data using both parts and choices; we will see an important example in the next section (where we will use special notation for the structures).

Linked Lists and Recursive Data

A *linked list*, or *list*, is a sequence of elements where the sequence can be extended on the left in constant time, the *head* and the *tail* can be accessed in constant time, but accessing an arbitrary element takes linear time. We use the notation $[77,88,99]$ to denote a list containing the elements 77, 88, and 99 in that order, and we use a double colon to denote extending the list. And so if $\ell = [77,88,99]$ then $66 :: \ell = [66,77,88,99]$; in general, $[66,77,88,99]$ is synonymous with $66 :: 77 :: 88 :: 99 :: []$. Again, if $\ell = [77,88,99]$ then the head of ℓ is 77 and the tail of ℓ is $[88,99]$; in general, the head of the list is its first element and the tail is the list of remaining elements. Vertical bars are overloaded and when used on a list denotes its length; if ℓ is defined as before then $|\ell| = 3$. Plus is overloaded and when used on lists denotes their concatenation; $[77,88,99] + [100,200] = [77,88,99,100,200]$.

We can use a *list comprehension* to describe a list that is characterized relative to another list. For example, if $\ell = [-1,12,-2,21,30,-3]$, then $[x \in \ell \mid x > 0] = [12,21,30]$; in general, $[x \in xs \mid p(x)] = \mathit{filter}(xs, p)$, where the function *filter* is defined in the subsequent section.

We can define the list data structure more formally using the ideas of data consisting of parts, data consisting of choices, and recursion. Recursion is natural because a non-empty sequence by its nature contains a subsequence.

Definition 1. *A* list *is one of the following.*

- $[]$, *the empty list, or*
- $x :: xs$, *a non-empty list or* cons, *where*
  - x, *the head, is a data value; and*
  - xs, *the tail, is a* list.

Pattern Matching

When writing naming-expressions or when defining functions, we allow more on the left-hand side than variables; *patterns* are permitted. Their use is traditional in mathematics and makes for succinct expression of naming components and of performing certain kinds of case analyses. For example, we can extract a component of a tuple using pattering matching: $(\text{let } (x,\_) = (3,4) \text{ in } x) = 3$. This example contains a tuple-pattern that contains a variable-pattern and a wildcard.

A pattern consists of either a constant-pattern, a variable-pattern, a wildcard, a tuple-pattern, a structure-pattern, or a numeric-pattern. A constant-pattern looks like a value and matches the value it looks like. A variable-pattern looks like a variable and matches anything; it can be used in an expression, but there is a restriction: a variable-pattern cannot be repeated in a pattern. A wildcard looks like an underscore (_) and matches anything and can be repeated in a pattern but cannot occur in an expression. A tuple-pattern looks like a tuple, but its components are patterns; a tuple-pattern matches a tuple if the component patterns of the tuple pattern match the components of the tuple. Similarly, a structure-pattern looks like a structure, but its components are patterns; a structure-pattern matches a structure if

it is the same kind of structure and the component patterns of the structure pattern match the components of the structure. Finally, a numeric-pattern has the form $k+1$, $2k$, or $2k+1$; these patterns match non-zero natural numbers — the last two match even and odd numbers, respectively. For example, (let $k+1=3$ in k) $=2$, (let $2k=4$ in k) $=2$, and (let $2k+1=5$ in k) $=2$.

When defining functions, patterns are used to perform a case analysis on data characterized by choices. Testing for a pattern match occurs from top to bottom and terminates with the first successful match. To illustrate the use of patterns, we explicitly define length, concatenation, and filter on lists using pattern matching.

To compute the length we observe that if the list is empty (constant-pattern), the answer is zero. If the list is non-empty (structure-pattern), the answer is one more than the length of the tail.

$$\begin{array}{lcll} |[]| & = & 0 & \text{length of the empty list} \\ |x :: xs| & = & 1 + |xs| & \text{length of a non-empty list} \end{array}$$

To compute the concatenation of two lists we observe that if the first list is empty, the answer is the second list. If the first list is non-empty, the answer must start with the head of the first; the rest is the tail of the first concatenated with the second.

$$\begin{array}{lcll} [] + ys & = & ys & \text{concatenating the empty list} \\ (x :: xs) + ys & = & x :: (xs + ys) & \text{concatenating a non-empty list} \end{array}$$

To filter a list given a predicate we observe that if the list is empty, the answer must also be empty. If the list is non-empty and the first element satisfies the predicate, the answer must start with the head of the list; the rest is just the filtered tail. If the list is non-empty but the first element does not satisfy the predicate, the answer is just the filtered tail.

$$\begin{array}{lcll} \mathit{filter}([], p) & = & [] & \text{filtering the empty list} \\ \mathit{filter}(x :: xs, p) & = & \begin{cases} x :: \mathit{filter}(xs, p) & \text{if } p(x) \\ \mathit{filter}(xs, p) & \text{otherwise} \end{cases} & \text{filtering a non-empty list} \end{array}$$

1.2.2 Imperative Notation

There are many similarities between the functional and the imperative notation. Expressions are also used in imperative notation, but they can only be written as part of a statement. Further, just as functions were central to the functional notation, procedures are central to the imperative notation. A procedure definition is introduced with the keyword **def**, followed by the name of the function and its formal parameters, where the line ends with a colon (:). We express conditionals with an **if**-statement and use an explicit **return**-statement to return values. So we express the definition of the absolute value function as follows.

def ABS(x) :
1. **if** $x \geq 0$:
2. **return** x
3. **else** :
4. **return** $-x$

Here too we take for granted that the execution of a procedure application evaluates all the actual parameters before the body is executed. We also assume that no data structures are copied. There are

also some differences from the functional notation. Pattern matching is severely limited. While we feel free to use recursion, there are several looping constructs. There is assignment, which is expressed using an arrow.

Looping Constructs, Assignment, and Arrays

The fundamental looping construct is the **while**-loop. It is introduced with the keyword **while**, followed by a Boolean valued expression, where the line ends with a colon (:). The **while**-loop statements are indented underneath. For example, the pseudo-code below displays the numbers from one to ten; assignment is used to initialize and update the variable i.

```
1. i ← 1
2. while i < 11 :
3.     PRINT(i)
4.     i ← i + 1
```

A **for**-loop can achieve the same effect more easily but is less general. It is introduced with the keyword **for**, followed by a variable, an arrow, an integer valued expression, the keyword **to**, and another integer valued expression; the line ends with a colon (:). The values are understood as inclusive. The **for**-loop statements are indented underneath. For example, the pseudo-code below also displays the numbers from one to ten.

```
1. for i ← 1 to 10 :
2.     PRINT(i)
```

A **foreach**-loop is for iterating through a data structure. It is introduced with the keyword **for**, followed by a variable, the symbol $\in$, and a data structure valued expression; the line ends with a colon (:). The **foreach**-loop statements are indented underneath.

Arrays are created with the expression **array**(n). The natural number n is the size of the array. In the following example, an array is created; subsequently it is initialized to contain the numbers from one to ten. Then the contents of the array is displayed; here too, the numbers from one to ten.

```
1. a ← array(10)
2. for i ← 0 to 9 :
3.     a[i] ← i + 1
4. for x ∈ a :
5.     PRINT(x)
```

Chapter 2

Design Overview**

2.1 Perspectives on Design

The key design technique is recursion. We will see that there are various forms of recursion such as *structural recursion* and *divide and conquer*.[1] A recursive algorithm implementation can consume a substantial amount of resources; therefore, there are additional design techniques that mitigate this issue: *tail recursion*, *dynamic programming*, and *greedy algorithms*. We consider each of these in turn.

2.2 Structural Recursion

Given a recursive data structure, *structural recursion* is a recursive approach to algorithm design which restricts the form of the algorithm so as to follow the form of the recursive data structure. This restriction entails that the case analysis in the algorithm is the same case analysis that occurs in the definition of the data structure, and that recursion happens exactly in those places where there is recursive substructure. The benefit of these restrictions is that it turns algorithm design into fill in the blanks. The downside is that the performance of structurally recursive algorithms tends to be mediocre. While this approach applies to any recursive data structure, we consider as concrete examples the list data structure and the natural numbers.
Recall the definition of a list.

A *list* is one of the following.

- [], the empty list, or
- $x :: xs$, a non-empty list or cons, where
  - x, the head, is a data value; and
  - xs, the tail, is a *list*.

[1] Some refer to the divide and conquer form of recursion as *generative recursion*.

A structurally recursive function on a list will take the following abstract form.[2]

$$\begin{array}{lcll} f([]; y_1, \cdots, y_n) & = & \ldots y_1 \ldots y_n \ldots & \text{The answer for the empty list.} \\ f(x :: xs; y_1, \cdots, y_n) & = & \ldots x :: xs \ldots y_1 \ldots y_n \ldots f(xs; y_1, \cdots, y_n) \ldots & \text{The recursive call occurs on the recursive part.} \end{array}$$

The natural numbers ($\mathbb{N}$) can also be characterized recursively.

- $0 \in \mathbb{N}$, and
- $n+1 \in \mathbb{N}$ if $n \in \mathbb{N}$

A structurally recursive[3] function on the natural numbers will take the following abstract form.

$$\begin{array}{lcll} f(0; y_1, \cdots, y_n) & = & \ldots y_1 \ldots y_n \ldots & \text{The answer for zero.} \\ f(n+1; y_1, \cdots, y_n) & = & \ldots n+1 \ldots y_1 \ldots y_n \ldots f(n; y_1, \cdots, y_n) \ldots & \text{The recursive call occurs on the recursive part.} \end{array}$$

The examples of length and concatenation of lists from Chapter 1 are examples of structural recursion. To further illustrate these ideas, we consider the following examples.

2.2.1 An Extended Example: Reversal

Consider the following problem.

Problem 1. *Given a list ℓ, what is the reversal of ℓ?*

Example

Given that $\ell = [1,2,3]$, the reversal is $[3,2,1]$.

Let r be the name of the reversal function. To fill in the details, we ask two questions. First, what is the reversal of the empty list? Second, if r works on the tail of a list (say $r([2,3]) = [3,2]$) what do we do to get the answer for the whole thing? For example, how do we go from $[3,2]$ to $[3,2,1]$?

The answer to the first question is that the reversal of the empty list is the empty list. The answer to the second question is that we concatenate the singleton list containing the first element to the end of the result on the tail. These answers lead to the following definition.

$$\begin{array}{lcl} r([]) & = & [] \\ r(x :: xs) & = & r(xs) + [x] \end{array}$$

2.2.2 An Extended Example: Multiplication

Consider the following problem.

Problem 2. *Given a two natural numbers m and n, what is their product $m \times n$?*

[2] This abstract form can be reified as code. A simplified version of this function is called foldRight.

[3] Structural recursion on natural numbers is also called *primitive recursion.*

Example

Given 3 and 4 , $3 \times 4 = 12$.

Let $\odot$ be the name of the multiplication function. To fill in the details, we ask two questions. First, what is the product of zero times a number? Second, if we can compute the product with the predecessor, how do we do to get the answer for the whole thing? For example, how do we go from $2 \times 4 = 8$ to $(2+1) \times 4 = 12$?

The answer to the first question is that zero times any number is zero. The answer to the second question is that we use the distributive property to see that we need only add n to the product with the predecessor. These answers lead to the following definition.

$$\begin{array}{lcl} 0 \odot n & = & 0 \\ (m+1) \odot n & = & (m \odot n) + n \end{array}$$

2.3 Divide and Conquer

Divide and conquer is a recursive approach to algorithm design, which is implicitly non-structural. Relaxing that restriction allows for the design of more performant algorithms, but finding them becomes an art. *Divide* involves breaking the input in order to obtain one or more subproblems. Although the breaking is non-structural and hence involves no rule, a common tactic is to break the input into roughly equal sized pieces. *Conquer* is simply solving the subproblems recursively. Like all forms of recursion, often there is also a *combine* that puts the solutions to the subproblems together to obtain the solution for the original problem. To illustrate this idea we consider the following example.

2.3.1 An Extended Example: Raising to a Power

Consider the following problem.

Problem 3. *Given a number b and a number $n \in \mathbb{N}$, what is the number b^n?*

Coupled with algebraic facts, structural recursion yields a simple solution.

$$\begin{array}{lcl} b^0 & = & 1 \\ b^{n+1} & = & b^n \times b \end{array}$$

But we want to do better than that, so we consider dividing n by 2. That is possible only if n is even, so our divide and conquer approach leads us to consider the cases for n being even and n being odd (the only other choice). Recall that if n is even then $n = 2k$ for some k, and if n is odd then $n = 2k+1$ for some k. Once we have come to this case analysis, algebraic facts yield the following significantly more efficient algorithm.

$$\begin{array}{lcl} b^0 & = & 1 \\ b^{2k} & = & (b^2)^k \\ b^{2k+1} & = & (b^2)^k \times b \end{array}$$

2.4 Tail Recursion

Recursive algorithms can consume space merely because they are recursive, but not all do. Here we consider a special case for recursion that does not consume space: *tail recursion*. To illustrate this idea

we consider the following example.

2.4.1 An Extended Example: Multiplication Revisited

Recall the multiplication algorithm from section 2.2.2. Let us use equational reasoning to examine how the computation of $3 \odot 4$ unfolds.

$$\begin{aligned}
3 \odot 4 &= (2 \odot 4) + 4 \\
&= ((1 \odot 4) + 4) + 4 \\
&= (((0 \odot 4) + 4) + 4) + 4 \\
&= ((0 + 4) + 4) + 4 \\
&= (4 + 4) + 4 \\
&= 8 + 4 \\
&= 12
\end{aligned}$$

Notice that each call to $\odot$ except the first occurs in an evaluation context. Further this evaluation context expands as the computation proceeds: the lines get longer until hitting a maximum and then shrink down to the final answer. On the page, we see that the longer line takes up more space. This space usage is real; in a practical implementation, the larger the input, the more space on the run-time stack is needed.

But not all calls exhibit this behavior of expanding the evaluation context. Calls that do not cause the evaluation context to expand are called *tail calls*; when such a call is a recursive call, it is called *tail recursive* and the recursion that takes place is called *tail recursion*. Tail calls can be identified in function definitions syntactically since the lack of context in the definition entails that evaluation will not dynamically expand the evaluation context. Consider the following definition of the function prodAccum.

$$\begin{aligned}
\text{prodAccum}(0, n; a) &= a \\
\text{prodAccum}(m+1, n; a) &= \text{prodAccum}(m, n; n+a)
\end{aligned}$$

Notice that in the definition of the function prodAccum the recursive function call on the right-hand side occurs in no context — it is a tail recursive call. When we express our algorithm using tail recursion we reduce or eliminate the need for stack space. The following example illustrates that the dynamic evaluation context does not expand.

$$\begin{aligned}
\text{prodAccum}(3,4;0) &= \text{prodAccum}(2,4;4+0) \\
&= \text{prodAccum}(2,4;4) \\
&= \text{prodAccum}(1,4;4+4) \\
&= \text{prodAccum}(1,4;8) \\
&= \text{prodAccum}(0,4;4+8) \\
&= \text{prodAccum}(0,4;12) \\
&= 12
\end{aligned}$$

This function can be expressed using the imperative pseudo-code as follows.

def PRODACCUM$(m, n; a)$:

1. **if** $m = 0$:
2. **return** a
3. **else** :
4. **return** PRODACCUM$(m-1, n; n+a)$

In an imperative setting we observe that there is no reason to "return." The tail call can be viewed as a **goto** statement. Therefore, the recursive code can be changed to the following iterative code with assignment to update the variables.

def PRODACCUM$(m, n; a)$:

1. **label** *top*
2. **if** $m = 0$:
3. **return** a
4. **else** :
5. $m \leftarrow m - 1$
6. $a \leftarrow n + a$
7. **goto** *top*

Since most modern programming languages do not have a **goto** statement, we see that we can simulate it with a **while** loop as follows.

def PRODACCUM$(m, n; a)$:

1. **while** $\top$:
2. **if** $m = 0$:
3. **return** a
4. **else** :
5. $m \leftarrow m - 1$
6. $a \leftarrow n + a$

Often in imperative code it is convenient to initialize the accumulator in the same procedure.

def PRODACCUM(m, n) :

1. $a \leftarrow 0$
2. **while** $\top$:
3. **if** $m = 0$:
4. **return** a
5. **else** :
6. $m \leftarrow m - 1$
7. $a \leftarrow n + a$

Finally, we can make the code look more natural by moving the test from the **if** statement to the **while**.

def PRODACCUM(m, n) :

1. $a \leftarrow 0$
2. **while** $m \neq 0$:
3. $m \leftarrow m - 1$
4. $a \leftarrow n + a$
5. **return** a

2.5 Dynamic Programming

Multiple recursive procedure calls can lead to exponential time performance. Sometimes all those procedure calls are necessary and sometimes they are not. The computer cannot tell the difference, so it does all the work in either case. If we want to compute a recurrence more efficiently, we need to organize the computation differently. *Dynamic programming* is an organizational strategy where the computation occurs in a tabular bottom-up fashion; this strategy leads to more efficient programs when not all of the procedure calls are necessary.

2.5.1 An Extended Example: Powers of Two

Consider the following problem.

Problem 4. *Given a number $n \in \mathbb{N}$, what is the number 2^n?*

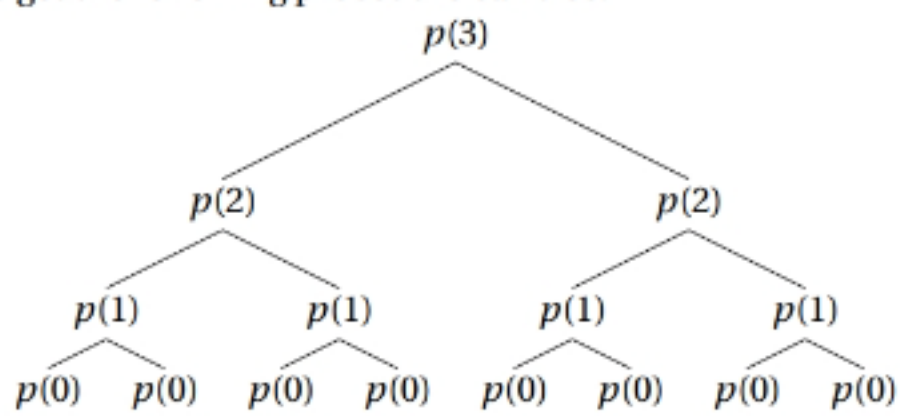
The following algorithm computes 2^n.

$$\begin{aligned} p(0) &= 1 \\ p(n+1) &= p(n) + p(n) \end{aligned}$$

We can write that using the imperative pseudo-code as follows.

def $p(n)$:
1. **if** $n = 0$:
2. **return** 1
3. **else** :
4. **return** $p(n-1) + p(n-1)$

When computing $p(3)$, we get the following procedure call tree.



We observe that there are 15 procedure calls when $n = 3$. If that seems like a lot, it is. In general, the number of procedure calls is $2^{n+1} - 1$.

Now it is clear that the function p will have particularly poor performance; it takes exponential time in the magnitude of n. But it need not be the case that it takes so much time to compute 2^n. From the tree it is apparent that many computations are being repeated.

We will still use the recursive formula, but instead of using procedure calls, we will use array accesses. Since we will call the array p, we change the name of the function. Then it is merely a matter of creating the array, introducing a loop that iterates over k, changing the name from n to k and changing calls to array accesses in the body of the function, replacing "**return**" with "$p[k] \leftarrow$", and finally returning $p[n]$

at the end. A finite portion of the function p will be stored in an array starting with the base case; the domain of p will be used to index into the array. The pseudo-code to compute p follows.

def POW(n) :

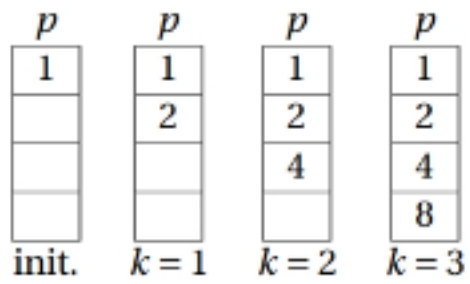
```
1. p ← array(n+1)
2. for k ← 0 to n :
3.     if k = 0 :
4.         p[k] ← 1
5.     else :
6.         p[k] ← p[k−1] + p[k−1]
7. return p[n]
```

In the pseudo-code above, it is clear when $k = 0$. We can move that case out of the loop to improve the code. This variation is still considered dynamic programming, and is, in fact, typical.

def POW(n) :

```
1. p ← array(n+1)
2. p[0] ← 1
3. for k ← 1 to n :
4.     p[k] ← p[k−1] + p[k−1]
5. return p[n]
```

The performance is quite different; instead of $2^{n+1} - 1$ procedure calls, we have $n+1$ array updates. The diagram below illustrates the algorithm's behavior.



2.5.2 Issues and Alternatives

The reason it was straightforward to replace procedure calls with array accesses is because the domain of the recurrence is the natural numbers (which coincides with the set of indices). When the domain of a recurrence is not the natural numbers it is often less obvious how to apply dynamic programming. A common case is when the domain is some set of sequences. We can turn such a problem back into a dynamic programming problem by viewing the sequence as an array and the subsequences as slices. The slices are then characterized by indices and we are back to natural numbers.

Dynamic programming can be useful for solving optimization problems when there is an inefficient recursive solution. But for a recursive solution to work, it must be the case that the solution can be constructed from the solutions of the parts. While it is often the case that this is possible for an optimization problem, it is not always the case. If it is possible, we say the problem has the property of *optimal substructure*.

Instead of taking the bottom-up approach of dynamic programming, it is possible to take a top-down approach. The recurrence is augmented so that the answers are recorded in a data structure when they are computed the first time; subsequent calls find the result in the data structure instead of recomputing them from scratch. This approach to dealing with the problem of redundant procedure calls is called *memoization*. In a high-level language, it is often easier to implement than dynamic programming, but it is typically less performant.

2.6 Greedy Algorithms

An optimization algorithm is said to be *greedy* if it computes this optimum result by computing the local optimum of another result so that the algorithm only involves a single recursive call. To illustrate this idea we consider the following example.

2.6.1 An Extended Example: Minimum Change

Consider the following problem.

Problem 5. *Given an amount of money a and a list of denominations $ds = [d_1, \cdots, d_n]$, what is the minimum (number of "coins") C such that $a = \sum_{i=1}^{n} c_i \times d_i$ and $C = \sum_{i=1}^{n} c_i$?*

Example

Given that $a = 75$ and $ds = [25, 10, 5, 1]$, the minimum number is 3 since $3 \times 25 = 75$.

Example

Given that $a = 74$ and $ds = [25, 10, 5, 1]$, the minimum number is 8 since $2 \times 25 + 2 \times 10 + 4 \times 1 = 74$.

Since the key design technique is recursion, we first attempt a recursive solution to this problem. In fact, the minimum change problem does have the property of optimal sub-structure and we proceed as follows. For a given denomination, if it is possible to use it, we consider two cases: we do use it, or we do not use it. If we use it, the amount a goes down. If we do not, the denominations list becomes shorter. The answer is then the minimum number of coins involved between these possibilities.

$$
\begin{array}{lcl}
minChange(0, ds) & = & 0 \\
minChange(a, []) & = & \text{Failure} \\
minChange(a, d :: ds) & = & \begin{cases} minChange(a, ds) & \text{if } d > a \\ \min(1 \oplus minChange(a - d, d :: ds), minChange(a, ds)) & \text{otherwise} \end{cases}
\end{array}
$$

The $\oplus$ operation is understood here as an extension of regular addition so that it also works on Failure; Failure plus anything is Failure. The function min is also extended to work with Failure; numbers are understood as smaller than Failure.

This approach works, but it can be slow because there are two calls to *minChange*. In an effort to develop a more efficient algorithm, we will use a greedy approach and locally optimize another result: we will find the largest denomination. Then we will see how many coins of that denomination can be used. It appears that the answer is then the sum of that number and the minimum number of coins with

the other denominations. (In fact, this is the algorithm that is typically used.) To simplify finding the largest denomination, we require that the denominations list is sorted in descending order.

$$
\begin{array}{lcl}
\mathit{greedyMinChange}(0, ds) & = & 0 \\
\mathit{greedyMinChange}(a, []) & = & \text{Failure} \\
\mathit{greedyMinChange}(a, d :: ds) & = & \begin{cases} \mathit{greedyMinChange}(a, ds) & \text{if } d > a \\ q \oplus \mathit{greedyMinChange}(r, ds) & \text{otherwise} \end{cases} \\
 & & \text{where } q = \text{quotient}(a, d), r = \text{remainder}(a, d)
\end{array}
$$

Indeed, this algorithm is much faster since there is only one function call. Unfortunately, it doesn't always work! Consider the following examples. Observe that $\mathit{greedyMinChange}(18, [11, 9, 3])$ = Failure. (The correct answer is 2.) Also observe that $\mathit{greedyMinChange}(10, [7, 5, 1]) = 4$. (Here too the correct answer is 2.)

It does work for the traditional denominations ($[25, 10, 5, 1]$). It also works for certain other classes of denominations. In general, the key question for a greedy algorithm is does it work, or does it work for the class of inputs that we're interested in? It is necessary to answer this question before using a greedy algorithm. One theoretical approach to answering this question involves framing the objects of interest in terms of the mathematical *weighted matroid* objects.[4]

[4]There is a greedy algorithm to find the optimal subset of the matroid.

Chapter 3

Analysis Overview**

3.1 Perspectives on Analysis

Suppose we have written an algorithm to solve a particular problem. When analyzing an algorithm, we will focus on correctness (i.e., for all possible inputs our algorithm halts with the right answer) and resource requirements. In particular, the resources we typically care about are time and space as a function of the size of the input.

When analyzing the resource requirements, there are different points of view. One of these perspectives concerns the nature of the characterization of resource requirements. Another concerns whether the analysis is empirical or theoretical. And if it is theoretical whether it is exact or approximate.

With respect to the nature of the characterization of resource requirements, we may want to consider how bad the performance can get and conduct a *worst case* analysis. We may want to know how good it can get and conduct a *best case* analysis. Or we may want to know what we typically get and conduct an *average case* analysis; this analysis typically requires assumptions concerning the distribution of the inputs. Another important analysis is an *amortized* analysis. Usually, we will focus on the worst case analysis; we are motivated to do other analyses if the worst case does not seem to jibe with our experience.

If our analysis is to be empirical, first it is necessary to implement the algorithm and instrument it. We can investigate the resource requirements of an algorithm by setting up a test suite. We then run the test suite on the instrumented implementation and collect data. At that point it is necessary to analyze the data. Although empirical analysis is important, this book emphasizes theoretical analysis.

If our analysis is to be theoretical, we must derive a *complexity measure* from the algorithm, a mathematical function that maps the size of the input to the amount of resources.[1] Technically, we need a formal model of computation for a detailed analysis. For a less detailed analysis, we can get away with being vague about our model of computation; in fact, being vague in this way is typical and this book will do the same. Nevertheless, it is possible to be completely exact even without a model of computation by counting a particular operation or operations.

[1] A complexity measure is not always a useful way to understand the performance of an algorithm. For a counterexample, consider an interpreter.

3.2 Example: Insertion Sort

As an example, consider the insertion sort algorithm below.

$$
\begin{aligned}
i(x,[]) &= [x] \\
i(x, y :: ys) &= \begin{cases} x :: y :: ys & \text{if } x \leq y \\ y :: i(x, ys) & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
iSort([]) &= [] \\
iSort(x :: xs) &= i(x, iSort(xs))
\end{aligned}
$$

We would like to get a sense of the time this algorithm takes to sort a sequence of size n. To do so, we will count the number of times the function i is called.

Example: An Ascending Sequence

Consider the sequence $[1,2,3,4]$. Let us simply count how many times the function i is called. Recall that $1 :: [2,3,4] = [1,2,3,4]$.

$$
\begin{aligned}
iSort([1,2,3,4]) &= i(1, iSort([2,3,4])) \\
&= i(1, i(2, iSort([3,4]))) \\
&= i(1, i(2, i(3, iSort([4])))) \\
&= i(1, i(2, i(3, i(4, iSort([])))))) \\
&= i(1, i(2, i(3, \mathbf{i}(4, [])))) \\
&= i(1, i(2, \mathbf{i}(3, [4]))) \\
&= i(1, \mathbf{i}(2, [3,4])) \\
&= \mathbf{i}(1, [2,3,4]) \\
&= [1,2,3,4]
\end{aligned}
$$

To count the number of distinct calls, we can count a call before it disappears. These calls have been marked in bold to make them easier to identify. If we count carefully, we find the i is called four times.

Example: A Descending Sequence

Consider the sequence $[4,3,2,1]$. Again, let us simply count how many times the function i is called.

$$
\begin{aligned}
iSort([4,3,2,1]) &= i(4, iSort([3,2,1])) \\
&= i(4, i(3, iSort([2,1]))) \\
&= i(4, i(3, i(2, iSort([1])))) \\
&= i(4, i(3, i(2, i(1, iSort([])))))) \\
&= i(4, i(3, i(2, \mathbf{i}(1, [])))) \\
&= i(4, i(3, \mathbf{i}(2, [1]))) \\
&= i(4, i(3, 1 :: \mathbf{i}(2, []))) \\
&= i(4, i(3, 1 :: [2])) \\
&= i(4, \mathbf{i}(3, [1,2])) \\
&= i(4, 1 :: \mathbf{i}(3, [2]))
\end{aligned}
$$

$$
\begin{aligned}
&= i(4, 1 :: 2 :: \mathbf{i}(3, [])) \\
&= i(4, 1 :: 2 :: [3]) \\
&= \mathbf{i}(4, [1,2,3]) \\
&= 1 :: \mathbf{i}(4, [2,3]) \\
&= 1 :: 2 :: \mathbf{i}(4, [3]) \\
&= 1 :: 2 :: 3 :: \mathbf{i}(4, []) \\
&= 1 :: 2 :: 3 :: [4] \\
&= [1,2,3,4]
\end{aligned}
$$

If we count carefully, this time we find the i is called ten times.

A Theoretical Approach

For the theoretical approach, we will consider functions that map the size of the input, n, to the number of times i is called. Let the function $T_{iSort}(n)$ be the number of times i is called from s, and let the function $T_i(n)$ be the number of times i is called from i not including that original call.
Now, recall the sorting algorithm.

$$
\begin{aligned}
iSort([]) &= [] \\
iSort(x :: xs) &= i(x, iSort(xs))
\end{aligned}
$$

Since *iSort* is recursive, we can recursively characterize $T_{iSort}(n)$.

$$
\begin{aligned}
T_{iSort}(0) &= 0 \\
T_{iSort}(n+1) &= (1 + T_i(n)) + T_{iSort}(n)
\end{aligned}
$$

Now, recall the insertion algorithm.

$$
\begin{aligned}
i(x, []) &= [x] \\
i(x, y :: ys) &= x :: y :: ys, \text{ if } x \leq y \\
i(x, y :: ys) &= y :: i(x, ys)
\end{aligned}
$$

What about $T_i(n)$? The analysis now becomes more complicated. We always have $T_i(0) = 0$ since there are no calls to i on the right-hand side. But there are two cases for a non-empty sequence. If we always find that $x \leq y$, then $T_i(n+1) = 0$. On the other hand, if we always find that $x > y$, then $T_i(n+1) = 1 + T_i(n)$. Observe that if we decide that $T_i(n+1) = 0$, we will get a best case analysis, and if we decide that $T_i(n+1) = 1 + T_i(n)$, we will get a worst case analysis.
Let us start with a best case analysis, and consider the following recurrence for $T_i(n)$.

$$
\begin{aligned}
T_i(0) &= 0 \\
T_i(n+1) &= 0
\end{aligned}
$$

Clearly it follows that $T_i(n) = 0$. Hence we get the following recurrence for $T_{iSort}(n)$.

$$
\begin{aligned}
T_{iSort}(0) &= 0 \\
T_{iSort}(n+1) &= 1 + T_{iSort}(n)
\end{aligned}
$$

It turns out that the solution to the recurrence above is $T_{iSort}(n) = n$. In particular, $T_{iSort}(4) = 4$, which is what we found for the ascending sequence.
Now, let us perform a worst case analysis, and consider the following recurrence for $T_i(n)$.

$$\begin{aligned} T_i(0) &= 0 \\ T_i(n+1) &= 1 + T_i(n) \end{aligned}$$

We see from our experience above that the solution to this recurrence is $T_i(n) = n$. Hence we get the following recurrence for $T_{iSort}(n)$.

$$\begin{aligned} T_{iSort}(0) &= 0 \\ T_{iSort}(n+1) &= (n+1) + T_{iSort}(n) \end{aligned}$$

It turns out that the solution to the recurrence above is $T_{iSort}(n) = n^2/2 + n/2$. In particular, $T_{iSort}(4) = 16/2 + 4/2 = 8 + 2 = 10$, which is what we found for the descending sequence.

3.3 Example: Factorial

As an example, consider the factorial algorithm.

$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1) \times n! \end{aligned}$$

We would like to get a sense of the space this algorithm takes to compute $n!$. To do so, we will count the maximum number of multiplication signs written on a line.

Example: Computing Factorial

Consider the computation of 3!.

$$\begin{aligned} 3! &= 3 \times 2! \\ &= 3 \times (2 \times 1!) \\ &= 3 \times (2 \times (1 \times 0!)) \\ &= 3 \times (2 \times (1 \times 1)) \\ &= 3 \times (2 \times 1) \\ &= 3 \times 2 \\ &= 6 \end{aligned}$$

Notice that the maximum number of multiplication signs on a line is 3.

A Theoretical Approach

Now we will write down a recurrence that characterizes the space usage of $n!$. When $n = 0$, there are no multiplication signs. When $n > 0$, the maximum number of multiplication signs is one more than the maximum number for $n - 1$.
Thus we get the following recurrence.

$$\begin{aligned} S(0) &= 0 \\ S(n+1) &= 1 + S(n) \end{aligned}$$

It turns out that the solution to the recurrence above is $S(n) = n$.

An Alternative Formulation

Now consider the following alternative algorithm for computing $n!$.

$$\begin{aligned} f(0;a) &= a \\ f(n+1;a) &= f(n;(n+1)\times a) \end{aligned}$$

The significance of this alternative algorithm for computing $n!$ is that it requires less space for the computation since it is tail recursive. But clearly this function definition looks different from the definition of $n!$, so we are concerned with whether or not f is correct. By correctness, we mean that $f(n;1) = n!$.

The correctness of f is tightly linked to the invariant $f(n;a) = n! \times a$. If this invariant holds, we have the following correctness theorem.

Theorem 1. *For any $n \in \mathbb{N}$, $f(n;1) = n!$.*

Proof. $f(n;1) = n! \times 1 = n!$ □

We can establish the invariant in two ways: we can derive the algorithm from the invariant or we can use induction to prove the algorithm satisfies the invariant. In this book, we will typically take the first approach; here we will do both.
The derivation involves a case analysis. First we derive the base case.

$$\begin{aligned} f(0;a) &= 0! \times a \\ &= 1 \times a \\ &= a \end{aligned}$$

Then we derive the recursive case.

$$\begin{aligned} f(n+1;a) &= (n+1)! \times a \\ &= ((n+1) \times n!) \times a \\ &= n! \times ((n+1) \times a) \\ &= f(n;(n+1) \times a) \end{aligned}$$

The alternative inductive proof follows.

Lemma 1. *For any $n \in \mathbb{N}$, for any $a \in \mathbb{N}$, $f(n;a) = n! \times a$.*

Proof. By mathematical induction on n.

- Observe $f(0;a) = a = 1 \times a = 0! \times a$.
- Assume $f(k;a) = k! \times a$.

$$\begin{aligned} f(k+1;a) &= f(k;(k+1) \times a) \\ &= k! \times ((k+1) \times a) \\ &= (k! \times (k+1)) \times a \\ &= ((k+1) \times k!) \times a \\ &= (k+1)! \times a \end{aligned}$$

□

In the imperative pseudo-code, algorithm f takes the following form.

def $f(n)$:

1. $a \leftarrow 1$
2. **while** $n \neq 0$:
3. $\quad a \leftarrow n \times a$
4. $\quad n \leftarrow n-1$
5. **return** a

Example: Computing the Alternative Factorial

Consider the computation of $f(3;1)$.

$$\begin{aligned} f(3;1) &= f(2;3\times 1)\\ &= f(2;3)\\ &= f(1;2\times 3)\\ &= f(1;6)\\ &= f(0;1\times 6)\\ &= f(0;6)\\ &= 6 \end{aligned}$$

Notice that no line has more than one multiplication sign.

A Theoretical Approach Continued

When $n = 0$, here too there are no multiplication signs. When $n > 0$, there is a multiplication sign, but it is immediately evaluated, so the maximum is the maximum of 1 and the maximum on a line when computing the result for $n-1$. Thus we get the following recurrence.

$$\begin{aligned} S_f(0) &= 0\\ S_f(n+1) &= \max(1, S_f(n)) \end{aligned}$$

It is then clear that the space needed is merely a constant. $S_f(n) = \begin{cases} 0 & \text{if } n = 0\\ 1 & \text{otherwise} \end{cases}$

Chapter 4

Asymptotics**

4.1 Introduction to Asymptotics

In the previous chapter, we saw examples of complexity functions characterizing both time and space for particular algorithms. For example, we saw that the time complexity of insertion sort is $T_{iSort}(n) = n^2/2+n/2$. Sometimes, this amount of detail (or more) feels overwhelming. It would be simpler to merely write $T_{iSort}(n) = n^2$. Such a simplification is also useful because it characterizes the *asymptotic* behavior of $T_{iSort}(n)$, or the behavior when n is large; we are less concerned about the behavior when n is small because most algorithms are fast on small inputs. Unfortunately, it is simply not true that $T_{iSort}(n) = n^2$. A statement that is true is that $T_{iSort}(n) \in \Theta(n^2)$. We define this notation next.

4.2 Formal Definitions

We formally characterize asymptotic function classes below.

Definition 2. *Given $f : D \to \mathbb{R}$, $g : D \to \mathbb{R}$ where $D \subseteq \mathbb{R}$, $f(x) \in O(g(x))$ if there exists $N \in \mathbb{N}$ and $c \in \mathbb{R}^+$ such that for any $x \in D$, $x \geq N$ implies $|f(x)| \leq c|g(x)|$.*

The assumption that $x \geq N$ gets at the notion that we are concerned about the behavior when x is large. The inequality entails that we are bounding f from above. The hypothesis of a constant c allows us to eliminate constant factors. Further, the combination of a constant factor and a bound entails that we can often eliminate sums as well.

Definition 3. *Given $f : D \to \mathbb{R}$, $g : D \to \mathbb{R}$ where $D \subseteq \mathbb{R}$, $f(x) \in \Omega(g(x))$ if there exists $N \in \mathbb{N}$ and $c \in \mathbb{R}^+$ such that for any $x \in D$, $x \geq N$ implies $|f(x)| \geq c|g(x)|$.*

The definition for Ω is similar to the previous definition, but the bound is reversed.

Definition 4. *Given $f : D \to \mathbb{R}$, $g : D \to \mathbb{R}$ where $D \subseteq \mathbb{R}$, $f(x) \in \Theta(g(x))$ if $f(x) \in O(g(x))$ and $f(x) \in \Omega(g(x))$.*

Notation 1. *It is customary to write $f(x) = O(g(x))$ to mean $f(x) \in O(g(x))$, and similarly for Ω and Θ.*

The definition of Θ requires that both bounds hold; this property is similar to equality.[1] Indeed, a reasonable intuition is that O is like $\leq$, Ω is like $\geq$, and Θ is like $=$. One might then wonder why we bother with this notation rather than using an infix relational notation. The answer is that it is convenient to use the asymptotic notation (O, etc.) within an expression.

Notation 2. *We can write expressions involving asymptotic notation (O, etc.) by understanding functions as singleton sets and extending mathematical operations to sets in the usual way. For example,* $f(x) + O(g(x))$ *means* $\{f(x)\} + O(g(x))$*, and the addition of sets* $S + T$ *means* $\{s + t \mid s \in S, t \in T\}$*. We can also say* $f(x) + O(g(x)) = O(h(x))$*, where here the equal sign is understood as indicating the subset relation (i.e.,* $\{f(x)\} + O(g(x)) \subseteq O(h(x))$*).*

4.3 Proofs Using the Definitions

We will now establish asymptotic bounds for a particular function using just the definitions. The nature of these proofs is then to satisfy the conditions of the definitions. We need to show that N and c exist. We prove this by giving examples. Of course, it is necessary to show that the property involving N and c is satisfied for the given example.

Lemma 2. *If* $f(x) = x^2/2 + x/2$ *then* $f(x) \in O(x^2)$.

Proof.
Consider $N = 1$, $c = 1$.
Suppose $x \geq 1$.

$$\begin{aligned}
1 \leq x &\Rightarrow x \leq x^2 \\
&\Rightarrow x/2 \leq x^2/2 \\
&\Rightarrow x^2/2 + x/2 \leq x^2/2 + x^2/2 \\
&\Rightarrow x^2/2 + x/2 \leq x^2 \\
&\Rightarrow |x^2/2 + x/2| \leq |x^2| \text{ since the results are positive} \\
&\Rightarrow |x^2/2 + x/2| \leq 1 \cdot |x^2|
\end{aligned}$$

□

Lemma 3. *If* $f(x) = x^2/2 + x/2$ *then* $f(x) \in \Omega(x^2)$.

Proof.
Consider $N = 0$, $c = 1/2$.
Suppose $x \geq 0$.

$$\begin{aligned}
x \geq 0 &\Rightarrow x/2 \geq 0 \\
&\Rightarrow x^2/2 + x/2 \geq x^2/2 \\
&\Rightarrow |x^2/2 + x/2| \geq (1/2)|x^2| \text{ since the results are non-negative}
\end{aligned}$$

□

Corollary 1. *If* $f(x) = x^2/2 + x/2$ *then* $f(x) \in \Theta(x^2)$.

4.4 Properties of Asymptotic Notation

Here we start by stating some facts concerning asymptotic notation. These facts are useful for reasoning about asymptotic notation at a level higher than the level of the definitions.

[1] It is possible to define an equivalence relation using Θ.

4.4.1 Facts about O

These facts involve O. Please be aware that the "equations" are one way.

Fact 1. *If* $k \leq k'$ *then* $x^k = O(x^{k'})$.

Fact 2. $O(f(x)) + O(f(x)) = O(f(x))$.

Fact 3. $O(f(x)) + O(g(x)) = O(|f(x)| + |g(x)|)$.

Fact 4. $c \cdot O(f(x)) = O(f(x))$.

4.4.2 Facts about Ω

These facts involve Ω. Please be aware that the "equations" are one way.

Fact 5. *If* $k \geq k'$ *then* $x^k = \Omega(x^{k'})$.

Fact 6. *If* $a > 0$ *then* $ax^k + O(x^{k-1}) = \Omega(x^k)$.

Fact 7. *If* $c \neq 0$ *then* $c \cdot \Omega(f(x)) = \Omega(f(x))$.

4.4.3 Example Using the Facts

From the definition it was not too hard to prove that a particular function is bounded by a monomial. Here we will use the facts to prove results for arbitrary polynomial functions.

Lemma 4. *If* $f(x) = a_d x^d + \cdots + a_1 x + a_0$ *then* $f(x) \in O(x^d)$.

Proof.

$$\begin{aligned} a_d x^d + \cdots + a_1 x + a_0 &= a_d O(x^d) + \cdots + a_1 O(x^d) + a_0 O(x^d) \text{ by Fact 1} \\ &= O(x^d) + \cdots + O(x^d) + O(x^d) \text{ by Fact 4} \\ &= O(x^d) \text{ by Fact 2} \end{aligned}$$

□

Lemma 5. *If* $f(x) = a_d x^d + \cdots + a_1 x + a_0$ *and* $a_d > 0$ *then* $f(x) \in \Omega(x^d)$.

Proof.

$$\begin{aligned} a_d x^d + \cdots + a_1 x + a_0 &= a_d x^d + O(x^{d-1}) \text{ by Lemma 4} \\ &= \Omega(x^d) \text{ by Fact 6} \end{aligned}$$

□

Corollary 2. *If* $f(x) = a_d x^d + \cdots + a_1 x + a_0$ *and* $a_d > 0$ *then* $f(x) \in \Theta(x^d)$.

4.4.4 Some Proofs

To get a feel for how to prove the facts above, we will see some proofs for a couple of them.

Theorem 2. $k \cdot O(g(x)) \subseteq O(g(x))$.

Proof.
Without loss of generality, let D be the domain of g.
Suppose $k \cdot f(x) \in k \cdot O(g(x))$.
Then there exists $N_f \in \mathbb{N}$ and $c_f \in \mathbb{R}^+$ such that for any $x \in D$, $x \geq N_f$ implies $|f(x)| \leq c_f|g(x)|$.
(We want to show that $k \cdot f(x) \in O(g(x))$.)
Consider $N = N_f$ and $c = |k|c_f$.
Suppose $x \geq N$.
Since $x \geq N_f$, we have that $|f(x)| \leq c_f|g(x)|$.

$$\begin{aligned} |f(x)| \leq c_f|g(x)| \quad &\Rightarrow \quad |k| \cdot |f(x)| \leq |k|(c_f|g(x)|) \\ &\Rightarrow \quad |k \cdot f(x)| \leq (|k|c_f)|g(x)| \end{aligned}$$

□

Theorem 3. *If $a > 0$ then $ax^k + O(x^{k-1}) \subseteq \Omega(x^k)$.*

Proof.
Without loss of generality, let D be the domain.
Suppose $ax^k + f(x) \in ax^k + O(x^{k-1})$.
Then there exists $N_f \in \mathbb{N}$ and $c_f \in \mathbb{R}^+$ such that for any $x \in D$, $x \geq N_f$ implies $|f(x)| \leq c_f|x^{k-1}|$. Note that $-c_f|x^{k-1}| \leq f(x)$.
(We want to show that $ax^k + f(x) \in \Omega(x^k)$.)
Consider $c < a$ and $N \geq \frac{c_f}{a-c}$.
Suppose $x \geq N$.
Since x is positive, we have that x^k is positive, x^{k-1} is positive, and $f(x) \geq -c_f x^{k-1}$ which entails $ax^k + f(x) \geq ax^k - c_f x^{k-1}$.

$$\begin{aligned} x \geq \tfrac{c_f}{a-c} \quad &\Rightarrow \quad x^k \geq (\tfrac{c_f}{a-c})x^{k-1} \\ &\Rightarrow \quad (a-c)x^k \geq c_f x^{k-1} \\ &\Rightarrow \quad ax^k - c_f x^{k-1} \geq cx^k \\ &\Rightarrow \quad ax^k + f(x) \geq cx^k \\ &\Rightarrow \quad |ax^k + f(x)| \geq c|x^k| \end{aligned}$$

□

4.5 Additional Properties of Asymptotic Notation

Fact 8. *If* $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$ *then* $f(x) = O(g(x))$.

Fact 9. *If* $\lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty$ *then* $f(x) = \Omega(g(x))$.

Fact 10. *If* $\lim_{x\to\infty} \frac{f(x)}{g(x)} = c$ *and* $c \neq 0$ *is a constant then* $f(x) = \Theta(g(x))$.

Theorem 4. *For any $b > 1$, if $f(x) = \log_b(x)$ then $f(x) = \Theta(\lg(x))$.*

Proof. We will prove equality which entails both inequalities.
Consider $N = 1$ and $c = 1/\lg(b)$.
Suppose $x \geq N$.

$$\begin{aligned} \log_b(x) = \lg(x)/\lg(b) \quad &\Rightarrow \quad |\log_b(x)| = |\lg(x)/\lg(b)| \\ &\Rightarrow \quad |\log_b(x)| = (1/\lg(b))|\lg(x)| \end{aligned}$$

□

Fact 11. $\lg(n!) = \Theta(n\lg(n))$

Chapter 5

Sums and Floors**

5.1 Introduction to Sums and Floors

Facts concerning sums and floors are extremely useful when analyzing algorithms. We review first some results about sums and then some results about floors.

5.2 Sums

There is a tremendous amount that could be said about sums. We will not even attempt any sort of broad coverage. Rather we will focus on a few fundamental results. But we will frequently provide multiple arguments to help the concepts stick. We also discuss circumstances in which a sum can be bounded by an integral.

5.2.1 Fundamental Formulas

We start with the classic arithmetic sum.

Theorem 5. *For any* $n \in \mathbb{N}$, $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$.

Example

To make things concrete, suppose that $n = 4$. Then $\sum_{i=0}^{n} i$, the sum of the numbers to n, is $0+1+2+3+4$, which is 10. Now let's use the formula and observe that the result is the same: $\frac{4(4+1)}{2} = 2 \times 5 = 10$.

Proofs

We start with a proof by mathematical induction because induction is such an important technique.

Proof. By mathematical induction on n.

- Observe that $\sum_{i=0}^{0} i = 0 = \frac{0(0+1)}{2}$.
- Assume $\sum_{i=0}^{k} i = \frac{k(k+1)}{2}$.

$$\begin{aligned}\sum_{i=0}^{k+1} i &= (k+1) + \sum_{i=0}^{k} i \\ &= (k+1) + \frac{k(k+1)}{2} \\ &= \frac{k(k+1)+2(k+1)}{2} \\ &= \frac{(k+1)(k+2)}{2}\end{aligned}$$

□

Here is another proof.

Proof. Using the fact that addition is commutative.
Let $S = 1 + \cdots + n$.
Then $S = n + \cdots + 1$.
Add S to itself.

S	$=$	1	$+$	2	$+$	$\cdots$	$+$	$n-1$	$+$	n
S	$=$	n	$+$	$n-1$	$+$	$\cdots$	$+$	2	$+$	1
$2S$	$=$	$n+1$	$+$	$n+1$	$+$	$\cdots$	$+$	$n+1$	$+$	$n+1$

Thus $S = \frac{n(n+1)}{2}$.

□

The Geometric Sum

Next we consider the classic geometric sum.

Theorem 6. *For any $n \in \mathbb{N}$, for any $x \in \mathbb{R}$, if $x \neq 1$ then* $\sum_{m=0}^{n} x^m = \frac{x^{n+1}-1}{x-1}$.

Example

To make things concrete, suppose that $n = 3$ and $x = 2$. Then $\sum_{m=0}^{n} x^m$, the sum of x^m as m goes from 0 to n, is $2^0+2^1+2^2+2^3 = 1+2+4+8$, which is 15. Now let's use the formula and observe that the result is the same: $\frac{2^4-1}{2-1} = \frac{16-1}{1} = 15$

Proofs

Again, we start with a proof by mathematical induction because of the importance of induction.

Proof. By mathematical induction on n.

- Observe that $\sum_{m=0}^{0} x^m = x^0 = 1 = \frac{x-1}{x-1} = \frac{x^{0+1}-1}{x-1}$.
- Assume $\sum_{m=0}^{k} x^m = \frac{x^{k+1}-1}{x-1}$.

$$\begin{aligned}
\sum_{m=0}^{k+1} x^m &= x^{k+1} + \sum_{m=0}^{k} x^m \\
&= x^{k+1} + \frac{x^{k+1}-1}{x-1} \\
&= \frac{x^{k+1}(x-1)+x^{k+1}-1}{x-1} \\
&= \frac{x^{k+2}-x^{k+1}+x^{k+1}-1}{x-1} \\
&= \frac{x^{k+2}-1}{x-1}
\end{aligned}$$

□

Here is another proof.

Proof.
Let $S = x^0 + \cdots + x^n$.
Multiply by x.
Then $xS = x^1 + \cdots + x^{n+1}$.
Subtract them.

$$\begin{array}{rcccccccccc} xS & = & x^{n+1} & + & x^n & + & \cdots & + & x^1 & & \\ S & = & & & x^n & + & \cdots & + & x^1 & + & x^0 \\ \hline xS-S & = & x^{n+1} & & & & & & & - & x^0 \end{array}$$

$$\text{Thus } S = \frac{x^{n+1}-1}{x-1}.$$

□

Another way to understand this result involves picking a particular value for x. Let $x = 10$. For concreteness, let's also look at a particular value for n at the same time. Let $n = 2$.

$$\begin{array}{rcll} \dfrac{x^{n+1}-1}{x-1} & = & \dfrac{10^{n+1}-1}{10-1} & \dfrac{10^3-1}{10-1} \\ & = & \dfrac{\overbrace{10\cdots0}^{n+1}-1}{9} & \dfrac{1000-1}{9} \\ & = & \dfrac{\overbrace{9\cdots9}^{n+1}}{9} & \dfrac{999}{9} \\ & = & \overbrace{1\cdots1}^{n+1} & 111 \\ & = & 10^n + \cdots + 10^0 & 10^2 + 10^1 + 10^0 \\ & = & x^n + \cdots + x^0 & \\ & = & \displaystyle\sum_{m=0}^{n} x^m & \end{array}$$

Related Ideas

A natural mathematical question is "what happens to the geometric sum if we add an infinite number of terms?" If x is too big, say one or more, we can see that the more terms we add the larger the sum gets. We see that the sum is unbounded, and we say it *diverges*. However, if x is small, say a half, it turns out the infinite sum converges to a number.

A natural question for a computer scientist is "when would we ever add up an infinite number of terms?" Indeed, computations do *not* involve adding an infinite number of terms. Nevertheless, if there are a lot of terms, it can be simpler to see what happens when there are an infinite number rather than calculating the exact result.

Corollary 3. *For any $x \in \mathbb{R}$, if $|x| < 1$ then* $\displaystyle\sum_{m=0}^{\infty} x^m = \frac{1}{1-x}$.

Example

Here too we are going to compute the sum in two different ways where we see that we get the same result when we use the formula. However, computing without the formula is more subtle because the sum must be infinite. Consider the following value: $9.\bar{9}$ — that is 9 with an infinite number of 9s following after the decimal point. It turns out that there is another way to write this number. Let $y = 0.\bar{9}$. Then $10y = 9.\bar{9}$. Subtract the two: $10y - y = 9$. Thus $9y = 9$. So $y = 1$ and $9.\bar{9} = 10$.

Now, let's use the corollary. We will express $9.\bar{9}$ as a sum: $9.\bar{9} = 9((\frac{1}{10})^0 + (\frac{1}{10})^1 + (\frac{1}{10})^2 + \cdots)$. By the theorem $(\frac{1}{10})^0 + (\frac{1}{10})^1 + (\frac{1}{10})^2 + \cdots = \frac{1}{1-\frac{1}{10}} = \frac{1}{\frac{9}{10}} = \frac{10}{9}$. And so $9.\bar{9} = 9 \times \frac{10}{9} = 10$.

Proof

The proof hinges on the fact that a fraction when raised to larger and larger powers becomes smaller and smaller.

Proof.

$$\begin{aligned}
\sum_{m=0}^{\infty} x^m &= \lim_{n\to\infty} \sum_{m=0}^{n} x^m \\
&= \lim_{n\to\infty} \frac{x^{n+1}-1}{x-1} \\
&= \lim_{n\to\infty} \frac{1-x^{n+1}}{1-x} \\
&= \frac{1}{1-x}
\end{aligned}$$

□

Another result that is related is the sum of reciprocals. The terms are fractions, but they are all different and no powers are involved. It turns out that taking such a sum to infinity diverges. One way of understanding why is that sums are like integrals, and the corresponding integral diverges. Indeed, we give a name, H_n, to the finite sum which is like a logarithm. We will go into more detail on this point later in this chapter.

Definition 5. *The nth harmonic number is denoted by H_n, where* $H_n = \sum_{k=1}^{n} \frac{1}{k}$.

The Telescoping Sum

The last fundamental sum we consider is the telescoping sum.

Theorem 7. *For any $n \in \mathbb{N}$, for any sequence $y_0, \cdots, y_{n+1}$,* $\sum_{i=0}^{n} y_{i+1} - y_i = y_{n+1} - y_0$.

Examples

For the first example, let $y_i = i$ and let $n = 3$. Then $\sum_{i=0}^{n} y_{i+1} - y_i$, the sum of the differences, is $(4-3) + (3-2) + (2-1) + (1-0) = 1+1+1+1$, which is 4. Now let's use the formula and observe that the result is the same: $y_{3+1} - y_0 = 4 - 0 = 4$.

The next "example" is notation. This notation emphasizes that this theorem is saying something significant.

Notation 3. *Let us return to the generic function notation and let $f(k) = y_k$. Now we define a difference operator on functions: $(\Delta f)(k) = f(k+1) - f(k)$. We also define the following alternative notation for sums: $\sum_0^n f\,\delta k = \sum_{k=0}^{n-1} f(k)$.*

When putting these notations together, the theorem above takes the following form: $\sum_0^n \Delta f\,\delta k = f(n) - f(0)$.

Note the resemblance to the fundamental theorem of calculus: $\int_0^b \frac{df}{dx}\,dx = f(b) - f(0)$.

The last example shows that we can use the telescoping sum result to tackle challenging sums.

Theorem 8. *For any $n \in \mathbb{N}$, if $n > 1$ then $\sum_{i=0}^{n-2} \frac{1}{(i+2)(i+1)} = 1 - \frac{1}{n}$.*

Example

n	$\sum_{i=0}^{n-2} \frac{1}{(i+2)(i+1)}$	$1 - \frac{1}{n}$
2	$\frac{1}{(0+2)(0+1)} = \frac{1}{2}$	$1 - \frac{1}{2} = \frac{1}{2}$
3	$\frac{1}{2} + \frac{1}{(1+2)(1+1)} = \frac{2}{3}$	$1 - \frac{1}{3} = \frac{2}{3}$
4	$\frac{2}{3} + \frac{1}{(2+2)(2+1)} = \frac{3}{4}$	$1 - \frac{1}{4} = \frac{3}{4}$

Proof.

Observe that $\frac{-1}{(i+2)(i+1)} = \frac{1}{i+2} - \frac{1}{i+1}$.

Let $y_i = \frac{1}{i+1}$. Then $y_{i+1} - y_i = \frac{-1}{(i+2)(i+1)}$.

$$\begin{aligned}
\sum_{i=0}^{n-2} \frac{1}{(i+2)(i+1)} &= -\sum_{i=0}^{n-2} y_{i+1} - y_i \\
&= -(y_{n-1} - y_0) \\
&= -(1/n - 1) \\
&= 1 - 1/n
\end{aligned}$$

□

Proofs

Here too we start with a proof by mathematical induction of the telescoping sum theorem because of the importance of induction.

Proof. By mathematical induction on n.

- Observe $\sum_{i=0}^{0} y_{i+1} - y_i = y_{0+1} - y_0$.
- Assume $\sum_{i=0}^{k} y_{i+1} - y_i = y_{k+1} - y_0$.

$$\begin{aligned}\sum_{i=0}^{k+1} y_{i+1} - y_i &= (y_{k+2} - y_{k+1}) + \sum_{i=0}^{k} y_{i+1} - y_i \\ &= (y_{k+2} - y_{k+1}) + (y_{k+1} - y_0) \\ &= y_{k+2} + (-y_{k+1} + y_{k+1}) - y_0 \\ &= y_{k+2} - y_0\end{aligned}$$

□

The idea behind the next proof is the same but without the induction.

Proof.

$$\begin{aligned}\sum_{i=0}^{k+1} y_{i+1} - y_i &= (y_{k+2} - y_{k+1}) + (y_{k+1} - y_k) + \cdots + (y_1 - y_0) \\ &= y_{k+2} + (y_{k+1} - y_{k+1}) + \cdots + (y_1 - y_1) - y_0 \\ &= y_{k+2} + 0 + \cdots + 0 - y_0 \\ &= y_{k+2} - y_0\end{aligned}$$

□

5.2.2 Bounding Sums

Although we have only discussed a few sums, the previous formulas will take us far. Even so, there will be times when we encounter a difficult sum where those formulas will not be enough. Instead of getting an exact formula, we can bound such sums from above and below with integrals.

Initially, it may be surprising to introduce an integral. One might ask, "aren't integrals even harder than sums?" In fact, the short answer is "no." The formula for a sum is invariably more complicated than that for the corresponding integral. And if we remember a little calculus, the integral is often straightforward to compute.

The next two definitions set the stage for the context we wish to consider: monotonically increasing or decreasing functions. When studying computation we typically find that some resource is steadily going up (or sometimes steadily going down). We formalize these notions below.

Definition 6. *Given an interval $I = [a..b]$ and a function $f : D \to \mathbb{R}$, where $I \subseteq D \subseteq \mathbb{R}$, f is* monotonically increasing on I *if for any $x, y \in I$, $x \leq y$ implies $f(x) \leq f(y)$.*

The definition below is similar to the definition above, but the direction of the last inequality is reversed.

Definition 7. *Given an interval $I = [a..b]$ and a function $f : D \to \mathbb{R}$, where $I \subseteq D \subseteq \mathbb{R}$, f is* monotonically decreasing on I *if for any $x, y \in I$, $x \leq y$ implies $f(x) \geq f(y)$.*

Note that monotonicity allows for staying put rather than always increasing (or decreasing). That entails that there are functions that are both monotonically increasing and monotonically decreasing on the same interval. Which ones?

When thinking about integration, we usually imagine integrating a continuous function. Surprisingly continuity is not necessary! The property of a function being monotonically increasing (or monotonically decreasing) is sufficient to prevent it from jumping around too crazily.

Fact 12. *Given an interval $I = [a..b]$ and a function $f : D \to \mathbb{R}$ where $I \subseteq D \subseteq \mathbb{R}$, if f is monotonically increasing on I or monotonically decreasing on I, then f is Reimann integrable on I.*

Now we state the details of how to bound a sum using an integral. The intuition behind the results involves understanding both the sum and the integrals in terms of area. The discrete columns of the sum either stick out above the curve for the lower bound or are contained below the curve for the upper bound.

Fact 13. *Given an interval $I = [m-1..n+1]$ and a function $f : D \to \mathbb{R}$ where $I \subseteq D \subseteq \mathbb{R}$, if f is monotonically increasing on I then* $\int_{m-1}^{n} f(x)\,dx \leq \sum_{k=m}^{n} f(k) \leq \int_{m}^{n+1} f(x)\,dx.$

Fact 14. *Given an interval $I = [m-1..n+1]$ and a function $f : D \to \mathbb{R}$ where $I \subseteq D \subseteq \mathbb{R}$, if f is monotonically decreasing on I then* $\int_{m}^{n+1} f(x)\,dx \leq \sum_{k=m}^{n} f(k) \leq \int_{m-1}^{n} f(x)\,dx.$

We can use this last fact to derive asymptotic bounds for H_n. Let $f(x) = 1/x$, and observe that f is monotonically decreasing on $[1..n+1]$. It follows from Fact 14 that $\int_{2}^{n+1} f(x)\,dx \leq \sum_{k=2}^{n} f(k) \leq \int_{1}^{n} f(x)\,dx.$ Hence $\ln(n+1) - \ln(2) + 1 \leq H_n \leq \ln(n) + 1$. From these two inequalities, it follows that $H_n \in O(\log(n))$ and $H_n \in \Omega(\log(n))$. Thus $H_n \in \Theta(\log(n))$.

5.3 Floors and Ceilings

We have seen that when analyzing an algorithm we are often counting some operation; this entails whole numbers — no fractions. Yet there is already a sense that functions such as $\sqrt{}$ and log are useful. We need some operation to get rid of the fractional part. And so we turn our attention to floors and ceilings; the operations that do exactly that.

Definition 8. *Given $x \in \mathbb{R}$, $\lfloor x \rfloor = \max\{y \in \mathbb{Z} \mid y \leq x\}$. We read "$\lfloor x \rfloor$" as "the floor of x."*

Examples

$$\begin{aligned} \lfloor 2.6 \rfloor &= \max\{\cdots, -2, -1, 0, 1, 2\} \\ &= 2 \end{aligned}$$

$$\begin{aligned} \lfloor -2.6 \rfloor &= \max\{\cdots, -5, -4, -3\} \\ &= -3 \end{aligned}$$

Definition 9. *Given $x \in \mathbb{R}$, $\lceil x \rceil = \min\{y \in \mathbb{Z} \mid y \geq x\}$. We read "$\lceil x \rceil$" as "the ceiling of x."*

We omit examples of the ceiling operation since it is similar to the floor.

It is also convenient on occasion to be able to talk about just the fractional part of a number. We define that operation next.

Definition 10. *Given $x \in \mathbb{R}$, $\varepsilon_x = x - \lfloor x \rfloor$. We read "$\varepsilon_x$" as "the fractional part of x."*

The next two results may be a little surprising. Usually we reason about inequalities using the transitive property. Not here. The lemma says that if an integer is less than a number then it is also less than an even smaller number! How can we be sure? It is because we're comparing an integer to the number and the smaller number is the floor.

Lemma 6. *For any $x \in \mathbb{R}$, for any $m \in \mathbb{Z}$, if $m \leq x$ then $m \leq \lfloor x \rfloor$.*

Proof.
Since $m \leq x$, $m \in \{y \in \mathbb{Z} \mid y \leq x\}$. Recall that $\lfloor x \rfloor$ is the largest element in that set. Thus for every $n \in \{y \in \mathbb{Z} \mid y \leq x\}$, $n \leq \lfloor x \rfloor$. And so $m \leq \lfloor x \rfloor$. □

The corollary is simply the contrapositive.

Corollary 4. *For any $x \in \mathbb{R}$, for any $m \in \mathbb{Z}$, if $m > \lfloor x \rfloor$ then $m > x$.*

The next lemma verifies that the fractional part is, in fact, small in magnitude.

Lemma 7. *Given $x \in \mathbb{R}$, $0 \leq \varepsilon_x < 1$.*

Proof.

- First we will show that $\varepsilon_x \geq 0$.
  It follows from the definition that $x \geq \lfloor x \rfloor$. Hence $x - \lfloor x \rfloor \geq 0$.
- Now we will show by contradiction that $\varepsilon_x < 1$.
  Suppose $\varepsilon_x \geq 1$.
  $$\begin{aligned} \varepsilon_x \geq 1 \quad &\Rightarrow \quad \lfloor x \rfloor + \varepsilon_x \geq \lfloor x \rfloor + 1 \\ &\Rightarrow \quad x \geq \lfloor x \rfloor + 1 \\ &\Rightarrow \quad \lfloor x \rfloor \geq \lfloor x \rfloor + 1 \text{ By Lemma 6. Contradiction!} \end{aligned}$$

□

We would like to be able to reason algebraically with the floor (and the ceiling). It is not clear how to do that. The next couple of lemmas show how. They state that the floor (ceiling) is equivalent to a pair of inequalities. For one set of inequalities, the floor is in the middle; for the other the real number is in the middle. In this form, traditional reasoning on inequalities can be used.

Lemma 8. *For any $x \in \mathbb{R}$, for any $n \in \mathbb{Z}$, $\lfloor x \rfloor = n$ if and only if $x - 1 < n \leq x$.*

Proof.

- First we will show that $\lfloor x \rfloor = n$ implies $x - 1 < n \leq x$.
  Suppose $\lfloor x \rfloor = n$.
  - It follows from definition 4 that $\lfloor x \rfloor \leq x$. Hence $n \leq x$.
  - Recall that $\varepsilon_x < 1$.
    $$\begin{aligned} \varepsilon_x < 1 \quad &\Rightarrow \quad n + \varepsilon_x < n + 1 \\ &\Rightarrow \quad n + \varepsilon_x - 1 < n \\ &\Rightarrow \quad \lfloor x \rfloor + \varepsilon_x - 1 < n \\ &\Rightarrow \quad x - 1 < n \end{aligned}$$
- Now we will show that $x - 1 < n \leq x$ implies $\lfloor x \rfloor = n$.
  Suppose $x - 1 < n \leq x$
  Since $n \leq x$, $n \in \{y \in \mathbb{Z} \mid y \leq x\}$. Further, $x - 1 < n$ implies $x < n + 1$. Thus $n + 1 \notin \{y \in \mathbb{Z} \mid y \leq x\}$, and so n must be the maximum element. Hence $\lfloor x \rfloor = n$.

□

Lemma 9. *For any $x \in \mathbb{R}$, for any $n \in \mathbb{Z}$, $\lceil x \rceil = n$ if and only if $x \leq n < x + 1$.*

Lemma 10. *For any $x \in \mathbb{R}$, for any $n \in \mathbb{Z}$, $\lfloor x \rfloor = n$ if and only if $n \leq x < n+1$.*

Proof.

- First we will show that $\lfloor x \rfloor = n$ implies $n \leq x < n+1$.
  Suppose $\lfloor x \rfloor = n$.
  - It follows from definition 4 that $\lfloor x \rfloor \leq x$. Hence $n \leq x$.
  - Since $\lfloor x \rfloor = \max\{y \in \mathbb{Z} \mid y \leq x\}$, we have that $n = \max\{y \in \mathbb{Z} \mid y \leq x\}$. And so, because $n+1 > n$, it must be that $n+1 \notin \{y \in \mathbb{Z} \mid y \leq x\}$ which entails that $n+1 > x$.
- Now we will show that $n \leq x < n+1$ implies $\lfloor x \rfloor = n$.
  Suppose $n \leq x < n+1$.
  Since $n \leq x$, $n \in \{y \in \mathbb{Z} \mid y \leq x\}$. Further, since $n+1 > x$, we have that for any $m > n$, $m \notin \{y \in \mathbb{Z} \mid y \leq x\}$. Thus n must be the maximum element. Hence $\lfloor x \rfloor = n$.

□

The next two corollaries follow from Lemma 10. They are intended to allow for reasoning about floors at a higher level. We can move integers out of the floor and we can throw away fractions inside the floor.

Corollary 5. *For any $x \in \mathbb{R}$, for any $m \in \mathbb{Z}$, $\lfloor x+m \rfloor = \lfloor x \rfloor + m$.*

Corollary 6. *For any $n \in \mathbb{Z}$, for any $\epsilon \in \mathbb{R}$, if $0 \leq \epsilon < 1$ then $\lfloor n+\epsilon \rfloor = n$.*

Lemma 11. *For any $x \in \mathbb{R}$, for any $n \in \mathbb{Z}$, $\lceil x \rceil = n$ if and only if $n-1 < x \leq n$.*

The final three results concern an equation of floors. The floor of the floor looks really messy. Under the right circumstances, we can remove the inner floor.

Theorem 9. *For any $x \in \mathbb{R}$, for any $a, b \in \mathbb{Z}^+$, $\lfloor x/(ab) \rfloor \geq \lfloor \lfloor x/a \rfloor / b \rfloor$.*

Proof.
Observe that $\lfloor \lfloor x/a \rfloor / b \rfloor \leq \lfloor x/a \rfloor / b \leq x/(ab)$.
The result follows from Lemma 6. □

Theorem 10. *For any $x \in \mathbb{R}$, for any $a, b \in \mathbb{Z}^+$, $\lfloor x/(ab) \rfloor \leq \lfloor \lfloor x/a \rfloor / b \rfloor$.*

Proof. By contradiction.
Suppose $\lfloor x/(ab) \rfloor > \lfloor \lfloor x/a \rfloor / b \rfloor$.

$\lfloor x/(ab) \rfloor > \lfloor \lfloor x/a \rfloor / b \rfloor$	$\Rightarrow$	$\lfloor x/(ab) \rfloor > \lfloor x/a \rfloor / b$	by Corollary 4
	$\Rightarrow$	$b\lfloor x/(ab) \rfloor > \lfloor x/a \rfloor$	
	$\Rightarrow$	$b\lfloor x/(ab) \rfloor > x/a$	by Corollary 4
	$\Rightarrow$	$\lfloor x/(ab) \rfloor > x/(ab)$	Contradiction!

□

Corollary 7. *For any $x \in \mathbb{R}$, for any $a, b \in \mathbb{Z}^+$, $\lfloor x/(ab) \rfloor = \lfloor \lfloor x/a \rfloor / b \rfloor$.*

Chapter 6

Recurrence Relations**

6.1 Introduction to Recurrence Relations

We have seen a couple of recurrence relations in previous chapters. In particular, we have seen the following recurrences.

$$\begin{aligned} T_i(0) &= 0 \\ T_i(n+1) &= 1 + T_i(n) \end{aligned}$$

$$\begin{aligned} T_{iSort}(0) &= 0 \\ T_{iSort}(n+1) &= (n+1) + T_{iSort}(n) \end{aligned}$$

These recurrences and others are important for analyzing algorithms. How do we go about solving recurrence relations? There are various approaches.

Substitution Here we plug values into the recurrence and attempt to recognize the function on the basis of the inputs and outputs.

Iteration Here we expand the recurrence so that it becomes a sum. Then we determine a closed form for the sum.

Induction Here we "guess" a solution and check that it is correct using mathematical induction (or the strong form of induction).

Difference Equation Methods Here we use methods that are similar to those used for solving differential equations.

Generating Functions Here we use formal power series to characterize the solution as a coefficient of a term and then extract a closed form of that coefficient.

Look It Up Here we find the answer from a reliable source or use a general theorem.

In the sections that follow we will focus on the first three approaches. At the end, we will also include sections outlining some techniques from difference equations and generating functions. Looking up a solution is a last resort since that approach provides little insight about the behavior of the recurrence.

6.2 Substitution By Example

Let us start with the example $T_i(n)$. We will start with the smallest domain element (zero) and work our way up.

n	$T_i(n)$
0	0
1	$1 + T_i(0) = 1 + 0 = 1$
2	$1 + T_i(1) = 1 + 1 = 2$
3	$1 + T_i(2) = 1 + 2 = 3$
4	$1 + T_i(3) = 1 + 3 = 4$

For every example, we see that the output of $T_i(n)$ is identical to the input. Thus it becomes apparent that $T_i(n) = n$.

Now let us try substitution with the example $T_{iSort}(n)$.

n	$T_{iSort}(n)$
0	0
1	$1 + T_{iSort}(0) = 1 + 0 = 1$
2	$2 + T_{iSort}(1) = 2 + 1 = 3$
3	$3 + T_{iSort}(2) = 3 + 3 = 6$
4	$4 + T_{iSort}(3) = 4 + 6 = 10$

Here the input-output relationship is much less obvious. So we turn to another technique.

6.3 Iteration by Example

Consider the example $T_i(n)$. Assume that n is sufficiently large, and expand.

$$\begin{aligned}
T_i(n) &= 1 + T_i(n-1) \\
&= 1 + (1 + T_i(n-2)) \\
&= 1 + 1 + (1 + T_i(n-3)) \\
&= 1 + 1 + 1 + T_i(n-3) \\
&= \underbrace{1 + \cdots + 1}_{k} + T_i(n-k) \leftarrow \text{Identify the pattern} \\
&= k + T_i(n-k)
\end{aligned}$$

The pattern should be a generalization of all lines. When $k = 1$, we should get the first line. When $k = 2$, we should get the second line. And so on. We can pick k to be whatever we wish, so we choose what is convenient: the value for k such that $n - k = 0$ (the base case). Let $k = n$. Thus we see that $T_i(n) = n + T_i(0) = n + 0 = n$.

Now let us expand $T_{iSort}(n)$. Again, assume that n is sufficiently large.

$$\begin{aligned} T_{iSort}(n) &= n + T_{iSort}(n-1) \\ &= n + ((n-1) + T_{iSort}(n-2)) \\ &= n + ((n-1) + ((n-2) + T_{iSort}(n-3))) \\ &= n + (n-1) + (n-2) + \cdots + (n-(k-1)) + T_{iSort}(n-k) \leftarrow \text{Identify the pattern} \end{aligned}$$

Once we have identified the pattern, let $k = n$. Thus we see that $T_{iSort}(n) = n + (n-1) + (n-2) + \cdots + (n-(n-1)) + T_{iSort}(0) = \sum_{i=0}^{n} i = n(n+1)/2$.

6.4 Induction By Example

Now let us prove those results that we found in the previous section using induction. We start with an inductive proof for the result involving $T_i(n)$.

Theorem 11. *For any $n \in \mathbb{N}$, $T_i(n) = n$.*

Proof. By mathematical induction.

- Observe that $T_i(0) = 0$.
- Assume that $T_i(k) = k$.
  $$\begin{aligned} T_i(k+1) &= 1 + T_i(k) \\ &= 1 + k \\ &= k + 1 \end{aligned}$$

□

Now we consider an inductive proof for the result involving $T_{iSort}(n)$.

Theorem 12. *For any $n \in \mathbb{N}$, $T_{iSort}(n) = n(n+1)/2$.*

Proof. By mathematical induction.

- Observe that $T_{iSort}(0) = 0 = 0(0+1)/2$.
- Assume that $T_{iSort}(k) = k(k+1)/2$.
  $$\begin{aligned} T_{iSort}(k+1) &= (k+1) + T_{iSort}(k) \\ &= (k+1) + k(k+1)/2 \\ &= 2(k+1)/2 + k(k+1)/2 \\ &= k(k+1)/2 + 2(k+1)/2 \\ &= (k+1)(k+2)/2 \end{aligned}$$

□

6.5 A More Challenging Example Involving Floor

In this section we will consider the following new recurrence.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(\lfloor n/2 \rfloor) + 1 \end{aligned}$$

We often see recurrences of this kind when studying divide and conquer algorithms. It is more challenging mostly because of the floor function. First we will try substitution, then we will use iteration to find the solution, and finally we will use induction to prove that that solution is indeed correct.

6.5.1 Substitution

Here too we will work our way up, but this time the smallest domain element is one.

n	$T(n)$
1	1
2	$T(1)+1=1+1=2$
3	$T(1)+1=1+1=2$
4	$T(2)+1=2+1=3$
5	$T(2)+1=2+1=3$
6	$T(3)+1=2+1=3$

Again, the input-output relationship is not obvious.

6.5.2 Iteration

Let us expand $T(n)$. Assume that n is sufficiently large.

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + 1 \\ &= (T(\lfloor \lfloor n/2 \rfloor /2 \rfloor) + 1) + 1 \\ &= (T(\lfloor \lfloor n/2^2 \rfloor /2 \rfloor) + 1) + 1 + 1 \\ &= T(\lfloor n/2^3 \rfloor) + 1 + 1 + 1 \\ &= T(\lfloor n/2^k \rfloor) + k \leftarrow \text{Identify the pattern} \end{aligned}$$

Once we have identified the pattern, let $k = k_0$ such that $\lfloor n/2^{k_0} \rfloor = 1$. Thus we see that $T(n) = T(\lfloor n/2^{k_0} \rfloor) + k_0 = T(1) + k_0 = 1 + k_0$. But what is k_0?

$$\begin{aligned} \lfloor n/2^{k_0} \rfloor = 1 \quad &\Rightarrow \quad 1 \leq n/2^{k_0} < 2 \\ &\Rightarrow \quad 2^{k_0} \leq n < 2^{k_0+1} \\ &\Rightarrow \quad k_0 \leq \lg(n) < k_0 + 1 \\ &\Rightarrow \quad \lfloor \lg(n) \rfloor = k_0 \end{aligned}$$

Thus $T(n) = 1 + \lfloor \lg(n) \rfloor$.

Numerical Check

n	$T(n)$	$1+\lfloor \lg(n) \rfloor$
1	1	$1+0=1$
2	2	$1+1=2$
3	2	$1+1=2$
4	3	$1+2=3$
5	3	$1+2=3$
6	3	$1+2=3$

6.5.3 Induction

In this section we make use of induction to prove that $T(n) = 1 + \lfloor \lg(n) \rfloor$. It turns out the induction argument needs a boost from an additional lemma. We start by assuming the lemma, and then prove it afterward.

Theorem 13. *For any $n \in \mathbb{N}$, if $n > 0$ then $T(n) = 1 + \lfloor \lg(n) \rfloor$.*

Proof. By the strong form of induction.

- Observe $T(1) = 1 = 1 + 0 = 1 + \lfloor 0 \rfloor = 1 + \lfloor \lg(1) \rfloor$.
- Assume $T(k) = 1 + \lfloor \lg(k) \rfloor$ if $0 < k < n$.

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + 1 \\ &= (1 + \lfloor \lg(\lfloor n/2 \rfloor) \rfloor) + 1 \\ &= (1 + (\lfloor \lg(n) \rfloor - 1)) + 1 \text{ why?} \\ &= 1 + \lfloor \lg(n) \rfloor \end{aligned}$$

□

The induction goes through when assuming $\lfloor \lg(\lfloor n/2 \rfloor) \rfloor = \lfloor \lg(n) \rfloor - 1$. This equation seems plausible, but it has not yet been proved. We start with an easy case — when n is even.

Lemma 12. *For any $n \in \mathbb{N}$, if $n > 1$ and n is even then $\lfloor \lg(\lfloor n/2 \rfloor) \rfloor = \lfloor \lg(n) \rfloor - 1$.*

Proof.
Suppose n is even.

$$\begin{aligned} \lfloor \lg(\lfloor n/2 \rfloor) \rfloor &= \lfloor \lg(n/2) \rfloor \text{ since } n \text{ is even} \\ &= \lfloor \lg(n) - \lg(2) \rfloor \\ &= \lfloor \lg(n) - 1 \rfloor \\ &= \lfloor \lg(n) \rfloor - 1 \end{aligned}$$

□

The case when n is odd is more involved. We need the following technical lemma.

Lemma 13. *For any $m \in \mathbb{N}$, if $m > 0$ then $\lfloor \lg(2m+1) \rfloor = \lfloor \lg(2m) \rfloor$.*

Proof.
Let $k = \lfloor \lg(2m+1) \rfloor$.

$$\begin{aligned}
\lfloor \lg(2m+1) \rfloor = k \quad &\Rightarrow \quad k \le \lg(2m+1) < k+1 \\
&\Rightarrow \quad 2^k \le 2m+1 < 2^{k+1} \\
&\Rightarrow \quad 2^k - 1 \le 2m < 2^{k+1} - 1 \\
&\Rightarrow \quad 2^k - 1 \le 2m < 2^{k+1} \qquad \text{since } 2^{k+1} - 1 < 2^{k+1} \\
&\Rightarrow \quad 2^k \le 2m < 2^{k+1} \qquad \text{since } 2m \text{ is even} \\
&\Rightarrow \quad k \le \lg(2m) < k+1 \\
&\Rightarrow \quad \lfloor \lg(2m) \rfloor = k
\end{aligned}$$

□

Now we are ready to tackle the lemma showing that $\lfloor \lg(\lfloor n/2 \rfloor) \rfloor = \lfloor \lg(n) \rfloor - 1$ even when n is odd.

Lemma 14. *For any $n \in \mathbb{N}$, if $n > 1$ and n is odd then $\lfloor \lg(\lfloor n/2 \rfloor) \rfloor = \lfloor \lg(n) \rfloor - 1$.*

Proof.

$$\begin{aligned}
\lfloor \lg(\lfloor n/2 \rfloor) \rfloor &= \lfloor \lg((n-1)/2) \rfloor \\
&= \lfloor \lg(n-1) - \lg(2) \rfloor \\
&= \lfloor \lg(n-1) - 1 \rfloor \\
&= \lfloor \lg(n-1) \rfloor - 1 \\
&= \lfloor \lg(n) \rfloor - 1 \text{ by Lemma 13}
\end{aligned}$$

□

Corollary 8. *For any $n \in \mathbb{N}$, if $n > 1$ then $\lfloor \lg(\lfloor n/2 \rfloor) \rfloor = \lfloor \lg(n) \rfloor - 1$.*

6.6 A Recurrence Relation with Full History

A *recurrence relation with full history* is a recurrence relation that involves the sum of all of the previous results. We will consider the following example.

$$\begin{aligned}
T(0) &= 1 \\
T(n) &= \sum_{i=0}^{n-1} T(i)
\end{aligned}$$

6.6.1 Substitution

$$\begin{aligned}
T(0) & & & = 1 \\
T(1) &= T(0) & & = 1 \\
T(2) &= T(0) + T(1) & & = 2 \\
T(3) &= T(0) + T(1) + T(2) & & = 4 \\
T(4) &= T(0) + T(1) + T(2) + T(3) & & = 8
\end{aligned}$$

6.6.2 Observations

When $n > 0$, $T(n) = \sum_{i=0}^{n-1} T(i)$. It follows that $T(n+1) - T(n) = \sum_{i=0}^{n} T(i) - \sum_{i=0}^{n-1} T(i) = T(n)$. After simplifying we get $T(n+1) - T(n) = T(n)$.

It is also worth noting that if we use the notation discussed in Chapter 5, we have $\Delta T = T$. This equation is reminiscent of $\frac{df}{dx} = f$. The solution of the differential equation is Ce^x. That leads us to suspect that the solution to our equation is also some exponential function.

6.6.3 Iteration

Here we have in mind that the base case for the derived recurrence is $T(1) = 1$; this entails that for the recursive case $n \geq 2$. We will solve $T(n) - T(n-1) = T(n-1)$ for $T(n)$. Note that the equation is equivalent to $T(n) = 2T(n-1)$.

Assume n is sufficiently large, and expand.

$$\begin{aligned} T(n) &= 2T(n-1) \\ &= 2(2T(n-2)) \\ &= 2(2(2T(n-3))) \\ &= 2^k T(n-k) \leftarrow \text{Identify the pattern} \end{aligned}$$

Once we have identified the pattern, let $k = n-1$. Thus we see that $T(n) = 2^{n-1}T(1) = 2^{n-1} \times 1 = 2^{n-1}$. So the solution to the original recurrence is $T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2^{n-1} & \text{otherwise} \end{cases}$.

6.6.4 Induction

Theorem 14. *For any $n \in \mathbb{N}$, if $n > 0$ then $T(n) = 2^{n-1}$.*

Proof. By the strong form of induction.

- Observe $T(1) = T(0) = 1 = 2^0 = 2^{1-1}$.
- Assume $T(k) = 2^{k-1}$ if $0 < k < n$.

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} T(i) \\ &= \sum_{i=1}^{n-1} T(i) + T(0) \\ &= \sum_{i=1}^{n-1} 2^{i-1} + T(0) \\ &= \sum_{i=1}^{n-1} 2^{i-1} + 1 \\ &= \sum_{i=0}^{n-2} 2^{i} + 1 \\ &= (2^{n-1} - 1) + 1 \\ &= 2^{n-1} \end{aligned}$$

□

6.7 A Difference Equation Method Example

Consider the recurrence for the Fibonacci numbers.

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_{n+1} &= F_n + F_{n-1} \end{aligned}$$

In this case, iteration doesn't help and the correct closed form is difficult to guess. It is worth noting that if we use the notation discussed in Chapter 5, we have $\Delta F = F_{n-1}$. This equation is reminiscent of $\frac{df}{dx} = f$, and so we are inclined to guess an exponential solution, say a^n where a is a constant. Assuming a solution of that form leads to the following equation.

$$a^{n+1} - a^n - a^{n-1} = 0$$

Factoring that equation yields the following.

$$a^{n-1}(a^2 - a - 1) = 0$$

Since it is clear from the initial conditions that $a \neq 0$, we focus on the solutions to the quadratic equation:

$$a = \frac{1 \pm \sqrt{5}}{2}.$$

Let $\varphi = \frac{1+\sqrt{5}}{2}$ and let $\hat{\varphi} = \frac{1-\sqrt{5}}{2}$. It turns out that neither φ or $\hat{\varphi}$ by itself can be used to compute the Fibonacci numbers. However, if some collection of functions are all solutions to a recurrence, then linear combinations of those functions are also solutions. Thus we should look for a solution of the form $F_n = c_1\varphi^n + c_2\hat{\varphi}^n$. We can solve for c_1 and c_2 using the initial conditions, and so we find that $F_n = \frac{1}{\sqrt{5}}(\varphi^n - \hat{\varphi}^n)$.

6.8 A Very Brief Introduction to Generating Functions

Given a recurrence $T(n)$, the generating function associated with $T(n)$ is the formal power series $\mathcal{G}(x) = \sum_{n=0}^{\infty} T(n)x^n$.

There are many valuable reasons to turn a recurrence into a generating function. One is that there is a rich set of operations that can be performed on generating functions such as scaling by constants, variables, computing derivatives, integrals, and so on. It is often straightforward to find a closed form for a generating function. At that point, with a little effort, the general coefficient can be extracted.

To turn a recurrence into a generating function, multiply both sides of the recurrence by x^n and sum over n. To extract the general coefficient, it is useful to know some formulas. For example, $\frac{1}{1-cx} = \sum_{n=0}^{\infty} c^n x^n$.

6.8.1 A Simple Example

Recall the following recurrence.

$$\begin{aligned} T_i(0) &= 0 \\ T_i(n+1) &= 1 + T_i(n) \end{aligned}$$

Let $\mathscr{G}(x) = \sum_{n=0}^{\infty} T_i(n)x^n$.

$$
\begin{aligned}
T_i(n+1) = 1 + T_i(n) \quad &\Rightarrow \quad \sum_{n=0}^{\infty} T_i(n+1)x^n = \sum_{n=0}^{\infty} x^n + \sum_{n=0}^{\infty} T_i(n)x^n \\
&\Rightarrow \quad \frac{\mathscr{G}(x)}{x} = \frac{1}{1-x} + \mathscr{G}(x) \\
&\Rightarrow \quad (1-x)\mathscr{G}(x) = \frac{x}{1-x} \\
&\Rightarrow \quad \mathscr{G}(x) = \frac{x}{(1-x)^2} \\
&\Rightarrow \quad \mathscr{G}(x) = x\frac{d}{dx}\frac{1}{(1-x)} \\
&\Rightarrow \quad \mathscr{G}(x) = \sum_{n=0}^{\infty} nx^n \\
&\Rightarrow \quad T_i(n) = n
\end{aligned}
$$

6.8.2 Another Example

Consider again the recurrence for the Fibonacci numbers.

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_{n+2} &= F_{n+1} + F_n
\end{aligned}
$$

Let $\mathscr{F}(x) = \sum_{n=0}^{\infty} F_n x^n$.

$$
\begin{aligned}
F_{n+2} = F_{n+1} + F_n \quad &\Rightarrow \quad \sum_{n=0}^{\infty} F_{n+2}x^n = \sum_{n=0}^{\infty} F_{n+1}x^n + \sum_{n=0}^{\infty} F_n x^n \\
&\Rightarrow \quad \frac{\mathscr{F}(x) - x}{x^2} = \frac{\mathscr{F}(x)}{x} + \mathscr{F}(x) \\
&\Rightarrow \quad \mathscr{F}(x) - x = x\mathscr{F}(x) + x^2\mathscr{F}(x) \\
&\Rightarrow \quad (1 - x - x^2)\mathscr{F}(x) = x \\
&\Rightarrow \quad \mathscr{F}(x) = \frac{x}{1 - x - x^2} \\
&\Rightarrow \quad \mathscr{F}(x) = \frac{x}{(1-\varphi x)(1-\hat{\varphi}x)} \\
&\Rightarrow \quad \mathscr{F}(x) = \frac{x}{\varphi - \hat{\varphi}}\left(\frac{\varphi}{1-\varphi x} - \frac{\hat{\varphi}}{1-\hat{\varphi}x}\right) \\
&\Rightarrow \quad \mathscr{F}(x) = \frac{x}{\sqrt{5}}\left(\frac{\varphi}{1-\varphi x} - \frac{\hat{\varphi}}{1-\hat{\varphi}x}\right) \\
&\Rightarrow \quad \mathscr{F}(x) = \frac{x}{\sqrt{5}}\left(\sum_{n=0}^{\infty} (\varphi^{n+1} - \hat{\varphi}^{n+1})x^n\right) \\
&\Rightarrow \quad F_n = \frac{1}{\sqrt{5}}(\varphi^n - \hat{\varphi}^n)
\end{aligned}
$$

Part II

Searching and Sorting

Chapter 7

Linear Search**

7.1 Introduction to Linear Search

Consider the following problem.

Problem 6. *Given a collection C of data values and a data value v, is*[1] *v in C?*

Notice that nothing was stated concerning the details of the data values, so we assume that the only operation we have available on our data values is equality testing. And so to find one in a collection, it may be necessary to test all the data values. Therefore sequence data structures such as lists and arrays seem no worse than any other, and we explicitly derive algorithms for both of those concrete data structures. Note that going through all of the data values implies linear time; hence the name *linear search*. However, at the end of this chapter we will see how to avoid that performance over time.

7.2 A Structurally Recursive Approach

We start by thinking about finding an element in a list. We first ask, "Is the element we are searching for in the empty list?" The trivial answer is false; whatever we are looking for cannot be found in the empty list. Next we ask, "If we know whether or not the element is in the tail, how can we determine whether or not the element is in the list?" Here too an answer presents itself fairly readily: the element is either (in) the head or in the tail. These observations yield the following algorithm.

$$\begin{array}{lcl} lSearch([], v) & = & \bot \\ lSearch(x :: xs, v) & = & (v = x) \text{ or } lSearch(xs, v) \end{array}$$

[1] It is straightforward to generalize this decision problem to the computation of a finite function.

Example

$$\begin{aligned}
\mathit{lSearch}([3,1,5,2,4],5) &= (5=3) \text{ or } \mathit{lSearch}([1,5,2,4],5)\\
&= \bot \text{ or } \mathit{lSearch}([1,5,2,4],5)\\
&= \mathit{lSearch}([1,5,2,4],5)\\
&= (5=1) \text{ or } \mathit{lSearch}([5,2,4],5)\\
&= \bot \text{ or } \mathit{lSearch}([5,2,4],5)\\
&= \mathit{lSearch}([5,2,4],5)\\
&= (5=5) \text{ or } \mathit{lSearch}([2,4],5)\\
&= \top \text{ or } \mathit{lSearch}([2,4],5)\\
&= \top
\end{aligned}$$

7.2.1 Complexity

We will measure performance by counting the equality tests. The recurrence relation will echo the structure of the recursive algorithm. The key question is what happens with the recursive call; this is determined by which kind of analysis we are doing.

Worst Case

In the worst case, we always have a recursive call. And so we get the following recurrence.

$$\begin{aligned}
T(0) &= 0\\
T(n+1) &= 1+T(n)
\end{aligned}$$

The solution is $T(n)=n$.

Best Case

In the best case, we find what we are looking for right away and we never have a recursive call. And so we get the following recurrence.

$$\begin{aligned}
T(0) &= 0\\
T(n+1) &= 1
\end{aligned}$$

The solution is $T(n)=\begin{cases} 0 & \text{if } n=0\\ 1 & \text{otherwise}\end{cases}$.

Average Case

The average case requires a more detailed analysis and some assumptions. We will be assuming that the element we are searching for is in the list. Therefore the list must be at least length one, and so there is one equality test. For the recursive case, we always must do that first equality test; so we will still add one. Notice that what varies with the worst and best case is the recursive call. Instead of one or zero, we will have a probability in between. We assume the list has length $n+1$ and that all the values are distinct. We pick one of those to be the element we are searching for. Assuming a uniform distribution, the probability that the first value is the one we are searching for is $\frac{1}{n+1}$, and the probability that it is not is $\frac{n}{n+1}$. And so we get the following recurrence.

$$\begin{aligned} T(1) &= 1 \\ T(n+1) &= 1 + \frac{n}{n+1} T(n) \end{aligned}$$

This recurrence requires a little more effort to solve. The denominator makes the sum messy, so we will multiply through by $n+1$: $(n+1)T(n+1) = (n+1) + nT(n)$. Multiplying n by $T(n)$ is also messy, but fortunately there is a similar product on the left-hand side. Let $\hat{T}(n) = nT(n)$. This renaming yields the following recurrence.

$$\begin{aligned} \hat{T}(1) &= 1 \\ \hat{T}(n+1) &= (n+1) + \hat{T}(n) \end{aligned}$$

The solution for $\hat{T}(n)$ is $\hat{T}(n) = \frac{n(n+1)}{2}$. Thus the solution for $T(n)$ is $T(n) = \frac{n+1}{2}$.

7.3 Searching Arrays

In the previous section, we derived a simple algorithm to search a list. Now we modify that algorithm so that we can search arrays instead. Instead of taking the head, we will use array indexing and compute the result $a[0]$. And instead of taking the tail, we will use slicing and compute the result $a[1:]$. These changes lead to the following algorithm.

$$\begin{aligned} aSearch(\langle\rangle, v) &= \bot \\ aSearch(a, v) &= (v = a[0]) \text{ or } aSearch(a[1:], v) \end{aligned}$$

Example

$$\begin{aligned} aSearch(\langle 3,1,5,2,4\rangle, 5) &= (5 = 3) \text{ or } aSearch(\langle 1,5,2,4\rangle, 5) \\ &= \bot \text{ or } aSearch(\langle 1,5,2,4\rangle, 5) \\ &= aSearch(\langle 1,5,2,4\rangle, 5) \\ &= (5 = 1) \text{ or } aSearch(\langle 5,2,4\rangle, 5) \\ &= \bot \text{ or } aSearch(\langle 5,2,4\rangle, 5) \\ &= aSearch(\langle 5,2,4\rangle, 5) \\ &= (5 = 5) \text{ or } aSearch(\langle 2,4\rangle, 5) \\ &= \top \text{ or } aSearch(\langle 2,4\rangle, 5) \\ &= \top \end{aligned}$$

Complexity

Unfortunately, with this modification, the performance becomes much worse since a single slice takes linear time. In fact, the use of slices is conceptual. For a more realistic implementation, we make use of virtual slices and so introduce a slice parameter.
The invariant is $aSearchHelper(a, v; i) = aSearch(a[i:], v)$.

First consider the case when $i \geq |a|$.

$$\begin{aligned} aSearchHelper(a, v; i) &= aSearch(a[i:], v) \\ &= aSearch(\langle\rangle, v) \\ &= \bot \end{aligned}$$

Now consider the case when $i < |a|$.

$$\begin{aligned} aSearchHelper(a, v; i) &= aSearch(a[i:], v) \\ &= (v = a[i:][0]) \text{ or } aSearch(a[i:][1:], v) \\ &= (v = a[i]) \text{ or } aSearch(a[i:][1:], v) \\ &= (v = a[i]) \text{ or } aSearch(a[i+1:], v) \\ &= (v = a[i]) \text{ or } aSearchHelper(a, v; i+1) \end{aligned}$$

The slicing parameter formulation now follows.

$$aSearchHelper(a, v; i) = \begin{cases} \bot & \text{if } i \geq |a| \\ (v = a[i]) \text{ or } aSearchHelper(a, v; i+1) & \text{otherwise} \end{cases}$$

This formulation does take only linear time, so we define *lSearch* by $lSearch(a, v) = aSearchHelper(a, v; 0)$.

Example

Let $a = \langle 3, 1, 5, 2, 4 \rangle$.

$$\begin{aligned} aSearchHelper(a, 5; 0) &= (5 = 3) \text{ or } aSearchHelper(a, 5; 1) \\ &= \bot \text{ or } aSearchHelper(a, 5; 1) \\ &= aSearchHelper(a, 5; 1) \\ &= (5 = 1) \text{ or } aSearchHelper(a, 5; 2) \\ &= \bot \text{ or } aSearchHelper(a, 5; 2) \\ &= aSearchHelper(a, 5; 2) \\ &= (5 = 5) \text{ or } aSearchHelper(a, 5; 3) \\ &= \top \text{ or } aSearchHelper(a, 5; 3) \\ &= \top \end{aligned}$$

7.4 Dynamic Linear Searching

When doing the average case analysis, we made a big assumption: we picked a particular probability distribution. In the absence of additional information, that is about the best guess we can make. Nevertheless, it may be that the actual probability distribution is quite different. As an extreme[2] example, imagine that all the searches made on a given sequence s are for the same element. If that element is at the end of s we will always get the worst case performance, but if that element is at the beginning of s we will get the best case constant time performance every time!

If we could look into the future and see a list of the searches made, we could compute the actual probability distribution function. Then we could determine the optimal static ordering: the ordering in which the most probable element is first in the sequence, the second most probable element is second in the sequence, and so on. This optimal static ordering could dramatically improve the performance of linear search.

Of course, we cannot actually look into the future. However, we can improve the ordering and approximate the optimal static ordering, by modifying the sequence based on the searches as they occur dynamically. In particular, if there is a search for an element e and it is found, e is moved closer to the

[2]This example is not as outlandish as it might at first seem. We might use linear search to look up the value of variables in a programming language implementation. It often happens that a program spends much of its time in a tight loop — such a loop might involve only a couple of variables.

front. Thus, over time, the elements with the highest probabilities will tend to be at the front, and the elements with low probabilities will be pushed to the end.

There are two concrete implementation strategies: *move-to-front* and *transpose*. The move-to-front strategy moves the element all the way to the front but otherwise leaves the ordering unchanged; this approach works well for sequences implemented as linked lists. The transpose strategy swaps the element with the element just before it. Note that after several searches for the same element, it will ripple up to the front. This approach works well for sequences implemented as arrays.

Chapter 8

Binary Search**

8.1 Introduction to Binary Search

Consider the following problem.

Problem 7. *Given a collection C of orderable data values and a data value v, is*[1] *v in C?*

There are various possible data structures and algorithms that can be used to solve this problem. We will focus here on *binary search*. First we will define binary search trees. We will define the binary search algorithm for binary search trees. Then we will turn our attention to representing binary search trees as arrays.

8.2 Binary Search Trees and Binary Search

Recall the definition of binary trees. First we specify the additional properties required of a binary search tree. Then we write the algorithms for the key operations on binary search trees culminating in the binary search algorithm. We conclude this section with a brief discussion of complexity.

8.2.1 Definition

A *binary search tree* is a binary tree where the data values must come from a totally ordered set, the left sub-trees contain the small data values, and the right sub-trees contain the large data values. The formal definition is below.

Definition 11. *A* binary search tree, *b, is a binary tree with the following property:*

If $b = \mathsf{nonEmptyBin}(\ell, v, r)$, *then*

- ℓ *is a binary search tree, and all data values of* ℓ *are less than* v, *and*
- r *is a binary search tree, and all data values of* r *are greater than* v.

[1] It is straightforward to generalize this decision problem to the computation of a finite function.

8.2.2 Operations

The following list summarizes the key operations on binary search trees.

- *bInsert*
  This operation also takes a value; it creates and/or extends a binary search tree by inserting that value in a correct position in the tree. In a functional context, a tree is returned similar to the supplied tree but with that value.
- *bMin*
  This operation finds the smallest/minimal value in the binary search tree.
- *bMax*
  This operation finds the largest/maximal value in the binary search tree.
- *bDeleteMin*
  This operation eliminates the smallest/minimal value in the binary search tree. In a functional context, a tree is returned similar to the supplied tree but without the minimal value.
- *bDeleteMax*
  This operation eliminates the largest/maximal value in the binary search tree. In a functional context, a tree is returned similar to the supplied tree but without the maximal value.
- *bDelete*
  This operation also takes a value; that value is eliminated from the binary search tree. In a functional context, a tree is returned similar to the supplied tree but without that value.
- *bSearch*
  This operation also takes a value; it determines whether or not that value is one of the values in the binary search tree.

Most of the algorithms for these operations follow from the definition of binary search trees. The operation *bDelete* is the trickiest.
We start by defining the insertion algorithm. The algorithm, following the definition, recursively inserts on the right or the left depending on whether the value is greater than or less than the value at the root. If the tree is empty, then the answer is a singleton tree.

$$
\begin{aligned}
\mathit{bInsert}(\mathsf{emptyBin}, v) &= \mathsf{nonEmptyBin}(\mathsf{emptyBin}, v, \mathsf{emptyBin}) \\
\mathit{bInsert}(\mathsf{nonEmptyBin}(\ell, \hat{v}, r), v) &= \begin{cases} \mathsf{nonEmptyBin}(\ell, \hat{v}, r) & \text{if } v = \hat{v} \\ \mathsf{nonEmptyBin}(\mathit{bInsert}(\ell, v), \hat{v}, r) & \text{if } v < \hat{v} \\ \mathsf{nonEmptyBin}(\ell, \hat{v}, \mathit{bInsert}(r, v)) & \text{if } v > \hat{v} \end{cases}
\end{aligned}
$$

Next we define the algorithm to find the minimum value. Since, by definition, small values are on the left, it merely involves looking to the left.

$$
\begin{aligned}
\mathit{bMin}(\mathsf{emptyBin}) &= \text{error “No Minimum”} \\
\mathit{bMin}(\mathsf{nonEmptyBin}(\mathsf{emptyBin}, v, r)) &= v \\
\mathit{bMin}(\mathsf{nonEmptyBin}(\ell, v, r)) &= \mathit{bMin}(\ell)
\end{aligned}
$$

The definition of the algorithm to find the maximum is similar.

Now we define the algorithm to delete the minimum value. The tree with the minimum value is replaced with its right sub-tree.

$$
\begin{array}{lcl}
\mathit{bDeleteMin}(\mathsf{emptyBin}) & = & \text{error ``No Minimum''} \\
\mathit{bDeleteMin}(\mathsf{nonEmptyBin}(\mathsf{emptyBin}, v, r)) & = & r \\
\mathit{bDeleteMin}(\mathsf{nonEmptyBin}(\ell, v, r)) & = & \mathsf{nonEmptyBin}(\mathit{bDeleteMin}(\ell), v, r)
\end{array}
$$

The definition of the algorithm to delete the maximum is similar.
At this point, we can define the algorithm to delete a supplied value. To delete this value, we must search for it. When found, it will be the root of a tree. If one of its sub-trees is empty, the tree can be replaced with the non-empty sub-tree. But if both sub-trees are non-empty, then the value at the root must be replaced. With what? There are two options: the minimum of the right sub-tree, or the maximum of the left sub-tree. We implement the first option.

$$
\begin{array}{lcl}
\mathit{bDeleteRoot}(\mathsf{emptyBin}) & = & \text{error ``No Root''} \\
\mathit{bDeleteRoot}(\mathsf{nonEmptyBin}(\mathsf{emptyBin}, v, r)) & = & r \\
\mathit{bDeleteRoot}(\mathsf{nonEmptyBin}(\ell, v, \mathsf{emptyBin})) & = & \ell \\
\mathit{bDeleteRoot}(\mathsf{nonEmptyBin}(\ell, v, r)) & = & \mathsf{nonEmptyBin}(\ell, \mathit{bMin}(r), \mathit{bDeleteMin}(r))
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{bDelete}(\mathsf{emptyBin}, v) & = & \text{error ``Value Not Found''} \\
\mathit{bDelete}(\mathsf{nonEmptyBin}(\ell, \hat{v}, r), v) & = & \begin{cases} \mathit{bDeleteRoot}(\mathsf{nonEmptyBin}(\ell, \hat{v}, r)) & \text{if } v = \hat{v} \\ \mathsf{nonEmptyBin}(\mathit{bDelete}(\ell, v), \hat{v}, r) & \text{if } v < \hat{v} \\ \mathsf{nonEmptyBin}(\ell, \hat{v}, \mathit{bDelete}(r, v)) & \text{if } v > \hat{v} \end{cases}
\end{array}
$$

Finally we define the binary search algorithm. The algorithm, following the definition, recursively searches on the right or the left depending on whether the value is greater than or less than the value at the root. If the tree is empty, then the value is certainly not there.

$$
\begin{array}{lcl}
\mathit{bSearch}(\mathsf{emptyBin}, v) & = & \bot \\
\mathit{bSearch}(\mathsf{nonEmptyBin}(\ell, \hat{v}, r), v) & = & \begin{cases} \top & \text{if } v = \hat{v} \\ \mathit{bSearch}(\ell, v) & \text{if } v < \hat{v} \\ \mathit{bSearch}(r, v) & \text{if } v > \hat{v} \end{cases}
\end{array}
$$

8.2.3 Time Complexity

All of the key operations on binary search trees involve search. Hence all the operations have the same time complexity. We argue somewhat abstractly concerning the best case, the worst case, and the average case.

In the best case, about half of the data values are in left sub-tree and about half are in the right sub-tree. Then the number of recursive calls is $\Theta(\log(n))$ since each call reduces the number of data values by about half.

In the worst case, the tree has empties all on one side and it looks like a list. Then eliminating an entire (empty) sub-tree accomplishes nothing, and the number of recursive calls is $\Theta(n)$.

If every tree is equally likely, then most trees will not have empties all on one side. Therefore, in the average case, the number of recursive calls is again $\Theta(\log(n))$.

We might wish to be able to guarantee that the worst case for the binary search algorithm never happens. *Balanced binary search trees* are binary search trees that yield the best case behavior of the binary search algorithm. There are many approaches for forcing binary search trees to be balanced. One is

red-black trees.

8.3 Representing Binary Search Trees as Arrays

Binary search trees are very flexible — we can add new data values dynamically. Nevertheless, frequently it is convenient to represent the binary search tree using a less flexible data structure: the array. We will first consider how to convert back and forth between binary search trees and arrays. Then we will consider an initial formulation of the binary search algorithm on arrays and conclude with a more efficient formulation.

8.3.1 Converting Between Binary Search Trees and Arrays

We can use the following algorithm to convert binary search trees to arrays.

$$\begin{array}{lcl} \mathit{arrayFromBST}(\mathsf{emptyBin}) & = & \langle\rangle \\ \mathit{arrayFromBST}(\mathsf{nonEmptyBin}(\ell, v, r)) & = & \mathit{arrayFromBST}(\ell) + \langle v \rangle + \mathit{arrayFromBST}(r) \end{array}$$

Notice that the resulting array will contain the data values in order.

When converting from an array to a binary search tree, we will assume that the data values are in order and that the array contains no repetitions. Still the array contains no information about which of the data values is the root. This lack is an advantage. We can pick the root that is best: the middle data value. That choice entails that the resulting tree is balanced.

$$\begin{array}{lcl} \mathit{BSTFromArray}(\langle\rangle) & = & \mathsf{emptyBin} \\ \mathit{BSTFromArray}(a) & = & \mathsf{nonEmptyBin}(\mathit{BSTFromArray}(a[0:m-1]), a[m], \mathit{BSTFromArray}(a[m+1:])) \\ & & \text{where } m = \lfloor (|a|-1)/2 \rfloor \end{array}$$

One might be surprised by the subtraction $(|a|-1)$. It is natural because the array indexing is zero-based. For example, if the length is five, the middle index is two.

8.3.2 Binary Search on Arrays with Slicing

We now state the definition of *searchSlice*. Then we can use the definitions to determine how to compute it.

$$\mathit{searchSlice}(a, v) = \mathit{bSearch}(\mathit{BSTFromArray}(a), v)$$

For the empty array we can compute the result.

$$\begin{array}{lcl} \mathit{searchSlice}(\langle\rangle, v) & = & \mathit{bSearch}(\mathit{BSTFromArray}(\langle\rangle), v) \\ & = & \mathit{bSearch}(\mathsf{emptyBin}, v) \\ & = & \bot \end{array}$$

For a non-empty array, there is a case analysis.

$$
\begin{array}{lcl}
searchSlice(a, v) & = & bSearch(BSTFromArray(a), v) \\
& = & bSearch(\mathsf{nonEmptyBin}(BSTFromArray(a[0:m-1]), a[m], BSTFromArray(a[m+1:])), v) \\
& = & \begin{cases} \top & \text{if } v = a[m] \\ bSearch(BSTFromArray(a[0:m-1]), v) & \text{if } v < a[m] \\ bSearch(BSTFromArray(a[m+1:]), v) & \text{if } v > a[m] \end{cases} \\
& = & \begin{cases} \top & \text{if } v = a[m] \\ searchSlice(a[0:m-1], v) & \text{if } v < a[m] \\ searchSlice(a[m+1:], v) & \text{if } v > a[m] \end{cases}
\end{array}
$$

The array-based slicing formulation now follows.

$$
\begin{array}{lcl}
searchSlice(\langle\rangle, v) & = & \bot \\
searchSlice(a, v) & = & \begin{cases} \top & \text{if } v = a[m] \\ searchSlice(a[0:m-1], v) & \text{if } v < a[m] \\ searchSlice(a[m+1:], v) & \text{if } v > a[m] \end{cases} \\
& & \text{where } m = \lfloor (|a|-1)/2 \rfloor
\end{array}
$$

Example

$$
\begin{array}{lcl}
searchSlice(\langle 10,20,30,40,50,60,70,80,90\rangle, 30) & = & searchSlice(\langle 10,20,30,40\rangle, 30) \\
& = & searchSlice(\langle 30,40\rangle, 30) \\
& = & \top
\end{array}
$$

Unfortunately, array slicing takes linear time, so our algorithm has become very inefficient. We will see how to overcome this problem in the next section.

8.3.3 Binary Search Using Slicing Parameters Instead of Slicing

We now state the definition of *searchHelp*. In this function, the slicing is not performed; rather, the slicing information is carried around in the two additional parameters.

$$searchHelp(a, v; \ell, h) = searchSlice(a[\ell : h], v)$$

First we consider the case when $\ell > h$.

$$
\begin{array}{lcl}
searchHelp(a, v; \ell, h) & = & searchSlice(a[\ell : h], v) \\
& = & searchSlice(\langle\rangle, v) \\
& = & \bot
\end{array}
$$

For the case when $\ell \leq h$, there is a case analysis. Recall that $|a[\ell : h]| = h - \ell + 1$. Hence we have $m = \lfloor (|a[\ell : h]| - 1)/2 \rfloor = \lfloor ((h - \ell + 1) - 1)/2 \rfloor = \lfloor (h - \ell)/2 \rfloor$. Note that $\ell + m \leq h$. Now define $\hat{m} = \ell + m$.

$$
\begin{aligned}
searchHelp(a, v; \ell, h) &= searchSlice(a[\ell : h], v) \\
&= \begin{cases} \top & \text{if } v = a[\ell : h][m] \\ searchSlice(a[\ell : h][0 : m-1], v) & \text{if } v < a[\ell : h][m] \\ searchSlice(a[\ell : h][m+1 :], v) & \text{if } v > a[\ell : h][m] \end{cases} \\
&= \begin{cases} \top & \text{if } v = a[\ell + m] \\ searchSlice(a[\ell : \ell + m - 1], v) & \text{if } v < a[\ell + m] \\ searchSlice(a[\ell + m + 1 : h], v) & \text{if } v > a[\ell + m] \end{cases} \\
&= \begin{cases} \top & \text{if } v = a[\hat{m}] \\ searchSlice(a[\ell : \hat{m} - 1], v) & \text{if } v < a[\hat{m}] \\ searchSlice(a[\hat{m} + 1 : h], v) & \text{if } v > a[\hat{m}] \end{cases} \\
&= \begin{cases} \top & \text{if } v = a[\hat{m}] \\ searchHelp(a, v; \ell, \hat{m} - 1) & \text{if } v < a[\hat{m}] \\ searchHelp(a, v; \hat{m} + 1, h) & \text{if } v > a[\hat{m}] \end{cases}
\end{aligned}
$$

The slicing parameter formulation now follows.

$$
\begin{aligned}
searchHelp(a, v; \ell, h) &= \begin{cases} \bot & \text{if } \ell > h \\ \top & \text{if } v = a[\hat{m}] \\ searchHelp(a, v; \ell, \hat{m} - 1) & \text{if } v < a[\hat{m}] \\ searchHelp(a, v; \hat{m} + 1, h) & \text{if } v > a[\hat{m}] \end{cases} \\
&\quad \text{where } \hat{m} = \ell + \lfloor (h - \ell)/2 \rfloor \\
search(a, v) &= searchHelp(a, v; 0, |a| - 1)
\end{aligned}
$$

Observe that an alternative formula for $\hat{m}$ may be derived as follows.

$$
\begin{aligned}
\hat{m} &= \lfloor (h - \ell)/2 \rfloor + \ell \\
&= \lfloor (h - \ell)/2 + \ell \rfloor \\
&= \lfloor (h - \ell)/2 + 2\ell/2 \rfloor \\
&= \lfloor (h + \ell)/2 \rfloor
\end{aligned}
$$

The derived formulation of $\hat{m}$ is often encountered in descriptions of binary search. The second formulation requires one fewer addition, but the first formulation has the advantage that there is no risk of overflow.

Example

Let $a = \langle 10, 20, 30, 40, 50, 60, 70, 80, 90 \rangle$.

$$
\begin{aligned}
searchHelp(a, 30; 0, 8) &= searchHelp(a, 30; 0, 3) \\
&= searchHelp(a, 30; 2, 3) \\
&= \top
\end{aligned}
$$

8.3.4 Time Complexity

We will use the function $T(n)$ to measure the worst case performance where $T(n)$ is the most number of calls needed to determine that a data value is in an ordered array as a function of the slice length; we assume that the data value really is in the ordered array. Note that since we are assuming that the data value is there, we ignore the case when the array slice is empty. On a singleton array, since we are assuming the data value is there, there is exactly one call. Thus $T(1) = 1$. For a larger array slice of length

n, we have the original call and all the calls generated by the recursive call on a slice of length not more than $\lfloor n/2 \rfloor$. Thus $T(n) = 1 + T(\lfloor n/2 \rfloor)$. Recall that the solution to this recurrence is $T(n) = 1 + \lfloor \lg(n) \rfloor$ and the asymptotic complexity is $\Theta(\log(n))$.

The most calls needed if the data value is not there is the most number of calls to get to the singleton case and then one more when it is determined that $a[m] \neq v$. Therefore, we define $T(0) = 0$ and identify the very worst case as $T(n) + 1$. Of course the asymptotic complexity is unchanged.

Chapter 9

Naive Sorting**

9.1 Introduction to Naive Sorting

Consider the sorting problem.

Problem 8. *Given a sequence s of orderable values, what is s' where $s' = \pi(s)$, π is a permutation, and $s'_1 \leq s'_2 \leq \cdots \leq s'_{|s'|-1} \leq s'_{|s'|}$?*

When beginning our study of sorting, we start with the two basic sorting algorithms: insertion sort and selection sort. These are the algorithms that come naturally when initially thinking about sorting — for good reason. They follow from formulaic approaches to recursion. And they are important. Although their asymptotic performance is mediocre, for smaller inputs they are quite speedy because of their simplicity. Further, one observes that the more sophisticated sorting algorithms are enhancements of these fundamental ones.

9.2 Insertion Sort

We have already encountered the insertion sort algorithm in chapter 3. We repeat it here.

$$
\begin{array}{lcl}
i(x, []) & = & [x] \\
i(x, y :: ys) & = & \begin{cases} x :: y :: ys & \text{if } x \leq y \\ y :: i(x, ys) & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
iSort([]) & = & [] \\
iSort(x :: xs) & = & i(x, iSort(xs))
\end{array}
$$

This algorithm is structurally recursive. We find it by first asking, "What is the result of sorting the empty list?" The trivial answer being once again the empty list. Then by asking, "If the tail is sorted, what can we do to find the sorted form of the entire list?" Here too an answer presents itself fairly readily: we insert the first element into its proper position in the sorted tail.

The performance is mediocre. As we have seen, the worst case time complexity $T_{iSort}(n) = \Theta(n^2)$. In the remainder of this section, we will not attempt to improve upon the time complexity, but we will

see about improving upon the space complexity by deriving a tail recursive accumulative form and then subsequently an imperative formulation that requires no additional space.[1]

9.2.1 Accumulative Insertion Sort

We start by considering an algorithm that generalizes the notion of insertion. Instead of combining a single value with a sorted list, we combine two sorted lists. We call this operation merging and use the notation $\bowtie$.

Merging considers both sequences and allows only the smallest of each to pass through when constructing the merged sequence. Since, by assumption, both sequences are already sorted it is only necessary to look at the first element of each sequence each time around.

$$
\begin{array}{lcl}
[] \bowtie ys & = & ys \\
xs \bowtie [] & = & xs \\
(x :: xs) \bowtie (y :: ys) & = & \begin{cases} x :: (xs \bowtie (y :: ys)) & \text{if } x < y \\ y :: ((x :: xs) \bowtie ys) & \text{if } x > y \\ x :: y :: (xs \bowtie ys) & \text{if } x = y \end{cases}
\end{array}
$$

We define *iSortAccum* via the following invariant: $iSortAccum(a; \ell) = iSort(\ell) \bowtie a$. The accumulation parameter comes first to make it easier to compare to the subsequent imperative formulation. Correct sorting occurs when we initialize a to the empty list. Now we can derive the algorithm.
We start with the base case.

$$
\begin{array}{lcl}
iSortAccum(a; []) & = & iSort([]) \bowtie a \\
 & = & [] \bowtie a \\
 & = & a
\end{array}
$$

Then we derive the recursive case.

$$
\begin{array}{lcl}
iSortAccum(a; x :: xs) & = & iSort(x :: xs) \bowtie a \\
 & = & i(x, iSort(xs)) \bowtie a \\
 & = & ([x] \bowtie iSort(xs)) \bowtie a \\
 & = & (iSort(xs) \bowtie [x]) \bowtie a \\
 & = & iSort(xs) \bowtie ([x] \bowtie a) \\
 & = & iSort(xs) \bowtie i(x, a) \\
 & = & iSortAccum(i(x, a); xs)
\end{array}
$$

Thus we have derived the following algorithm.

$$
\begin{array}{lcl}
iSortAccum(a; []) & = & a \\
iSortAccum(a; x :: xs) & = & iSortAccum(i(x, a); xs)
\end{array}
$$

Example: An Ascending Sequence

$$
\begin{array}{lcl}
iSortAccum([]; [1,2,3,4]) & = & iSortAccum(i(1, []); [2,3,4]) \\
 & = & iSortAccum([1]; [2,3,4])
\end{array}
$$

[1] An algorithm that requires no additional space is called an *in-place* algorithm.

$$
\begin{aligned}
&= iSortAccum(i(2,[1]);[3,4])\\
&= iSortAccum(1::i(2,[]);[3,4])\\
&= iSortAccum(1::[2];[3,4])\\
&= iSortAccum([1,2];[3,4])\\
&= iSortAccum(i(3,[1,2]);[4])\\
&= iSortAccum(1::i(3,[2]);[4])\\
&= iSortAccum(1::2::i(3,[]);[4])\\
&= iSortAccum(1::2::[3];[4])\\
&= iSortAccum([1,2,3];[4])\\
&= iSortAccum(i(4,[1,2,3]);[])\\
&= iSortAccum(1::i(4,[2,3]);[])\\
&= iSortAccum(1::2::i(4,[3]);[])\\
&= iSortAccum(1::2::3::i(4,[]);[])\\
&= iSortAccum(1::2::3::[4];[])\\
&= iSortAccum([1,2,3,4];[])\\
&= [1,2,3,4]
\end{aligned}
$$

Example: A Descending Sequence

$$
\begin{aligned}
iSortAccum([];[4,3,2,1]) &= iSortAccum(i(4,[]);[3,2,1])\\
&= iSortAccum([4];[3,2,1])\\
&= iSortAccum(i(3,[4]);[2,1])\\
&= iSortAccum([3,4];[2,1])\\
&= iSortAccum(i(2,[3,4]);[1])\\
&= iSortAccum([2,3,4];[1])\\
&= iSortAccum(i(1,[2,3,4]);[])\\
&= iSortAccum([1,2,3,4];[])\\
&= [1,2,3,4]
\end{aligned}
$$

9.2.2 Imperative Insertion Sort

To develop the idea of how an imperative insertion sort should work, we summarize the computation from section 9.2.1.

accumulator	input
[]	[4,3,2,1]
[4]	[3,2,1]
[3,4]	[2,1]
[2,3,4]	[1]
[1,2,3,4]	[]

Observe that the accumulator on the left is a sorted sequence but the sequence on the right is not. The element at the beginning of the sequence on the left is moved over repeatedly until that sequence is exhausted and the accumulator has all the elements (in order). When sorting an array we will represent both of these sequences in a single array.

Notation 4. *Given an array a, a* cut *in a is a conceptual division between elements in a.*

Notation 5. *Given an array a, a* mark *m in a is an index that represents a cut. Note that there are multiple index representations of a cut.*

In the diagram below we see the single array representing the previous accumulator and input sequences. A cut separates the two; it is expressed as a vertical line. In general, we understand the part of the array to the left of the cut contains sorted elements but the elements in the part to the right are unconstrained: | sorted | unconstrained |. The nature of sorting is then to pull the cut from left to right while maintaining this property.

4	3	2	1
4	3	2	1
3	4	2	1
2	3	4	1
1	2	3	4

Here we let the initial sorted region be the singleton region (second line above) rather than the empty region, and we represent the cut by the mark that is the index to the immediate left of the cut. The pseudo-code below advances the mark after inserting into the proper place in the sorted region.

def ISORT(a) :

1. **for** $m \leftarrow 0$ **to** $|a| - 2$:
2. INSERT(a, m)

The index of the value to be inserted is one more than the mark, so it is enough for the INSERT procedure to take as parameters the array a and the mark m. Insertion into the sorted region of the array works differently from the way list insertion works. Since it is equally easy to move through the array from left to right or from right to left, we start at the cut and move left. If the smaller element is too far to the right, we repeatedly swap it with what is on its left until it is in the correct position.

def INSERT(a, m) :

1. $\hat{m} \leftarrow m$
2. **while** $\hat{m} \geq 0$ **and** $a[\hat{m}] > a[\hat{m}+1]$:
3. **swap** $a[\hat{m}], a[\hat{m}+1]$
4. $\hat{m} \leftarrow \hat{m} - 1$

To get a handle on the time complexity, we should count comparisons, swaps, or both. Because of that, some of the details are different. But the asymptotic complexity in the best and worst case of the imperative formulation are nonetheless the same as in the original discussion of the time complexity of insertion sort.

9.3 Selection Sort

From one point of view the selection sort algorithm emerges from a dual to structural recursion. With structural recursion we take a list apart based on its structure. The dual is then to put together a list based

on its structure. And so, if the answer has the form $y :: ys$, y must be the smallest element and ys must be the ordered list of elements taken from the input but excluding y.

9.3.1 A Slow Sort

At this point we abandon the notion of a dual to structural recursion. Instead we will select the maximum rather than the minimum and formulate the selection sort algorithm using concatenation at the end of the list. This alternative formulation emphasizes the similarity among the different formulations that follow.

$$\begin{array}{lcl} sSort([]) & = & [] \\ sSort(xs) & = & \text{let } (m,o) = getMax(xs) \text{ in } sSort(o) + [m] \end{array}$$

The function *getMax* uses accumulative recursion to break the list into the maximum and everything else. An incidental consequence is that the list of everything else is reversed. That property can be avoided with a little additional code. The maximum is determined by a "champion" algorithm. If there are multiple occurrences of the maximum, the first one remains the champion and the others are included in everything else.

$$\begin{array}{lcl} getMax(x :: xs) & = & getMax3(xs, x, []) \\ \\ getMax3([], c, o) & = & (c, o) \\ getMax3(x :: xs, c, o) & = & \begin{cases} getMax3(xs, x, c :: o) & \text{if } x > c \\ getMax3(xs, c, x :: o) & \text{otherwise} \end{cases} \end{array}$$

Example: An Ascending Sequence

We elide the computations involving the function *getMax*.

$$\begin{array}{lcl} sSort([1,2,3,4]) & = & sSort([3,2,1]) + [4] \\ & = & (sSort([1,2]) + [3]) + [4] \\ & = & ((sSort([1]) + [2]) + [3]) + [4] \\ & = & (((sSort([]) + [1]) + [2]) + [3]) + [4] \\ & = & ((([] + [1]) + [2]) + [3]) + [4] \\ & = & (([1] + [2]) + [3]) + [4] \\ & = & ([1,2] + [3]) + [4] \\ & = & [1,2,3] + [4] \\ & = & [1,2,3,4] \end{array}$$

Example: A Descending Sequence

We elide the computations involving the function *getMax*.

$$\begin{array}{lcl} sSort([4,3,2,1]) & = & sSort([1,2,3]) + [4] \\ & = & (sSort([2,1]) + [3]) + [4] \\ & = & ((sSort([1]) + [2]) + [3]) + [4] \end{array}$$

$$
\begin{aligned}
&= (((sSort([]) + [1]) + [2]) + [3]) + [4] \\
&= ((([] + [1]) + [2]) + [3]) + [4] \\
&= (([1] + [2]) + [3]) + [4] \\
&= ([1,2] + [3]) + [4] \\
&= [1,2,3] + [4] \\
&= [1,2,3,4]
\end{aligned}
$$

Just as we counted the number of calls to i to talk about the performance of *iSort*, here we will count the number of calls to *getMax* to talk about the performance of *sSort*. Let the function $T_g(n)$ be the number of times *getMax* is called from *getMax* not including that original call. Since *sSort* is recursive, we can recursively characterize $T_{sSort}(n)$.

$$
\begin{aligned}
T_{sSort}(0) &= 0 \\
T_{sSort}(n+1) &= (1 + T_g(n)) + T_{sSort}(n)
\end{aligned}
$$

Consider the following recurrence for $T_g(n)$. Notice that it is the same for both the best and worst cases.

$$
\begin{aligned}
T_g(0) &= 0 \\
T_g(n+1) &= 1 + T_g(n)
\end{aligned}
$$

The solution to this recurrence is $T_g(n) = n$. Hence we get the following recurrence for $T_{sSort}(n)$.

$$
\begin{aligned}
T_{sSort}(0) &= 0 \\
T_{sSort}(n+1) &= (n+1) + T_{sSort}(n)
\end{aligned}
$$

We see from our experience that the solution to the recurrence above is $T_{sSort}(n) = n^2/2 + n/2$.

9.3.2 Accumulative Selection Sort

We define *sSortAccum* via the following invariant: $sSortAccum(\ell; a) = sSort(\ell) + a$. Correct sorting occurs when we initialize a to the empty list. Now we can derive the algorithm.
We start with the base case.

$$
\begin{aligned}
sSortAccum([]; a) &= sSort([]) + a \\
&= [] + a \\
&= a
\end{aligned}
$$

Then we derive the recursive case, where we assume that $|xs| > 0$.

$$
\begin{aligned}
sSortAccum(xs; a) &= sSort(xs) + a \\
&= (\text{let } (m, o) = getMax(xs) \text{ in } sSort(o) + [m]) + a \\
&= \text{let } (m, o) = getMax(xs) \text{ in } sSort(o) + ([m] + a) \\
&= \text{let } (m, o) = getMax(xs) \text{ in } sSort(o) + (m :: a) \\
&= \text{let } (m, o) = getMax(xs) \text{ in } sSortAccum(o; m :: a)
\end{aligned}
$$

Thus we have derived the following algorithm.

$$
\begin{aligned}
sSortAccum([]; a) &= a \\
sSortAccum(xs; a) &= \text{let } (m, o) = getMax(xs) \text{ in } sSortAccum(o; m :: a)
\end{aligned}
$$

Example: An Ascending Sequence

We elide the computations involving the function *getMax*.

$$\begin{aligned}
sSortAccum([1,2,3,4];[]) &= sSortAccum([3,2,1];4::[]) \\
&= sSortAccum([3,2,1];[4]) \\
&= sSortAccum([1,2];3::[4]) \\
&= sSortAccum([1,2];[3,4]) \\
&= sSortAccum([1];2::[3,4]) \\
&= sSortAccum([1];[2,3,4]) \\
&= sSortAccum([];1::[2,3,4]) \\
&= sSortAccum([];[1,2,3,4]) \\
&= [1,2,3,4]
\end{aligned}$$

Example: A Descending Sequence

We elide the computations involving the function *getMax*.

$$\begin{aligned}
sSortAccum([4,3,2,1];[]) &= sSortAccum([1,2,3];4::[]) \\
&= sSortAccum([1,2,3];[4]) \\
&= sSortAccum([2,1];3::[4]) \\
&= sSortAccum([2,1];[3,4]) \\
&= sSortAccum([1];2::[3,4]) \\
&= sSortAccum([1];[2,3,4]) \\
&= sSortAccum([];1::[2,3,4]) \\
&= sSortAccum([];[1,2,3,4]) \\
&= [1,2,3,4]
\end{aligned}$$

9.3.3 Imperative Selection Sort

To develop the idea of how an imperative insertion sort should work, we start by summarizing the computation from section 9.3.2.

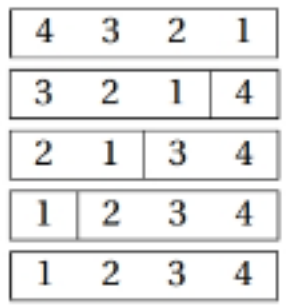
input	accumulator
[4,3,2,1]	[]
[1,2,3]	[4]
[2,1]	[3,4]
[1]	[2,3,4]
[]	[1,2,3,4]

Observe that the accumulator on the right is a sorted sequence containing the largest elements but the sequence on the left is not. The maximal element in the sequence on the left is moved over repeatedly until that sequence is exhausted and the accumulator has all the elements (in order). Since the reversal of the input is incidental, we consider a variation in which the reversal does not occur.

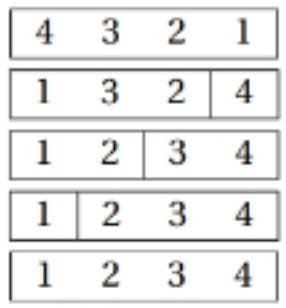
input	accumulator
[4,3,2,1]	[]
[3,2,1]	[4]
[2,1]	[3,4]
[1]	[2,3,4]
[]	[1,2,3,4]

When sorting an array we will represent both of these sequences in a single array.

In the diagram below we see the single array representing the previous input and accumulator sequences. A cut separates the two; it is expressed as a vertical line. In general, we understand the part of the array to the right of the cut contains maximal sorted elements but the elements in the part to the left are unconstrained: | unconstrained | maximal sorted |. The nature of sorting is then to pull the cut from right to left while maintaining this property.



The nature of the movement in the imperative form is still slightly different. Instead of simply moving the maximal element from one region to the other, it is swapped with the element adjacent to the sorted region. This behavior is illustrated below.



The initial sorted region is the empty region (first line above), and we represent the cut by the mark that is the index to the immediate left of the cut. The pseudo-code below moves (decrements) the mark after finding the position of the maximal element and swapping that value with the value at the mark.

def SSORT(a) :

1. **for** $m \leftarrow |a| - 1$ **downto** 1 :
2. $\quad i \leftarrow$MAXPOS(a, m)
3. $\quad$ **swap** $a[m], a[i]$

The procedure MAXPOS only finds the index of the maximal element in the unconstrained region, so the only parameters needed are the array a and the mark m. The maximal element index is found using a "champion" algorithm in the imperative version as well.

def MAXPOS(a, m) :

1. $v \leftarrow a[m]$ # Champion Value
2. $i \leftarrow m$ # Champion Index
3. **for** $j \leftarrow m-1$ **downto** 0 :
4. **if** $a[j] > v$:
5. $v \leftarrow a[j]$
6. $i \leftarrow j$
7. **return** i

Again here in the imperative case, some of the details are different. But the asymptotic complexity in the best and worst case are nonetheless the same: $T_{\text{SSORT}}(n) \in \Theta(n^2)$.

Chapter 10

Heaps**

10.1 Introduction to Heaps

Consider the following problem.

Problem 9. *Given a collection C of orderable data values, what is the minimum value (maximum value) in C?*

A *heap*, or *priority queue*, is an abstract data type that supports the operations of finding the minimum value (maximum value), removing the minimum value (maximum value), and adding a value to the collection. Often other operations are also supported such as merging two heaps. And if the implementation is imperative, updating the values in the heap may also be desirable.

Finding both the minimum and the maximum is not allowed. Often "min" or "max" is used when speaking of a heap to clarify which kind is being used.

It is straightforward to implement a min-heap using a list. Two implementations follow. The first represents the min-heap as an unordered list. Finding the minimum just involves searching the list for the smallest element. To remove the minimum, a function *getMin*, similar to *getMax* from selection sort, can be used. Inserting an element is easy; just put it on the beginning of the list.

$$
\begin{array}{lcl}
\mathit{findMinHeap}([]) & = & \text{error "empty heap"} \\
\mathit{findMinHeap}(xs) & = & \min(xs) \\
\\
\mathit{removeMinHeap}([]) & = & \text{error "empty heap"} \\
\mathit{removeMinHeap}(xs) & = & \text{let } (\_, o) = \mathit{getMin}(xs) \text{ in } o \\
\\
\mathit{insertMinHeap}(x, xs) & = & x :: xs
\end{array}
$$

For the unordered list approach, the worst case asymptotic time complexity of *findMinHeap* is $\Theta(n)$.

For the unordered list approach, the worst case asymptotic time complexity of *removeMinHeap* is $\Theta(n)$.

For the unordered list approach, the asymptotic time complexity of *insertMinHeap* is $\Theta(1)$.

A max-heap could be implemented in a similar fashion.

The second implementation represents the min-heap as an ordered list. It is easy to find the minimum since it is at the beginning. Removing the minimum is also easy; it is just removing the head of the list, which is the tail. Inserting while maintaining the order of the list takes more work, but we can use the insertion function from insertion sort.

$$
\begin{array}{lcl}
\mathit{findMinHeap}([]) & = & \text{error ``empty heap''} \\
\mathit{findMinHeap}(x :: xs) & = & x \\
 & & \\
\mathit{removeMinHeap}([]) & = & \text{error ``empty heap''} \\
\mathit{removeMinHeap}(x :: xs) & = & xs \\
 & & \\
\mathit{insertMinHeap}(x, ys) & = & i(x, ys)
\end{array}
$$

For the ordered list approach, the asymptotic time complexity of *findMinHeap* is $\Theta(1)$.

For the ordered list approach, the asymptotic time complexity of *removeMinHeap* is $\Theta(1)$.

For the ordered list approach, the worst case asymptotic time complexity of *insertMinHeap* is $\Theta(n)$.

A max-heap could be implemented in a similar fashion.

Notice that both implementations have at least one operation that takes linear time. In the following sections we examine alternative approaches for implementing the heap abstract data type. All of these approaches involve clever tree representations.

10.2 Binary Search Trees

We can represent the min-heap using a binary search tree. In fact, in Chapter 8 we already defined the operations necessary to implement both a min-heap and a max-heap. Here we show how to use those operations to specifically define a min-heap.

$$
\begin{array}{lcl}
\mathit{findMinHeap}(\text{emptyBin}) & = & \text{error ``empty heap''} \\
\mathit{findMinHeap}(b) & = & \mathit{bMin}(b) \\
 & & \\
\mathit{removeMinHeap}(\text{emptyBin}) & = & \text{error ``empty heap''} \\
\mathit{removeMinHeap}(b) & = & \mathit{bDeleteMin}(b) \\
 & & \\
\mathit{insertMinHeap}(x, b) & = & \mathit{bInsert}(b, x)
\end{array}
$$

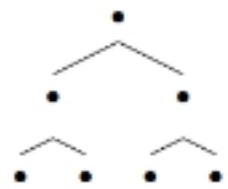
Assuming the binary search tree representation is a balanced binary tree, the asymptotic complexity for all of these operations is $\Theta(\log(n))$.

10.3 Left-Complete Binary Trees

With the binary tree approach, we assume the ordering and subsequently attempt to keep the tree balanced. This time let's start with the idea of a balanced binary tree and see about imposing the ordering

afterward. Recall the notion of a complete binary tree: it is a binary tree where every level is completely filled.

Example



Complete binary tree with two levels

Unfortunately, the number of nodes in a complete binary tree cannot be any number.

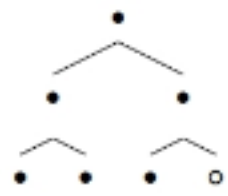
Fact 15. *For any $i \in \mathbb{N}$, if b is a complete binary tree with i levels then $|b| = 2^{i+1} - 1$.*

We can have any number of nodes if we relax the condition that every level is completely filled. Specifically, we will say that the last level need not be completely filled.

Definition 12. *A binary tree b with i levels is said to be a* left-complete binary tree *with i levels if b has the structure of a complete binary tree with i levels where the k rightmost nodes on level i are missing and $0 \leq k < 2^i$.*

Note that a left-complete binary tree with i levels has leaves only on levels i and $i-1$.

Example



Left-complete binary tree with two levels

A left-complete binary tree is still sufficiently balanced so that the longest path from the root is fairly short.

Theorem 15. *Given a left-complete binary tree b with i levels, $i = \lceil \lg(|b|+1) \rceil - 1$.*

Proof.

$$
\begin{aligned}
2^i - 1 < |b| \leq 2^{i+1} - 1 \quad &\Rightarrow \quad 2^i < |b| + 1 \leq 2^{i+1} \\
&\Rightarrow \quad i < \lg(|b|+1) \leq i+1 \\
&\Rightarrow \quad i+1 = \lceil \lg(|b|+1) \rceil \\
&\Rightarrow \quad i = \lceil \lg(|b|+1) \rceil - 1
\end{aligned}
$$

□

Now that we have established the structure of the tree, we introduce the ordering so that we can use this tree as a heap. A *binary min-(max-)heap* is a binary tree where the data values must come from a totally ordered set, and the data value at any node is less than (greater than) or equal to the value of all its descendants. Thus, the extreme value is at the root of the tree. The formal definition is below.

Definition 13. *A* binary min-heap, *b, is a binary tree with the following property:*
If $b = \mathsf{nonEmptyBin}(\ell, v, r)$, *then*

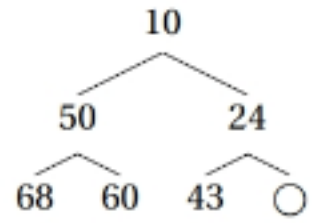
- ℓ *is a binary min-heap, and all data values of* ℓ *are greater than or equal to* v, *and*
- r *is a binary min-heap, and all data values of* r *are greater than or equal to* v.

A binary max-heap is defined similarly.

Definition 14. *A left-complete binary tree with* i *levels,* b, *is said to be a* left-complete binary min-heap *with* i *levels if* b *is a binary min-heap.*

A left-complete binary max-heap with i levels is defined similarly.

Example



Left-complete binary min-heap with two levels

10.3.1 Illustrating the Heap Operations for a Left-Complete Binary Min-Heap

Given a left-complete binary min-heap with i levels, b, consider the following conceptual implementation of the heap operations. We postpone the concrete implementation to the next section.

findMinHeap Finding the minimum element is easy. The minimum element is always at the root of the tree. Thus the asymptotic time complexity is $\Theta(1)$.

removeMinHeap Removing the minimum is more subtle because removing the root breaks the structure of the tree. We start by immediately repairing the structure of the tree by removing the rightmost leaf on level i and placing its value at the root. This modification potentially disrupts the binary min-heap ordering property — but only locally. We compare the modified node and its two children. The smallest of the three is swapped with the node value. This process is then repeated if one of the children was modified. (See Figure 10.1.) The time to repair the binary min-heap ordering property is proportional to the number of levels i. Thus, if n is the size of b, then the asymptotic time complexity is $\Theta(\log(n))$.

insertMinHeap Inserting an element also involves some repair. Placing the value initially simply involves either adding the value at the leftmost empty position at level i, or, if level i is full, adding the value at the leftmost position at level $i+1$. This addition potentially disrupts the binary min-heap ordering property — but only locally. We compare the modified node to its parent and swap them if they are out of order. This process is then repeated if the parent was modified. (See Figure 10.2.) The time to repair the binary min-heap ordering property is proportional to the number of levels i. Thus, if n is the size of b, then the asymptotic time complexity is $\Theta(\log(n))$.

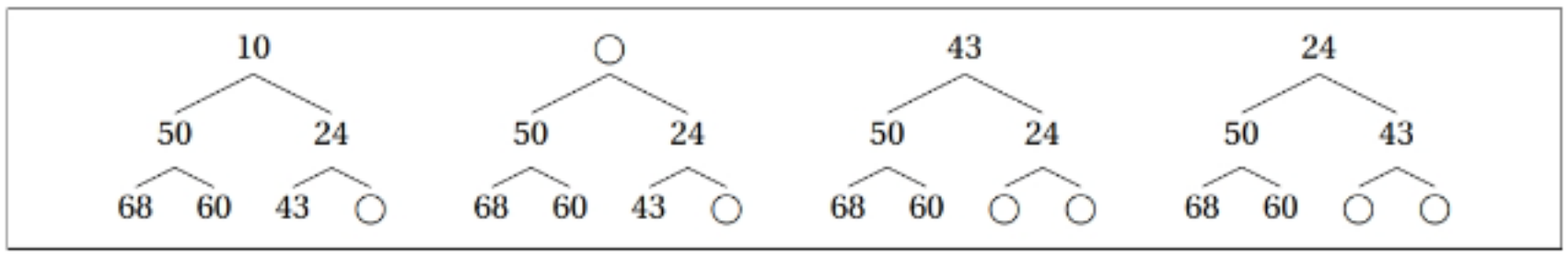


Figure 10.1: Removing the minimum from a left-complete binary min-heap

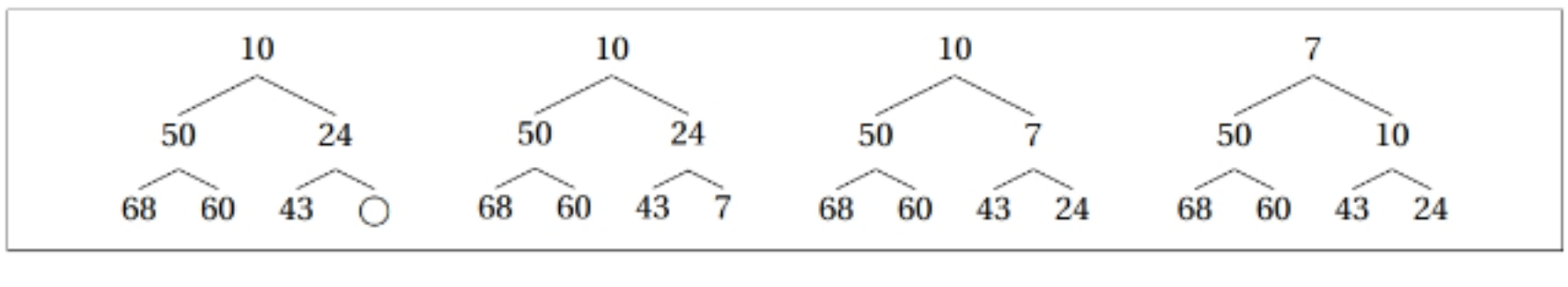


Figure 10.2: Inserting an element into a left-complete binary min-heap

10.3.2 Representing the Tree as an Array

Usually a recursive definition leads to a recursive implementation. For left-complete binary heaps, that is not the case. Note that we need to be able to easily access both the root and the leaves, and to move both up and down the tree. Even a pointer based implementation would be quite complicated. Fortunately, there is an alternative: a left-complete binary heap can be represented as an array coupled with an index — where the index points to the next available leaf. The root is index zero and the indices increase across a level; see Figure 10.3.

Given an index m representing a node in the tree, notice that $2m+1$ is the left child and that $2m+2$ is the right child. Further, we can also go the other way: $\lfloor (m-1)/2 \rfloor$ is the parent. The imperative pseudo-code for these operations follows.

def PARENT(m) :

1. **return** $\lfloor (m-1)/2 \rfloor$

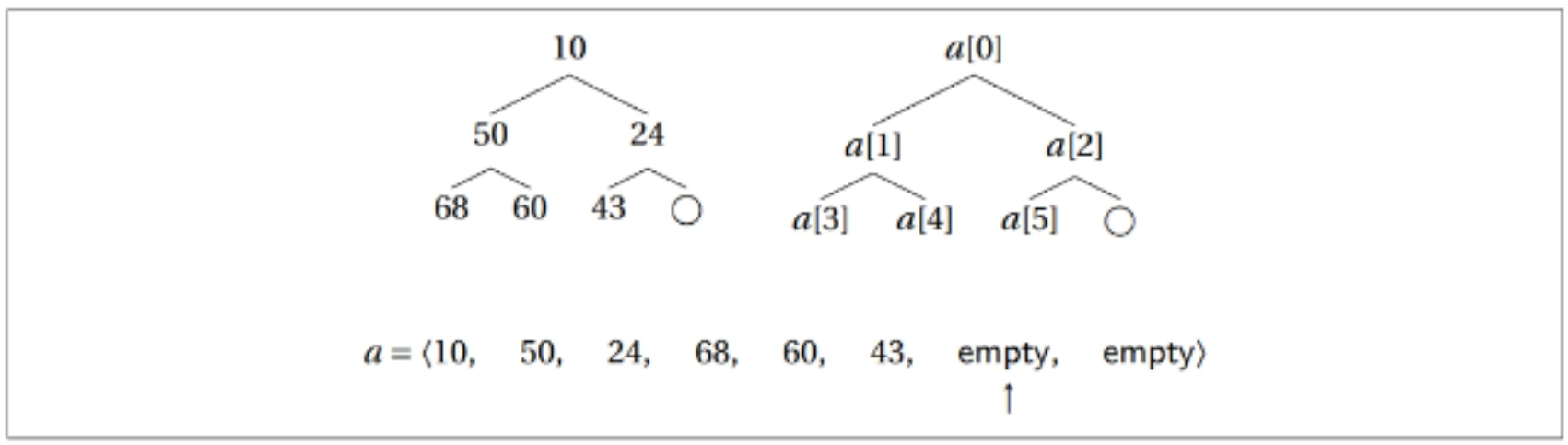


Figure 10.3: Representing a left-complete binary min-heap as an array

def LEFTCHILD(m) :

1. **return** $2 \times m + 1$

def RIGHTCHILD(m) :

1. **return** $2 \times m + 2$

Since we need two values, the array and the index identifying the next available leaf, in this section, we will represent the heap as a structure: $h = \text{heap}(a, e)$. The array portion will be accessed via h.a, and the index will be accessed via h.e. Given an array-based representation of a left-complete binary min-heap, consider the following pseudo-code implementations of the heap operations.

def *findMinHeap*(h) :

1. $a, e \leftarrow h.\text{a}, h.\text{e}$
2. **if** $e = 0$:
3. error "empty heap"
4. **else** :
5. **return** $a[0]$

def *removeMinHeap*(h) :

1. $a, e \leftarrow h.\text{a}, h.\text{e}$
2. **if** $e = 0$:
3. error "empty heap"
4. **else** :
5. $e \leftarrow e - 1$
6. $h.\text{e} \leftarrow e$
7. $a[0] \leftarrow a[e]$
8. SIFTDOWN($a, e, 0$)

def *insertMinHeap*(h, v) :

1. $a, e \leftarrow h.\text{a}, h.\text{e}$
2. **if** $e \geq |a|$:
3. error "heap overflow"
4. **else** :
5. $a[e] \leftarrow v$
6. SIFTUP(a, e)
7. $h.\text{e} \leftarrow e + 1$

The support functions for the heap operations follow. Note that, although conceptually *removeMinHeap* is no more complicated than *insertMinHeap*, that the actual implementation involves more details since one or both of the children might not exist.

def FIRSTOFTHREE(a, e, m) :

1. $\ell, r \leftarrow$ LEFTCHILD(m), RIGHTCHILD(m)
2. **if** $\ell < e$ **and** $a[\ell] \leq a[m]$:
3. $\quad m \leftarrow \ell$
4. **if** $r < e$ **and** $a[r] \leq a[m]$:
5. $\quad m \leftarrow r$
6. **return** m

def SIFTDOWN(a, e, m) :

1. $j \leftarrow$FIRSTOFTHREE(a, e, m)
2. **while** $j \neq m$:
3. $\quad$ **swap** $a[m], a[j]$
4. $\quad m \leftarrow j$
5. $\quad j \leftarrow$FIRSTOFTHREE(a, e, m)

def SIFTUP(a, m) :

1. **while** $m > 0$ **and not** $a[$PARENT(m)$] \leq a[m]$:
2. $\quad$ **swap** $a[m], a[$PARENT(m)$]$
3. $\quad m \leftarrow$PARENT(m)

Chapter 11

Tree-Based Sorting**

11.1 Introduction to Tree-Based Sorting

We have seen that operations on trees are typically faster than operations on sequences such as lists and arrays. We leverage this idea and develop sorting algorithms that are asymptotically faster than the naive algorithms.

11.2 Fast Insertion Using a Tree

A binary search tree contains its elements in order. Further, if the binary search tree is balanced then insertion is very fast: $\Theta(\log(n))$. Thus, we can implement a fast sorting algorithm that is similar to insertion sort by repeatedly inserting into a binary search tree and then reading the tree into a sequence. When using a binary search tree, we must take special care to deal with duplicates. Further, although reading the tree into a sequence does not sound too bad, we must take care to do so efficiently.

11.3 Fast Selection Using a Tree

A heap can be used to rapidly access and remove the maximum element. Therefore, selection is very fast: $\Theta(\log(n))$.[1] Thus, we can implement a fast sorting algorithm that is similar to selection sort by first turning the sequence into a heap and then repeatedly selecting the maximum element from the heap and placing it where it belongs in the result sequence. Since there are both functional and imperative heap formulations, we discuss both. Note that in either case the heap-based implementation closely resembles the selection sort algorithm.

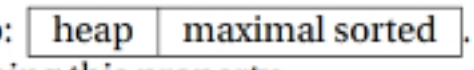
11.3.1 A Functional Formulation

The functional formulation starts by converting the list to a heap. Then the maximum is repeatedly extracted and put onto a list. When the heap is exhausted the list with the sorted elements is returned.

[1] Even though finding the maximum only takes constant time, removal is an important part of selection.

$$
\begin{array}{lcl}
hSortAccum(\mathsf{emptyHeap}; a) & = & a \\
hSortAccum(h; a) & = & \text{let } (m, o) = (findMaxHeap(h), removeMaxHeap(h)) \text{ in } hSortAccum(o; m :: a) \\
 & & \\
hSort(xs) & = & hSortAccum(heapFromList(xs); [])
\end{array}
$$

11.3.2 An Imperative Formulation

The imperative formulation starts by converting the array, in place, into an array-based heap. Then the maximum is repeatedly extracted and placed where it belongs. The heap shrinks just enough each time to make space for the element at the end. This algorithm is characterized by a diagram similar to that of selection sort. In general, we understand the part of the array to the right of the cut contains maximal sorted elements but the elements in the part to the left are understood as a heap: 

. The nature of sorting is then to pull the cut from right to left while maintaining this property.

def HEAPSORT(a) :
1. HEAPFROMARRAY(a)
2. **for** $m \leftarrow |a| - 1$ **downto** 1 :
3. $\quad v \leftarrow findMaxHeap(a)$
4. $\quad removeMaxHeap(a, m)$
5. $\quad a[m] \leftarrow v$

For an efficient implementation, we pierce the heap abstraction and explicitly pass both the array and the index when necessary, where the index here identifies the last leaf rather than the next available leaf.

def $findMaxHeap(a)$:
1. **return** $a[0]$

def $removeMaxHeap(a, d)$:
1. $a[0] \leftarrow a[d]$
2. SIFTDOWN$'(a, d, 0)$

An easy way to implement HEAPFROMARRAY is to repeatedly insert elements from the array into the heap.

Chapter 12

Divide and Conquer Sorting**

12.1 Introduction to Divide and Conquer Sorting

We now apply the divide and conquer approach for algorithm design to sorting algorithms. Division entails dividing the sequence into smaller subsequences. Once again, it will turn out that dividing the sequence non-structurally into two roughly equal sized pieces will be the efficient approach. The conquer phase merely corresponds to recursively sorting those subsequences. For the combine phase it is necessary to explain how to put the sorted subsequences together to get the sorted sequence that includes all the elements of the original sequence.

12.2 Merge Sort

We can view the merge sort algorithm as a generalization of the insertion sort algorithm. Instead of breaking the sequence into the head and the tail, we break it into two roughly equal sized subsequences. (We will see that there is more than one way to do this breaking.) Since both subsequences are non-trivial, it is necessary to recursively sort both of them. Finally, instead of inserting a single element, we merge the two sorted subsequences.

One way of breaking a sequence in two is to break it by chopping it in half. Another way is to put all the even numbered elements in one subsequence and all the odd numbered elements in another sequence. (It doesn't matter if the elements themselves are even or odd.) We will take this second approach since it works well for lists.

$$
\begin{array}{lcl}
d([]) & = & ([],[]) \\
d([x]) & = & ([x],[]) \\
d(x_0 :: x_1 :: xs) & = & \text{let } (b_0, b_1) = d(xs) \text{ in } (x_0 :: b_0, x_1 :: b_1)
\end{array}
$$

We have already encountered the merge algorithm in chapter 9. We repeat it here.

$$
\begin{array}{lcl}
[] \bowtie ys & = & ys \\
xs \bowtie [] & = & xs \\
(x :: xs) \bowtie (y :: ys) & = & \begin{cases} x :: (xs \bowtie (y :: ys)) & \text{if } x < y \\ y :: ((x :: xs) \bowtie ys) & \text{if } x > y \\ x :: y :: (xs \bowtie ys) & \text{if } x = y \end{cases}
\end{array}
$$

Given the divide and the merge operations above, merge sort itself can now be expressed.

$$
\begin{array}{lcl}
mSort([]) & = & [] \\
mSort([x]) & = & [x] \\
mSort(xs) & = & \text{let } (b_0, b_1) = d(xs) \text{ in } mSort(b_0) \bowtie mSort(b_1)
\end{array}
$$

Note that the merge sort algorithm has two base cases. Why?

Example

Consider the sequence $[4,1,3,2]$. Below we compute, but omit the some of the details.

$$
\begin{array}{lcl}
mSort([4,1,3,2]) & = & \text{let } (b_0, b_1) = d([4,1,3,2]) \text{ in } mSort(b_0) \bowtie mSort(b_1) \\
& = & mSort([4,3]) \bowtie mSort([1,2]) \\
& = & (mSort([4]) \bowtie mSort([3])) \bowtie mSort([1,2]) \\
& = & ([4] \bowtie mSort([3])) \bowtie mSort([1,2]) \\
& = & ([4] \bowtie [3]) \bowtie mSort([1,2]) \\
& = & (3 :: ([4] \bowtie [])) \bowtie mSort([1,2]) \\
& = & (3 :: [4]) \bowtie mSort([1,2]) \\
& = & [3,4] \bowtie (mSort([1]) \bowtie mSort([2])) \\
& = & [3,4] \bowtie ([1] \bowtie mSort([2])) \\
& = & [3,4] \bowtie ([1] \bowtie [2]) \\
& = & [3,4] \bowtie (1 :: ([] \bowtie [2])) \\
& = & [3,4] \bowtie (1 :: [2]) \\
& = & [3,4] \bowtie [1,2] \\
& = & [1,2,3,4]
\end{array}
$$

12.2.1 Time Complexity

For merge sort we will focus on the number of comparisons to characterize the time complexity. Since there are no comparisons performed in d, we can omit its analysis. The worst case for $\bowtie$ is when we hit the third and fourth lines, but that can only happen when both lists are non-empty. Thus we get the following recurrence.

$$
\begin{array}{lcl}
T_{\bowtie}(0) & = & 0 \\
T_{\bowtie}(1) & = & 0 \\
T_{\bowtie}(n) & = & 1 + T_{\bowtie}(n-1)
\end{array}
$$

We see that the solution for $n > 0$ is $T_{\bowtie}(n) = n - 1$.

For merge sort, the number of comparisons is the number of comparisons of the recursive calls plus the number for the top level merge. Thus we get the following recurrence.

$$\begin{aligned} T_{mSort}(0) &= 0 \\ T_{mSort}(1) &= 0 \\ T_{mSort}(n) &= T_{mSort}(\lfloor n/2 \rfloor) + T_{mSort}(\lceil n/2 \rceil) + T_{\bowtie}(n) \end{aligned}$$

For the moment, we will simplify by assuming that $n = 2^m$. And so, we will focus on the following simplified recurrence.

$$\begin{aligned} T(2^0) &= 0 \\ T(2^m) &= 2T(2^{m-1}) + (2^m - 1) \end{aligned}$$

Solving the Recurrence by Iteration

$$\begin{aligned} T(2^m) &= 2T(2^{m-1}) + (2^m - 1) \\ &= 2(2T(2^{m-2}) + (2^{m-1} - 1)) + (2^m - 1) \\ &= 2^2 T(2^{m-2}) + (2^m - 2) + (2^m - 1) \\ &= 2^2(2T(2^{m-3}) + (2^{m-2} - 1)) + (2^m - 2) + (2^m - 1) \\ &= 2^3 T(2^{m-3}) + (2^m - 2^2) + (2^m - 2^1) + (2^m - 2^0) \\ &= 2^k T(2^{m-k}) + k \cdot 2^m - \sum_{i=0}^{k-1} 2^i \leftarrow \text{Identify the pattern} \end{aligned}$$

Once we have identified the pattern, let $k = m$. Thus we can simplify as follows.

$$\begin{aligned} T(2^m) &= 2^m T(2^0) + m \cdot 2^m - \sum_{i=0}^{m-1} 2^i \\ &= m \cdot 2^m - (2^m - 1) \\ &= m \cdot 2^m - 2^m + 1 \end{aligned}$$

Expressing this result in terms of n, we have $T(n) = \lg(n) \cdot n - n + 1$. And asymptotically, we have $T(n) = \Theta(n \log(n))$. This suggests that $T_{mSort}(n) = \Theta(n \log(n))$. We will explore the details $T_{mSort}(n)$ in the exercises.

Solving the Recurrence by Induction

Theorem 16. *For any $m \in \mathbb{N}$, $T(2^m) = m \cdot 2^m - 2^m + 1$.*

Proof. By mathematical induction.

- Observe that $T(2^0) = 0 = 0 \cdot 1 - 1 + 1 = 0 \cdot 2^0 - 2^0 + 1$.
- Assume that $T(2^k) = k \cdot 2^k - 2^k + 1$.

$$\begin{aligned} T(2^{k+1}) &= 2T(2^k) + (2^{k+1} - 1) \\ &= 2(k \cdot 2^k - 2^k + 1) + (2^{k+1} - 1) \\ &= k \cdot 2^{k+1} - 2^{k+1} + 2 + 2^{k+1} - 1 \\ &= (k+1)2^{k+1} - 2^{k+1} + 1 \end{aligned}$$

□

12.2.2 Imperative Formulation

The entry point is the procedure MSORT which calls MSORTHELP initializing b to a copy of a, ℓ to the first index of a, and h to the last index of a. The procedure MSORTHELP is sorting the region in a corresponding to the slice $a[\ell : h]$. The base cases are when $\ell > h$ (the empty sequence) and $\ell = h$ (the singleton sequence), where no action is taken. Thus, the case with the recursive calls is when $\ell < h$. Here in the imperative setting, dividing the array does involve chopping it in half, and is easily computed by calculating the middle index, which is put into m. Note the asymmetry, since the value at the middle index must be included in one of the regions.

def MSORT(a) :

1. MSORTHELP($a, a[:], 0, |a| - 1$)

def MSORTHELP(a, b, ℓ, h) :

1. **if** $\ell < h$:
2. $\quad m \leftarrow \ell + \lfloor (h - \ell)/2 \rfloor$
3. $\quad$ MSORTHELP(a, b, ℓ, m)
4. $\quad$ MSORTHELP($a, b, m + 1, h$)
5. $\quad$ MERGE(a, b, ℓ, m, h)

To finish the sorting algorithm, we must actually say how to merge the subsequences in the array. The algorithm is tedious but straightforward. We copy the elements from the array a to the array b in order, and then copy that portion of b back to a.

def MERGE(a, b, ℓ, m, h) :

1. $i \leftarrow \ell$ # index into first sorted region
2. $j \leftarrow m + 1$ # index into second sorted region
3. $k \leftarrow \ell$ # index into destination region
4. **while** $i \leq m$ **and** $j \leq h$:
5. $\quad$ **if** $a[i] < a[j]$:
6. $\quad\quad b[k] \leftarrow a[i]; i \leftarrow i + 1; k \leftarrow k + 1$
7. $\quad$ **else if** $a[i] > a[j]$:
8. $\quad\quad b[k] \leftarrow a[j]; j \leftarrow j + 1; k \leftarrow k + 1$
9. $\quad$ **else** :
10. $\quad\quad b[k] \leftarrow a[i]; i \leftarrow i + 1; k \leftarrow k + 1$
11. $\quad\quad b[k] \leftarrow a[j]; j \leftarrow j + 1; k \leftarrow k + 1$
12. **while** $i \leq m$:
13. $\quad b[k] \leftarrow a[i]; i \leftarrow i + 1; k \leftarrow k + 1$
14. **while** $j \leq h$:
15. $\quad b[k] \leftarrow a[j]; j \leftarrow j + 1; k \leftarrow k + 1$
16. **for** $k \leftarrow \ell$ **to** h :
17. $\quad a[k] \leftarrow b[k]$

12.3 Quick Sort

We can view quick sort as a generalization of selection sort. Instead of placing a single element at the end, we place an entire subsequence. Such a placement is possible if we require that all the elements of

the subsequence are greater than a given element. It is customary to call the element that the others are compared to the *pivot*. Of course, there is also a subsequence whose elements are all less than (or equal to) the pivot. Identifying these subsequences is called *partitioning*. Since the subsequences that are less than or greater than the pivot could contain multiple distinct elements, it is necessary to recursively sort them. The combination phase is easy, since it merely involves placement.

$$\begin{array}{lcl} qSort([]) & = & [] \\ qSort(x :: xs) & = & qSort([y \in xs \mid y < x]) + [y \in x :: xs \mid y = x] + qSort([y \in xs \mid y > x]) \end{array}$$

Example

Consider the sequence $[3,1,4,2]$. We compute the result of applying the quick sort algorithm to the sequence below.

$$\begin{array}{lcl} qSort([3,1,4,2]) & = & qSort([1,2]) + [3] + qSort([4]) \\ & = & (qSort([]) + [1] + qSort([2])) + [3] + qSort([4]) \\ & = & ([] + [1] + qSort([2])) + [3] + qSort([4]) \\ & = & ([1] + qSort([2])) + [3] + qSort([4]) \\ & = & ([1] + (qSort([]) + [2] + qSort([]))) + [3] + qSort([4]) \\ & = & ([1] + ([] + [2] + qSort([]))) + [3] + qSort([4]) \\ & = & ([1] + ([2] + qSort([]))) + [3] + qSort([4]) \\ & = & ([1] + ([2] + [])) + [3] + qSort([4]) \\ & = & ([1] + [2]) + [3] + qSort([4]) \\ & = & [1,2] + [3] + qSort([4]) \\ & = & [1,2,3] + qSort([4]) \\ & = & [1,2,3] + (qSort([]) + [4] + qSort([])) \\ & = & [1,2,3] + ([] + [4] + qSort([])) \\ & = & [1,2,3] + ([4] + qSort([])) \\ & = & [1,2,3] + ([4] + []) \\ & = & [1,2,3] + [4] \\ & = & [1,2,3,4] \end{array}$$

12.3.1 Quicker Quick Sorts

Our initial quick sort algorithm could be much faster. For example, the partitioning algorithm above makes three passes. To what extent can we reduce that? Further, although combination is easy, could it involve even less computation?

We can reduce the number of passes from three to two by putting all the elements that are equal to the pivot into the first subsequence.

$$\begin{array}{lcl} qSort([]) & = & [] \\ qSort(x :: xs) & = & qSort([y \in xs \mid y \leq x]) + [x] + qSort([y \in xs \mid y > x]) \end{array}$$

We can reduce the number of passes from two to one by writing a separate partition function.

$$
\begin{array}{lcl}
p_(x, ys) & = & \hat{p}(x, ys, [], []) \\
\hat{p}(x, [], \ell, g) & = & (\ell, g) \\
\hat{p}(x, y :: ys, \ell, g) & = & \begin{cases} \hat{p}(x, ys, y :: \ell, g) & \text{if } y \leq x \\ \hat{p}(x, ys, \ell, y :: g) & \text{if } y > x \end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
qSort([]) & = & [] \\
qSort(x :: xs) & = & \text{let } (\ell, g) = p(x, xs) \text{ in } qSort(\ell) + [x] + qSort(g)
\end{array}
$$

We can even eliminate the append operations (+) by introducing an accumulation parameter. The invariant is $qSortAccum(xs; a) = qSort(xs) + a$.

$$
\begin{array}{lcl}
qSortAccum([]; a) & = & a \\
qSortAccum(x :: xs; a) & = & \text{let } (\ell, g) = p(x, xs) \text{ in } qSortAccum(\ell; x :: qSortAccum(g; a))
\end{array}
$$

Seeing that we can eliminate a linear time operator and use only a constant time operator, we suspect that we can make the combining phase take no effort if we reformulate the algorithm so that partitioning is done in-place. Recall that an *in-place* algorithm uses no additional space. The array that represents the sequence is modified so that different regions of the array correspond to the subsequences.

This idea is expressed as follows, where the sequence is represented as an array. Here we don't change the size of the array but focus on regions characterized by slicing parameters.

def QSORT(a) :

1. QSORTHELP($a, 0, |a| - 1$)

def QSORTHELP(a, ℓ, h) :

1. **if** $\ell < h$:
2. $\quad m \leftarrow$ PARTITION(a, ℓ, h)
3. $\quad$ QSORTHELP($a, \ell, m - 1$)
4. $\quad$ QSORTHELP($a, m + 1, h$)

The procedure QSORTHELP is sorting the region in a corresponding to the slice $a[\ell : h]$. The base case is when $\ell > h$, where no action is taken. Thus the case with the recursive calls would be when $\ell \leq h$, but by merely changing the test to $\ell < h$ we add a base case for the singleton region as well. The procedure PARTITION returns the index of the pivot. That index is put into m and is used to characterize both the region of the smaller elements and the region of the larger elements. Thus the first recursive call is sorting the elements between indices ℓ and $m-1$ inclusive, and the second recursive call is sorting the elements between indices $m+1$ and h inclusive. Note that like the functional pseudo-code above, the pivot sits in the middle and is not sorted.

To finish the sorting algorithm, we must actually say how to partition the sequence in place. There are two popular approaches, and we consider both of them.

Fast and Slow

The first approach we will consider is easier to implement correctly. There is a fast cut and a slow cut. The slow cut identifies a region from the beginning of the sequence to that cut. In this region are the values less than or equal to the pivot. The fast cut identifies a region from the slow cut to that cut. In this region are the values greater than the pivot. The conjunction of the properties of the regions is the invariant that entails correctness. In this algorithm, the pivot is safely at the end of the sequence.

less than or equal	greater than	unconstrained	pivot

The algorithm moves the fast cut expanding the greater region. It checks to prevent such an expansion from including a value too small. If it finds such a value, it expands the size of the smaller region so that it encroaches on the greater region but then swaps the values so that the small region continues to have smaller values and the greater region continues to have greater values. The pseudo-code now follows.

def PARTITION(a, ℓ, h) :

1. $x \leftarrow a[h]$ # pivot
2. $i \leftarrow \ell - 1$ # slow mark, cut between i and $i+1$
3. $j \leftarrow \ell$ # fast mark, cut between $j-1$ and j
4. **while** $j < h$:
5. **if** $a[j] \leq x$:
6. $i \leftarrow i+1$ # move the slow mark forward
7. **swap** $a[i], a[j]$ # restore invariant
8. $j \leftarrow j+1$ # always move the fast mark forward
9. **swap** $a[i+1], a[h]$
10. **return** $i+1$

Example

We apply the partition algorithm above to a specific array. The green is the fast cut and the red is the slow cut. The pivot is singled out in every line. Note that it is at the end in every line but the last; there it is swapped with the element just past the slow cut.

1	5	6	2	3	8	7	4
1	5	6	2	3	8	7	4
1	5	6	2	3	8	7	4
1	5	6	2	3	8	7	4
1	5	6	2	3	8	7	4
1	5	6	2	3	8	7	4
1	2	6	5	3	8	7	4
1	2	6	5	3	8	7	4
1	2	3	5	6	8	7	4
1	2	3	5	6	8	7	4
1	2	3	5	6	8	7	4
1	2	3	5	6	8	7	4
1	2	3	4	6	8	7	5

Moving Toward Each Other

In the previous example we see the first element being swapped with itself! The second approach we will consider is more efficient because no self swaps occur. There are two cuts: a left cut and a right cut. The cuts are moving toward each other from opposite ends of the sequence. The left cut identifies a region from just after the pivot to that cut. In this region are the values less than or equal to the pivot. The right cut identifies a region from that cut to the end of the sequence. In this region are the values greater than the pivot. The conjunction of the properties of the regions is the invariant that entails correctness.

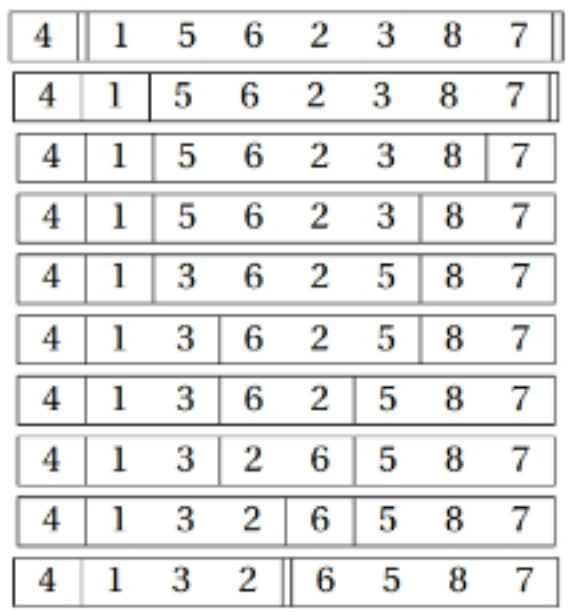
pivot	less than or equal	unconstrained	greater than

The algorithm tries to move one of the cuts toward the other. If it cannot, that entails that there is an element too large just after the left cut and an element too small just before the right cut. When that happens, we can swap them and then continue to make progress.

def PARTITION(a, ℓ, h) :

```
1.  x ← a[ℓ] # pivot
2.  i ← ℓ + 1
3.  j ← h
4.  while i ≤ j :
5.      if a[i] ≤ x :
6.          i ← i + 1
7.      else if x < a[j] :
8.          j ← j − 1
9.      else :
10.         swap a[i], a[j]
11. swap a[ℓ], a[i − 1]
12. return i − 1
```

Example



12.3.2 Time Complexity

We get the following recurrence for quick sort, where p is a free variable indicating what happened with the partition.

$$\begin{aligned} T_{qSort}(0) &= 0 \\ T_{qSort}(n) &= (n-1) + T_{qSort}(p) + T_{qSort}(n-p-1) \end{aligned}$$

Adjusting p yields best and worst case asymptotic results. In the best case, $T_{qSort}(n) = \Theta(n\log(n))$. In the worst case, $T_{qSort}(n) = \Theta(n^2)$. Often we stop with the worst case, but that does not seem to reflect what users of the algorithm typically experience. Therefore we consider the expected or average case complexity.

For the average case, we assume that the set of possible inputs are all permutations of n distinct numbers (e.g., the numbers from 1 to n), and that each permutation is equally likely. Thus the probability of getting a particular pivot is $1/n$. And so the average case recurrence is the following.

$$\begin{aligned} T_{qSort}(0) &= 0 \\ T_{qSort}(n) &= (n-1) + \frac{1}{n}\sum_{p=0}^{n-1}(T_{qSort}(p) + T_{qSort}(n-p-1)) \end{aligned}$$

It turns out that this recurrence requires a little extra effort to solve. If we modify the linear term, so that instead of $n-1$ we have $n+1$, the recurrence becomes easier. So consider the modified recurrence.

$$\begin{aligned} T(0) &= 0 \\ T(n) &= (n+1) + \frac{1}{n}\sum_{p=0}^{n-1}(T(p) + T(n-p-1)) \end{aligned}$$

Simplifying yields the following recurrence with full history.

$$\begin{aligned} T(0) &= 0 \\ T(n) &= (n+1) + \frac{2}{n}\sum_{p=0}^{n-1} T(p) \end{aligned}$$

Let us focus on the second line. Multiply both sides by n.

$$nT(n) = n(n+1) + 2\sum_{p=0}^{n-1} T(p)$$

When n is sufficiently large, consider the equation for $n-1$.

$$(n-1)T(n-1) = n(n-1) + 2\sum_{p=0}^{n-2} T(p)$$

Now subtract.

$$\begin{array}{lcll} nT(n)-(n-1)T(n-1) & = & n(n+1)+2\sum_{p=0}^{n-1}T(p)-n(n-1)-2\sum_{p=0}^{n-2}T(p) & \Rightarrow \\ nT(n)-(n-1)T(n-1) & = & n(n+1)-n(n-1)+2T(n-1) & \Rightarrow \\ nT(n)-(n-1)T(n-1) & = & 2n+2T(n-1) & \Rightarrow \\ nT(n) & = & 2n+(n+1)T(n-1) & \end{array}$$

In chapter 7 we renamed $nT(n)$. Here we cannot do that yet because the left-hand side and the right-hand side do not have the same form, so we divide by $n(n+1)$ and derive the following recurrence.

$$\begin{array}{lcl} T(1) & = & 2 \\ \dfrac{T(n)}{n+1} & = & \dfrac{2}{n+1}+\dfrac{T(n-1)}{n} \end{array}$$

At this point, we are in a position to rename. Let $\hat{T}(n)=\dfrac{T(n)}{n+1}$. Thus we derive one more variation of the recurrence.

$$\begin{array}{lcl} \hat{T}(1) & = & 1 \\ \hat{T}(n) & = & \dfrac{2}{n+1}+\hat{T}(n-1) \end{array}$$

Now we use iteration on this derived recurrence.

$$\begin{array}{lcl} \hat{T}(n) & = & \dfrac{2}{n+1}+\hat{T}(n-1) \\ & = & \dfrac{2}{n+1}+\dfrac{2}{n}+\hat{T}(n-2) \\ & = & \dfrac{2}{n+1}+\dfrac{2}{n}+\dfrac{2}{n-1}+\hat{T}(n-3) \\ & = & \dfrac{2}{n+1}+\cdots+\dfrac{2}{n-(k-2)}+\hat{T}(n-k) \leftarrow \text{Identify the pattern} \end{array}$$

Once we have identified the pattern, let $k=n-1$.
Thus we derive that $\hat{T}(n)=2\left(\dfrac{1}{n+1}+\cdots+\dfrac{1}{3}\right)+\hat{T}(1)$.
Simplifying we get $\hat{T}(n)=2(H_{n+1}-3/2)+1$, and so $T(n)=2(n+1)(H_{n+1}-1)$. Hence, $T(n)=\Theta(n\log(n))$.
This suggests that for the average case $T_{qSort}(n)=\Theta(n\log(n))$.

Chapter 13

Selection**

13.1 Introduction to Selection

Consider the selection problem.

Problem 10. *Given a non-empty sequence s of orderable values and an index i, what is s'_i where s' is the sorted sequence derived from s?*

Selection picks out the ith element from the corresponding sorted sequence. Thus selection is a general algorithm that can be used to calculate the minimum, the maximum, the median value, and more. Assuming the array interface for lists, we can use the problem statement to write down the definition algorithmically: $select(\ell, i) = qSort(\ell)[i]$.

Examples

$$
\begin{array}{lcll}
select([3,1,5,2,4],0) & = & 1 & \text{the minimum} \\
select([3,1,5,2,4],4) & = & 5 & \text{the maximum} \\
select([3,1,5,2,4],2) & = & 3 & \text{the median}
\end{array}
$$

13.2 Deriving an Algorithm

The algorithm in section 13.1 is simple, but its performance can't be any better than $\Theta(n\log(n))$. We wonder if we had an algorithm that doesn't call sort whether we could get better asymptotic performance. After all, computing the minimum and the maximum require only $\Theta(n)$ time. Let's view the algorithm obtained from the problem statement as merely definitional, and derive[1] from it a recursive algorithm.

First we derive the case when the list is empty.

$$
\begin{array}{lcl}
select([], n) & = & qSort([])[n] \\
 & = & qSort([])!!n \\
 & = & []!!n \\
 & = & \text{error “Bad index”}
\end{array}
$$

[1] The operation !! is the indexing operation for lists.

Then we derive the case when the list is non-empty.

$$
\begin{aligned}
select(x :: xs, n) &= qSort(x :: xs)[n] \\
&= qSort(x :: xs)!!n \\
&= (qSort(\ell) + (s + qSort(g)))!!n \\
&\quad \text{where } \ell = [y \in xs \mid y < x],\ s = [y \in x :: xs \mid y = x],\ g = [y \in xs \mid y > x] \\
&= \begin{cases} qSort(\ell)!!n & \text{if } n < |qSort(\ell)| \\ (s + qSort(g))!!(n - |qSort(\ell)|) & \text{if } |qSort(\ell)| \leq n \end{cases} \\
&\quad \text{where } \ell = [y \in xs \mid y < x],\ s = [y \in x :: xs \mid y = x],\ g = [y \in xs \mid y > x] \\
&= \begin{cases} qSort(\ell)!!n & \text{if } n < |\ell| \\ (s + qSort(g))!!(n - |\ell|) & \text{if } |\ell| \leq n \end{cases} \\
&\quad \text{where } \ell = [y \in xs \mid y < x],\ s = [y \in x :: xs \mid y = x],\ g = [y \in xs \mid y > x] \\
&= \begin{cases} qSort(\ell)!!n & \text{if } n < |\ell| \\ s!!(n - |\ell|) & \text{if } |\ell| \leq n \text{ and } n - |\ell| < |s| \\ qSort(g)!!((n - |\ell|) - |s|) & \text{if } |\ell| \leq n \text{ and } |s| \leq (n - |\ell|) \end{cases} \\
&\quad \text{where } \ell = [y \in xs \mid y < x],\ s = [y \in x :: xs \mid y = x],\ g = [y \in xs \mid y > x] \\
&= \begin{cases} qSort(\ell)!!n & \text{if } n < |\ell| \\ s!!(n - |\ell|) & \text{if } |\ell| \leq n < |\ell| + |s| \\ qSort(g)!!((n - |\ell|) - |s|) & \text{if } |\ell| + |s| \leq n \end{cases} \\
&\quad \text{where } \ell = [y \in xs \mid y < x],\ s = [y \in x :: xs \mid y = x],\ g = [y \in xs \mid y > x] \\
&= \begin{cases} select(\ell, n) & \text{if } n < |\ell| \\ x & \text{if } |\ell| \leq n < |\ell| + |s| \\ select(g, n - (|\ell| + |s|)) & \text{if } |\ell| + |s| \leq n \end{cases} \\
&\quad \text{where } \ell = [y \in xs \mid y < x],\ s = [y \in x :: xs \mid y = x],\ g = [y \in xs \mid y > x]
\end{aligned}
$$

The recursive algorithm now follows.

$$
\begin{aligned}
select([], i) &= \text{error "Bad index"} \\
select(x :: xs, i) &= \begin{cases} select(\ell, i) & \text{if } i < |\ell| \\ x & \text{if } |\ell| \leq i < |\ell| + |s| \\ select(g, i - (|\ell| + |s|)) & \text{if } |\ell| + |s| \leq i \end{cases} \\
&\quad \text{where } \ell = [y \in xs \mid y < x],\ s = [y \in x :: xs \mid y = x],\ g = [y \in xs \mid y > x]
\end{aligned}
$$

Examples

We extend the examples in section 13.1 to include some of the computational detail. The partitioning and the case analysis calculations are omitted. We start with finding the minimum.

$$
\begin{aligned}
select([3,1,5,2,4],0) &= select([1,2],0) \\
&= 1
\end{aligned}
$$

Next we find the maximum.

$$
\begin{aligned}
select([3,1,5,2,4],4) &= select([5,4],1) \\
&= 5
\end{aligned}
$$

We identify the median immediately since it is the pivot.

$$
select([3,1,5,2,4],2) = 3
$$

A Variation

If we use the partition function from chapter 12 we get the following variation.

$$
\begin{array}{lcl}
\mathit{select}([], i) & = & \text{error "Bad index"} \\
\mathit{select}(x :: xs, i) & = & \begin{cases} \mathit{select}(\ell, i) & \text{if } i < |\ell| \\ x & \text{if } i = |\ell| \\ \mathit{select}(g, i - (|\ell| + 1)) & \text{if } |\ell| + 1 \leq i \end{cases} \\
 & & \text{where } (\ell, g) = p(x, xs)
\end{array}
$$

13.3 Time Complexity

For the time complexity analysis, we have in mind the slightly better algorithm from section 13.2 that does the partitioning in a single pass. We will measure performance by counting comparisons; we note that partitioning in a single pass requires $n-1$ comparisons for a sequence of length n. We assume that the index is valid. Note that the smallest valid list is a singleton list, in which case no comparisons are made. Thus $T(1) = 0$. The time complexity inherits a lot from quick sort. In the worst case, the pivot element can be an extreme element and the recursive call is on a list that is just one smaller. Thus $T(n) = n - 1 + T(n-1)$.

$$
\begin{array}{rcl}
T(n) & = & (n-1) + T(n-1) \\
 & = & (n-1) + (n-2) + T(n-2) \\
 & = & (n-1) + (n-2) + (n-3) + T(n-3) \\
 & = & (n-1) + \cdots + (n-k) + T(n-k) \leftarrow \text{Identify the pattern}
\end{array}
$$

Let $k = n-1$.

$$
\begin{array}{rcl}
T(n) & = & (n-1) + \cdots + (n-(n-1)) + T(n-(n-1)) \\
 & = & (n-1) + \cdots + 1 + T(1) \\
 & = & (n-1) + \cdots + 0 \\
 & = & \sum_{i=0}^{n-1} i \\
 & = & n(n-1)/2
\end{array}
$$

So we find that this algorithm still has the awful property of quick sort: the worst case performance is $\Theta(n^2)$.

For our best case analysis we will "ignore" the index parameter; rather only the size of the list will dictate when to stop — when the list is a singleton. In contrast to the worst case, the list splits into two roughly equal size pieces. We take a simplified approach here and express this fact as $T(n) = n - 1 + T(n/2)$.

Assume $n = 2^m$

$$
\begin{array}{rcl}
T(2^m) & = & (2^m - 1) + T(2^{m-1}) \\
 & = & (2^m - 1) + (2^{m-1} - 1) + T(2^{m-2}) \\
 & = & (2^m - 1) + (2^{m-1} - 1) + (2^{m-2} - 1) + T(2^{m-3}) \\
 & = & \sum_{i=0}^{k-1} (2^{m-i} - 1) + T(2^{m-k}) \leftarrow \text{identify pattern}
\end{array}
$$

Let $k = m$

$$\begin{aligned}
T(2^m) &= \sum_{i=0}^{m-1} (2^{m-i} - 1) + T(2^0) \\
&= \sum_{i=0}^{m-1} (2^{m-i} - 1) + 0 \\
&= \sum_{i=0}^{m-1} 2^{m-i} - \sum_{i=0}^{m-1} 1 \\
&= \sum_{i=0}^{m-1} 2^{m-i} - m \\
&= \sum_{i=0}^{m} 2^i - 1 - m \\
&= (2^{m+1} - 1) - 1 - m \\
&= 2 \times 2^m - 2 - m
\end{aligned}$$

So $T(n) = 2n - 2 - \lg(n)$.

Note that here we do even better than quick sort. Like computing the maximum, the best case performance is merely $\Theta(n)$. Further, the average case of this algorithm is the same as the best case and is also $\Theta(n)$. We leave that calculation as an exercise.

Chapter 14

Lower Bounds for Sorting**

14.1 Introduction to Lower Bounds for Sorting

In chapters 9, 11, and 12, we see a number of sorting algorithms. When we first consider sorting, the naive algorithms have a worst case performance of $\Theta(n^2)$. At first we may resign ourselves to that kind of performance, but shortly thereafter we encounter more sophisticated sorting algorithms with $\Theta(n \log(n))$ worst case performance. Can we go any lower? What is the best possible worst case performance?

For sequential sorting algorithms, we can make the following simple argument: To sort a sequence of length n, we must at least look at every single element of the sequence. If not, how can we be confident that the algorithm has the unseen element in the right place! Thus the worst case performance of any sequential sorting algorithm must be $\Omega(n)$.

But it is not clear whether or not that is a tight lower bound. We will see that we can derive more nuanced results by introducing an alternative tree-based model for sorting computations. The next section introduces this model, and the section after that discusses the results obtained from using that model.

14.2 An Alternative Computational Model

The definitions for the alternative tree-based computational model are first. After that are some examples.

Here we start to define the sorting tree model of computation: the *questionnaire*. An internal node indicates what elements of the sequence to compare. A leaf is a permutation[1] that indicates how to change the ordering of the input sequence so that the output sequence is in order. We start by recursively defining the structure of the tree, which is called a *pre-questionnaire*. Then we define the additional desired property: the tree actually does sorting. This section also includes other definitions such as the height of a questionnaire and how many leaves it has. Finally these notions are connected back to our study of algorithms via the definition of a *comparison sort*.

We define a pre-questionnaire as follows.

[1] We write a permutation in the traditional way where domain of the mapping is from 1 to n. We have in mind that 1 is a way of referring to the first element regardless of whether or not that first element is indexed using 0.

Definition 15. *A* pre-questionnaire *q for length n is one of the following.*

- leaf(p), *where*
  - p *is a permutation of the numbers from* 1 *to* n
- internal(ℓ, i, j, r), *where*
  - ℓ *is a* pre-questionnaire *for length* n*;*
  - $0 \leq i \leq n-1$
  - $0 \leq j \leq n-1$
  - r *is a* pre-questionnaire *for length* n.

The number of this tree's leaves is the usual notion, but it is convenient to define it explicitly.

Definition 16. *Given a* pre-questionnaire *q for length n, the leaf count,* $|q|_L$*, is defined as follows.*

$$\begin{array}{lcl} |\mathsf{leaf}(p)|_L & = & 1 \\ |\mathsf{internal}(\ell, i, j, r)|_L & = & |\ell|_L + |r|_L \end{array}$$

So too the notion of the height of this tree is the usual one, but it is also convenient to define it explicitly.

Definition 17. *Given a* pre-questionnaire *q for length n, the height*(q) *is defined as follows.*

$$\begin{array}{lcl} \mathit{height}(\mathsf{leaf}(p)) & = & 0 \\ \mathit{height}(\mathsf{internal}(\ell, i, j, r)) & = & 1 + \max(\mathit{height}(\ell), \mathit{height}(r)) \end{array}$$

To state the property that trees should sort, we must articulate what the behavior of a tree is. Internal nodes act by comparing elements and based on the comparison choose a sub-tree. The leaves permute the sequence.

Definition 18. *Given a* pre-questionnaire *q for length n,* $[\![q]\!]$ *is a function that maps orderable sequences of length n to orderable sequences of length n, where* $[\![q]\!]$ *is defined as follows.*

$$\begin{array}{lcl} [\![\mathsf{leaf}(p)]\!](x) & = & p(x) \\ [\![\mathsf{internal}(\ell, i, j, r)]\!](x) & = & \begin{cases} [\![\ell]\!](x) & \text{if } x_i \leq x_j \\ [\![r]\!](x) & \text{if } x_i > x_j \end{cases} \end{array}$$

Now we can use the behavior of a tree to identify the property of trees that sort.

Definition 19. *Given a* pre-questionnaire *q for length n, q is a* sorting *pre-questionnaire for length n if for any orderable sequence x of length n,* $x' = [\![q]\!](x)$ *implies* x' *is in order.*

We can now give a name to trees that sort: *questionnaires.*

Definition 20. *A* questionnaire *q for length n is a pre-questionnaire for length n such that q is a sorting pre-questionnaire for length n.*

The significance of questionnaires is their relationship to a certain general class of sorting algorithms. We define this class in terms of questionnaires.

Definition 21. *Given a sorting algorithm A, A is a* comparison sort *if for any natural number $n > 0$, for every input sequence of length n, A is characterized by some questionnaire q for length n. And given an input sequence of length n, A is* characterized by some questionnaire *q for length n if A's ordering of the elements is determined by comparisons A makes between elements; the first comparison A makes is the comparison identified in the root of q, the remaining comparisons A makes correspond to those in the appropriate sub-tree of q.*

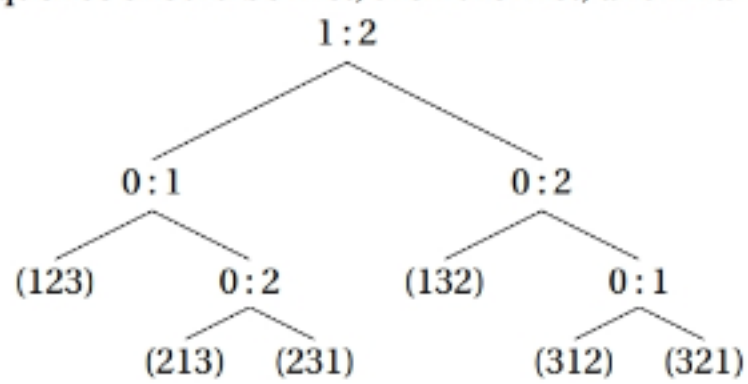
A question we might ask is, "Which of the sorting algorithms from chapters 9, 11, and 12 are comparison sorts?" The answer is *all of them*! Insertion sort, selection sort, heap sort, merge sort, and quick sort are all comparison sorts.

We generally prefer comparison sorts because of their flexibility. We do not need to know anything about the data other than how to compare elements — which was the requirement going in: the data is orderable. However, there are more special purpose sorting algorithms.

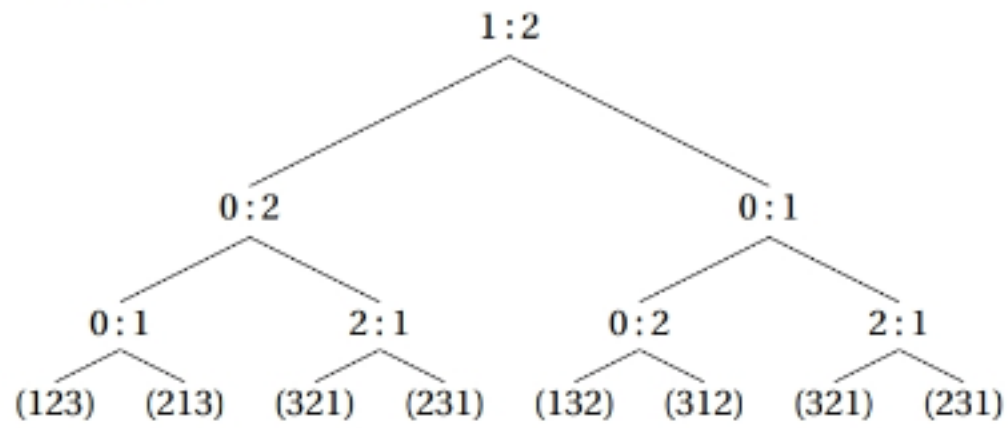
Examples

For the following examples $n = 3$. We draw a diagram of internal(ℓ, i, j, r) by writing $i : j$ at the root, and ℓ and r as sub-trees below.

The first example is for the recursive version of insertion sort. Because of the recursion, the last two elements are compared first. For the example sequence $\langle 8,9,4\rangle$ we arrive at the leaf (312). The permutation says that the last in the sequence should be first, then the first, and finally the middle.



This example is for the iterative version of selection sort. Because this algorithm explicitly starts with the last element of the sequence, the last two elements are compared first. For the example sequence $\langle 8,9,4\rangle$ we arrive at the leaf (312).



14.3 Consequences of the Alternative Model

We start with a couple of lemmas about questionnaires. The first is a general result about the relationship between the height of a binary tree and the number of its leaves framed in the language of questionnaires.

The second is an observation about how big these trees must be to sort. The conclusion is that the worst case performance of any comparison sort algorithm must be $\Omega(n\log(n))$.

Lemma 15. *For any pre-questionnaire q for length n, $|q|_L \leq 2^{height(q)}$.*

Proof. By structural induction.

- Observe that if $q = \mathsf{leaf}(p)$ then the result follows.

$$\begin{aligned} |q|_L &= |\mathsf{leaf}(p)|_L \\ &= 1 \\ &= 2^0 \\ &= 2^{height(\mathsf{leaf}(p))} \\ &= 2^{height(q)} \end{aligned}$$

- Assume $|\hat{q}|_L \leq 2^{height(\hat{q})}$ if $\hat{q}$ is a substructure of q.
  Suppose $q = \mathsf{internal}(\ell, i, j, r)$.

$$\begin{aligned} |q|_L &= |\mathsf{internal}(\ell, i, j, r)|_L \\ &= |\ell|_L + |r|_L \\ &\leq 2^{height(\ell)} + 2^{height(r)} \\ &\leq 2^m + 2^m && \text{where } m = \max(height(\ell), height(r)) \\ &= 2 \times 2^m \\ &= 2^{1+m} \\ &= 2^{height(\mathsf{internal}(\ell,i,j,r))} \\ &= 2^{height(q)} \end{aligned}$$

□

A sorting tree must be able to handle any possible input sequence of length n. Thus there must be a lot of leaves to handle all those inputs.

Lemma 16. *Given a questionnaire q for length n, $|q|_L \geq n!$.*

Proof. Let q be a questionnaire for length n, and without loss of generality let x be a sequence of distinct elements of length n. Since $[\![q]\!]$ is a sort function, q itself must contain a permutation for every possible permutation of x. Hence the number of leaves of q must be at least $n!$. □

We have seen is other chapters that we sometimes choose to count comparisons to get a sense of the performance of a sorting algorithm even if that is not all that contributes to the performance. Since these trees emphasize comparisons, they can give us a fairly detailed view of how many are involved.

Theorem 17. *For any comparison sort algorithm A, if $T_A(n)$ is the worst case number of comparisons performed by A on a sequence of length n then $T_A(n) \in \Omega(n\log(n))$*

Proof. Let A be a comparison sort algorithm, and let n be the length of an input sequence.
Suppose $T_A(n)$ is the worst case number of comparisons performed by A on a sequence of length n.
Since A is characterized by some questionnaire q for length n, $T_A(n) = height(q)$. By the previous two lemmas, we have that $2^{height(q)} \geq |q|_L \geq n!$. Hence $height(q) \geq \lg(n!)$. Thus $T_A(n) \in \Omega(n\log(n))$. □

This theorem entails that no comparison sort algorithm can have a better worst case performance than $\Theta(n\log n)$. Thus heap sort and merge sort are asymptotically optimal comparison sorts.

Part III

Semi-Numerical Algorithms

Chapter 15

Strassen's Method**

15.1 Introduction to Matrix Multiplication

Consider the following problem.

Problem 11. *Given two $n \times n$ (square) matrices A, B, what is their product AB?*

Example

$$\begin{pmatrix} 1 & 4 \\ 7 & 5 \end{pmatrix}\begin{pmatrix} 6 & 8 \\ 3 & 2 \end{pmatrix} = \begin{pmatrix} 1\cdot 6+4\cdot 3 & 1\cdot 8+4\cdot 2 \\ 7\cdot 6+5\cdot 3 & 7\cdot 8+5\cdot 2 \end{pmatrix}$$
$$= \begin{pmatrix} 6+12 & 8+8 \\ 42+15 & 56+10 \end{pmatrix}$$
$$= \begin{pmatrix} 18 & 16 \\ 57 & 66 \end{pmatrix}$$

Recall that if $A = (a_{i,j})$ and $B = (b_{i,j})$ are $n \times n$ matrices then $C = AB$ is defined by $C = (c_{i,j})$ where C is an $n \times n$ matrix and $c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$. An algorithm follows fairly directly from this characterization.

def SqMatMult(A, B) :
1. $n \leftarrow A$.rows
2. $C \leftarrow 0_{n\times n}$
3. **for** $i \leftarrow 1$ **to** n :
4. **for** $j \leftarrow 1$ **to** n :
5. **for** $k \leftarrow 1$ **to** n :
6. $C[i, j] \leftarrow C[i, j] + A[i, k] \times B[k, j]$
7. **return** C

We can characterize the time complexity $T(n)$ by counting the number of scalar arithmetic operations. Note that there are two arithmetic operations in the innermost loop. Each loop takes n iterations. Thus $T(n) = 2n^3$ which entails that $T(n) = \Theta(n^3)$.

15.2 A Divide and Conquer Approach

The algorithm in the previous section seems very natural, yet we wonder whether we might do any better using a divide and conquer algorithm. Assume for the moment that n has the form $n = 2^m$. Then we can break the matrices A and B into quarters. Thus $A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}$ and $B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$ where $A_{i,j}$ and $B_{i,j}$ are sub-matrices. The product $C = AB$ will have the form $C = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$ where $C_{i,j}$ are defined by the following equations.

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

If we have in mind the multiplications in those equations are computed via recursion, we get the following divide and conquer algorithm (which we now express using functional notation). We write $\odot$ for the name of the function.

$$A \odot B = \begin{cases} A \times B & \text{if } A.\text{rows} = 1 \\ \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix} & \text{otherwise} \end{cases}$$

$$\text{where } \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} = A$$

$$\begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = B$$

$$\begin{aligned} C_{1,1} &= A_{1,1} \odot B_{1,1} + A_{1,2} \odot B_{2,1} \\ C_{1,2} &= A_{1,1} \odot B_{1,2} + A_{1,2} \odot B_{2,2} \\ C_{2,1} &= A_{2,1} \odot B_{1,1} + A_{2,2} \odot B_{2,1} \\ C_{2,2} &= A_{2,1} \odot B_{1,2} + A_{2,2} \odot B_{2,2} \end{aligned}$$

Here too we will characterize the time complexity $\hat{T}(n)$ by counting the number of scalar arithmetic operations. Since this algorithm is recursive, we first derive a recurrence relation and then we will solve it. When the matrices are in fact scalars, $n = 1$, and there is a single scalar (multiply) operation. Thus $\hat{T}(1) = 1$. When the matrices are larger, there are four matrix additions and eight recursive multiplications. Matrix addition is pointwise. If A and B are $n \times n$ matrices, then the components are $n/2 \times n/2$. Hence a single matrix addition involves exactly $n^2/4$ scalar (addition) operations. And so we have the following recurrence relation.

$$\begin{aligned} \hat{T}(1) &= 1 \\ \hat{T}(n) &= 8\hat{T}(n/2) + n^2 \end{aligned}$$

Since we are assuming that $n = 2^m$, we can alternately express the recurrence relation as follows.

$$\begin{aligned} \hat{T}(2^0) &= 1 \\ \hat{T}(2^m) &= 8\hat{T}(2^{m-1}) + (2^m)^2 \end{aligned}$$

Let us use iteration and expand $\hat{T}(n)$.

$$\begin{aligned}
\hat{T}(2^m) &= 8\hat{T}(2^{m-1}) + (2^m)^2 \\
&= 8(8\hat{T}(2^{m-2}) + (2^{m-1})^2) + (2^m)^2 \\
&= 8^2\hat{T}(2^{m-2}) + 8(2^{m-1})^2 + (2^m)^2 \\
&= 8^2(8\hat{T}(2^{m-3}) + (2^{m-2})^2) + 8(2^{m-1})^2 + (2^m)^2 \\
&= 8^3\hat{T}(2^{m-3}) + 8^2(2^{m-2})^2) + 8(2^{m-1})^2 + (2^m)^2 \\
&= 8^k\hat{T}(2^{m-k}) + 8^{k-1}(2^{m-(k-1)})^2) + \cdots + 8^1(2^{m-1})^2 + 8^0(2^m)^2 \leftarrow \text{Identify the pattern}
\end{aligned}$$

Once we have identified the pattern, let $k = m$. Now we have a sum to simplify.

$$\begin{aligned}
\hat{T}(2^m) &= \sum_{i=0}^{m} 8^i(2^{m-i})^2 \\
&= \sum_{i=0}^{m} 8^i(2^{2m-2i}) \\
&= \sum_{i=0}^{m} 8^i(2^{2m})(2^{-2i}) \\
&= (2^m)^2\sum_{i=0}^{m} 8^i(2^{-2i}) \\
&= (2^m)^2\sum_{i=0}^{m} (8/4)^i \\
&= (2^m)^2\sum_{i=0}^{m} 2^i \\
&= (2^m)^2(2^{m+1}-1)
\end{aligned}$$

If we re-express this result in terms of n, we have that $\hat{T}(n) = n^2(2n-1)$. And so $\hat{T}(n) = \Theta(n^3)$. It may appear that we have gained nothing from all this work. Not so! The equations reveal how the performance is dependent on the number of recursive calls.

Suppose we could reduce the number of recursive calls just by one from 8 to 7. Let us make that replacement and see what happens.

$$\begin{aligned}
\hat{T}(2^m) &= (2^m)^2\sum_{i=0}^{m} (7/4)^i \\
&= (2^m)^2\left(\frac{(7/4)^{m+1}-1}{7/4-1}\right) \\
&= (2^m)^2(4/3)((7/4)(7/4)^m - 1) \\
&= (2^m)^2(4/3)((7/4)(2^m)^\alpha - 1) \text{ where } \alpha = \lg(7/4)
\end{aligned}$$

Note that $\alpha \approx 0.8073549$. So if we can get the recursive calls down to only 7, we will have an algorithm with performance $\hat{T}(n) = O(n^{2.81})$, which would be superior to our original algorithm. Now the question is, can we do it?

15.3 Strassen's Enhancement

It turns out that we can reduce the number of recursive calls by one. Doing so involves trading off that recursive call for multiple additions, but it is worth it. Even so, a lot of cleverness is involved. The sums and products are simply given below followed by verifications.

15.3.1 Seven Products

We need the following ten sums (each labeled with an S).

$$\begin{aligned}
S_1 &= B_{1,2} - B_{2,2} \\
S_2 &= A_{1,1} + A_{1,2} \\
S_3 &= A_{2,1} + A_{2,2} \\
S_4 &= B_{2,1} - B_{1,1} \\
S_5 &= A_{1,1} + A_{2,2} \\
S_6 &= B_{1,1} + B_{2,2} \\
S_7 &= A_{1,2} - A_{2,2} \\
S_8 &= B_{2,1} + B_{2,2} \\
S_9 &= A_{1,1} - A_{2,1} \\
S_{10} &= B_{1,1} + B_{1,2}
\end{aligned}$$

With those sums, we can compute the seven needed products (each labeled with a P).

$$\begin{aligned}
P_1 &= A_{1,1}S_1 \\
P_2 &= S_2B_{2,2} \\
P_3 &= S_3B_{1,1} \\
P_4 &= A_{2,2}S_4 \\
P_5 &= S_5S_6 \\
P_6 &= S_7S_8 \\
P_7 &= S_9S_{10}
\end{aligned}$$

Computing the components of the answer requires a few more sums, but no more products.

$$\begin{aligned}
C_{1,1} &= P_5 + P_4 - P_2 + P_6 \\
C_{1,2} &= P_1 + P_2 \\
C_{2,1} &= P_3 + P_4 \\
C_{2,2} &= P_5 + P_1 - P_3 - P_7
\end{aligned}$$

15.3.2 Verification

Verifying that the computations yield correct results entails substituting into the formulas and simplifying. We verify the $C_{1,1}$ and $C_{1,2}$, but leave the remaining two verifications as exercises.

$$
\begin{aligned}
C_{1,1} &= P_5 + P_4 - P_2 + P_6 \\
&= S_5 S_6 + A_{2,2} S_4 - S_2 B_{2,2} + S_7 S_8 \\
&= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) + A_{2,2} S_4 - S_2 B_{2,2} + S_7 S_8 \\
&= A_{1,1} B_{1,1} + A_{1,1} B_{2,2} + A_{2,2} B_{1,1} + A_{2,2} B_{2,2} + A_{2,2} S_4 - S_2 B_{2,2} + S_7 S_8 \\
&= A_{1,1} B_{1,1} + A_{1,1} B_{2,2} + A_{2,2} B_{1,1} + A_{2,2} B_{2,2} + A_{2,2}(B_{2,1} - B_{1,1}) - S_2 B_{2,2} + S_7 S_8 \\
&= A_{1,1} B_{1,1} + A_{1,1} B_{2,2} + A_{2,2} B_{1,1} + A_{2,2} B_{2,2} + A_{2,2} B_{2,1} - A_{2,2} B_{1,1} - S_2 B_{2,2} + S_7 S_8 \\
&= A_{1,1} B_{1,1} + A_{1,1} B_{2,2} + A_{2,2} B_{2,2} + A_{2,2} B_{2,1} - S_2 B_{2,2} + S_7 S_8 \\
&= A_{1,1} B_{1,1} + A_{1,1} B_{2,2} + A_{2,2} B_{2,2} + A_{2,2} B_{2,1} - (A_{1,1} + A_{1,2}) B_{2,2} + S_7 S_8 \\
&= A_{1,1} B_{1,1} + A_{1,1} B_{2,2} + A_{2,2} B_{2,2} + A_{2,2} B_{2,1} - A_{1,1} B_{2,2} - A_{1,2} B_{2,2} + S_7 S_8 \\
&= A_{1,1} B_{1,1} + A_{2,2} B_{2,2} + A_{2,2} B_{2,1} - A_{1,2} B_{2,2} + S_7 S_8 \\
&= A_{1,1} B_{1,1} + A_{2,2} B_{2,2} + A_{2,2} B_{2,1} - A_{1,2} B_{2,2} + (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \\
&= A_{1,1} B_{1,1} + A_{2,2} B_{2,2} + A_{2,2} B_{2,1} - A_{1,2} B_{2,2} + A_{1,2} B_{2,1} + A_{1,2} B_{2,2} - A_{2,2} B_{2,1} - A_{2,2} B_{2,2} \\
&= A_{1,1} B_{1,1} + A_{2,2} B_{2,1} - A_{1,2} B_{2,2} + A_{1,2} B_{2,1} + A_{1,2} B_{2,2} - A_{2,2} B_{2,1} \\
&= A_{1,1} B_{1,1} - A_{1,2} B_{2,2} + A_{1,2} B_{2,1} + A_{1,2} B_{2,2} \\
&= A_{1,1} B_{1,1} + A_{1,2} B_{2,1}
\end{aligned}
$$

The calculation for $C_{1,2}$ follows. This one is significantly shorter.

$$
\begin{aligned}
C_{1,2} &= P_1 + P_2 \\
&= A_{1,1} S_1 + S_2 B_{2,2} \\
&= A_{1,1}(B_{1,2} - B_{2,2}) + (A_{1,1} + A_{1,2}) B_{2,2} \\
&= A_{1,1} B_{1,2} - A_{1,1} B_{2,2} + A_{1,1} B_{2,2} + A_{1,2} B_{2,2} \\
&= A_{1,1} B_{1,2} + A_{1,2} B_{2,2}
\end{aligned}
$$

Part IV

Graph Algorithms

Chapter 16

Shortest Paths**

16.1 Introduction to Shortest Paths

Consider the problem of finding a shortest path in a weighted graph.

Problem 12. *Given a weighted graph $W = ((V, E), w)$, a source vertex $u \in V$, and a destination vertex $v \in V$, what is a shortest path from u to v?*

To simplify, we will focus on merely computing the shortest path weight.

Problem 13. *Given a weighted graph $W = ((V, E), w)$, a source vertex $u \in V$, and a destination vertex $v \in V$, what is the shortest path weight from u to v?*

We expect it to be straightforward to augment an algorithm that finds the shortest path weight so that it finds the path as well: since the weight calculation involves vertices, record those vertices when performing the shortest path weight calculation.

Recall that we write $\delta(u, v)$ for the shortest path weight from u to v. Where might one begin to think about how to compute $\delta(u, v)$? We will start by imagining that we already have the solution. Note that the solution need not involve a path. It could be that there simply is no path from u to v; in that case $\delta(u, v) = \infty$. Or, it could be that there is no *shortest* path from u to v — it could be that there is a negative path weight cycle; in that case $\delta(u, v) = -\infty$. But if there is a shortest path P, $\delta(u, v) = w^*(P)$. Following the structure in the definition, there are two possible forms for P: either $P = v$ or $P = P', v$. If $P = v$ then it must be that $v = u$, and $\delta(u, v) = \delta(u, u) = w^*(u) = 0$. Otherwise $\delta(u, v) = w^*(P) = w^*(P', v) = w^*(P') + w(v', v)$, where v' is the penultimate vertex of P. Observe that P' must be a shortest path to v' because otherwise we could reduce $\delta(u, v)$ by replacing $w^*(P')$ with the weight of a shortest path. Thus $w^*(P')$ is the shortest path weight from u to v', so $w^*(P') = \delta(u, v')$, and $\delta(u, v) = \delta(u, v') + w(v', v)$.

These observations lead to the following recursive characterization of $\delta(v)$, where the source vertex u is now implicit.

$$\delta(v) = \begin{cases} \infty & \text{if there is no path to } v \\ -\infty & \text{if there is a path to } v \text{ that has a negative path weight cycle.} \\ 0 & \text{if } v = u \text{ and there are no paths to } u \text{ with negative path weight cycles} \\ \delta(v') + w(v', v) & \text{if there is a path to } v, \text{ there are no paths with negative path weight cycles,} \\ & \text{and } v' \text{ is the penultimate vertex on a shortest path} \end{cases}$$

This characterization looks almost like an algorithm; however, it is not obvious how to compute some of the results used. In particular, it is not clear how to compute the penultimate vertex of a shortest path. In fact, it is not necessary to know a shortest path or a vertex on a shortest path. Instead, it is enough to consider predecessors of v, and then determine the minimum path weight from among those paths involving the predecessors. Thus the equation becomes the following.

$$\delta(v) = \begin{cases} \infty & \text{if there is no path to } v \\ -\infty & \text{if there is a path to } v \text{ that has a negative path weight cycle.} \\ 0 & \text{if } v = u \text{ and there are no paths to } u \text{ with negative path weight cycles} \\ \min\{\delta(v') + w(v', v)\} & \text{if there is a path to } v, \text{ there are no paths with negative path weight cycles, and } v' \text{ is a predecessor of } v \end{cases}$$

The single-source shortest path weight problem naturally follows from this equation since we may need to know the shortest path weight to any of the vertices in the course of finding the shortest path weight to v.

Problem 14. *Given a weighted graph $W = ((V, E), w)$ and $u \in V$, what is the shortest path weight from u to v, for every $v \in V$.*

The equation above looks suggestively like a recursive algorithm for solving the single-source shortest path weight problem. However, there are two issues.

1. *How can the computation be organized so that its execution is efficient?* If the equation is to be interpreted as an algorithm, there are multiple calls to δ. Multiple recursive calls can be computationally expensive; further, a recursive call may be unnecessary if the result has already been calculated. Dynamic programming is an excellent option for improving performance by eliminating redundant calls.

2. *How can we be sure the sub-problems are getting smaller?* If the equation is to be interpreted as an algorithm, the sub-problems must be getting smaller. Recall that for functions that operate on lists, the recursive call is on a smaller list. In what way is v' smaller than v? There is no clear answer. To ensure the sub-problem is smaller, we add another parameter that is reduced at every recursive call. In particular, $\delta(v; n)$ will be understood as the shortest path weight to v in "generation" n.

16.2 The Bellman–Ford Algorithm

The Bellman–Ford algorithm ensures the sub-problems are getting smaller in a natural way: the shortest path weight to v in "generation" n is the shortest path weight to v considering only paths with n or fewer edges. We formalize this notion below.

Definition 22. *Given a weighted graph $W = ((V, E), w)$, the* Bellman–Ford path weight *is the function $\delta : V \to (V \times \mathbb{N} \to \bar{\mathbb{R}})$, where $\delta_u(v; n) = \inf\{w^*(P) \mid P \text{ is a path with source } u, \text{ destination } v, \text{ and } |P| \leq n\}$. If u is understood from context, then the Bellman–Ford path weight is denoted simply by $\delta(v; n)$.*

We will see in the next section that the recursive characterization of the Bellman–Ford path weight can be used for computation. But first we are concerned with establishing a connection between the Bellman–Ford path weight and the shortest path weight because we are, in fact, interested in computing the shortest path weight. The next lemma is an answer to the question, "When is the Bellman–Ford path weight the same as the shortest path weight?"

Lemma 17. *Given a weighted graph $W = ((V,E),w)$, if, for every $v \in V$, $\delta(v) > -\infty$ then, for every $v \in V$, $\delta(v) = \delta(v;|V|-1)$.*

Proof. Observe that if $\delta(v) > -\infty$ for all $v \in V$, then the shortest paths will not have cycles, and so the longest shortest path can involve all the vertices no more than once. Thus, there can be at most $|V|-1$ edges in the path; one edge for each vertex on the path but the last. Hence, $\delta(v) = \delta(v;|V|-1)$.

□

Thus the Bellman–Ford path weight is the same as the shortest path weight when the graph has no negative path weight cycles. The remaining lemmas in this section lead to the final corollary. It is this corollary that answers the question, "How can we determine whether or not the graph has negative path weight cycles?"

Lemma 18. *Given a weighted graph $W = ((V,E),w)$, if, for every $v \in V$, $\delta(v) > -\infty$ then, for every $v \in V$, for any $N \in \mathbb{N}$, if $N \geq |V|-1$ then $\delta(v;N) = \delta(v;|V|-1)$.*

In addition to the general lemma above, we explicitly write the corollary when $N = |V|$. We will use the contrapositive of this special case below.

Corollary 9. *Given a weighted graph $W = ((V,E),w)$, if, for every $v \in V$, $\delta(v) > -\infty$ then, for every $v \in V$, $\delta(v;|V|) = \delta(v;|V|-1)$.*

And so if there are no negative path weight cycles then there is no change to the Bellman–Ford path weight if we increase the number of edges allowed from $|V|-1$ to $|V|$. The contrapositive indicates that if the Bellman–Ford path weight does go down then there is a negative path weight cycle.

Lemma 19. *Given a weighted graph $W = ((V,E),w)$, if there exists some $v \in V$ such that $\delta(v;|V|) < \delta(v;|V|-1)$ then there exists some $v \in V$ such that $\delta(v) = -\infty$.*

Proof. This result is the contrapositive of Corollary 9. □

But can we be sure that the Bellman–Ford path weight will go down if there is a negative path weight cycle? Yes, the converse is also true.

Lemma 20. *Given a weighted graph $W = ((V,E),w)$, if there exists some $v \in V$ such that $\delta(v) = -\infty$ then there exists some $v \in V$ such that $\delta(v;|V|) < \delta(v;|V|-1)$.*

Proof. By contradiction.

Suppose there exists some $v \in V$ such that $\delta(v) = -\infty$, but for all $v \in V$, $\delta(v;|V|) = \delta(v;|V|-1)$.

First note that for a shortest path weight path $P = P', v$ of length n, where the destination of P' is v', $\delta(v;n) = \delta(v';n-1) + w(v',v)$. Further, if we don't know whether or not P is a shortest path weight path, we still have the relation $\delta(v;n) \leq \delta(v';n-1) + w(v',v)$.

Since $\delta(v) = -\infty$ for some $v \in V$, there exists a negative path weight cycle reachable from the source vertex u. Let C be this cycle, where $C = v_0 \cdots v_k$, $v_0 = v_k$, and $\sum_{i=1}^{k} w(v_{i-1},v_i) < 0$.

Now consider the sum $\sum_{i=1}^{k} \delta(v_i;|V|)$. Note that this sum is finite since every v_i is reachable from the source vertex u.

$$\begin{aligned}
\sum_{i=1}^{k} \delta(v_i;|V|) &\leq \sum_{i=1}^{k} (\delta(v_{i-1};|V|-1) + w(v_{i-1},v_i)) && \text{since } v_{i-1} \text{ is a predecessor of } v_i \\
&= \sum_{i=1}^{k} \delta(v_{i-1};|V|-1) + \sum_{i=1}^{k} w(v_{i-1},v_i) \\
&= \sum_{i=1}^{k} \delta(v_{i-1};|V|) + \sum_{i=1}^{k} w(v_{i-1},v_i) && \text{by supposition} \\
&= \sum_{i=1}^{k} \delta(v_i;|V|) + \sum_{i=1}^{k} w(v_{i-1},v_i) && \text{since } v_0 = v_k
\end{aligned}$$

Hence $\sum_{i=1}^{k} w(v_{i-1},v_i) \geq 0$.

Since $\sum_{i=1}^{k} w(v_{i-1},v_i) < 0$ and $\sum_{i=1}^{k} w(v_{i-1},v_i) \geq 0$ are contradictory, $\delta(v;|V|) = \delta(v;|V|-1)$ for all $v \in V$ cannot be the case.

□

Putting the lemmas together yields a corollary that we can use as a test to determine whether or not there is a negative path weight cycle.

Corollary 10. *Given a weighted graph $W = ((V,E),w)$, W has a negative path weight cycle iff there exists some $v \in V$ such that $\delta(v;|V|) < \delta(v;|V|-1)$.*

16.2.1 A Recursive Formulation

The Bellman–Ford path weight can be characterized recursively. The optimal sub-structure of shortest paths is used here coupled with the insight that since the sub-path is shorter the numerical parameter can go down by one. Since that measure always decreases, the recursion will terminate.

$$\begin{aligned}
\delta(v;0) &= \begin{cases} 0 & \text{if } v = u \\ \infty & \text{otherwise} \end{cases} \\
\delta(v;n+1) &= \begin{cases} \min(\delta(v;n), \inf\{\delta(v';n) + w(v',v) \mid (v',v) \in E\}) & \text{if } v = u \\ \inf\{\delta(v';n) + w(v',v) \mid (v',v) \in E\} & \text{otherwise} \end{cases}
\end{aligned}$$

Unfortunately, there are many recursive calls, so a direct implementation will take exponential time.

16.2.2 A Dynamic Programming Formulation

It is natural to use dynamic programming to reduce the exponential time complexity of the direct recursive implementation.

A Dynamic Programming Algorithm

Here we represent the function by an array; thus $d[v,n] = \delta(v;n)$. Further, distinguishing between $v = u$ and otherwise for the recursive case does not seem worthwhile in practice and we use the more general formula in both cases.

def INIT-DYNAMIC(V, u, d):

```
1. for v ∈ V :
2.     d[v,0] ← ∞
3. d[u,0] ← 0
```

def BELLMAN-FORD-DYNAMIC(G, w, u):

```
1. d ← array(|G.V|, |G.V|)
2. INIT-DYNAMIC(G.V, u, d)
3. for n ← 1 to |G.V| − 1 :
4.     for v ∈ G.V :
5.         d[v,n] ← min(d[v,n−1], inf{d[v',n−1] + w(v',v) | (v',v) ∈ G.E})
6. for v ∈ G.V :
7.     if d[v,|G.V|−1] > inf{d[v',|G.V|−1] + w(v',v) | (v',v) ∈ G.E} :
8.         return ⊥, d # detected a negative path weight cycle
9. return ⊤, d
```

Implementing the Inner Loop

Recall the pseudo-code for the inner loop. It uses mathematical notation. We would like to see how the mathematics is implemented here as well.

```
1. for v ∈ G.V :
2.     d[v,n] ← min(d[v,n−1], inf{d[v',n−1] + w(v',v) | (v',v) ∈ G.E})
```

There are two implementation variations that we will consider. First assume the graph representation has a predecessor operator $G.P$. Then we can write the loop as follows.

```
1. for v ∈ G.V :
2.     d[v,n] ← d[v,n−1]
3.     for v' ∈ G.P(v) :
4.         if d[v,n] > d[v',n−1] + w(v',v) :
5.             d[v,n] ← d[v',n−1] + w(v',v)
```

The time complexity for the inner loop is $\Theta(V+E)$, and so the time complexity for the entire algorithm is $\Theta(V(V+E))$.

Now we assume that we can merely iterate through all the edges in the graph, and we write the loop as follows.

```
1. for v ∈ G.V :
2.     d[v,n] ← d[v,n−1] # v is constrained by the outer loop
3.     for (v',v) ∈ G.E :
4.         if d[v,n] > d[v',n−1] + w(v',v) :
5.             d[v,n] ← d[v',n−1] + w(v',v)
```

The time complexity for the inner loop is $\Theta(V(V+E))$ if the graph is represented by an adjacency list, and $\Theta(VE)$ if the graph representation contains a representation of all the edges explicitly. Thus without

the predecessors available the entire algorithm is slowed down by a factor of V. Ultimately, we will see that this less stringent variation will be efficient.

In either case, the space complexity is V^2. We observe, though, that the algorithm never goes back further than one step, and so we do not need all that space.

16.2.3 Beyond Dynamic Programming

We go beyond dynamic programming by eliminating most of the table. The new algorithm now retains only the previous step.

def BELLMAN-FORD-2ARRAY(G, w, u):

1. $c \leftarrow$ **array**($|G.V|$)
2. $d \leftarrow$ **array**($|G.V|$)
3. INIT-SINGLE-SOURCE($G.V, u, d$)
4. **for** $n \leftarrow 1$ **to** $|G.V| - 1$:
5. **swap** c, d
6. **for** $v \in G.V$:
7. $d[v] \leftarrow \min(c[v], \inf\{c[v'] + w(v', v) \mid (v', v) \in G.E\})$
8. **for** $v \in G.V$:
9. **if** $d[v] > \inf\{d[v'] + w(v', v) \mid (v', v) \in G.E\}$:
10. **return** $\bot, d$
11. **return** $\top, d$

def INIT-SINGLE-SOURCE(V, u, d):

1. **for** $v \in V$:
2. $d[v] \leftarrow \infty$
3. $d[u] \leftarrow 0$

Assuming $G.P$ is available, the time complexity is the same as before: $\Theta(V(V+E))$. The space complexity for this algorithm is $2V$.

16.2.4 Throwing Caution to the Wind

In the algorithm above we need the array that has the path weights from the previous step in order to calculate the path weights of the next step. Eliminating an array changes the results. We throw caution to the wind and do just that. Intuitively, the results are ultimately the same because that worst that happens is that we use a more recent value instead of an old value — but the more recent values are closer to the actual path weight, so it is alright. We also change to using the second approach to implement the inner loop since initialization of the array is no longer necessary — we use what is already there as the implicit initialization.

def BELLMAN-FORD-RELAX(G, w, u):
1. $d \leftarrow$ **array**($|G.V|$)
2. INIT-SINGLE-SOURCE($G.V, u, d$)
3. **for** $n \leftarrow 1$ **to** $|G.V| - 1$:
4. **for** $v \in G.V$:
5. $d[v] \leftarrow \min(d[v], \inf\{d[v'] + w(v', v) \mid (v', v) \in G.E\})$
6. **for** $v \in G.V$:
7. **if** $d[v] > \inf\{d[v'] + w(v', v) \mid (v', v) \in G.E\}$:
8. **return** $\bot, d$
9. **return** $\top, d$

The procedure INIT-SINGLE-SOURCE is defined in the previous section. Below we include the variation where the computation of the inner loop is explicit.

def BELLMAN-FORD-RELAX(G, w, u):
1. $d \leftarrow$ **array**($|G.V|$)
2. INIT-SINGLE-SOURCE($G.V, u, d$)
3. **for** $n \leftarrow 1$ **to** $|G.V| - 1$:
4. **for** $(v', v) \in G.E$:
5. **if** $d[v] > d[v'] + w(v', v)$:
6. $d[v] \leftarrow d[v'] + w(v', v)$
7. **for** $(v', v) \in G.E$:
8. **if** $d[v] > d[v'] + w(v', v)$:
9. **return** $\bot, d$
10. **return** $\top, d$

The time complexity is slightly better than before. We have $\Theta(V(V+E))$ with the traditional adjacency list representation and $\Theta(VE)$ if the edges are explicitly available. The space complexity for this algorithm is also slightly smaller: V.

The next lemma states more formally that the shortest path weights calculated are still correct.

Lemma 21. *Let P be a shortest path with source u and destination v. For all i, if $0 \leq i \leq k$ then $d[v_i] = \delta(v_i; i)$ at iteration i (and afterward).*

Part V

Complexity Theory

Chapter 17

Complexity Theory Overview**

17.1 Introduction to Complexity Theory

We have seen that some algorithms take exponential time, yet some additional effort yielded a polynomial time algorithm. Sometimes we were even able to achieve linear time. We might wonder whether there are problems for which there is no fast algorithm that solves it. Certainly such problems should be considered intractable, or impractical for computing actual solutions when the input is large. If an answer requires billions of years to compute, we are not interested in waiting. Further, even if the answer only takes a year to compute, but we need it tomorrow, we are still not interested in waiting. On the other hand, problems that do have fast algorithms to solve them are tractable.

The notion of a problem being tractable is intuitive and informal. First we will introduce some formal definitions in an attempt to characterize the class of tractable decision problems. Then we will consider a problem that seems very difficult. Finally we define another formal class of decision problems which includes that difficult problem and others that do not appear to be tractable.

17.2 Formalizing the Class of Tractable Problems

In an attempt to formalize the notion of tractable, we start by formalizing the notion of problem. The essence of a problem is a function; given an instance, there is an answer but we do not have to worry about how it is computed. As is often the case, we focus on decision problems where there are only two possible answers; this simplifies the theory and we give up little in doing so.

Definition 23. *Given a countably infinite set of instances I, a* decision problem *is a function $Q : I \to \mathbb{B}$.*

Note that we require the domain of a problem to be countably infinite because a function with a finite domain would be uninteresting and we could not even write down all the instances of a function with an uncountably infinite domain.

Now we introduce the notion of an algorithm *solving* a problem. We understand that to mean that the algorithm on the same input as the problem always gives the output that the problem says is the answer.

Definition 24. *Given a decision problem $Q : I \to \mathbb{B}$ and an algorithm A, A* solves *Q if for any $i \in I$, $A(i) = Q(i)$.*

We formalize the intuitive notion of an problem being tractable by stipulating that there must be a polynomial time algorithm that solves it. Note that polynomial time does *not* mean that the time complexity is a polynomial. Rather the time complexity is *bounded* by a polynomial, so functions like $\log(n)$ are included since $\log(n) \in O(n)$ and n is a polynomial.

Definition 25. *A decision problem $Q : I \to \mathbb{B}$ is* solvable in polynomial time *if there exists an algorithm A such that A solves Q, the time complexity of A is $T_A(n)$, and there exists $k \in \mathbb{N}$ such that $T_A(n) \in O(n^k)$.*

Then we lump all of those problems into one class.

Definition 26. *The class $\mathscr{P}$ is the set of all decision problems that are solvable in polynomial time.*

Although it is possible to challenge this association of $\mathscr{P}$ and the class of tractable problems, we will accept it and proceed.

17.3 Thinking About $\mathscr{P}$

We might wonder whether there are many problems in $\mathscr{P}$. Indeed, even though we are restricting our attention to problems with only two answers, there are a lot! Some examples follow.

- Given a natural number n, is n even?
- Given a graph G, does G have an Euler circuit?
- Given $\langle M, x\rangle$ where M is a DFA and x is a string, does M accept x?
- Given $\langle G, x\rangle$ where G is a context free grammar and x is a string, is $x \in L(G)$?

Further, many problems that are not decision problems can be reformulated so that they become decision problems. For example, consider the problem of finding the shortest path in a graph G from one vertex to another. We can formulate this problem as a decision problem as follows. Given $\langle G, v_1, v_2, k\rangle$ where $G = (V, E)$ is a graph, $v_1, v_2 \in V$, and $k \in \mathbb{N}$, is there a path from v_1 to v_2 such that the length of the path is less than k? That decision problem is also in $\mathscr{P}$.

At this point, we might wonder if there are any decision problems that are not in $\mathscr{P}$. There are. The halting problem is not in $\mathscr{P}$ since it is not even recursive. The following related decision problem is recursive, but is still not in $\mathscr{P}$. Given $\langle P, x, k\rangle$ where P is a program, x is an input for P, and $k \in \mathbb{N}$, does P halt on input x in k steps? This problem takes exponential time if k is expressed as a non-unary numeral.

There are also decision problems that no one knows whether or not they are in $\mathscr{P}$. We consider such a problem in the next section.

17.4 The Satisfiability Problem

The satisfiability problem concerns whether or not the variables in a propositional formula can be assigned truth values so that the propositional formula evaluates to true. Of course all of these concepts need to be formally defined, but first consider the following example.

Example

Let V be the set of variables $V = \{x, y, z, w\}$.
Let ψ be the propositional formula $\psi = (x \vee \neg y) \wedge (y \vee z \vee \neg w) \wedge (\neg x \vee \neg z \vee \neg w)$.
Is it possible to assign truth values to the variables in V so that ψ evaluates to true? Yes!
Consider the truth assignment $\mathscr{A} : V \to \mathbb{B}$, where $\mathscr{A}(x) = \top$, $\mathscr{A}(y) = \top$, $\mathscr{A}(z) = \bot$, $\mathscr{A}(w) = \top$. Notice that given the truth assignment, we can quickly verify that ψ is satisfiable. We will call such evidence a *certificate*.

The next section defines propositional formulas. The section after that defines truth assignment and evaluation. And the section after that defines satisfiability and the satisfiability problem.

17.4.1 Propositional Formula Definition

While we have already seen an example of a propositional formula, the next definition spells out exactly what it looks like. Note that this notion is purely syntactic and so we do not use values from $\mathbb{B}$.

Definition 27. *Given a set of variables V, a propositional formula ψ (over V) is one of the following.*

- **TRUE**, *or,*
- **FALSE**, *or,*
- p *if* $p \in V$, *or,*
- $(\psi_1 \vee \psi_2)$ *if* ψ_1 *and* ψ_2 *are propositional formulas, or,*
- $(\psi_1 \wedge \psi_2)$ *if* ψ_1 *and* ψ_2 *are propositional formulas, or,*
- $(\neg\psi)$ *if* ψ *is a propositional formula.*

The set of all propositional formulas over V is denoted by $\mathscr{PROP}(V)$.

It is conventional among mathematicians to blur the distinction between concrete syntax and abstract syntax. Thus we allow propositional formulas to be written without parentheses with the understanding that the or operator ($\vee$) has lowest precedence and the negation operator ($\neg$) has the highest precedence.

17.4.2 Truth Assignment Definitions

We have already seen that a truth assignment is a function, but it is repeated here for emphasis.

Definition 28. *Given a set of variables V, a* truth assignment *is a function $\mathscr{A}_V : V \to \mathbb{B}$.*

Note that although the domain of a truth assignment $\mathscr{A}_V$ is only the set of variables V, it induces a mapping from propositional formulas over V to the truth values. We define this mapping next.

Definition 29. *Given a truth assignment $\mathscr{A}_V$, the propositional evaluation function $\mathscr{E}_{\mathscr{A}_V} : \mathscr{PROP}(V) \to \mathbb{B}$ is defined as follows.*

$$
\begin{array}{lcl}
\mathscr{E}_{\mathscr{A}_V}(\textbf{TRUE}) & = & \top \\
\mathscr{E}_{\mathscr{A}_V}(\textbf{FALSE}) & = & \bot \\
\mathscr{E}_{\mathscr{A}_V}(p) & = & \mathscr{A}_V(p) \\
\mathscr{E}_{\mathscr{A}_V}(\psi_1 \vee \psi_2) & = & \mathscr{E}_{\mathscr{A}_V}(\psi_1) \vee \mathscr{E}_{\mathscr{A}_V}(\psi_2) \\
\mathscr{E}_{\mathscr{A}_V}(\psi_1 \wedge \psi_2) & = & \mathscr{E}_{\mathscr{A}_V}(\psi_1) \wedge \mathscr{E}_{\mathscr{A}_V}(\psi_2) \\
\mathscr{E}_{\mathscr{A}_V}(\neg\psi) & = & \neg\mathscr{E}_{\mathscr{A}_V}(\psi)
\end{array}
$$

Note that the left-hand side of these equations is syntactic, but the right-hand side is semantic. We can see that clearly for the truth values. With the logical operators the distinction is less obvious. In particular, the $\vee$ on the left is a syntactic constructor, while the $\vee$ on the right is the Boolean function that computes the Boolean result using the traditional truth table.

17.4.3 Satisfiability Problem Formal Definition

Now that we have a formal definition of formula evaluation, we can formally define satisfiability rather than merely saying "a formula evaluates to true." Some additional customary logical notation is introduced as well.

Definition 30. *Given a propositional formula ψ over V and a truth assignment $\mathscr{A}_V$, $\mathscr{A}_V$* satisfies *ψ, or $\mathscr{A}_V \models \psi$, if $\mathscr{E}_{\mathscr{A}_V}(\psi) = \top$.*

We now have the notation to state the satisfiability problem.

Problem 15 (SAT)**.** *Given a propositional formula ψ over V, is there a truth assignment $\mathscr{A}_V$ such that $\mathscr{A}_V \models \psi$?*

A naive algorithm for solving this problem involves enumerating all possible truth assignments. Note that if there are n variables, there are 2^n truth assignments. Is there an efficient way of finding a satisfying truth assignment? No one knows.

17.5 Another Class of Problems

In the previous section we introduced the satisfiability problem both formally and informally. We observed that no one has been able to show that SAT is an element of $\mathscr{P}$. (Perhaps the reason why is because it is not.) In this section, we define another class of problems that does contain SAT.

We start with the notion of verification, where additional information is allowed to make it easier to compute the decision problem answer.

Definition 31. *Given a decision problem $Q : I \to \mathbb{B}$ and an algorithm A, algorithm A* verifies *Q if for any $i \in I$, $Q(i) = \top$ implies there exists a certificate c such that $A(i,c) = \top$, and $Q(i) = \bot$ implies there does not exist a certificate c such that $A(i,c) = \top$.*

In the example of the satisfiability problem we already saw that the certificate is the truth assignment. If we have the right truth assignment $\mathscr{A}_V$, then running the algorithm that implements $\mathscr{E}_{\mathscr{A}_V}$ on the formula is enough to determine if the formula is satisfiable. And when we have the certificate, the computation is fast. Next we generalize this notion of speedy verification to an arbitrary decision problem.

Definition 32. *A decision problem $Q : I \to \mathbb{B}$* can be verified in polynomial time *if there exists an algorithm A that verifies Q, the time complexity of A is $T_A(n)$ (where n is the size of i), and there exists $k \in \mathbb{N}$ such that $T_A(n) \in O(n^k)$.*

Note that the satisfiability problem can be verified in polynomial time. In particular, running the algorithm that implements $\mathscr{E}_{\mathscr{A}_V}$ on the formula involves a linear time traversal of the formula.

We can now lump all of those problems together and define our new class: $\mathscr{NP}$. We also define co-$\mathscr{NP}$.

Definition 33. *The class $\mathcal{NP}$ is the set of all decision problems that can be verified in polynomial time. The class co-$\mathcal{NP}$ is the set of all decision problems Q such that $(\neg \circ Q) \in \mathcal{NP}$.*

Since SAT can be verified in polynomial time, $\mathsf{SAT} \in \mathcal{NP}$. In addition to SAT, $\mathcal{NP}$ contains many other decision problems as well; some are difficult and some are not.

Lemma 22. $\mathcal{P} \subseteq \mathcal{NP}$.

Proof.
Suppose $Q \in \mathcal{P}$. Then there exists a polynomial time algorithm A that solves Q.
Consider the verifier A' defined by $A'(i,c) = A(i)$.
If $Q(i) = 1$ then $A(i) = 1$, and so $A'(i,c) = 1$. If $Q(i) = 0$ then $A(i) = 0$, and so for any c, $A'(i,c) = 0$; thus there does not exist a c such that $A'(i,c) = 1$.
Finally, A' is polynomial time since A is polynomial time. □

So all the (easy) decision problems in $\mathcal{P}$ are also in $\mathcal{NP}$. We would like to be able to say definitively that $\mathcal{P} \neq \mathcal{NP}$, but no one has been able to prove that result. The best we can do is focus on the hardest decision problems in $\mathcal{NP}$. To do that, we introduce a notion of "at least as hard as" to compare decision problems. If we have two decision problems Q and Q', we know that Q is hard to compute, we know that the function C is easy to compute, and we know that $Q = Q' \circ C$, then Q' must be hard to compute — for if Q' were easy to compute that would imply that Q is easy to compute!

Definition 34. *Given decision problems $Q : I \to \mathbb{B}$ and $Q' : I' \to \mathbb{B}$, Q is* polynomially reducible to *Q', or $Q \leq_{\mathsf{P}} Q'$, if there exists a polynomial time function $C : I \to I'$ such that $Q = Q' \circ C$. Note that though it is traditional to read "$Q \leq_{\mathsf{P}} Q'$" as "Q reduces to Q'," we will read "$Q \leq_{\mathsf{P}} Q'$" as "Q' is at least as hard as Q."*

So now we can compare two decision problems. We would like to compare a decision problem against a really hard one. But which one? To be sure we are comparing against a hard decision problem we will compare against *all* of them.

Definition 35. *Given a decision problem Q', Q' is $\mathcal{NP}$-*Hard *if for any $Q \in \mathcal{NP}$, $Q \leq_{\mathsf{P}} Q'$.*

It is possible that a decision problem that is $\mathcal{NP}$-Hard is so hard that it is not even in $\mathcal{NP}$. Fortunately, our "at least as hard as" relation is reflexive, so while comparing a decision problem that is in $\mathcal{NP}$ against all such problems there is no issue comparing it against itself.

Fact 16. *The relation $\leq_{\mathsf{P}}$ is reflexive.*

We refine our notion to explicitly require that the decision problem in question is in $\mathcal{NP}$.

Definition 36. *Given a decision problem Q, Q is $\mathcal{NP}$-*Complete *if $Q \in \mathcal{NP}$ and Q is $\mathcal{NP}$-*Hard.

At this point we might feel intimidated at the prospect of proving that a decision problem is $\mathcal{NP}$-Complete. Showing that it is in $\mathcal{NP}$ is not so bad, but how do we show that our decision problem is at least as hard as *every* decision problem in $\mathcal{NP}$?! One ray of hope is that our "at least as hard as" relation is transitive.

Fact 17. *The relation $\leq_{\mathsf{P}}$ is transitive.*

Once we know one decision problem is $\mathcal{NP}$-Complete, we can use that problem to verify that a our decision problem is $\mathcal{NP}$-Complete merely by showing that it is at least as hard as the known $\mathcal{NP}$-Complete problem; it follows from the transitive property that our decision problem is at least as hard as every decision problem in $\mathcal{NP}$.

But how do we find that first $\mathcal{NP}$-Complete problem? It turns out that we can use a specific form of the satisfiability problem to characterize arbitrary polynomial time verification. The following definition characterizes the specific form used.

Definition 37. *A propositional formula ψ is in* conjunctive normal form *if ψ has the form $\psi = \varphi_1 \wedge \cdots \wedge \varphi_n$, where $\varphi_i = \ell^i_1 \vee \cdots \vee \ell^i_{m_i}$ and ℓ^i_j, a literal, is either a variable or the negation of a variable.*

The example propositional formula $\psi = (x \vee \neg y) \wedge (y \vee z \vee \neg w) \wedge (\neg x \vee \neg z \vee \neg w)$ is in conjunctive normal form.

Now we can state the decision problem variation.

Problem 16 (CNFSAT)**.** *Given a propositional formula ψ over V in conjunctive normal form, is there a truth assignment $\mathcal{A}_V$ such that $\mathcal{A}_V \models \psi$?*

It turns out that we can establish that CNFSAT is $\mathcal{NP}$-Complete. It is the $\mathcal{NP}$-Complete problem that is the starting point.

Theorem 18 (Cook–Levin)**.** CNFSAT *is $\mathcal{NP}$-Complete.*

Chapter 18

Additional $\mathcal{NP}$-Complete Problems**

18.1 Introduction to Additional $\mathcal{NP}$-Complete Problems

Now that we have CNFSAT as a known $\mathcal{NP}$-Complete decision problem, we can more easily prove that other decision problems are $\mathcal{NP}$-Complete. In this chapter we study what is involved in proving a decision problem $\mathcal{NP}$-Complete. Then we turn our attention to particular $\mathcal{NP}$-Complete decision problems and their proofs. We start with a trivial example but follow up with serious ones.

18.1.1 Proving a Problem $\mathcal{NP}$-Complete

In short, proving that a decision problem is $\mathcal{NP}$-Complete involves establishing what is stipulated by the definitions. Recall that the definition of a decision problem being $\mathcal{NP}$-Complete is that it is in $\mathcal{NP}$ and it is $\mathcal{NP}$-Hard. Thus to prove that a decision problem Q is $\mathcal{NP}$-Complete it is necessary to prove both that $Q \in \mathcal{NP}$ and that Q is $\mathcal{NP}$-Hard.

To prove that a decision problem is in $\mathcal{NP}$ it is necessary to prove that it can be verified in polynomial time. That means proving that there exists a polynomial time algorithm A that verifies the problem. To prove the existence of a verification algorithm, it is enough to write one down. This example must be followed by an argument that the algorithm is, in fact, polynomial time. Remember that the algorithm must take two arguments: the instance and the certificate. The instance is simply a domain element of the problem; for example, for CNFSAT an instance is a propositional formula in conjunctive normal form. The certificate is sometimes less obvious. To identify what the certificate should be, notice what the existential quantifier concerns in the statement of the problem; for example, for CNFSAT (and other satisfiability problems) the question of existence concerns a truth assignment.

The definition says that to prove that a decision problem is $\mathcal{NP}$-Hard it is necessary to prove that it is at least as hard as every decision problem in $\mathcal{NP}$. At this point, we can establish that result by proving that our problem is at least as hard as a problem that is known to be $\mathcal{NP}$-Hard (or $\mathcal{NP}$-Complete). It will make the argument easier if we can find a decision problem that is $\mathcal{NP}$-Hard which is already similar to our problem. It is also worthwhile repeating the language of the definition to help orient the reader. This time it is necessary to show the existence of a polynomial time conversion function. Again, it is enough to write one down, but it must be followed by an argument that the conversion function is, in fact, polynomial time. Then it is necessary to prove that the functions are equal.

There are two standard approaches for showing that the functions are equal. Ultimately, they are based on the definition of function equality.

Definition 38. *Given functions $f : D \to D'$ and $g : D \to D'$, $f = g$ if for every $x \in D$, $f(x) = g(x)$.*

Since the functions we are working with are decision problems (i.e., Boolean valued functions), we can specialize the definition to this case.

Fact 18. *Given functions $f : D \to \mathbb{B}$ and $g : D \to \mathbb{B}$, $f = g$ if for every $x \in D$, $f(x) = \top$ iff $g(x) = \top$.*

The two approaches are then two different ways of expanding the "if and only if."

Corollary 11. *Given functions $f : D \to \mathbb{B}$ and $g : D \to \mathbb{B}$, $f = g$ if for every $x \in D$, $f(x) = \top$ implies $g(x) = \top$ and $g(x) = \top$ implies $f(x) = \top$.*

In this first approach, we suppose $f(x) = \top$ and show that it leads to $g(x) = \top$. Then we have an argument where we suppose $g(x) = \top$ and show that it leads to $f(x) = \top$.

Corollary 12. *Given functions $f : D \to \mathbb{B}$ and $g : D \to \mathbb{B}$, $f = g$ if for every $x \in D$, $f(x) = \top$ implies $g(x) = \top$ and $f(x) = \bot$ implies $g(x) = \bot$.*

In the second approach, we start off as in the first approach. But instead of supposing $g(x) = \top$, we suppose $f(x) = \bot$ and show that it leads to $g(x) = \bot$.

18.1.2 Initial Example

We start with a trivial example: SAT. It simply does not feel that different from CNFSAT. Further, we can even get a little more technical and observe that since the domain of SAT is larger than the domain of CNFSAT then any difficult instances from CNFSAT should also be difficult instances for SAT, so we should expect SAT to be at least as hard as CNFSAT. Nevertheless, going through the proof is instructive because we see short versions of all the arguments mentioned in the previous section.

Theorem 19. SAT *is $\mathscr{NP}$-Complete.*

Proof.

- First we show that $\mathsf{SAT} \in \mathscr{NP}$. Observe that an instance is a propositional formula ψ, a certificate is a truth assignment $\mathscr{A}_V$, and the verification algorithm $A(\psi, \mathscr{A}_V)$ is the algorithm that implements $\mathscr{E}_{\mathscr{A}_V}(\psi)$. Note that $A(\psi, \mathscr{A}_V)$ is polynomial time since the computation involves merely a linear traversal of the propositional formula ψ.

- Now we show that SAT is $\mathscr{NP}$-Hard by showing $\mathsf{CNFSAT} \leq_\mathsf{p} \mathsf{SAT}$.
  Let $Q : I \to \mathbb{B}$ be the formal problem of CNFSAT where I is the set of propositional formulas in conjunctive normal form, and let $Q' : I' \to \mathbb{B}$ be the formal problem of SAT where I' is the set of propositional formulas.
  Consider $C : I \to I'$ where $C(i) = i$.
  Clearly C is computable in polynomial time.

  - Suppose $Q(\psi) = \top$.
    Then there exists a truth assignment $\mathscr{A}_V$ such that $\mathscr{A}_V \models \psi$. Hence $Q'(\psi) = \top$. Further, since $\psi = C(\psi)$, it follows that $Q'(C(\psi)) = \top$. And so $(Q' \circ C)(\psi) = \top$.
  - Suppose $Q(\psi) = \bot$.
    Then for any truth assignment $\mathscr{A}_V$, $\mathscr{A}_V \not\models \psi$. Hence $Q'(\psi) = \bot$. Further, since $\psi = C(\psi)$, it follows that $Q'(C(\psi)) = \bot$. And so $(Q' \circ C)(\psi) = \bot$. □

18.2 3SAT

Consider one more variation on satisfiability. The motivation for this variation is that it is easier to use in proofs that are not about satisfiability.

Problem 17 (3SAT)**.** *Given a propositional formula ψ over V in conjunctive normal form such that each component φ_i contains exactly three literals, is there a truth assignment $\mathscr{A}_V$ such that $\mathscr{A}_V \models \psi$?*

The example propositional formula $\psi = (x \vee \neg y) \wedge (y \vee z \vee \neg w) \wedge (\neg x \vee \neg z \vee \neg w)$ does *not* have exactly three literals per component. However, it is possible to modify it slightly, preserving the semantics, so that it does: $\psi' = (x \vee \neg y \vee \neg y) \wedge (y \vee z \vee \neg w) \wedge (\neg x \vee \neg z \vee \neg w)$.

Theorem 20. 3SAT *is $\mathscr{NP}$-Complete.*

The proof of this theorem has the same structure as the proof that SAT is $\mathscr{NP}$-Complete, but all of the arguments are more elaborate. The reason why is that this result is much less obvious. When we considered SAT, we noticed that the domain is larger than CNFSAT. In this case, it is the opposite; the domain of 3SAT is smaller than the domain of CNFSAT. How do we know that we did not reduce the domain sufficiently to leave out all the difficult instances? That can happen. Consider a similar problem but instead of φ_i containing exactly three literals, φ_i contains exactly one literal; call this problem 1SAT. Notice that since this problem merely concerns the conjunction of literals, it is easy to solve in polynomial time. Reducing the domain sufficiently entails that 1SAT is *not* $\mathscr{NP}$-Complete.

The conversion function C is more elaborate here. Given an arbitrary formula in conjunctive normal form, C must transform it into one where each component has exactly three literals. That is easy if a component has three or fewer, but involves introducing fresh variables if there are four or more. For example, suppose $\varphi = \ell_1 \vee \ell_2 \vee \ell_3 \vee \ell_4 \vee \ell_5$, then[1] $D(\varphi) = (\ell_1 \vee \ell_2 \vee z_1) \wedge (\bar{z}_1 \vee \ell_3 \vee z_2) \wedge (\bar{z}_2 \vee \ell_4 \vee \ell_5)$ where the zs are fresh variables.

Then the proof needs to show that if φ is satisfiable, so is $D(\varphi)$ and if φ is not satisfiable, neither is $D(\varphi)$. If φ is satisfiable then at least one ℓ_i is true. For concreteness sake, suppose it is ℓ_3. To see that $D(\varphi)$ is satisfiable, pick truth values for the zs as follows. Regarding the zs to the right of ℓ_3, choose them all to be false. Notice that because of the negations the component(s) on the right will be true. Regarding the zs to the left of ℓ_3, choose them all to be true. Thus, the component(s) on the left will be true. And the component in the middle (containing ℓ_3) will be true because ℓ_3 is true. Hence $D(\varphi)$ is true. However, if φ is not satisfiable then all of the ℓs must be false. There are only two choices for z_1. Picking z_1 to be false entails that the first component is false, and so $D(\varphi)$ is also false. Picking z_1 to be true seems like a promising way of making $D(\varphi)$ true, but doing so entails also picking z_2 to be true. But in our example that entails that the last component is false. And so no matter what truth values we pick for the zs, $D(\varphi)$ is always false.

Proof.

- First we show that 3SAT $\in \mathscr{NP}$. Observe that an instance is a propositional formula ψ in conjunctive normal form such that each component contains exactly three literals, a certificate is a truth assignment $\mathscr{A}_V$, and the verification algorithm $A(\psi, \mathscr{A}_V)$ is the algorithm that implements $\mathscr{E}_{\mathscr{A}_V}(\psi)$. Note that $A(\psi, \mathscr{A}_V)$ is polynomial time since the computation involves merely a linear traversal of the propositional formula ψ.

[1] The overbar notation $\bar{z}$ expresses logical negation and means the same thing as $\neg z$.

- Next we show that CNFSAT is $\mathcal{NP}$-Hard by showing CNFSAT $\leq_p$ 3SAT.
  Let $Q : I \to \mathbb{B}$ be the formal problem of CNFSAT where I is the set of propositional formulas in conjunctive normal form, and let $Q' : I' \to \mathbb{B}$ be the formal problem of 3SAT where I' is the set of propositional formulas in conjunctive normal form such that each component contains exactly three literals.
  Consider $C : I \to I'$ where C is defined as follows.

$$\begin{aligned}
C(\varphi_1 \wedge \cdots \wedge \varphi_n) &= D(\varphi_1) \wedge \cdots \wedge D(\varphi_n) \\
D(\ell_1) &= \ell_1 \vee \ell_1 \vee \ell_1 \\
D(\ell_1 \vee \ell_2) &= \ell_1 \vee \ell_2 \vee \ell_2 \\
D(\ell_1 \vee \ell_2 \vee \ell_3) &= \ell_1 \vee \ell_2 \vee \ell_3 \\
D(\ell_1 \vee \cdots \vee \ell_m) &= (\ell_1 \vee \ell_2 \vee z_1) \wedge (\bar{z}_1 \vee \ell_3 \vee z_2) \wedge \cdots \wedge (\bar{z}_{m-3} \vee \ell_{m-1} \vee \ell_m) \quad \text{if } m > 3 \\
&\quad \text{and } \{z_1, \cdots, z_{m-3}\} \text{ is a set of fresh variables}
\end{aligned}$$

  Note that C is computable in polynomial time since it involves no more than expanding each literal by a constant factor.

  - Suppose $Q(\psi) = \top$.
    Recall that ψ is a propositional formula over V, $\psi = \varphi_1 \wedge \cdots \wedge \varphi_n$, φ_i is a propositional formula over V_i, and $\varphi_i = \ell^i_1 \vee \cdots \vee \ell^i_{m_i}$.
    Since $Q(\psi) = \top$, there exists a truth assignment $\mathscr{A}_V$ such that $\mathscr{A}_V \models \psi$. And so $\mathscr{A}_V \models \varphi_i$. Hence there exists a j such that $\mathscr{A}_V \models \ell^i_j$.

    * Suppose $m_i \leq 3$.
      Let $\hat{V}_i = V$ and $\hat{\mathscr{A}}_{\hat{V}_i} = \mathscr{A}_V$. It follows immediately that $\hat{\mathscr{A}}_{\hat{V}_i} \models D(\varphi_i)$.
    * Suppose $m_i > 3$.
      Let $\hat{V}_i = V \cup \{z_1, \cdots, z_{m_i-3}\}$. We define $\hat{\mathscr{A}}_{\hat{V}_i}(x) = \mathscr{A}_V(x)$ for every $x \in V$. To determine the truth values for the remaining variables, consider the possible values for j.

      · Suppose $j = 1$ or $j = 2$.
        We define $\hat{\mathscr{A}}_{\hat{V}_i}(z_k) = \bot$ for every k. Observe that $\hat{\mathscr{A}}_{\hat{V}_i} \models \ell^i_1 \vee \ell^i_2 \vee z_1$ because one of the literals is true, $\hat{\mathscr{A}}_{\hat{V}_i} \models \bar{z}_k \vee \ell^i_{k+2} \vee z_{k+1}$ because $\bar{z}_k$ is true, and $\hat{\mathscr{A}}_{\hat{V}_i} \models \bar{z}_{m_i-3} \vee \ell^i_{m_i-1} \vee \ell^i_{m_i}$ because $\bar{z}_{m_i-3}$ is true . Hence $\hat{\mathscr{A}}_{\hat{V}_i} \models D(\varphi_i)$.

      · Suppose $j = m_i - 1$ or $j = m_i$.
        We define $\hat{\mathscr{A}}_{\hat{V}_i}(z_k) = \top$ for every k. Observe that $\hat{\mathscr{A}}_{\hat{V}_i} \models \ell^i_1 \vee \ell^i_2 \vee z_1$ because z_1 is true, $\hat{\mathscr{A}}_{\hat{V}_i} \models \bar{z}_{k-1} \vee \ell^i_{k+1} \vee z_k$ because z_k is true, and $\hat{\mathscr{A}}_{\hat{V}_i} \models \bar{z}_{m_i-3} \vee \ell^i_{m_i-1} \vee \ell^i_{m_i}$ because one of the literals is true. Hence $\hat{\mathscr{A}}_{\hat{V}_i} \models D(\varphi_i)$.

      · Suppose $2 < j < m_i - 1$.
        We define $\hat{\mathscr{A}}_{\hat{V}_i}(z_k) = \begin{cases} \top & \text{if } k \leq j-2 \\ \bot & \text{if } k \geq j-1 \end{cases}$. Observe that $\hat{\mathscr{A}}_{\hat{V}_i} \models \ell^i_1 \vee \ell^i_2 \vee z_1$ because z_1 must be true; $\hat{\mathscr{A}}_{\hat{V}_i} \models \bar{z}_{k-2} \vee \ell^i_k \vee z_{k-1}$ because either $k = j$ so the literal is true, $k < j$ so the positive z_{k-1} is true, or $k > j$ so the negative z_{k-2} is true; and $\hat{\mathscr{A}}_{\hat{V}_i} \models \bar{z}_{m_i-3} \vee \ell^i_{m_i-1} \vee \ell^i_{m_i}$ because $\bar{z}_{m_i-3}$ must be true . Hence $\hat{\mathscr{A}}_{\hat{V}_i} \models D(\varphi_i)$.

  Let $V' = \bigcup_i \hat{V}_i$ and let $\mathscr{A}'_{V'} = \bigcup_i \hat{\mathscr{A}}_{\hat{V}_i}$. Note that the second union makes sense because the $\hat{\mathscr{A}}_{\hat{V}_i}$ agree on the set V and there is no intersection of the new variables. By construction, $\mathscr{A}'_{V'} \models C(\psi)$. Therefore $(Q' \circ C)(\psi) = \top$.

- Suppose $Q(\psi) = \bot$.
  Recall that ψ is a propositional formula over V, $\psi = \varphi_1 \wedge \cdots \wedge \varphi_n$, φ_i is a propositional formula over V_i, and $\varphi_i = \ell^i_1 \vee \cdots \vee \ell^i_{m_i}$.
  Since $Q(\psi) = \bot$, for any truth assignment $\mathscr{A}_V$, $\mathscr{A}_V \not\models \psi$. And so there exists an i such that $\mathscr{A}_V \not\models \varphi_i$.
  - Suppose $m_i \leq 3$.
    Then $D(\varphi_i)$ is a propositional formula over V. Further, $\mathscr{A}_V \not\models D(\varphi_i)$. Hence for any extension $\mathscr{A'}_{V'}$ of $\mathscr{A}_V$, $\mathscr{A'}_{V'} \not\models C(\psi)$. So in this case $(Q' \circ C)(\psi) = \bot$.
  - Suppose $m_i > 3$.
    Since $\mathscr{A}_V \not\models \varphi_i$, it must be that for every literal ℓ^i_j, $\mathscr{A}_V \not\models \ell^i_j$ (i.e., all the literals are false).
    Let $\hat{V}_i = V \cup \{z_1, \cdots, z_{m_i-3}\}$. And consider $\hat{\mathscr{A}}_{\hat{V}_i}$, an extension to $\mathscr{A}_V$.
    - Suppose $\hat{\mathscr{A}}_{\hat{V}_i}(z_1) = \top$.
      Then $\hat{\mathscr{A}}_{\hat{V}_i} \models \ell_1 \vee \ell_2 \vee z_1$. Further for the middle components, $\bar{z}_{k-1} \vee \ell_{k+1} \vee z_k$, to be satisfied, it must be that $\hat{\mathscr{A}}_{\hat{V}_i}(z_k) = \top$ for every k. However, that entails that $\hat{\mathscr{A}}_{\hat{V}_i} \not\models \bar{z}_{m_i-3} \vee \ell^i_{m_i-1} \vee \ell^i_{m_i}$. Hence $\hat{\mathscr{A}}_{\hat{V}_i} \not\models D(\varphi_i)$. Therefore for any extension $\mathscr{A'}_{V'}$ of $\hat{\mathscr{A}}_{\hat{V}_i}$, $\mathscr{A'}_{V'} \not\models C(\psi)$. So in this case $(Q' \circ C)(\psi) = \bot$.
    - Suppose $\hat{\mathscr{A}}_{\hat{V}_i}(z_1) = \bot$.
      Then it immediately follows that $\hat{\mathscr{A}}_{\hat{V}_i} \not\models \ell^i_1 \vee \ell^i_2 \vee z_1$. Hence $\hat{\mathscr{A}}_{\hat{V}_i} \not\models D(\varphi_i)$. Therefore for any extension $\mathscr{A'}_{V'}$ of $\hat{\mathscr{A}}_{\hat{V}_i}$, $\mathscr{A'}_{V'} \not\models C(\psi)$. So in this case $(Q' \circ C)(\psi) = \bot$.

  Thus we find in all cases that $Q(\psi) = \bot$ implies $(Q' \circ C)(\psi) = \bot$.

□

18.3 Vertex Cover

At this point we leave the realm of satisfiability. The vertex cover problem is a graph problem.

Definition 39. *Given an undirected graph $G = (V, E)$, a* vertex cover *is a set $C \subseteq V$ such that for any edge $e \in E$ there exists a vertex $v \in C$ such that e is incident to v.*

Problem 18 (VERTEXCOVER)**.** *Given $\langle G, k \rangle$, where G is an undirected graph and $k \in \mathbb{N}$, is there a vertex cover C for G such that $|C| = k$.*

Theorem 21. VERTEXCOVER *is $\mathscr{NP}$-Complete.*

There are many more $\mathscr{NP}$-Complete decision problems.

- Suppose $Q(\psi) = \bot$.
  Recall that ψ is a propositional formula over V, $\psi = \varphi_1 \wedge \cdots \wedge \varphi_n$, φ_i is a propositional formula over V_i, and $\varphi_i = \ell^i_1 \vee \cdots \vee \ell^i_{m_i}$.
  Since $Q(\psi) = \bot$, for any truth assignment $\mathscr{A}_V$, $\mathscr{A}_V \not\models \psi$. And so there exists an i such that $\mathscr{A}_V \not\models \varphi_i$.
  - Suppose $m_i \leq 3$.
    Then $D(\varphi_i)$ is a propositional formula over V. Further, $\mathscr{A}_V \not\models D(\varphi_i)$. Hence for any extension $\mathscr{A'}_{V'}$ of $\mathscr{A}_V$, $\mathscr{A'}_{V'} \not\models C(\psi)$. So in this case $(Q' \circ C)(\psi) = \bot$.
  - Suppose $m_i > 3$.
    Since $\mathscr{A}_V \not\models \varphi_i$, it must be that for every literal ℓ^i_j, $\mathscr{A}_V \not\models \ell^i_j$ (i.e., all the literals are false).
    Let $\hat{V}_i = V \cup \{z_1, \cdots, z_{m_i-3}\}$. And consider $\hat{\mathscr{A}}_{\hat{V}_i}$, an extension to $\mathscr{A}_V$.
    - Suppose $\hat{\mathscr{A}}_{\hat{V}_i}(z_1) = \top$.
      Then $\hat{\mathscr{A}}_{\hat{V}_i} \models \ell_1 \vee \ell_2 \vee z_1$. Further for the middle components, $\bar{z}_{k-1} \vee \ell_{k+1} \vee z_k$, to be satisfied, it must be that $\hat{\mathscr{A}}_{\hat{V}_i}(z_k) = \top$ for every k. However, that entails that $\hat{\mathscr{A}}_{\hat{V}_i} \not\models \bar{z}_{m_i-3} \vee \ell^i_{m_i-1} \vee \ell^i_{m_i}$. Hence $\hat{\mathscr{A}}_{\hat{V}_i} \not\models D(\varphi_i)$. Therefore for any extension $\mathscr{A'}_{V'}$ of $\hat{\mathscr{A}}_{\hat{V}_i}$, $\mathscr{A'}_{V'} \not\models C(\psi)$. So in this case $(Q' \circ C)(\psi) = \bot$.
    - Suppose $\hat{\mathscr{A}}_{\hat{V}_i}(z_1) = \bot$.
      Then it immediately follows that $\hat{\mathscr{A}}_{\hat{V}_i} \not\models \ell^i_1 \vee \ell^i_2 \vee z_1$. Hence $\hat{\mathscr{A}}_{\hat{V}_i} \not\models D(\varphi_i)$. Therefore for any extension $\mathscr{A'}_{V'}$ of $\hat{\mathscr{A}}_{\hat{V}_i}$, $\mathscr{A'}_{V'} \not\models C(\psi)$. So in this case $(Q' \circ C)(\psi) = \bot$.

  Thus we find in all cases that $Q(\psi) = \bot$ implies $(Q' \circ C)(\psi) = \bot$.

□

18.3 Vertex Cover

At this point we leave the realm of satisfiability. The vertex cover problem is a graph problem.

Definition 39. *Given an undirected graph $G = (V, E)$, a* vertex cover *is a set $C \subseteq V$ such that for any edge $e \in E$ there exists a vertex $v \in C$ such that e is incident to v.*

Problem 18 (VERTEXCOVER)**.** *Given $\langle G, k \rangle$, where G is an undirected graph and $k \in \mathbb{N}$, is there a vertex cover C for G such that $|C| = k$.*

Theorem 21. VERTEXCOVER *is $\mathscr{NP}$-Complete.*

There are many more $\mathscr{NP}$-Complete decision problems.