

# How to Think About Algorithms

## Loop Invariants and Recursion

Jeff Edmonds

Winter 2005, Version 0.9



There are many algorithm texts which provide lots of well polished code and proofs of correctness. Instead, we present insights, notations, and analogies to help the novice describe and think about algorithms like an expert. It is a bit like a carpenter studying hammers instead of houses. We provide both the big picture and easy step by step methods for developing algorithms, while avoiding the common pitfalls. Paradigms like loop invariants and recursion help to unify a huge range of algorithms into a few meta-algorithms. Part of the goal is to teach the students to think abstractly. Without getting bogged with formal proofs, a deeper understanding is fostered so that how and why each algorithm works is transparent. These insights are presented in a slow and clear manner accessible to any second or third year student of computer science, preparing him to find on his own innovative ways to solve problems.

The study of algorithms forms a core course in every computer science program. However, different schools need different approaches. Institutes of Technology support the students to be able to use algorithms as black boxes. Engineering schools push an encyclopedia of different algorithms with only the most superficial understanding of the underling principles and why the algorithms work. An ivy league computer science school focuses on a mathematical formalism beyond the abilities of most students. In contrast, this book is the first that presents algorithm in a slow and clear manner accessible to any second or third year student preparing him to think on his own in abstract and innovative ways and giving him the tools for understanding, describing, and developing algorithms.



Problem Solving  
Out of the Box Leaping  
Deep Thinking  
Creative Abstracting  
Logical Deducing  
With Friends Working  
Fun Having  
Fumbling & Bumbling  
Bravely Persevering  
Joyfully Succeed-  
ing





Dedicated to my father, Jack, and my sons, Joshua and Micah. May the love and the mathematics continue to flow between the generations.

# Contents

<b>Contents by Application</b>	<b>vi</b>
<b>Preface</b>	<b>ix</b>
To the Educator and the Student . . . . .	ix
Feedback Requested . . . . .	1
<b>I Relevant Mathematics</b>	<b>2</b>
<b>1 Existential and Universal Quantifiers</b>	<b>3</b>
<b>2 Logarithms and Exponentials</b>	<b>11</b>
<b>3 Asymptotic Notations and their Properties</b>	<b>14</b>
3.1 The Classic Classifications of Functions . . . . .	15
3.2 Comparing the Classes of Functions . . . . .	19
3.3 Other Useful Notations . . . . .	20
3.4 Different Levels of Detail When Classifying a Function . . . . .	21
3.5 Constructing and Deconstructing the Classes of Functions . . . . .	21
3.6 The Formal Definition of $\Theta$ and $\mathcal{O}$ Notation . . . . .	23
3.7 Formal Proofs . . . . .	27
3.8 Solving Equations by Ignoring Details . . . . .	28
3.9 Exercises . . . . .	29
<b>4 The Time (and Space) Complexity of an Algorithm</b>	<b>30</b>
4.1 Different Models of Time and Space Complexity . . . . .	31
4.2 Examples . . . . .	35
<b>5 Adding Made Easy Approximations</b>	<b>36</b>
5.1 Intuition and Techniques . . . . .	36
5.2 Proofs of the Adding Made Easy Technique . . . . .	42
5.3 Simple Analytical Functions . . . . .	48
<b>6 Recurrence Relations</b>	<b>52</b>
6.1 Relating Recurrence Relations to the Timing of Recursive Programs . . . . .	53
6.2 Summary . . . . .	54
6.3 The Classic Techniques . . . . .	58
6.4 Other Examples . . . . .	64

<b>7 Abstractions</b>	<b>68</b>
7.1 Different Representations of Algorithms . . . . .	68
7.2 Abstract Data Types (ADTs) . . . . .	70
<b>II Iterative Algorithms &amp; Loop Invariants</b>	<b>77</b>
<b>8 Abstractions, Techniques and Theory</b>	<b>78</b>
8.1 Assertions and Invariants as Boundaries between Parts . . . . .	78
8.2 An Introduction to Iterative Algorithms . . . . .	80
8.3 Examples: Quadratic Sorts . . . . .	83
8.4 The Steps in Developing an Iterative Algorithm . . . . .	84
8.5 A Formal Proof of Correctness . . . . .	92
8.5.1 Using Assertions . . . . .	92
8.5.2 Proving Correctness of Iterative Algorithms with Induction . . . . .	94
<b>9 More of The Input Loop Invariant</b>	<b>97</b>
9.1 Colouring the Plane . . . . .	97
9.2 Deterministic Finite Automaton . . . . .	99
9.3 More of the Input vs More of the Output . . . . .	106
<b>10 Implementations of Abstract Data Types</b>	<b>107</b>
10.1 Array Implementations . . . . .	107
10.2 Linked List Implementations . . . . .	109
10.3 Merging With a Queue . . . . .	115
10.4 Parsing With a Stack . . . . .	116
10.5 Implementations of Graphs, Trees, and Sets . . . . .	117
<b>11 Narrowing the Search Space</b>	<b>120</b>
11.1 Binary Search . . . . .	120
11.2 Binary Search Trees . . . . .	125
11.3 Magic Sevens . . . . .	126
<b>12 Iterative Sorting Algorithms</b>	<b>129</b>
12.1 Bucket Sort by Hand . . . . .	129
12.2 Counting Sort (A Stable Sort) . . . . .	130
12.3 Radix Sort . . . . .	132
12.4 Radix/Counting Sort . . . . .	133
<b>13 Euclid's GCD Algorithm</b>	<b>135</b>
<b>III Recursion</b>	<b>139</b>
<b>14 Abstractions, Techniques, and Theory</b>	<b>140</b>
14.1 Different Abstractions . . . . .	142
14.2 Circular Argument? Looking Forwards vs Backwards . . . . .	143
14.3 The Friends' Recursion Level of Abstraction . . . . .	144
14.4 Proving Correctness with Strong Induction . . . . .	146

14.5 The Stack Frame Levels of Abstraction . . . . .	147
<b>15 Some Simple Examples of Recursive Algorithms</b>	<b>149</b>
15.1 Sorting and Selecting Algorithms . . . . .	149
15.2 Operations on Integers . . . . .	155
15.3 Ackermann's Function . . . . .	159
<b>16 Recursion on Trees</b>	<b>161</b>
16.1 Abstractions, Techniques, and Theory . . . . .	161
16.2 Simple Examples . . . . .	163
16.3 Generalizing the Problem Solved . . . . .	168
16.4 Heap Sort and Priority Queues . . . . .	170
16.5 Representing Expressions with Trees . . . . .	177
16.6 Pretty Tree Print . . . . .	180
<b>17 Recursive Images</b>	<b>184</b>
17.1 Drawing a Recursive Image from a Fixed Recursive and Base Case Images . . . . .	184
17.2 Randomly Generating a Maze . . . . .	187
<b>18 Parsing with Context-Free Grammars</b>	<b>189</b>
<b>IV Optimization Problems</b>	<b>196</b>
<b>19 Definition of Optimization Problems</b>	<b>197</b>
<b>20 Graph Search Algorithms</b>	<b>199</b>
20.1 A Generic Search Algorithm . . . . .	199
20.2 Breadth-First Search/Shortest Paths . . . . .	204
20.3 Dijkstra's Shortest-Weighted Paths Algorithm . . . . .	208
20.4 Depth-First Search . . . . .	212
20.5 Recursive Depth-First Search . . . . .	215
20.6 Linear Ordering of a Partial Order . . . . .	217
<b>21 Network Flows</b>	<b>220</b>
21.1 A Hill Climbing Algorithm with a Small Local Maximum . . . . .	221
21.2 The Primal-Dual Hill Climbing Method . . . . .	226
21.3 The Steepest Assent Hill Climbing Algorithm . . . . .	233
21.4 Linear Programming . . . . .	237
<b>22 Greedy Algorithms</b>	<b>242</b>
22.1 Abstractions, Techniques and Theory . . . . .	242
22.2 Examples . . . . .	250
22.2.1 A Fixed Priority Greedy Algorithm for the Job/Event Scheduling Problem .	250
22.2.2 An Adaptive Priority Greedy Algorithm for the Interval Cover Problem .	253
22.2.3 The Minimum-Spanning-Tree Problem . . . . .	256

<b>23 Recursive Backtracking and Dynamic Programming Algorithms</b>	<b>262</b>
23.1 Recursive Backtracking Algorithms . . . . .	262
23.2 The Steps in Developing a Recursive Backtracking . . . . .	266
23.3 The Steps in Developing a Dynamic Programming Algorithm . . . . .	270
23.4 More Theory . . . . .	275
23.4.1 The Question For the Little Bird . . . . .	275
23.4.2 Subinstances and Subsolutions . . . . .	277
23.4.3 The Set of Subinstances . . . . .	279
23.4.4 Decreasing Time and Space . . . . .	282
23.4.5 Counting The Number of Solutions . . . . .	285
23.4.6 The New Code . . . . .	286
<b>24 Examples of Dynamic Programs</b>	<b>288</b>
24.1 The Printing Neatly Problem . . . . .	288
24.2 The Longest Common Subsequence Problem . . . . .	292
24.3 More of the Input Iterative Loop Invariant Perspective . . . . .	297
24.4 A Greedy Dynamic Program: The Weighted Job/Activity Scheduling Problem . . . . .	299
24.5 Exponential Time? The Integer-Knapsack Problem . . . . .	302
24.6 The Solution Viewed as a Tree: Chains of Matrix Multiplications . . . . .	305
24.7 Generalizing the Problem Solved: Best AVL Tree . . . . .	310
24.8 All Pairs Shortest Paths with Negative Cycles . . . . .	312
24.9 All Pairs Using Matrix Multiplication . . . . .	322
24.10 Parsing with Context-Free Grammars . . . . .	323
24.11 More Examples . . . . .	328
<b>25 Examples of Recursive Backtracking</b>	<b>329</b>
25.1 Pruning Branches . . . . .	329
25.2 Satisfiability . . . . .	330
25.3 Scrabble . . . . .	333
25.4 Queens . . . . .	334
<b>26 Reductions</b>	<b>335</b>
26.1 Satisfiability Is At Least As Hard As Any Optimization Problem . . . . .	336
26.2 Steps to Prove NP-Completeness . . . . .	339
26.3 3-Colouring is NP-Complete . . . . .	346
26.4 Integer Factorization and Cryptography are Difficult . . . . .	350
26.5 An Algorithm for Bipartite Matching using the Network Flow Algorithm . . . . .	351
<b>27 Randomized Algorithms</b>	<b>354</b>
27.1 Introduction to Probability Theory . . . . .	354
27.2 Using Randomness to Hide The Worst Cases . . . . .	361
27.3 Solutions of Optimization Problems with a Random Structure . . . . .	364
<b>28 Lower Bounds</b>	<b>367</b>
28.1 Deterministic Worst-Case Time Complexity . . . . .	367
28.2 Information Theoretic Lower Bound for Sorting . . . . .	368



# Contents by Application

Though this book is organized with respect to the algorithmic technique used, you can read it in almost any order. Another reasonable order is with respect to application. The only restriction is that you read the “techniques and theory” section of a chapter before you read any of the examples within it.

## Relevant Mathematics

Existential and Universal Quantifiers .....	1
Adding Made Easy .....	3.1
Recurrence Relations .....	5.1
Information Theory .....	28.2
Probability Theory .....	27.1

## Computational Complexity of a Problem

The Time (and Space) Complexity of an Algorithm .....	4
Asymptotic Notations and their Properties .....	1.2
Formal Proof of Correctness .....	8.5,14.4
The Integer-Knapsack Problem .....	24.5
Expressing Time Complexity with Existential and Universal Quantifiers .....	28.1
Lower Bounds for Sorting using Information Theory .....	28.2
Nondeterminism .....	23.1,26.2
NP-Completeness .....	26

## Data Structures

Stacks and Queues .....	7.2,10
Priority Queues .....	7.2,16.4
Sets .....	7.2,10.5
Union-Find Set Systems .....	7.2,10.5,22.2.3
Dictionaries .....	7.2
Graphs .....	7.2,10.5
Trees .....	7.2,10.5,16
Binary Search Trees .....	10.5,10.5

AVL trees .....	10.5,16.3,24.7
-----------------	----------------

## Algorithmic Techniques

Loop Invariants for Iterative Algorithms .....	8
Recursion .....	14
Hill Climbing .....	21
Greedy Algorithms .....	22
Dynamic Programming .....	23
Randomized Algorithms .....	27
Reductions (Upper and Lower Bounds) .....	26

## Sorting and Selection

Iterative Quadratic Sorts .....	8.3
Binary Search .....	11.1
Recursive Sorting and Selecting Algorithms .....	15.1
Heap Sort and Priority Queues .....	16.4
Linear Radix/Counting Sort .....	12
Lower Bounds for Sorting using Information Theory .....	28.2

## Numerical Calculations

Euclid's GCD Algorithm .....	13
Mod and Adding Integers .....	9.2
Multiplication, Powers, and FFT .....	15.2
Representing Expressions with Trees .....	16
Chains of Matrix Multiplications .....	24.6
Primality .....	27.2,26.4

## Graph Algorithms

Data Structures .....	4
A Generic Search Algorithm .....	20.1
Breadth-First Search/Shortest Paths .....	20.2
Shortest-Weighted Paths .....	20.3
Depth-First Search .....	20.4
Linear Ordering of a Partial Order .....	20.6
Network Flows .....	21
Recursive Depth-First Search .....	20.5
The Minimum-Spanning-Tree Problem .....	22.2.3
All Pairs Shortest Paths .....	24.8
Expander Graphs .....	27.3

Max Cut .....	27.3
3-Colouring .....	26.3

## Parsing with Context-Free Grammars

Recursive Look Ahead Once .....	18
Dynamic Programming Parsing .....	24.10

# Preface

## To the Educator and the Student

These are a preliminary draft of notes to be used in a twelve week, second or third year algorithms course. The goal is to teach the students to think abstractly about algorithms and about the key algorithmic techniques used to develop them.

**Meta-Algorithms:** There are so many algorithms that the students must learn that some get overwhelmed. In order to facilitate their understanding, most textbooks cover the standard themes of iterative algorithms, recursion, greedy algorithms, and dynamic programming. However, generally once it comes to presenting the algorithms and their proofs of correctness, these concepts are hidden within optimized code and slick proofs. One goal of these notes is to present a uniform and clean way of thinking about algorithms. We do this by focusing on the structure and proof of correctness of the “meta-algorithms” *Iterative* and *Recursive* and within these the meta-algorithms *Greedy* and *Dynamic Programming*. By learning these and their proofs of correctness most other algorithms follow easily. The challenge is that thinking about meta-algorithms requires a great deal of abstract thinking.

**Abstract Thinking:** Students are very good at learning how to apply a concrete algorithm to a concrete input instance. They tend, however, to find it difficult to think abstractly about the problem. I maintain, however, that the more abstractions a person has from which to view the problem, the deeper his understanding of it will be, the more tools he will have at his disposal, and the better prepared he will be to design his own innovative ways to solve the problems that may arise in other courses or in the work place. Hence, we present a number of different notations, analogies, and paradigms within which to develop and to think about algorithms.



**Way of Thinking:** People who develop algorithms have various ways of thinking and intuition that tend not to get taught. The assumption, I suppose, is that these cannot be taught but must be figured out on ones own. This text attempts to teach students to think like a designer of algorithms.

**Not a Reference Book:** Our intention is not to teach a specific selection of algorithms for specific purposes. Hence, the notes are not organized according to the application of the algorithms

but according to the techniques and abstractions used to develop them. For this, one may also want refer to *Introduction to Algorithms* by Cormen, Leiserson, and Rivest.

**Developing Algorithms:** The goal is not to present completed algorithms in a nice clean package; but to go slowly through every step of the development. Many false starts have been added. The hope is that this will help students learn to develop algorithms on their own. The difference is a bit like the difference between studying carpentry by looking at houses and by looking at hammers.

**Proof of Correctness:** Our philosophy is not to follow an algorithm with a formal proof that it is correct. Instead, this course is about learning how to think about, develop, and describe algorithms in such way that their correctness is transparent. For example, I strongly believe that loop invariants and recursion are very important in the development and understanding of algorithms.

**Big Picture vs Small Steps:** For each topic, I attempt to give both the big picture and to break it down into easily understood steps.

**Level of Presentation:** This material is difficult: there is no getting around that. I have tried to figure out where confusion may arise and to cover these points in more detail.

**Long and Informal:** I apologize how many pages are spent covering the material. This is because, though we are covering difficult material, I wanted the presentation to be slow gentle and without assumptions. In the end, I hope that it will take you less time and not more to read the material.

**Point Form:** The text is organized into blocks each containing a title and a single thought. Hopefully, this will make the notes easier to lecture and study from.

**Prerequisites:** The course assumes that the students have completed a first year programming course and have a general mathematical maturity. The first chapter covers much of the mathematics that will be needed for the course.

**Homework Questions:** A few homework questions are included. I am hoping to develop many more along with their solutions. Contributions are welcome.

**Read Ahead:** The student is expected to read the material **before** the lecture. This will facilitate productive discussion during class.

**Explaining:** To be able to prove yourself on a test or on the job, you need to be able to explain the material well. In addition, explaining it to someone else is the best way to learn it yourself. Hence, I highly recommend spending a lot of time explain the material over and over again out loud to yourself, to each other, and to your stuffed bear.

**Dreaming:** When designing an algorithm, the tendency is to start coding. When studying, the tendency is to read or to do problems. Though these are important, I would like to emphasize the importance of thinking, even day dreaming, about the material. This can be done while going along with your day, while swimming, showering, cooking, or laying in bed. Ask questions. Why is it done this way and not that way? Invent other algorithms for solving a problem. Then look for input instances for which your algorithm gives the wrong answer. Mathematics is not all linear thinking. If the essence of the material, what the questions are really asking, is allowed to seep down into your subconscious then with time little thoughts will begin to percolate up. Pursue these ideas. Sometimes even flashes of inspiration appear.



## Feedback Requested

Using the feedback that I have received, I have worked hard this last month passing over 90% of the notes. Shortening it when possible. Adding more concrete descriptions is some place, more intuition in others. One message for you here is that I may have introduced more typos in this last hectic edit so please help me with these. Another message is that your suggestions are helpful so please keep giving them to me. What has been helpful to you? What did you find hard to follow? What would make it easier to follow? Please send feedback to [jeff@cs.yorku.ca](mailto:jeff@cs.yorku.ca). Thanks.

## Acknowledgments

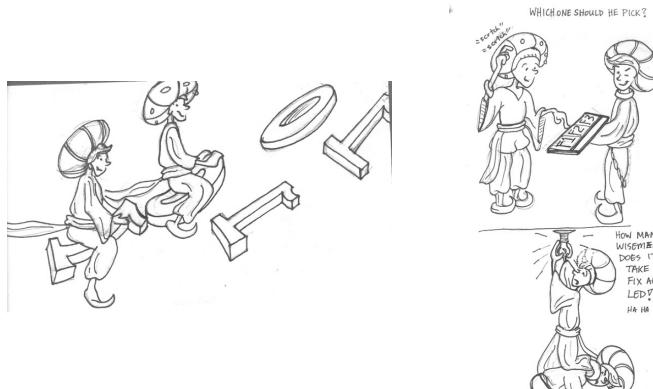
I would like to thank Andy Mirzaian, with whom I co-taught Algorithms with for many years, for many interesting problems and teaching techniques, Franck van Breugel for many useful discussions, Jennifer Wolfe for a fantastic editing job, and Carolin Taron and Gara Pruesse for support.

# **Part I**

# **Relevant Mathematics**

# Chapter 1

## Existential and Universal Quantifiers



Existential and universal quantifiers provide an extremely useful language for making formal statements. You must understand them. A game between a prover and a verifier is a level of abstraction within which it is easy to understand and prove such statements.

**The *Loves* Example:** Suppose the relation (predicate)  $Loves(b, g)$  means that the boy  $b$  loves the girl  $g$ .

### Expression

- $\exists g Loves(Sam, g)$
- $\forall g Loves(Sam, g)$
- $\exists b \forall g Loves(b, g)$
- $\forall b \exists g Loves(b, g)$
- $\exists g \forall b Loves(b, g)$
- $\exists b \exists g ( Loves(b, g) \text{ and } \neg Loves(g, b) )$

### Meaning

- “Sam loves some girl.”
- “Sam loves every girl.”
- “Some boy loves every girl.”
- “Every boy loves some girl.”
- “Some girl is loved by every boy.”
- “Some boy loves in vain.”



**Definition of Relation:** A relation like  $Loves(b, g)$  states for every pair of objects  $b = Sam$  and  $g = Mary$  that the relation either holds between them or does not. Though we will use the word *relation*,  $Loves(b, g)$  is also considered to be a *predicate*. The difference is that a predicate takes only one argument and hence focuses on whether the property is *true* or *false* about the given tuple  $\langle b, g \rangle = \langle Sam, Mary \rangle$ .

**Representations:** Relations (predicates) can be represented in a number of ways.

**Functions:** A relation can be viewed as a function mapping tuples of objects either to *true* to *false*, namely  $Loves : \{b \mid b \text{ is a boy}\} \times \{g \mid g \text{ is a girl}\} \Rightarrow \{\text{true}, \text{false}\}$  or more generally  $Loves : \{p_1 \mid p_1 \text{ is a person}\} \times \{p_2 \mid p_2 \text{ is a person}\} \Rightarrow \{\text{true}, \text{false}\}$ .

**Set of Tuples:** Alternatively, it can be viewed as a set containing the tuples for which it is true, namely  $Loves = \{\langle Sam, Mary \rangle, \langle Sam, Ann \rangle, \langle Bob, Ann \rangle, \dots\}$ .  $\langle Sam, Mary \rangle \in Loves$  iff  $Loves(Sam, Mary)$  is true.

**Directed Graph Representation:** If the relation only has two arguments, it can be represented by a directed graph. The nodes consist of the objects in the domain. We place a directed edge  $\langle b, g \rangle$  between pairs for which the relation is true. If the domains for the first and second objects are disjoint, then the graph is bipartite. Of course, the *Loves* relation could be defined to include  $Loves(Sam, Bob)$ . One would then need to also consider  $Loves(Sam, Sam)$ . See Figure 1.1 below.

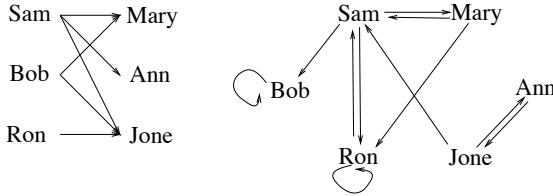


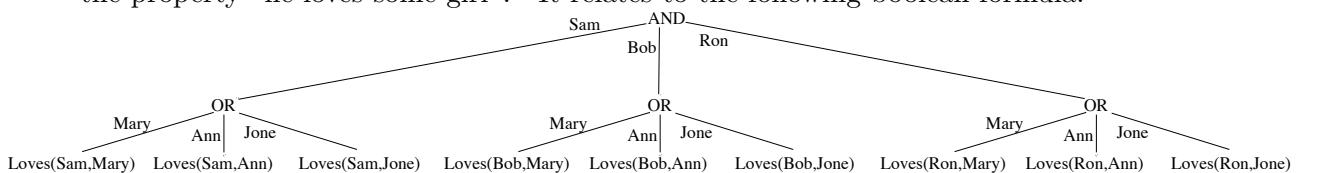
Figure 1.1: A directed graph representation of the *Loves* relation.

**Quantifiers:** You will be using the following quantifiers and properties.

**The Existence Quantifier:** The exists quantifier  $\exists$  means that there is at least one object in the domain with the property. This quantifier relates the boolean operator *OR*. For example,  $\exists g Loves(Sam, g) \equiv [Loves(Sam, Mary) OR Loves(Sam, Ann) OR Loves(Sam, Jone) OR \dots]$ .

**The Universal Quantifier:** The universal quantifier  $\forall$  means that all of the objects in the domain have the property. It relates the boolean operator *AND*. For example,  $\forall g Loves(Sam, g) \equiv [Loves(Sam, Mary) AND Loves(Sam, Ann) AND Loves(Sam, Jone) AND \dots]$ .

**Combining Quantifiers:** Quantifiers can be combined. The order of operations is such that  $\forall b \exists g Loves(b, g)$  is understood to be bracketed as  $\forall b [\exists g Loves(b, g)]$ , namely “Every boy has the property “he loves some girl”.” It relates to the following boolean formula.



**Order of Quantifiers:** The order of the quantifiers matters. For example,  $\forall b \exists g \text{ Loves}(b, g)$  and  $\exists g \forall b \text{ Loves}(b, g)$  mean different things. The second one states that “The same girl is loved by every boy.” To be true, there needs to be a Marilyn Monroe sort of girl that all the boys love. The first statement says that “Every boy loves some girl.” A Marilyn Monroe sort of girl will make this statement true. However, it is also true in a monogamous situation in which every boy loves a different girl. Hence, the first statement can be true in more different ways than the second one. In fact, the second statement implies the first one, but not vice versa.

**Definition of Free and Bound Variables:** As I said, the statement  $\exists g \text{ Loves}(Sam, g)$  means “Sam loves some girl.” This is a statement about Sam. Similarly, the statement  $\exists g \text{ Loves}(b, g)$  means “ $b$  loves some girl.” This is a statement about the boy  $b$ . Whether the statement is true depends on which boy  $b$  is referring to. The statement is *not* about the girl  $g$ . The variable  $g$  is used as a local variable (similar to `for(i = 1; i <= 10; i++)`) to express “some girl.” In this expression, we say that the variable  $g$  is *bound*, while  $b$  is *free*, because  $g$  has a quantifier and  $b$  does not.

**Defining Other Relations:** You can define other relations by giving an expression with free variables. For example, you can define the relations  $\text{LovesSomeone}(b) \equiv \exists g \text{ Loves}(b, g)$ .

**Building Expressions:** Suppose you wanted to state that “Every girl has been cheated on” using the *Loves* relation. It may be helpful to break the problem into three steps.

**Step 1) Assuming Other Relations:** Suppose you have the relation *Cheats(Sam, Mary)*, indicating that “Sam cheats on Mary.” How would you express the fact that “Every girl has been cheated on”? The advantage of using this function is that we can focus on this one part of the statement. (Note that we are not claiming that every boy cheats. For example, one boy may have broken every girl’s heart. Nor are we claiming that any girl has been cheated on in every relationship she has had.)

Given this, the answer is  $\forall g \exists b \text{ Cheats}(b, g)$ .

**Step 2) Constructing the Other Predicate:** Here we do not have a *Cheats* function. Hence, we must construct a sentence from the *loves* function stating that “Sam cheats on Mary.”

Clearly, there must be another girl involved besides Mary, so let’s start with  $\exists g'$ . Now, in order for cheating to occur, who needs to love whom? (For simplicity’s sake, let’s assume that cheating means loving more than one person at the same time.) Certainly, Sam must love the other girl. He must also love Mary. If he did not love her, then he would not be cheating on her. Must Mary love Sam? No. If Sam tells Mary he loves her dearly and then a moment later he tells Sue he loves *her* dearly, then he has cheated on Mary regardless of how Mary feels about him. Therefore, Mary does not have to love Sam. In conclusion, we might define  $\text{Cheats}(Sam, Mary) \equiv \exists g' (\text{Loves}(Sam, Mary) \text{ and } \text{Loves}(Sam, g'))$ .

However, we have made a mistake here. In our example, the other girl and Mary cannot be the same person. Hence, we must define the relation as  $\text{Cheats}(Sam, Mary) \equiv \exists g' (\text{Loves}(Sam, Mary) \text{ and } \text{Loves}(Sam, g') \text{ and } g' \neq Mary)$ .

**Step 3) Combining the Parts:** Combining the two relations together gives you  $\forall g \exists b \exists g' (\text{Loves}(b, g) \text{ and } \text{Loves}(b, g') \text{ and } g' \neq g)$ . This statement expresses that “Every girl has been cheated on.” See Figure 1.2.

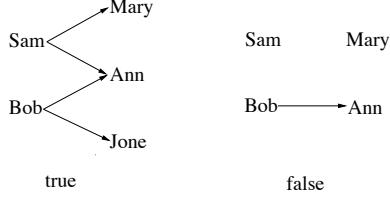


Figure 1.2: Consider the statement  $\forall g \exists b \exists g' (\text{Loves}(b, g) \text{ and } \text{Loves}(b, g') \text{ and } g \neq g')$ , “Every girl has been cheated on.” On the left is an example of a situation in which the statement is true, and on the right is one in which it is false.

**Polyamory:** One problem with such simple models of the world is that they make assumptions. In reality, Sam is not “cheating” on Mary by loving someone else if this is not against their agreement. Using the word “cheating” lays on a negative connotation and an assumption that Mary is not happy with the situation. She may in fact believe that more love is better.

**The Domain of a Variable:** Whenever you state  $\exists g$  or  $\forall g$ , there must be an understood set of values that the variable  $g$  might take on. This set is called the *domain* of the variable. It might be explicitly given or implied, but it must be understood. Here the domain is “the” set of girls. You must make clear whether this means all girls in the room, all the girls currently in the world, or all girls that have ever existed.

The statements  $\exists g P(g)$  and  $\forall g P(g)$  do not say anything about a particular girl. Instead, they say something about the domain of girls:  $\exists g P(g)$  states that there is at least one girl from the domain that has the property  $P$  and  $\forall g P(g)$  states that all girls from the domain have the property  $P$ . Whether the statement is true depends on the domain. For example,

$$\forall x \exists y x \times y = 1$$

asks whether every value has an inverse. Whether this is true depends on the domain. It is certainly not true of the domain of integers. For example, two does not have an integer inverse. It seems to be true of the domain of reals. Be careful, however; zero does not have an inverse. It would be better to write.

$$\forall x \neq 0, \exists y x \times y = 1$$

or equivalently

$$\forall x \exists y (x \times y = 1 \text{ OR } x = 0).$$

**The Negation of a Statement:** The negation of a statement is formed by putting a negation on the left-hand side. (Brackets sometimes help.) A negated statement, however, is best understood by moving the negation as deep (as far right) into the statement as possible. This is done as follows.

**Negating *AND* and *OR*:** A negation on the outside of an *AND* or an *OR* statement can be moved deeper into the statement using De Morgan’s law. Recall that the *AND* is replaced by an *OR* and the *OR* is replaced with an *AND*.

$\neg (\text{Loves}(S, M) \text{ AND } \text{Loves}(S, A))$  iff  $\neg \text{Loves}(S, M) \text{ OR } \neg \text{Loves}(S, A)$ : The negation of “Sam loves Mary and Ann” is “Either Sam does not love Mary or he does not love Ann.” He can love one of the girls, but not both.

A common mistake is to make the negation be  $\neg \text{Loves}(Sam, Mary) \text{ AND } \neg \text{Loves}(Sam, Ann)$ . However, this says that “Sam loves neither Mary nor Ann.”

$\neg (\text{Loves}(S, M) \text{ OR } \text{Loves}(S, A))$  iff  $\neg \text{Loves}(S, M) \text{ AND } \neg \text{Loves}(S, A)$ : The negation of “Sam either loves Mary or he loves Ann” is “Sam does not love Mary and he does not love Ann.”

**Negating Quantifiers:** Similarly, a negation can be moved past one or more quantifiers either to the right or to the left. However, you must then change these quantifiers from existential to universal and vice versa.

$\neg (\exists g \text{ Loves}(Sam, g))$  iff  $\forall g \neg \text{Loves}(Sam, g)$ : The negation of “There is a girl that Sam loves” is “There are no girls that Sam loves” or “Every girl is not loved by Sam.” A common mistake is to state the negation as  $\exists g \neg \text{Loves}(Sam, g)$ . However, this says that “There is a girl that is not loved by Sam.”

$\neg (\forall g \text{ Loves}(Sam, g))$  iff  $\exists g \neg \text{Loves}(Sam, g)$ : The negation of “Sam loves every girl” is “There is a girl that Sam does not love.”

$\neg (\exists b \forall g \text{ Loves}(b, g))$  iff  $\forall b \neg (\forall g \text{ Loves}(b, g))$  iff  $\forall b \exists g \neg \text{Loves}(b, g)$ : The negation of “There is a boy that loves every girl” is “There are no boys that love every girl” or “For every boy, it is not the case that he loves every girl” or “For every boy, there is a girl that he does not love.”

$\neg (\exists g_1 \exists g_2 \text{ Loves}(Sam, g_1) \text{ AND } \text{Loves}(Sam, g_2) \text{ AND } g_1 \neq g_2)$   
iff  $\forall g_1 \forall g_2 \neg (\text{Loves}(Sam, g_1) \text{ AND } \text{Loves}(Sam, g_2) \text{ AND } g_1 \neq g_2)$   
iff  $\forall g_1 \forall g_2 \neg \text{Loves}(Sam, g_1) \text{ OR } \neg \text{Loves}(Sam, g_2) \text{ OR } g_1 = g_2$ :

The negation of “There are two (distinct) girls that Sam loves” is “Given any pair of (distinct) girls, Sam does not love both” or “Given any pair of girls, either Sam does not love the first or he does not love the second, or you gave me the same girl twice.”

**The Domain Does Not Change:** The negation of  $\exists x \geq 5, x + 2 = 4$  is  $\forall x \geq 5, x + 2 \neq 4$ . The negation is NOT  $\exists x < 5 \dots$ . The reason is that both the statement and its negation are asking a question about numbers greater than 5. Is there or is there not a number with the property such that  $x + 2 = 4$ ?

**Proving a Statement True:** There are a number of seemingly different techniques for proving that an existential or universal statement is true. The core of all these techniques, however, is the same. Personally, I like to view the proof as a strategy for winning a game against an adversary.

#### Techniques for Proving $\exists g \text{ Loves}(Sam, g)$ :

**Proof by Example or by Construction:** The classic technique to prove that something with a given property exists is by example. You either directly provide an example, or you describe how to construct such an object. Then you prove that your example has the property. For the above statement, the proof would state “Let  $g'$  be the girl Mary” and then would prove that “Sam loves Mary.”

**Proof by Adversarial Game:** Suppose you claim to an adversary that “There is a girl that Sam loves.” What will the adversary say? Clearly he challenges, “Oh, yeah! Who?” You then meet the challenge by producing a specific girl  $g'$  and proving that  $\text{Loves}(\text{Sam}, g')$ , that is that Sam loves  $g'$ . The statement is true if you have a strategy guaranteed to beat any adversary in this game.

- If the statement is true, then you can produce such a girl  $g'$ .
- If the statement is false, then you will not be able to.

### Techniques for Proving $\forall g \text{ Loves}(\text{Sam}, g)$ :

**Proof by Example Does NOT Work:** Proving that Sam loves Mary is interesting, but it does not prove that he loves all girls.

**Proof by Case Analysis:** The laborious way of proving that Sam loves all girls is to consider each and every girl, one at a time, and prove that Sam loves her. This method is impossible if the domain of girls is infinite.

**Proof by “Arbitrary” Example:** The classic technique to prove that every object from some domain has a given property is to let some symbol represent an arbitrary object from the domain and then to prove that that object has the property. Here the proof would begin “Let  $g'$  be any arbitrary girl.” Because we don’t actually know which girl  $g'$  is, we must either prove  $\text{Loves}(\text{Sam}, g')$  (1) simply from the properties that  $g'$  has because she is a girl or (2) go back to doing a case analysis, considering each girl  $g'$  separately.

**Proof by Adversarial Game:** Suppose you claim to an adversary that “Sam loves every girl.” What will the adversary say? Clearly he challenges, “Oh, yeah! What about Mary?” You meet the challenge by proving that Sam loves Mary. In other words, the adversary provides a girl  $g'$ . You win if you can prove that  $\text{Loves}(\text{Sam}, g')$ .

The only difference between this game and the one for existential quantifiers is who provides the example. Interestingly, the game only has one round. The adversary is only given one opportunity to challenge you.

A proof of the statement  $\forall g \text{ Loves}(\text{Sam}, g)$  consists of a strategy for winning the game. Such a strategy takes an arbitrary girl  $g'$ , provided by the adversary, and proves that “Sam loves  $g'$ .” Again, because we don’t actually know *which* girl  $g'$  is, we must either prove (1) that  $\text{Loves}(\text{Sam}, g')$  simply from the properties that  $g'$  has because she is a girl or (2) go back to doing a case analysis, considering each girl  $g'$  separately.

- If the statement  $\forall g \text{ Loves}(\text{Sam}, g)$  is true, then you have a strategy. No matter how the adversary plays, no matter which girl  $g'$  he gives you, Sam loves her. Hence, you can win the game by proving that  $\text{Loves}(\text{Sam}, g')$ .
- If the statement is false, then there is a girl  $g'$  that Sam does not love. Any true adversary (and not just a friend) will produce this girl and you will lose the game. Hence, you cannot have a winning strategy.

**Proof by Contradiction:** A classic technique for proving the statement  $\forall g \text{ Loves}(\text{Sam}, g)$  is proof by contradiction. Except in the way that it is expressed, it is exactly the same as the proof by an adversary game.

By way of contradiction (BWOC) assume that the statement is false, i.e.,  $\exists g \neg \text{Loves}(\text{Sam}, g)$  is true. Let  $g'$  be some such girl that Sam does not love.

Then you must prove that in fact Sam *does* love  $g'$ . This contradicts the statement that Sam does not love  $g'$ . Hence, the initial assumption is false and  $\forall g \text{ Loves}(Sam, g)$  is true.

**Proof by Adversarial Game for More Complex Statements:** As I said, I like viewing the proof that an existential or universal statement is true as a strategy for a game between a prover and an adversary. The advantage to this technique is that it generalizes into a nice game for arbitrarily long statements.

#### The Steps of Game:

**Left to Right:** The game moves from left to right, providing an object for each quantifier.

**Prover Provides  $\exists b$ :** You, as the prover, must provide any existential objects.

**Adversary Provides  $\forall g$ :** The adversary provides any universal objects.

**To Win, Prove the Relation  $\text{Loves}(b', g')$ :** Once all the objects have been provided, you (the prover) must prove that the innermost relation is in fact true. If you can, then you win. Otherwise, you lose.

**Proof Is a Strategy:** A proof of the statement consists of a strategy such that you win the game no matter how the adversary plays. For each possible move that the adversary takes, such a strategy must specify what move you will counter with.

**Negations in Front:** To prove a statement with a negation in the front of it, first put the statement into “standard” form with the negation moved to the right. Then prove the statement in the same way.

#### Examples:

**$\exists b \forall g \text{ Loves}(b, g)$ :** To prove that “There is a boy that loves every girl”, you must produce a specific boy  $b'$ . Then the adversary, knowing your boy  $b'$ , tries to prove that  $\forall g \text{ Loves}(b', g)$  is false. He does this by providing an arbitrary girl  $g'$  that he hopes  $b'$  does not love. You must prove that “ $b'$  loves  $g'$ .”

**$\neg (\exists b \forall g \text{ Loves}(b, g))$  iff  $\forall b \exists g \neg \text{Loves}(b, g)$ :** With the negation moved to the right, the first quantifier is universal. Hence, the adversary first produces a boy  $b'$ . Then, knowing the adversary’s boy, you produce a girl  $g'$ . Finally, you prove that  $\neg \text{Loves}(b', g')$ .

Your proof of the statement could be viewed as a function  $G$  that takes as input the boy  $b'$  given by the adversary and outputs the girl  $g' = G(b')$  countered by you. Here,  $g' = G(b')$  is an example of a girl that boy  $b'$  does not love. The proof must prove that  $\forall b \neg \text{Loves}(b, G(b))$

**Exercise 1.0.1** Let  $\text{Loves}(b, g)$  denote that boy  $b$  loves girl  $g$ . If Sam loves Mary and Mary does not love Sam back, then we say that “Sam loves in vain.”

1. Express the following statements using universal and existential quantifiers. Move any negations to the right.

- (a) “Sam has loved in vain.”
- (b) “There is a boy who has loved in vain.”
- (c) “Every boy has loved in vain.”
- (d) “No boy has loved in vain.”

2. For each of the above statements and each of the two relations below either prove that the statement is true for the relation or that it is false.

Sam

Sam —————→ Mary

Mary  
Bob ↗ ↘

true

false

**Exercise 1.0.2** Consider the following game, Ping. There are two rounds. Player-A goes first. Let  $m_1^A$  denote his first move. Player-B goes next. Let  $m_1^B$  denote his move. Then player-A goes  $m_2^A$  and player-B goes  $m_2^B$ . The relation  $AWins(m_1^A, m_1^B, m_2^A, m_2^B)$  is true iff player-A wins with these moves.

1. Use universal and existential quantifiers to express the fact that player-A has a strategy in which he wins no matter what player-B does.
2. What steps are required in the Prover/Adversary technique to prove this statement?
3. What is the negation of the above statement in standard form?
4. What steps are required in the Prover/Adversary technique to prove this negated statement?

## Chapter 2

# Logarithms and Exponentials

Logarithms  $\log_2(n)$  and exponentials  $2^n$  arise often in this course and should be understood.

**Uses:** There are three main reasons for these to arise.

**Divide Logarithmic Number of Times:** Many algorithms repeatedly cut the input instance in half. A classic example is binary search. If you take something of size  $n$  and you cut it in half; then you cut one of these halves in half; and one of these in half; and so on. Even for a very large initial object, it does not take very long until you get a piece of size below 1. The number of times that you need to cut it is denoted by  $\log_2(n)$ . Here the base 2 is because you are cutting them in half. If you were to cut them into thirds, then the number of times to cut is denoted by  $\log_3(n)$ .

**A Logarithmic Number of Digits:** Logarithms are also useful because writing down a given integer value  $n$  requires  $\lceil \log_{10}(n+1) \rceil$  decimal digits. For example, suppose that  $n = 1,000,000 = 10^6$ . You would have to divide this number by 10 six times to get to 1. Hence, by our previous definition,  $\log_{10}(n) = 6$ . This, however, is the number of zeros, not the number of digits. We forgot the leading digit 1. The formula  $\lceil \log_{10}(n+1) \rceil = 7$  does the trick. For the value  $n = 6,372,845$ , the number of digits is given by  $\log_{10}(6,372,846) = 6.804333$ , rounded up is 7. Being in computer science, we store our values using bits. Similar arguments give that  $\lceil \log_2(n+1) \rceil$  is the number of bits needed.

**Exponential Search:** Suppose a solution to your problem is represented by  $n$  digits. There are  $10^n$  such strings of  $n$  digits. One way to count them is by computing the number of choices needed to choose one. There are 10 choices for the first digit. For each of those, there are 10 choices for the second. This gives  $10 \times 10 = 100$  choices. Then for each of these 100 choices, there are 10 choices for the third, and so on. Another way to count them is that there are  $1,000,000 = 10^6$  integers represented by 6 digits, namely 000,000 up to 999,999. The difficulty in there being so many potential solutions to your problem is that doing a blind search through them all to find your favorite one will take about  $10^n$  time. For any reasonable  $n$ , this is a huge amount of time.

**Rules:** There are lots of rules about logs and exponentials that one might learn. Personally, I like to minimize it to the following.

$b^n = \overbrace{(b \times b \times b \times \dots \times b)}^n$ : This is the definition of exponentiation.  $b^n$  is  $n$   $b$ 's multiplied together.

$b^n \times b^m = b^{n+m}$ : This is simply by counting the number of  $b$ 's being multiplied.

$$\underbrace{(b \times b \times b \times \dots \times b)}_n \times \underbrace{(b \times b \times b \times \dots \times b)}_m = \underbrace{b \times b \times b \times \dots \times b}_{n+m}.$$

$b^0 = 1$ : One might guess that zero  $b$ 's multiplied together is zero, but it needs to be one.

One argument for this is as follows.  $b^n = b^{0+n} = b^0 \times b^n$ . For this to be true,  $b^0$  must be one.

$b^{\frac{1}{2}} = \sqrt{n}$ : By definition,  $\sqrt{n}$  is the positive number which when multiplied by itself gives  $b$ .  $b^{\frac{1}{2}}$  meets this definition because  $b^{\frac{1}{2}} \times b^{\frac{1}{2}} = b^{\frac{1}{2} + \frac{1}{2}} = b^1 = b$ .

$b^{-n} = \frac{1}{b^n}$ : The fact that this needs to be true can be argued in a similar way.  $1 = b^{n+(-n)} = b^n \times b^{-n}$ . For this to be true,  $b^{-n}$  must be  $\frac{1}{b^n}$ .

$(b^n)^m = b^{n \times m}$ : Again we count the number of  $b$ 's.

$$\underbrace{(b \times b \times b \times \dots \times b)}_n \times \underbrace{(b \times b \times b \times \dots \times b)}_n \times \dots \times \underbrace{(b \times b \times b \times \dots \times b)}_n = \underbrace{b \times b \times b \times \dots \times b}_{n \times m}.$$

If  $x = \log_b(n)$  then  $n = b^x$ : This is the definition of logarithms.

$\log_b(1) = 0$ : This follows from  $b^0 = 1$ .

$\log_b(b^x) = x$  and  $b^{\log_b(n)} = n$ : Substituting  $n = b^x$  into  $x = \log_b(n)$  gives the first and substituting  $x = \log_b(n)$  into  $n = b^x$  gives the second.

$\log_b(n \times m) = \log_b(n) + \log_b(m)$ : The number of digits to write down the product of two integers is the number to write down each of them separately (modulo rounding errors).

We prove it by applying the definition of logarithms and the above rules.  $b^{\log_b(n \times m)} = n \times m = b^{\log_b(n)} \times b^{\log_b(m)} = b^{\log_b(n) + \log_b(m)}$ . It follows that  $\log_b(n \times m) = \log_b(n) + \log_b(m)$ .

$\log_b(n^d) = d \times \log_b(n)$ : This is an extension of the above rule.

$\log_b(n) - \log_b(m) = \log_b(n) + \log_b(\frac{1}{m}) = \log_b(\frac{n}{m})$ : This is another extension of the above rule.

$d^{c \log_2(n)} = n^{c \log_2(d)}$ : This rule states that you can move things between the base to the exponent as long as you add or remove a log. The proof is as follows.  $d^{c \log_2(n)} = (2^{\log_2(d)})^{c \log_2(n)} = 2^{\log_2(d) \times c \log_2(n)} = 2^{\log_2(n) \times c \log_2(d)} = (2^{\log_2(n)})^{c \log_2(d)} = n^{c \log_2(d)}$ .

$\log_2(n) = 3.32.. \times \log_{10}(n)$ : The number of bits needed to express the integer  $n$  is 3.32.. times the number of decimal digits needed. This can be seen as follows. Suppose  $x = \log_2 n$ . Then  $n = 2^x$ , giving  $\log_{10} n = \log_{10}(2^x) = x \cdot \log_{10} 2$ . Finally,  $x = \frac{1}{\log_{10} 2} \log_{10}(n) = 3.32.. \log_{10} n$ .

**Which Base:** We will write  $\Theta(\log(n))$  without giving an explicit base. A high school student might use base 10 as the default, a scientist base  $e = 2.718..$ , and computer scientists base 2. My philosophy is that I exclude the base when it does not matter. As seen above,  $\log_{10}(n)$ ,  $\log_2(n)$ , and  $\log_e(n)$ , differ only by a multiplicative constant. In general, we will be ignoring multiplicative constants, and hence which base used is irrelevant. I will only include the base when the base matters. For example,  $2^n$  and  $10^n$  differ by much more than a multiplicative constant.

**The Ratio  $\frac{\log a}{\log b}$ :** When computing the ratio between two logarithms, the base used does not matter because changing the base will introduce the same constant both on the top and the bottom, which will cancel. Hence, when computing such a ratio, you can choose whichever

base makes the calculation the easiest. For example, to compute  $\frac{\log 16}{\log 8}$ , the obvious base to use is 2, because  $\frac{\log_2 16}{\log_2 8} = \frac{4}{3}$ . On the other hand, to compute  $\frac{\log 9}{\log 27}$ , the obvious base to use is 3, because  $\frac{\log_3 9}{\log_3 27} = \frac{2}{3}$ .

## Chapter 3

# Asymptotic Notations and their Properties

BigOh ( $\mathcal{O}$ ) and Theta ( $\Theta$ ) are notations for classifying functions.

### Purpose:

**Time Complexity:** Generally for this course, the functions that we will be classifying will be the time complexities of programs. On the other hand, these ideas can also be used to classify any function.

**Function Growth:** The purpose of classifying a function is to give an idea of how fast grows without going into too much detail.

**Example:** While on a walk in the woods, someone asks me, “What is that?” I could answer:

“It is an adult male red-bellied sapsucker of the American northeast variety.” or I could say:

“It’s a bird.”

Which is better? It depends on how much detail is needed.

**Classifying:** We classify the set of all vertebrate animals into mammal, bird, reptile, and so on; we further classify mammals into humans, cats, dogs, and so on; finally, among others, the class of humans will contain me. Similarly, we will classify functions into constants  $\Theta(1)$ , poly-logarithms  $\log^{\Theta(1)}(n)$ , polynomial  $n^{\Theta(1)}$ , exponentials  $2^{\Theta(n)}$ , double exponentials  $2^{2^{\Theta(n)}}$ , and so on; we further classify the polynomials  $n^{\Theta(1)}$  into linear functions  $\Theta(n)$ , the time for sorting  $\Theta(n \log n)$ , quadratics  $\Theta(n^2)$ , cubics  $\Theta(n^3)$ , and so on; finally, among others, the class of linear functions  $\Theta(n)$  will contain  $f(n) = 2n + 3\log^2 n + 8$ .



**Asymptotic Growth Rate:** When classifying animals, Darwin decided to not consider whether the animal sleeps during the day, but whether it has hair. When classifying functions, complexity theorists decided to not consider its behavior on small values of  $n$  or even whether it is monotone increasing, but how quickly it grows when its input  $n$  grows really big. This is

referred to as the *asymptotics* of the function. Here are some examples of different growth rates.

Function	Approximate value of $T(n)$ for $n =$			
$T(n)$	10	100	1,000	10,000
$\log n$	3	6	9	13
$\sqrt{n}$	3	10	31	100
$n$	10	100	1,000	10,000
$n \log n$	30	600	9,000	130,000
$n^2$	100	10,000	$10^6$	$10^8$
$n^3$	1,000	$10^6$	$10^9$	$10^{12}$
$2^n$	1,024	$10^{30}$	$10^{300}$	$10^{3000}$
				the universe

Note: The universe contains approximately  $10^{80}$  particles.

### 3.1 The Classic Classifications of Functions

The following are the main classifications of functions that will arise in this course as the running times of an algorithm or as the amount of space that it uses.

**Functions with A Basic Form:** Most functions considered will be the sum of a number of terms, where each term has the basic form  $f(n) = c \cdot b^{an} \cdot n^d \cdot \log^e n$ , where  $a, b, c, d$ , and  $e$  are real constants. Only the largest such term is considered significant. In order from smallest to largest, these functions (terms) are classified as follows.

<b>Constants:</b>	$\Theta(1)$	$f(n) = c,$	for each $c > 0$ .
<b>Logarithms:</b>	$\Theta(\log(n))$	$f(n) = c \cdot \log n,$	
<b>Poly-Logarithms:</b>	$\log^{\Theta(1)}(n)$	$f(n) = c \cdot \log^e n,$	$e > 0.$
<b>Linear Functions:</b>	$\Theta(n)$	$f(n) = c \cdot n,$	
<b>Time for Sorting:</b>	$\Theta(n \log n)$	$f(n) = c \cdot n \log n,$	
<b>Quadratics:</b>	$\Theta(n^2)$	$f(n) = c \cdot n^2,$	
<b>Cubics:</b>	$\Theta(n^3)$	$f(n) = c \cdot n^3,$	
<b>Polynomial:</b>	$n^{\Theta(1)}$	$f(n) = c \cdot n^d \cdot \log^e n,$	$d > 0, e \in (-\infty, \infty).$
<b>Exponentials:</b>	$2^{\Theta(n)}$	$f(n) = c \cdot b^{an} \cdot n^d \cdot \log^e n,$	$a > 0, b > 1, d, e \in (-\infty, \infty).$
<b>Double Exp.:</b>	$2^{2^{\Theta(n)}}$		

These classes will now get discussed in more detail.

**Constants  $\Theta(1)$ :** A constant function is one whose output does not depend on its input. For example,  $f(n) = 7$ . One algorithm for which this function arises is popping an element off a stack that is stored as a linked list. This takes maybe 7 “operations” independent of the number  $n$  of elements in the stack. Someone else’s implementation may require 8 “operations.”  $\Theta(1)$  is a class of functions that include all such functions. We say  $\Theta(1)$  instead of 7 when we do not care what the actually constant is. Determining whether it is 7, 9, or 8.829 may be more work than it is really worth. A function like  $f(n) = 8 + \sin(n)$ , on the other hand, continually changes between 7 and 9 and  $f(n) = 8 + \frac{1}{n}$  continually changes approaching 8. However, if we are not caring whether it is 7, 9, or 8.829, why should we care if it is changing between them? Hence, both of these functions are included in  $\Theta(1)$ . On the other hand, in most applications being negative  $f(n) = -1$  or zero  $f(n) = 0$  would be quite a different

matter. Hence, these are excluded. Similarly, the function  $f(n) = \frac{1}{n}$  is not included, because the only constant that it is bounded below by is zero and the zero function is not included.

#### Examples Included:

- $f(n) = 7$  and  $f(n) = 8.829$
- $f(n) = 8 + \sin(n)$
- $f(n) = 8 + \frac{1}{n}$
- $f(n) = \{ 1 \text{ if } n \text{ is odd, } 2 \text{ if } n \text{ is even } \}$

#### Examples Not Included:

- $f(n) = -1$  and  $f(n) = 0$  (fails  $c > 0$ )
- $f(n) = \sin(n)$  (fails  $c > 0$ )
- $f(n) = \frac{1}{n}$  (too small)
- $f(n) = \log_2 n$  (too big)

**Logarithms  $\Theta(\log(n))$ :** See Section 3 for how logarithmic functions like  $\log_2(n)$  arise in this course and for some of their rules. The class of functions  $\Theta(\log(n))$  consists of those functions  $f(n)$  that are “some constant” times this initial function  $\log_2(n)$ . For example,  $f(n) = 2\log_2(n)$  is included.

**Slowly Growing:** The functions  $f(n) = \log_2(n)$  and  $f(n) = \log_{10}(n)$  grow as  $n$  gets large, but very slowly. For example, even if  $n$  is  $10^{80}$ , which is roughly the number of particles in the universe, still  $f(n) = \log_{10}(n)$  is only 80. Because it grows with  $n$ , this function certainly is not included in  $\Theta(1)$ . However, it is much much smaller than the linear function  $f(n) = n$ .

**Which Base:** We write  $\Theta(\log(n))$  without giving an explicit base. As shown in the list of rules about logarithms,  $\log_{10}(n)$ ,  $\log_2(n)$ , and  $\log_e(n)$ , differ only by a multiplicative constant. Because we are ignoring multiplicative constants anyway, which base is used is irrelevant. The rules also indicate that  $8 \cdot \log_2(n^5)$  also differs only by a multiplicative constant. All of these functions are include in  $\Theta(\log(n))$ .

**Some Constant:** See  $\Theta(1)$  above for what we mean by the “some constant” that can be multiplied by the  $\log n$ .

#### Examples Included:

- $f(n) = 7\log_2 n$  and  $f(n) = 8.829\log_{10}(n)$
- $f(n) = \log(n^8)$  and  $f(n) = \log(n^8 + 2n^3 - 10)$
- $f(n) = (8 + \sin(n))\log_2(n)$ ,  $f(n) = 8\log_2 n + \frac{1}{n}$ , and  $f(n) = 3\log_2 n - 1,000,000$

#### Examples Not Included:

- $f(n) = -\log_2 n$  (fails  $c > 0$ )
- $f(n) = 5$  (too small)
- $f(n) = (\log_2 n)^2$  (too big)

**Poly-Logarithms  $\log^{\Theta(1)}(n)$ :** Powers of logs like  $(\log n)^3$  are referred to as *poly-log*. These are often written as  $\log^3 n = (\log n)^3$ . Note that this is different than  $\log(n^3) = 3\log n$ . The class is denoted  $\log^{\Theta(1)}(n)$ . It is a super set of  $\Theta(\log(n))$ , because every function in  $\Theta(\log(n))$  is also in  $\log^{\Theta(1)}(n)$ , but  $f(n) = \log^3(n)$  is included in the latter but not the first.

**Examples Included:**

- $f(n) = 7(\log_2 n)^5$ ,  $f(n) = 7 \log_2 n$ , and  $f(n) = 7\sqrt{\log_2 n}$
- $f(n) = 7(\log_2 n)^5 + 6(\log_2 n)^3 - 19 + 7(\log_2 n)^2/n$

**Examples Not Included:**

- $f(n) = n$  (too big)

**Linear Functions  $\Theta(n)$ :** Given an input of  $n$  items, it takes  $\Theta(n)$  time simply to look at the input. Looping over the items and doing constant amount of work for each takes another  $\Theta(n)$  time. Say,  $t_1(n) = 2n$  and  $t_2(n) = 4n$  for a total of  $6n$  time. Now if you do not want to do more than linear time, are you allowed to do any more work? Sure. You can do something that takes  $t_3(n) = 3n$  time and something else that takes  $t_4(n) = 5n$  time. You are even allowed to do a few things that take a constant amount of time, totaling to say  $t_5(n) = 13$ . The entire algorithm then takes the sum of these  $t(n) = 14n + 13$  time. This is still considered to be linear time.

**Examples Included:**

- $f(n) = 7n$  and  $f(n) = 8.829n$
- $f(n) = (8 + \sin(n))n$  and  $f(n) = 8n + \log^{10} n + \frac{1}{n} - 1,000,000$

**Examples Not Included:**

- $f(n) = -n$  and  $f(n) = 0n$  (fails  $c > 0$ )
- $f(n) = \frac{n}{\log_2 n}$  (too small)
- $f(n) = n \log_2 n$  (too big)

**Time for Sorting  $\Theta(n \log n)$ :** Another running time that arises often in algorithms is  $\Theta(n \log n)$ .

For example, this is the number of comparisons needed to sort  $n$  elements. What should be noted is that the function  $f(n) = n \log n$  is just slightly too big to be in the linear class of functions  $\Theta(n)$ . This is because  $n \log n$  is  $\log n$  times  $n$  and  $\log n$  is not constant.

**Examples Included:**

- $f(n) = 7n \log_2(n)$  and  $f(n) = 8.829n \log_{10}(n) + 3n - 10$

**Examples Not Included:**

- $f(n) = n$  (too small)
- $f(n) = n \log^2 n$  (too big)

**Quadratic Functions  $\Theta(n^2)$ :** Two nested loops from 1 to  $n$  takes  $\Theta(n^2)$  time if each inner iteration takes a constant amount of time. A  $n \times n$  matrix requires  $\Theta(n^2)$  space if each element takes constant space.

**Examples Included:**

- $f(n) = 7n^2 - 8n \log n + 2n - 17$

**Examples Not Included:**

- $f(n) = \frac{n^2}{\log_2 n}$  and  $f(n) = n^{1.9}$  (too small)
- $f(n) = n^2 \log_2 n$  and  $f(n) = n^{2.1}$  (too big)

**Cubic Functions  $\Theta(n^3)$ :** Similarly, three nested loops from 1 to  $n$  and a  $n \times n \times n$  cube matrix requires  $\Theta(n^3)$  time and space.

**$\tilde{\Theta}(n)$ ,  $\tilde{\Theta}(n^2)$ ,  $\tilde{\Theta}(n^3)$ , and so on:** Stating that the running time of an algorithm is  $\Theta(n^2)$  allows us to ignore the multiplicative constant in front. Often, however, we will in fact ignore logarithmic factors. These factors might either arise because we say that the “size” of the input is  $n$  when in fact it consists of  $n$  elements each of  $\Theta(\log n)$  bits or because we say that the algorithm does  $\Theta(n^2)$  “operations” when in fact each operation consists of adding together  $\Theta(\log n)$  bit values. The only place that this book cares about these factors is when it is determining whether the running time of sorting algorithms is  $\Theta(n)$  or  $\Theta(n \log n)$ . When one does want to ignore these logarithmic factors, a notation sometimes used is  $\tilde{\Theta}(n^2)$ . It is a short form for  $\log^{\pm\Theta(1)}(n) \times \Theta(n^2)$ . Though sometimes maybe we should, we do not use this notation.

#### Examples Included in $\tilde{\Theta}(n^2)$ :

- $f(n) = 7n^2 \log^7 n + 2n - 17$
- $f(n) = 7\frac{n^2}{\log^7 n} + 2n - 17$

#### Examples Not Included in $\tilde{\Theta}(n^2)$ :

- $f(n) = n^{2.1}$  (too big)

**Polynomial  $n^{\Theta(1)}$ :** As the notation  $n^{\Theta(1)}$  implies, the class of polynomials consists of functions  $f(n) = n^c$  where  $c > 0$  is “some constant” from  $\Theta(1)$ . For example,  $\sqrt{n}$ ,  $n$ ,  $n^2$ ,  $n^3$ , but not  $n^0 = 1$  or  $n^{-1} = \frac{1}{n}$ .

By closing the class of functions under addition, this class includes functions like  $f(n) = 4n^3 + 7n^2 - 2n + 8$ . What about the function  $f(n) = n \log n$ ? If the context was the study of polynomials, I would not consider this to be one. However, what we care about here is growth rate. Though  $f(n) = n \log n$  belongs neither to the class  $\Theta(n)$  nor to the class  $\Theta(n^2)$ , it is bounded below by the function  $f_1(n) = n$  and above by  $f_2(n) = n^2$ . Both of these are included in the class  $n^{\Theta(1)}$ . Hence,  $f(n) = n \log n$  must also have the right kind of grow rate and hence should be included too. In contrast, the function  $f(n) = \log n$  is not included, because to make  $n^c \leq \log n$ ,  $c$  would have to be zero and  $n^0 = 1$  is not included.

**Definition of a Feasible Algorithm:** An algorithm is considered to be *feasible* if it runs in polynomial time.

#### Examples Included:

- $f(n) = 7n^2 - 8n \log n + 2n - 17$
- $f(n) = n^2 \log_2 n$  and  $f(n) = \frac{n^2}{\log_2 n}$
- $f(n) = \sqrt{n}$  and  $f(n) = n^{3.1}$
- $f(n) = 7n^3 \log^7 n - 8n^2 \log n + 2n - 17$

#### Examples Not Included:

- $f(n) = \log n$  (too small)
- $f(n) = n^{\log n}$  and  $f(n) = 2^n$  (too big)

**Exponentials  $2^{\Theta(n)}$ :** As the notation  $2^{\Theta(n)}$  implies, the class of exponentials consists of functions  $f(n) = 2^{cn}$  where  $c > 0$  is “some constant” from  $\Theta(1)$ . For example,  $2^{0.001n}$ ,  $2^n$ , and  $2^{2n}$ . Again,  $2^{0n} = 1$  and  $2^{-n} = \frac{1}{2^n}$  are not included.

**Definition of an Infeasible Algorithm:** An algorithm is considered to be *infeasible* if it runs in exponential time.

**The Base:** At first it looks like the function  $4^n$  should not be in  $2^{\Theta(n)}$ , because the base is 4 not 2. However,  $4^n = 2^{2n}$  and hence it is included. In general, the base of the exponential does not matter when determining whether it is exponential, as long as it is bigger than one. This is because for any  $b > 1$ , the function  $f(n) = b^n = 2^{(\log_2 b)n}$  is included in  $2^{\Theta(n)}$ , since  $(\log_2 b) > 0$  is a constant. (See the logarithmic rules above.) In contrast,  $4^n$  is far too big to be in  $\Theta(2^n)$  because  $\frac{4^n}{2^n} \gg \Theta(1)$ .

**Incorrect Notation  $(\Theta(1))^n$ :** One might think that we would denote this class of functions as  $(\Theta(1))^n$ . However, this would imply  $f(n) \approx c^n$  for some constant  $c > 0$ . This would include  $\left(\frac{1}{2}\right)^n$  which decreases with  $n$ .

#### Examples Included:

- $f(n) = 2^n$  and  $f(n) = 3^{5n}$
- $f(n) = 2^n \cdot n^2 \log_2 n - 7n^8$  and  $f(n) = \frac{2^n}{n^2}$

#### Examples Not Included:

- $f(n) = n^{1,000,000}$  (too small)
- $f(n) = n! \approx n^n = 2^{n \log_2 n}$  and  $f(n) = 2^{n^2}$  (too big)

**Double Exponentials  $2^{2^{\Theta(n)}}$ :** These functions grow very fast. The only algorithm that runs this slowly considered in this text is Ackermann's function. See Section 15.3.

## 3.2 Comparing the Classes of Functions

These classes, except when one is a subset of the other, have very different grow rates. Look at the chart above.

**Linear vs Quadratic vs Cubic:** Whether the running time is  $T(n) = 2n^2$  or  $T(n) = 4n^2$ , though a doubling in time, will likely not change whether or not the algorithm is practical or not. On the other hand,  $T_1(n) = n$  is a whole  $n$  times better than  $T_2(n) = n^2$ . Now, one might argue that an algorithm whose running time is  $T_1(n) = 1,000,000n$  is worse than one whose running time is  $T_2(n) = n^2$ . However, when  $n = 1,000,000$ , the first one becomes faster. In our analysis, we will consider large values of  $n$ . Hence, we will consider  $T_1(n) = 1,000,000n$  to be better than  $T_2(n) = n^2$ . Similarly, this is better than  $T_3(n) = n^3$ .

**Poly-Logarithms vs Polynomials:** For sufficiently large  $n$ , polynomials are always asymptotically bigger than any poly-logarithm. For example,  $(\log n)^{1,000,000}$  is very big for a poly-logarithmic function, but it is considered to be much smaller than  $n^{0.00001}$ , which in turn is very small for a polynomial. Similarly,  $n^4 \log^7 n$  is between  $n^4$  and  $n^{4+\epsilon}$  even for  $\epsilon = 0.00000001$ .

**Polynomials vs Exponentials:** For sufficiently large  $n$ , polynomials are always asymptotically smaller than any exponential. For example,  $n^{1,000,000}$  is very big for a polynomial function, but is considered to be much smaller than  $2^{0.00001n}$ , which is very small for an exponential function.

#### When BigOh and Theta Notation are Misleading:

- Sometimes constant factors matter. For example, if an algorithm requires 1,000,000,000- $n$  time, then saying that it is  $\Theta(n)$  is true but misleading. In practice, constants are generally small.
- One algorithm may only be better for unrealistically large input sizes. For example,  $(\log n)^{100}$  is considered to be smaller than  $n$ . However,  $n$  only gets bigger when  $n$  is approximately  $2^{1024}$ .  $(\log n)^{100} = (\log 2^{1024})^{100} = 1024^{100} = 2^{(10 \cdot 100)} < 2^{1024} = n$ .

### 3.3 Other Useful Notations

The following are different notations for comparing the asymptotic grow of a function  $f(n)$  to some “constant” times the function  $g(n)$ .

Greek Letter	Standard Notation	My Notation	Meaning
Theta	$f(n) = \Theta(g(n))$	$f(n) \in \Theta(g(n))$	$f(n) \approx c \cdot g(n)$
BigOh	$f(n) = \mathcal{O}(g(n))$	$f(n) \leq \mathcal{O}(g(n))$	$f(n) \leq c \cdot g(n)$
Omega	$f(n) = \Omega(g(n))$	$f(n) \geq \Omega(g(n))$	$f(n) \geq c \cdot g(n)$
Little Oh	$f(n) = o(g(n))$	$f(n) \ll o(g(n))$	$f(n) \ll g(n)$
Little Omega	$f(n) = \omega(g(n))$	$f(n) \gg \omega(g(n))$	$f(n) \gg g(n)$

**Examples:**

**Same:**  $7 \cdot n^3$  is within a constant of  $n^3$ . Hence, it is in  $\Theta(n^3)$ ,  $\mathcal{O}(n^3)$ , and  $\Omega(n^3)$ . However, because it is not much smaller than  $n^3$ , it is not in  $o(n^3)$  and because it is not much bigger, it not in  $\omega(n^3)$ .

**Smaller:**  $7 \cdot n^3$  is asymptotically much smaller than  $n^4$ . Hence, it is in  $\mathcal{O}(n^4)$  and in  $o(n^4)$ , but it is not in  $\Theta(n^4)$ ,  $\Omega(n^4)$ , or  $\omega(n^4)$ .

**Bigger:**  $7 \cdot n^3$  is asymptotically much bigger than  $n^2$ . Hence, it is in  $\Omega(n^2)$  and in  $\omega(n^2)$ , but it is not in  $\Theta(n^2)$ ,  $\mathcal{O}(n^2)$ , or in  $o(n^2)$ .

You should always give the smallest possible approximation. Even when someone asks for BigOh notation, what they really want is Theta notation.

**Notation Considerations:**

**“ $\in$ ” vs “ $=$ ”:** I consider  $\Theta(n)$  to be a class of functions. Given this one should use the set notation,  $f(n) \in \Theta(g(n))$ , to denote membership.

On the other hand, the meaning is that, ignoring constant multiplicative factors  $f(n)$  has the same asymptotic growth as  $g(n)$ . Given this, the notation  $7n = \Theta(n)$  makes sense. This notation is standard.

Even the statements  $3n^2 + 5n - 7 = n^{\Theta(1)}$  and  $2^{3n} = 2^{\Theta(n)}$  make better sense when you think of the symbol  $\Theta$  to mean “some constant.” However, be sure to remember that  $4^n \cdot n^2 = 2^{\Theta(n)}$  is also true.

**“ $=$ ” vs “ $\leq$ ”:**  $7n = \mathcal{O}(n^2)$  is also standard notation. This makes less sense to me. Because it means that  $7n$  is at most some constant times  $n^2$ , a better notation would be  $7n \leq \mathcal{O}(n^2)$ . The standard notation is even more awkward, because  $\mathcal{O}(n) = \mathcal{O}(n^2)$  should be true, but  $\mathcal{O}(n^2) = \mathcal{O}(n)$  should be false. What sense does this make?

## 3.4 Different Levels of Detail When Classifying a Function

One can decide how much information about a function they want to reveal. We have already discussed how we could reveal only that the function  $f(n) = 7n^2 + 5n$  is a polynomial by stating that it is in  $n^{\Theta(1)}$  or reveal more that it is a quadratic by stating it is in  $\Theta(n^2)$ . However, there are even more options. We could give more detail by saying that  $f(n) = (7 + o(1))n^2$ . This is a way of writing  $7n^2 + o(n^2)$ . It reveals that the constant in front of the high-order term is 7. We are told that there may be low order terms, but not what they are. Even more detail would be given by saying that  $f(n) = 7n^2 + \Theta(n)$ .

As another example, let  $f(n) = 7 \cdot 2^{3n^5} + 8n^4 \log^7 n$ . The following classifications of functions are sorted from the least inclusive to the most inclusive, meaning that every function included in the first is also included in the second, but not vice versa. They all contain  $f$ , giving more and more details about it.

$$\begin{aligned} f(n) &= 7 \cdot 2^{3n^5} + 8n^4 \log^7 n \\ &\in 7 \cdot 2^{3n^5} + \Theta(n^4 \log^7 n) \\ &\subseteq 7 \cdot 2^{3n^5} + \tilde{\Theta}(n^4) \\ &\subseteq 7 \cdot 2^{3n^5} + n^{\Theta(1)} \\ &\subseteq (7 + o(1))2^{3n^5} \\ &\subseteq \Theta(2^{3n^5}) \\ &\subseteq 2^{\Theta(n^5)} \\ &\subseteq 2^{n^{\Theta(1)}}. \end{aligned}$$

**Exercise 3.4.1** (*See solution in Section V*) For each of these classes of functions, give a function that is in it but not in the next smaller class.

## 3.5 Constructing and Deconstructing the Classes of Functions

We will understand better which functions are in each of these classes, by carefully considering how both the classes and the functions within them are constructed and conversely, deconstructed.

**The Class of Functions  $\Theta(g(n))$ :** Above we discussed the classes of functions  $\Theta(1)$ ,  $\Theta(\log n)$ ,  $\Theta(n)$ ,  $\Theta(n \log n)$ ,  $\Theta(n^2)$ , and  $\Theta(n^3)$ . Of course, we could similarly define  $\Theta(n^4)$ . How about  $\Theta(2^n \cdot n^2 \cdot \log^3 n)$ ? Yes, this is a class of functions too. For every function  $g(n)$ ,  $\Theta(g(n))$  is the class of functions that are similar within a constant factor in asymptotic growth rate to  $g(n)$ . In fact, one way to think of the class  $\Theta(g(n))$  is  $\Theta(1) \cdot g(n)$ . Note that the classes  $\log^{\Theta(1)}(n)$ ,  $n^{\Theta(1)}$ , and  $2^{\Theta(n)}$  do not have this form. We will cover them next.

**Constructing The Functions in The Class:** A class of functions like  $\Theta(n^2)$  is constructed by starting with some function  $g(n) = n^2$  and from it constructing other functions that are also in the class by repeatedly doing one of the following:

**Closed Under Scalar Multiplication:** You can multiply any function  $f(n)$  in the class by a strictly positive constant  $c > 0$ . This gives us that  $7n^2$  and  $9n^2$  are in  $\Theta(n^2)$ .

**Closed Under Addition:** The class of functions  $\Theta(g(n))$  is closed under addition.

What this means is that if  $f_1(n)$  and  $f_2(n)$  are both functions in the class, then their sum  $f_1(n) + f_2(n)$  is also in it. Moreover, if  $f_2(n)$  grows too slowly to be in the class then  $f_1(n) + f_2(n)$  is still in it and so is  $f_1(n) - f_2(n)$ . For example,  $7n^2$  being in  $\Theta(n^2)$  and  $3n + 2$  growing smaller gives that  $7n^2 + 3n + 2$  and  $7n^2 - 3n - 2$  are in  $\Theta(n^2)$ . Be aware that when subtracting that the dominating term may cancel. For example, both  $3n^2 + 4n$  and  $3n^2 + 2n$  are in  $\Theta(n^2)$ , but  $(3n^2 + 4n) - (3n^2 + 2n) - 2n$  is not.

**Bounded Between:** You may also throw in any function that is bounded below and above by functions that you have constructed before.  $7n^2$  and  $9n^2$  being in  $\Theta(n^2)$  gives that  $(8 + \sin(n))n^2$  is in  $\Theta(n^2)$ . Note that  $7n^2 + 100n$  is also in  $\Theta(n^2)$  because it is bounded below by  $7n^2$  and above by  $8n^2 = 7n^2 + 1n^2$ , because the  $1n^2$  eventually dominates the  $100n$ .

**Determining Which Class  $\Theta(g(n))$  of Functions  $f(n)$  is in:** Given a function  $f(n)$ , one task that you will often be asked to do is determine which “ $\Theta$ ” class it is in. This amounts to finding a function  $g(n)$  that is as simple as possible and for which  $f(n)$  is contained in  $\Theta(g(n))$ . This can be done by deconstructing  $f(n)$  in the opposite way that we constructed functions to put into the class of functions  $\Theta(g(n))$ .

**Drop Low Order Terms:** If  $f(n)$  is a number of things added or subtracted together, then each of these things is called at *term*. Determine which of the terms grows the fastest. The slower growing terms are referred to as *low order terms*. Drop them.

**Drop Multiplicative Constant:** Drop the multiplicative constant in front of the largest term.

### Examples:

- Given  $3n^3 \log n - 1000n^2 + n - 29$ , the terms are  $3n^3 \log n$ ,  $1000n^2$ ,  $n$ , and  $29$ . Dropping the low order terms gives  $3n^3 \log n$ . Dropping the multiplicative constant 3, gives  $n^3 \log n$ . Hence,  $3n^3 \log n - 1000n^2 + n - 29$  is in the class  $\Theta(n^3 \log n)$ .
- Given  $7 \cdot 4^n \cdot n^2 / \log^3 n + 8 \cdot 2^n + 17 \cdot n^2 + 1000 \cdot n$ , the terms are  $7 \cdot 4^n \cdot n^2 / \log^3 n$ ,  $8 \cdot 2^n$ ,  $17 \cdot n^2$ , and  $1000 \cdot n$ . Dropping the low order terms gives  $7 \cdot 4^n \cdot n^2 / \log^3 n$ . Dropping the multiplicative constant 7, gives  $4^n \cdot n^2 / \log^3 n$ . Hence, this function  $7 \cdot 4^n \cdot n^2 / \log^3 n + 8 \cdot 2^n + 17 \cdot n^2 + 1000 \cdot n$  is in the class  $\Theta(4^n \cdot n^2 / \log^3 n)$ .
- $\frac{1}{n} + 18$  is in the class  $\Theta(1)$ . Since  $\frac{1}{n}$  is a lower-order term than 18, it is dropped.
- $\frac{1}{n^2} + \frac{1}{n}$  is in the class  $\Theta(\frac{1}{n})$  because  $\frac{1}{n^2}$  is a smaller term.

**The Class of Functions  $(g(n))^{\Theta(1)}$ :** Above we discussed the classes of functions  $\log^{\Theta(1)}(n)$ ,  $n^{\Theta(1)}$ , and  $2^{\Theta(n)}$ . Instead of forming the class  $\Theta(g(n))$  by multiplying some function  $g(n)$  by a constant from  $\Theta(1)$ , the form of these classes is  $(g(n))^{\Theta(1)}$  and hence are formed by raising some function  $g(n)$  to the power of some constant from  $\Theta(1)$ .

**Constructing The Functions in The Class:** The poly-logarithms  $\log^{\Theta(1)}(n)$ , polynomial  $n^{\Theta(1)}$ , and exponential  $2^{\Theta(n)}$  are constructed by starting respectively with the functions  $g(n) = \log n$ ,  $g(n) = n$ , and  $g(n) = 2^n$  and from them constructing other functions by repeatedly doing one of the following:

**Closed Under Scalar Multiplication and Additions:** As with  $\Theta(g(n))$ , if  $f_1(n)$  is in the class and  $f_2(n)$  is either in it or grows too slowly to be in it, then  $cf_1(n)$ ,

$f_1(n) + f_2(n)$ , and  $f_1(n) - f_2(n)$  are also in it. For example,  $n^2$  and  $n$  being in  $n^{\Theta(1)}$  and  $\log n$  growing smaller gives that  $7n^2 - 3n + 8\log n$  is in it as well.

**Closed Under Multiplication:** Unlike  $\Theta(g(n))$ , these classes are closed under multiplication as well. Hence,  $f_1(n) \cdot f_2(n)$  is in it as well, and if  $f_2(n)$  is sufficiently smaller that everything does not cancel, then so is  $\frac{f_1(n)}{f_2(n)}$ . For example,  $n^2$  and  $n$  being in  $n^{\Theta(1)}$  and  $\log n$  growing smaller gives that  $n^2 \cdot n = n^3$ ,  $\frac{n^2}{n} = n$ ,  $n^2 \cdot \log n$ , and  $\frac{n^2}{\log n}$  are in it as well.

**Closed Under Raising to Constant  $c > 0$  Powers:** Functions in these classes can be raised to any constant  $c > 0$ . This is why  $\log^c(n)$  is in  $\log^{\Theta(1)}(n)$ ,  $n^c$  is in  $n^{\Theta(1)}$ , and  $(2^n)^c = 2^{cn}$  is in  $2^{\Theta(n)}$ .

**Different Constant  $b > 1$  Bases:** As shown in Section 2,  $b^n = 2^{\log_2(b)n}$ . Hence, as long as  $b > 1$ , it does not matter what the base is.

**Bounded Between:** As with  $\Theta(g(n))$ , you can also throw in any function that is bounded below and above by functions that you have constructed before. For example,  $n^2 \cdot \log n$  is in  $n^{\Theta(1)}$  because it is bounded between  $n^2$  and  $n^3$ .

**Determining Whether  $f(n)$  is in the Class:** You can determine whether  $f(n)$  is in  $\log^{\Theta(1)}(n)$ ,  $n^{\Theta(1)}$ , or  $2^{\Theta(n)}$  by deconstructing it.

**Drop Low Order Terms:** As with  $\Theta(g(n))$ , the first step is to drop the low order terms.

**Drop Low Order Multiplicative Factors:** Now instead of just multiplicative constants in front of the largest term, we are able to drop low order multiplicative factors.

**Drop Constants  $c > 0$  in the Exponent:** Drop the 3 in  $\log^3 n$ ,  $n^3$ , and  $2^{3n}$ , but not the -1 in  $2^{-n}$ .

**Change Constants  $b > 1$  in the Base to 2:** Change  $3^n$  to  $2^n$ , but not  $(\frac{1}{2})^n$ .

### Examples:

- Given  $3n^3 \log n - 1000n^2 + n - 29$ , we drop the low order terms to get  $3n^3 \log n$ . The factors of this term are 3,  $n^3$ , and  $\log n$ . Dropping the low order ones gives  $n^3$ . This is  $n$  to the power of a constant. Hence,  $3n^3 \log n - 1000n^2 + n - 29$  is in the class  $n^{\Theta(1)}$ .
- Given  $7 \cdot 4^n \cdot n^2 / \log^3 n + 8 \cdot 2^n + 17 \cdot n^2 + 1000 \cdot n$ , we drop the low order terms to get  $7 \cdot 4^n \cdot n^2 / \log^3 n$ . The factors of this term are 7,  $4^n$ ,  $n^2$ , and  $\log^3 n$ . Dropping the low order ones gives  $4^n$ . Changing the base gives  $2^n$ . Hence,  $7 \cdot 4^n \cdot n^2 / \log^3 n + 8 \cdot 2^n + 17 \cdot n^2 + 1000 \cdot n$  is in the class  $2^{\Theta(n)}$ .

## 3.6 The Formal Definition of $\Theta$ and $\mathcal{O}$ Notation

We have given a lot of intuition about the above classes. We will now give the formal definitions of these classes.

**The Formal Definition of  $\Theta(g(n))$ :** Informally,  $\Theta(g(n))$  is the class of functions that are some “constant” times the function  $g(n)$ . This is complicated by the fact that the “constant” can be anything  $c(n)$  in  $\Theta(1)$ , which includes things like  $7 + \sin(n)$  and  $7 + \frac{1}{n}$ . The formal

definition, which will be explained in parts below, for the function  $f(n)$  to be included in the class  $\Theta(g(n))$  is

$$\exists c_1, c_2 > 0, \exists n_0, \forall n \geq n_0, c_1g(n) \leq f(n) \leq c_2g(n)$$

**Concrete Examples:** To make this as concrete as possible for you, substitute in the constant 1 for  $g(n)$  and in doing so define  $\Theta(1)$ . The name *constant* for this class of functions is a bit of a misnomer. *Bounded between strictly positive constants* would be a better name. Once you understand this definition within this context, try another one. For example, set  $g(n) = n^2$  in order to define  $\Theta(n^2)$ .

**Bounded Below and Above by a Constant Times  $g(n)$ :** We prove that the asymptotic growth of the function  $f(n)$  is comparable to a constant  $c$  times the function  $g(n)$ , by proving that it is *bounded below* by  $c_1$  times  $g(n)$  for a sufficiently small constant  $c_1$  and *bounded above* by  $c_2$  times  $g(n)$  for a sufficiently large constant  $c_2$ . See Figure 3.1. This is expressed in the definition by

$$c_1g(n) \leq f(n) \leq c_2g(n)$$

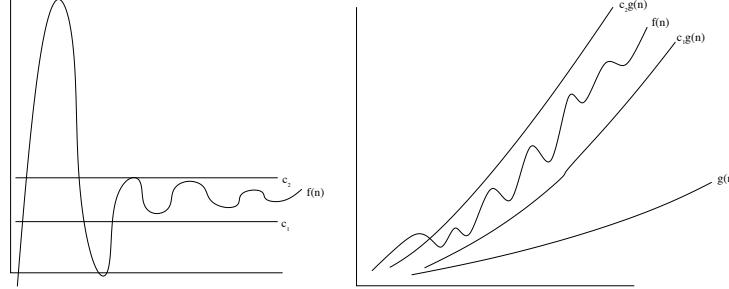


Figure 3.1: The left figure shows a function  $f(n) \in \Theta(1)$ . For sufficiently large  $n$ , it is bounded above and below by some constant. The right figure shows a function  $f(n) \in \Theta(g(n))$ . For sufficiently large  $n$ , it is bounded above and below by some constant times  $g(n)$ .

### Requirements on $c_1$ and $c_2$ :

**Strictly Positive:** Because we want to include neither  $f(n) = -1 \cdot g(n)$  nor  $f(n) = 0 \cdot g(n)$ , we require that  $c_1 > 0$ . Similarly, the function  $f(n) = \frac{1}{n} \cdot g(n)$  is not included, because this is bounded below by  $c_1 \cdot g(n)$  only when  $c_1 = 0$ .

**Allowing Big and Small Constants:** We might not care whether the multiplicative constant is 2, 4, or even 100. But what if it is 1,000, or 1,000,000, or even  $10^{10^{10}}$ ? Similarly, what if it is 0.01, or 0.00000001, or even  $10^{-100}$ ? Surely, this will make a difference? Despite the fact that these concerns may lead to miss leading statements at times, all of these are included.

**An Unbounded Class of Bounded Functions:** One might argue that allowing  $\Theta(1)$  to include arbitrarily large values contradicts the fact that the functions in it are bounded. However, this is a mistake. Though it is true that the class of functions in  $\Theta(1)$  is not bounded, each individual function included is bounded. For example, though  $f(n) = 10^{10^{10}}$  big, in itself, it is bounded.

**Reasons:** There are a few reasons for including all of these big constants. First, if you were to set a limit, what limit would you set? Every application has its

own definitions of “reasonable” and even these are open for debate. In contrast, including all strictly positive constants  $c$  is a much cleaner mathematical statement. A final reason for including all of these big constants is because they do not, in fact, arrive often and hence are not a big concern. In practice, one writes  $\Theta(g(n))$  to mean  $g(n)$  times a “reasonable” constant. If it is an unreasonable constant, one will still write  $\Theta(g(n))$ , but will include a foot note to the fact.

**Existential Quantifier  $\exists c_1, c_2 > 0$ :** No requirements are made on the constants  $c_1$  and  $c_2$  other than them being strictly positive. It is sufficient that constants that do the job exist. Formally, we write

$$\exists c_1, c_2 > 0, c_1 g(n) \leq f(n) \leq c_2 g(n)$$

In practice, you could assume that  $c_1 = 0.00000001$  and  $c_2 = 1,000,000$ , however, if asked for the bounding constants for a given function, it is better to give constants that bound as tightly as possible.

**Which Input Values  $n$ :**

**Universal Quantifier  $\forall n$ :** We have not spoken yet about for which input  $n$ , the function  $f(n)$  must be bounded. Ideally for all inputs. Formally, this would be

$$\exists c_1, c_2 > 0, \forall n, c_1 \leq f(n) \leq c_2$$

**Order of Quantifiers Matters:** The following is wrong.

$$\forall n, \exists c_1, c_2 > 0, c_1 \leq f(n) \leq c_2$$

The first requires the existence of one constant  $c_1$  and one constant  $c_2$  that bound the function for all inputs  $n$ . The second allows there to be a different constant for each  $n$ . This is much easier to do. For example, the following is true.  $\forall n, \exists c_1, c_2 > 0, c_1 \leq n \leq c_2$ . Namely, given an  $n$ , simply set  $c_1 = c_2 = n$ .

**Sufficiently Large  $n$ :** Requiring the function to be bounded for all input  $n$ , excludes the function  $f(n) = \frac{1}{n-3} + 1$  from  $\Theta(1)$ , because it goes to infinity for the one moment that  $n = 3$ . It also excludes  $f(n) = 2n^2 - 6n$  from  $\Theta(n^2)$ , because it is negative until  $n$  is 3. Now one may fairly argue that such functions should not be included, just as one may argue whether a platypus is a mammal or a tomato is a fruit. However, it turns out to be useful to include them. Recall, when classifying functions, we are not considering its behavior on small values of  $n$  or even whether it is monotone increasing, but on how quickly it grows when its input  $n$  grows really big. All three of the examples above are bounded for all sufficiently large  $n$ . This is why they are included. Formally, we write

$$\exists c_1, c_2 > 0, \forall n \geq n_0, c_1 \leq f(n) \leq c_2$$

where  $n_0$  is our definition of “sufficiently large” input  $n$ .

**For Some Definition of Sufficiently Large  $\exists n_0$ :** Like with the constants  $c_1$  and  $c_2$ , different applications have different definitions of how large  $n$  needs to be to be “sufficiently large.” Hence, to make the mathematics clean, we will simply require that there exists some definition  $n_0$  of sufficiently large that works.

**The Formal Definition of  $\Theta(g(n))$ :** This completes the discussion of the formal requirement for the function  $f(n)$  to be included in  $\Theta(g(n))$ .

$$\exists c_1, c_2 > 0, \exists n_0, \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$$

**The Formal Definitions of BigOh and Omega:** The definition of  $f(n) = \mathcal{O}(g(n))$  includes only the upper bound part of the Theta definition.

$$\exists c > 0, \exists n_0, \forall n \geq n_0, 0 \leq f(n) \leq c \cdot g(n)$$

Similarly, the definition of  $f(n) = \Omega(g(n))$  includes only the lower bound part.

$$\exists c > 0, \exists n_0, \forall n \geq n_0, f(n) \geq c \cdot g(n)$$

Note that  $f(n) = \Theta(g(n))$  is true if both  $f(n) = \mathcal{O}(g(n))$  and  $f(n) = \Omega(g(n))$  are true.

**The Formal Definition of Polynomial  $n^{\Theta(1)}$  and of Exponential  $2^{\Theta(n)}$ :** The function  $f(n)$  is included in the class of polynomials  $n^{\Theta(1)}$  if

$$\exists c_1, c_2 > 0, \exists n_0, \forall n \geq n_0, n^{c_1} \leq f(n) \leq n^{c_2}$$

and in the class of exponentials  $2^{\Theta(n)}$  if

$$\exists c_1, c_2 > 0, \exists n_0, \forall n \geq n_0, 2^{c_1 n} \leq f(n) \leq 2^{c_2 n}$$

**Bounded Below and Above:** The function  $f(n) = 2^n \cdot n^2$  is in  $2^{\Theta(n)}$  because it is bounded below by  $2^n$  and above by  $2^{2n} = 2^n \cdot 2^n$ , because the  $2^n$  eventually dominates the factor of  $n^2$ .

**The Formal Definitions of Little Oh and Little Omega:** Another useful way to compare  $f(n)$  and  $g(n)$  is to look at the ratio between them for extremely large values of  $n$ .

Class	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} =$	A practically equivalent definition
$f(n) = \Theta(g(n))$	Some constant	$f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$
$f(n) = o(g(n))$	Zero	$f(n) = \mathcal{O}(g(n))$ , but $f(n) \neq \Omega(g(n))$
$f(n) = \omega(g(n))$	$\infty$	$f(n) \neq \mathcal{O}(g(n))$ , but $f(n) = \Omega(g(n))$

- $2n^2 + 100n = \Theta(n^2)$  and  $\lim_{n \rightarrow \infty} \frac{2n^2 + 100n}{n^2} = 2$ .
- $2n + 100 = o(n^2)$  and  $\lim_{n \rightarrow \infty} \frac{2n + 100}{n^2} = 0$ .
- $2n^3 + 100n = \omega(n^2)$  and  $\lim_{n \rightarrow \infty} \frac{2n^3 + 100n}{n^2} = \infty$ .
- Another possibility not listed is that the limit  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$  does not exist, because the ratio oscillates between two bounded constants  $c_1$  and  $c_2$ . See Figure 3.1. In this case,  $f(n) = \Theta(g(n))$ . This occurs with Sine, but will not occur for functions that are expressed with  $n$ , real constants, plus, minus, times, divide, exponentiation, and logarithms. See Section 5.3.

## 3.7 Formal Proofs

Sometimes you are asked to formally proof that a function  $f(n)$  is in  $\Theta(g(n))$  or that it is not.

**Proving  $f(n) \in \Theta(g(n))$ :** Use the prover-adversary game for proving statements with existential and universal quantifiers to prove the statement  $\exists c_1, c_2 > 0, \exists n_0, \forall n \geq n_0, c_1g(n) \leq f(n) \leq c_2g(n)$ .

- You as the prover provide  $c_1, c_2$ , and  $n_0$ .
- Some adversary gives you an  $n$  that is at least your  $n_0$ .
- You then prove that  $c_1g(n) \leq f(n) \leq c_2g(n)$ .

**Example 1:** For example,  $2n^2 + 100n = \Theta(n^2)$ . Let  $c_1 = 2, c_2 = 3$  and  $n_0 = 100$ . Then, for all  $n \geq 100$ ,  $c_1g(n) = 2n^2 \leq 2n^2 + 100n = f(n)$  and  $f(n) = 2n^2 + 100n \leq 2n^2 + n \cdot n = 3n^2 = c_2g(n)$ . The values of  $c_1, c_2$ , and  $n_0$  are not unique. For example,  $n_0 = 1, c_2 = 102$ , and  $n_0 = 1$  also work because for all  $n \geq 1$ ,  $f(n) = 2n^2 + 100n \leq 2n^2 + 100n^2 = 102n^2 = c_2g(n)$ .

**Example 2:** Another example is  $2n^2 - 100n \in \Theta(n^2)$ . This is negative for small  $n$ , but is positive for sufficiently large  $n$ . Here we could set  $c_1 = 1, c_2 = 2$ , and  $n_0 = 100$ . Then, for all  $n \geq 100$ ,  $c_1g(n) = 1n^2 = 2n^2 - n \cdot n \leq 2n^2 - 100n = f(n)$  and  $f(n) \leq 2n^2 = c_2g(n)$ .

**Negation of  $f(n) \in \Theta(g(n))$  is  $f(n) \notin \Theta(g(n))$ :** Recall that to take the negation of a statement, you must replace the existential quantifiers with universal quantifiers and vice versa. Then you move the negation in to the right. The formal negation of

$$\exists c_1, c_2 > 0, \exists n_0, \forall n \geq n_0, c_1g(n) \leq f(n) \leq c_2g(n)$$

is

$$\forall c_1, c_2 > 0, \forall n_0, \exists n \geq n_0, [c_1g(n) > f(n) \text{ or } f(n) > c_2g(n)]$$

Note that  $f(n)$  can either be excluded because it is too small or because it is too big. To see why is it not  $\forall c \leq 0$  review the point “The Domain Does Not Change” within the “Existential and Universal Quantifiers” section.

**Proving  $f(n) \notin \Theta(g(n))$ :**

- Some adversary is trying to prove that  $f(n) \in \Theta(g(n))$ . You are trying to prove him wrong.
- Just as you did you when you were proving  $f(n) \in \Theta(g(n))$ , the first step for the adversary is to give you constants  $c_1, c_2$ , and  $n_0$ . To be fair to the adversary, you must let him choose any constants he likes.
- You as the prover must consider his  $c_1, c_2$ , and  $n_0$  and then come up with an  $n$ . There are two requirements on this  $n$ .
  - You must be able to bound  $f(n)$ . For this you make need to make  $n$  large and perhaps even be selective as to which  $n$  you take.
  - You must respect the adversary’s request that  $n \geq n_0$ .

One option is to choose an  $n$  that is the maximum between the value that meets the first requirement and the value meeting the second.

- Finally, you must either prove that  $f(n)$  is too small, namely that  $c_1g(n) > f(n)$ , or that it is too big, namely that  $f(n) > c_2g(n)$ .

**Example Too Big:** We can prove that  $f(n) = 14n^8 + 100n^6 \notin \Theta(n^7)$  as follows. Let  $c_1, c_2$ , and  $n_0$  be arbitrary values. Here, the issue is that  $f(n)$  is too big. Hence, we will ignore the constant  $c_1$  and let  $n = \max(c_2, n_0)$ . Then we have that  $f(n) = 14n^8 + 100n^6 > n \cdot n^7 \geq c_2 \cdot n^7$ .

Similarly, we can prove that  $2^{2n} \neq \Theta(2^n)$  as follows: Let  $c_1, c_2$ , and  $n_0$  be arbitrary values. Let  $n = \max(1 + \log_2 c_2, n_0)$ . Then we have that  $f(n) = 2^{2n} = 2^n \cdot 2^n > c_2 \cdot g(n)$ .

**Example Too Small:** We can prove that  $f(n) = 14n^8 + 100n^6 \notin \Theta(n^9)$  as follows. Let  $c_1, c_2$ , and  $n_0$  be arbitrary values. Here, the issue is that  $f(n)$  is too small. Hence, we will ignore the constant  $c_2$ . Note that the adversary's goal is to prove that  $c_1n^9$  is smaller than  $f(n)$  and hence will likely choose to make  $c_1$  something like 0.00000001. The smaller he makes his  $c_1$ , the larger we will have to make  $n$ . Let us make  $n = \max(\frac{15}{c_1}, 11, n_0)$ . Then we demonstrate that  $c_1g(n) = c_1n^9 = n \cdot c_1n^8 \geq \frac{15}{c_1} \cdot c_1n^8 = 14n^8 + n^8 = 14n^8 + n^2 \cdot n^6 \geq 14n^8 + (11)^2 \cdot n^6 > 14n^8 + 100n^6 = f(n)$ .

## 3.8 Solving Equations by Ignoring Details

One technique for solving equations is with the help of Theta notation. Using this technique, you can keep making better and better approximations of the solution until it is as close to the actual answer as you like.

For example, consider the equation  $x = 7y^3(\log_2 y)^{18}$ . It is in fact impossible to solve this equation for  $y$ . However, we will be able to approximate the solution. We have learned that the  $y^3$  is much more significant than the  $(\log_2 y)^{18}$ . Hence, our first approximation will be  $x = 7y^3(\log_2 y)^{18} \in y^{3+o(1)} = [y^3, y^{3+\epsilon}]$ . This approximation can be solved easily, namely,  $y = x^{\frac{1}{3+o(1)}}$ . This is a fairly good approximation, but maybe we can do better.

Above, we ignored the factor  $(\log_2 y)^{18}$  because it made the equation hard to solve. However, we can use what we learned above to approximate it in terms of  $x$ . Substituting in  $y = x^{\frac{1}{3+o(1)}}$  gives  $x = 7y^3(\log_2 y)^{18} = 7y^3(\log_2 x^{\frac{1}{3+o(1)}})^{18} = \frac{7}{(3+o(1))^{18}} y^3 (\log_2 x)^{18}$ . This approximation can be solved easily, namely,  $y = \frac{(3+o(1))^6}{7^{\frac{1}{3}}} \frac{x^{\frac{1}{3}}}{(\log x)^6} = \left( \frac{3^6}{7^{\frac{1}{3}}} + o(1) \right) \frac{x^{\frac{1}{3}}}{(\log x)^6} = \Theta \left( \frac{x^{\frac{1}{3}}}{(\log x)^6} \right)$ .

In the above calculations, we ignored the constant factors  $c$ . Sometimes, however, this constant CANNOT be ignored. For example, suppose that  $y = \Theta(\log x)$ . Which of the following is true:  $x = \Theta(2^y)$  or  $x = 2^{\Theta(y)}$ ? Effectively,  $y = \Theta(\log x)$  means that  $y = c \cdot \log x$  for some unknown constant  $c$ . This gives  $\log x = \frac{1}{c}y$  and  $x = 2^{\frac{1}{c}y} = 2^{\Theta(y)}$ . It also gives that  $x = (2^{\frac{1}{c}})^y \neq \Theta(2^y)$ , because we do not know what  $c$  is.

With practice, you will be able to determine quickly what matters and what does not matter when solving equations.

You can use similar techniques to solve something like  $100n^3 = 3^n$ . Setting  $n = 10.65199$  gives  $100n^3 = 3^n = 120,862.8$ . You can find this using binary search or using Newton's method. My quick and dirty method uses approximations. As before, the first step is to note that  $3^n$  is more significant than  $100n^3$ . Hence, the “ $n$ ” in the first is more significant than the one in the second. Let's denote the more significant “ $n$ ” with  $n_{i+1}$  and the less significant one with  $n_i$ . This gives  $100(n_i)^3 = 3^{n_{i+1}}$ . Solving for the most significant “ $n$ ” gives  $n_{i+1} = \frac{\log(100) + 3\log(n_i)}{\log 3}$ . Thus we have

better and better approximations of  $n$ . Plugging in  $n_1 = 1$  gives  $n_2 = 4.1918065$ ,  $n_3 = 8.1052849$ ,  $n_4 = 9.9058778$ ,  $n_5 = 10.453693$ ,  $n_6 = 10.600679$ ,  $n_7 = 10.638807$ ,  $n_8 = 10.648611$ ,  $n_9 = 10.651127$ ,  $n_{10} = 10.651772$ ,  $n_{11} = 10.651937$ ,  $n_{12} = 10.651979$ , and  $n_{13} = 10.65199$ .

## 3.9 Exercises

**Exercise 3.9.1** Formally prove or disprove the following.

1.  $f(n) = 14n^9 + 1000n^7 + 23n^2 \log n = \mathcal{O}(n^9)$
2.  $f(n) = 14n^8 - 100n^6 = \mathcal{O}(n^7)$
3.  $2^{n+1} = \mathcal{O}(2^n)$
4.  $2^{2n} = \mathcal{O}(2^n)$
5.  $7 \cdot 2^{3 \cdot n^5} \in 3^{8n^{\Theta(1)}}$

**Exercise 3.9.2** Regarding  $f(n) = n^{\Theta(1)}$ .

1. Informally, which functions are included in the classification  $f(n) = n^{\Theta(1)}$ ?
2. What is the formal definition of  $f(n) = n^{\Theta(1)}$ ?
3. Which of the following are  $f(n) = n^{\Theta(1)}$ ? If so, give suitable values of  $c_1$  and  $c_2$  for when  $n_0 = 1000000$ .
  - (a)  $f(n) = 5n^3 + 17n^2 + 4$
  - (b)  $f(n) = 5n^3 \log n$
  - (c)  $f(n) = 5n^{3 \log n}$
  - (d)  $f(n) = 5 \log n$
  - (e)  $7^{3 \log n}$

**Exercise 3.9.3** For each pair of classes of functions, how are they the same? How are they different? If possible give a function that is included in one of these but not included in the other. If possible do the reverse, giving a function that is included in the other of these but not included in the first.

1.  $\Theta(2^{2n})$  and  $\Theta(2^{3n})$ .
2.  $\Theta(2^n)$  and  $3^{\Theta(n)}$ .
3.  $2n^2 + \mathcal{O}(n)$  and  $(2 + o(1))n^2$ .

**Exercise 3.9.4** Suppose that  $y = \Theta(\log x)$ . Which of the following are true:  $x = \Theta(2^y)$  and  $x = 2^{\Theta(y)}$ ? Why?

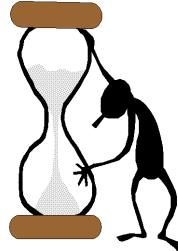
**Exercise 3.9.5** Let  $x$  be a real value. As you know,  $\lfloor x \rfloor$  rounds it down to the next integer. Explain what each of the following do:  $2 \cdot \lfloor \frac{x}{2} \rfloor$ ,  $\frac{1}{2} \cdot \lfloor 2 \cdot x \rfloor$ , and  $2^{\lfloor \log_2 x \rfloor}$ .

**Exercise 3.9.6** Let  $f(n)$  be a function. As you know,  $\Theta(f(n))$  drops low order terms and the leading coefficient. Explain what each of the following do:  $2^{\Theta(\log_2 f(n))}$ , and  $\log_2(\Theta(2^{f(n)}))$ . For each, explain to what extent the function is approximated.

## Chapter 4

# The Time (and Space) Complexity of an Algorithm

It is important to classify algorithms based on their time and space complexities.



### Purpose:

**Estimate Duration:** To estimate how long a program will run.

**Estimate Input Size:** To estimate the largest input that can reasonably be given to the program.

**Compare Algorithms:** To compare the efficiency of different algorithms for solving the same problem.

**Parts of Code:** To help you focus your attention on the parts of code that are executed the largest number of times. This is the code you need to improve to reduce running time.

**Choose Algorithm:** To choose an algorithm for an application.

- If the input won't be larger than six, don't waste your time writing an extremely efficient algorithm.
- If the input size is a thousand, then be sure the program runs in polynomial not exponential time.
- If you are working on the Gnome project and the input size is a billion, then be sure the program runs in linear time.

**Time and Space Complexities are Functions,  $T(n)$  and  $S(n)$ :** The time complexity of an algorithm is not a single number but is a function indicating how the running time depends on the *size* of the input. We often denote this by  $T(n)$ , giving the number of "operations" executed on the worst case input instance of "size"  $n$ . An example would be  $T(n) = 3n^2 + 7n + 23$ . Similarly,  $S(n)$  gives the "size" of the rewritable memory the algorithm requires.

**Ignoring Details,  $\Theta(T(n))$  and  $\mathcal{O}(T(n))$ :** Generally, we ignore the low order terms in the function  $T(n)$  and the multiplicative constant in front. We also ignore the function for small values of  $n$  and focus the *asymptotic* behavior as  $n$  becomes very large. Some of the reasons are the following.

**Model Dependent:** The multiplicative constant in front of the time depends on how fast the computer is and on the precise definition of “size” and “operation.”

**Too Much Work:** Counting every operation that the algorithm executes in precise detail is more work than it is worth.

**Not Significant:** It is much more significant whether the time complexity is  $T(n) = n^2$  or  $T(n) = n^3$  than whether it is  $T(n) = n^2$  or  $T(n) = 3n^2$ .

**Large  $n$  Matter:** One might say that we only consider large input instances in our analysis, because the running time of an algorithm only becomes an issue when the input is large. However, the running time of some algorithms on small input instances is quite critical. In fact, the size  $n$  of a realistic input instance depends on both on the problem and on the application. The choice was made to consider only large  $n$  in order to provide a clean and consistent mathematical definition.

See Theta and Big Oh notations in Section 3.

## 4.1 Different Models of Time and Space Complexity

We will now formally define the time (and space) complexity of an algorithm and of a computational problem.

**Size and Operations:** A model of time complexity is defined by what is considered to be the “size” of an input instance and what is considered to be a single algorithmic “operation.” Different models are used for different applications depending on what we want to learn about the algorithm and at what level of detail.

Typically, the definitions of “size” and an “operation” are tied together. A basic unit is defined and then the size of an input instance is defined to be the number of these units needed to express the input and a single operation is able to do basic operations on one or two of these units. The following are the classic models used.

**Ignoring Multiplicative Constants:** The following models are equivalent up to some multiplicative constant.

**Intuitive: Seconds.** The obvious unit to measure *time* complexity is in seconds. In fact, a very intuitive definition of time complexity  $T(n)$ , is that an adversary is allowed to send you a hard input instance along a narrow channel and  $T(n)$  is the number of seconds that it takes you to solve the problem as a function of the number of seconds it takes the adversary to send the instance. The problem with this definition is that it is very dependent on the speed of communication channel and of the computer used. However, this difference is only up to some multiplicative constant.

An algorithm is thought to be slow if an adversary can quickly give you an instance to the problem that takes the algorithm a huge amount of time to solve, perhaps

even the age of the universe. In contrast, the algorithm is considered fast, if in order to force it to work hard, the adversary must also work hard communicating the instance.

**Intuitive: The Size of Your Paper.** Similarly, the size of an input instance can be defined to be the number of square inches of paper used by the adversary to write down the instance given a fixed symbol size. Correspondingly, as single operation would be to read, write, or manipulate one or two of these symbols. Similarly, the space requirements of the algorithm is the number of square inches of paper, with eraser, used by the algorithm to solve the given input instance.

**Formal: Number of Bits.** In theoretical computer science, the formal definition of the size of an instance is the number of binary bits required to encode it. For example, if the input is  $41_{10} = 101001_2$ , then  $n = 6$ . The only allowable operations are *AND*, *OR*, and *NOT* of pairs of bits. The algorithm, in this context, can be viewed as some form of a *Turing Machine* or a circuit of *AND*, *OR*, and *NOT* gates. Most people get nervous around this definition. That's okay. We have others.

**Practical: Number of Digits, Characters, or Words.** It is easier to think of integers written in decimal notation and to think of other things as written in sentences composed of the letters A-Z. A useful definition of the size of an instance is the number of digits or characters required to encode it. You may argue that an integer can be stored as a single variable, but for a really big value, your algorithm will need a data structure consisting of a number of 32-bit words to store it. In this case, the size of an instance would be the number of these words needed to represent it. For each of these definitions of size, the definition of an operation would be a single action on a pair of digits, characters, or 32-bit words.

**Within a Constant:** These definitions of size are equivalent within a multiplicative constant. Given a value  $N$ , the number of bits to represent it will be  $\lceil \log_2(N + 1) \rceil$ , the number of digits will be  $\lceil \log_{10}(N + 1) \rceil = \lceil \frac{1}{3.32} \log_2(N + 1) \rceil$ , and the number of 32-bit words will be  $\lceil \log_{(2^{32})}(N + 1) \rceil = \lceil \frac{1}{32} \log_2(N + 1) \rceil$ . See the rules about logarithms in Section 2.

**Ignoring Logarithmic Multiplicative Factors:** The following model can be practically easier to handle than the bit model and it gives the same measure of time complexity except for being a few logarithmic factors more pessimistic. This is not usually a problem. The only time when this text cares about these factors is when differentiating between  $T(n) = n$  time and  $T(n) = n \log_2 n$  time, particularly when regarding algorithms for sorting.

**Practical: Number of Bounded Elements.** Suppose the input consists of an array, tree, graph, or strange data structure consisting of  $n$  integers (or more complex objects), each in the range  $[1..n^5]$ . A reasonable definition of size would be  $n$ , the number of integers. A reasonable definition of an operation allows two such integers to be added or multiplied and the indexing into array using one integer in order to retrieve another.

**Within a Log:** Surprisingly perhaps, the time complexity of an algorithm is more pessimistic under this model than under the bit model. Suppose, for example, that under this model the algorithm requires  $T(n) = n^3$  integer operations. Each integer in the range  $[1..n^5]$  requires  $\log_2 n^5 = 5 \log_2 n$  bits. Hence, the size of the entire input instance is  $n_{bit} = n \times 5 \log_2 n$  bits. Adding two  $n_{bit}$  bit

integers requires some constant times  $n_{bit}$  bit operations. To be concrete, let us say  $3n_{bit}$  bit operations. This gives that the total number of bit operations required by the algorithm is  $T_{bit} = (15 \log_2 n) \cdot T(n) = 15n^3 \log_2 n$ . Rearranging  $n_{bit} = n \times 5 \log_2 n$  gives  $n = \frac{n_{bit}}{5 \log_2 n}$  and plugging this into  $T_{bit} = 15n^3 \log_2 n$  gives  $T_{bit}(n_{bit}) = 15 \left( \frac{n_{bit}}{5 \log_2 n} \right)^3 \log_2 n = \frac{0.12}{(\log_2 n)^2} (n_{bit})^3$  bit operations. This gives the time complexity of the algorithm in the bit model. Note that this is two logarithmic factors,  $\frac{0.12}{(\log_2 n)^2}$ , better than time complexity  $T(n) = n^3$  within the element model.

### An Even Stronger Complexity Measure:

**Practical: Number of Unbounded Elements.** Again suppose that the input consists of an array, tree, graph, or strange data structure consisting of  $n$  integers (or more complex objects), but now there is no bound on how large each can be. Again we will define the size of an input instance to be the number of integers and a single operation to be some operation on these integers, no matter how big these integers are. Being an even more pessimistic measure of time complexity, having an algorithm that is faster according to this measure is a much stronger statement. The reason is that the algorithm must be able to solve the problem in the same  $T(n)$  integer operations independent of how big the integers are that the adversary gives it.

### Too Strong A Complexity Measure:

**Incorrect: The Value of an Integer.** When the input is an integer, it is tempting to denote its value with  $n$  and use this as the size of the input. DO NOT DO THIS! The value or magnitude of an integer is very very different than the number of bits needed to write it down. For example,  $N = 100$  is a very big number. It is bigger than the number of atoms in the universe. However, we can write it on 4 inches of paper, using 50 digits. In binary, it would require 166 bits.

Any of the above models of time complexity are reasonable except for the last one.

### Which Input of Which Size:

**Input Size  $n$  Is Assumed to Be Very Large:** Suppose one program takes  $100n$  time and another takes  $n^2$  time. Which is faster depends on the input size  $n$ . The first is slower for  $n < 100$ , and the second for larger inputs. It is true that some applications deal with smaller inputs, but in order to have a consistent way to compare running times, we assume that the input is very large.

**Which Input Do We Use To Determine Time Complexity?:**  $T(n)$  is the time required to execute the given algorithm on an input of size  $n$ . However, which input instance of this size do you give the program to measure its complexity? After all, there are  $2^n$  input instances with  $n$  bits. Here are three possibilities:

**A Typical Input:** The problem with this is that it is not clear who decides what a “typical” input is. Different applications will have very different typical inputs. It is valid to design your algorithm to work well for a particular application, but if you do, this needs to be clearly stated.

**Average Case or Expected:** Average case analysis considers the time averaged over-all input instances of size  $n$ . This is equivalent to the expected time for a “random” input instance of size  $n$ . The problem with this definition is that it assumes that all instances are equally likely to occur. One might just as well consider a different probability distribution on the inputs.

**Worst Case:** The usual measure is to consider the instance of size  $n$  on which the given algorithm is the slowest. Namely,  $T(n) = \max_{I \in \{I \mid |I|=n\}} Time(I)$ . This measure provides a nice clean mathematical definition and is the easiest to analyze. The only problem is that sometimes the algorithm does much better than the worst case, because the worst case is not a reasonable input. One such algorithm is Quick Sort.

### Upper and Lower Bounds on Time Complexity:

**Running Time of an Algorithm:** The time complexity of an algorithm is the largest time required on any input instance of size  $n$ .

- O:** “My algorithm runs in  $\mathcal{O}(n^2)$  time” means that for some constant  $c$ , you have proven that for every input instance of size  $n$ , it does not take any more than  $cn^2$  time. It may be much faster.
- $\Omega$ :** “My algorithm runs in  $\Omega(n^2)$  time” means that for some constant  $c$ , you have found one input instance of every (most) sizes  $n$  for which your algorithm takes at least  $cn^2$  time. It may take even longer.
- $\Theta$ :** “My algorithm runs in  $\Theta(n^2)$  time” means that it is both  $\mathcal{O}(n^2)$  and  $\Omega(n^2)$ . For some constants  $0 < c_1 \leq c_2$ , the largest time required on any input instance of size  $n$  is between  $c_1 \times n^2$  and  $c_2 \times n^2$  time. Note that there may be instances of size  $n$  for which the algorithm runs much faster.
- $o$ :** Suppose that you are proving that your algorithm runs in  $\Theta(n^3)$  time, and there is a subroutine that gets called once that requires only  $\Theta(n^2)$  time. Because  $n^2 = o(n^3)$ , its running time is insignificant to the overall time. Hence, you would say, “Don’t worry about the subroutine, because it runs in  $o(n^3)$  time.”

**Time Complexity of a Problem:** The time complexity of a problem is the time complexity of the fastest algorithm that solves the problem.

- $\mathcal{O}$ :** You say a problem has time complexity  $\mathcal{O}(n^2)$  when you have a  $\Theta(n^2)$  algorithm for it. This does not tell you that the problem’s complexity is  $\Theta(n^2)$ , because there may be a faster algorithm to solve the problem that you do not know about.
- $\Omega$ :** To say that the problem has time complexity  $\Omega(n^2)$  is a big statement. It not only means that nobody knows a faster algorithm; it also means that no matter how smart you are, it is impossible to come up with a faster algorithm, because such an algorithm simply does not exist.
- $\Theta$ :** “The time complexity of the problem is  $\Theta(n^2)$ ” means that you have an upper bound and a lower bound on the time which match. The upper bound provides an algorithm, and the lower bound proves that no algorithm could do better.

See the lower bound in Section 28.1 for a discussion on what is needed to formally prove upper or lower bounds on the time complexity of a problem.

## 4.2 Examples

**Example 1:** Suppose program  $P_1$  requires  $T_1(n) = n^4$  operations and  $P_2$  requires  $T_2(n) = 2^n$ . Suppose that your machine executes  $10^6$  operations per second.

1. If  $n = 1,000$ , what is the running time of these programs?
  - Answer:
    - (a)  $T_1(n) = (10^3)^4 = 10^{12}$  operations, requiring  $10^6$  seconds or equivalently 11.6 days.
    - (b)  $T_2(n) = 2^{(10^3)}$  operations. The number of years is  $\frac{2^{(10^3)}}{10^6 \cdot 60 \cdot 60 \cdot 356}$ . This is too big for my calculator. The log of this number is  $\frac{10^3}{\log_2(10)} - \log_{10} 10^6 - \log_{10}(60 \cdot 60 \cdot 356) = 301.03 - 6 - 6.12 = 288.91$ . Therefore, the number of years is  $10^{288.91}$ . Don't wait for it.
  - 2. If you want to run your program in 24 hrs, how big can your input be?
    - Answer: The number of operations is  $T = 24 * 60 * 60 * 10^6 = 8.64 \cdot 10^{10}$ .  $n_1 = T^{1/4} = 542$ .  $n_2 = \log_2 T = \frac{\log T}{\log 2} = 36$ .
  - 3. Approximately, for which input size, do the programs have the same running times?
    - Answer: Setting  $n = 16$  gives  $n^4 = (16)^4 = (2^4)^4 = 2^{4 \cdot 4} = 2^{16} = 2^n$ .

**Example 2:** Two simple algorithms, summation and factoring.

### The Problems/Algorithms:

**Summation:** The task is to sum the  $N$  entries of an array, i.e.,  $A(1) + A(2) + A(3) + \dots + A(N)$ .

**Factoring:** The task is to find divisors of an integer  $N$ . For example, on input  $N = 5917$  we output that  $N = 97 \times 61$ . (This problem is central to cryptography.) The algorithm checks whether  $N$  is divisible by 2, by 3, by 4, ... by  $N$ .

**Time:** Both algorithms require  $T = N$  operations (additions or divisions).

**How Hard?** The Summing algorithm is considered to be very fast, while the factoring algorithm is considered to be very time consuming. However, both algorithms take  $T = N$  time to complete. The time complexity of these algorithms will explain why.

**Typical Values of  $N$ :** In practice, the  $N$  for Factoring is much larger than that for Summation. Even if you sum all the entries in the entire 8G hard drive, then  $N$  is still only  $N \approx 10^{10}$ . On the other hand, the military wants to factor integers  $N \approx 10^{100}$ . However, the complexity of an algorithm should not be based on how it happens to be used in practice.

**Size of the Input:** The input for Summation is  $n \approx 32N$  bits.

The input for Factoring is  $n = \log_2 N$  bits. Therefore, with a few hundred bits you can write down a difficult factoring instance that is seemingly impossible to solve.

**Time Complexity:** The running time of Summation is  $T(n) = N = \frac{1}{32}n$ , which is linear in its input size.

The running time of Factoring is  $T(N) = N = 2^n$ , which is exponential in its input size. This is why the Summation algorithm is considered to be *feasible*, while the Factoring algorithm is considered to be *infeasible*.

# Chapter 5

## Adding Made Easy Approximations

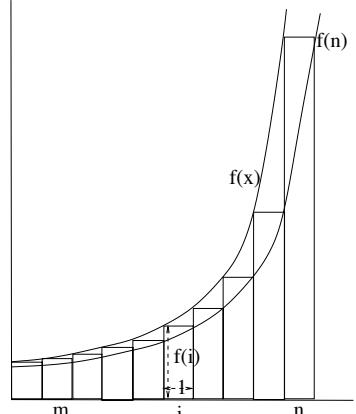
```

algorithm Eg(n)
    loop i = 1..n
        loop j = 1..i
            loop k = 1..j
                put "Hi"
            end loop
        end loop
    end loop
end algorithm

```

The inner loop requires time  $\sum_{k=1}^j 1 = j$ .  
The next requires  $\sum_{j=1}^i \sum_{k=1}^j 1 = \sum_{j=1}^i j = \Theta(i^2)$ .  
The total is  $\sum_{i=1}^n \sum_{j=1}^i \sum_{k=1}^j 1 = \sum_{i=1}^n \Theta(i^2) = \Theta(n^3)$ .

Sums arise often in the study of computer algorithms. For example, if the  $i^{th}$  iteration of a loop takes time  $f(i)$  and it loops  $n$  times, then the total time is  $f(1) + f(2) + f(3) + \dots + f(n)$ . This we denoted with  $\sum_{i=1}^n f(i)$ . It can be approximated by integrating the function  $\int_{x=1}^n f(x) \delta x$ , because the first is the area under the stairs of height  $f(i)$  and the second under the curve  $f(x)$ . (In fact, both  $\sum$  (the Greek letter sigma) and  $\int$  are “S” for sum.) Note that, even though the individual terms are indexed by  $i$  (or  $x$ ), the total is a function of  $n$ . The goal now is to approximate  $\sum_{i=1}^n f(i)$  for various functions  $f(i)$ .



Beyond learning the classic techniques for computing  $\sum_{i=1}^n 2^i$ ,  $\sum_{i=1}^n i$ , and  $\sum_{i=1}^n \frac{1}{i}$ , we do not cover how to evaluate sums exactly, but only how to approximate them to within a constant factor. Every computer scientist uses that if  $f(n) \in 2^{\Theta(n)}$ , then  $\sum_{i=1}^n f(i) \in \Theta(f(n))$ . This, however, is not usually the way it is taught, partly because it is not always true. We have formally proven when it is true and when not. We call it the *Adding Made Easy Technique*. It provides a few easy rules with which you will be able to compute  $\Theta(\sum_{i=1}^n f(i))$  for almost every function  $f(n)$  that you will encounter.

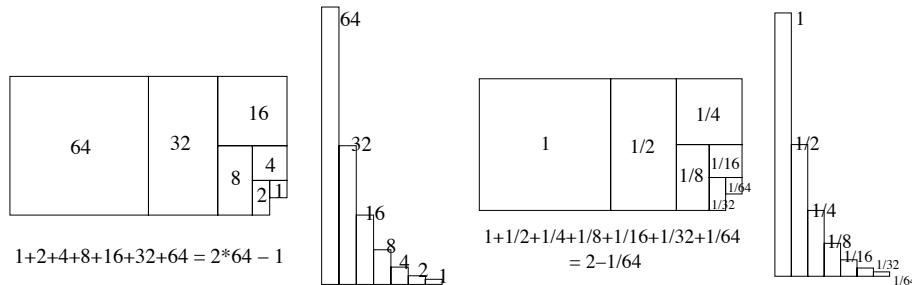
### 5.1 Intuition and Techniques

This subsection summarizes what will be proved within in this section.

**Four Different Classes of Solutions:** All of the sums that we will consider, have one of four different classes of solutions. The intuition for each is quite straight forward.

**Geometric Increasing:** If the terms grow very quickly, the total is dominated by the last and biggest term  $f(n)$ . Hence, one can approximate the sum by only considering the last term, namely  $\sum_{i=1}^n f(i) = \Theta(f(n))$ .

**Example:** Consider the classic sum in which each of the  $n$  terms is twice the previous,  $1 + 2 + 4 + 8 + 16 + \dots + 2^n$ . Either by examining areas within this picture or using simple induction, one can prove that the total is always one less than twice the biggest term, namely  $\sum_{i=0}^n 2^i = 2 \cdot 2^n - 1 = \Theta(2^n)$ .

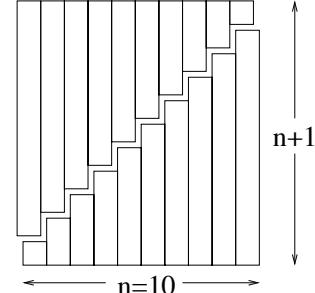


**Arithmetic:** If half of the terms are roughly the same size, then the total is roughly the number of terms times the last term, namely  $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$ .

**Constant Example:** Clearly the sum of  $n$  ones is  $n$ , i.e.,  $\sum_{i=1}^n 1 = n = \Theta(n \cdot f(n))$ .

**Example:** The classic example is the sum in which each of the  $n$  terms is only one bigger than the previous,  $1 + 2 + 3 + 4 + 5 + \dots + n$ . A proof that the total is  $\sum_{i=1}^n i = \frac{n(n+1)}{2}$  is as follows.

$$\begin{aligned} S &= 1 + 2 + 3 + \dots + n-2 + n-1 + n \\ S &= n + n-1 + n-2 + \dots + 3 + 2 + 1 \\ 2S &= n+1 + n+1 + n+1 + \dots + n+1 + n+1 + n+1 \\ &= n \cdot (n+1) \\ S &= \frac{1}{2}n \cdot (n+1) \end{aligned}$$

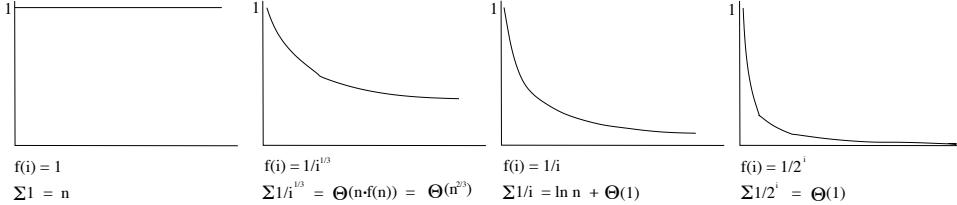


**Bounded Tail:** If the terms shrink quickly, the total is dominated by the first and biggest term  $f(1)$ , which is assumed here to be  $\Theta(1)$ , i.e.,  $\sum_{i=1}^n f(i) = \Theta(1)$ .

**Example:** The classic sum here is when each term is half of the previous,  $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots + \frac{1}{2^n}$ . Again using either the areas within the picture or induction, one can prove that the total approaches but never reaches 2, namely  $\sum_{i=1}^n (\frac{1}{2})^i = 1 - (\frac{1}{2})^n = \Theta(1)$ .

**The Harmonic Sum:** The sum  $\sum_{i=1}^n \frac{1}{i}$  is referred to as the *harmonic sum* because of its connection to music. It arises surprisingly often and it has an unexpected total.

**On the Boundary:** The boundary between those sums for which  $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$  and those for which  $\sum_{i=1}^n f(i) = \Theta(1)$  occurs when these approximations meet,



i.e. when  $\Theta(n \cdot f(n)) = \Theta(1)$ . This occurs at the harmonic function  $f(n) = \frac{1}{n}$ . Given that both approximations say the total is  $\sum_{i=1}^n \frac{1}{i} = \Theta(1)$ , it is reasonable to think that this is the answer, but it is not.

**The Total:** It turns out that the total is within one of the natural logarithm,  $\sum_{i=1}^n \frac{1}{i} = \ln n + \Theta(1)$ . This is, of course, relatively close to  $\Theta(1)$ .

**More Examples:** The following examples are sorted from the sum of very fast growing functions to the sum of very fast decreasing functions.

<b>Double Exponential:</b>	$\sum_{i=0}^n 2^{2^i}$	$\approx 1 \cdot 2^{2^n}$	$= \Theta(f(n))$ .
<b>General Exponential:</b>	$\sum_{i=0}^n b^i$	$\approx \frac{b}{b-1} \cdot b^n = \Theta(b^n)$	$= \Theta(f(n))$ .
<b>Classic Geometric:</b>	$\sum_{i=0}^n 2^i$	$= 2 \cdot 2^n - 1 = \Theta(2^n)$	$= \Theta(f(n))$ .
<b>General Polynomial:</b>	$\sum_{i=1}^n i^d$	$= \frac{1}{d+1} n^{d+1} + \Theta(n^d) = \Theta(n^{d+1})$	$= \Theta(n \cdot f(n))$ .
<b>Quadratic:</b>	$\sum_{i=1}^n i^2$	$= \frac{1}{3} n^3 + \frac{1}{2} n^2 + \frac{1}{6} n = \Theta(n^3)$	$= \Theta(n \cdot f(n))$ .
<b>Classic Arithmetic:</b>	$\sum_{i=1}^n i$	$= \frac{1}{2} n(n+1) = \Theta(n^2)$	$= \Theta(n \cdot f(n))$ .
<b>Constant:</b>	$\sum_{i=1}^n 1$	$= n = \Theta(n)$	$= \Theta(n \cdot f(n))$ .
<b>Above Harmonic:</b>	$\sum_{i=1}^n \frac{1}{n^{0.999}}$	$\approx 1,000 n^{0.001} = \Theta(n^{0.001})$	$= \Theta(n \cdot f(n))$ .
<b>Harmonic:</b>	$\sum_{i=1}^n \frac{1}{n}$	$= \ln n + \Theta(1)$	$= \Theta(\ln n)$ .
<b>Below Harmonic:</b>	$\sum_{i=1}^n \frac{1}{n^{1.001}}$	$\approx 1,000$	$= \Theta(1)$ .
<b>1 over a Polynomial:</b>	$\sum_{i=1}^n \frac{1}{n^2}$	$\approx \frac{\pi}{6} \approx 1.5497..$	$= \Theta(1)$ .
<b>Geometric Decreasing:</b>	$\sum_{i=1}^n \left(\frac{1}{2}\right)^i$	$= 1 - \left(\frac{1}{2}\right)^n$	$= \Theta(1)$ .

**The Adding Made Easy Technique:** Once you have determined which of these four classes a given function  $f(n)$  falls into, the following simple rules provide the approximation of the sum  $\Theta(\sum_{i=1}^n f(i))$ .

Geometric Increasing	Arithmetic	Harmonic	Bounded Tail
$\sum_{i=1}^n f(i) = \Theta(f(n))$	$\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$	$\sum_{i=1}^n f(i) = \Theta(\ln n)$	$\sum_{i=1}^n f(i) = \Theta(1)$

**Determining the Class:** Which of these classes a function is in depends on how quickly it grows and is determined by quick rules. See Figure 5.1.

**Functions with a Basic Form:** Most functions that you will need to sum up will fit into the basic form  $f(n) = \Theta(b^{an} \cdot n^d \cdot \log^e n)$ , where  $a, b, d$ , and  $e$  are real constants. The values of these constants determine which of the four classes the function falls into.

Geometric Increasing	Arithmetic	Harmonic	Bounded Tail
$f(n) = \Theta(b^{an} \cdot n^d \cdot \log^e n)$ , $a > 0, b > 1, d, e \in (-\infty, \infty)$	$f(n) = \Theta(n^d \cdot \log^e n)$ $d > -1, e \in (-\infty, \infty)$	$f(n) = \Theta(\frac{1}{n})$	$f(n) = \Theta(\frac{\log^e n}{n^d})$ $d > 1, e \in (-\infty, \infty)$ <hr/> $f(n) = \Theta(\frac{n^d \cdot \log^e n}{b^{an}})$ $a > 0, b > 1, d, e \in (-\infty, \infty)$

**Simple Analytical Functions:** Now consider functions that do not fit into the basic form  $f(n) = \Theta(b^{an} \cdot n^d \cdot \log^e n)$ , yet still can be expressed with  $n$ , real constants, plus, minus,

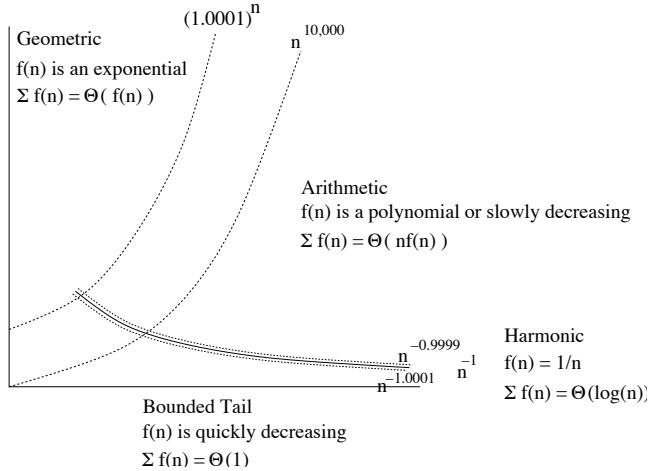


Figure 5.1: Boundaries between geometric, arithmetic, harmonic, and bounded tail.

times, divide, exponentiation, and logarithms. Such functions are said to be *simple analytical*. Which of the four classes the function falls into can be determined by the following simple rules.

Geometric Increasing	Arithmetic	Harmonic	Bounded Tail
$f(n) \geq 2^{\Omega(n)}$	$f(n) = n^{\Theta(1)-1}$	$f(n) = \Theta(\frac{1}{n})$	$f(n) \leq n^{-1-\Omega(1)}$

#### Functions That Lie Between These Cases:

**Between Arithmetic and Geometric:** Some functions,  $f(i)$ , grow too slowly to be “geometric” and too quickly to be “arithmetic.” For these, the behavior of the sum  $\sum_{i=1}^n f(i)$  is between that for the geometric and the arithmetic sums. Specifically,  $\Omega(f(n)) \leq \sum_{i=1}^n f(i) \leq O(n \cdot f(n))$ . Two examples are  $f(n) = n^{\log n}$  and  $f(n) = 2^{\sqrt{n}}$ . Their sums are  $\sum_{i=1}^n i^{\log i} = \Theta(\frac{n}{\log n} n^{\log n}) = \Theta(\frac{n}{\log n} f(n))$  and  $\sum_{i=1}^n 2^{\sqrt{i}} = \Theta(\sqrt{n} f(n))$ .

**Between Arithmetic, Harmonic, and Bounded Tail:** The function  $f(n) = \frac{\log n}{n}$ , for example is between being arithmetic and harmonic. The function  $f(n) = \frac{1}{n \log n}$  is between being harmonic and having a bounded tail. Their sums are  $\sum_{i=1}^n \frac{\log i}{i} = \Theta(\log^2 n)$  and  $\sum_{i=1}^n \frac{1}{i \log i} = \Theta(\log \log n)$ .

**Arbitrary Functions:** Recall that the goal here is to predict the sum  $\sum_{i=1}^n f(i)$  from the value of the last term  $f(n)$ . We are unable to do this if the terms change in an unpredictable way. Section 5.3 gives functions constructed from sines, cosines, floors, or ceilings that can fail to behave as is needed for our Adding Made Easy Technique, and hence will not be considered.

#### Examples:

**A Detailed Geometric Example:** Suppose that we want to determine the sum  $\sum_{i=1}^n 8 \frac{2^i}{i^{100}} + i^3$ .

**Determining the Class:** The first step is to classify the function  $f(n) = 8 \frac{2^n}{n^{100}} + n^3$ .

**Functions with a Basic Form:** Because  $f(n)$  has the basic form  $\Theta(b^{an} \cdot n^d \cdot \log^e n)$ , where  $a = 1 > 0$  and  $b = 2 > 1$ , we know immediately that  $f(n)$  follows the *geometric increasing* rule of the Adding Made Easy Technique.

**Simple Analytical Functions:** We could also determine this same result, using the more general technique. First we check that  $f(n) = 8\frac{2^n}{n^{100}} + n^3$  is a simple analytical function because it is expressed with  $n$ , real constants, plus, minus, times, divide, exponentiation, and logarithms and not with stranger things like sines or floors. From Section 3, we should already know that the term  $n^3$  and the constant 8 are not significant when determining that the order of the function is  $f(n) \in \Theta(\frac{2^n}{n^{100}})$ . Moreover the  $n^{100}$  is not significant in determining that the function also falls into the larger class of exponential functions, namely  $f(n) \in 2^{\Omega(n)}$ . From this, we reconfirm that this sum is *geometric increasing*.

**Four Different Solutions:** Now that we have determined that the  $\sum_{i=1}^n f(i)$  is a geometric increasing sum, we immediately know that the terms grow quickly enough for the total to be dominated by the last and biggest term  $f(n)$ . The following simple rule  $\sum_{i=1}^n f(i) = \Theta(f(n))$  approximates the sum.

**Determining the Solution:** Again, we use our knowledge from Section 3, that the order of the function is  $f(n) \in \Theta(\frac{2^n}{n^{100}})$  in order to conclude that  $\sum_{i=1}^n f(i) = \Theta(f(n)) = \Theta(\frac{2^n}{n^{100}})$ .

**Two Levels of Classification:** Knowing that your pet falls into the very large class of “mammal” tells you that he forms complex social networks. Knowing that he falls into the more specific class “dog” tells you moreover that he is man’s best friend. Similarly, knowing that your function falls into the very large class  $2^{\Omega(n)}$  tells you to use the adding rule  $\sum_{i=1}^n f(i) = \Theta(f(n))$  and knowing that it falls into the more specific class  $\Theta(\frac{2^n}{n^{100}})$  tells you that  $\sum_{i=1}^n f(i) = \Theta(f(n)) = \Theta(\frac{2^n}{n^{100}})$ .

**A Detailed Arithmetic Example:** Now suppose that we want to determine the sum  $\sum_{i=1}^n \frac{1}{i^3}$ .

**Determining the Class:** The first step is to classify the function  $f(n) = \frac{1}{n^{\frac{1}{3}}} = n^{-\frac{1}{3}}$ .

**Functions with a Basic Form:** Because  $f(n)$  has the basic form  $\Theta(b^{an} \cdot n^d \cdot \log^e n)$ , where  $a = 0$ ,  $b = 1$ , and  $d = -\frac{1}{3} > -1$ , we know that  $f(n)$  follows the *arithmetic* rule of the Adding Made Easy Technique.

**Simple Analytical Functions:** The property  $f(n) = n^{\Theta(1)-1}$  is an odd one that we have not seen before. However, remember that  $\Theta(1)$  basically means a “constant” strictly bigger than 0. Hence,  $n^{\Theta(1)-1}$  amounts to  $n^d$ , where  $d$  is a “constant” strictly bigger than -1. Hence,  $f(n) = n^{-\frac{1}{3}}$  easily passes this test.

**Determining the Solution:** Knowing that  $\sum_{i=1}^n f(i)$  is an arithmetic sum, we immediately know that the terms are similar enough that the total is approximated by the number of terms times the last term  $f(n)$ . Therefore,  $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n)) = \Theta(n \cdot \frac{1}{n^{\frac{1}{3}}}) = \Theta(n^{\frac{2}{3}})$  approximates the sum.

**More Examples:**

### Geometric Increasing:

- $\sum_{i=1}^n 3^{i \log i} + 5^i + i^{100} = \Theta(3^{n \log n})$ .
- $\sum_{i=1}^n 2^{i^2} + i^2 \log i = \Theta(2^{n^2})$ .
- $\sum_{i=1}^n 2^{2^i - i^2} = \Theta(2^{2^n - n^2})$ .

### Arithmetic (Increasing):

- $\sum_{i=1}^n i^4 + 7i^3 + i^2 = \Theta(n^5)$ .
- $\sum_{i=1}^n i^{4.3} \log^3 i + i^3 \log^9 i = \Theta(n^{5.3} \log^3 n)$ .

### Arithmetic (Decreasing):

- $\sum_{i=2}^n \frac{1}{\log i} = \Theta(\frac{n}{\log n})$ .
- $\sum_{i=1}^n \frac{\log^3 i}{i^{0.6}} = \Theta(n^{0.4} \log^3 n)$ .

### Bounded Tail:

- $\sum_{i=1}^n \frac{\log^3 i}{i^{1.6} + 3i} = \Theta(1)$ .
- $\sum_{i=1}^n \frac{i^{100}}{2^i} = \Theta(1)$ .
- $\sum_{i=1}^n \frac{1}{2^{2^i}} = \Theta(1)$ .

**Sums with a Different Form:** Above we considered sums of the form  $\sum_{i=1}^n f(i)$  for simple analytical functions  $f(n)$ . We will now consider sums that do not fit into this basic form.

**Other Ranges:** The sums considered so far have had  $n$  terms indexed by the variable  $i$  running from  $i = 1$  to  $i = n$ . Other ranges, however, might arise.

**Different Variable Names:** If  $\sum_{i=1}^n i^2 = \Theta(n^3)$ , it should not be too hard to know that  $\sum_{j=1}^m j^2 = \Theta(m^3)$ .

**Different Starting Indices:** A useful fact is  $\sum_{i=m}^n f(i) = \sum_{i=1}^n f(i) - \sum_{i=1}^{m-1} f(i)$ .

- $\sum_{i=m}^n i^2 = \Theta(n^3) - \Theta(m^3)$ , which is  $\Theta(n^3)$  unless  $m$  is very close to  $n$ .
- $\sum_{i=m}^n \frac{1}{i} = \Theta(\log n) - \Theta(\log m) = \Theta(\log \frac{n}{m})$ .

**Different Ending Indices:** Instead having  $n$  terms, the number of terms may be a function of  $n$ .

- $\sum_{i=1}^{5n^2+n} i^3 \log i$ : To solve this let  $N$  denote the number of terms. The Adding Made Easy Technique gives that  $\sum_{i=1}^N i^3 \log i = \Theta(N \cdot f(N)) = \Theta(N^4 \log N)$ . Substituting back in  $N = 5n^2 + n$  gives  $\sum_{i=1}^{5n^2+n} i^3 \log i = \Theta((5n^2 + n)^4 \log(5n^2 + n)) = \Theta(n^8 \log n)$ .
- $\sum_{i=1}^{\log n} 3^i i^2 = \Theta(f(N)) = \Theta(3^N N^2) = \Theta(3^{\log n} (\log n)^2) = \Theta(n^{\log 3} \log^2 n)$ .

**Terms Depending on  $n$ :** It is quite usual to have a sum where both the number of terms and the terms themselves depend on the same variable  $n$ , namely  $\sum_{i=1}^n i \cdot n$ . This can arise, for example, when computing the running time of two nested loops, where there are  $n$  iterations of the outer loop, the  $i^{th}$  of which requires  $i \cdot n$  time.

**The Variable  $n$  is a Constant:** Though  $n$  is a variable, dependent perhaps on the input size, for the purpose of computing the sum it is a constant. The reason is that its value does not change as we iterate from one term to the next. Only  $i$  changes. Hence, when determining whether the sum is arithmetic or geometric, the key is how the terms change with respect to  $i$ , not how it changes with respect to  $n$ . The only difference between this  $n$  within the term and say the constant 3 is that the  $n$  cannot be absorbed into the Theta.

**Factoring Out:** Often the dependence on  $n$  within the terms can be factored out. If so, this is the easiest way to handle the sums.

- $\sum_{i=1}^n i \cdot n \cdot m = nm \cdot \sum_{i=1}^n i = nm \cdot \Theta(n^2) = \Theta(n^3 m)$ .
- $\sum_{i=1}^n \frac{n}{i} = n \cdot \sum_{i=1}^n \frac{1}{i} = n \cdot \Theta(\log n) = \Theta(n \log n)$ .
- $\sum_{i=1}^{\log_2 n} 2^{(\log_2 n - i)} \cdot i^2 = 2^{\log_2 n} \cdot \sum_{i=1}^{\log_2 n} 2^{-i} \cdot i^2 = n \cdot \Theta(1) = \Theta(n)$ .

**The Adding Made Easy Technique:** The same general Adding Made Easy Technique can still be used when the indexing is not  $i = 1..n$  or when terms depend on  $n$ .

**Geometric Increasing,  $\Theta(f(n))$ :** If the terms are increasing geometrically, then the total is approximately the last term.

- $\sum_{i=1}^{\log_2 n} n^i$ . Note that this is a polynomial in  $n$  but is exponential in  $i$ . Hence, it is geometric increasing. The biggest term is  $n^{\log n}$  and the total is  $\Theta(n^{\log n})$ .

**Bounded Tail,  $\Theta(f(1))$ :** If the terms are decreasing geometrically or decreasing as fast as  $\frac{1}{i^2}$ , then the total is still Theta of the biggest term, but this is now the first term. The difference with what we were doing before, however, is that now this first term might not be  $\Theta(1)$ .

- $\sum_{i=\frac{n}{2}}^n \frac{1}{i^2} = \Theta\left(\frac{1}{(\frac{n}{2})^2}\right) = \Theta\left(\frac{1}{n^2}\right)$ .
- $\sum_{i=1}^{\log_2 n} 2^{(\log_2 n-i)} \cdot i^2$ . If you are nervous about this, the first thing to check is what the first and last terms are (and how many terms there are). The first term is  $f(1) = 2^{(\log n-1)} \cdot 1^2 = \Theta(n)$ . The last term is  $f(\log n) = 2^{(\log n-\log n)} \cdot (\log n)^2 = \Theta(\log^2 n)$ , which is significantly smaller than the first. This is an extra clue that the terms decrease geometrically in  $i$ . The total is then  $\Theta(f(1)) = \Theta(n)$ .

**Arithmetic,  $\Theta(\text{number of terms} \cdot \text{a typical term})$ :** If the sum is arithmetic then most of the terms contribute to the sum and the total is approximately the number of terms times a typical term. The new thing to note here is that the number of terms is not necessarily  $n$ .

- $\sum_{i=m}^n i \cdot n \cdot m$ . The number of terms is  $n - m$ . A typical term is  $n^2 m$ . The total is  $\Theta((n-m)n^2 m)$ .
- $\sum_{i=1}^{\log n} n \cdot i^2$ . The number of terms is  $\log n$ . A typical term is  $n \log^2 n$ . The total is  $\Theta(n \log^3 n)$ .

**Exercise 5.1.1** (*See solution in Section V*) Give the  $\Theta$  approximation of the following sums. Show your work.

1.  $\sum_{i=1}^n \sum_{j=1}^n i^j$
2.  $\sum_{i=1}^n \sum_{j=1}^n j^i$  (*Done separately*)
3.  $\sum_{i=1}^n \sum_{j=1}^i \frac{1}{j}$
4.  $\sum_{i=1}^n \sum_{j=1}^{i^2} ij \log(i)$

## 5.2 Proofs of the Adding Made Easy Technique

In this section, we use a few of the classic techniques and results about simple analytical functions to prove that the Adding Made Easy Technique works.

**Sufficiently Large  $n$ :** Though they are not likely to occur often in practice, we will be using our evaluation of sums by seeing how to handle functions  $f(n)$  which have one growth behavior for small values of  $n$ , i.e., for  $n \leq n_0$  for some constant  $n_0$ , and another growth behavior for large values, i.e., for  $n \geq n_0$ . For example,  $f(n) = n^{100} \cdot 2^n$  is dominated by the  $n^{100}$  for  $n \leq 1,024$  and by  $2^n$  for  $n \geq 1,024$ . Consequently, for  $n \leq n_0$ , the Adding Made Easy Technique gives

that  $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$  and for  $n \geq n_0$ , gives  $\sum_{i=1}^n f(i) = \Theta(f(n))$ . It turns out that both of these statements are true, but because  $\Theta$  notation considers asymptotic behavior only for sufficiently large values of  $n$ , we will concern ourselves only with approximating  $\sum_{i=1}^n f(i)$  for  $n \geq n_0$ . The way that we will do this is by breaking the sum into two parts,  $\sum_{i=1}^n f(i) = \sum_{i=1}^{n_0} f(i) + \sum_{i=n_0}^n f(i)$ . This is graphically seen in Figure 5.2. The first part of the sum  $\sum_{i=1}^{n_0} f(i)$  is at most  $[n_0 \cdot \max_{i=1}^{n_0} f(i)]$ , which may be a big amount, but whatever it is, it is a constant  $\Theta(1)$  independent of  $n$  because  $n_0$  is a constant. In the rest of this section, we will prove that the second part of the sum  $\sum_{i=n_0}^n f(i)$  is either  $\Theta(f(n))$ ,  $\Theta((n - n_0) \cdot f(n)) = \Theta(n \cdot f(n))$ , or  $\Theta(f(n_0)) = \Theta(1)$ . The conclusion is that for  $n \geq n_0$ , the total  $\sum_{i=1}^n f(i) = \sum_{i=1}^{n_0} f(i) + \sum_{i=n_0}^n f(i)$  is also respectively  $\Theta(f(n))$ ,  $\Theta(n \cdot f(n))$ , or  $\Theta(1)$ .

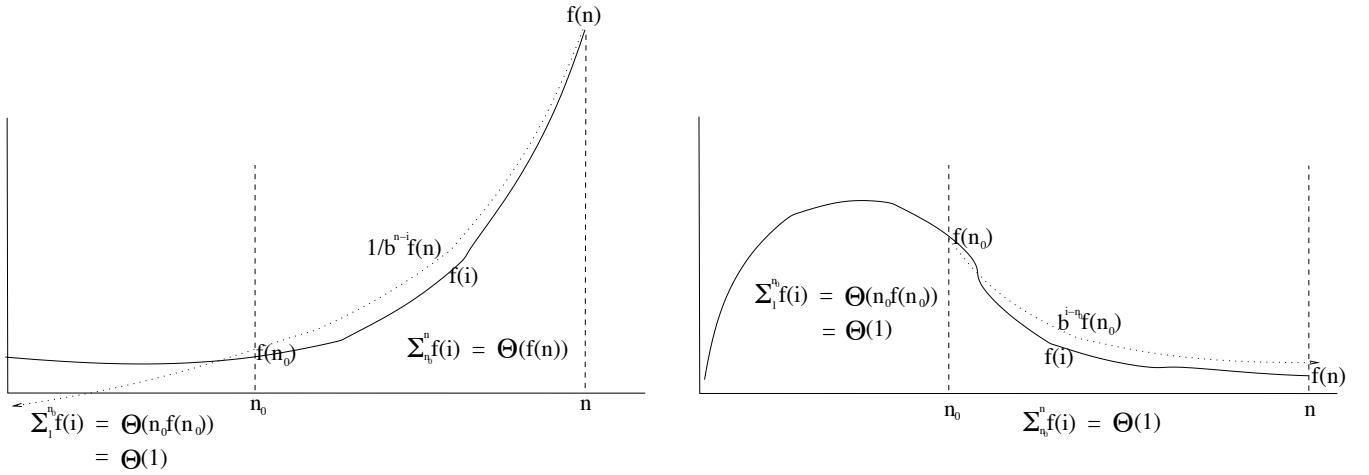


Figure 5.2: In both pictures, the total before  $n$  gets sufficiently large is some constant. In the first picture, the total for large  $n$  is bounded by an growing exponential and in the second by a decreasing exponential.

**Simple Increasing and Decreasing Geometric Sums:** Above we proved that  $\sum_{i=0}^n 2^i = \Theta(f(n)) = \Theta(2^n)$  and that  $\sum_{i=0}^n (\frac{1}{2})^i = \Theta(f(1)) = \Theta(1)$ . We will now generalize this to simple exponential functions  $f(n) = b^n$  for any constant  $b \neq 1$ .

**Theorem:** When  $b > 1$ ,  $\sum_{i=1}^n b^i = \Theta(f(n))$  and when  $b < 1$ ,  $\sum_{i=1}^n f(i) = \Theta(1)$ .

**Proof:**

$$\begin{aligned} S &= \sum_{i=0}^n b^i \\ &= 1 + b + b^2 + \dots + b^n \\ b \cdot S &= b + b^2 + b^3 + \dots + b^{n+1}. \end{aligned}$$

Subtract the above two lines gives

$$\begin{aligned} (1-b) \cdot S &= 1 - b^{n+1} \\ S &= \frac{1 - b^{n+1}}{1 - b} \text{ or } \frac{b^{n+1} - 1}{b - 1} \\ &= \frac{f(0) - f(n+1)}{1 - b} \text{ or } \frac{f(n+1) - f(0)}{b - 1} \\ &= \Theta(\max(f(0), f(n))). \end{aligned}$$

**Exercise 5.2.1** (See solution in Section V) Zeno's classic "paradox" is that Achilles has one kilometer to travel. First he must cover half his distance, then half of his remaining distance, then half of this remaining distance, .... He never arrives there. "The Story of Philosophy" by Bryan Magee states "People have found it terribly disconcerting. There must be a fault in the logic, they have said. But no one has yet been fully successful in demonstrating what it is. ... Perhaps one day it will be solved as someone has recently solved the problem of Fermat's Last Theorem." As a mathematician, I never found this to be a paradox. Certainly, Achilles must do an infinite number of such steps in order to reach his destination. Compute, assuming that he travels at one kilometer an hour, how much time his first such step takes him, his second, and his  $i^{\text{th}}$ . Resolve this ancient paradox by showing that the total time need for these infinite number of steps is in fact finite.

**Ratio Between Terms:** Considering again sums like  $f(n) = n^{100} \cdot 2^n$  which behaving exponentially for  $n \geq n_0$ . Here we prove that the total is dominated by the largest term, namely that either  $\sum_{i=n_0}^n f(i) = \Theta(f(n))$  or  $\sum_{i=n_0}^n f(i) = \Theta(f(n_0)) = \Theta(1)$ . Given that all the terms are positive, it is clear that the total is at least this largest term. Hence, what remains is that the total is not more than a constant times this term. To do this, we must compare each term  $f(i)$  with this largest term. One way to do this is to first compare each consecutive pairs of terms  $f(i)$  and  $f(i+1)$ .

**Theorem:** If for all sufficiently large  $i$ , the ratio between terms is bounded away from one, i.e.,  $\exists b > 1, \exists n_0, \forall i \geq n_0, \frac{f(i+1)}{f(i)} \geq b$ , then  $\sum_{i=1}^n f(i) = \Theta(f(n))$ .

Conversely, if  $\exists b < 1, \exists n_0, \forall i \geq n_0, \frac{f(i+1)}{f(i)} \leq b$ , then  $\sum_{i=1}^n f(i) = \Theta(1)$ .

**Examples:**

**Typical:** With  $f(i) = \frac{2^i}{i}$ , the ratio between consecutive terms is  $\frac{f(i+1)}{f(i)} = \frac{2^{i+1}}{i+1} \cdot \frac{i}{2^i} = 2 \cdot \frac{i}{i+1} = 2 \cdot \frac{1}{1+\frac{1}{i}}$  which is at least 1.99 for sufficiently large  $i$ .

**Not Bounded Away:** On the other hand, the arithmetic function  $f(i) = i$  has a ratio between the terms of  $\frac{i+1}{i} = 1 + \frac{1}{i}$ . Though this is always bigger than one, it is not bounded away from one by some constant  $b > 1$ .

**Simple Analytical Functions:** In Section 5.3, we will prove that if  $f(n)$  is simple analytical and  $f(n) \geq 2^{\Omega(n)}$ , then  $\exists b > 1, \exists n_0, \forall i \geq n_0, \frac{f(i+1)}{f(i)} \geq b$ , and hence  $\sum_{i=1}^n f(i) = \Theta(f(n))$ . Similarly, if  $f(n) \leq \frac{1}{2^{\Omega(n)}}$ , then  $\exists b < 1, \exists n_0, \forall i \geq n_0, \frac{f(i+1)}{f(i)} \leq b$ , and hence  $\sum_{i=1}^n f(i) = \Theta(1)$ .

**Proof:** If  $\forall i \geq n_0, \frac{f(i+1)}{f(i)} \geq b > 1$ , then it follows either by unwinding or induction that  $f(i) \leq (\frac{1}{b})^1 f(i+1) \leq (\frac{1}{b})^2 f(i+2) \leq (\frac{1}{b})^3 f(i+3) \leq \dots \leq (\frac{1}{b})^{n-i} f(n)$ .

See Figure 5.2. This gives that

$$\sum_{i=1}^n f(i) = \sum_{i=1}^{n_0} f(i) + \sum_{i=n_0}^n f(i) \leq \Theta(1) + \sum_{i=n_0}^n (\frac{1}{b})^{n-i} f(n) \leq \Theta(1) + f(n) \cdot \sum_{j=0}^n (\frac{1}{b})^j,$$

which we have already proved is  $\Theta(f(n))$ .

**Exercise 5.2.2** (See solution in Section V) Similarly prove that if  $\exists b < 1, \exists n_0, \forall i \geq n_0, \frac{f(i+1)}{f(i)} \leq b$ , then  $\sum_{i=n_0}^n f(i) = \Theta(f(n_0)) = \Theta(1)$ .

**Arithmetic Sums:** Above we proved that  $\sum_{i=1}^n i = \Theta(n \cdot f(n)) = \Theta(n^2)$ . We will now generalize this to any "increasing" simple analytical arithmetic function. We will consider the "decreasing" ones later.

**Either Increasing or Decreasing:** Because we define arithmetic functions to include polynomials  $f(n) = \Theta(n^d \cdot \log^e n)$  for  $d > -1$ ,  $e \in (-\infty, \infty)$  and more over generally functions  $f(n) \in n^{\Theta(1)-1}$ , these functions may increase or decrease. It will be useful for us to clearly distinguish between the “increasing” and the “decreasing” functions and to avoid problematic functions that both increase and decrease as  $n$  gets larger. The second corollary in Section 5.3 solves both of these problems by proving that if  $f(n)$  is simple analytical, then either there exists an  $n_0$  after which  $f(n)$  is non-decreasing or there exists an  $n_0$  after which  $f(n)$  is non-increasing. As said, we consider functions of the first type here and the second type later.

**Theorem:** If for sufficiently large  $n$ , the function  $f(n)$  is non-decreasing and  $f(\frac{n}{2}) = \Theta(f(n))$ , then  $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$ . (For simplicity, assume here that the function is non-decreasing after  $n_0 = 0$  and as an exercise, consider the case with larger  $n_0$ .)

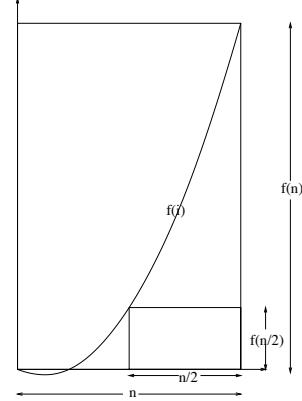
**Examples:**

**Typical:** The function  $f(n) = n^d$  for  $d \geq 0$  is non-decreasing and  $f(\frac{n}{2}) = (\frac{n}{2})^d = \frac{1}{2^d}f(n)$ .

**Without the Property:** The function  $f(n) = 2^n$  does not have this property because  $f(\frac{n}{2}) = 2^{\frac{n}{2}} = \frac{1}{2^2}f(n)$ .

**Simple Analytical Functions:** In Section 5.3, we will prove that if  $f(n)$  is simple analytical and  $f(n) = n^{\Theta(1)-1}$ , then  $f(\frac{n}{2}) = \Theta(f(n))$  and hence, if it also non-decreasing then  $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$ .

**Proof:** The method formally justifies the intuition that if half of the terms are roughly the same size, then the total is roughly the number of terms times the last term, namely  $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$ . Because  $f(i)$  is non-decreasing, half of the terms are at least the middle term  $f(\frac{n}{2})$  and all of the terms are at most the biggest term  $f(n)$ . Pictorially this is shown to the right. From these it follows that the sum  $\sum_{i=1}^n f(i)$  is at least half the number of terms times the middle term and at most the number of terms times the largest term, namely  $\frac{n}{2} \cdot f(\frac{n}{2}) \leq \sum_{i=1}^n f(i) \leq n \cdot f(n)$ . Because  $f(\frac{n}{2}) = \Theta(f(n))$ , these bounds match within a multiplicative constant and hence  $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$ .



**The Harmonic Sum:** The harmonic sum is a famous sum that arises surprisingly often. The total  $\sum_{i=1}^n \frac{1}{i}$  is within one of  $\log_e n$ . However, we will not bound it quite so closely.

**Theorem:**  $\sum_{i=1}^n \frac{1}{i} = \Theta(\log n)$ .

**Proof:** One way of approximating the harmonic sum is to break it into  $\log_2 n$  blocks with  $2^k$  terms in the  $k^{th}$  block and then to prove that the total for each block is between  $\frac{1}{2}$  and 1.

$$\sum_{i=1}^n \frac{1}{i} = \underbrace{\frac{1}{1}}_{\leq 1 \cdot 1 = 1} + \underbrace{\frac{1}{2} + \frac{1}{3}}_{\leq 2 \cdot \frac{1}{2} = 1} + \underbrace{\frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7}}_{\leq 4 \cdot \frac{1}{4} = 1} + \underbrace{\frac{1}{8} + \dots + \frac{1}{15}}_{\leq 8 \cdot \frac{1}{8} = 1} + \dots$$

$\overset{\geq 1 \cdot \frac{1}{2} = \frac{1}{2}}{\overbrace{}}$      $\overset{\geq 2 \cdot \frac{1}{4} = \frac{1}{2}}{\overbrace{}}$      $\overset{\geq 4 \cdot \frac{1}{8} = \frac{1}{2}}{\overbrace{}}$      $\overset{\geq 8 \cdot \frac{1}{16} = \frac{1}{2}}{\overbrace{}}$

From this, it follows that  $\frac{1}{2} \cdot \log_2 n \leq \sum_{i=1}^n \frac{1}{i} \leq 1 \cdot \log_2 n$ .

**Close to Harmonic:** We will now prove the remaining two cases of the adding made easy technique. “Decreasing” simple analytical arithmetic function are essentially functions that decrease but decrease slower than the harmonic function  $f(n) = \frac{1}{n}$ . We will prove that for these  $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$ . We will also prove that if a function decreases faster than the harmonic function yet possibly does decrease exponentially, then  $\sum_{i=1}^n f(i) = \Theta(1)$ .

**Theorem:** Considering non-increasing simple analytical functions,

if  $f(n) = n^{\Theta(1)-1}$ , then  $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$  and

if  $f(n) \leq n^{-1-\Omega(1)}$ , then  $\sum_{i=1}^n f(i) = \Theta(1)$ .

(As an exercise, consider functions that are non-increasing for  $n \geq n_0$  for some arbitrary  $n_0$ .)

**Proof:** Because these functions are non-increasing, all  $n$  terms are at least  $f(n)$  and hence the total is at least  $n \cdot f(n)$ . It is also at least the first term  $f(1) = \Theta(1)$ . What remains, in both cases, is to prove that the total is at most these.

As we did with the harmonic sum, we break it into blocks where the  $k^{th}$  block has the  $2^k$  terms  $\sum_{i=2^k}^{2^{k+1}-1} f(i)$ . Because the terms are decreasing, the total for the block is at most  $F(k) = 2^k \cdot f(2^k)$ . The total overall is then at most  $\sum_{k=0}^{\log_2 n} F(k) = \sum_{k=0}^{\log_2 n} 2^k \cdot f(2^k)$ .

If  $f(n) \geq n^{\Omega(1)-1}$ , then  $F(k) = 2^k \cdot (2^k)^{\Omega(1)-1} \geq 2^{\Omega(k)}$ . Because the sum of the blocks is simple analytical and exponentially increasing, the total is dominated by the total of its last block, namely  $F(\log_2 n) = \frac{n}{2} \cdot f(\frac{n}{2}) \geq \sum_{i=\frac{n}{2}}^n f(i)$ . As already said, we will prove in Section 5.3 that if a function is simple analytical and  $f(n) = n^{\Theta(1)-1}$ , then  $f(\frac{n}{2}) = \Theta(f(n))$ . We can conclude that  $\sum_{i=1}^n f(i) \leq \sum_{k=0}^{\log_2 n} F(k) = \Theta(F(\log_2 n)) = \Theta(\frac{n}{2} \cdot f(\frac{n}{2})) = \Theta(n \cdot f(n))$ .

Similarly, if  $f(n) \leq n^{-1-\Omega(1)}$ , then  $F(k) = 2^k \cdot (2^k)^{-1-\Omega(1)} \leq 2^{-\Omega(k)}$ . Because the sum of the blocks is exponentially decreasing, the total is dominated by the total of its first block, namely  $F(1) = \sum_{i=1}^1 f(i) = f(1) = \Theta(1)$ .

**Taking the Limit:** Consider the sum  $\sum_{i=1}^n \frac{1}{i^{(1-\epsilon)}}$ . When  $\epsilon > 0$ , the sum is arithmetic giving a total of  $\Theta(nf(n)) = \Theta(n^\epsilon)$ . When  $\epsilon = 0$ , the sum is harmonic giving a total of  $\Theta(\log_e n)$ . When  $\epsilon < 0$ , the sum has a bounded tail giving a total of  $\Theta(1)$ . To see how these answers blend together, consider the limit  $\lim_{\epsilon \rightarrow 0} \sum_{i=1}^n \frac{1}{i^{(1-\epsilon)}} \approx \lim_{\epsilon \rightarrow 0} \frac{n^\epsilon - 1}{\epsilon} = \lim_{\epsilon \rightarrow 0} \frac{e^{\epsilon \log_e n} - 1}{\epsilon}$ . Recall that L’Hopital’s rule can be applied when both numerator and denominator have a limit of the zero. Taking the derivative with respect to  $\epsilon$  separately of both numerator and the denominator gives  $\lim_{\epsilon \rightarrow 0} \frac{(\log_e n)e^{\epsilon \log_e n}}{1} = \log_e n$ . This corresponds to the harmonic sum.

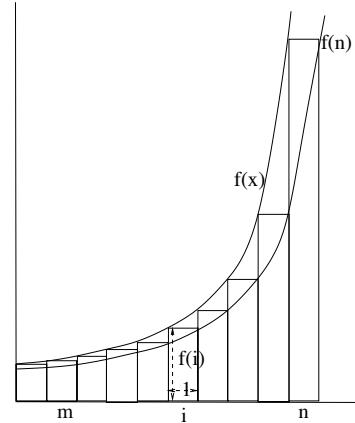
**Integrating:** As said, in the introduction, the sums of functions can be approximated by integration. Generally, it is safe to say that  $\sum_{i=1}^n f(i) = \Theta(\int_{x=1}^n f(x) \delta x)$ , though we can make this more formal.

**Theorem:**

- If  $f(i)$  is a monotonically increasing function, then  $\int_{x=m-1}^n f(x) \delta x \leq \sum_{i=m}^n f(i) \leq \int_{x=m}^{n+1} f(x) \delta x$ .

- If  $f(i)$  is a monotonically decreasing function, then  $\sum_{x=m}^{n+1} f(x) \delta x \leq \sum_{i=m}^n f(i) \leq \sum_{x=m-1}^n f(x) \delta x$ .

**Proof by Picture:** In the picture  $f(i)$  is the height and area of the  $i^{th}$  rectangle. Hence, sum  $\sum_{i=m}^n f(i)$  is the total area under these rectangles. Both curves plot the function  $f(x)$ . Yet one of the curves is shifted to the right by one. The areas under these curves are given by  $\sum_{x=m-1}^n f(x) \delta x$  and  $\sum_{x=m}^{n+1} f(x) \delta x$ . It is easy enough to see that the area under the top curve is slightly more than the discrete sum and that the area under the bottom curve is slightly less than this sum.



### Geometric:

$$\begin{array}{ll} \text{Sum} & \text{Integral} \\ \sum_{i=0}^n 2^i = 2 \cdot 2^n - 1 & \int_{x=0}^n 2^x \delta x = \frac{1}{\ln 2} 2^n \\ \sum_{i=0}^n b^i = \frac{1}{b-1} (b^{n+1} - 1) & \int_{x=0}^n b^x \delta x = \frac{1}{\ln b} b^n \end{array}$$

### Arithmetic:

$$\begin{array}{ll} \sum_{i=1}^n i = \frac{1}{2} n^2 + \frac{1}{2} n & \int_{x=0}^n x \delta x = \frac{1}{2} n^2 \\ \sum_{i=1}^n i^2 = \frac{1}{3} n^3 + \frac{1}{2} n^2 + \frac{1}{6} n & \int_{x=0}^n x^2 \delta x = \frac{1}{3} n^3 \\ \sum_{i=1}^n i^d = \frac{1}{d+1} n^{d+1} + \Theta(n^d) & \int_{x=0}^n x^d \delta x = \frac{1}{d+1} n^{d+1} \end{array}$$

### Harmonic:

$$\sum_{i=1}^n \frac{1}{i} \approx \int_{x=1}^{n+1} \frac{1}{x} \delta x = [\log_e x]_{x=1}^{n+1} = [\log_e n + 1] - [\log_e 1] = \log_e n + \Theta(1).$$

**Close to Harmonic:** Let  $\epsilon$  be some small positive constant (say 0.0001).

$$\begin{aligned} \sum_{i=1}^n \frac{1}{i^{(1+\epsilon)}} &\approx \int_{x=1}^{n+1} \frac{1}{x^{(1+\epsilon)}} \delta x = \int_{x=1}^{n+1} x^{-1-\epsilon} \delta x = \left[ \frac{1}{-\epsilon} x^{-\epsilon} \right]_{x=1}^{n+1} = \left[ -\frac{1}{\epsilon} (n+1)^{-\epsilon} \right] - \left[ -\frac{1}{\epsilon} 1^{-\epsilon} \right] \\ &= \Theta\left(\frac{1}{\epsilon}\right) = \Theta(1). \\ \sum_{i=1}^n \frac{1}{i^{(1-\epsilon)}} &\approx \int_{x=1}^{n+1} \frac{1}{x^{(1-\epsilon)}} \delta x = \int_{x=1}^{n+1} i^{-1+\epsilon} \delta x = \left[ \frac{1}{\epsilon} x^\epsilon \right]_{x=1}^{n+1} = \left[ \frac{1}{\epsilon} (n+1)^\epsilon \right] - \left[ \frac{1}{\epsilon} 1^\epsilon \right] = \Theta\left(\frac{1}{\epsilon} n^\epsilon\right) \\ &= \Theta(n^\epsilon). \end{aligned}$$

**Difficulty:** The problem with this method of evaluating a sum is that integrating can also be difficult. E.g.,  $\sum_{i=1}^n 3^i i^2 \log^3 i \approx \int_{x=0}^n 3^x x^2 \log^3 x \delta x$ . However, integrals can be approximated using the same techniques used here for sums.

**Exercise 5.2.3** (See solution in Section V) Give a function for which  $\sum_{i=1}^n f(i) \neq \Theta(\sum_{x=1}^n f(x) \delta x) \neq \Theta(\sum_{x=1}^{n+1} f(x) \delta x)$ .

**Exercise 5.2.4** A seeming paradox is how one could have a vessel that has finite volume and infinite surface area. This (theoretical) vessel could be filled with a small amount of paint but require an infinite amount of paint to paint. For  $h \in [1, \infty)$ , its cross section at  $h$  units from its top is a circle with radius  $r = \frac{1}{h^c}$  for some constant  $c$ . Integrate (or add up) its cross sectional circumference to compute its surface area and integrate (or add up) its cross sectional area to compute its volume. Give a value for  $c$  such that its surface area is infinite and its volume is finite. To help you explain why the paint paradox is not really a paradox, approximate the volume of paint required to cover the surface.

### 5.3 Simple Analytical Functions

The Adding Made Easy Technique requires that the function being summed is a simple analytical function. This section does two things. First, it proves that this requirement is necessary by providing functions that are not simple analytical and fail to behave as is needed for our Adding Made Easy Technique. Second, in order for the Adding Made Easy Technique to predict the total  $\sum_{i=1}^n f(i)$  from the value of the last term  $f(n)$ , it requires that the term  $f(n)$  is related in a predictable way to both the previous term  $f(n-1)$  and the middle term  $f(\frac{n}{2})$ . This section proves that if the function is a simple analytical, then these requirements are met. This section may be a little hard for this level of course, but are included for completion of the topic and because I, the author, worked hard to proving them.

**Definition:** Recall that a function is said to be *simple analytical* if it can be expressed with  $n$ , real constants, plus, minus, times, divide, exponentiation, and logarithms. Oscillating functions like sine, cosine, and floors are not allowed.

**Geometric Counterexample:** The functions  $f_{stair}(n) = 2^{2^{\lceil \log_2 n \rceil}}$  and  $f_{cosine}(n) = 2^{\lfloor \frac{1}{2} \cos(\pi \log_2 n) + 1.5 \rfloor \cdot n}$  given in Figure 5.3 are both counterexamples to the statement “If  $f(n) \geq 2^{\Omega(n)}$ , then  $\sum_{i=1}^n f(i) = \Theta(f(n))$ .”

**$f(n) = 2^{\Theta(n)}$ :** We see as follows that  $f_{stair}(n)$  is squeezed between the functions  $2^n$  and  $2^{2n}$ .

Let  $n = 2^k$ . Then  $\lceil \log_2 n \rceil = k$  and  $f_{stair}(n) = 2^{2^k} = 2^n$ , but for  $n' = n+1$ ,  $\lceil \log n' \rceil = k+1$  and  $f_{stair}(n') = 2^{2^{k+1}} = 2^{2n}$ . Similarly,  $f_{cosine}(n)$  is squeezed between  $2^n$  and  $2^{2n}$ . Being bounded in this way, both functions fit the formal definition of  $f(n) = 2^{\Theta(n)}$ , namely that  $\exists c_1, c_2, n_0, \forall n \geq n_0, 2^{c_1 n} \leq f(n) \leq 2^{c_2 n}$ , with  $c_1 = 1$  and  $c_2 = 2$ .

**$\sum_{i=1}^n f(i) \neq \Theta(f(n))$ :** With  $n = 2^k$ , both functions are more or less constant from  $f(\frac{n}{2})$  to  $f(n)$ . Because they behave like arithmetic functions within this range, the Adding Made Easy Techniques give that  $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$  and not  $\Theta(f(n))$ . This can be seen because the total  $\sum_{i=1}^n f(i)$  is at least the total within this range, which is approximately  $(n - \frac{n}{2})f(n) = \frac{1}{2}nf(n) \neq O(f(n))$ .

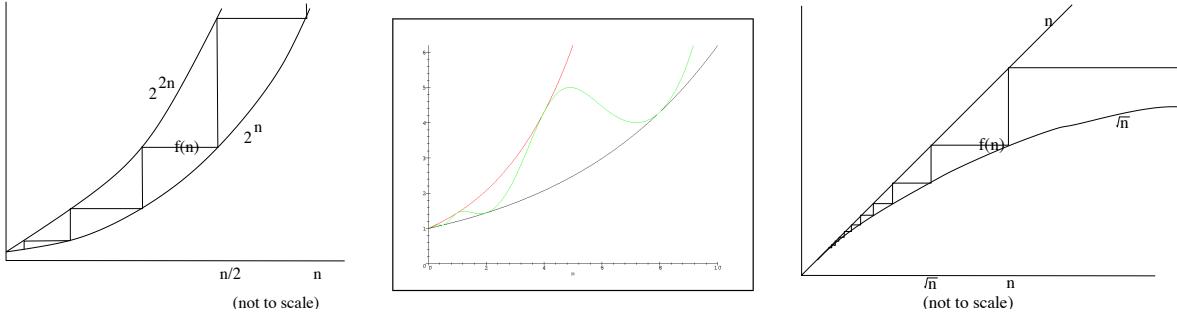


Figure 5.3: The geometric counterexamples  $f_{stair}(n) = 2^{2^{\lceil \log_2 n \rceil}}$  and  $f_{cosine}(n) = 2^{\lfloor \frac{1}{2} \cos(\pi \log_2 n) + 1.5 \rfloor \cdot n}$  and the arithmetic counterexample  $f(n) = 2^{2^{\lceil \log_2 n \rceil}}$

**Arithmetic Counterexample:** The function  $f(n) = 2^{2^{\lceil \log_2 n \rceil}}$  given in Figure 5.3 is a counterexample to the statement “If  $f(n) \in n^{\Theta(1)-1}$ , then  $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$ .”

**$f(n) \in n^{\Theta(1)-1}$ :** We see as follows that  $f(n)$  is bounded by  $\sqrt{n} \leq f(n) \leq n$ . Let  $n = 2^{2^k}$ .

Then  $\lfloor \log \log n \rfloor = k$  and  $f(n) = 2^{2^k} = n$ , but for  $n' = n - 1$ ,  $\lfloor \log \log n' \rfloor = k - 1$  and  $f(n') = 2^{2^{k-1}} = \sqrt{n}$ . Being bounded by  $\sqrt{n} \leq f(n) \leq n$ ,  $f(n) = n^{\Theta(1)-1}$ .

**$\sum_{i=1}^n f(i) \notin \Theta(n \cdot f(n))$ :** Again let  $n = 2^{2^k}$  so that  $f(n) = n$ , yet every previous term is most  $\sqrt{n}$ . The total of  $\sum_{i=1}^n f(i)$  then is at most  $(n-1) \cdot \sqrt{n} + n$ . Because the last term  $f(n)$  is so much bigger than the previous, the total is not  $\Theta(n \cdot f(n))$ , which is  $\Theta(n^2)$ .

**Predictable Terms for Geometric:** We will now prove that if  $f(n)$  is simple analytical function, then as required the term  $f(n)$  is related in a predictable way to the previous term  $f(n-1)$ .

**Theorem:** If  $f(n)$  is simple analytical and  $f(n) \geq 2^{\Omega(n)}$ , then  $\exists b > 1$ ,  $\exists n_0$ ,  $\forall n \geq n_0$ ,  $\frac{f(n+1)}{f(n)} \geq b$ .

Similarly, if  $f(n) \leq \frac{1}{2^{\Omega(n)}}$ , then  $\exists b \in (0, 1)$ ,  $\exists n_0$ ,  $\forall n \geq n_0$ ,  $\frac{f(n+1)}{f(n)} \leq b$ .

**Proof:** We will start by seeing what we learn from  $f(n) \geq 2^{\Omega(n)}$  and then what we learn from  $f(n)$  being simple analytical and then from these prove the result.

**Effect of Exponential:** Because the function is growing exponentially, we know that generally it grows at least as fast as  $2^{cn}$  for some constant  $c > 0$  or equivalently as fast as  $b^n$  for some constant  $b > 1$ . This does not mean that for every sufficiently large  $n$ , consecutive terms differ by a factor of at least  $b$ , but it does mean that this happens for an infinite number of consecutive  $n$ . The standard way of saying that something happens infinitely often is to say that for every  $n_0$  there is another  $n$  after this where it occurs.

**Exercise 5.3.1** Why does  $[\forall n_0, \exists n > n_0, P(n)]$  prove that there are an infinite number of values  $n$  for which the property  $P(n)$  is true?

**Theorem:** If  $f(n) = 2^{\Omega(n)}$ , then  $\exists b > 1$ ,  $\forall m_0$ ,  $\exists m \geq m_0$ ,  $\frac{f(m+1)}{f(m)} \geq b$ .

**Proof:** The formal meaning of  $f(n) = 2^{\Omega(n)}$  is that  $\exists c > 0$ ,  $\exists n_0$ ,  $\forall n > n_0$ ,  $f(n) \geq 2^{cn}$ . Fix such constants  $c$  and  $n_0$ . Set  $b' = 2^c$  and  $b = \sqrt{b'}$  so that  $b' > b > 1$  and  $f(n) \geq 2^{cn} = (b')^n = b^{2n} >> b^n$ , which gives us a little slack.

Recall that to prove something, it is sufficient to prove the contrapositive. Namely instead of proving  $A \Rightarrow B$ , where  $A = [\forall n > n_0, f(n) \geq 2^{cn} = b^{2n}]$  and  $B = [\forall m_0, \exists m \geq m_0, \frac{f(m+1)}{f(m)} \geq b]$ , we will prove that  $\neg B \Rightarrow \neg A$ , where  $\neg B = [\exists m_0, \forall m \geq m_0, \frac{f(m+1)}{f(m)} < b]$  and  $\neg A = [\exists n > n_0, f(n) < b^{2n}]$ . Convince yourself that this is equivalent.

Assume  $\neg B$  is true, fix a constant  $m_0$  that its states exist, and let  $n = \max(n_0, \frac{\log(f(m_0))-m_0}{\log b})$ . From  $\neg B$ , we know that at the point  $m_0$ , the function takes on some value  $f(m_0)$ , but that after this it cannot increase by more than a factor of  $b$  each term. This gives us that for our fixed  $n$ ,  $f(n) < b^1 f(n-1) < b^2 f(n-2) < b^3 f(n-3) < \dots < b^{n-m_0} f(m_0) = \frac{f(m_0)}{b^{n-m_0}} \cdot b^{2n}$ . The value  $n$  was set to be big enough so that this bound is at most  $b^{2n}$ . This completes the proof that  $\neg B \Rightarrow \neg A = [\exists n > n_0, f(n) < b^{2n}]$ .

**Effect of Simple Analytical Functions:** The following is the very deep and very difficult theorem of analysis that we will use but not prove about simple analytical functions.

**Theorem:** If  $h(n)$  is simple analytical and is not the identically zero function, then it has at most a finite number of places where its sign changes values, i.e. roots  $n$  where  $h(n) = 0$  or discontinuous places where  $h(n) = \infty$ , i.e., there are not an infinite number of such places.

**Examples:** A polynomial of degree  $d$  like  $h(n) = 5 - 2n + n^2 + \dots + 5n^d$  have at most  $d$  roots  $n$  where  $h(n) = 0$ . The function  $h(n) = 2^n - 100$  has one.  $h(n) = \frac{1}{n-5}$  has one place  $n = 5$  at which  $h(n) = \infty$ .

**Corollary:** If  $h(n)$  is simple analytical and is not the identically zero function, then  $\exists n_0, [\exists m \geq n_0, h(m) > 0] \Rightarrow [\forall n \geq n_0, h(n) > 0]$ .

**Proof of Corollary:** If  $h$  has a finite number of places where the sign changes, than it must have a last such place. Let  $n_0$  be beyond this place. After this place, the sign of  $h(n)$  either remains positive or remains negative. Of course, to determine whether it is remaining positive or negative thereafter, it is sufficient to check whether there is anyone place  $m$  thereafter in which it is positive or negative. The corollary follows.

**Corollary:** A similar statement is the following. If  $f(n)$  is simple analytical, then either there exists an  $n_0$  after which  $f(n)$  is non-decreasing or there exists an  $n_0$  after which  $f(n)$  is non-increasing.

**Proof:** If  $f(n)$  is simple analytical, then  $h(n) = \frac{\delta f(n)}{\delta n}$  is well defined and is also simple analytical. If  $h(n)$  is the zero function, then  $f(n)$  is constant. Otherwise,  $h(n)$  has a finite number of roots, hence  $f$  has a finite number of places where it switches from increasing to decreasing, and hence there exists an  $n_0$  after which  $f(n)$  is either only increasing or only decreasing.

**Concluding the Proof:** We will now complete the proof that if  $f(n)$  is simple analytical and  $f(n) \geq 2^{\Omega(n)}$ , then  $\exists b > 1, \exists n_0, \forall n \geq n_0, \frac{f(n+1)}{f(n)} \geq b$ . Because  $f$  grows exponentially, we learned in the theorem above that  $\exists b > 1, \forall m_0, \exists m \geq m_0, \frac{f(n+1)}{f(n)} \geq b$ . Fix such a constant  $b$ . The statement  $\frac{f(n+1)}{f(n)} \geq b$  is equivalent to  $h(n) \geq 0$ , where we define  $h(n) = \log(f(n+1)) - \log(f(n)) - \log b$ . If  $f(n)$  is simple analytical, then so is  $h(n)$ . Hence, we learned in the corollary above that  $\exists n_0, [\exists m \geq n_0, h(m) > 0] \Rightarrow [\forall n \geq n_0, h(n) > 0]$ . Fix such a constant  $n_0$  and let  $m_0$  be fixed to this  $n_0$ . With these settings, the growing exponentially fact gives that  $[\exists m \geq n_0, \frac{f(n+1)}{f(n)} \geq b]$ . Combining this with the simple analytical fact that  $[\exists m \geq n_0, \frac{f(n+1)}{f(n)} \geq b] \Rightarrow [\forall n \geq n_0, \frac{f(n+1)}{f(n)} \geq b]$  gives us that  $[\forall n \geq n_0, \frac{f(n+1)}{f(n)} \geq b]$ . This complete the proof. The proof of  $f(n) \leq \frac{1}{2^{\Omega(n)}}$  giving  $\frac{f(n+1)}{f(n)} \leq b$  is similar.

**Predictable Terms for Arithmetic:** We will now prove that if  $f(n)$  is simple analytical function, then as required the term  $f(n)$  is related in a predictable way to the middle term  $f(\frac{n}{2})$ .

**Theorem:** If  $f(n)$  is simple analytical and  $f(n) = n^{\Theta(1)-1}$ , then  $f(\frac{n}{2}) = \Theta(f(n))$ .

**Proof:** Consider a function  $f(n)$ . Define  $h(m) = (2^m) \cdot f(2^m)$  and  $n = 2^m$ . Because  $f(n)$  is simple analytical, so is  $h(m)$ . Because  $f(n) = n^{\Theta(1)-1}$ , we have that  $h(m) = (2^m) \cdot (2^m)^{\Theta(1)-1} = 2^{\Theta(m)}$ . Hence, by the second part of the geometric theorem,  $\exists b \in (0, 1), \exists m_0, \forall m > m_0, \frac{h(m)}{h(m-1)} \leq b$ . Solving gives that  $f(n) = f(2^m) = \frac{h(m)}{2^m} \leq b \cdot \frac{h(m-1)}{2^{m-1}} = \frac{b}{2} \cdot \frac{h(m-1)}{2^{m-1}} = \frac{b}{2} \cdot f(2^{m-1}) = \frac{b}{2} \cdot f(\frac{n}{2})$ . Hence,  $f(\frac{n}{2}) \geq \Omega(f(n))$ . Proving  $f(\frac{n}{2}) \leq \mathcal{O}(f(n))$  is similar.

This concludes what we have to say about simple analytical functions and about the Adding Made Easy Technique.

# Chapter 6

## Recurrence Relations

“Very well, sire,” the wise man said at last, “I asked only this: Tomorrow, for the first square of your chessboard, give me one grain of rice; the next day, for the second square, two grains of rice; the next day after that, four grains of rice; then, the following day, eight grains for the next square of your chessboard. Thus for each square give me twice the number of grains of the square before it, and so on for every square of the chessboard.”

“Now the King wondered, as anyone would, just how many grains of rice this would be.”

... thirty-two days later ...

“This must stop,” said the King to the wise man. “There is not enough rice in all of India to reward you.”

“No, indeed sire,” said the wise man. “There is not enough rice in all the world.”

The King’s Chessboard by David Birch

The number of grains of rice on square  $n$  in the story above is given by  $T(1) = 1$  and  $T(n) = 2T(n - 1)$ . This is called a recurrence relation. Such relations arise often in the study of computer algorithms. The most common place that they arise is in computing the running time of a recursive program. A recursive program is a routine or algorithm that calls itself on a smaller input instance. See Chapter 14. Similarly, a recurrence relation is a function that is defined in terms of itself on a smaller input instance.

**An Equation Involving an Unknown:** A recurrence relation is an equation (or a set of equations) involving an unknown variable. Solving it involves finding a “value” for the variable that satisfies the equations. It is similar to algebra.

**Algebra with Reals as Values:**  $x^2 = x + 2$  is an algebraic equation where the variable  $x$  is assumed to take on a real value. One way to solve it is to guess a solution and to check to see if it works. Here  $x = 2$  works, i.e.,  $(2)^2 = 4 = (2) + 2$ . However,  $x = -1$  also works,  $(-1)^2 = 1 = (-1) + 2$ . Making the further requirement that  $x \geq 0$  narrows the solution set to only  $x = 2$ .

**Differential Equations with Functions as Values:**  $\frac{\delta f(x)}{\delta x} = f(x)$  is a differential equation where the variable  $f$  is assumed to take on a function from reals to reals. One way to solve it is to guess a solution and to check to see if it works. Here  $f(x) = e^x$  works,

i.e.,  $\frac{\delta e^x}{\delta x} = e^x$ . However,  $f(x) = c \cdot e^x$  also works for each value of  $c$ . Making the further requirement that  $f(0) = 5$  narrows the solution set to only  $f(x) = 5 \cdot e^x$ .

**Recurrence Relations with Discrete Functions as Values:**  $T(n) = 2 \times T(n - 1)$  is a recurrence relation where the variable  $T$  is assumed to take on a function from integers to reals. One way to solve it is to guess a solution and to check to see if it works. Here  $T(n) = 2^n$  works, i.e.,  $2^n = 2 \times 2^{n-1}$ . However,  $T(n) = c \cdot 2^n$  also works for each value of  $c$ . Making the further requirement that  $T(1) = 1$  narrows the solution set to only  $T(n) = \frac{1}{2} \cdot 2^n = 2^{n-1}$ .

## 6.1 Relating Recurrence Relations to the Timing of Recursive Programs

One of the most common places for recurrence relations to arise is when analyzing the time complexity of recursive algorithms.

**Basic Code:** Suppose a recursive program, when given an instances of size  $n$ , recurses  $a$  times on subinstances of size  $\frac{n}{b}$ .

```
algorithm Eg(In)
⟨pre-cond⟩: In is an instance of size n.
⟨post-cond⟩: Prints T(n) “Hi”s.
```

```
begin
    n = |In|
    if( n ≤ 1) then
        put “Hi”
    else
        loop i = 1..f(n)
            put “Hi”
        end loop
        loop i = 1..a
            In/b = an input of size n/b
            Eg(In/b)
        end loop
    end if
end algorithm
```

**Stack Frames:** A single execution of the routine on a single instance ignoring subroutine calls is referred to as a *stack frame*. See Section 14.5. The top stack frame recurses  $b$  times creating  $b$  more stack frames. Each of these recurse  $b$  times for a total of  $b \cdot b$  stack frames at this third level. This continues until a base case is reached.

**Time for Base Cases:** Recursive programs must stop recursing when the input becomes sufficiently small. In this example, only one “Hi” is printed when the input has size zero or one. In general, we will assume that recursive programs spend  $\Theta(1)$  time for instances of size  $\Theta(1)$ . We express this as  $T(1) = 1$  or more generally as  $T(\Theta(1)) = \Theta(1)$ .

**Running Time:** Let  $T(n)$  denote the total computation time for instances of size  $n$ . This time can be determined in a number of different ways.

### Top Level plus Recursion:

**Top Level:** Before recursing, the top level stack frame prints “Hi”  $f(n)$  times. In general, we use  $f(n)$  to denote the computation time required by a stack frame on an instance of size  $n$ . This time includes the time needed to generate the subinstances and recombining their solutions into the solution for its instance, but excludes the time needed to recurse.

**Recursion:** The top stack frame, recurses  $a$  times on subinstances of size  $\frac{n}{b}$ . If  $T(n)$  is the total computation time for instances of size  $n$ , then it follows that  $T(\frac{n}{b})$  is the total computation time for instances of size  $\frac{n}{b}$ . Repeating this  $a$  times will take time  $a \cdot T(\frac{n}{b})$ .

**Total:** We have already said that the total computation time is  $T(n)$ . Now, however, we have also concluded that the total time is  $f(n)$  for the top level and  $T(\frac{n}{b})$  for each of the recursive calls, for a total of  $a \cdot T(\frac{n}{b}) + f(n)$ .

**The Recurrence Relation:** It follows that  $T(n)$  satisfies the recursive relation

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n).$$

The goal of this section is to determine which function  $T(n)$  satisfies this relation.

**Sum of Work at each Level of Recursion:** Another way of computing the total computation time  $T(n)$  is to sum up the time spent by each stack frame. These stack frames form a tree based on who calls who. The stack frames at the same *level* of this tree often have input instances that are of roughly the same size, and hence have roughly equivalent running times. Multiplying this time by the number of stack frames at the level gives the total time spent at this level. Adding up this time for all the levels is another way of computing the total time  $T(n)$ . This will be our primary way of evaluating recurrence relations.

**Instances of Size  $n-b$ :** Suppose instead of recursing  $a$  times on instances of size  $\frac{n}{b}$ , the routine recurses  $a$  times on instances of size  $n-b$ , for some constant  $b$ . The same discussion as above will apply, but the related recurrence relation will be

$$T(n) = a \cdot T(n-b) + f(n).$$

## 6.2 Summary

This subsection summarizes what will be discussed a length within in this section.

**The Basic Form:** Most recurrence relations that we will consider will have the form  $T(n) = a \cdot T(\frac{n}{b}) + f(n)$ , where  $a$  and  $b$  are some real positive constants and  $f(n)$  is a function. The second most common form is  $T(n) = a \cdot T(n-b) + f(n)$ . Section 6.3 will focus on the first of these forms. Section 6.4, we will apply the same idea to the second of these forms and to other recurrence relations.

**Base Cases:** Throughout, we will assume that  $T$  with a constant input has a constant output. We write this as  $T(\Theta(1)) = \Theta(1)$ .

**Examples:** The following examples include some of the classic recurrence relations as well as some other examples designed to demonstrate the various patterns that the results fall into.

**Subinstance of Size  $\frac{n}{2}$ :**

Dominated By	One Subinstance	Two Subinstances
Top Level	$T(n) = T(n/2) + n = \Theta(n)$	$T(n) = 2 * T(n/2) + n^2 = \Theta(n^2)$
Levels Arithmetic	$T(n) = T(n/2) + 1 = \Theta(\log n)$	$T(n) = 2 * T(n/2) + n = \Theta(n \log n)$
Base Cases	$T(n) = T(n/2) + 0 = \Theta(1)$	$T(n) = 2 * T(n/2) + 1 = \Theta(n)$

**Different Sized Subinstances:**

Dominated By	Three Subinstances
Top Level	$T(n) = T(\frac{1}{7}n) + T(\frac{2}{7}n) + T(\frac{3}{7}n) + n = \Theta(n)$
Levels Arithmetic	$T(n) = T(\frac{1}{7}n) + T(\frac{2}{7}n) + T(\frac{4}{7}n) + n = \Theta(n \log n)$
Base Cases	$T(n) = T(\frac{1}{7}n) + T(\frac{3}{7}n) + T(\frac{4}{7}n) + n = \Theta(n^?)$

**Subinstance of Size  $n - 1$ :**

Dominated By	One Subinstance	Two Subinstances
Top Level	$T(n) = T(n - 1) + 2^n = \Theta(2^n)$	$T(n) = 2 * T(n - 1) + 3^n = \Theta(3^n)$
Levels Arithmetic	$T(n) = T(n - 1) + n = \Theta(n^2)$	$T(n) = 2 * T(n - 1) + 2^n = \Theta(n2^n)$
Base Cases	$T(n) = T(n - 1) + 0 = \Theta(1)$	$T(n) = 2 * T(n - 1) + n = \Theta(2^n)$

**A Growing Number of Subinstances of a Shrinking Size:** Consider recurrence relations of the form  $T(n) = a \cdot T(\frac{n}{b}) + f(n)$  with  $T(\Theta(1)) = \Theta(1)$ . Each instance having  $a$  subinstances means that the number of subinstances grows exponentially by a factor of  $b$ . On the other hand, the size of the subinstances shrink exponentially. The amount of work that the instance must do is the function  $f$  of this instance size. Whether the growing or the shrinking dominates this process depends upon the relationship between  $a$ ,  $b$ , and  $f(n)$ .

**Four Different Solutions:** All recurrence relations that we will consider have one of the following four different types of solutions. These correspond to the four types of solutions for sums, see Section 5.

**Dominated by Top:** The *top level* of the recursion requires  $f(n)$  work. If this is sufficiently big then this time will dominate the total and the answer will be  $T(n) = \Theta(f(n))$ .

**Dominated by Base Cases:** We will see that the number of *base cases* is  $n^{\frac{\log a}{\log b}}$  when  $T(n) = a \cdot T(\frac{n}{b}) + f(n)$ . Each of these requires  $T(\Theta(1)) = \Theta(1)$  time. If this is sufficiently big then this time will dominate the total and the answer will be  $T(n) = \Theta\left(n^{\frac{\log a}{\log b}}\right)$ .

**Levels Arithmetic:** If the amount of work at the different levels of recursion are sufficiently close to each other, then the total is the number of levels times this amount of work. In this case, there are  $\Theta(\log n)$  levels, giving that  $T(n) = \Theta(\log n) \times \Theta(f(n))$ .

**Levels Harmonic:** A strange example, included only for completion, is when the work at each of the levels forms a harmonic sum. In this case,  $T(n) = \Theta(f(n) \log n \log \log n) = \Theta(n^{\frac{\log a}{\log b}} \log \log n)$ .

If  $T(n) = a \cdot T(n-b) + f(n)$ , then the types of solutions are the same, except that the number of base cases will be  $a^{\frac{n}{b}}$  and the number of levels is  $\frac{n}{b}$  giving the solutions  $T(n) = \Theta(f(n))$ ,  $T(n) = \Theta(a^{\frac{n}{b}})$ , and  $T(n) = \Theta(n \cdot f(n))$ .

**The Ratio  $\frac{\log a}{\log b}$ :** See Section 2 for a discussion about logarithms. One trick that it gives us is that when computing the ratio between two logarithms, the base used does not matter because changing the base will introduce the same constant both on the top and the bottom, which will cancel. Hence, when computing such a ratio, you can choose whichever base makes the calculation the easiest. For example, to compute  $\frac{\log 16}{\log 8}$ , the obvious base to use is 2, because  $\frac{\log_2 16}{\log_2 8} = \frac{4}{3}$ . This is useful in giving that  $T(n) = 16 \cdot T\left(\frac{n}{8}\right) + f(n) = \Theta\left(n^{\frac{\log 16}{\log 8}}\right) = \Theta\left(n^{4/3}\right)$ . On the other hand, to compute  $\frac{\log 9}{\log 27}$ , the obvious base to use is 3, because  $\frac{\log_3 9}{\log_3 27} = \frac{2}{3}$  and hence  $T(n) = 9 \cdot T\left(\frac{n}{27}\right) + f(n) = \Theta\left(n^{2/3}\right)$ . Another interesting fact given is that  $\log 1 = 0$ , which gives that  $T(n) = 1 \cdot T\left(\frac{n}{2}\right) + f(n)$ ,  $T(n) = \Theta\left(n^{\frac{\log 1}{\log 2}}\right) = \Theta\left(n^0\right) = \Theta(1)$ .

**Rules for Functions with Basic Form  $T(n) = a \cdot T\left(\frac{n}{b}\right) + \Theta(n^c \cdot \log^d n)$ :** To simplify things, let us assume that  $f(n)$  has the basic form  $\Theta(n^c \cdot \log^d n)$ , where  $c$  and  $d$  are some real positive constants. This recurrence relation can be quickly approximated using the following simple rules.

**algorithm** *BasicRecurrenceRelation(a, b, c, d)*

***⟨pre-cond⟩:*** We are given a recurrence relation with the basic form  $T(n) = a \cdot T\left(\frac{n}{b}\right) + \Theta(n^c \cdot \log^d n)$ . We assume  $a \geq 1$  and  $b > 1$ .

***⟨post-cond⟩:*** Computes the approximation  $\Theta(T(n))$ .

begin

if(  $c > \frac{\log a}{\log b}$  ) then

The recurrence is *dominated by the top level* and  $T(n) = \Theta(f(n)) = \Theta(n^c \cdot \log^d n)$ .

else if(  $c = \frac{\log a}{\log b}$  ) then

if(  $d > -1$  ) then

Then the *levels are arithmetic* and  $T(n) = \Theta(f(n) \log n) = \Theta(n^c \cdot \log^{d+1} n)$ .

else if(  $d = -1$  ) then

Then the *levels are harmonic* and  $T(n) = \Theta(n^c \cdot \log \log n)$ .

else if(  $d < -1$  ) then

The recurrence is *dominated by the base cases* and  $T(n) = \Theta(n^{\frac{\log a}{\log b}})$ .

end if

else if(  $c < \frac{\log a}{\log b}$  ) then

The recurrence is *dominated by the base cases* and  $T(n) = \Theta(n^{\frac{\log a}{\log b}})$ .

end if

end algorithm

### Examples:

- $T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^3)$ : Here  $a = 4$ ,  $b = 2$ ,  $c = 3$ , and  $d = 0$ . Applying the technique, we compare  $c = 3$  to  $\frac{\log a}{\log b} = \frac{\log_2 4}{\log_2 2} = 2$ . Because it is bigger, we know that  $T(n)$  is dominated by the top level and  $T(n) = \Theta(f(n)) = \Theta(n^3)$ .
- $T(n) = 4T\left(\frac{n}{2}\right) + \Theta\left(\frac{n^3}{\log^3 n}\right)$ : This example is the same except for  $d = -3$ . Decreasing the time for the top by this little amount does not change the fact that it dominates.  $T(n) = \Theta(f(n)) = \Theta\left(\frac{n^3}{\log^3 n}\right)$ .

- $T(n) = 27T(\frac{n}{3}) + \Theta(n^3 \log^4 n)$ : Because  $\frac{\log a}{\log b} = \frac{\log_3 27}{\log_3 3} = \frac{3}{1} = c$  and  $d = 4 > -1$ , all levels take roughly the same computation time and  $T(n) = \Theta(f(n) \log(n)) = \Theta(n^3 \log^5 n)$ .
- $T(n) = 4T(\frac{n}{2}) + \Theta(\frac{n^2}{\log n})$ : Because  $\frac{\log a}{\log b} = \frac{\log 4}{\log 2} = 2 = c$  and  $d = -1$ , this is the harmonic case and  $T(n) = \Theta(n^c \cdot \log \log n) = \Theta(n^2 \log \log n)$ .
- $T(n) = 4T(\frac{n}{2}) + \Theta(n)$ : Because  $\frac{\log a}{\log b} = \frac{\log 4}{\log 2} = 2 > 1 = c$ , the computation time is sufficiently dominated by the sum of the base cases and  $T(n) = \Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^2)$ .

**More General Functions using  $\Theta$  Notation:** The above rules for approximating  $T(n) = a \cdot T(\frac{n}{b}) + f(n)$  can be generalized even further to include low order terms and an even wider range of functions  $f(n)$ .

**Low Order Terms:** Sometimes the subinstances formed are not exactly of size  $\frac{n}{b}$ , but have a stranger size like  $\frac{n}{b} - \sqrt{n} + \log n - 5$ . Such low order terms are not significant to the approximation of the total and can simply be ignored. We can denote the fact that such terms are included by considering recurrence relations of the form  $T(n) = a \cdot T(\frac{n}{b} + o(n)) + f(n)$ . The key is that the difference between the size and  $\frac{n}{b}$  is smaller than  $\epsilon n$  for every constant  $\epsilon > 0$ . We will not be covering recurrence relations of this form.

**Simple Analytical Function  $f(n)$ :** The only requirement on  $f$  is that it is a *simple analytical* function, (See Section 5).

The first step in evaluating this recurrence relation is to determine the computation times  $\Theta(f(n))$  and  $\Theta(n^{\frac{\log a}{\log b}})$  for the top and the bottom levels of recursion. The second step is to take their ratio in order to determine whether one is “sufficiently” bigger than the other.

**Dominated by Top:** If  $\frac{f(n)}{n^{\frac{\log a}{\log b}}} \geq n^{\Omega(1)}$ , then  $T(n) = \Theta(f(n))$ .

**Levels Arithmetic:** If  $\frac{f(n)}{n^{\frac{\log a}{\log b}}} = [\log n]^{\Theta(1)-1} \approx \log^d n$ , for  $d > -1$ , then  $T(n) = \Theta(f(n) \log n)$ .

**Levels Harmonic:** If  $\frac{f(n)}{n^{\frac{\log a}{\log b}}} = \Theta(\frac{1}{\log n})$ , then  $T(n) = \Theta(n^{\frac{\log a}{\log b}} \log \log n)$ .

**Dominated by Base Cases:** If  $\frac{f(n)}{n^{\frac{\log a}{\log b}}} \leq [\log n]^{-1-\Omega(1)} \approx \log^d n$ , for  $d < -1$ , then  $T(n) = \Theta(n^{\frac{\log a}{\log b}})$ .

Recall that  $\Omega(1)$  includes any increasing function and any constant  $c$  strictly bigger than zero.

### Examples:

- $T(n) = 4T(\frac{n}{2}) + \Theta(n^3 \log \log n)$ : The times for the top and the bottom level of recursion are  $f(n) = \Theta(n^3 \log \log n)$  and  $\Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^{\frac{\log 4}{\log 2}}) = \Theta(n^2)$ . The top level sufficiently dominates the bottom level because  $f(n)/n^{\frac{\log a}{\log b}} = \frac{n^3 \log \log n}{n^2} = n^1 \log \log n \geq n^{\Omega(1)}$ . Hence, we can conclude that  $T(n) = \Theta(f(n)) = \Theta(n^3 \log \log n)$ .
- $T(n) = 4T(\frac{n}{2}) + \Theta(2^n)$ : The time for the top has increased to  $f(n) = \Theta(2^n)$ . Being even bigger, the top level dominates the computation even more. This can be seen because  $f(n)/n^{\frac{\log a}{\log b}} = \frac{2^n}{n^2} \geq n^{\Omega(1)}$ . Hence,  $T(n) = \Theta(f(n)) = \Theta(2^n)$ .

- $T(n) = 4T(\frac{n}{2}) + \Theta(n^2 \log \log n)$ : The times for the top and the bottom level of recursion are now  $f(n) = \Theta(n^2 \log \log n)$  and  $\Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^2)$ . Note  $f(n)/n^{\frac{\log a}{\log b}} = \Theta(n^2 \log \log n/n^2) = \Theta(\log \log n) = [\log n]^{\Theta(1)-1}$ . (If  $\log \log n$  is to be compared to  $\log^d n$ , for some  $d$ , then the closest one is  $d = 0$ , which is greater than  $-1$ .) It follows that  $T(n) = \Theta(f(n) \log n) = \Theta(n^2 \log n \log \log n)$ .
- $T(n) = 4T(\frac{n}{2}) + \Theta(\log \log n)$ : The times for the top and the bottom level of recursion are  $f(n) = \Theta(\log \log n)$  and  $\Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^{\frac{\log 4}{\log 2}}) = \Theta(n^2)$ . The computation time is sufficiently dominated by the sum of the base cases because  $f(n)/n^{\frac{\log a}{\log b}} = \frac{\Theta(\log \log n)}{n^2} \leq [\log n]^{-\Theta(1)-1}$ . It follows that  $T(n) = \Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^2)$ .
- $T(n) = 4T(\frac{n}{2} - \sqrt{n} + \log n - 5) + \Theta(n^3)$ : This looks ugly. However, because  $\sqrt{n} - \log n + 5$  is  $o(n)$ , it does not play a significant role in the approximation. Therefore, the answer is the same as it would be for  $T(n) = 4T(\frac{n}{2}) + \Theta(n^3)$ .

This completes the summary of the results.

### 6.3 The Classic Techniques

We now present a few of the classic techniques for computing recurrence relations. We will continue to focus on recurrence relations of the form  $T(n) = a \cdot T(\frac{n}{b}) + f(n)$  with  $T(\Theta(1)) = \Theta(1)$ . Later in Section 6.4, we will apply the same idea to other recurrence relations.

**Guess and Verify:** Consider the example  $T(n) = 4T(\frac{n}{2}) + n$  and  $T(1) = 1$ . If we could guess that the solution is  $T(n) = 2n^2 - n$ , we could verify this answer in the following two ways.

**Plugging In:** The first way to verify that  $T(n) = 2n^2 - n$  is the solution is to simply plug it into the two equations  $T(n) = 4T(\frac{n}{2}) + n$  and  $T(1) = 1$  and make sure that they are satisfied.

Left Side	Right Side
$T(n) = 2n^2 - n$	$4T(\frac{n}{2}) + n = 4 \left[ 2 \left( \frac{n}{2} \right)^2 - \left( \frac{n}{2} \right) \right] - n = 2n^2 - n$
$T(1) = 2n^2 - n = 1$	1

**Proof by Induction:** Similarly, we can use induction to prove that this is the solution for all  $n$ , (at least for all powers of 2, namely  $n = 2^i$ ).

**Base Case:** Because  $T(1) = 2(1)^2 - 1 = 1$ , it is correct for  $n = 2^0$ .

**Induction Step:** Let  $n = 2^i$ . Assume that it is correct for  $2^{i-1} = \frac{n}{2}$ . Because  $T(n) = 4T(\frac{n}{2}) + n = 4 \left[ 2 \left( \frac{n}{2} \right)^2 - \left( \frac{n}{2} \right) \right] - n = 2n^2 - n$ , it is also true for  $n$ .

**Guess Form and Calculate Coefficients:** Suppose that instead of guessing that the solution to  $T(n) = 4T(\frac{n}{2}) + n$  and  $T(1) = 1$  is  $T(n) = 2n^2 - n$ , we are only able to guess that it has the form  $T(n) = an^2 + bn + c$  for some constants  $a$ ,  $b$ , and  $c$ . We can plug in this solution as done before and solve for the  $a$ ,  $b$ , and  $c$ .

Left Side	Right Side
$T(n) = an^2 + bn + c$	$4T(\frac{n}{2}) + n = 4 \left[ a \left( \frac{n}{2} \right)^2 + b \left( \frac{n}{2} \right) + c \right] - n = an^2 + (2b+1)n + 4c$
$T(1) = a + b + c$	1

These left and right sides must be equal for all  $n$ . Both have  $a$  as the coefficient of  $n^2$ , which is good. To make the coefficient in front  $n$  be the same, we need that  $b = 2b + 1$ , which gives

$b = -1$ . To make the constant coefficient be the same, we need that  $c = 4c$ , which gives  $c = 0$ . To make  $T(1) = a(1)^2 + b(1) + c = a(1)^2 - (1) + 0 = 1$ , we need that  $a = 2$ . This gives us the solution  $T(n) = 2n^2 - n$  that we had before.

**Guessing  $\Theta(T(n))$ :** Another option is to prove by induction that the solution is  $T(n) = \mathcal{O}(n^2)$ . Sometimes one can proceed as above, but I find that often one gets stuck.

**Induction Step:** Suppose that by induction we assumed that  $T\left(\frac{n}{2}\right) \leq c \cdot \left(\frac{n}{2}\right)^2$  for some constant  $c > 0$ . Then we would try to prove that  $T(n) \leq c \cdot n^2$  for the same constant  $c$ . We would proceed in the following manner.  $T(n) = 4T\left(\frac{n}{2}\right) + n \leq 4\left[c \cdot \left(\frac{n}{2}\right)^2\right] + n = c \cdot n^2 + n$ . However, no matter how big we make  $c$ , this is not smaller than  $c \cdot n^2$ . Thus, this proof failed.

**Guessing Whether Top or Bottom Dominates:** Our goal is to estimate  $T(n)$ . We have that  $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ . One might first ask whether the  $a \cdot T\left(\frac{n}{b}\right)$  or the  $f(n)$  is more significant. There are three possibilities. The first is that  $f(n)$  is much bigger than  $a \cdot T\left(\frac{n}{b}\right)$ . The second is that  $a \cdot T\left(\frac{n}{b}\right)$  is much bigger than  $f(n)$ . The third is that the terms are close enough in value that they both make a significant contribution. The third case will be harder to analyze. But let us see what we can do in the first two cases.

**$f(n)$  is much bigger than  $a \cdot T\left(\frac{n}{b}\right)$ :** It follows that  $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) = \Theta(f(n))$  and hence, we are done. The function  $f(n)$  is given and the answer is  $T(n) = \Theta(f(n))$ .

**$a \cdot T\left(\frac{n}{b}\right)$  is much bigger than  $f(n)$ :** It follows that  $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \approx a \cdot T\left(\frac{n}{b}\right)$ . This case is harder. We must still solve this equation. With some insight, let us guess that the general form is  $T(n) = n^\alpha$  for some constant  $\alpha$ . We will plug this guess into the equation  $T(n) = a \cdot T\left(\frac{n}{b}\right)$  and solve for  $\alpha$ . Plugging this in gives  $n^\alpha = a \cdot \left(\frac{n}{b}\right)^\alpha$ . Dividing out the  $n^\alpha$ , gives  $1 = a \cdot \left(\frac{1}{b}\right)^\alpha$ . Bringing over the  $b^\alpha$  gives  $b^\alpha = a$ . Taking the log gives  $\alpha \cdot \log b = \log a$  and solving gives  $\alpha = \frac{\log a}{\log b}$ . The conclusion is that the solution to  $T(n) = a \cdot T\left(\frac{n}{b}\right)$  is  $T(n) = \Theta(n^{\frac{\log a}{\log b}})$ .

This was by no means formal. But it does give some intuition that the answer may well be  $T(n) = \Theta(\max(f(n), n^{\frac{\log a}{\log b}}))$ . We will now try to be more precise.

**Unwinding  $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ :** Our first formal step will be to express  $T(n)$  as the sum of the work completed at each level of the recursion. Later we will evaluate this sum. One way to express  $T(n)$  as a sum is to unwind it as follows.

$$\begin{aligned} T(n) &= f(n) + a \cdot T\left(\frac{n}{b}\right) \\ &= f(n) + a \cdot \left[ f\left(\frac{n}{b}\right) + a \cdot T\left(\frac{n}{b^2}\right) \right] \\ &= f(n) + af\left(\frac{n}{b}\right) + a^2 \cdot T\left(\frac{n}{b^2}\right) \\ &= f(n) + af\left(\frac{n}{b}\right) + a^2 \cdot \left[ f\left(\frac{n}{b^2}\right) + a \cdot T\left(\frac{n}{b^3}\right) \right] \end{aligned}$$

$$\begin{aligned}
&= f(n) + af\left(\frac{n}{b}\right) + a^2f\left(\frac{n}{b^2}\right) + a^3 \cdot T\left(\frac{n}{b^3}\right) \\
&= \dots \\
&= \sum_{i=0}^{h-1} a^i \cdot f\left(\frac{n}{b^i}\right) + a^h \cdot T(1) = \Theta\left(\sum_{i=0}^h a^i \cdot f\left(\frac{n}{b^i}\right)\right).
\end{aligned}$$

**Levels of The Recursion Tree:** We can understand the unwinding of the recurrence relation into the above sum better when we examine the tree consisting of one node for every stack frame executed during the computation.

Recall that the recurrence relation  $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$  relates to the computation time of a recursive program, such that when it is given an instance of size  $n$ , it recurses  $a$  times on subinstances of size  $\frac{n}{b}$ . The function  $f(n)$  denotes the computation time required by a single stack frame on an instance of size  $n$  and  $T(n)$  denotes the entire computation time to recurse.

**Level 0:** The top level of the recursion has an instance of size  $n$ . As stated, excluding the time to recurse, this top stack frame requires  $f(n)$  computation time to generate the subinstances and recombining their solutions.

**Level 1:** The single top level stack frame recurses  $a$  times. Hence, the second level (level 1) has  $a$  stack frames. Each of them is given a subinstance of size  $\frac{n}{b}$  and hence requires  $f\left(\frac{n}{b}\right)$  computation time. It follows that the total computation time for this level is  $a \cdot f\left(\frac{n}{b}\right)$ .

**Level 2:** Each of the  $a$  stack frames at the level 1 recurses  $a$  times giving  $a^2$  stack frames at the level 2. Each stack frame at the level 1 has a subinstance of size  $\frac{n}{b}$ , hence they recurse on subsubinstances of size  $\frac{n}{b^2}$ . Therefore, each stack frame at the level 2 requires  $f\left(\frac{n}{b^2}\right)$  computation time. It follows that the total computation time for this level is  $a^2 \cdot f\left(\frac{n}{b^2}\right)$ .

**Level 3, etc. ...**

**Level i:** Each successive level, the number of stack frames goes up by a factor of  $a$  and the size of the subinstance goes down by a factor of  $b$ .

**Number of Stack Frames at this Level:** It follows that at the level  $i$ , we have  $a^i$  subinstances

**Size of Subinstances at this Level:** Each subinstance at this level has size  $\frac{n}{b^i}$ .

**Time for each Stack Frame at this Level:** Given this size, the time taken is  $f\left(\frac{n}{b^i}\right)$ .

**Total Time for Level:** The total time for this level is the number at the level times the time for each, namely  $a^i \cdot f\left(\frac{n}{b^i}\right)$ .

**Level h:** The recursive program stops recursing when it reaches a *base case*.

**Size of the Base Case:** A base case is an input instance that has some minimum size. Whether this size is zero, one, or two, will not change the approximation  $\Theta(T(n))$  of the total time. This is why we specify only that the size of the base case is  $\Theta(1)$ .

**The Number of Levels h:** Let  $h$  denote the depth of the recursion tree. If an instance at level  $i$  has size  $\frac{n}{b^i}$ , then at the level  $h$ , it will have size  $\frac{n}{b^h}$ . Now we have a choice as to which constant we would like to use for the base case. For convenience sake, let us use a constant that makes our life easy.

**Size One:** If we set the size of the base case to be one, then this gives us the equation  $\frac{n}{b^h} = 1$ , and the number of levels needed to achieve this is  $h = \frac{\log n}{\log b}$ .

**Size Two:** If instead, we set the base case size to be  $\frac{n}{b^h} = 2$ , then the number of levels needed is  $h = \frac{\log(n)-1}{\log b}$ . Though this is only a constant less, it is messier.

**Size Zero:** Surely recursing to instances of size zero will also be fine. The equation becomes  $\frac{n}{b^h} = 0$ . Solving this for  $h$  gives  $h = \infty$ . Practically, what does this mean? You may have heard of zeno's paradox. See Exercise 5.2.1. If you cut the remaining distance in half and then in half again and so on, then though you get very close very fast, you never actually get there.

**Time for a Base Case:** Changing the computation time for a base case will change the total running time by only a multiplicative constant. This is why we specify only that it requires some fixed time. We write this as  $T(\Theta(1)) = \Theta(1)$ .

**Number of Base Cases:** The number of stack frames at the level  $i$  is  $a^i$ . Hence, the number of these base case subinstances is  $a^h = a^{\frac{\log n}{\log b}}$ . This looks really ugly. Is it exponential in  $n$ , or polynomial?, or something else weird? The place to look for help is Section 2, which gives rules on logarithms and exponentials. One rule state that you can move things between the base and the exponent as long as you add or remove a log. This gives that the number of base cases is  $a^h = a^{\frac{\log n}{\log b}} = n^{\frac{\log a}{\log b}}$ . Given that  $\frac{\log a}{\log b}$  is simply some constant,  $n^{\frac{\log a}{\log b}}$  is a simple polynomial in  $n$ .

**Time for Base Cases:** This gives the total time for the base cases to be  $a^h \cdot T(1) = \Theta(n^{\frac{\log a}{\log b}})$ . This is the same time that we got before.

**Sum of Time for Each Level:** We obtain the total time  $T(n)$  for the recursion by summing up the time at each of these levels. This gives

$$T(n) = \left[ \sum_{i=0}^{h-1} a^i \cdot f\left(\frac{n}{b^i}\right) \right] + a^h \cdot T(1) = \Theta\left(\sum_{i=0}^h a^i \cdot f\left(\frac{n}{b^i}\right)\right).$$

The key things to remember about this sum are that it has  $\Theta(\log n)$  terms, the first term being  $\Theta(f(n))$ , and the last being  $\Theta(n^{\frac{\log a}{\log b}})$ .

**Some Examples:** When working through an example, the following questions are useful.

Dominated by	Base Cases	Levels Arithmetic	Top Level
Examples	$T(n) = 4T(n/2) + n$	$T(n) = 9T(n/3) + n^2$	$T(n) = 2T(n/4) + n^2$
a) # frames at the $i^{th}$ level?	$4^i$	$9^i$	$2^i$
b) Size of instance at the $i^{th}$ level?	$\frac{n}{2^i}$	$\frac{n}{3^i}$	$\frac{n}{4^i}$
c) Time within one stack frame	$f\left(\frac{n}{2^i}\right) = \left(\frac{n}{2^i}\right)$	$f\left(\frac{n}{3^i}\right) = \left(\frac{n}{3^i}\right)^2$	$f\left(\frac{n}{4^i}\right) = \left(\frac{n}{4^i}\right)^2$
d) # levels	$\frac{n}{2^h} = 1$ $h = \frac{\log n}{\log 2} = \Theta(\log n)$	$\frac{n}{3^h} = 1$ $h = \frac{\log n}{\log 3} = \Theta(\log n)$	$\frac{n}{4^h} = 1$ $h = \frac{\log n}{\log 4} = \Theta(\log n)$
e) # base case stack frames	$4^h = 4^{\frac{\log n}{\log 2}}$ $= n^{\frac{\log 4}{\log 2}} = n^2$	$9^h = 9^{\frac{\log n}{\log 3}}$ $= n^{\frac{\log 9}{\log 3}} = n^2$	$2^h = 2^{\frac{\log n}{\log 4}}$ $= n^{\frac{\log 2}{\log 4}} = n^{\frac{1}{2}}$
f) $T(n)$ as a sum	$\sum_{i=0}^h (\# \text{ at level})$ (time each) $= \sum_{i=0}^{\Theta(\log n)} 4^i \cdot \left(\frac{n}{2^i}\right)$ $= n \cdot \sum_{i=0}^{\Theta(\log n)} 2^i$	$\sum_{i=0}^h (\# \text{ at level})$ (time each) $= \sum_{i=0}^{\Theta(\log n)} 9^i \cdot \left(\frac{n}{3^i}\right)^2$ $= n^2 \cdot \sum_{i=0}^{\Theta(\log n)} 1$	$\sum_{i=0}^h (\# \text{ at level})$ (time each) $= \sum_{i=0}^{\Theta(\log n)} 2^i \cdot \left(\frac{n}{4^i}\right)^2$ $= n^2 \cdot \sum_{i=0}^{\Theta(\log n)} \left(\frac{1}{8}\right)^i$
g) Dominated by?	geometric increasing: dominated by last term = number of base cases	arithmetic sum: levels roughly the same = time per level $\cdot$ # levels	geometric decreasing: dominated by first term = top level
h) $\Theta(T(n))$	$T(n) = \Theta\left(n^{\frac{\log a}{\log b}}\right)$ $= \Theta\left(n^{\frac{\log 4}{\log 2}}\right) = \Theta(n^2)$	$T(n) = \Theta(f(n) \log n)$ $= \Theta(n^2 \log n)$	$T(n) = \Theta(f(n))$ $= \Theta(n^2)$

**Evaluating The Sum:** To evaluate this sum, we will use the Adding Made Easy Approximations given in Section 5. The four cases below correspond to the sum being geometric increasing, arithmetic, harmonic, or bounded tail.

**Dominated by Top:** If the computation time  $\Theta(f(n))$  for the top level of recursion is sufficiently bigger than the computation time  $\Theta(n^{\frac{\log a}{\log b}})$  for the base cases, then the above sum decreases geometrically. The Adding Made Easy Approximations then tells us that such a sum is bounded by the first term, giving that  $T(n) = \Theta(f(n))$ . In this case, we say that the computation time of the recursive program is dominated by the time for the top level of the recursion.

**Sufficiently Big:** A condition for  $\Theta(f(n))$  to be sufficiently bigger than  $\Theta(n^{\frac{\log a}{\log b}})$  is that  $f(n) \geq n^c \cdot \log^d n$ , where  $c > \frac{\log a}{\log b}$ . A more general condition can be expressed as  $f(n)/n^{\frac{\log a}{\log b}} \geq n^{\Omega(1)}$ , where  $f$  is a simple analytical function.

**Proving Geometric Decreasing:** What remains is to prove that the sum is in fact geometric decreasing. To simplify things, consider  $f(n) = n^c$  for constant  $c$ . We want to evaluate  $T(n) = \Theta(\sum_{i=0}^h a^i f(\frac{n}{b^i})) = \Theta(\sum_{i=0}^h a^i (\frac{n}{b^i})^c) = \Theta(n^c \cdot \sum_{i=0}^h (\frac{a}{b^c})^i)$ . The Adding Made Easy Approximations state that this sum decreases geometrically if the terms decrease exponentially in terms of  $i$  (not in terms of  $n$ ). This is the case as long as the constant  $\frac{a}{b^c}$  is less than one. This occurs when  $a < b^c$  or equivalently when  $c > \frac{\log a}{\log b}$ .

**Even Bigger:** Having an  $f(n)$  that is even bigger means that the time is dominated even more by the top level of recursion. For example,  $f(n)$  could be  $2^n$  or  $2^{2^n}$ .

Having  $f(n)/n^{\frac{\log a}{\log b}} \geq n^{\Omega(1)}$  ensures that  $f(n)$  is at least as big as  $n^c$  for some  $c > \frac{\log a}{\log b}$ . Hence, for these it is still the case that  $T(n) = \Theta(f(n))$ .

**A Little Smaller:** This function  $f(n) \geq n^c \cdot \log^d n$  with  $c > \frac{\log a}{\log b}$  is a little smaller when we let  $d$  be negative. However, this small change does not change the fact that the time is dominated by the top level of recursion and that  $T(n) = \Theta(f(n))$ .

**Levels Arithmetic:** If the computation time  $\Theta(f(n))$  for the top levels of recursion is sufficiently close to the computation time  $\Theta(n^{\frac{\log a}{\log b}})$  for the base cases, then all levels of the recursion contribute roughly the same to the computation time and hence the sum is arithmetic. The Adding Made Easy Approximations give that the total is the number of terms times the “last” term. The number of terms is the height of the recursion which is  $h = \frac{\log(n)}{\log b} = \Theta(\log(n))$ . A complication, however, is that the terms are in the reverse order from what they should be to be arithmetic. Hence, the “last” term is top one,  $\Theta(f(n))$ . This gives  $T(n) = \Theta(f(n) \log n)$ .

**Sufficiently The Same:** A condition for  $\Theta(f(n))$  to be sufficiently close to  $\Theta(n^{\frac{\log a}{\log b}})$  is that  $f(n) = \Theta(n^c \cdot \log^d n)$ , where  $c = \frac{\log a}{\log b}$  and  $d > -1$ . A more general condition can be expressed as  $f(n)/n^{\frac{\log a}{\log b}} = [\log n]^{\Theta(1)-1}$ , where  $f$  is a simple analytical function.

**A Simple Case  $f(n) = n^{\frac{\log a}{\log b}}$ :** To simplify things, let us first consider  $f(n) = n^c$  where  $c = \frac{\log a}{\log b}$ . Above we computed  $T(n) = \Theta(\sum_{i=0}^h a^i f(\frac{n}{b^i})) = \Theta(\sum_{i=0}^h a^i (\frac{n}{b^i})^c) = \Theta(n^c \cdot \sum_{i=0}^h (\frac{a}{b^c})^i)$ . But now  $c$  is chosen so that  $\frac{a}{b^c} = 1$ . This simplifies the sum to  $T(n) = \Theta(n^c \cdot \sum_{i=0}^h (1)^i) = \Theta(n^c h) = \Theta(f(n) \log n)$ . Clearly, this sum is arithmetic.

**A Harder Case  $f(n) = n^{\frac{\log a}{\log b}} \log^d n$ :** Now let us consider how much  $f(n)$  can be shifted away from  $f(n) = n^{\frac{\log a}{\log b}}$  and still have the sum be arithmetic. Let us consider  $f(n) = n^c \cdot \log^d n$ , where  $c = \frac{\log a}{\log b}$  and  $d > -1$ . This adds an extra log to the sum, giving  $T(n) = \Theta(\sum_{i=0}^h a^i f(\frac{n}{b^i})) = \Theta(\sum_{i=0}^h a^i (\frac{n}{b^i})^c \log^d(\frac{n}{b^i})) = \Theta(n^c \cdot \sum_{i=0}^h (1)^i [\log(\frac{n}{b^i})]^d) = \Theta(n^c \cdot \sum_{i=0}^h [\log(n) - i \log(b)]^d)$ . This looks ugly, but we can simplify it by reversing the order of the terms.

**Reversing the Terms:** The expression  $\frac{n}{b^i}$  with  $i \in [0, h]$  takes on the values  $n, \frac{n}{b}, \frac{n}{b^2}, \dots, 1$ . Reversing the order of these values gives  $1, b, b^2, \dots, n$ . This is done more formally by letting  $i = h - j$  so that  $\frac{n}{b^i} = \frac{n}{b^{h-j}} = b^j$ . Now summing the terms in reverse order with  $j \in [0, h]$  gives  $T(n) = \Theta(n^c \cdot \sum_{i=0}^h [\log(\frac{n}{b^i})]^d) = \Theta(n^c \cdot \sum_{j=0}^h [\log(b^j)]^d) = \Theta(n^c \cdot \sum_{j=0}^h [j \log b]^d)$ . This  $[\log b]^d$  is a constant that we can hide in the Theta. This gives  $T(n) = \Theta(n^c \cdot \sum_{j=0}^h j^d)$ .

**Proving Reverse Arithmetic:** The Adding Made Easy Approximations state that this sum  $T(n) = \Theta(n^c \cdot \sum_{j=0}^h j^d)$  is arithmetic as long as  $d > -1$ . In this case, the total is  $T(n) = \Theta(n^c \cdot h \cdot h^d) = \Theta(n^c \cdot \log^{d+1} n)$ . Another way of viewing this sum is that it is approximately the “last term”, which after reversing the order is the top term  $f(n)$ , times the number of terms, which is  $h = \Theta(\log n)$ . In conclusion,  $T(n) = \Theta(f(n) \log n)$ .

**Levels Harmonic:** Taking the Adding Made Easy Approximations the obvious next step, we let  $d = -1$  in the above calculations in order to make a harmonic sum.

**Strange Requirement:** This new requirement is that  $f(n) = \Theta(n^c \cdot \log^d n)$ , where  $c = \frac{\log a}{\log b}$  and  $d = -1$ .

**The Total:** Continuing the above calculations gives  $T(n) = \Theta(n^c \cdot \sum_{j=0}^h j^d) = \Theta(n^c \cdot \sum_{j=0}^h \frac{1}{j})$ . This is a harmonic sum adding to  $T(n) = \Theta(n^c \cdot \log h) = \Theta(n^c \log \log n) = \Theta(n^{\frac{\log a}{\log b}} \log \log n)$  as required.

**Dominated by Base Cases:** If the computation time  $\Theta(f(n))$  for the top levels of recursion is sufficiently smaller than the computation time  $\Theta(n^{\frac{\log a}{\log b}})$  for the base cases, then the terms increase quickly enough for the sum to be bounded by the last term. This last term consists of  $\Theta(1)$  computation time for each of the  $n^{\frac{\log a}{\log b}}$  base cases. In conclusion,  $T(n) = \Theta(n^{\frac{\log a}{\log b}})$ .

**Sufficiently Big:** A condition for  $\Theta(f(n))$  to be sufficiently smaller than  $\Theta(n^{\frac{\log a}{\log b}})$  is that  $f(n) \leq n^c \cdot \log^d n$ , where either  $c < \frac{\log a}{\log b}$  or ( $c = \frac{\log a}{\log b}$  and  $d < -1$ ). A more general condition can be expressed as  $\frac{f(n)}{n^{\frac{\log a}{\log b}}} \leq [\log n]^{-1-\Omega(1)}$ , where  $f$  is a simple analytical function.

**Geometric Increasing:** To begin, suppose that  $f(n) = n^c$  for constant  $c < \frac{\log a}{\log b}$ . Before we evaluated  $T(n) = \Theta(\sum_{i=0}^h a^i f(\frac{n}{b^i})) = \Theta(\sum_{i=0}^h a^i (\frac{n}{b^i})^c) = \Theta(n^c \cdot \sum_{i=0}^h (\frac{n}{b^c})^i)$ .  $c < \frac{\log a}{\log b}$  gives  $\frac{a}{b^c} > 1$ . Hence, this sum increases exponentially and the total is bounded by the last term,  $T(n) = \Theta(n^{\frac{\log a}{\log b}})$ .

**Bounded Tail:** With the Adding Made Easy Approximations, however, we learned that the sum does not need to be exponentially decreasing (we are viewing the sum in reverse order) in order to have a bounded tail. Let us now consider  $f(n) = n^c \cdot \log^d n$ , where  $c = \frac{\log a}{\log b}$  and  $d < -1$ . Above we got that  $T(n) = \Theta(n^c \cdot \sum_{j=0}^h j^d)$ . This is a bounded tail, summing to  $T(n) = \Theta(n^c) \Theta(1) = \Theta(n^{\frac{\log a}{\log b}})$  as required.

**Even Smaller:** Having an  $f(n)$  that is even smaller means that the time is even less effected by the time for the top level of recursion. This can be expressed as  $\frac{f(n)}{n^{\frac{\log a}{\log b}}} \leq [\log n]^{-1-\Omega(1)}$ .

**Low Order Terms:** If the recurrence relation is  $T(n) = 4T(\frac{n}{2} - \sqrt{n} + \log n - 5) + \Theta(n^3)$  instead of  $T(n) = 4T(\frac{n}{2}) + \Theta(n^3)$ , these low order terms, though ugly, do not make a significant difference to the total. We will not prove this formally.

This concludes our approximation of  $\Theta(T(n))$ .

## 6.4 Other Examples

We will now apply the same idea to other recurrence relations.

**Exponential Growth for a Linear Number of Levels:** Here we consider recurrence relations of the form  $T(n) = aT(n-b) + f(n)$  for  $a > 1$ ,  $b > 0$ , and later  $c > 0$ . Each instances having  $a$  subinstances means that as before, the number of subinstances grows exponentially. Decreasing the subinstance by only an additive amount means that the number of levels of recursion is linear in the size  $n$  of the initial instance. This gives an exponential number

of base cases, which will dominate the time unless, the work  $f(n)$  in each stack frame is exponential itself.

**Using the Table:** As done before, a good way to evaluate a recurrence relation is to fill in the following table.

Dominated by	Base Cases	Levels Arithmetic	Top Level
Examples	$T(n) = aT(n - b) + n^c$	$T(n) = aT(n - b) + a^{\frac{1}{b}n}$	$T(n) = aT(n - b) + a^{\frac{2}{b}n}$
a) # frames at the $i^{th}$ level?		$a^i$	
b) Size of instance at the $i^{th}$ level?		$n - i \cdot b$	
c) Time within one stack frame	$f(n - i \cdot b) = (n - i \cdot b)^c$	$f(n - i \cdot b) = a^{\frac{n-i \cdot b}{b}}$ $= a^{\frac{1}{b}n} \cdot a^{-i}$	$f(n - i \cdot b) = a^{\frac{2(n-i \cdot b)}{b}}$ $= a^{\frac{2}{b}n} \cdot a^{-2i}$
d) # levels		$n - h \cdot b = 0, h = \frac{n}{b}$ Having a base case of size zero makes the math the cleanest.	
e) # base case stack frames		$a^h = a^{\frac{1}{b}n}$	
f) $T(n)$ as a sum	$\sum_{i=0}^h (\# \text{ at level})$ (time each) $= \sum_{i=0}^{\frac{n}{b}} a^i \cdot (n - i \cdot b)^c$	$\sum_{i=0}^h (\# \text{ at level})$ (time each) $= \sum_{i=0}^{\frac{n}{b}} a^i \cdot a^{\frac{1}{b}n} \cdot a^{-i}$ $= a^{\frac{1}{b}n} \cdot \sum_{i=0}^{\frac{n}{b}} 1$	$\sum_{i=0}^h (\# \text{ at level})$ (time each) $= \sum_{i=0}^{\frac{n}{b}} a^i \cdot a^{\frac{2}{b}n} \cdot a^{-2i}$ $= a^{\frac{2}{b}n} \cdot \sum_{i=0}^{\frac{n}{b}} a^{-i}$
g) Dominated by?	geometric increasing: dominated by last term $=$ number of base cases	arithmetic sum: levels roughly the same $=$ time per level $\cdot$ # levels	geometric decreasing: dominated by first term $=$ top level
h) $\Theta(T(n))$	$T(n) = \Theta(a^{\frac{n}{b}})$	$T(n) = \Theta(\frac{n}{b} \cdot a^{\frac{n}{b}})$	$T(n) = \Theta(a^{\frac{2}{b}n})$

**Fibonacci Numbers:** A famous recurrence relation is  $Fib(0) = 0$ ,  $Fib(1) = 1$ , and  $Fib(n) = Fib(n - 1) + Fib(n - 2)$ . Clearly,  $Fib(n)$  grows slower than  $T(n) = 2T(n - 1) = 2^n$  and faster than  $T(n) = 2T(n - 2) = 2^{n/2}$ . It follows that  $Fib(n) = 2^{\Theta(n)}$ . If one wants more details then following calculations amazingly enough give the exact value.

**Careful Calculations:** Let us guess the solution  $Fib(n) = \alpha^n$ . Plugging this into  $Fib(n) = Fib(n - 1) + Fib(n - 2)$  gives that  $\alpha^n = \alpha^{n-1} + \alpha^{n-2}$ . Dividing through by  $\alpha^{n-2}$  gives  $\alpha^2 = \alpha + 1$ . Using the formula for quadratic roots gives that either  $\alpha = \frac{1+\sqrt{5}}{2}$  or  $\alpha = \frac{1-\sqrt{5}}{2}$ . Any linear combination of these two solutions will also be a valid solution, namely  $Fib(n) = c_1 \cdot (\alpha_1)^n + c_2 \cdot (\alpha_2)^n$ . Using the fact that  $Fib(0) = 0$  and  $Fib(1) = 1$  and solving for  $c_1$  and  $c_2$  gives that  $Fib(n) = \frac{1}{\sqrt{5}} \left[ \left[ \frac{1+\sqrt{5}}{2} \right]^n - \left[ \frac{1-\sqrt{5}}{2} \right]^n \right]$ .

**One Subinstance Each:** If each stack frame calls recursively at most once, then there is only a single line of stack frames, one per level. The total time  $T(n)$  is the sum of these stack frames.

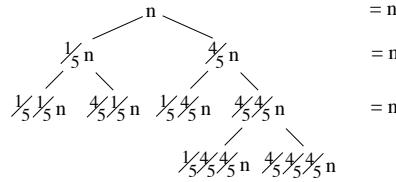
Dominated by	Base Cases	Levels Arithmetic	Top Level
Examples	$T(n) = T(n - b) + 0$	$T(n) = T(n - b) + n^c$	$T(n) = T(n - b) + 2^{cn}$
a) # frames at the $i^{th}$ level?		one	
b) Size of instance at the $i^{th}$ level?		$n - i \cdot b$	
c) Time within one stack frame	$f(n - i \cdot b) = \text{zero}$ Except for the base case which has work $\Theta(1)$	$f(n - i \cdot b) = (n - i \cdot b)^c$	$f(n - i \cdot b) = 2^{c(n-i \cdot b)}$
d) # levels		$n - h \cdot b = 0, h = \frac{n}{b}$	
e) # base case stack frames		one	
f) $T(n)$ as a sum	$\sum_{i=0}^h (\# \text{ at level}) \cdot (\text{time each}) \\ = \left( \sum_{i=0}^{\frac{n}{b}-1} 1 \cdot 0 \right) + 1 \cdot \Theta(1) = \Theta(1)$	$\sum_{i=0}^h (\# \text{ at level}) \cdot (\text{time each}) \\ = \sum_{i=0}^{\frac{n}{b}} 1 \cdot (n - i \cdot b)^c$	$\sum_{i=0}^h (\# \text{ at level}) \cdot (\text{time each}) \\ = \sum_{i=0}^{\frac{n}{b}} 1 \cdot 2^{c(n-i \cdot b)}$
g) Dominated by?	geometric increasing: dominated by last term $=$ number of base cases	arithmetic sum: levels roughly the same $=$ time per level $\cdot$ # levels	geometric decreasing: dominated by first term $=$ top level
h) $\Theta(T(n))$	$T(n) = \Theta(1)$	$T(n) = \Theta\left(\frac{n}{b} \cdot n^c\right) \\ = \Theta(n^{c+1})$	$T(n) = \Theta(2^{cn})$

**Recurising on Instances of Different Sizes and Linear Work:** We will now consider recurrence relations of the form  $T(n) = T(u \cdot n) + T(v \cdot n) + \Theta(n)$  for constants  $0 < u, v < 1$ . Note that each instance does an amount of work that is linear in its input instance and it produces two smaller subinstances of sizes  $u \cdot n$  and  $v \cdot n$ . Such recurrence relations will arise in Section 15.1 for quick sort.

**$u = v = \frac{1}{b}$ :** To tie in with what we did before, note that if  $u = v = \frac{1}{b}$ , then this recurrence relation becomes  $T(n) = 2T\left(\frac{1}{b}n\right) + \Theta(n)$ . We have seen that the time is  $T(n) = \Theta(n)$  for  $b > 2$  or  $u = v < \frac{1}{2}$ ,  $T(n) = \Theta(n \log n)$  for  $u = v = \frac{1}{2}$ , and  $T(n) = \Theta(n^{\frac{\log 2}{\log b}})$  for  $u = v > \frac{1}{2}$ .

**For  $u + v = 1$ ,  $T(n) = \Theta(n \log n)$ :** The pivotal values  $u = v = \frac{1}{2}$  is generalized to  $u+v = 1$ . The key here is that each stack frame forms two subinstances by splitting its input instance into two pieces, whose sizes sum to that of the given instance.

**Total of Instance Sizes at Each Level is  $n$ :** If you trace out the tree of stack frames, you will see that at each level of the recursion, the sum of the sizes of the input instances is  $\Theta(n)$ , at least until some base cases are reached. To make this concrete, suppose  $u = \frac{1}{5}$  and  $v = \frac{4}{5}$ . The sizes of the instance at each level are given below. Note the sum of any sibling pair sums to their parent and the total at the each level is  $n$ .



**Total of Work at Each Level is  $n$ :** Because the work done by each stack frame is linear in its size and the sizes at each level sums to  $n$ , it follows that the work at each level adds up to  $\Theta(n)$ . As an example, if the sum of their sizes is  $(\frac{1}{5})(\frac{1}{5})n + (\frac{4}{5})(\frac{1}{5})n + (\frac{1}{5})(\frac{4}{5})n + (\frac{4}{5})(\frac{4}{5})n = n$ , then the sum of their work is  $c(\frac{1}{5})(\frac{1}{5})n + c(\frac{4}{5})(\frac{1}{5})n + c(\frac{1}{5})(\frac{4}{5})n + c(\frac{4}{5})(\frac{4}{5})n = cn$ . Note that the same would not be true if the time in a stack frame was say quadratic,  $c((\frac{1}{5})(\frac{1}{5})n)^2$ .

**Number of Levels is  $\Theta(\log n)$ :** The left-left-left branch terminates quickly after  $\frac{\log n}{\log u} = 0.43 \log n$  for  $u = \frac{1}{5}$ . The right-right-right branch terminates slower, but still after only  $\frac{\log n}{\log u} = 3.11 \log n$  levels. The middle paths terminate somewhere in between these two extremes.

**Total Work is  $\Theta(n \log n)$ :** Given that there is  $\Theta(n)$  work at each of  $\Theta(\log n)$  levels gives that the total work is  $T(n) = \Theta(n \log n)$ .

**More Parts:** The same principle applies no matter how many parts the instance is split into. For example, the solution of  $T(n) = T(u \cdot n) + T(v \cdot n) + \dots + T(w \cdot n) + \Theta(n)$  is  $T(n) = \Theta(n \log n)$  as long as  $u + v + \dots + w = 1$ .

**Less Even Split Means Bigger Constant:** Although solution of  $T(n) = T(u \cdot n) + T((1-u) \cdot n) + \Theta(n)$  is  $T(n) = \Theta(n \log n)$  for all constants  $0 < u < 1$ , the constant hidden in the  $\Theta$  for  $T(n)$  is smaller when the split is closer to  $u = v = \frac{1}{2}$ . If you are interested, the constant is roughly  $\frac{1}{u} / \log(\frac{1}{u})$  for small  $u$ , which grows as  $u$  gets small.

**For  $u + v < 1$ ,  $T(n) = \Theta(n)$ :** If  $u + v < 1$ , then the sum of the sizes of the sibling subinstances is smaller than that of the parent subinstance. In this case, the sum work at each level decreases by the constant factor  $(u + v)$  each level. The total work is then  $T(n) = \sum_{i=0}^{\Theta(\log n)} (u + v)^i \cdot n$ . This computation is dominated by the top level and hence  $T(n) = \Theta(n)$ .

**For  $u + v > 1$ ,  $T(n) = \Theta(n^\alpha)$ :** In contrast, if  $u + v > 1$ , then the sum of the sizes of the sibling subinstances is larger than that of the parent subinstance and the sum work at each level increases by the constant factor  $(u + v)$  each level. The total work is dominated by the base cases. The total time will be  $T(n) = \Theta(n^\alpha)$  for some constant  $\alpha > 1$ . One could compute how this  $\alpha$  is a function of  $u$  and  $v$ , but we will not bother.

**Exercise 6.4.1** (*See solution in Section V*) Use the above table method to compute each of the following recursive relations.

1.  $T(n) = nT(n - 1) + 1$
2.  $T(n) = 2T(\sqrt{n}) + n$

This completes the discussion on recurrence relations.

# Chapter 7

## Abstractions

It is hard to think about love in terms of the firing of neurons or to think about a complex data base as a sequence of magnetic charges on a disk. In fact, a ground stone of human intelligence is the ability to build hierarchies of abstractions within which to understand things. This requires cleanly partitioning details into objects, attributing well-defined properties and human characteristics to these objects, and seeing the whole as being more than the sum of the parts. This book focuses on different abstractions, analogies, and paradigms of algorithms and of data structures. The tunneling of electrons through doped silicon are abstracted as *AND*, *OR*, and *NOT* gate operations; which are further abstracted as machine code operations; the executing of lines of Java code; subroutines; algorithms; and the abstractions of algorithmic concepts. Magnetic charges on a disk are abstracted as sequences of zeros and ones; which are abstracted as integers, strings and pointers; which are abstracted as a graph; which is used to abstract a system of trucking routes. The key to working within a complex system is the ability to both simultaneously and separately operate with each of these abstractions and to be able to translate this work between the levels. It is useful to use different notations, analogies, and metaphors to view the same ideas. Each representation at your disposal provides new insights and new tools.



### 7.1 Different Representations of Algorithms

In your courses and in your jobs, you will need to be proficient not only at writing working code, but also at understanding algorithms, designing new algorithms, and describing algorithms to others in such way that their correctness is transparent. Some useful ways of representing an algorithm for this purpose are code, tracing a computation on an example input, mathematical logic, personified analogies and metaphors, and various other higher abstractions. These can be so different from each other that it is hard to believe that they describe the same algorithm. Each has its own advantages and disadvantages.

**Computers vs Humans:** One spectrum in which representations can differ arises in what is useful for computers versus what is useful for humans. Machine code, which is necessary for computers because they can only blindly follow instructions, is painfully tedious for humans. Higher and higher level programming languages are being developed with which it is easier

for humans to write and understand algorithms. Many people (and texts) have the perception that program code is the only representation of an algorithm that they will ever need. However, I recommend using higher levels abstraction for giving a human an intuitive understanding of an algorithm.

**What vs Why:** Code focuses on what the algorithm does. We, however, need to understand why it works.

**Concrete vs Abstract:** It is useful when attempting to understand or explain an algorithm to trace out the code on a few example input instances. This has the advantages of being concrete, dynamic, and often visual. It is best to find the simplest examples that capture the key ideas. Elaborate examples are prone to mistakes, and it is hard to find the point being made amongst all of the details. However, this process does not prove that the algorithm gives the correct answer on every possible input. To do this, you must be able to prove it works for an arbitrary input. Neither does tracing the algorithm provide the higher-level intuition needed. For this, you need to see the pattern in what the code is doing and convince yourself that this pattern of actions solves the problem.

**Details vs Big Picture:** Computers require that every implementation detail is filled in. Humans, on the other hand, are more comfortable with a higher level of abstraction. Lots of implementation details distract from the underlying meaning. Gauge your audience. Leave some details until later and don't insult them by giving those that your audience can easily fill in. One way of doing this is by describing how subroutines and data structures are used without describing how they are implemented. I also recommend ignoring code and instead using the other algorithmic abstractions that we will cover in this text. Once the big picture is understood, however, code has the advantage of being precise and succinct. Sometimes pseudo code, which is a mixture of code and English, can be helpful. Part of the art of describing an algorithm is knowing when to use what level of abstraction and which details to include.

**Formal vs Informal:** We all tend to resist anything that is too abstract or too mathematical. However, math and logic are able to provide a precise and succinct clarity that neither code nor English can give. For example, some representations of algorithms like DFA and Turing Machines provide a formalism that is useful for specific applications or for proving theorems. On the other hand, personified analogies and metaphors help to provide both intuition and humor.

**Value Simplicity:** Abstract away the unnecessary features of a problem. Our goal is to understand and think about complex algorithms in simple ways. If you have seen an algorithm before, don't tune out when it is being explained. There are deep ideas within the simplicity.

**Don't Describe Code:** When told to describe an algorithm without giving code, do not describe the code in a paragraph, as in "We loop through i, being from 1 to n-1. For each of these loops, ..."

**Incomplete Instructions:** If an algorithm works either way, it is best not to specify it completely. For example, you may talk of selecting an item from a set without specifying which item is selected. If you then go on to prove that the algorithm works, you are effectively saying that the algorithm works for every possible way of choosing that value. This flexibility may be useful for the person who later implements the algorithm.

I recommend learning how to develop, think about, and explain algorithms within all of these different representations. Practice. Design your own algorithms, day dream about them, ask questions about them, and perhaps the most important, explain them to other people.

## 7.2 Abstract Data Types (ADTs)

In addition to algorithms, we will develop abstractions of data objects.

**Definition of an ADT:** Abstract data types are completely described by their functional specification - their effect, independent of implementation. In contrast, data structures and algorithms are implementations of ADTs.

**Abstract Objects:** The description of an algorithm may talk of integers, reals, strings, sets, sorted lists, queues, stacks, graphs, binary trees, and other such abstract objects without mentioning the data structure or algorithms that handle them. In the end, computers represent these abstract objects as strings of zeros and ones; this does not mean that we have to.

**Abstract Operations:** It is also useful to group a number of steps into one abstract operation, such as “sort a list”, “pop a stack”, or “find a path in a graph.” It is easier to associate these actions with the object they are acting upon than with the algorithm that is using them.

**Relationships between Objects:** We can also talk abstractly about the relationships between objects with concepts such as paths, parent node, child node, and the like. The following two algorithmic conditions are equivalent. Which do you find more intuitively understandable?

- if  $A[i] \leq A[\lfloor i/2 \rfloor]$
- if the value of the node in the binary tree is at most that of its parent

**Advantages:** ADTs make it easier in the following ways to code, understand, design, and describe algorithms.

**Clear Specifications:** Each ADT requires a clear specification so that the boss is clear about what he wants, the person implementing it knows what to code, the user knows how to use it, and the tester knows what it is supposed to do.

**Information Hiding:** It is generally believed that global variables are bad form. If many routines in different parts of the system directly modify a variable in undocumented ways, then the program is hard to understand, modify, and debug. For this same reason, programmers who include an ADT in their code are not allowed to access or modify the data except via the operations provided. This is referred to as *information hiding*.

**Signal:** Using an ADT like a stack in your algorithm automatically tells someone attempting to understand your algorithm a great deal about the purpose of this data structure.

**Clean Boundaries:** Using abstract data types breaks your project into smaller parts that are easy to understand and provides clean boundaries between these parts.

**User:** The clean boundaries allow one to understand and to use an ADT to develop other algorithms without being concerned with how it is implemented. The information hiding means that the user does not need to worry about accidentally messing up the data structure in ways not intended.

**Implementer:** The clean boundaries also allow someone else to implement the ADT without knowing how it will be used. Given this, it also allows the implementation to be modified without unexpected effects to the rest of the code.

**Code Reuse:** Data structures like stack, sets, and graphs are used in many applications. By defining a general purpose ADT, the code, the understanding, and the mathematical theory developed for it can be reused over and over. In fact, an abstract data type can be used many times within the same algorithm by having many instances of it.

**Optimizing:** Having a limited set of operations guides the implementer to use techniques that are efficient for these operations yet may be slow for the operations excluded.

**Running Time:** Generally, the running time of an operation is not a part of the description of an ADT, but is tied to a particular implementation. The thinking is that the ADT is only concerned with what it does, not how it does it. On the other hand, the description of an ADT needs to tell the user about the ADT's functionality and if the user cares not only about whether his project works, but also about how fast it is, then he needs to be told, if not the running times, at least the relative running times: which operations are expensive and which are cheap. Sometimes there is a trade off. One of two operations is going to be expensive and the other cheap. There is a choice which is which. The user of the ADT does not need to know everything about the implementation, but might want to know this information so that he can make his own choices about which ADTs and which operations to use.

**Examples of Abstract Data Types:** The following are examples frequently used.

**Simple Types:** Integers, floating point numbers, strings, arrays, and records are abstract data types provided by all programming languages. The limited sets of operations that are allowed on each of these structures are well documented.

**The List ADT:** The concept of a list, eg. shopping list, is well known to everyone.

**Abstract Objects:** A list consists of sequence of objects or elements. You can have a list of anything, even a list of lists. You can also have the empty list.

**Relations between Objects:** A list also specifies the order of the elements within it. Unlike arrays, there are no empty positions in this ordering.

**Abstract Operations:** The  $i^{th}$  element in the list can be read and modified. A specific element can be searched for, returning its index. An element can be deleted or inserted at a specified index. This shifts the indices of the elements after it.

**Uses:** Most algorithms need to keep track of some unpredetermined number of elements. These can be stored in a list.

**Running Times:** There are trade offs. Some implementations (eg. arrays) access the  $i^{th}$  element in  $\Theta(1)$  time, but require  $\Theta(n)$  time to insert an element. Others (eg. linked lists) allow insertions in  $\Theta(1)$  time, but require  $\Theta(n)$  time to access the  $i^{th}$  element.

**The Set ADT:** A set is basically a bag within which you can put any elements that you like.

**Abstract Objects:** A set is the same as a list, except that the elements are not ordered. Again, the set can contain any types of element. Sets of sets is common. So is the empty set.

**Relations between Objects:** The only relation is knowing which elements are in the set and which are not.

**Abstract Operations:** Given an element and a set, one can determine whether or not the element is contained in the set. This is often called *a membership query*. One can ask for an arbitrary element from a set, determine the number of elements (size) of a set, iterate through all the elements, and add and delete elements.

**Uses:** Often an algorithm does not require its elements in a specific order. Using the set abstract data type instead of a list gives more freedom to the implementer.

**Running Times:** Not being required to maintain an order of the elements, the set ADT can be implemented much more efficiently than a list. One would think that searching for an element would take  $\Theta(n)$  time as you compare it with each element in the set. Surprisingly, however, all of these set operations can be done in constant time, i.e., independent of the number of items in the set.

**The Set System ADT:** A set system allows you to have a set (or list) of sets.

**Abstract Objects:** The additional operations of a system of sets are being able to form the *union*, *intersection*, or *subtraction* of sets to create new sets. Also one can use an operation called *find* to determine which set a given element is contained in.

**Running Times:** Taking intersections and subtractions of two sets requires  $\Theta(n)$  time. However, another quite surprising result is that on disjoint sets, the union and find operations can be done on average in a constant amount of time for all practical purposes. See Section 10.5.

**Stack ADT:** A stack is analogous to a stack of plates, in which a plate can be removed or added only at the top.

**Abstract Objects:** It is the same as a list, except that its set of operations is restricted.

**Abstract Operations:** A *push* is the operation of adding a new element to the *top* of the stack. A *pop* is the operation of removing and returning the top element from the stack. One can determine whether a stack is empty and what the top element is without popping it. However, the rest of the stack is hidden from view. This order of accessing the elements is referred to as *Last-In-First-Out* (LIFO).

**Uses:** Stacks are the key data structure for recursion and parsing.

**Running Times:** The major advantage of restricting oneself to stack operations instead of using a general list is that both push and pop can be done in constant time, i.e., independent of the number of items in the stack. See Section 10.

**The Queue ADT:** A queue is analogous to a line-up for movie tickets; the first person to have arrived is the first person served.

**Abstract Objects:** Like a stack, a queue is the same as a list, except with a different restricted set of operations.

**Abstract Operations:** The basic operations of a queue are to *inserting* an element into the *rear* and to *remove* the element that is at the *front*. Again, the rest of the queue is hidden from view. This order of accessing the elements is referred to as *First-In-First-Out* (FIFO).

**Uses:** An operating system will have a queue of jobs to run; a printer server, a queue of jobs to print; a network hub, a queue of packets to transmit; and a simulation of a bank, a queue of customers.

**Running Times:** Queue operations can also be done in constant time.

**The Priority Queue ADT:** In some queues, the more important elements are allowed to move to the front of the line.

**Abstract Objects:** Associated with each element is a priority.

**Abstract Operations:** When *inserting* an element, its priority must be specified. This priority can later be changed. When *removing*, the element with the highest priority in the queue is removed and returned. Ties are broken arbitrarily.

**Uses:** Priority queues can be used instead of a queue when the priority scheme is needed.

**Running Times:** There are implementations in which inserting takes  $\Theta(1)$  time, but removing takes  $\Theta(n)$  time. For others, these are reversed. Surprisingly, however, there are implementations (eg. Heaps and AVL trees) in which both can be done in  $\Theta(\log n)$  time. See Sections 10.5 and 16.4.

**The Dictionary ADT:** A dictionary associates a meaning with each word. Similarly, a dictionary ADT associates data with each *key*.

**Abstract Objects:** A dictionary is similar to an array, except that each element is indexed by its key instead of by its integer location in the array.

**Abstract Operations:** Given a key one can *retrieve* the data associated with the key. One can also *insert* a new element into the dictionary by providing both the key and the data.

**Uses:** A name or social insurance number can be the key used to access all the relevant data about the person. Algorithms can also use such abstract data types to organize their data.

**Running Time:** The dictionary abstract data type can be implemented as a set. Hence, all the operations can be done in constant time using hash tables.

**The Graph ADT:** A *graph* is an abstraction of a binary relation, such as network of roads between cities or of the love relation between people. The cities or people are called *nodes* and the roads or love connections are called *edges*. The information stored is which pairs of nodes are connected by an edge. Though a drawing implicitly places each node at some location on the page, a key abstraction of a graph is that the location of a node is not specified. For example, the three pairs of graphs in Figure 7.1 are considered to be the same (isomorphic). In this way, a graph can be used to represent any set of relationships between pairs of objects from any fixed set of elements.

**Abstract Objects:** A formal graph consists only of a set of nodes and a set of edges, where a node is simply a label (eg. [1..n]) and an edge is a pair of nodes. More generally, however, more data can be associated with each node or with each edge. For example, often each edge is associated with a number denoting its weight, cost, or length.

**Relations between Objects:** An edge can also be thought of as a relation between its nodes.

**Abstract Operations:** The basic operations are to determine whether an edge is in a graph, to add or delete an edge, and to iterate through the neighbors of a node. There is a huge literature of more complex operations that one might want to do. For example, one might want to determine which nodes have paths between them or to find the shortest path between two nodes. See Chapter 20.

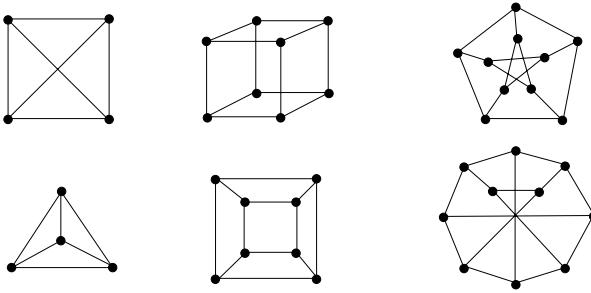


Figure 7.1: The top three graphs are famous: the complete graph on four nodes, the cube, and the Peterson graph. The bottom three graphs of the same three graphs with their nodes laid out differently.

**Uses:** A graph is a nice clean ADT to use to represent the data from a surprisingly large number of applications. The advantage is that there is a vast literature of theorems and algorithms to draw on to solve the problems that arise in your particular application.

**Space and Time:** One implementation, *adjacency matrix*, requires  $\Theta(n^2)$  space, which is the number of potential edges, but only  $\Theta(1)$  time to access the edges. Another, *adjacency list*, requires  $\Theta(E)$  space, which is the number of actual edges, but requires time proportional to the degree of a node to access the edges. See Section 10.5.

**Exercise 7.2.1** For each of the three pairs of graphs in Figure 7.1, number the nodes in such the way that  $\langle i, j \rangle$  is an edge in one if and only if it is an edge in the other.

**The Tree ADT:** Data is often organized into a hierarchy. A person has children, who have children of their own. The boss has people under her, who have people under them. The abstract data type for organizing this data is a *tree*.

**Abstract Objects:** A tree is a hierarchy of nodes, each with information associated with it.

**Relations between Objects:** Each node has a level in the tree, a list of children, and a parent (unless it is the root).

**Abstract Operations:** The basic operations are to retrieve the information about a node, loop through its children, and determine its parent. One also can search for a specific node or traverse through all the nodes in the tree in some specific order.

**Uses:** Often data falls naturally into a hierarchy. For example, expressions like  $f = x \times (y + 7)$  can be expressed as a tree. Also binary search trees and heaps are tree data structures that are alternatives to sorted data. See Sections 10.5, 16.5, and 16.4.

**Running Time:** Given a pointer to a node, one can find its parent and children in constant time. Traversing the tree, clearly takes time proportional to its depth, which in the case of balanced trees is  $\Theta(\log n)$ .

Graphs	
$G = \langle V, E \rangle$	A graph $G$ is specified by a set of nodes $V$ and a set of edges $E$ .
Vertex	Another name for a node
$u, v \in V$	Two nodes within the set of nodes
$\langle u, v \rangle \in E$	An edge within the set of edges
$n$ and $m$	The number of nodes and edges respectively
Directed vs Undirected	The edges of a graph can be either directed (drawn with an arrow and stored as an ordered pair of nodes) or undirected, (drawn as a line and stored as a unordered set $\{u, v\}$ of two nodes).
Adjacent	Nodes $u$ and $v$ are said to be adjacent if they have an edge between them.
Neighbors, $N(u)$	The neighbors of node $u$ are those nodes that are adjacent to it.
Degree, $d(v)$	The degree of a node is the number of neighbors that it has.
Path	A path from $u$ to $v$ is a sequence of nodes (or edges) such that between consecutive nodes there is an edge.
Simple Path	A path is simple if no node is visited more than once.
Connected	A graph is connected if there is a path between every pair of nodes.
Connected Component	The nodes of an undirected graph can be partitioned based on which nodes are connected.
Cycle	A cycle is a path from $u$ back to itself.
Subgraph	A subgraph is a subset of the nodes and a subset of the edges that are between them.
Acyclic	An acyclic graph is a graph that contains no cycles.
DAG	A DAG is a directed acyclic graph
Tree	A tree is an undirected acyclic graph
Complete Graph	In a complete graph every pair of nodes has an edge.
Dense vs Sparse	A dense graph contains most of the possible edges and a sparse graph contains few of them.
Singleton	A node with no neighbors

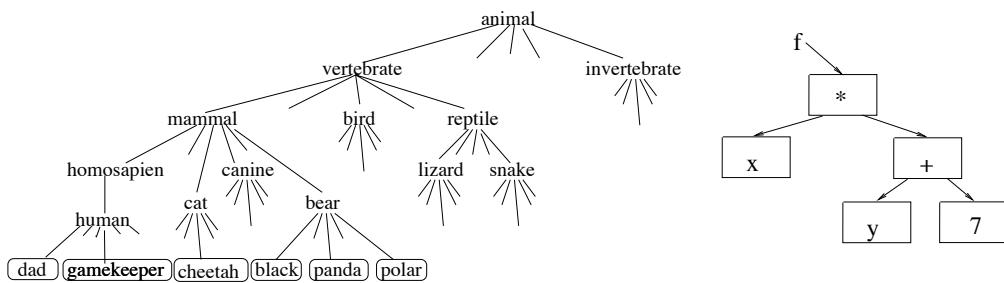


Figure 7.2: Classification Tree of Animals and a tree representing the expression  $f = x \times (y + 7)$ .

Trees	
Root	Node at the top
RootInfo(tree)	The information stored at the root node
Child of node $u$	One of the nodes just under node $u$
Parent of node $u$	The unique node immediately above node $u$
Siblings	The nodes with same parent
Ancestors of node $u$	The nodes on the unique path from node $u$ to the root
Descendants of node $u$	All the nodes below node $u$
Leaf	A node with no children
Level of a node	The number of nodes in the unique path from the node to the root Some definitions say that the root is in level 0, others say level 1
Height of tree	The maximum level. Some definitions say that a node with a single node has height 0, others say height 1. It depends on whether you count nodes or edges.
Depth of node $u$	The number of nodes (or edges) on the unique path from node $u$ to the root
Binary Tree	A binary tree is a tree in which each node has at most two children. Each of these children is designated as either the right child or as the left child.
leftSub(tree)	Left subtree of root
rightSub(tree)	Right subtree of root

## Part II

# Iterative Algorithms & Loop Invariants

## Chapter 8

# Iterative Algorithms & Loop Invariants Abstractions, Techniques and Theory

A technique used in many algorithms is to start at the beginning and take one step at a time towards the final destination. An algorithm that proceeds in this way is referred to as an *iterative* algorithm.

Though this sounds simple, designing, understanding, and describing such an algorithm can still be a daunting task. It becomes less daunting if we do not need to worry about the entire journey at once but can focus separately on one step at a time.

The classic proverb advises us to focus first on the first step. This, however, is difficult to do before we know where we are going. Instead, I advise, first make a general statement about the types of places that the computation might be during its algorithmic journey. Then, given anyone such place, consider what single step the computation should take from there. This is the method of *assertions* and *loop invariants*.



### 8.1 Assertions and Invariants as Boundaries between Parts

Whether you are designing an algorithm, coding an algorithm, trying to understand someone else's algorithm, describing an algorithm to someone else, or formally proving that an algorithm works, you do not want to do it all at once. It is much easier to first break the algorithm into clear well-defined pieces and then to separately design, code, understand, describe, or prove correct each of the pieces. Assertions provide clean boundaries between these parts. They state what the part can expect to have already been accomplished when the part begins and what must be accomplished when it completes. Invariants are the same, except they apply either to a loop that is executed many times or to object oriented data structure that has an ongoing life.

**Assertions as Boundaries around a Part:** An algorithm is broken into systems, subsystems, routines, and subroutines. You must be very clear about the goals of the overall algorithm and

of each of these parts. Pre and postconditions are assertions that provide a clean boundary around each of these.

**Specifications:** Any computational problem or subproblem is defined as follows:

**Preconditions:** The preconditions state any assumptions that must be true about the input instance for the algorithm to operate correctly.

**Postconditions:** The postconditions are statements about the output that must be true when the algorithm returns.

**Goal:** An algorithm for the problem is correct if for *every* legal input instance, i.e., every instance meeting the preconditions, the required output is produced, i.e., the output meets the postconditions. On the other hand, if the input instance does not meet the preconditions, then all bets are off (but the program should be polite). Formally, we express this as  $\langle \text{pre-cond} \rangle \ \& \ \text{code}_{\text{alg}} \Rightarrow \langle \text{post-cond} \rangle$ .

**Example:** The problem *Sorting* is defined as:

**Preconditions:** The input is a list of  $n$  values with the same value possibly repeated.

**Postconditions:** The output is a list consisting of the same  $n$  values in non-decreasing order.

**One Step At A Time:** Worry only about your job.

**Implementing:** When you are writing a subroutine, you do not need to know how it will be used. You only need to make sure that it works correctly, namely if the input instance for the subroutine meets its preconditions then you must ensure that its output meets its postconditions.

**Using:** Similarly, when you are using the subroutine, you do not need to know how it was implemented. You only need to ensure that the input instance you give it meets the subroutine's preconditions. Then you can trust that the output meets its postconditions.

**More Examples:**

**Directions:** When planning or describing a trip from my house to York University, I assume that the traveler is initially at my house without considering how he got there. In the end, he must be at York University.

**Books:** Both the author and the reader of a chapter or even of a paragraph need to have a clear description of what the author assumes the reader to know before reading the part and what he hopes the part will accomplish.

**Proofs:** A lemma needs a precise statement of what it proves, both so that the proof of the theorem can use it and so that it is clear what the proof of the lemma is supposed to prove.

**Assertions as Check Points:** Assertions are also able to provide check points along the path of the computation to allow everyone to know where the computation is and where it is going next. Each assertion is a statement about the current state of the computation's data structures that is either true or false. It is made at some particular point during the execution of an algorithm. If it is false, then something has gone wrong in the logic of the algorithm.

**Example:** The task of getting from my home to York University is broken into stages. Assertions along the way are: "We are now at home." "We are now at Ossington

Station.” “We are now at St. George.” “We are now at Downsview.” “We are now at York.”

**Designing, Understanding, and Proving Correct:** Generally, assertions are not tasks for the algorithm to perform, but are only comments that are added to assist the designer, the implementer, and the reader in understanding the algorithm and its correctness.

**Debugging:** Some languages allow you to insert assertions as lines of code. If during the execution such an assertion is false, then the program automatically stops with a useful error message. This is very helpful when debugging your program. Even after the code is complete it is useful to leave these checks in place. This way if something goes wrong, it is easier to determine why. This is what is occurring when an error box pops up during the execution of a program telling you to contact the vendor if the error persists. Not all interesting assertions, however, can be tested feasibly within the computation itself.

**Invariants of Ongoing Systems:** Algorithms for computational problems with pre and postconditions compute one function, taking an input at the beginning and stopping once the output has been produced. Other algorithms, however, are more dynamic. These are for systems or data structures that periodically receive information to which they must react in a manner that is sensitive to its history, i.e., reflects the information already received. In an object oriented language, these are implemented with objects each of which has its own internal variables and tasks. A calculator example is presented in Section 9.2. The data structures described in Chapter 10 are other examples. Each such system has a set of integrity constraints that must be maintained. These are basically assertions that must be true every time the system is entered or left. We refer to them as *invariants*. They come in two forms.

**Public Specifications:** Each specification of a system has a number of invariants that any outside user of the system needs to know about so that he can use the system correctly. For example, a user should be assured that his bank account will always correctly reflect the amount of money that he has. He should also know what happens when this amount goes negative. Similarly, a user of a stack should know that when he pops it, the last object that he pushed will be returned.

**Hidden Invariants:** Each implementation of a system has a number of invariants that only the system’s designers need to know about and maintain. For example, a stack may be implemented using a linked list which always maintains a pointer to the first object.

**Loop Invariants:** A special type of assertions consist of those that are placed at the top of a loop. They are referred to as *loop invariants*, because they must hold true every time the computation returns to the top of the loop.

**Algorithmic Technique:** Loop invariants are the focus of a major part of this text because they form the basis of the algorithmic technique referred to as *iterative algorithms*.

## 8.2 An Introduction to Iterative Algorithms

An algorithm that consists of small steps implemented by a main loop is called an *iterative*. Understanding what happens within a loop can be surprisingly difficult. Formally, one proves that an iterative algorithm works correctly using loop invariants. (See Section 8.5.2.) I believe that this

concept can also be used as *a level of abstraction* within which to design, understand, and explain iterative algorithms.

**Iterative Algorithms:** A good way to structure many computer programs is to store the key information you currently know in some data representation and then each iteration of the main loop takes a step towards your destination by making a simple change to this data.

**Loop Invariants:** A loop invariant is an assertion that must be true about the state of this data structure at the start of every iteration and when the loop terminates. It expresses important relationships among the variables and in doing so expresses the progress that has been made towards the postcondition.

**Analogy:** The following are three analogies of these ideas.

**One Step At A Time:** The Buddhist way is not to have cravings or aversions about the past or the future, but to be in the here and now. Though you do not want to have a fixed predetermined fantasy about your goal, you do need to have some guiding principles to point you more or less in the correct direction. Meditate until you understand the key aspects of your current situation. Trust that your current simple needs are met. Rome was not built in a day. Your current task is only to make some small simple step. With this step, you must both ensure that your simple needs are met tomorrow and that you make some kind of progress towards your final destination. Don't worry. Be happy.

**Steps From A Safe Location To A Safe Location:** The algorithm's attempt to get from the preconditions to the postconditions is like being dropped off in a strange city and weaving a path to some required destination. The current *state* or location of the computation is determined by values of all the variables. Instead of worrying about the entire computation, take one step at a time. Given the fact that your single algorithm must work for an infinite number of input instances and given all the obstacles that you pass along the way, it can be difficult to predict where computation might be in the middle of the execution. Hence, for every possible location that the computation might be, the algorithm attempts to define what step to take next. By ensuring that each such step will make some progress towards the destination, the algorithm can ensure that the computation does not wander aimlessly, but weaves and spirals in towards the destination. The problem with this approach is that there are far too many locations that the computation may be in and some of these may be very confusing.

Let us say a location is *safe* if the loop invariant holds. The algorithm then defines how to take a step from each such safe location. This step now has two requirements. It must make progress and it must not go from a safe location to an unsafe location. Assuming that initially the computation is in a safe location, this ensures that it remains in a safe location. This ensures that the next step is always defined. Assuming that initially the computation is not infinitely far from its destination, making progress each step ensures that eventually the computation will have made enough progress that the computation stops. Finally, the design of the algorithm must ensure that being in a safe location with this much progress made is enough to ensure that the final destination is reached. This completes the design of the algorithm.



**A Relay Race:** The loop can be viewed as a relay race. Your task is not to run the entire race. You take the baton from a friend. Though in the back of your mind you know that the baton has traveled many times around the track already, this fact does not concern you. You have been assured that the baton meets the conditions of the *loop invariants*. Your task is to carry the baton once around the track. You must make progress, and you must ensure that the baton still meets the conditions after you have taken it around the track, i.e., that the loop invariants have been maintained. Then, you hand the baton on to another friend. This is the end of your job. You do not worry about how it continues to circle the track. The difficulty is that you must be able to handle *any* baton that meets the loop invariants.

### Structure of An Iterative Algorithm:

```

begin routine
  ⟨pre-cond⟩
  codepre-loop    % Establish loop invariant
  loop
    ⟨loop-invariant⟩
    exit when ⟨exit-cond⟩
    codeloop      % Make progress while maintaining the loop invariant
  end loop
  codepost-loop   % Clean up loose ends
  ⟨post-cond⟩
end routine

```

**The Most Important Steps:** The most important steps when developing an iterative algorithm within the loop invariant level of abstraction are the following:

### The Steps:

- What invariant is maintained?
- How is this invariant initially obtained?
- How is progress made while maintaining the invariant?
- How does the exit condition together with the invariant ensure that the problem is solved?

**Induction Justification:** Section 8.5.2 uses induction to prove that if the loop invariant is initially established and is maintained then it will be true at the beginning of each iteration. Then in the end, this loop invariant is used to prove that problem is solved.

**Faith in the Method:** Instead of rethinking difficult things every day, it is better to have some general principles with which to work. For example, every time you walk into a store, you do not want to be rethinking the issue of whether or not you should steal. Similarly, every time you consider a hard algorithm, you do not want to be rethinking the issue of whether or not you believe in the loop invariant method. Understanding the algorithm itself will be hard enough. Hence, while reading this chapter you should once and for all come to understand and believe down to the depth of your soul how the above mentioned steps are sufficient to develop an algorithm. Doing this can be difficult. It requires a whole new way of looking at algorithms. However, at least for the duration of this course, adopt this as something that you believe in.

### 8.3 Examples: Quadratic Sorts

The algorithms Selection Sort and Insertion Sort are classic examples of iterative algorithms. Even though you have likely seen them before, let us use them to understand these required steps.

**Selection Sort:** We maintain that the  $k$  smallest of the elements are sorted in a list. The larger elements are in a set on the side. Progress is made by finding the smallest element in the remaining set of large elements and adding this selected element at the end of the sorted list of elements. This increases  $k$  by one. Initially, with  $k = 0$ , all the elements are set aside. Stop when  $k = n$ . At this point, all the elements have been selected and the list is sorted.

If the input is presented as an array of values, then sorting can happen in place. The first  $k$  entries of the array store the sorted sublist, while the remaining entries store the set of values that are on the side. Finding the smallest value from  $A[k + 1] \dots A[n]$  simply involves scanning the list for it. Once it is found, moving it to the end of the sorted list involves only swapping it with the value at  $A[k + 1]$ . The fact that the value  $A[k + 1]$  is moved to an arbitrary place in the right-hand side of the array is not a problem, because these values are considered to be an unsorted set anyway.

**Running Time:** We must select  $n$  times. Selecting from a sublist of size  $i$  takes  $\Theta(i)$  time. Hence, the total time is  $\Theta(n + (n-1) + \dots + 2 + 1) = \Theta(n^2)$ .

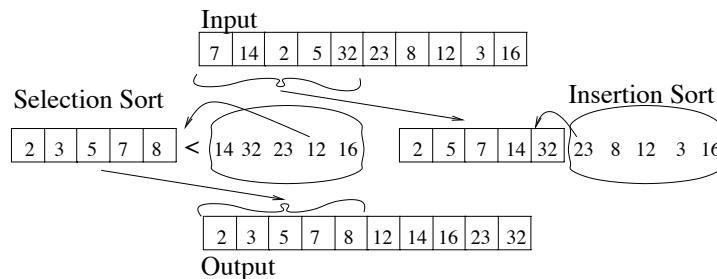


Figure 8.1: The loop invariants for insertion sort and selection sort are demonstrated. We will see that these are examples of more of the “more of the output” and of the “more of the input” types of loop invariants.

**Insertion Sort:** We maintain a subset of elements sorted within a list. The remaining elements are off to the side somewhere. Progress is made by taking one of the elements that is off to

the side and *inserting* it into the sorted list where it belongs. This gives a sorted list that is one element longer than it was before. Initially, think of the first element in the array as a sorted list of length one. When the last element has been inserted, the array is completely sorted.

There are two steps involved in inserting an element into a sorted list. The most obvious step is to locate where it belongs. The second step to shift all the elements that are bigger than the new element one to the right to make room for it. You can find the location for the new element using a binary search. However, it is easier to search and shift the larger elements simultaneously.

**Running Time:** We must insert  $n$  times. Inserting into a sublist of size  $i$  takes  $\Theta(i)$  time. Hence, the total time is  $\Theta(1 + 2 + 3 + \dots + n) = \Theta(n^2)$ .

**For  $i = 1..n$ :** One might think that one does not need a loop invariant when accessing each element of an array. Using the statement “for  $i = 1..n$  do”, one probably does not. However, without a loop invariant code like “`i=1; while( $i \leq n$ )  $A[i] = 0$ ;  $i = i + 1$ ; end while`” is surprisingly prone to the error of being off by one. The loop invariant is that when at the top of the loop,  $i$  indexes the next element to handle.

## 8.4 The Steps in Developing an Iterative Algorithm

This section presents the more detailed steps that I recommend using when developing an iterative algorithm.

**Preliminaries:** Before we can design an iterative algorithm, we need to know precisely what it is supposed to do and have some idea about the kinds of steps it will take.

**1) Specifications:** Carefully write the specifications for the problem.

**Preconditions:** What are the legal instances (inputs) for which the algorithm should work?

**Postconditions:** What is the required output for each legal instance?

**2) Basic Steps:** As a preliminary to designing the algorithm it can be helpful to consider what basic steps or operations might be performed in order to make progress towards solving this problem. Take a few of these steps on a simple input instance in order to get some intuition as to where the computation might go. How might the information gained narrow down the computation problem?

**A Middle Iteration on a General Instance:** Although the algorithm that you are designing needs to work correctly on each possible input instance, do not start by considering special case input instances. Instead, consider a large and general instance. If there are a number of different types of instances, consider the one that seems to be the most general.

Being an iterative algorithm, this algorithm will execute its main loop many times. Each of these is called an *iteration*. Each iteration must work correctly and must connect well with the previous and the next iterations. Do not start designing the algorithm by considering the steps before the loop or by considering the first iteration. In order to see the big picture of the algorithm more clearly, jump into the middle of the computation. From there, design the

main loop of your algorithm to work correctly on a single iteration. The steps to do this are as follows.

**3) Loop Invariant:** Designing the loop invariant is the most difficult and creative part of designing an algorithm.

**Picture from the Middle:** Describe what you would like the data structure to look like when the computation is at the beginning of this middle iteration.

**Draw a Picture:** Your description should leave your reader with a visual image. Draw a picture if you like.

**Don't Be Frightened:** A loop invariant should not consist of formal mathematical mumbo jumbo if an informal description would get the idea across better. On the other hand, English is sometimes misleading and hence a more mathematical language sometimes helps. Say things twice if necessary. In general, I recommend pretending that you are describing the algorithm to a first year student.

**Safe Place:** A loop invariant must ensure that the computation is still within a safe place along the road and has not fallen into a ditch or landed in a tree.

**The Work Completed:** The loop invariant must characterize what work has been completed towards solving the problem and what work still needs to be done.

**Typical Techniques:** The following are typical techniques that arise in many iterative algorithms.

**More of the Input:** Suppose the input consists of  $n$  objects (eg. an array of  $n$  integers or a graph with  $n$  nodes). It would be reasonable for the algorithm to read them in one at a time. After reading the first  $i$  of them, what would we like that the algorithm to have accomplished? One common idea is to for the algorithm to temporarily pretend that this prefix of the input is in fact the entire input. Then a good loop invariant might include “I currently have a solution for the input consisting solely of these first  $i$  objects.”

**Insertion Sort:** An example of an algorithm using the “more of the input” type of loop invariant is insertion sort. After  $i$  iterations, the first  $i$  elements of the input are sorted. See Figure 8.1.

**Not Handled Each:** A common description students give of the “more of the input” type of loop invariant is “I have handled and have a solution for the first  $i$  objects in the input.” If the problem is, given an array of integers, double each integer, then this is a good loop invariant. However, this is NOT correct for most algorithms. A single object in the input does not need a separate solution, the input as a whole does. Instead, assume that the prefix consisting of the first  $i$  objects was the entire input instance. We have a solution for this instance.

**More of the Output:** If the solution is a structure composed of many pieces (eg. an array of integers, a set, or a path), a natural thing to try is to construct the solution one piece at a time. A good loop invariant might include “I have constructed  $i$  correct pieces of the output so far.”

**Selection Sort:** An example of an algorithm using the “more of the output” type of loop invariant is selection sort. After  $i$  iterations, the first  $i$  elements of the output have been produced. Again, see Figure 8.1.

**Narrow Search Space:** If you are searching for something, try narrowing the search space, maybe decreasing it by one or even better cutting it in half. A good loop invariant might include “I know that the thing being searched for is not outside of this narrowed search space.”

**Binary Search:** Binary search is the obvious example of an algorithm using type of loop invariant. See Section 11.1.

**Case Analysis:** Try the obvious thing. For which input instances does it work and for which does it not work? Now you only need to find an algorithm that works for those later cases. An invariant might include “I have tried these cases.”

**Work Done:** Time is wasted if the algorithm redoing the same work more than once. Try to arrange things so that the work can be reused. A good loop invariant clearly states what work has been done so that it can be reused.

- 4) **Measure of Progress:** At each iteration, the computation must make progress. You, however, have complete freedom to decide how this progress is measured. A significant step in the design of an algorithm is defining this *measure* according to which you gage the amount of progress that the computation has made. For example, a measure of progress might include

**More of the Input:** how much of the input you have a solution for,

**More of the Output:** how much of the output you have constructed,

**Narrow Search Space:** the size of the search space in which you have narrowed the search,

**Case Analysis:** how many cases you have tried,

**Work Done:** or how much of the work has been done.

- 5) **Main Steps:** You are now ready to make your first guess as to what the algorithmic steps within the main loop will be. When doing this, recall that your only goal is to make progress while maintaining the loop invariant. The main step might

**More of the Input:** extend a solution for the input consisting of the first  $i$  objects into a solution for the input consisting of the first  $i + 1$  of them.

**More of the Output:** construct the next piece of the output,

**Narrow Search Space:** narrow down the search space farther,

**Case Analysis:** try the next case,

**Work Done:** or do some more work.

- 6) **Maintaining the Loop Invariant:** To check whether you have succeeded in the last step, you must prove that the loop invariant is maintained by these steps that you have put within the loop.

Again, assume that you are in the middle of the computation at the top of the loop. Your loop invariant describes the state of the data structure. Refer back to the picture that you drew. Execute one iteration of the loop. You must prove that when you get back to the top of the loop again, the requirements set by the loop invariant are met once more. For example, you must make sure that

**More of the Input:** your new solution is correct when considering more of the input,

**More of the Output:** you constructed this next piece of the output correctly,

**Narrow Search Space:** the thing being searched for is not outside of the new narrow down search space,

**Case Analysis:** you applied the latest case correctly,

**Work Done:** and that your new work has been done correctly.

- 7) **Making Progress:** You must also prove that progress of at least one (according to your measure) is made every time the algorithm goes around the loop. Sometimes, according to the most obvious measure of progress, the algorithm can iterate around the loop without making any measurable progress. This is not acceptable. The danger is that the algorithm will loop forever. In this case, you must define another measure that better shows how you are making progress during such iterations.

**Beginning & Ending:** Once you have passed through steps 2-7 enough times that you get them to work smoothly together, you can consider beginning the first iteration and ending the last iteration.

- 8) **Initial Conditions:** Now that you have an idea of where you are going, you have a better idea about how to begin. In this step, you must develop the initiating pseudo code for the algorithm. Your only task is to initially establish the loop invariant. Do it in the easiest way possible. For example, if you need to construct a set such that all the dragons within it are purple, the easiest way to do it is to construct the empty set. Note that all the dragons in this set are purple, because it contains no dragons that are not purple.

**More of the Input:** What is the solution when the input instance contains no objects?

**More of the Output:** Initially, you have constructed none of the output.

**Narrow Search Space:** You know that the thing, if anywhere, must be somewhere in the initial search space.

**Case Analysis:** You have tried no cases.

**Work Done:** You have done no work.

Careful. Sometimes it is difficult to know how to initially set the variable to make the loop invariant initially true. In such cases, try setting them to ensure that it is true after the first iteration. For example, what is the maximum value within an empty list of values? One might think 0 or  $\infty$ . However, a better answer is  $-\infty$ . When adding a new value, one uses the code  $newMax = \max(oldMax, newValue)$ . Starting with  $oldMax = -\infty$ , gives the correct answer when the first value is added.

- 9) **Exit Condition:** Now that you have an idea of where you will be in the middle of your computation, you know from which direction you will be coming when you finally arrive at the destination. The next thing that you must design in your algorithm is the condition that causes the computation to break out of the loop. For example, your algorithm might exit when

**More of the Input:** it has a solution for the entire input,

**More of the Output:** the output is complete

**Narrow Search Space:** it has found the thing being searched for or has narrowed the search space down to nothing,

**Case Analysis:** you have tried all the cases,

**Work Done:** or you have done all the needed work.

**Stuck:** Sometimes, however, though your intuition is that your algorithm designed so far is making progress each iteration, you have no clue whether heading in this direction the algorithm will ever solve the problem or how you would know it if it

happens. Consider situations in which your algorithm gets stuck, i.e., is unable to execute its main loop and make progress. For such situations, you must either think of other ways for your algorithm to make progress or have it exit. A good first step is to exit. In step 11, you will have to prove that when your algorithm exits, you actually are able to solve the problem. If you are unable to do this, then you will have to go back and redesign your algorithm.

**Loop While vs Exit When:** As an aside, note that the following are equivalent:

while( <i>A</i> and <i>B</i> )	loop
....	<i>&lt;loop-invariant&gt;</i>
end while	exit when (not <i>A</i> or not <i>B</i> )
	...
	end loop

The second is more useful here because it focuses on the conditions needed to exit the loop, while the first focuses on the conditions needed to continue. A secondary advantage of the second is that it also allows you to slip in the loop invariant between the top of the loop and the exit condition.

- 10) **Termination and Running Time:** In this next step, you must ensure that the algorithm does not loop forever. To do this, you do not need to precisely determine the exact point at which the exit condition will be met. Instead, state some upper bound and prove that if this much progress has been made, then the exit condition has definitely been met. If it exits earlier than this, all the better. For example, you might know that your algorithm will stop by the time the amount of progress made is

**More of the Input:** the size of the input  $n$ ,

**More of the Output:** the size of the output,

**Narrow Search Space:** a search space size of zero,

**Case Analysis:** the number of different cases,

**Work Done:** or the amount of work needed.

There is no point in wasting a lot of time developing an algorithm that does not run fast enough to meet your needs. Hence, it is worth estimating at this point the running time of the algorithm. Generally, this done by dividing the total progress required by the amount of progress made each iteration.

- 11) **Ending:** In this step, you must ensure that once the loop has exited that you will be able to solve the problem. Develop the pseudo code needed after the loop to complete these last steps. To help you with this task, you know two things about the state your data structures will be in at this point in time. First, you know that the loop invariant will be true. You know this because the loop invariant must always be true. Second, you know that the exit condition is true. This you know, by the fact that the loop has exited. From these two facts and these facts alone, you must be able to deduce that the problem can be solved with only a few last touches.

**More of the Input:** If you have a solution for the input consisting of the first  $i$  objects in the input and  $i = n$ , then you have a solution for your input instance.

**More of the Output:** If you have correctly constructed  $i$  pieces of the output and  $i$  is the required size of the output, then you have correctly constructed the output.

**Narrow Search Space:** We have proved that if the thing being searched for is anywhere then it must be in this narrowed search space. The size of the narrowed search space is zero. Hence, the thing is not anywhere.

**Case Analysis:** If you have exhausted all the cases and the cases cover all the possible cases, then your case much have been covered.

**Work Done:** If the amount of work that you have done equals the amount needed (and you have done the right work) then you are done.

**12) Testing:** Try your algorithm by hand on a couple of examples.

**Fitting the Pieces Together:** The above steps complete all the parts of the algorithm. Though the steps listed are independent and can be completed in any order, there is an interdependence between them. In fact, you really cannot complete anyone step until you have an understanding of all of them. Sometimes after completing the steps, things do not quite fit together. It is then necessary to cycle through the steps again using your new understanding. I recommend cycling through the steps a number of times. The following are more ideas to use as you cycle through these steps.

**Flow Smoothly:** The loop invariant should flow smoothly from the beginning to the end of the algorithm.

- At the beginning, it should follow easily from the preconditions.
- It should progress in small natural steps.
- Once the exit condition has been met, the postconditions should easily follow.



**Ask for 100%:** A good philosophy in life is to ask for 100% of what you want, but not to assume that you will get it.

**Dream:** Do not be shy. What would you like to be true in the middle of your computation? This may be a reasonable loop invariant, or it may not be.

**Pretend:** Pretend that a genie has granted your wish. You are now in the middle of your computation and your dream loop invariant is true.

**Maintain the Loop Invariant:** From here, are you able to take some computational steps that will make progress while maintaining the loop invariant? If so, great. If not, there are two common reasons.

**Too Weak:** If your loop invariant is too weak, then the genie has not provided you with everything you need to move on.

**Too Strong:** If your loop invariant is too strong, then you will not be able to establish it initially or maintain it.

**No Unstated Assumptions:** Often students give loop invariants that lack detail or are too weak to proceed to the next step. Don't make assumptions that you don't state. As a check, pretend that you are a Martian who has jumped into the top of the loop knowing *nothing* that is not stated in the loop invariant.



- 13) Special Cases:** In the above steps, you were considering one general type of large input instances. If there is a type of input that you have not considered, repeat the steps considering them. There may also be special case interactions that need to be considered. These likely occur near the beginning or the end of the algorithm. Continue repeating the steps until you have considered all of the special case input instances.

Though these special cases may require separate code, start by tracing out what the algorithm that you have already designed would do given such an input. Often this algorithm will just happen to handle a lot of these cases automatically without requiring separate code. When adding code to handle a special case, be sure to check that the previously handled cases still are handled.

- 14) Coding and Implementation Details:** Now you are ready to put all the pieces together and produce pseudo code for the algorithm. It may be necessary at this point to provide extra implementation details.
- 15) Formal Proof:** After the above steps seem to fit well together, you should cycle one last time through them being particularly careful that you have met all the formal requirements needed to prove that the algorithm works. These requirements are as follows.

**6') Maintaining Loop Invariant (Revisited):** Many subtleties can arise given the huge number of different input instances and the huge number of different places the computation might find itself. In this step, you must double check that you have caught all of these subtleties. To do this, you must ensure that the loop invariant is maintained when the iteration starts in *any* of the possible places that the computation might be in. This is particularly important for those complex algorithms for which you have little grasp of where the computation will go.

**Proof Technique:** To prove this, pretend that you are at the top of the loop. It does not matter how you got there. As said, you may have dropped in from Mars. You can assume that the loop invariant is satisfied, because your task here is only to maintain the loop invariant. You can also assume that the exit condition is not satisfied, because otherwise the loop would exit at this point and there would be no need to maintain the loop invariant during an iteration. However, besides these two things you know nothing about the state of the data structure. Make no other

assumptions. Execute one iteration of the loop. You must then be able to prove that when you get back to the top of the loop again, the requirements set by the loop invariant are met once more.

**Differentiating Between Iterations:** The assignment “ $x = x + 2$ ” is meaningful to a computer scientist as a line of code because it is an action to be taken. However, to a mathematician it is a statement that is false unless you are working over the integers modulo 2. We do not want to see such statements in mathematical proofs. Hence, it is useful to have a way to differentiate between the values of the variables at the beginning of the iteration and the new values after going around the loop one more time. One notation is to denote the former with  $x'$  and the latter with  $x''$ . (One could also use  $x_i$  and  $x_{i+1}$ .) Similarly,  $\langle \text{loop-invariant}' \rangle$  can be used to state that the loop invariant is true for the  $x'$  values and  $\langle \text{loop-invariant}'' \rangle$  that it is true for the  $x''$  values.

**The Formal Statement:** Whether or not you want to prove it formally, the formal statement that must be true is  $\langle \text{loop-invariant}' \rangle$  & not  $\langle \text{exit-cond} \rangle$  &  $\text{code}_{\text{loop}} \Rightarrow \langle \text{loop-invariant}'' \rangle$ .

**The Formal Proof Technique:** Assume

- $\langle \text{loop-invariant}' \rangle$
- not  $\langle \text{exit-cond} \rangle$
- the effect of code B (e.g.,  $x'' = x' + 2$ )

and then prove from these assumptions the conclusion that  $\langle \text{loop-invariant}'' \rangle$ .

**7') Making Progress (Revisited):** These special cases may also affect whether progress is made during each iteration. Again, assume nothing about the state of the data structure except that the loop invariant is satisfied and the exit condition is not. Execute one iteration of the loop. Prove that significant progress has been made according to your measure.

**8') Initial Conditions (Revisited):** You must ensure that the initial code establishes the loop invariant. The difficulty, however, is that you must ensure that this happens no matter what the input instance is. When the algorithm begins, the one thing that you can assume is that the preconditions are true. You can assume this because, if it happens to be that they are not true then you are not expected to solve the problem.

The formal statement that must be true is  $\langle \text{pre-cond} \rangle$  &  $\text{code}_{\text{pre-loop}} \Rightarrow \langle \text{loop-invariant} \rangle$ .

**11') Ending:** In this step, you must ensure that once the loop has exited that the loop invariant (which you know will always be true) and the exit condition (which caused the loop to exit) together will give you enough so that your last few touches solves the problem. The formal statement that must be true is  $\langle \text{loop-invariant} \rangle$  &  $\langle \text{exit-cond} \rangle$  &  $\text{code}_{\text{post-loop}} \Rightarrow \langle \text{post-cond} \rangle$ .

**10') Running Time:** The task of designing an algorithm is not complete until you have determined its running time. Your measure of progress will be helpful when bounding the number of iterations that are executed. You must also consider the amount of work per iteration. Sometimes each iteration requires a different amount of work. In this case, you will need to approximate the sum.

**12') Testing:** Try your algorithm by hand on more examples. Consider both general input instances and special cases. You may also want to code it up and run it.

This completes the steps for developing an iterative algorithm. See Section 11.1 for example in which these examples are done in detail. Likely you will find that coming up with the loop invariant is the hardest part of designing an algorithm. It requires practice, perseverance, and insight. However, from it, the rest of the algorithm follows easily with little extra work. Here are a few more pointers that should help you design loop invariants.

**A Starry Night:** How did Van Gogh come up with his famous painting, A Starry Night? There's no easy answer. In the same way, coming up with loop invariants and algorithms is an art form.



**Use This Process:** Don't come up with the loop invariant after the fact to make me happy. Use it to design your algorithm.

**Know What a Loop Invariant is:** Be clear about what a loop invariant is. On midterms, many students write, "the LI is ..." and then give code, a precondition, a postcondition, or some other inappropriate piece of information. For example, stating something that is ALWAYS true, such as  $1 + 1 = 2$  or "The root is the max of any heap", may be useful information for the answer to the problem, but should not be a part of the loop invariant.

## 8.5 A Formal Proof of Correctness

Our philosophy is about learning how to think about, develop, and describe algorithms in such way that their correctness is transparent. In order to accomplish this, one needs to at least understand the required steps in a formal proof of correctness.

### 8.5.1 Using Assertions

We will start with seeing how assertions are used to prove the correctness of a program.

**Definition of the Correctness of a Program:** An algorithm works correctly on every input instance if, for an arbitrary instance,  $\langle \text{pre-cond} \rangle \& \text{code}_{\text{alg}} \Rightarrow \langle \text{post-cond} \rangle$ .

Consider some instance. If this instance meets the preconditions, then the output must meet the postconditions. If this instance does not meet the preconditions, then all bets are off (but the program should be polite).

The correctness of an algorithm is only with respect to the stated specifications. It does not guarantee that it will work in situations that are not taken into account by this specification.

**Breaking into Parts:** The method of proving that an algorithm is correct is by breaking the algorithm into smaller and smaller well defined parts. If needed, one can break the code down to single lines of code.

**Assertions are Pre and Postconditions:** The assertions at the beginning and the end of a block of code act as the pre and postconditions for the block, defining what task of the block.

**A Single Line of Code:** When a part of the algorithm is broken down to a single line of code then this line of code has implied pre and postconditions and we must trust the compiler to translate the line of code correctly. For example:

$\langle pre-assignment-cond \rangle$ : The variables  $x$  and  $y$  have meaningful values.

$z = x + y$

$\langle post-assignment-cond \rangle$ : The variable  $z$  takes on the sum of the value of  $x$  and the value of  $y$ .  
The previous value of  $z$  is lost.

**Combining the Parts:** Once the individual parts of the algorithm are proved to be correct, the correctness of the combined parts can be proved as follows.

**Structure of Algorithmic Fragment:**

```
<assertion0>
  code1
<assertion1>
  code2
<assertion2>
```

**Steps in Proving Its Correctness:**

- The proof of correctness of the first part proves that  $\langle assertion_0 \rangle \ \& code_1 \Rightarrow \langle assertion_1 \rangle$
- and of the second part that  $\langle assertion_1 \rangle \ \& code_2 \Rightarrow \langle assertion_2 \rangle$ .
- Formal logic allows us to combine these to give  $\langle assertion_0 \rangle \ \& \langle code_1 \& code_2 \rangle \Rightarrow \langle assertion_2 \rangle$ .

**Proving the Correctness of an *if* Statement:** Assertions can also be used to prove the correctness of code whose computation flow is determined at execution time.

**Structure of Algorithmic Fragment:**

```
<pre-if-cond>
  if( <condition> ) then
    codetrue
  else
    codefalse
  end if
<post-if-cond>
```

**Steps in Proving Its Correctness:** Its correctness  $\langle pre-if-cond \rangle \ \& \ code \Rightarrow \langle post-if-cond \rangle$  is proved by proving that each of the two paths through the proof are correct, namely

- $\langle pre-if-cond \rangle \ \& \ \langle condition \rangle \ \& \ code_{true} \Rightarrow \langle post-if-cond \rangle$  and
- $\langle pre-if-cond \rangle \ \& \ \neg \langle condition \rangle \ \& \ code_{false} \Rightarrow \langle post-if-cond \rangle$ .

**Exponential Number of Paths:** An additional advantage of this proof approach is that it substantially decreases the number of different things that you need to prove. Suppose that you have a sequence of  $n$  of these *if* statements. There would be  $2^n$  different paths that a computation might take through the code. If you had to prove separately for each of these paths that the computation works correctly, it would take you a long time. It is much easier to prove the above two statements for each of the  $n$  *if* statements.

**Proving the Correctness of a *loop* Statement:** Loops are another construct that must be proven to be correct.

**Structure of Algorithmic Fragment:**

```
 $\langle pre-loop-cond \rangle$ 
loop
   $\langle loop-invariant \rangle$ 
  exit when  $\langle exit-cond \rangle$ 
  codeloop
end loop
 $\langle post-loop-cond \rangle$ 
```

**Steps in Proving Correctness:** Its correctness  $\langle pre-loop-cond \rangle$  &  $code \Rightarrow \langle post-loop-cond \rangle$  is proved by breaking a path through the code into subpaths and proving that each of these parts works correctly, namely

- $\langle pre-loop-cond \rangle \Rightarrow \langle loop-invariant \rangle$
- $\langle loop-invariant' \rangle \& \text{not } \langle exit-cond \rangle \& \text{code}_{loop} \Rightarrow \langle loop-invariant'' \rangle$
- $\langle loop-invariant \rangle \& \langle exit-cond \rangle \Rightarrow \langle post-loop-cond \rangle$
- Termination (or even better, giving a bound on the running time).

**Infinite Number of Paths:** Depending on the number of times around the loop, this code has an infinite number of possible paths through it. We, however, are able to prove all at once that each iteration of the loop works correctly.

This technique is discussed more in the next section.

**Proving the Correctness of a Function Call:** Function and subroutine calls are also a key algorithmic technique that must be proven correct.

**Structure of Algorithmic Fragment:**

```
 $\langle pre-call-cond \rangle$ 
output = RoutineCall( input )
 $\langle post-call-cond \rangle$ 
```

**Steps in Proving Correctness:** Its correctness  $\langle pre-call-cond \rangle$  &  $code \Rightarrow \langle post-call-cond \rangle$  is proved by ensuring that the input instance passed to the routine meets the routine's precondition, trusting that the routine ensures that its output meets its postconditions, and ensuring that this postcondition meets the needs of the algorithm fragment.

- $\langle pre-call-cond \rangle \Rightarrow \langle pre-cond-cond \rangle_{RoutineCall}$
- $\langle post-cond-cond \rangle_{RoutineCall} \Rightarrow \langle post-call-cond \rangle$

This technique is discussed more in Chapter 14 on recursion.

### 8.5.2 Proving Correctness of Iterative Algorithms with Induction

Section 8.4 describes the *loop invariant* level of abstraction and what is needed to develop an algorithm within it. The next step is to use induction to prove that this process produces working programs.

**Structure of An Iterative Algorithm:**

```

⟨pre-cond⟩
codepre-loop      % Establish loop invariant
loop
    ⟨loop-invariant⟩
    exit when ⟨exit-cond⟩
    codeloop      % Make progress while maintaining the loop invariant
end loop
codepost-loop      % Clean up loose ends
⟨post-cond⟩

```

### Steps in Proving Correctness:

- $\langle \text{pre-cond} \rangle \& \text{code}_{\text{pre-loop}} \Rightarrow \langle \text{loop-invariant} \rangle$
- $\langle \text{loop-invariant}' \rangle \& \text{not } \langle \text{exit-cond} \rangle \& \text{code}_{\text{loop}} \Rightarrow \langle \text{loop-invariant}'' \rangle$
- $\langle \text{loop-invariant} \rangle \& \langle \text{exit-cond} \rangle \& \text{code}_{\text{post-loop}} \Rightarrow \langle \text{post-cond} \rangle$
- Termination (or even better, giving a bound on the running time).

**Mathematical Induction:** Induction is an extremely important mathematical technique for proving statements with a universal quantifier. (See Section 1.)

**A Statement for Each  $n$ :** For each value of  $n \geq 0$ , let  $S(n)$  represent a boolean statement.  
This statement may be true for some values of  $n$  and false for others.

**Goal:** The goal is to prove that for every value of  $n$  the statement is true, namely that  $\forall n \geq 0, S(n)$ .

**Proof Outline:** The proof is by induction on  $n$ .

**Induction Hypothesis:** The first step is to state the induction hypothesis clearly:  
“For each  $n \geq 0$ , let  $S(n)$  be the statement that ...”.

**Base Case:** Prove that the statement  $S(0)$  is true.

**Induction Step:** For each  $n \geq 1$ , prove  $S(n-1) \Rightarrow S(n)$ . The method is to assume that  $S(n-1)$  is true and then to prove that it follows that  $S(n)$  must also be true.

**Conclusion:** By way of induction, you can conclude that  $\forall n \geq 0, S(n)$ .

**Types:** Mind the “type” of everything (as you do when writing a program).  $n$  is an integer, not something to be proved. Do not say “Assume  $n-1$  and prove  $n$ .” Instead, say “Assume  $S(n-1)$  and prove  $S(n)$ ” or “Assume that it is true for  $n-1$  and prove that it is true for  $n$ .”

### The “Process” of Induction:

```

S(0) is true      (by base case)
S(0) ⇒ S(1)      (by induction step, n = 1)
    hence, S(1) is true
S(1) ⇒ S(2)      (by induction step, n = 2)
    hence, S(2) is true
S(2) ⇒ S(3)      (by induction step, n = 3)
    hence, S(3) is true
...

```

## The Connection between Loop Invariants and Induction:

**Induction Hypothesis:** For each  $n \geq 0$ , let  $S(n)$  be the statement, “If the loop has not yet exited, then the loop invariant is true when you are at the top of the loop after going around  $n$  times.”

**Goal:** The goal is to prove that  $\forall n \geq 0, S(n)$ , namely that “As long as the loop has not yet exited, the loop invariant is always true when you are at the top of the loop.”

**Proof Outline:** Proof by induction on  $n$ .

**Base Case:** Proving  $S(0)$  involves proving that the loop invariant is true when the algorithm first gets to the top of the loop. This is achieved by proving the statement  $\langle pre-cond \rangle \& code_{pre-loop} \Rightarrow \langle loop-invariant \rangle$ .

**Induction Step:** Proving  $S(n-1) \Rightarrow S(n)$  involves proving that the loop invariant is maintained. This is achieved by proving the statement  $\langle loop-invariant' \rangle \& \neg \langle exit-cond \rangle \& code_{loop} \Rightarrow \langle loop-invariant'' \rangle$ .

**Conclusion:** By way of induction, we can conclude that  $\forall n \geq 0, S(n)$ , i.e., that the loop invariant is always true when at the top of the loop.

**Proving that the Iterative Algorithm Works:** To prove that the iterative algorithm works correctly on every input instance, you must prove that, for an arbitrary instance,  $\langle pre-cond \rangle \& code_{alg} \Rightarrow \langle post-cond \rangle$ .

Consider some instance. Assume that this instance meets the preconditions. Proof by induction proves that as long as the loop has not yet exited, the loop invariant is always true.

The algorithm does not work correctly if it executes forever. Hence, you must prove that the loop eventually exits. According to the loop invariant level of abstraction, the algorithm must make progress of at least one at each iteration. It also gives an upper bound on the amount of progress that can be made. Hence, you know that after this number of iterations at most, the exit condition will be met.

Thus, you also know that, at some point during the computation, both the loop invariant and the exit condition will be simultaneously met. You then use the fact that  $\langle loop-invariant \rangle \& \langle exit-cond \rangle \& code_{post-loop} \Rightarrow \langle post-cond \rangle$  to prove that the postconditions will be met at the end of the algorithm.

# Chapter 9

## More of The Input Loop Invariant

We are now ready to give more examples of iterative algorithms. For each example, look for the key steps of the loop invariant paradigm. What is the loop invariant? How is it obtained and maintained? What is the measure of progress? How is the correct final answer ensured?

In this chapter, we will include those algorithms that use the “more of the input” type of loop invariant. The algorithm reads the  $n$  objects making up the input one at a time. After reading the first  $i$  of them, the algorithm temporarily pretends that this prefix of the input is in fact the entire input. The loop invariant is “I currently have a solution for the input consisting solely of these first  $i$  objects (and maybe some additional information).”

### 9.1 Colouring the Plane

**Specification:** An input instance consists of a set of  $n$  (infinitely long) lines. These lines form a subdivision of the plane, that is, they partition the plane into a finite number of regions (some of them unbounded). The output consists of a colouring of each region with either black or white so that any two regions with a common boundary have different colours.

**Start with Small Steps:** When an instance consists of a set of objects, a common technique is to consider them one at a time, incrementally solving the problem for those objects considered so far.

**The Loop Invariant:** We have considered the first  $i$  lines.  $C$  is a proper colouring of the plane subdivided by these lines.

**More of the Input:** This is a classic “more of the input” loop invariant because the algorithm maintains a solution for the first  $i$  input lines.

**The Measure of Progress:** Clearly the progress is made when another line is considered.

**Main Steps:** We have a proper colouring  $C$  for the first  $i$  lines. Line  $i + 1$  cuts the plane in half, cutting many regions in half. Each of these halves needs a different colour. Then when we change the colour of one region, its neighbors must change colour too. We will accomplish this by keeping all the colours on one side of line  $i + 1$  the same and flipping those on the other side from white to black and from black to white.

**Maintaining the Loop Invariant:**  $\langle \text{loop-invariant}' \rangle \& \text{not } \langle \text{exit-cond} \rangle \& \text{code}_{\text{loop}} \Rightarrow \langle \text{loop-invariant}'' \rangle$ . We need to check each boundary to make sure that the regions

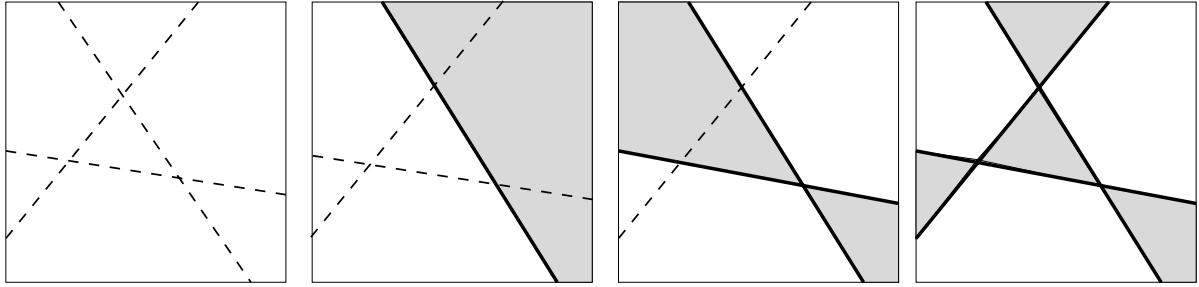


Figure 9.1: An example of colouring the plane.

on either side have opposite colours. Boundaries formed by one of the first  $i$  lines had opposite colours before the change. The colours of the regions on either side were either neither flipped or both flipped. Hence, they still have opposite colours. Boundaries formed by line  $i + 1$  had the same colours before the change. One of these colours was flipped, the other not. Hence, they now have opposite colours.

**Initial Code:**  $\langle \text{pre-cond} \rangle \& \text{code}_{\text{pre-loop}} \Rightarrow \langle \text{loop-invariant} \rangle$ . With  $i = 0$  lines, the plane is all one region. The colouring that makes the entire plane white works.

**Exiting Loop:**  $\langle \text{loop-invariant} \rangle \& \langle \text{exit-cond} \rangle \& \text{code}_{\text{post-loop}} \Rightarrow \langle \text{post-cond} \rangle$ . If  $C$  is a proper colouring given the first  $i$  lines and  $i = n$ , then clearly  $C$  is a proper colouring given all of the lines.

**Code:**

**algorithm** *ColouringPlane(lines)*

**$\langle \text{pre-cond} \rangle$ :** *lines* specifies  $n$  (infinitely long) lines.

**$\langle \text{post-cond} \rangle$ :**  $C$  is a proper colouring of the plane subdivided by the lines.

begin

$C$  = “the colouring that colours the entire plane white.”

$i = 0$

    loop

**$\langle \text{loop-invariant} \rangle$ :**  $C$  is a proper colouring of the plane subdivided by the first  $i$  lines.

        exit when ( $i = n$ )

        % Make progress while maintaining the loop invariant

        Line  $i + 1$  cuts the plane in half.

        On one half, the new colouring  $C'$  is the same as the old one  $C$ .

        On the other half, the new colouring  $C'$  is the same as the old one  $C$ , except white is switched to black and black to white.

$i = i + 1 \& C = C'$

    end loop

    return( $C$ )

end algorithm

## 9.2 Deterministic Finite Automaton

One large class of problems that can be solved using an iterative algorithm with the help of a loop invariant is the class of *regular languages*. You may have learned that this is the class of languages that can be decided by a Deterministic Finite Automata (DFA) or described using a regular expression.

**Examples:** This class is useful for modeling

- simple iterative algorithms
- simple mechanical or electronic devices like elevators and calculators
- simple processes like the job queue of an operating system
- simple patterns within strings of characters.

**Similar Features:** All of these have the following similar features.

**Input Stream:** They receive a stream of information to which they must react. For example, the stream of input for a simple algorithm consists of the characters read from input; for a calculator, it is the sequence of buttons pushed; for the job queue, it is the stream of jobs arriving; and for the pattern within a string, one scans the string once from left to right.

**Read Once Input:** Once a token of the information has arrived it cannot be requested for again.

**Bounded Memory:** The algorithm/device/process/pattern matcher has limited memory with which to remember the information that it has seen so far. Though the amount of memory can be any fixed amount, the key is that this amount can not grow if the input instance becomes really big.

**Example 9.2.1:** Given a string  $\alpha$ , determine whether it is contained in the set (*language*)  
 $L = \{\alpha \in \{0,1\}^* \mid \alpha \text{ has length at most three and the number of 1's is odd}\}$ .

**Ingredients of An Iterative Algorithm:**

**The Loop Invariant:** Recall the *More of the Input* type of iterative algorithms (see Section 8.4). Each iteration of the algorithm reads in the next character of the input string. Suppose that you have already read in some large prefix of the complete input. With bounded memory you cannot remember everything you have read. However, you will never be able to go back and read it again. Hence, you must remember a few key facts about the prefix read. The loop invariant states what information is remembered.

In Example 9.2.1, the most obvious thing to remember about it would be length and the number of 1's read so far. However, with a large input these counts can grow arbitrarily large. Hence, with only bounded memory you cannot remember them. Luckily, the language is only concerned with this length up to four and whether the number of 1's is even or odd. This can be done with two variables: length,  $l \in \{0, 1, 2, 3, \text{more}\}$ , and parity,  $r \in \{\text{even}, \text{odd}\}$ . This requires only a fixed amount of memory.

**More of the Input:** This is a classic “more of the input” loop invariant because the algorithm maintains a solution and some additional information for the prefix consisting of the first  $i$  characters of the input instance.

**Maintaining Loop Invariant:** Let  $\omega$  denote the prefix of the input string read so far. You do not know all of  $\omega$ , but by the loop invariant you know something about it. Now suppose you read another character  $c$ . What you have read now is  $\omega c$ . What must you remember about  $\omega c$  in order to meet the loop invariant? Is what you know about  $\omega$  and  $c$  enough to know what you need to know about  $\omega c$ ?

In Example 9.2.1, if we know the length  $l \in \{0, 1, 2, 3, \text{more}\}$  of the prefix  $\omega$  and whether the number of 1's in it is  $r \in \{\text{even}, \text{odd}\}$ , then it is easy to know that the length of  $\omega c$  is one more and the number of 1's is either one more mod 2 or the same depending on whether or not the new character  $c$  is a 1 or not.

**Initial Conditions:** At the beginning of the computation, the prefix that has been read so far is the empty string  $\omega = \epsilon$ . Which values should the state variables have to establish the loop invariant?

In Example 9.2.1, the length of  $\omega = \epsilon$  is  $l = 0$  and the number of 1's is  $r = \text{even}$ .

**Ending:** When the input string has been completely read in, the knowledge that your loop invariant states that you know must be sufficient for you to now compute the final answer. The code outside the loop does this.

#### Code of Example 9.2.1:

**algorithm** *DFA()*

***⟨pre-cond⟩:*** The input string  $\alpha$  will be read in one character at a time.

***⟨post-cond⟩:*** The string will be *accepted* if it has length at most three and the number of 1's is odd.

begin

$l = 0$  and  $r = \text{even}$

loop

***⟨loop-invariant⟩:*** When the iterative program has read in some prefix  $\omega$  of the input string  $\alpha$ , the bounded memory of the machine remembers the length  $l \in \{0, 1, 2, 3, \text{more}\}$  of this prefix and whether the number of 1's in it is  $r \in \{\text{even}, \text{odd}\}$ .

exit when end of input

get( $c$ ) % Reads next character of input

if( $l < 4$ ) then  $l = l + 1$

if( $c = 1$ ) then  $r = r + 1 \bmod 2$

end loop

if( $l < 4$  AND  $r = \text{odd}$ ) then

accept

else

reject

end if

end algorithm

**Mechanically Compiling an Iterative Program into a DFA:** Any iterative program with bounded memory and an input stream can be mechanically compiled into a DFA that solves the same problem. This provides another model or notation for understanding the algorithm. The DFA for Example 9.2.1 is represented by the graph in Figure 9.2.

A DFA is specified by  $M = \langle \Sigma, Q, \delta, s, F \rangle$ .

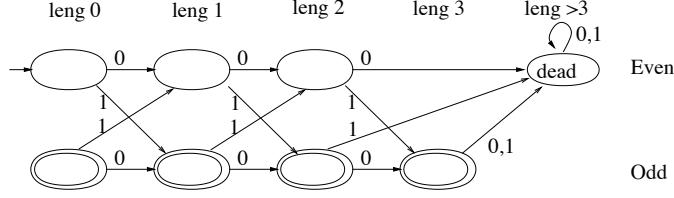


Figure 9.2: The graphical representation of the DFA corresponding to the iterative program for Example 9.2.1.

**Alphabet  $\Sigma$ :** Here  $\Sigma$  specifies the alphabet of characters that might appear in the input string. This may be  $\{0, 1\}$ ,  $\{a, b\}$ ,  $\{a, b, \dots, z\}$ , ASCII, or any other finite set of tokens that the program may input.

**Set of States  $Q$ :**  $Q$  specifies the set of different states that the iterative program might be in when at the top of the loop. Each state  $q \in Q$  specifies a value for each of the program's variables. If the variables are allocated in total  $r$  bits of memory, then they can hold  $2^r$  different values. Conversely, with  $|Q|$  states a DFA can “remember”  $\log_2 |Q|$  bits of information. In the graph representation of a DFA, there is a node for each state. Instead of assigning the names  $q_0, q_1, \dots, q_{|Q|}$  to the states, assign names with more meaning.

**Transition Function  $\delta$ :** The DFA’s transition function  $\delta$  defines how the machine transitions from state to state. Formally, it is a function  $\delta : Q \times \Sigma \rightarrow Q$ . If the DFA’s current state is  $q \in Q$  and the next input character is  $c \in \Sigma$ , then the next state of the DFA is given by  $q' = \delta(q, c)$ . Consider some state  $q \in Q$  and some character  $c \in \Sigma$ . Set the program’s variables to the values corresponding to state  $q$ , assume the character read is  $c$ , and execute the code once around the loop. The new state  $q' = \delta(q, c)$  of the DFA is defined to be the state corresponding to the values of the program’s variables when the computation has reached the top of the loop again. In a graph representation of a DFA, for each state  $q$  and character  $c$ , there is an edge labeled  $c$  from node  $q$  to node  $q' = \delta(q, c)$ .

**The Start State  $s$ :** The start state  $s$  of the DFA  $M$  is the state in  $Q$  corresponding to the initial values that the program assigns to its variables before reading any input characters. In the graph representation, the corresponding node has an arrow to it.

**Accept States  $F$ :** A state in  $Q$  will be considered an *accept* state of the DFA  $M$  if it corresponds to a setting of the variables for which the program accepts. In Figure 9.2, these nodes are denoted by double circles.

**Adding:** We will use the standard elementary school algorithm. The input consists of two integers  $x$  and  $y$  represented as strings of digits. The output is the sum  $z$  also represented as a string of digits. The input can be viewed as a stream if the algorithm is first given the lowest digits of  $x$  and of  $y$ , then the second lowest, and so on. The algorithm outputs the characters of  $z$  as it proceeds. The only memory required is a single bit to store the carry bit. Because of these features, the algorithm can be modeled as a DFA.

**algorithm** *Adding()*

**$\langle pre-cond \rangle$ :** The digits of two integers  $x$  and  $y$  are read in backwards in parallel.

**$\langle post-cond \rangle$ :** The digits of their sum will be outputted backwards.

```

begin
    allocate carry  $\in \{0, 1\}$ 
    carry = 0
    loop
         $\langle loop-invariant \rangle$ : If the low order  $i$  digits of  $x$  and of  $y$  have been read,
        then the low order  $i$  digits of the sum  $z = x + y$  have been outputted.
        The bounded memory of the machine remembers the carry.
        exit when end of input
        get( $\langle x_i, y_i \rangle$ )
         $s = x_i + y_i + carry$ 
         $z_i$  = low order digit of  $s$ 
         $carry$  = high order digit of  $s$ 
        put( $z_i$ )
    end loop
    if( $carry = 1$ ) then
        put( $carry$ )
    end if
end algorithm

```

The DFA is as follows.

**Set of States:**  $Q = \{q_{\langle carry=0 \rangle}, q_{\langle carry=1 \rangle}\}$ .

**Alphabet:**  $\Sigma = \{\langle x_i, y_i \rangle \mid x_i, y_i \in [0..9]\}$ .

**Start state:**  $s = q_{\langle carry=0 \rangle}$ .

**Transition Function:**  $\delta(q_{\langle carry=c \rangle}, \langle x_i, y_i \rangle) = \langle q_{carry=c'}, z_i \rangle$

where  $c'$  is the high order digit and  $z_i$  is the low order digit of  $x_i + y_i + c$ .

**Dividing:** Dividing an integer by seven is reasonably complex algorithm. Surprisingly, it can be done by a DFA. Consider the language  $L = \{w \in \{0, 1, \dots, 9\}^* \mid w \text{ is divisible by 7 when viewed as an integer in normal decimal notation}\}$ . Recall that standard elementary school algorithm to divide an integer by 7 considers the digits one at a time. Try it yourself on say 39592. We do not care about how many times 7 divides into our number, but only whether or not it divides evenly. Hence, when considering the prefix 395, we remember only that its remainder is 3. The next step computes  $3959 \bmod 7 = (395 \cdot 10 + 9) \bmod 7 = ((395 \bmod 7) \cdot 10 + (9 \bmod 7)) \bmod 7 = ((3) \cdot 10 + (2)) \bmod 7 = 32 \bmod 7 = 4$ . More generally, suppose that we have read in the prefix  $\omega$ . We store a value  $r \in \{0, 1, \dots, 6\}$  and maintain the loop invariant that  $r = \omega \bmod 7$ , when the string  $\omega$  is viewed as an integer. Now suppose that the next character is  $c \in \{0, 1, \dots, 9\}$ . The current string is then  $\omega c$ . We must compute  $\omega c \bmod 7$  and set  $r$  to this new remainder in order to maintain the loop invariant. The integer  $\omega c$  is  $(\omega \cdot 10 + c)$ . Hence, we can compute  $r = \omega c \bmod 7 = (\omega \cdot 10 + c) \bmod 7 = (\omega \bmod 7) \cdot 10 + c \bmod 7 = r \cdot 10 + c \bmod 7$ . The code for the loop is simply  $r = r \cdot 10 + c \bmod 7$ . Initially, the prefix read so far is the empty string. The empty string viewed as an integer is 0. Hence, the initial setting is  $r = 0$ . In the end, we accept the string if when viewed as an integer it is divisible by 7. This is true when  $r = 0$ . This completes the development of the iterative program. The DFA to compute this will have seven states  $q_0, \dots, q_6$ . The transition function is  $\delta(q_r, c) = q_{\langle r \cdot 10 + c \bmod 7 \rangle}$ . The start state is  $s = q_0$ . The set of accept states is  $F = \{q_0\}$ .

**Calculator:** Invariants can be used to understand a computer system that, instead of simply computing one function, continues dynamically to take in inputs and produce outputs. See Chapter 8.1 for further discussion of such systems.

In a simple calculator, the keys are limited to  $\Sigma = \{0, 1, 2, \dots, 9, +, clr\}$ . You can enter a number. As you do so it appears on the screen. The  $+$  key adds the number on the screen to the accumulated sum and displays the sum on the screen. The  $clr$  key resets both the screen and the accumulator to zero. The machine only can store positive integers from zero to 99999999. Additions are done mod  $10^8$ .

**algorithm** *Calculator()*

***⟨pre-cond⟩:*** A stream of commands are entered.

***⟨post-cond⟩:*** The results are displayed on a screen.

begin

    allocate *accum, current*  $\in \{0..10^8 - 1\}$

    allocate *screen*  $\in \{showA, showC\}$

*accum* = *current* = 0

*screen* = *showC*

    loop

***⟨loop-invariant⟩:*** The bounded memory of the machine remembers the current value of the accumulator and the current value being entered. It also has a boolean variable which indicates whether the screen should display the current or the accumulator value.

        get(*c*)

        if( *c*  $\in \{0..9\}$  ) then

*current* =  $10 \times current + c \text{ mod } 10^8$

*screen* = *showC*

        else if( *c* = ' + ' ) then

*accum* = *accum* + *current* mod  $10^8$

*current* = 0

*screen* = *showA*

        else if( *c* = ' clr' ) then

*accum* = 0

*current* = 0

*screen* = *showC*

        end if

        if( *screen* = *showC* ) then

            display(*current*)

        else

            display(*accum*)

        end if

    end loop

end algorithm

The input is the stream of keys that the user presses. It uses only bounded memory to store the eight digits of the accumulator and of the current value and the extra bit. Because of these features, the algorithm can be modeled as a DFA.

**Set of States:**  $Q = \{q_{\langle acc, cur, scr \rangle} \mid acc, cur \in \{0..10^8 - 1\} \text{ and } scr \in \{showA, showC\}\}$ .  
 Note that there are  $10^8 \times 10^8 \times 2$  states in this set so you would not want to draw the diagram.

**Alphabet:**  $\Sigma = \{0, 1, 2, \dots, 9, +, clr\}$ .

**Start state:**  $s = q_{\langle 0, 0, showC \rangle}$ .

**Transition Function:**

- For  $c \in \{0..9\}$ ,  $\delta(q_{\langle acc, cur, scr \rangle}, c) = q_{\langle acc, 10 \times cur + c, showC \rangle}$ .
- $\delta(q_{\langle acc, cur, scr \rangle}, +) = q_{\langle acc + cur, cur, showA \rangle}$ .
- $\delta(q_{\langle acc, cur, scr \rangle}, clr) = q_{\langle 0, 0, showC \rangle}$ .

**Longest Block of Ones:** Suppose that the input consists of a sequence  $A[1..n]$  of zeros and ones and we want to find a longest contiguous block  $A[p, q]$  of ones. For example, on input  $A[1..n] = [1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0]$ , the block  $A[5..7]$  of length 3 is a suitable solution and so is the block  $A[10..12]$ , but  $A[1..2]$  is not.

**Practice with Loop Invariants:** It is an easy piece of code to write, but half the class got the loop invariant wrong on a midterm.

**Non-Finite Memory:** Both the size of the longest block and the indices of its beginning and end are integers in  $[1..n]$ . These require  $\mathcal{O}(\log n)$  bits to remember. Hence, this algorithm will not be a Deterministic Finite Automata because it stores more than a constant amount of information at each point in time. However, we are including it in this section in order to see some of the issues that arise.

**Remember the Solution for the Prefix:** After reading the prefix  $A[1..i]$ , it is clear that you need to remember the longest block of ones read so far and its size. Many students have stated that this and this alone is the loop invariant. Is this enough? How would you maintain this loop invariant when reading in only the next character  $A[i+1]$ . For example if  $A[1..i] = [0, 1, 1, 0, 0, 1, 1]$ , then the loop invariant may give us only the block  $A[2..3]$  of length 2. Then if we read  $A[i+1] = 1$ , then the longest block of  $A[1..i+1] = [0, 1, 1, 0, 0, 1, 1, 1]$  becomes  $A[6..8]$  of length 3. How would your program know about this block?

**Remember the Longest Current Block:** When this is pointed out to students, they then say that the algorithm keeps a pointer to the beginning of the current block being worked on, i.e. the longest one ending in the value  $A[i]$ , and its size. With this the algorithm can know whether the current increasing contiguous subsequence gets to be longer than the previous one. My response to this student is that this then needs to be included in the loop invariant.

**Maintaining the Loop Invariant:** If you have this information about  $A[1..i]$ , then you can learn it about  $A[1..i+1]$  as follows. If  $A[i+1] = 1$ , then the longest block of ones ending in the current value increases in length by one. Otherwise, it shrinks to being the empty string. If this block increases to be longer than our previous longest, then it replaces the previous longest. In the end, we know the longest block of ones.

**Code:**

```
algorithm LongestBlockOfOnes(A, n)
```

***⟨pre-cond⟩:*** The input is  $A$ , a 0, 1-array of length  $n$ .

***⟨post-cond⟩:*** The output is the location  $A[k_1..k_2]$  of the longest block of ones and its length  $leng$ .

begin

```

 $i = 0; p_{max} = 1; q_{max} = 0; leng_{max} = 0; p_{current} = 1; leng_{current} = 0 \in \{0..n\}$ 
loop   ⟨loop-invariant⟩:  $A[p_{max}, q_{max}]$  is a longest block of ones in  $A[1..i]$  and
        $leng_{max} = q_{max} - p_{max} + 1$  is its length.
        $A[p_{current}, i]$  is the longest block of ones in  $A[1..i]$  ending in  $A[i]$  and
        $leng_{current} = i - p_{current} + 1$  is its length.
if(  $A[i+1] = 1$  ) then
     $leng_{current} = leng_{current} + 1$ 
else
     $p_{current} = i + 2$ 
     $length_{current} = 0$ 
end if
if(  $leng_{max} < leng_{current}$  ) then
     $p_{max} = p_{current}$ 
     $q_{max} = i + 1$ 
     $length_{max} = length_{current}$ 
end if
 $i = i + 1$ 
end loop
end algorithm
```

**Empty Blocks:** Note that  $A[3..3]$  is a block of length 1 and  $A[4..3]$  is a block of length zero ending in  $A[3]$ . This is why initially with  $i = 0$ , the blocks are set to  $A[1..0]$  and when the current block ending in  $A[i+1]$  becomes empty, it is set to  $A[i+2..i+1]$ .

**Dynamic Programming:** Dynamic Programming, covered in Section 23, is a very powerful technique for solving optimization problems. Many of these amount to reading the elements of the input instance  $A[1..n]$  one at a time and when at  $A[i]$  saving the optimal solution for the prefix  $A[1..i]$  and its cost. This amounts to a Deterministic Non-Finite Automata. The maximum block of ones problem above is a trivial example of this. The solutions to the following two problems and more problems can be found in Section 24.3.

**Longest Increasing Contiguous SubSequence:** The input consists of a sequence  $A[1..n]$  of integers and we want to find the longest contiguous subsequence  $A[k_1..k_2]$  such that the elements are monotonically increasing. For example, the optimal solution for  $[5, 3, 1, 3, 7, 9, 8]$  is  $[1, 3, 7, 9]$ .

**Longest Increasing SubSequence:** The following is a harder problem. Again the input consists of a sequence  $A$  of integers of size  $n$ . However now we want to find the longest (not necessarily contiguous) subsequence  $S \subseteq [1..n]$  such that the elements, in the order that they appear in  $A$ , are monotonically increasing. For example, an optimal solution for  $[5, 1, 5, 7, 2, 4, 9, 8]$  is  $[1, 5, 7, 9]$  and so is  $[1, 2, 4, 8]$ .

### 9.3 More of the Input vs More of the Output

I find that when given a problem, students often want to solve it using the “more of the output” type of loop invariants. For example, consider the following problem.

**Exercise 9.3.1** *A tournament is a directed graph formed by taking the complete undirected graph and assigning arbitrary directions on the edges, i.e., a graph  $G = (V, E)$  such that for each  $u, v \in V$ , exactly one of  $\langle u, v \rangle$  or  $\langle v, u \rangle$  is in  $E$ . A Hamiltonian path is a path through a graph that can start and finish anywhere but must visit every node exactly once each. Design an algorithm which given a tournament finds a Hamiltonian path through it. Note that because this algorithm finds a hamiltonian path for each tournament, this algorithm ,in itself, acts as proof that that every tournament has a Hamiltonian path.*

The output of this problem is a path. A “more of the output” loop invariant would say “I have the first  $i$  nodes (edges) in the final path.” Maintaining this loop invariant would require finding the next node in the path. This requires extending the path constructed so far by one more node. However, the algorithm might get stuck, when the path constructed so far has no edges leaving the last node to a node that has not yet been visited. The student is tempted to have the algorithm “back track” when it gets stuck and try in a initial different direction for the path to go. This is a fine algorithm. See recursive back tracking algorithms is Chapter 23. However, such algorithms, unless one is really careful, tend to be exponential time.

Instead, try solving this problem using a “more of the input” loop invariant. Assume the nodes are numbered 1 to  $n$  in an arbitrary way. The algorithm temporarily pretends that the sub-graph on the first  $i$  of the nodes is the entire input instance. The loop invariant is “I currently have a solution for this sub-instance.” Such a solution is a hamiltonian path that visits each of the first  $i$  nodes exactly once each, which in turn is simply a permutation the first  $i$  nodes. Maintaining this loop invariant requires constructing a path for the first  $i + 1$  nodes. There is no requirement that this new path resembles the previous path. However, in fact, it can be accomplished by finding a place to insert the  $i + 1^{st}$  node within the permutation of the first  $i$  nodes. In this way, the algorithm looks like insertion sort.

In contrast, this next problem is solved with an algorithm using something that looks a lot more like a “more of the input” algorithm.

**Exercise 9.3.2 Euler Tour problem:** *An Euler tour in an undirected graph is a cycle that passes through each edge exactly once. A graph contains an Eulerian cycle iff it is connected and the degree of each vertex is even. Give an  $\mathcal{O}(|E|)$  algorithm to find an Eulerian cycle if one exists.*

Try building the path one edge at a time in a blind or greedy way, hoping for best. When do you get stuck? When you do get stuck, how can you fix it?

This problem is searching for middle number within  $2n$  numbers. If one could do binary search, cutting the search space in half each iteration, then it would take only  $\log_2(2n)$  iterations. This can be achieved. Try comparing  $S[i]$  and  $T[n - i]$ .

# Chapter 10

# Implementations of Abstract Data Types

This chapter will implement some of the abstract data types (ADTs) listed in Section 7.2. From the user's perspective, these consist of a data structure and a set of operations with which to access the data. From the perspective of the data structure itself, it is an ongoing system that continues to receive a stream of commands to which it must dynamically react. As described in Section 8.1, abstract data types have both public invariants that any outside user of the system needs to know about and hidden invariants that only the system's designers need to know about and maintain. Section 7.2 focused on the first. This chapter focuses on the second. These consist of a set integrity constraints or assertions that must be true every time the system is entered or left. Imagining a big loop around the system, with one iteration for each interaction the system has with the outside world, motivates us to include them as an example of loop invariants.

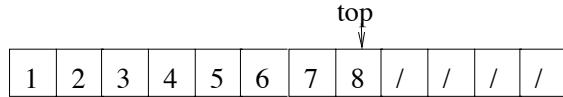
Lists, stacks, and queues are extremely useful. We will give both an array and a linked list implementation and an application of each. Priority Queues will be implemented in Section 16.4. Here, we will also give implementations of graphs, trees, and sets.

## 10.1 Array Implementations

### Stack Implementation:

**Stack Specifications:** Recall that a stack is analogous to a stack of plates, in which a plate can be added or removed only at the top. The public invariant is that the order in which the elements were added to the stack is maintained. The only operations are *pushing* a new element onto the top of the stack and *popping* the top element off the stack. This is referred to as *Last-In-First-Out* (LIFO).

**Hidden Invariants:** The hidden invariants in an array implementation of a stack are that the elements in the stack are stored in an array starting with the bottom of the stack and that a variable *top* indexes the entry of the array containing the top element. When the stack is empty, *top* = 0 if the array indexes from 1 and *top* = -1 if it indexes from 0. With these pictures in mind, it is not difficult to implement push and pop. The stack grows to the right as elements are pushed and shrinks to the left as elements are popped.



**Code:**

```
algorithm Push(newElement)
```

*<pre-cond>*: This algorithm implicitly acts on some stack ADT via *top*.

*newElement* is the information for a new element.

*<post-cond>*: The new element is pushed onto the top of the stack.

```
begin
```

```
    if( top ≥ MAX ) then
```

```
        put "Stack is full"
```

```
    else
```

```
        top = top + 1
```

```
        A[top] = newElement
```

```
    end if
```

```
end algorithm
```

```
algorithm Pop()
```

*<pre-cond>*: This algorithm implicitly acts on some stack ADT via *top*.

*<post-cond>*: The top element is removed and its information returned.

```
begin
```

```
    if( top ≤ 0 ) then
```

```
        put "Stack is empty"
```

```
    else
```

```
        element = A[top]
```

```
        top = top - 1
```

```
        return(element)
```

```
    end if
```

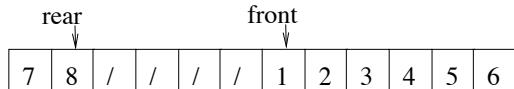
```
end algorithm
```

## Queue Implementation:

**Queue Specifications:** The queue, in contrast to a stack, is only able to remove the element that has been in the queue the longest. Elements are added at the *rear* and are removed from the *front*. This is referred to as *First-In-First-Out* (FIFO).

**Trying Small Steps:** If the front element is always stored at index 1 of the array, then when the current front is removed all the remaining elements would need to shift by one to take its place. To save time, once an element is placed in the array, we do not want to move it until it is removed. The effect is that the rear moves to the right as elements arrive and the front moves to the right as elements are removed. We use two different variables, *front* and *rear* to index their location. As the queue migrates to the right, eventually it will reach the end of the array. To avoid getting stuck, we will treat the array as a circle, indexing modulo the size of the array. This allows the queue to migrate around and around as elements arrive and leave.

**Hidden Invariants:** The elements are stored in order from the entry indexed by *front* to that indexed by *rear* possibly wrapping around the end of the array.



**Extremes:** It turns out that the cases of a completely empty and a completely full queue are indistinguishable because with both *front* will be one to the left of *rear*. The easiest solution is not to let the queue get completely full.

**Code:**

**algorithm** *Add(newElement)*

***pre-cond*:** This algorithm implicitly acts on some queue ADT via *front* and *rear*.  
*newElement* is the information for a new element.

***post-cond*:** The new element is added to the rear of the queue.

begin

if( *rear* = *front* - 2 mod *MAX* ) then

put "Queue is full"

else

*rear* = (*rear* mod *MAX*) + 1

*A*[*rear*] = *newElement*

end if

end algorithm

**algorithm** *Remove()*

***pre-cond*:** This algorithm implicitly acts on some queue ADT via *front* and *rear*.

***post-cond*:** The front element is removed and its information returned.

begin

if( *rear* = *front* - 1 mod *MAX* ) then

put "Stack is empty"

else

*element* = *A*[*front*]

*front* = (*front* mod *MAX*) + 1

*return(element)*

end if

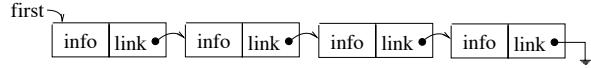
end algorithm

**Exercise 10.1.1** What is the difference between "*rear* = (*rear* + 1) mod *MAX*" and "*rear* = (*rear* mod *MAX*) + 1" and when should each be used.

Each of these operations take a constant amount of time, independent of the number of elements in the stack or queue.

## 10.2 Linked List Implementations

A problem with the array implementation is that the array needs to be allocated some fixed size of memory when the stack or queue is initialized. A *linked list* is an alternative implementation in which the memory allocated can grow and shrink dynamically with the needs of the program.



**Hidden Invariants:** In a linked list, each node contains the information for one element and a pointer to the next node in the list. The first node is pointed to by a variable that we will call *first*. The other nodes are accessed by walking down the list. The last node is distinguished by having its pointer variable contain the value zero. We say that it points to *nil*. When the list contains no nodes, the variable *first* will also point to *nil*. This is all that an implementation of a stack requires, because its list can only be accessed at one end. A queue, however, requires a pointer to the last node of the linked list.

**Pointers:** A pointer, such as *first*, is a variable that is used to store a value that is essentially an integer except that it is used to address a block of memory in which other useful memory is stored. In this application, these blocks of memory are called *nodes*. Each has two fields denoted *info* and *link*.

**Pseudo Code:** The information stored in the *info* field of the node pointed to by the pointer *first* is denoted by *first.info* in JAVA and *first->info* in C. We will adopt the first notation. Similarly, *first.link* denotes the pointer field of the node. Being a pointer itself, *first.link.info* denotes the information stored in second node of the linked list and *first.link.link.info* denotes that in the third.

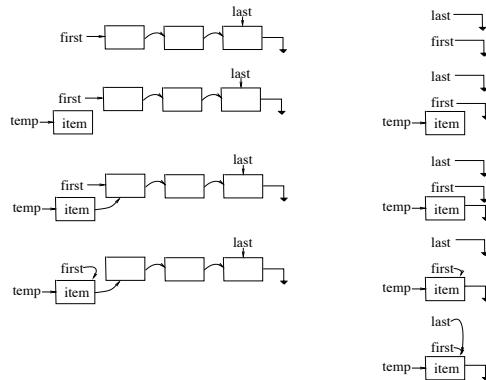
**Operations:** We want to be able to add a new node with a given value and delete a node returning its value. We want to be able to do these operations both at the front and the rear of the linked list. We will see that of these, the only difficult operation is deleting the rear node. We will also consider operations of walking the linked list and inserting a node into the middle of it.

### Adding Node to Front:

**The General Case:** We will start by designing the steps required for the general case.

- Allocate space for the new node.
- Store the information for the new element.
- Point the new node at the rest of the list.
- Point *first* at the new node.

$\text{new } temp$   
 $temp.info = item$   
 $temp.link = first$   
 $first = temp$



**Special Cases:** The main special case is an empty list. Start with both *first* and *last* pointing to *nil*. Sometimes we are lucky and the code written for the general case also works for such special cases. Here everything works except for *last*. Add the following to the bottom of the code.

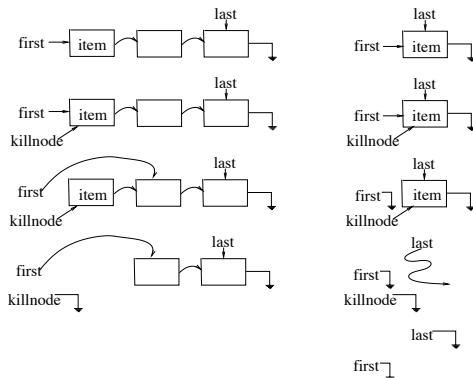
```
if( last = nil ) then
    last = temp          % Point last to the new and only node.
    end if
```

Whenever adding code to handle a special case, be sure to check that the previously handled cases still are handled.

### Removing Node From Front:

**The General Case:** Following the instructions from Section 8.4, let us start designing this algorithm by considering the steps required when we are starting with a large and a general linked list.

- We will need a temporary variable, denoted *killNode*, to point to the node to be removed.
- Move *first* to point to the second node in the list by pointing it at the node that the node it points to points to.
- Save the value to be returned.  $item = killNode.info$
- Deallocate the memory for the first node.  $free killNode$
- Return the element.  $return(item)$



**Special Cases:** If the list is already empty, a node cannot be removed. The implementer may decide to make it a precondition of the routine that the list is not empty. If the list is empty, the routine may call an exception or give an error message. The only other special case occurs when the list becomes empty. Trace through the general case code. Start with one element pointed to by both *first* and *last*. In the end, *first* points to *nil*, which is correct for an empty list. However, *last* still points to the node that has been deleted. This can be solved by adding the following to the bottom of the code.

```
if( first = nil ) then
    last = nil
end if
```

**Implementation Details:** Note that the value of *first* and *last* change. If the routine *Pop* passes these parameters in by value, the routine needs to be written to allow this to happen.

### Testing Whether Empty:

**Code:**

```
algorithm IsEmpty()
⟨pre-cond⟩: This algorithm implicitly acts on some queue ADT.
⟨post-cond⟩: The output indicates whether it is empty.
begin
    if( first = nil ) then
        return( true )
    else
        return( false )
    end if
end algorithm
```

**Information Hiding:** It does not look like this routine does much, but it serves two purposes. It hides these implementation details from the user and by calling this routine instead of doing the test directly, the users code becomes more readable.

### Adding A Node to End:

**Code:**

```
new temp
if( first = nil) then
    first = temp
else
    last.link = temp
end if
temp.info = item
temp.link = nil
last = temp
```

% Allocate space for the new node.  
% The new node is also the only node.  
% Point first at this node.  
% Link the previous last node to the new node.  
% Store the information for the new element.  
% Being the last node in the list, it needs a nil pointer.  
% Point last to the new last node.

### Removing Node From End:

**Easy Part:** By the hidden invariant of the list implementation, the pointer *last* points to the last node in the list. Because of this, it is easy to access the information in this node and to deallocate the memory for this node.

**Hard Part:** In order to maintain this invariant, when the routine returns, the variable *last* must be pointing at the last node in the list. This had been the second last node. The difficulty is that our only access to this node is to start at the front of the list and *walk* all the way down the list. This would take  $\Theta(n)$  time instead of  $\Theta(1)$  time.

**Exercise 10.2.1 Double Pointers:** *Describe how this operation can be done in  $\Theta(1)$  time if there are pointers in each node both the previous and the next node.*

**Stacks and Queues:** Neither stacks nor queues need this operation if the first node is used as the top of the stack and the front of the queue.

**Walking Down the Linked List:** To find a node (or a space between nodes) that is not at the front or at the end of the list, we will have a loop to step down the list. If we

are inserting a node, then the goal is obtain a pointer to the node before and another to the node after the desired location. If we are wanting the information in a specific node or are deleting it, then the goal is to obtain a pointer to it and to the previous node. The loop invariant will be that *prev* and *next* sandwich a location that is either before or at the desired location. We stop when we either reach the location or the end of the list. If the list is sorted, then the desired location is the first for which the information contained in the *next* node is greater or equal to that being search for.

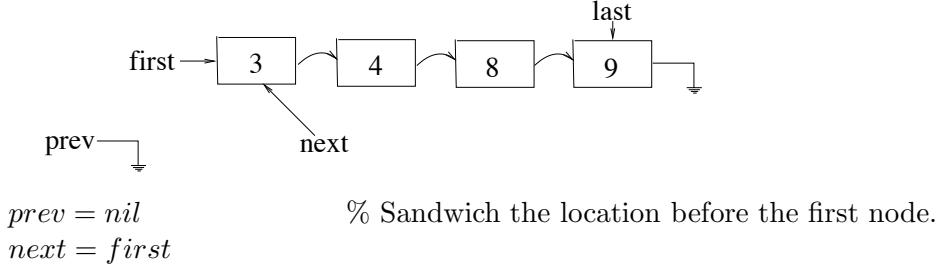
### Code for Walking:

```
loop
```

```
    exit when next = nil or next.info  $\geq$  newElement
    prev = next          % Point prev where next is pointing.
    next = next.link    % Point next to the next node.
end loop
```

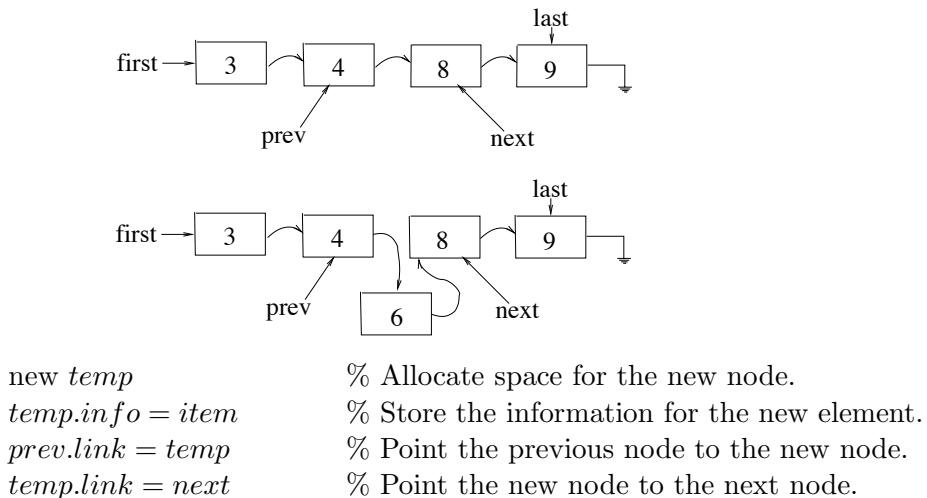
**Exercise 10.2.2** (*See solution in Section V*) *What effect if any would it have if the order of the exit conditions would be switched to “exit when *next.info*  $\geq$  *newElement* or *next* = *nil*?”*

**Initialize the Walk:** To initially establish the loop invariant, *prev* and *next* must sandwich the location before the first node. We do this as follows.

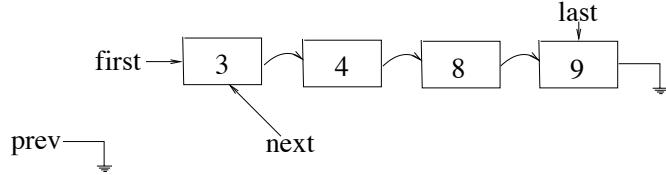


**Adding A Node into the Middle:** Suppose that we are to insert a new node (say with the value 6) into a linked list that is to remain sorted.

### The General Case:



**Special Cases:** If the new node belongs at the beginning of the list (say value 2), then the data structure is as follows.



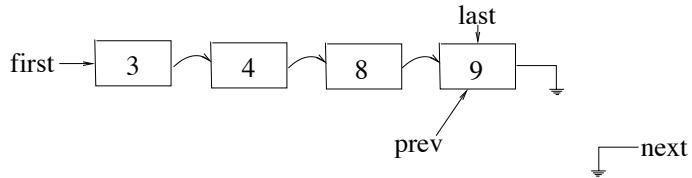
In this case,  $prev.link = temp$  would not work because  $prev$  is not pointing at a node. We will replace this line with the following

```

if  $prev = nil$  then
     $first = temp$           % The new node will be the first node.
else
     $prev.link = temp$       % Point the previous node to the new node.
end if

```

Now what if the new node is to be added on the end, (e.g. value 12)?



The variable  $last$  will no longer point at the last node. Adding the following code to the bottom will solve the problem.

```

if  $prev = last$  then
     $last = temp$           % The new node will be the last node.
end if

```

Another case to consider is when the initial list is empty. In this case, all the variables,  $first$ ,  $last$ ,  $prev$ , and  $next$  will be nil. The new code works in this case as is.

**Code:** Putting together the pieces gives the following code.

```

algorithm Insert(newElement)
pre-cond: This algorithm implicitly acts on some sorted list ADT via first and last.
newElement is the information for a new element.
post-cond: The new element is added where it belongs in the ordered list.
begin
    % create node to insert
    new temp                  % Allocate space for the new node.
    temp.info = item        % Store the information for the new element.

    % Find the location to insert the new node.
    prev = nil            % Sandwich the location before the first node.
    next = first
    loop
        exit when next = nil or next.info  $\geq$  newElement
        prev = next          % Point prev where next is pointing.
        next = next.link     % Point next to the next node.
    end loop

```

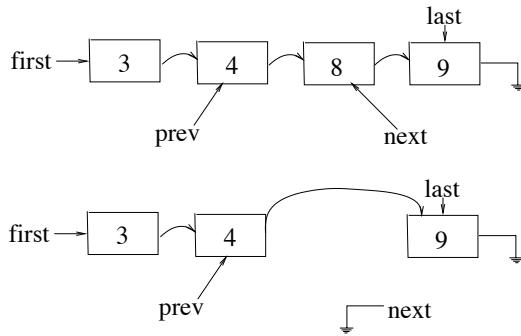
```

% Insert the new node.
if prev = nil then
    first = temp          % The new node will be the first node.
else
    prev.link = temp    % Point the previous node to the new node.
end if
temp.link = next      % Point the new node to the next node.
if prev = last then
    last = temp        % The new node will be the last node.
end if end algorithm

```

**Running Time:** This can require  $\mathcal{O}(n)$  time, where  $n$  is the length of the list.

#### Deleting a Node in the Middle:



We must maintain the linked list BEFORE destroying the node. Otherwise, we will drop the list.

```

prev.link = next.link    % By-pass the node being deleted.
free next                % Deallocate the memory pointed to by next.

```

### 10.3 Merging With a Queue

Merging consists of combining two sorted lists,  $A$  and  $B$ , into one completely sorted list,  $C$ .  $A$ ,  $B$ , and  $C$  are each implemented as queues. The loop invariant maintained is that the  $k$  smallest of the elements are sorted in  $C$ . The larger elements are still in their original lists  $A$  and  $B$ . The next smallest element will either be the first element in  $A$  or the the first element in  $B$ . Progress is made by removing the smaller of these two first elements and adding it to the back of  $C$ . In this way, the algorithm proceeds like two lanes of traffic merging into one. At each iteration, the first car from one of the incoming lanes is chosen to move into the merged lane. This increases  $k$  by one. Initially, with  $k = 0$ , we simply have the given two lists. We stop when  $k = n$ . At this point, all the elements would be sorted in  $C$ . Merging is a key step in the merge sort algorithm presented in Section 15.1.

**algorithm** *Merge(list : A, B)*

***{pre-cond}*:**  $A$  and  $B$  are two sorted lists.

***{post-cond}*:**  $C$  is the sorted list containing the elements of the other two.

```

begin
  loop
    {loop-invariant}: The  $k$  smallest of the elements are sorted in  $C$ . The
      larger elements are still in their original lists  $A$  and  $B$ .
    exit when  $A$  and  $B$  are both empty
    if( the first in  $A$  is smaller than the first in  $B$  or  $B$  is empty ) then
      nextElement = Remove first from  $A$ 
    else
      nextElement = Remove first from  $B$ 
    end if
    Add nextElement to  $C$ 
  end loop
  return(  $C$  )
end algorithm

```

## 10.4 Parsing With a Stack

One important use of stack is for parsing.

### Specifications:

**Preconditions:** An input instance consists of a string of brackets.

**Postconditions:** The output indicates whether the brackets match. Moreover, each left bracket is allocated an integer  $1, 2, 3, \dots$  and each right bracket is allocated the integer from its matching left bracket.

**Example:** Input: ( [ { } ( ) ] ( ) { ( ) } )  
           Output: 1 2 3 3 4 4 2 5 5 6 7 7 6 1

**The Loop Invariant:** Some prefix of the input instance has been read and the correct integer allocated to each of these brackets. The left brackets that have been read and not matched are stored along with their integer in left to right order in a stack with the right most on the top. The variable  $c$  indicates the next integer to be allocated to a left bracket.

**Maintaining the Loop Invariant:** If the next bracket read is a left bracket, then it is allocated the integer  $c$ . Not being matched, it is pushed onto the stack.  $c$  is incremented. If the next bracket read is a right bracket, then it must match the right most left bracket that has been read in. This will be on the top of the stack. The top bracket on the stack is popped. If it matches the right bracket, i.e., (), {}, or [], then the right bracket is allocated the integer for of this left bracket. If not, then an error message is printed.

**Initial Conditions:** Initially, nothing has been read and the stack is empty.

**Ending:** If the stack is empty after the last bracket has been read, then the string has been parsed.

### Code:

```

algorithm Parsing( $s$ )
{pre-cond}:  $s$  is a string of brackets.

```

**$\langle post-cond \rangle$ :** prints out a string of integers that indicate how the brackets match.

```

begin
     $i = 0, c = 1$ 
loop
     $\langle loop-invariant \rangle$ : Prefix  $s[1, i]$  has been allocated integers and its left
        brackets are on the stack.
    exit when  $i = n$ 
    if( $s[i + 1]$  is a left bracket) then
        print( $c$ )
        push( $\langle s[i + 1], c \rangle$ )
         $c = c + 1$ 
    elseif( $s[i + 1]$  = right bracket) then
        if( stackempty() ) return("Cannot parse")
         $\langle left, d \rangle = pop()$ 
        if( $left$  matches  $s[i + 1]$ ) then print( $d$ )
        else return("Cannot parse")
    else
        return("Invalid input character")
    end if
     $i = i + 1$ 
end loop
if( stackempty() ) return("Parsed") else return("Cannot parse")
end algorithm

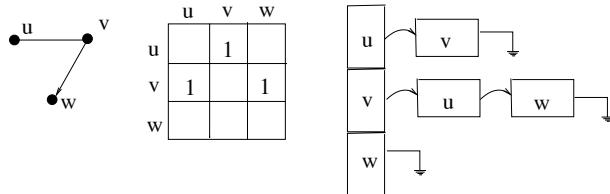
```

**Parsing only “()”:** If you only need to parse one type of brackets and you only want to know whether or not the brackets match, then you do not need the stack in the above algorithm, only an integer storing the number of left brackets in the stack.

**Parsing with Context-Free Grammars:** To parse more complex sentences see Sections 18 and 24.10.

## 10.5 Implementations of Graphs, Trees, and Sets

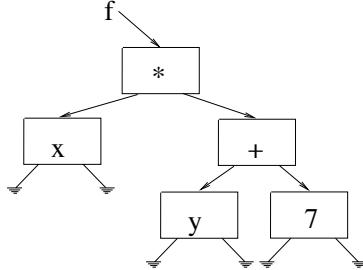
**Graphs:** The following are the standard ways of implementing a graph.



**Adjacency Matrix:** It consists of an  $n \times n$  matrix with  $M(u, v) = 1$  if  $\langle u, v \rangle$  is an edge. It requires  $\Theta(n^2)$  space, which is the number of potential edges,  $\Theta(1)$  time to access a given edge, but  $\Theta(n)$  time to find the edges adjacent to a given node and  $\Theta(n^2)$  to iterate through all the nodes. This is only a problem when the graph is large and sparse.

**Adjacency List:** It lists for each node the nodes adjacent to it. It requires  $\Theta(E)$  space, which is the number of actual edges and can iterate quickly through the edges adjacent to a give node, but requires time proportional to the degree of a node to access specific edge.

**Trees:** Trees are generally implemented by having each node point to each of its children.



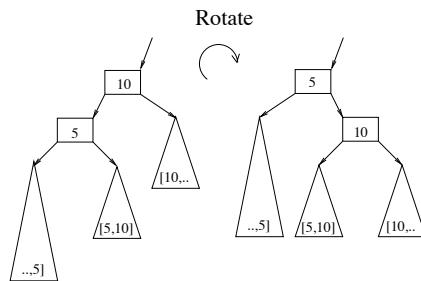
**Orders:** Imposing rules on how the nodes can be ordered, speeds up certain operations.

**Binary Search Tree:** A binary search tree is a data structure used to store keys along with associated data. The nodes are ordered such that for each node all the keys in its left subtree are smaller than its key and all those in the right are larger. Elements can be found in such a tree using binary search in  $\mathcal{O}(\text{height})$  instead of  $\mathcal{O}(n)$  time. See Sections 11.2 and 16.2.

**Heaps:** A heap requires that the key of each node is bigger than that of both its children. This allows one to find the maximum key in  $\mathcal{O}(1)$  time. All updates can be done in  $\mathcal{O}(\log n)$  time. They are useful for a sorting algorithm known as Heap Sort and for the implementation of Priority Queues. See Section 16.4.

**Balanced Trees:** If a binary tree is balanced, it takes less time to traverse down it because it has height at most  $\log_2 n$ . It is too much work maintaining a perfectly balanced tree as nodes are added and deleted. There are a number data structures that are able to add and delete in  $\mathcal{O}(\log_2 n)$  time while ensuring that the tree remains almost balanced. Here are two.

**AVL Trees:** Every node has a balance factor of -1, 0, or 1, where its balance factor is the difference between the height of its left and its right subtrees. As nodes are added or deleted, this invariant is maintained using rotations like the following.



**Red/Black Trees:** Every node is either red or black. If a node is red, then both its children are black. Every path from the root to a leaf contains the same number of black nodes.

**The Set ADT:** One would think that searching for an element in a set would take  $\Theta(n)$ . However, one can do much better.

**Balanced Binary Search Tree:** By storing the elements in a balanced binary search tree, insertions, deletions, and searches can be done in  $\Theta(\log n)$  time.

**Indicator Vector:** If the universe of possible elements is relatively small, then a good data structure is to have a boolean array indexed with each of these possible elements. An entry being true will indicate that the corresponding element is in the set.

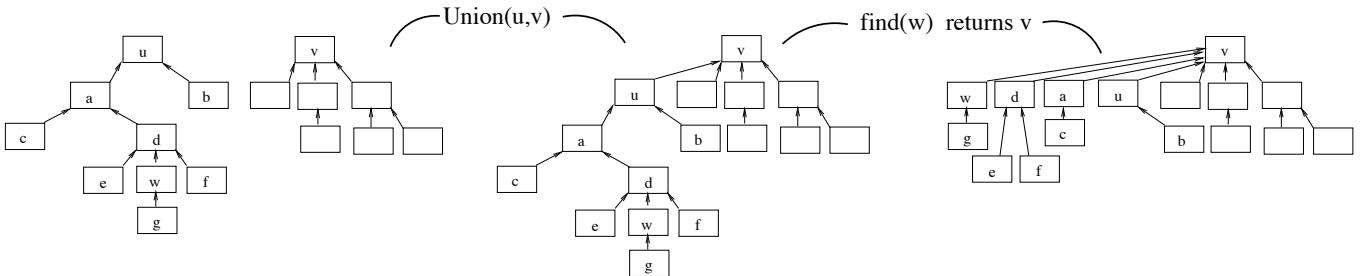
**Hash Tables:** Surprisingly, even if the universe of possible elements is infinite, a similar trick can be done using a data structure called *Hash Tables*. A pseudo random function  $H$  is chosen that maps possible elements of the set to the entries  $[1, N]$  in the table. It is a deterministic function, in that it is easy to compute and always maps an element to the same entry. It is pseudo random, in that it appears to map each element into a random place. Hopefully, all the elements that are in your set happen to be placed into different entries in the table. In this case, one can determine whether or not an element is contained in the set, ask for an arbitrary element from a set, determine the number of elements in the set, iterate through all the elements, and add and delete elements all in constant time, i.e., independent of the number of items in the set. If collisions occur, meaning that two of your set elements get mapped to the same entry, then there are a number of possible methods to rehash them somewhere else.

**Union-Find Set System:** This data structure maintains a number of disjoint sets of elements.

**Operations:** 1)  $Makeset(v)$  which creates a new set containing the specified element  $v$ ; 2)  $Find(v)$  which determines the “name” of the set containing a specified element (Each set is given a distinct but arbitrary name); and 3)  $Union(u,v)$  which merges the sets containing the specified elements  $u$  and  $v$ .

**Running Time:** On average, for all practical purposes, each of these operations can be completed in a constant amount of time. More formally, the total time to do  $m$  of these operations on  $n$  elements is  $\Theta(m\alpha(n))$  where  $\alpha$  is *inverse Ackermann’s function*. This function is so slow growing that even if  $n$  equals the number of atoms in the universe, then  $\alpha(n) \leq 4$ .

**Implementation:** The data structure used is a set of rooted trees for each set containing a node for each element in the set. The difference is that each node points to its parent instead of to its children. The “name” of the set is the contents of the root node.  $Find(w)$  is accomplished by tracing up the tree from  $w$  to the root  $u$ .  $Union(u,v)$  is accomplished by having node  $u$  point to node  $v$ . From then on,  $Find(w)$  for a node  $w$  in  $u$ ’s tree will trace up and find  $v$  instead. What makes this fast on average is that whenever a  $Find$  operation is done, all nodes are changed to point directly to the root of the tree collapsing the tree into a shorter tree.



# Chapter 11

## Narrowing the Search Space

In this chapter, we will include those algorithms that use the “narrowing the search space” type of loop invariant. If you are searching for something, try narrowing the search space, maybe decreasing it by one or even better cutting it in half. A good loop invariant might include “I know that the thing being searched for is not outside of this narrowed search space.”

### 11.1 Binary Search

In a study, a group of experienced programmers was asked to code Binary Search. Easy, yes? 80% got it wrong! My guess is that if they had used loop invariants, they all would have got it correct.

**Exercise 11.1.1** *Before reading this section, try writing code for binary search without the use of loop invariants. Then think about loop invariants and try it again. Finally, read this section. Which of these algorithms work correctly?*

I will use the detailed steps given above to develop a binary search algorithm. Formal proofs can seem tedious when the result seems to be intuitively obvious. However, you need to know how to do these tedious proofs – not so you can do it for every loop you write, but to develop your intuition about where bugs may lie.

#### 1) Specifications:

**Preconditions:** An input instance consists of a sorted list  $L(1..n)$  of elements and a key to be searched for. Elements may be repeated. The key may or may not appear in the list.

**Postconditions:** If the key is in the list, then the output consists of an index  $i$  such that  $L(i) = \text{key}$ . If the key is not in the list, then the output reports this.

**2) Basic Steps:** The basic step compares the key with the element at the center of the sublist. This tells you which half of the sublist the key is in. Keep only the appropriate half.

**3) Loop Invariant:** The algorithm maintains a sublist that contains the key. English sometimes being misleading, a more mathematical statement might be “If the key is contained in the original list, then the key is contained in the sublist  $L(i..j)$ .” Another way of saying this is that we have determined that the key is not within  $L(1..i-1)$  or  $L(j+1..n)$ . It is also worth being clear, as we have done with the notation  $L(i..j)$ , that the sublist includes the end points  $i$  and  $j$ . Confusions in details like this are the cause of many bugs.

**4) Measure of Progress:** The obvious measure of progress is the number of elements in our sublist, namely  $j - i + 1$ .

**5) Main Steps:** As I said, each iteration compares the key with the element at the center of the sublist. This determines which half of the sublist the key is not in and hence which half to keep. It sounds easy, however, there are a few minor decisions to make which may or may not have an impact.

1. If there are an even number of elements in the sublist, do we take the “middle” to be the element slightly to the left of the true middle or the one slightly to the right?
2. Do we compare the key with the middle element using  $\text{key} < L(\text{mid})$  or  $\text{key} \leq L(\text{mid})$ ?
3. Should we also check whether or not  $\text{key} = L(\text{mid})$ ?
4. When splitting the sublist  $L(i..j)$  into two halves do we include the middle in the left half or in the right half?

Decisions like these have the following different types of impact on the correctness of the algorithm.

**Does Not Matter:** If it really does not make a difference which choice you make and you think that the implementer may benefit from having this flexibility, then document this fact.

**Consistent:** Often it does not matter which choice is made, but bugs can be introduced if you are not consistent and clear as to which choice has been made.

**Matters:** Sometimes these subtle points matter.

**Interdependence:** In this case, we will see that each choice can be made either way, but some combinations of choices work and some do not.

When in the first stages of designing an algorithm, I usually do something between flipping a coin and sticking to this decision until a bug arises and attempting to keep all of the possibilities open until I see reasons that it matters.

**6) Maintaining the Loop Invariant:** You must prove that  $\langle \text{loop-invariant}' \rangle \& \text{not } \langle \text{exit-cond} \rangle \& \text{code}_{\text{loop}} \Rightarrow \langle \text{loop-invariant}'' \rangle$ .

If the key is not in the original list, then the statement of the loop invariant is trivially true. Hence, let us assume for now that we have a reasonably large sublist  $L(i..j)$  containing the key. Consider the following three cases:

- Suppose that  $\text{key}$  is strictly less than  $L(\text{mid})$ . Because the list is sorted,  $L(\text{mid}) \leq L(\text{mid} + 1) \leq L(\text{mid} + 2) \leq \dots$ . Combining these facts tells us that the key is not contained in  $L(\text{mid}, \text{mid}+1, \text{mid}+2, \dots)$  and hence it must be contained in  $L(i'..mid-1)$ .
- The case in which  $\text{key}$  is strictly more than  $L(\text{mid})$  is similar.
- Care needs to be taken when  $\text{key}$  is equal to  $L(\text{mid})$ . One option, as has been mentioned, is to test for this case separately. Though finding the key in this way would allow you to stop early, extensive testing shows that this extra comparison slows down the computation. The danger of not testing for it, however, is that we may skip over the key by including the middle in the wrong half. If the test  $\text{key} < L(\text{mid})$  is used, the test

will fail when  $key$  and  $L(mid)$  are equal. Thinking that the key is bigger, the algorithm will keep the right half of the sublist. Hence, the middle element should be included in this half, namely, the sublist  $L(i..j)$  should be split into  $L(i..mid-1)$  and  $L(mid..j)$ . Conversely, if  $key \leq L(mid)$  is used, the test will pass and the left half will be kept. Hence, the sublist should be split into  $L(i..mid)$  and  $L(mid+1..j)$ .

- 7) **Making Progress:** You must be sure to make progress with each iteration, i.e., the sublist must get strictly smaller. Clearly, if the sublist is large and you throw away roughly half of it at every iteration, then it gets smaller. We take note to be careful when the list becomes small.
- 8) **Initial Conditions:** Initially, you obtain the loop invariant by considering the entire list as the sublist. It trivially follows that if the key is in the entire list, then it is also in this sublist.
- 9) **Exit Condition:** One might argue that a sublist containing only one element, cannot get any smaller and hence the program must stop. The exit condition would then be defined to be  $i = j$ . In general having such specific exit condition is prone to bugs, because if by mistake  $j$  were to become less than  $i$ , then the computation might loop forever. Hence, we will use the more general exit condition like  $j \leq i$ . The bug in the above argument is that sublists of size one can be made smaller. You need to consider the possibility of the empty list.
- 10) **Termination and Running Time:** Initially, our measure of progress is the size of the input list. The assumption is that this is finite. Step 10 proves that this measure decreases each iteration by at least one. In fact, progress is made fast. Hence, this measure will quickly be less than or equal to one. At this point, the exit condition will be met and the loop will exit.
- 11) **Ending:** We must now prove that, with the loop invariant and the exit condition together, the problem can be solved, namely that  $\langle \text{loop-invariant} \rangle \And \langle \text{exit-cond} \rangle \And \text{code}_{\text{post-loop}} \Rightarrow \langle \text{post-cond} \rangle$ .

The exit condition gives that  $j \leq i$ . If  $i = j$ , then this final sublist contains one element. If  $j = i - 1$ , then it is empty. If  $j < i - 1$ , then something is wrong. Assuming that the sublist is  $L(i = j)$ , reasonable final code would be to test whether the key is equal to this element  $L(i)$ . (This is our first test for equality.) If they are equal, the index  $i$  is returned. If they are not equal, then we claim that the key is not in the list. On the other hand, if the final sublist is empty then we will simply claim that the key is not in the list. We must now prove that this code ensures that the postcondition has been met.

By the loop invariant, we know that “If the key is contained in the original list, then the key is contained in the sublist  $L(i..j)$ .” Hence, if the key is contained in the original list, then  $L(i..j)$  cannot be empty and the program will find it in the one location  $L(i)$  and give the correct answer. On the other hand, if it is not in the list, then it is definitely not in  $L(i)$  and the program will correctly state that it is not in the list.

- 12) **Testing:** Test with some examples on your own.

- 13) **Special Cases:**

- 6) **Maintaining Loop Invariant (Revisited):** When proving that the loop invariant was maintained, we assumed that we still had a large sublist for which  $i < mid < j$  were three distinct indices. When the sublist has three elements, this is still the case. However,

when the sublist has two elements, the “middle” element must be either the first or the last. This may add some subtleties.

- 7) Making Progress (Revisited):** You must also make sure that you continue to make progress when the sublist becomes small. A sublist of three elements divides into one list of one element and one list of two elements. Hence, progress has been made. Be careful, however, with sublists of two elements.

The following is a common bug: Suppose that we decide to consider the element just to the left of center as being the middle and we decide that the middle element should be included in the right half. Then given the sublist  $L(3, 4)$ , the middle will be element indexed with 3 and the right sublist will be still be  $L(mid..j) = L(3, 4)$ . If this sublist is kept, no progress will be made and the algorithm will loop forever. We ensure that this sublist is cut in half either by having the middle be the one to the left and including it on the left or the one to the right and including it on the right. As seen, in the first case, we must use the test  $key \leq L(mid)$  and in the second case the test  $key < L(mid)$ . Typically, the algorithm is coded using the first option.

#### 14) Implementation Details:

**Math Details:** Small math operations like computing the index of the middle element are prone to bugs. (Try it on your own.) The exact middle is the average between  $i$  and  $j$ , namely  $\frac{i+j}{2}$ . If you want the integer slightly to the left of this, then you round down. Hence,  $mid = \lfloor \frac{i+j}{2} \rfloor$ . It is a good idea to test these operations on small examples.

**Won’t Happen:** We might note that the sublist never becomes empty, assuming that the initial list is not empty. (Check this yourself.) Hence, the code does not need to consider this case.

**Code:** Putting all these pieces together gives the following code.

```

algorithm BinarySearch ( $\langle L(1..n), key \rangle$ )
<pre-cond>:  $L(1..n)$  is a sorted list and  $key$  is an element.
<post-cond>: If the key is in the list, then the output consists of an index  $i$  such that
 $L(i) = key$ .

begin
   $i = 1, j = n$ 
  loop
    <loop-invariant>: If the key is contained in  $L(1..n)$ , then the key is
    contained in the sublist  $L(i..j)$ .
    exit when  $j \leq i$ 
     $mid = \lfloor \frac{i+j}{2} \rfloor$ 
    if( $key \leq L(mid)$ ) then
       $j = mid$            % Sublist changed from  $L(i, j)$  to  $L(i..mid)$ 
    else
       $i = mid + 1$      % Sublist changed from  $L(i, j)$  to  $L(mid+1, j)$ 
    end if
  end loop
  if( $key = L(i)$ ) then
    return(  $i$  )
  else

```

```

        return( "key is not in list" )
    end if
end algorithm

```

- 15) Formal Proof:** Now that we have fixed the algorithm, all the proofs should be checked again. The following proof is a more formal presentation of that given before.

**6') Maintaining the Loop Invariant:** We must prove that  $\langle \text{loop-invariant}' \rangle$  &  $\text{not } \langle \text{exit-cond} \rangle$  &  $\text{code}_{\text{loop}} \Rightarrow \langle \text{loop-invariant}'' \rangle$ . Assume that the computation is at the top of the loop, the loop invariant is true, and the exit condition is not true. Then, let the computation go around the loop one more time. When it returns to the top, you must prove that the loop invariant is again true. To distinguish between these states let  $i'$  and  $j'$  denote the values of the variables  $i$  and  $j$  before executing the loop and let  $i''$  and  $j''$  denote these values afterwards. By  $\langle \text{loop-invariant}' \rangle$ , you know that “If the key is contained in the original list, then the key is contained in the sublist  $L(i'..j')$ .” By  $\text{not } \langle \text{exit-cond} \rangle$ , we know that  $i' < j'$ . From these assumptions, you must prove  $\langle \text{loop-invariant}'' \rangle$ , namely that “If the key is contained in the original list, then the key is contained in the sublist  $L(i''..j'')$ .” If the key is not in the original list, then the statement  $\langle \text{loop-invariant}'' \rangle$  is trivially true, so assume that the key is in the original list. By  $\langle \text{loop-invariant}' \rangle$ , it follows that the key is in  $L(i'..j')$ . There are three cases to consider:

- Suppose that  $\text{key}$  is strictly less than  $L(\text{mid})$ . The comparison  $\text{key} \leq L(\text{mid})$  will pass,  $j''$  will be set to  $\text{mid}$ ,  $i''$  by default will be  $i'$ , and the new sublist will be  $L(i'..mid)$ . Because the list is sorted,  $L(\text{mid}) \leq L(\text{mid} + 1) \leq L(\text{mid} + 2) \leq \dots$ . Combining this fact with the assumption that  $\text{key} < L(\text{mid})$  tells you that the key is not contained in  $L(\text{mid}, \text{mid} + 1, \text{mid} + 2, \dots)$ . You know that the key is contained in  $L(i'..mid, \dots j')$ . Hence, it must be contained in  $L(i'..mid - 1)$ , which means that it is also contained in  $L(i''..j'') = L(i'..mid)$  as required.
- Suppose that  $\text{key}$  is strictly more than  $L(\text{mid})$ . The test will fail. The argument that the key is contained in the new sublist  $L(\text{mid} + 1..j')$  is similar to that above.
- Finally, suppose that  $\text{key}$  is equal to  $L(\text{mid})$ . The test will pass and hence the new sublist will be  $L(i'..mid)$ , which contains the key  $L(\text{mid})$ .

- 10') Running Time:** The sizes of the sublists are approximately  $n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \frac{n}{16}, \dots, 8, 4, 2, 1$ . Hence, only  $\Theta(\log n)$  splits are needed. Each split takes  $O(1)$  time. Hence, the total time is  $\Theta(\log n)$ .

- 12') Testing:** Most of the testing I will leave as an exercise for you. One test case to definitely consider is the input instance in which the initial list to be searched in is empty. In this case, the sublist would be  $L(1..n)$  with  $n = 0$ . Tracing the code, everything looks fine. The loop breaks before iterating. The key won't be found and the correct answer is returned. On closer examination, however, note that the *if* statement accesses the element  $L(0)$ . If the array  $L$  is allocated zero elements, then this may cause a run-time error. In a language like *C*, the program will simply access this memory cell, even though it may not belong to the array and perhaps not even to the program. Such accesses might not cause a bug for years. However, what if the memory cell  $L(0)$  happens to contain the key? Then the program returns that the key is contained at index 1. This would be the wrong answer.

To recap, the goal of a binary search is to find a key within a sorted list. We maintain a sublist that contains the key (if the key is in the original list). Originally, the sublist consists of the entire list. Each iteration compares the key with the element at the center of the sublist. This tells us which half of the sublist the key is in. Keep only the appropriate half. Stop when the sublist contains only one element. Using the loop invariant, we know that if the key was in the original list, then it is this remaining element.

**Exercise 11.1.2** Step 6 “*Maintaining the Loop Invariant*” within 13 “*Special Cases*” considers the case when the sublist has fewer than 3 elements. Check that the three cases considered in the original “*Maintaining the Loop Invariant*” part still hold.

**Exercise 11.1.3** As done in step 15 “*Formal Proof*” for 6’ “*Maintaining the Loop Invariant*”, make the steps 7’, 8’, and 11’ more formal.

**Exercise 11.1.4** As done in step 12’ “*Testing*”, find more cases that might be problematic and prove that they work correctly.

## 11.2 Binary Search Trees

**Binary Search Tree:** Section 10.5 defines a *binary search tree* to be a binary tree data structure in which each node of the tree stores a keys and some associated data. The nodes are ordered such that for each node all the keys in its left subtree are smaller than its key and all those in the right are larger. We will show here how to search quickly for a node with a given tree.

**Binary Search:** Just as in binary search our goal is to cut the search space in half.

**Loop Invariant:** Our loop invariant will be “We currently have a subtree of the binary search tree such that if the key is contained in binary search tree, then the key is contained in our subtree.”

**Making Progress:** We make progress by comparing the key we are searching for with the root of our current subtree. If this root contains the key we are looking for, then we have found it. If the key is smaller than it, then we know that if the key is anywhere then it is in the left subtree of our current tree, else we know that it is in the right subtree.

**Code:**

```
algorithm SearchBST(tree, keyToFind)
⟨pre-cond⟩: tree is a binary tree whose nodes contain key and data fields. keyToFind is
a key.
⟨post-cond⟩: If there is a node with this key is in the tree, then the associated data is
returned.
```

```
begin
    subtree = tree
    loop
        ⟨loop-invariant⟩: If the key is contained in tree, then the key is con-
        tained in subtree.
        if( tree = emptyTree ) then
```

```

        result( "key not in tree" )
else if( keyToFind < rootKey(subtree) ) then
    subtree = leftSub(subtree)
else if( keyToFind = rootKey(subtree) ) then
    result( rootData(subtree) )
else if( keyToFind > rootKey(subtree) ) then
    subtree = rightSub(subtree)
end if
end loop
end algorithm

```

**Running Time:** The number of iterations of this algorithm is at most the height of the binary search tree, which if balanced is  $\Theta(\log n)$ .

**Recursive Algorithm:** This algorithm can be implemented recursively. See Section 16.2.

### 11.3 Magic Sevens

My mom gave my son Joshua a book of magic tricks. The book writes, “This trick really is magic. It comes right every time you do it, but there is no explanation why.” As it turns out, there is a bug in the way that they explain the trick. Our task is to fix their bug and to counter their “there is no explanation why.” The only Magic is that of Loop Invariants. The algorithm is a variant on binary search.



#### The Trick:

- Let  $c$ , an odd integer, be the number of “columns.” They use  $c = 3$ .
- Let  $r$ , an odd integer, be the number of “rows.” They use  $r = 7$ .
- Let  $n = c \cdot r$  be the number of cards. They use  $n = 21$ .
- Let  $t$  be the number of iterations. They use  $t = 2$ .
- Let  $f$  be the final index of the selected card. They use  $f = 11$ .
- Ask someone to select one of the  $n$  cards and then shuffle the deck.
- Repeat  $t$  times:
  - Spread the cards out as follows. Put  $c$  cards in a row left to right face up. Put a second row on top, but shifted down slightly so that you can see what both the first and second row of cards are. Repeat for  $r$  rows. This forms  $c$  columns of  $r$  cards each.
  - Ask which column the selected card is in.

- Stack the cards in each column. Put the selected column in the middle. (This is why  $c$  is odd.)
- Output the  $f^{th}$  card.

Our task is to determine for which values  $c$ ,  $r$ ,  $n$ ,  $t$ , and  $f$  this trick finds the selected card.

**Easier Version:** Analyzing this trick, turns out to be harder than I initially thought. Hence, consider the following easier trick first. Instead of putting the selected column in the middle of the other columns, put it in the front.

**Basic Steps:** Each iteration we gain some information about which card had been selected.

The trick seems to be similar to binary search. A difference is that binary search splits the current sublist into two parts while this trick splits the entire pile into  $c$  parts. In both algorithms, each iteration we learn which of these piles the sought after element is in.

**The Loop Invariant:** A good first guess for a loop invariant would be that used by binary search. The loop invariant will state that some subset  $S_i$  of the cards contains the selected card. In this easier version, the column containing the card is continuously moved to the front of the stack. Hence, let us guess that  $S_i = \{1, 2, \dots, s_i\}$  indexes the first  $s_i$  cards in the deck.

**Narrow Search Space:** This is a classic “narrowed search space” loop invariant. The algorithm maintains that the thing being searched for is not outside of some continually narrowed search space.

**Initial Conditions:** Again, as done in binary search, we initially obtain the loop invariant by considering the entire stack of cards.

**Maintaining the Loop Invariant:** Suppose that before the  $i^{th}$  iteration, the selected card is one of the first  $s_{i-1}$  in the deck. When the cards are laid out, the first  $s_{i-1}$  cards will be spread on the tops of the  $c$  columns. Some columns will get  $\lceil \frac{s_{i-1}}{c} \rceil$  of these cards and some will get  $\lfloor \frac{s_{i-1}}{c} \rfloor$  of them. When we are told which column the selected card is in, we will know that the selected card is one of the first  $\lceil \frac{s_{i-1}}{c} \rceil$  cards in this column. In conclusion  $s_i = \lceil \frac{s_{i-1}}{c} \rceil$ . We solve this recurrence relation by guessing that for each  $i$ ,  $s_i = \lceil \frac{n}{c^i} \rceil$ . We verify this guess by checking that  $s_0 = \lceil \frac{n}{c^0} \rceil = n$  and that  $s_i = \lceil \frac{s_{i-1}}{c} \rceil = \left\lceil \frac{\lceil \frac{n}{c^{i-1}} \rceil}{c} \right\rceil = \lceil \frac{n}{c^i} \rceil$ .

**Exit Condition:** Let  $t = \lceil \log_c n \rceil$  and let  $f = 1$ . After  $t$  rounds, the loop invariant will establish that the selected card is one of the first  $s_t = \lceil \frac{n}{c^t} \rceil = 1$  in the deck. Hence, the selected card must be the first card.

**Lower Bound:** For a matching lower bound on the number of iterations needed see Section 28.2.

**Trick in Book:** The book has  $n = 21$ ,  $c = 3$ , and  $t = 2$ . Because  $21 = n \not\leq c^t = 3^2 = 9$ , the trick in the book does not work. Two rounds is not enough. There needs to be three.

**Original Trick:** Consider again the original trick where the selected column is put into the middle.

**The Loop Invariant:** Because the selected column is put into the middle, let us guess that  $S_i$  consists of the middle  $s_i$  cards. More formally, let  $d_i = \frac{n-s_i}{2}$ . Neither the first nor the last  $d_i$  cards will be the selected card. Instead it will be one of  $S_i = \{d_i + 1, \dots, d_i + s_i\}$ . Note, however, that this requires that  $s_i$  is odd.

**Initial Conditions:** For  $i = 0$ ,  $s_0 = n$ ,  $d_0 = 0$ , and the selected card can be any card in the deck.

**Maintaining the Loop Invariant:** Suppose that before the  $i^{th}$  iteration, the selected card is not one of the first  $d_{i-1}$  cards, but is one of the middle  $s_{i-1}$  in the deck. Then when the cards are laid out, the first  $d_{i-1}$  cards will be spread on the tops of the  $c$  columns. Some columns will get  $\lceil \frac{d_{i-1}}{c} \rceil$  of these cards and some will get  $\lfloor \frac{d_{i-1}}{c} \rfloor$  of them. In general, however, we can say that the first  $\lfloor \frac{d_{i-1}}{c} \rfloor$  cards of each column are not the selected card. We use the floor instead of the ceiling here, because this is the worst case. By symmetry, we also know that the selected card is not one of the last  $\lfloor \frac{d_{i-1}}{c} \rfloor$  cards in each column. When the person points at a column, we learn that the selected card is somewhere in that column. However, from before we knew that the selected card is not one of the first or last  $\lfloor \frac{d_{i-1}}{c} \rfloor$  cards in this column. There are only  $r$  cards in the column. Hence, the selected card must be one of the middle  $r - 2\lfloor \frac{d_{i-1}}{c} \rfloor$  cards in the column. Define  $s_i$  to be this value. The new deck is formed by stacking the columns together with these cards in the middle.

**Running Time:** When sufficient rounds have occurred so that  $s_t = 1$ , then the selected card will be in the middle indexed by  $f = \lceil \frac{n}{2} \rceil$ .

**Trick in Book:** The book has  $n = 21$ ,  $c = 3$ , and  $r = 7$ .

$$\begin{array}{lll}
 s_i = r - 2\lfloor \frac{d_{i-1}}{c} \rfloor & d_i = \frac{n-s_i}{2} & S_i = \{d_i + 1, \dots, d_i + s_i\} \\
 s_0 = n = 21 & d_0 = \frac{21-21}{2} = 0 & S_0 = \{1, 2, \dots, 21\} \\
 s_1 = 7 - 2\lfloor \frac{0}{3} \rfloor = 7 & d_1 = \frac{21-7}{2} = 7 & S_1 = \{8, 9, \dots, 14\} \\
 s_2 = 7 - 2\lfloor \frac{7}{3} \rfloor = 3 & d_2 = \frac{21-3}{2} = 9 & S_2 = \{10, 11, 12\} \\
 s_3 = 7 - 2\lfloor \frac{9}{3} \rfloor = 1 & d_3 = \frac{21-1}{2} = 10 & S_3 = \{11\}
 \end{array}$$

Again three and not two rounds are needed.

**Running Time:** Temporarily ignoring the floor in the equation for  $s_i$  makes the analysis easier.  $s_i = r - 2\lfloor \frac{d_{i-1}}{c} \rfloor \approx r - 2\frac{d_{i-1}}{c} = \frac{n}{c} - 2\frac{(n-s_{i-1})/2}{c} = \frac{s_{i-1}}{c}$ . Again, this recurrence relation gives that  $s_i = \frac{n}{c^i}$ . If we include the floor, challenging manipulations give that  $s_i = 2\lfloor \frac{s_{i-1}}{2c} - \frac{1}{2} \rfloor + 1$ . More calculations give that  $s_i$  is always  $\frac{n}{c^i}$  rounded up to the next odd integer.

Here are a few more problems that use a “narrowing the search space” type of loop invariant.

**Exercise 11.3.1** Suppose  $S$  and  $T$  are sorted arrays, each containing  $n$  elements. The problem is to find the  $n^{th}$  smallest of the  $2n$  numbers.

binary search

## Chapter 12

# Iterative Sorting Algorithms

Sorting is a classic computational problem. During the first few decades of computers, almost all computer time was devoted to sorting. Many sorting algorithms have been developed. It is useful to know a number of them. A practical reason is that sorting needs to be done in many different situations. Some depend on low time complexity, others on small memory, others on simplicity. Throughout the book, we consider a number of sorting algorithms because they are simple yet provide a rich selection of examples for demonstrating different algorithmic techniques. They are Selection Sort and Insertion Sort in Section 8.3, Radix/Counting Sort here, Merge Sort and Quick Sort in Section 15.1, and Heap Sort in Section 16.4.

### 12.1 Bucket Sort by Hand

As a professor, I often have to sort a large stack of student's papers by the last name. The algorithm that I use is an iterative version of *quick sort* and *bucket sort*. See Section 15.1.

**Partitioning Into 5 Buckets:** Computers are good at using a single comparison to determine whether an element is greater than the pivot value or not. Humans, on the other hand, tend to be good at quickly determining which of 5 buckets an element belongs in. I first partition the papers based on which of the following ranges the first letter of the name is within: [A-E], [F-K], [L-O], [P-T], or [U-Z]. Then I partition the [A-E] bucket into the sub-buckets [A], [B], [C], [D], and [E]. Then I partition the [A] bucket based on the second letter of the name. This works for this application because the list to be sorted consist of names whose first letters are fairly predictably distributed through the alphabet.

**A Stack of Buckets:** One difficulty with this algorithm is keeping track of all the buckets. For example, after the second partition, we will have nine buckets: [A], [B], [C], [D], [E], [F-K], [L-O], [P-T], and [U-Z]. After the third, we will have 13. On a computer, the recursion of the algorithm is implemented with a stack of stack frames. Correspondingly, when I sort the student's papers, I have a stack of buckets.

**The Loop Invariant:** I use the following loop invariant to keep track of what I am doing.

I always have a pile (initially empty) of papers that are sorted. These papers belong before all other papers. I also have a stack of piles. Though the papers within each pile are out of order, each paper in a pile belongs before each paper in a later pile. For example, at some point in the algorithm, the papers starting with [A-C] will be sorted and the piles in my stack will consist of [D], [E], [F-K], [L-O], [P-T], and [U-Z].

**Maintain the Loop Invariant:** I make progress while maintaining this loop invariant as follows. I take the top pile off the stack, here the [D]. If it only contains a half dozen or so papers, I sort them using insertion sort. These are then added to the top of the sorted pile, [A-C], giving [A-D]. On the other hand, if the pile [D] taken off the stack is larger than this, I partition it into 5 piles, [DA-DE], [DF-DK], [DL-DO], [DP-DT], and [DU-DZ], which I push back onto the stack. Either way, my loop invariant is maintained.

**Exiting:** When the last bucket has been removed from the stack, the papers are sorted.

## 12.2 Counting Sort (A Stable Sort)

We present Radix/Counting Sort here because it uses loop invariants. It is, however, a little strange. Instead of taking steps from the start to the destination in a direct way, it heads in an unexpected direction, go around the block, and heads in the back door.

Most sorting algorithms are said to be *comparison based*, because the only way of accessing the input values is by comparing pairs of them, i.e.,  $a_i \leq a_j$ . Radix/Counting, presented here, manipulates the elements in other ways.

This algorithm is said to run in linear  $\Theta(n)$  time, while Merge, Quick, and Heap Sorts are said to run in  $\Theta(n \log n)$  time. This appears to be faster, but this is confusing and misleading. Radix/Counting requires  $\Theta(n)$  bit operations when  $n$  is the total number of bits in the input instance. Merge, Quick, and Heap Sort requires  $\Theta(N \log N)$  comparisons when  $N$  is the number of numbers in the list. Assuming that the  $N$  numbers to be sorted are distinct, each needs  $\Theta(\log N)$  bits to be represented for a total of  $n = \Theta(N \log N)$  bits. Hence, Merge, Quick, and Heap Sort is also linear time in that it requires  $\Theta(n)$  bit operations when  $n$  is the total number of bits in the input instance.

In practice, the Radix/Counting algorithm may be a little faster than the other algorithms. However, Quick and Heap Sorts have the advantage of being done “in place” in memory, while the Radix/Counting Sort requires an auxiliary array of memory to transfer the data to.

I will present the counting sort algorithm first. It is only useful in the special case where the elements to be sorted have very few possible values. Radix sort, described next, uses counting sort as a subroutine.

### Specifications:

**Preconditions:** The input is a list of  $N$  values  $a_0, \dots, a_{N-1}$ , each within the range  $0 \dots k-1$ .

**Postconditions:** The output is a list consisting of the same  $N$  values in non-decreasing order. The sort is *stable*, meaning that if two elements have the same value, then they must appear in the same order in the output as in the input. (This is important when extra data is carried with each element.)

### The Main Idea:

**Where an Element Goes:** Consider any element of the input. By counting, we will determine where this element belongs in the output and then we simply put it there. Where it belongs is determined by the number of elements that must appear before it. To simplify the argument, let’s index the locations with  $[0, N - 1]$ . This way, the element in the location indexed by zero has zero elements before it and that in location  $\hat{c}$  has  $\hat{c}$  elements before it.

Suppose that the element  $a_i$  has the value  $v$ . Every element that has a strictly smaller value must go before it. Let's denote this count with  $\hat{c}_v$ , i.e.,  $\hat{c}_v = |\{j \mid a_j < v\}|$ . The only other elements that go before  $a_i$  are elements with exactly the same value. Because the sort must be *stable*, the number of these that go before it is the same as the number that appear before it in the input. If this number happens to be  $q_{a_i}$ , then element  $a_i$  would belong in location  $\hat{c}_v + q_{a_i}$ . In particular, the first element in the input with value  $v$  goes in location  $\hat{c}_v + 0$ .

**Example 12.1:**

Input:	1 0 1 0 2 0 0 1 2 0
Output:	0 0 0 0 1 1 1 2 2
Index:	0 1 2 3 4 5 6 7 8 9

The first element to appear in the input with value 0 goes into location 0 because there are  $\hat{c}_0 = 0$  elements with smaller values. The next such element goes into location 1, the next into 2, and so on.

The first element to appear in the input with value 1 goes into location 5 because there are  $\hat{c}_1 = 5$  elements with smaller values. The next such element goes into location 6, and the next into 7.

Similarly, the first element with value 2 goes into location  $\hat{c}_2 = 8$ .

**Computing  $\hat{c}_v$ :** You could compute  $\hat{c}_v$  by making a pass through the input counting the number of elements that have values smaller than  $v$ . Doing this separately for each value  $v \in [0..k - 1]$ , however, would take  $\mathcal{O}(kN)$  time, which is too much.

Instead, let's first count how many times each value occurs in the input. For each  $v \in [0..k - 1]$ , let  $c_v = |\{i \mid a_i = v\}|$ . This count can be computed with one pass through the input. For each element, if the element has value  $v$ , increment the counter  $c_v$ . This requires only  $\mathcal{O}(N)$  addition and indexing operations.

Given the  $c_v$  values, you could compute  $\hat{c}_v = \sum_{v'=0}^{v-1} c_{v'}$ . Computing one such  $\hat{c}_v$  would require  $\mathcal{O}(k)$  additions and computing all of them would take  $\mathcal{O}(k^2)$  additions, which is too much.

Alternatively, note that  $\hat{c}_0 = 0$  and  $\hat{c}_v = \hat{c}_{v-1} + c_{v-1}$ . Of course one must have computed the previous values before computing the next. Now computing one such  $\hat{c}_v$  requires  $\mathcal{O}(1)$  additions and computing all of them takes only  $\mathcal{O}(k)$  additions.

**The Main Loop of the Counting Sort Algorithm:** The main loop in the algorithm considers the input elements one at a time in the order  $a_0, \dots, a_{N-1}$  that they appear in the input and places them in the output array where they belong.

**The Loop Invariant:**

1. The input elements that have already been considered have been put in their correct places in the output.
2. For each  $v \in [0..k - 1]$ ,  $\hat{c}_v$  gives the index in the output array where the next input element with value  $v$  goes.

**Establishing the Loop Invariant:** Compute the counts  $\hat{c}_v$  as described above. This establishes the loop invariant before any input elements are considered, because this  $\hat{c}_v$  value gives the location where the first element with value  $v$  goes.

**Body of the Loop:** Take the next input element. If it has value  $v$ , place it in the output location indexed by  $\hat{c}_v$ . Then increment  $\hat{c}_v$ .

**Maintaining the Loop Invariant:** By the loop invariant, we know that if the next input element has value  $v$ , then it belongs in the output location indexed by  $\hat{c}_v$ . Hence, it is being put in the correct place. The next input element with value  $v$  will then go immediately after this current one in the output, i.e., into location  $\hat{c}_v + 1$ . Hence, incrementing  $\hat{c}_v$  maintains the second part of the loop invariant.

**Exiting the Loop:** Once all the input elements have been considered, the first loop invariant establishes that the list has been sorted.

**Code:**

```

 $\forall v \in [0..k - 1], c_v = 0$ 
loop  $i = 0$  to  $N - 1$ 
     $++ c_{a[i]}$ 
 $\hat{c}_0 = 0$ 
loop  $v = 1$  to  $k - 1$ 
     $\hat{c}_v = \hat{c}_{v-1} + c_{v-1}$ 
loop  $i = 0$  to  $N - 1$ 
     $b[\hat{c}_{a[i]}] = a[i]$ 
     $++ \hat{c}_{a[i]}$ 

```

**Running Time:** The total time is  $\mathcal{O}(N + k)$  addition and indexing operations. If the input can only contain  $k = \mathcal{O}(N)$  possible values, then this algorithm works in linear time. It does not work well if the number of possible values is much higher.

### 12.3 Radix Sort

The radix sort is a useful algorithm that dates back to the days of card-sorting machines, now found only in computer museums.

**Specifications:**

**Preconditions:** The input is a list of  $N$  values. Each value is an integer with  $d$  digits. Each digit is a value from 0 to  $k - 1$ , i.e., the value is viewed as an integer base  $k$ .

**Postconditions:** The output is a list consisting of the same  $N$  values in non-decreasing order.

**The Main Step:** For some digit  $i \in [1..d]$ , sort the input according to the  $i^{th}$  digit, ignoring the other digits. Use a stable sort, eg. counting sort.

**Examples:** Old computer punch cards were organized into  $d = 80$  columns, and in each column a hole could be punched in one of  $k = 12$  places. A card-sorting machine could mechanically examine each card in a deck and distribute the card into one of 12 bins, depending on which hole had been punched in a specified column.

A “value” might consist of a year, a month, and a day. You could then sort the elements by the year, by the month, or by the day.

**Order in which to Consider the Digits:** It is most natural to sort with respect to the most significant digit first. The final sort, after all, has all the elements with a 0 as the first digit at the beginning, followed by those with a 1.

If the operator of the card-sorting machine sorted first by the most significant digit, he would get 12 piles. Each of these piles would then have to be sorted separately, according to the remaining digits. Sorting the first pile according to the second digit would produce 12 more piles. Sorting the first of those piles according to the third digit would produce 12 more piles. The whole process would be a nightmare.

Sorting with respect to the least significant digit seems silly at first. Sorting  $\langle 79, 94, 25 \rangle$  gives  $\langle 94, 25, 79 \rangle$ , which is completely wrong. Even so, this is what the algorithm does.

**The Algorithm:** Loop through the digits from low to high order. For each, use a stable sort to sort the elements according to the current digit, ignoring the other digits.

**Example:**

Sorted by first 3 digits	Considering 4 <sup>th</sup> digit	Stable sorted by 4 <sup>th</sup> digit
184	3184	1195
192	5192	1243
195	1195	1311
243	1243	3184
271	3271	3271
311	1311	5192

The result is sorted by first 4 digits.

**Proof of Correctness:**

**The Loop Invariant:** After sorting wrt (with respect to) the first  $i$  low order digits, the elements are sorted wrt the value formed from these  $i$  digits.

**Establishing the Loop Invariant:** The LI is initially trivially true, because initially no digits have been considered.

**Maintaining the Loop Invariant:** Suppose that the elements are sorted wrt the value formed from the  $i - 1$  lowest digits. For the elements to be sorted wrt the value formed from the  $i$  lowest digits, all the elements with a 0 in the  $i^{\text{th}}$  digit must come first, followed by those with a 1, and so on. This can be accomplished by sorting the elements wrt the  $i^{\text{th}}$  digit while ignoring the other digits. Moreover, the block of elements with a 0 in the  $i^{\text{th}}$  digit must be sorted wrt the  $i - 1$  lowest digits. By the loop invariant, they were in this order and because the sorting wrt the  $i^{\text{th}}$  digit was stable, these elements will remain in the same relative order. The same is true for the block of elements with a 1 or 2 or so on in the  $i^{\text{th}}$  digit.

**Ending:** When  $i = d$ , they are sorted wrt the value formed from all  $d$  digits, and hence are sorted.

## 12.4 Radix/Counting Sort

I will now combine the radix and counting sorts to give a linear-time sorting algorithm.

## Specifications:

**Preconditions:** The input is a list of  $N$  values. Each value is an  $l$ -bit integer.

**Postconditions:** The output is a list consisting of the same  $N$  values in non-decreasing order.

**The Algorithm:** The algorithm is to use radix sort with counting sort to sort each digit. To do this, we need to view each  $l$ -bit value as an integer with  $d$  digits, where each digit is a value from 0 to  $k - 1$ . This is done by splitting the  $l$  bits into  $d$  blocks of  $\frac{l}{d}$  bits each and treating each such block as a digit between 0 and  $k - 1$ , where  $k = 2^{\frac{l}{d}}$ . Here,  $d$  is a parameter to set later.

**Example:** Consider sorting the numbers 30, 41, 28, 40, 31, 26, 47, 45. Here  $N = 8$  and  $l = 6$ . Let's set  $d = 2$  and split the  $l = 6$  bits into  $d = 2$  blocks of  $\frac{l}{d} = 3$  bits each. Treat each of these blocks as a digit between 0 and  $k - 1$ , where  $k = 2^3 = 8$ . For example,  $30 = 011110_2$  gives the blocks  $011_2 = 3$  and  $110_2 = 6$ .

Doing this for all the numbers gives:

- $30 = 36_8 = 011\ 110_2$
- $41 = 51_8 = 101\ 001_2$
- $28 = 34_8 = 011\ 100_2$
- $40 = 50_8 = 101\ 000_2$
- $31 = 37_8 = 011\ 111_2$
- $26 = 32_8 = 011\ 010_2$
- $47 = 57_8 = 101\ 111_2$
- $45 = 55_8 = 101\ 101_2$

Stable sorting wrt the first digit gives:

- $40 = 50_8 = 101\ 000_2$
- $41 = 51_8 = 101\ 001_2$
- $26 = 32_8 = 011\ 010_2$
- $28 = 34_8 = 011\ 100_2$
- $45 = 55_8 = 101\ 101_2$
- $30 = 36_8 = 011\ 110_2$
- $31 = 37_8 = 011\ 111_2$
- $47 = 57_8 = 101\ 111_2$

Stable sorting wrt the second digit gives:

- $26 = 32_8 = 011\ 010_2$
- $28 = 34_8 = 011\ 100_2$
- $30 = 36_8 = 011\ 110_2$
- $31 = 37_8 = 011\ 111_2$
- $40 = 50_8 = 101\ 000_2$
- $41 = 51_8 = 101\ 001_2$
- $45 = 55_8 = 101\ 101_2$
- $47 = 57_8 = 101\ 111_2$

This is sorted!

**Running Time:** Using the counting sort to sort with respect to one of the  $d$  digits takes  $\Theta(N + k)$  “operations.” Hence, the entire algorithm takes  $\Theta(d \cdot (N + k))$  operations. We argue that this is minimized by setting  $k = \mathcal{O}(N)$  and  $d = \frac{l}{\log k} = \frac{l}{\log N}$ . The reason is that increasing  $k$  decreases  $d$ , but increasing  $k$  beyond  $N$  makes the  $k$  significant to the time. This gives  $T = \Theta(d \cdot (N + k)) = \Theta(\frac{l}{\log N} N)$  “operations.”

Formally time complexity measures the number of bit operations performed as a function of the number of bits to represent the input. When we say that counting sort takes  $\Theta(N + k)$  “operations”, a single “operation” must be able to add two values with magnitude  $\Theta(N)$  or to index into arrays of size  $N$ . Each of these takes  $\Theta(\log N)$  bit-operations. Hence, the total time to sort is  $T = \Theta(\frac{l}{\log N} N)$  “operations”  $\times \log N \frac{\text{bit-operations}}{\text{“operation”}} = \Theta(l \cdot N)$  bit-operations. The input, consisting of  $N$   $l$ -bit values, requires  $n = l \cdot N$  bits to represent. Hence, the running time  $\Theta(l \cdot N) = \Theta(n)$  is linear in the size of the input.

One example is when you are sorting  $N$  values in the range 0 to  $N^r$ . Each value requires  $l = \log N^r = r \log N$  bits to represent it, for a total of  $n = N \log(N^r) = rN$  bits. Our settings would then be  $k = N$ ,  $d = \frac{l}{\log N} = r$ , and  $T = \Theta(d \cdot N) = \Theta(rN) = \Theta(n)$ .

# Chapter 13

## Euclid's GCD Algorithm

Recursive algorithms and more of the input type of iterative algorithms both extend a solution for a smaller input instance into a larger one. The following is an amazing algorithm which also does this. It finds the greatest common divisor (GCD) of two integers. For example,  $GCD(18, 12) = 6$ . It was first done by Euclid, an ancient Greek. Without the use of loop invariants, you would never be able to understand what the algorithm does; with their help, it is easy.

### Specifications:

**Preconditions:** An input instance consists of two positive integers,  $a$  and  $b$ .

**Postconditions:** The output is  $GCD(a, b)$ .

**The Loop Invariant:** Like many loop invariants, designing this one required creativity. The algorithm maintains two variables  $x$  and  $y$  whose values change with each iteration of the loop under the invariant that their  $GCD$ ,  $GCD(x, y)$ , does not change, but remains equal to the required output  $GCD(a, b)$ .

**Type of Loop Invariant:** This is a strange loop invariant. The algorithm is more like recursion. A solution to a smaller instance of the problem gives the solution for the original.

**Initial Conditions:** The easiest way of establishing the loop invariant that  $GCD(x, y) = GCD(a, b)$  is by setting  $x$  to  $a$  and  $y$  to  $b$ .

**The Measure of Progress:** Progress is made by making  $x$  or  $y$  smaller.

**Ending:** We will exit when  $x$  or  $y$  is small enough that we can compute their GCD easily. By the loop invariant, this will be the required answer.

**A Middle Iteration on a General Instance:** Let us first consider a general situation in which  $x$  is bigger than  $y$  and both are positive.

**Main Steps:** Our goal is to make  $x$  or  $y$  smaller without changing their GCD. A useful fact is that  $GCD(x, y) = GCD(x - y, y)$ , eg.  $GCD(52, 10) = GCD(42, 10) = 2$ . The reason is that any value that divides into  $x$  and  $y$  also divides into  $x - y$  and similarly any value that divides into  $x - y$  and  $y$  also divides into  $x$ . Hence, replacing  $x$  with  $x - y$  would make progress while maintaining the loop invariant.

**Exponential?:** Let's jump ahead in designing the algorithm and estimate its running time.

A loop executing only  $x = x - y$  will iterate  $\frac{a}{b}$  times. If  $b$  is much smaller than  $a$ , then this may take a while. However, even if  $b = 1$ , this is only  $a$  iterations. This looks like it is linear time. However, you should express the running time of an algorithm as a function of input size. See Section 4. The number of bits needed to represent the instance  $\langle a, b \rangle$  is  $n = \log a + \log b$ . Expressed in these terms, the running time is  $Time(n) = \Theta(a) = \Theta(2^n)$ . This is exponential time. For example, if  $a = 1,000,000,000,000,000$  and  $b = 1$ , I would not want to wait for it.

**Faster Main Steps:** Instead of subtracting one  $y$  from  $x$  each iteration, why not speed up the process by subtracting a multiple of  $y$  all at once. We could set  $x_{new} = x - d \cdot y$  for some integer value of  $d$ . Our goal is to make  $x_{new}$  as small as possible without making it negative. Clearly,  $d$  should be  $\lfloor \frac{x}{y} \rfloor$ . This gives  $x_{new} = x - \lfloor \frac{x}{y} \rfloor \cdot y = x \bmod y$ , which is within the range  $[0..y - 1]$  and is the remainder when dividing  $y$  into  $x$ . For example,  $52 \bmod 10 = 2$ .

**Maintaining the Loop Invariant:** The step  $x_{new} = x \bmod y$ , maintains the loop invariant because  $GCD(x, y) = GCD(x \bmod y, y)$ , e.g.,  $GCD(52, 10) = GCD(2, 10) = 2$ .

**Making Progress:** The step  $x_{new} = x \bmod y$  makes progress by making  $x$  smaller only if  $x \bmod y$  is smaller than  $x$ . This is only true if  $x$  is greater or equal to  $y$ . Suppose that initially this is true because  $a$  is greater than  $b$ . After one iteration of  $x_{new} = x \bmod y$  becomes smaller than  $y$ . Then the next iteration will do nothing. A solution is to then swap  $x$  and  $y$ .

**New Main Steps:** Combining  $x_{new} = x \bmod y$  with a swap gives the main steps of  $x_{new} = y$  and  $y_{new} = x \bmod y$ .

**Maintaining the Loop Invariant:** This maintains our original loop invariant because  $GCD(x, y) = GCD(y, x \bmod y)$ , e.g.,  $GCD(52, 10) = GCD(10, 2) = 2$ . It also maintains the new loop invariant that  $0 \leq y \leq x$ .

**Making Progress:** Because  $y_{new} = x \bmod y \in [0..y - 1]$  is smaller than  $y$ , we make progress by making  $y$  smaller.

**Special Cases:** Setting  $x = a$  and  $y = b$  does not establish the loop invariant which says that  $x$  is at least  $y$  if  $a$  is smaller than  $b$ . An obvious solution is to initially test for this and to swap them if necessary. However, as advised in Section 8.4, it is sometimes fruitful to try tracing out what the algorithm that you have already designed would do given such an input. Suppose  $a = 10$  and  $b = 52$ . The first iteration would set  $x_{new} = 52$  and  $y_{new} = 10 \bmod 52$ . This last value is a value within the range  $[0..51]$  that is the remainder when dividing 10 by 52. Clearly this is 10. Hence, the code automatically swaps the values by setting  $x_{new} = 52$  and  $y_{new} = 10$ . Hence, no new code is needed. Similarly, if  $a$  and  $b$  happen to be negative, the initial iteration will make  $y$  positive and the next will make both  $x$  and  $y$  positive.

**Exit Condition:** We are making progress by making  $y$  smaller. We should stop when  $y$  is small enough that we can solve compute the GCD easily. Let's try small values of  $y$ . Using  $GCD(x, 1) = 1$ , the GCD is easy to compute when  $y = 1$ , however, we will never get this unless  $GCD(a, b) = 1$ . How about  $GCD(x, 0)$ ? This turns out to be  $x$  because  $x$  divides evenly into both  $x$  and 0. Let's try an exit condition of  $y = 0$ .

**Termination:** We know that the program will eventually stop as follows.  $y_{new} = x \bmod y \in [0..y - 1]$  ensures that each step  $y$  gets strictly smaller and does not go negative. Hence, eventually  $y$  must be zero.

**Ending:** Formally we prove that  $\langle \text{loop-invariant} \rangle \& \langle \text{exit-cond} \rangle \& \text{code}_{\text{post-loop}} \Rightarrow \langle \text{post-cond} \rangle$ .  $\langle \text{loop-invariant} \rangle$  gives  $\text{GCD}(x, y) = \text{GCD}(a, b)$  and  $\langle \text{exit-cond} \rangle$  gives  $y = 0$ . Hence  $\text{GCD}(a, b) = \text{GCD}(x, 0) = x$ . The final code will return the value of  $x$ . This establishes the  $\langle \text{post-cond} \rangle$  that  $\text{GCD}(a, b)$  is returned.

**Code:**

```
algorithm GCD(a, b)
⟨pre-cond⟩: a and b are integers.
⟨post-cond⟩: Returns GCD(a, b).
```

```
begin
    int x,y
    x = a
    y = b
    loop
        ⟨loop-invariant⟩: GCD(x,y) = GCD(a,b).
        if(y = 0) exit
        xnew = y  ynew = x mod y
        x = xnew
        y = ynew
    end loop
    return( x )
end algorithm
```

**Example:** The following traces the algorithm give  $a = 22$  and  $b = 32$ .

iteration	value of $x$	value of $y$
1st	22	32
2nd	32	22
3rd	22	10
4th	10	2
5th	2	0

$$\text{GCD}(22, 32) = 2.$$

**Running Time:** For the running time to be linear in the size of the input, the number of bits  $\log y$  to represent  $y$  must decrease by at least one each iteration. This means that  $y$  must decrease by at least a factor of two. Consider the example of  $x = 19$  and  $y = 10$ .  $y_{new}$  becomes  $19 \bmod 10 = 9$ , which is only a decrease of one. However, the next value of  $y$  will be  $10 \bmod 9 = 1$  which is a huge drop.

We will be able to prove that every two iterations,  $y$  drops by a factor of two, namely that  $y_{k+2} < y_k/2$ . There are two cases. In the first case,  $y_{k+1} \leq y_k/2$ . Then we are done because as stated above  $y_{k+2} < y_{k+1}$ . In the second case,  $y_{k+1} \in [y_k/2 + 1, y_k - 1]$ . Unwinding the algorithm gives that  $y_{k+2} = x_{k+1} \bmod y_{k+1} = y_k \bmod y_{k+1}$ . One algorithm for computing

$y_k \bmod y_{k+1}$  is to continually subtract  $y_{k+1}$  from  $y_k$  until the amount is less than  $y_{k+1}$ . Because  $y_k$  is more than  $y_{k+1}$ , this  $y_{k+1}$  is subtracted at least once. It follows that  $y_k \bmod y_{k+1} \leq y_k - y_{k+1}$ . By the case,  $y_{k+1} > y_k/2$ . In conclusion  $y_{k+2} = y_k \bmod y_{k+1} \leq y_k - y_{k+1} < y_k/2$ .

We prove that the number of times that the loop iterates is  $\mathcal{O}(\log(\min(a, b))) = \mathcal{O}(n)$  as follows. After the first or second iteration,  $y$  is  $\min(a, b)$ . Every iteration  $y$  goes down by at least a factor of 2. Hence, after  $k$  iterations,  $y_k$  is at most  $\min(a, b)/2^k$  and after  $\mathcal{O}(\log(\min(a, b)))$  iterations is at most one.

The algorithm iterates a linear number  $\mathcal{O}(n)$  of times. Each iteration must do a *mod* operation. Poor Euclid had to compute these by hand, which must have gotten very tedious. A computer may be able to do *mods* in one operation, however, the number of bit operations need for two  $n$  bit inputs is  $\mathcal{O}(n \log n)$ . Hence, the time complexity of this GCD algorithm is  $\mathcal{O}(n^2 \log n)$ .

**Lower Bound:** We will prove a lower bound, not of the minimum time for any algorithm to find the GCD, but of this particular algorithm by finding a family of input values  $\langle a, b \rangle$  for which the program loops  $\Theta(\log(\min(a, b)))$  times. Unwinding the code gives  $y_{k+2} = y_{k+1} \bmod y_{k+1} = y_k \bmod y_{k+1}$ . As stated,  $y_k \bmod y_{k+1}$  is computed by subtracting  $y_{k+1}$  from  $y_k$  a number of times. We want the  $y$ 's to shrink as slowly as possible. Hence, let us say that it is subtracted only once. This gives  $y_{k+2} = y_k - y_{k+1}$  or  $y_k = y_{k+1} + y_{k+2}$ . This is the definition of Fibonacci numbers only backwards, i.e.,  $Fib(0) = 0$ ,  $Fib(1) = 1$  and  $Fib(n) = Fib(n-1) + Fib(n-2)$ . See Section 6.4. On input  $a = Fib(n+1)$  and  $b = Fib(n)$ , the program iterates  $n$  times. This is  $\Theta(\log(\min(a, b)))$ , because  $Fib(n) = 2^{\Theta(n)}$ .

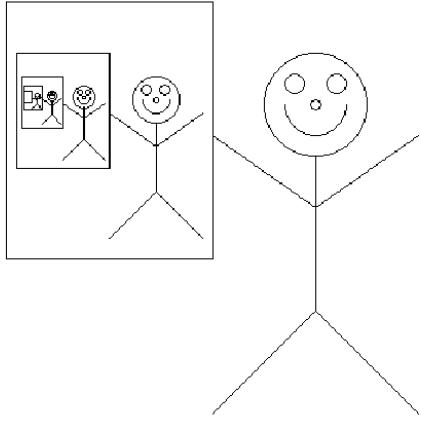
# **Part III**

# **Recursion**

## Chapter 14

# Recursion Abstractions, Techniques, and Theory

Iterative algorithms start at the beginning and take one step at a time towards the final destination. Another technique used in many algorithms is to slice the given task into a number of disjoint pieces, solve each of these separately, and then combine these answers into an answer for the original task. This is the *divide and conquer* method. When the subtasks are different, this leads to different subroutines. When they are instances of the original problem, it leads to recursive algorithms.



People often find recursive algorithms very difficult. To understand them, it is important to have a good solid understanding of the theory and techniques presented in this chapter. We will start with a simple example.

**Towers of Hanoi:** The towers of hanoi is a classic puzzle for which the only possible way of solving it is to think recursively.

**Description of the Problem:** The puzzle starts with a stack of disks of different sizes on the first of three poles. The goal is to move the stack over to the last pole. See the first and the last figures of Figure 14.1. You are only allowed to take one disk from the top of the stack on one pole and place it on the top of the stack on another pole. Another rule is that no disk can be placed on top of a smaller disk.



Figure 14.1: The Towers of Hanoi Problem

**Lost With First Step:** The first step must be to move the smallest disk. But it is by no means clear whether to move it to the second or to the third pole.

**Divide:** Jump into the middle of the computation. One thing that is clear is that at some point, you must move the biggest disk from the first pole to the last. In order to do this, there can be no other disks on either the first or the last pole. Hence, all the other disks need to be stacked on the second pole. See the second and the third figures. This point in the computation splits the problem into two subproblems that must be solved. The first is how to move all the disks except for largest from the first pole to the second. See the first and the second figures. The second is how to move these same disks from the second pole to the third. See the third and the last figures.

**Conquer:** Together these steps solve the entire problem.

**Friend Solves Subproblems:** In order to make a clear separation between task of solving the entire problem and that of solving each of the subproblems, I like to say that we delegate to one “friend” the task of solving one of the subproblems and delegate to another “friend” the other.

**Recurse:** The subproblem of moving all but the largest disk from the first to the second pole turns out to be an instance of the more general Towers of Hanoi problem. The input to this more general problem specifies an integer  $n$  and two poles,  $pole_{source}$  and  $pole_{destination}$ . The task assumes that the smallest  $n$  disks are currently on  $pole_{source}$ , it does not care where the larger disks are, and it must move these smallest  $n$  disks to  $pole_{destination}$ .

**Code:**

```

algorithm TowersOfHanoi( $n$ , source, destination, spare)
  <pre-cond>: The  $n$  smallest disks are on polesource.
  <post-cond>: They are moved to poledestination.

  begin
    if( $n \leq 0$ )
      Nothing to do
    else
      TowersOfHanoi( $n - 1$ , source, spare, destination)
      Move the  $n^{th}$  disk from polesource to poledestination.
      TowersOfHanoi( $n - 1$ , spare, destination, source)
    end if
  end algorithm

```

**Running Time:** Let  $T(n)$  be the time to move  $n$  disks. Clearly,  $T(1) = 1$  and  $T(n) = 2 \cdot T(n - 1) + 1$ . Solving this gives,  $T(n) = 2^n - 1$ .

**Exercise 14.0.1** Let  $S_1, S_2, S_3 \subseteq [1..n]$  be a partition of the disks onto the three poles. The disks on each pole are sorted by size. Give an algorithm and its running time (as a function of  $S_1, S_2, S_3$ ) to move all the disks to the third pole from this initial position.

## 14.1 Different Abstractions

There are a number of different abstractions within which one can view a recursive algorithm. Though the resulting algorithm is the same, having the different paradigms at your disposal can be helpful.

**Code:** Code is useful for implementing an algorithm on a computer. It is precise and succinct. However, as said, code is prone to bugs, language dependent, and often lacks higher level of intuition.

**Stack of Stack Frames:** Recursive algorithms are executed using a stack of stack frames. Though this should be understood, tracing out such an execution is painful.

**Tree of Stack Frames:** This is a useful way of viewing the entire computation at once. It is particularly useful when computing the running time of the algorithm. However, the structure of the computation tree might be very complex and difficult to understand all at once.

**Friends & Strong Induction:** The easiest method is to focus separately on one step at a time. Suppose that someone gives you an instance of the computational problem. You solve it as follows. If it is sufficiently small, solve it yourself. Otherwise, you have a number of friends to help you. You construct for each friend an instance of the same computational problem that is *smaller* than your own. We refer to these as *subinstances*. Your friends magically provide you with the solutions to these. You then combine these *subsolutions* into a solution for your original instance.



I refer to this as the *friends* level of abstraction. If you prefer, you can call it the *strong induction* level of abstraction and use the word “recurse” instead of “friend.” Either way, the key is that you concern yourself only about your stack frame. Do not worry about how your friends solve the subinstances that you assigned them. Similarly, do not worry about whomever gave you your instance and what he does with your answer. Leave these things up to them.

**Use It:** Though students resist it, I strongly recommend using this method when designing, understanding, and describing a recursive algorithm.

**Faith in the Method:** As said for the loop invariant method, you do not want to be rethinking the issue of whether or not you should steal every time you walk into a store. It is better to have some general principles with which to work. Every time you consider a hard algorithm, you do not want to be rethinking the issue of whether or not you believe in recursion. Understanding the algorithm itself will be hard enough. Hence, while reading this chapter you should once and for all come to understand and believe to the depth of your soul how the above mentioned steps are sufficient to describing a recursive algorithm. Doing this can be difficult. It requires a whole new way of looking at algorithms. However, at least for the duration of this course, adopt this as something that you believe in.

## 14.2 Circular Argument? Looking Forwards vs Backwards

**Circular Argument?:** Recursion involves designing an algorithm by using it as if it already exists. At first this looks paradoxical. Consider the following related problem. You must get into a house, but the door is locked and the key is inside. The magical solution is as follows. If I could get in, I could get the key. Then I could open the door, so that I could get in. This is a circular argument. However, it is not a legal recursive program because the subinstance is not smaller.



**One Problem and a Row of Instances:** Consider a row of houses. The recursive problem consists of getting into any specified house. Each house in the row is a separate instance of this problem. Each house is bigger than the next. Your task is to get into the biggest one. You are locked out of all the houses. However, the key to a house is locked in the house of the next smaller size.

**The Algorithm:** The task is no longer circular. The algorithm is as follows. The smallest house is small enough that one can use brute force to get in. For example, one could simply lift off the roof. Once in this house, we can get the key to the next house, which is then easily opened. Within this house, we can get the key to the house after that and so on. Eventually, we are in the largest house as required.

**Focus On One Step:** Though this algorithm is quite simple to understand, more complex algorithms are harder to understand all at once. Instead we want to focus on one step at a time. Here, one step consists of the following. We are required to open house  $i$ . We ask a friend to open house  $i-1$ , out of which we take the key with which we open house  $i$ .

**Working Forwards vs Backwards:** An iterative algorithm works forward. It knows about house  $i-1$ . It uses a loop invariant to assume that this house has been opened. It searches this house and learns that the key within it is that for house  $i$ . Because of this, it decides that house  $i$  would be a good one to go to next.

A recursion algorithm works backwards. It knows about house  $i$ . It wants to get it open. It determines that the key for house  $i$  is contained in house  $i-1$ . Hence, opening house  $i-1$  is a subtask that needs to be accomplished.

There are two advantages of recursive algorithms over iterative ones. The first is that sometimes it is easier to work backwards than forwards. The second is that a recursive algorithm is allowed to have more than one subtask to be solved. This forms a tree of houses to open instead of a row of houses.

**Do Not Trace:** When designing a recursive algorithm it is tempting to trace out the entire computation. “I must open house  $n$ , so I must open house  $n-1$ , .... The smallest house I rip the roof off. I get the key for house 1 and open it. I get the key for house 2 and open it. .... I get the key for house  $n$  and open it.” Such an explanation is bound to be incomprehensible.

**Solving Only Your Instance:** An important quality of any leader is knowing how to delegate. Your job is to open house  $i$ . Delegate to a friend the task of opening house  $i-1$ . Trust him and leave the responsibility to him.

### 14.3 The Friends' Recursion Level of Abstraction

The following are the steps to follow when developing a recursive algorithm within the friends level of abstraction.

**Specifications:** Carefully write the specifications for the problem.

**Preconditions:** The preconditions state any assumptions that must be true about the input instance for the algorithm to operate correctly.

**Postconditions:** The postconditions are statements about the output that must be true when the algorithm returns.

This step is even more important for recursive algorithms than for other algorithms, because there must be a tight agreement between what is expected from you in terms of pre and postconditions and what is expected from your friends.

**Size:** Devise a measure of the “size” of each instance. This measure can be anything you like and corresponds to the measure of progress within the loop invariant level of abstraction.

**General Input:** Consider a large and general instance of the problem.

**Magic:** Assume that by “magic” a friend is able to provide the solution to any instance of your problem as long as the instance is strictly smaller than the current instance (according to your measure of size). More specifically, if the instance that you give the friend meets the stated preconditions, then his solution will meet the stated postconditions. Do **not**, however, expect your friend to accomplish more than this. (In reality, the friend is simply a mirror image of yourself.)

**Subinstances:** From the original instance, construct one or more *subinstances*, which are smaller instances of the same problem. Be sure that the preconditions are met for these smaller instances. Do not refer to these as “subproblems.” The problem does not change, just the input instance to the problem.

**Subsolutions:** Ask your friend to (recursively) provide solutions for each of these subinstances. We refer to these as *subsolutions* even though it is not the solution, but the instance that is smaller.

**Solution:** Combine these subsolutions into a solution for the original instance.

**Generalizing the Problem:** Sometimes a subinstance you would like your friend to solve is not a legal instance according to the preconditions. In such a case, start over redefining the preconditions in order to allow such instances. Note, however, that now you too must be able to handle these extra instances. Similarly, the solution provided by your friend may not provide enough information about the subinstance for you to be able to solve the original problem. In such a case, start over redefining the postcondition by increasing the amount of information that your friend provides. Note again that now you too must also provide this extra information. See Section 16.3.

**Minimizing the Number of Cases:** You must ensure that the algorithm that you develop works for *every* valid input instance. To achieve this, the algorithm will often require many separate pieces of code to handle inputs of different types. Ideally, however, the algorithm developed has as few such cases as possible. One way to help you minimize the number of cases needed is as follows. Initially, consider an instance that is as large and as general as possible. If there are a number of different types of instances, choose one whose type is as general as possible. Design an algorithm that works for this instance. Afterwards, if there is another type of instance that you have not yet considered, consider a general instance of this type. Before designing a separate algorithm for this new instance, try executing your existing algorithm on it. You may be surprised to find that it works. If, on the other hand, it fails to work for this instance, then repeat the above steps to develop a separate algorithm for this case. This process may need to be repeated a number of times.

For example, suppose that the input consists of a binary tree. You may well find that the algorithm designed for a tree with a full left child and a full right child also works for a tree with a missing child and even for a child consisting of only a single node. The only remaining case may be the empty tree.

**Base Cases:** When all the remaining unsolved instances are sufficiently small, solve them in a brute force way.

**Running Time:** Use a recurrence relation or the tree of stack frames to estimate the running time.

**A Link to the Techniques for Iterative Algorithms:** The techniques that often arise in iterative algorithms also arise in recursive algorithms, though sometimes in a slightly different form.

**More of the Input:** When the input includes  $n$  objects, this technique for iterative algorithms extends (for  $i = 1..n-1$ ) a solution for the first  $i-1$  objects into a solution for the first  $i$ . This same technique also can be used for recursive algorithms. Your friend provides you a solution for the first  $n-1$  objects in your instance and then you extend this to a solution to your entire instance. This iterative algorithm and this recursive algorithm would be two implementations of the same algorithm. The recursion is more interesting when one friend can provide you a solution for the first  $\lfloor \frac{n}{2} \rfloor$  objects in your instance, another friend can provide a solution for the next  $\lceil \frac{n}{2} \rceil$  objects, and you combine them into a solution for the whole.

**More of the Output:** This technique for iterative algorithms builds the output one piece at a time. Again a recursive algorithm could have a friend build all but the last piece and have you add the last piece. However, it is more interesting to have one friend build the first half of the output, another the second half, and you combine them somehow.

**Narrow Search Space:** Some iterative algorithms might repeatedly narrow the search space in which to look for something. Instead, a recursive algorithm might split the search space in half and have a friend search each half.

**Case Analysis:** Instead of trying each of the cases oneself, one could give one case to each friend.

**Work Done:** Work does not accumulate in recursive algorithms as it does in iterative algorithms. We get each friend to do some work and then we do some work ourselves to combine these solutions.

## 14.4 Proving Correctness with Strong Induction

Whether you give your subinstances to friends or you recurse on them, this level of abstraction considers only the algorithm for the “top” stack frame. We must now prove that this suffices to produce an algorithm that successfully solves the problem on every input instance. When proving this, it is tempting to talk about stack frames. This stack frame calls this one, which calls that one, until you hit the base case. Then the solutions bubble back up to the surface. These proofs tend to make little sense and get very low marks. Instead, we use strong induction to prove formally that the friends level of abstraction works.

**Strong Induction:** Strong induction is similar to induction except instead of assuming only  $S(n-1)$  to prove  $S(n)$ , you must assume all of  $S(0), S(1), S(2), \dots, S(n-1)$ .

**A Statement for each  $n$ :** For each value of  $n \geq 0$ , let  $S(n)$  represent a boolean statement. For some values of  $n$ , this statement may be true and for others it may be false.

**Goal:** Our goal is to prove that it is true for every value of  $n$ , namely that  $\forall n \geq 0, S(n)$ .

**Proof Outline:** Proof by strong induction on  $n$ .

**Induction Hypothesis:** “For each  $n \geq 0$ , let  $S(n)$  be the statement that ...”. (It is important to state this clearly.)

**Base Case:** Prove that the statement  $S(0)$  is true.

**Induction Step:** For each  $n \geq 0$ , prove  $S(0), S(1), S(2), \dots, S(n-1) \Rightarrow S(n)$ .

**Conclusion:** “By way of induction, we can conclude that  $\forall n \geq 0, S(n)$ .”

**Exercise 14.4.1** Give the “process” of strong induction as we did for regular induction.

**Exercise 14.4.2** (See solution in Section V) As a formal statement, the base case can be eliminated because it is included in the formal induction step. How is this? (In practice, the base cases are still proved separately.)

### Proving The Recursive Algorithm Works:

**Induction Hypothesis:** For each  $n \geq 0$ , let  $S(n)$  be the statement, “The recursive algorithm works for *every* instance of size  $n$ .”

**Goal:** Our goal is to prove that  $\forall n \geq 0, S(n)$ , namely that “The recursive algorithm works for *every* instance.”

**Proof Outline:** The proof is by strong induction on  $n$ .

**Base Case:** Proving  $S(0)$  involves showing that the algorithm works for the base cases of size  $n = 0$ .

**Induction Step:** The statement  $S(0), S(1), S(2), \dots, S(n-1) \Rightarrow S(n)$  is proved as follows. First assume that the algorithm works for every instance of size strictly smaller than  $n$  and then proving that it works for every instance of size  $n$ . This mirrors exactly what we do in the friends level of abstraction. To prove that the algorithm works for every instance of size  $n$ , consider an arbitrary instance of size  $n$ . The algorithm constructs subinstances that are strictly smaller. By our induction hypothesis we know that our algorithm works for these. Hence, the recursive calls

return the correct solutions. Within the friends level of abstraction, we proved that the algorithm constructs the correct solutions to our instance from the correct solutions to the subinstances. Hence, the algorithm works for this arbitrary instance of size  $n$ .  $S(n)$  follows.

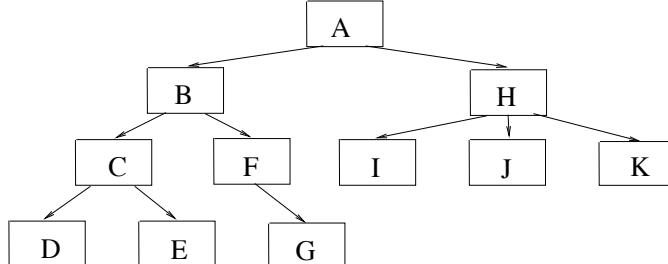
**Conclusion:** By way of strong induction, we can conclude that  $\forall n \geq 0$ ,  $S(n)$ , i.e., The recursive algorithm works for every instance.

## 14.5 The Stack Frame Levels of Abstraction

**Tree Of Stack Frames Level Of Abstraction:** Tracing out the entire computation of a recursive algorithm, one line of code at a time, can get incredibly complex. This is why the friend's level of abstraction, which considers one stack frame at a time, is the best way to understand, explain, and design a recursive algorithm. However, it is also useful to have some picture of the entire computation. For this, the tree of stack frames level of abstraction is best.

The key thing to understand is the difference between a particular routine and a particular execution of a routine on a particular input instance. A single routine can at one moment in time have many instances of it being “executed.” Each such execution is referred to as a *stack frame*. You can think of each as the task given to a separate friend. Note that even though each routine may be executing exactly the same routine, each routine may currently be on a different line of code and have different values for the local variables.

If each routine makes a number of subroutine calls (recursive or not), then the stack frames that get executed form a tree.



In this example, instance  $A$  is called first. It executes for a while and at some point recursively calls  $B$ . When  $B$  returns,  $A$  then executes for a while longer before calling  $H$ . When  $H$  returns,  $A$  executes for a while before completing. We skipped over the details of the execution of  $B$ . Let's go back to when instance  $A$  calls  $B$ .  $B$  then calls  $C$ , which calls  $D$ .  $D$  completes, then  $C$  calls  $E$ . After  $E$ ,  $C$  completes. Then  $B$  calls  $F$  which calls  $G$ .  $G$  completes,  $F$  completes,  $B$  completes and  $A$  goes on to call  $H$ . It does get complicated.

**Stack Of Stack Frames Level Of Abstraction:** Both the friend level of abstraction and the tree of stack frames level are only abstract ways of understanding the computation. The algorithm is actually implemented on a computer by a stack of stack frames. What is actually stored in the computer memory at any given point in time is a only single path down the tree. The tree represents what occurs throughout time. In the above example, when instance  $G$  is active,  $A$ ,  $B$ ,  $F$ , and  $G$  are in the stack.  $C$ ,  $D$  and  $E$  have been removed from memory as these have completed.  $H$ ,  $I$ ,  $J$ , and  $K$  have not been started yet. Although we speak of many separate stack frames executing on the computer, the computer is not a parallel machine. Only top stack frame  $G$  is actively being executed. The other instances are on hold waiting for a subroutine call that it made to return.

It is useful to understand how memory is managed for the simultaneous execution of many instances of the same routine. The routine itself is described only once by a block of code that appears in static memory. This code declares a set of variables. Each instance of this routine that is currently being executed, on the other hand, may be storing different values in these variables and hence needs to have its own separate copy of these variables. The memory requirements of each of these instances are stored in a separate “stack frame.” These frames are stacked on top of each other within stack memory.

Recall that a stack is a data structure in which either the last element to be added is *popped* off or a new element is *pushed* onto the top. Let us denote the top stack frame by  $A$ . When the execution of  $A$  makes a subroutine call to a routine with some input values, a stack frame is created for this new instance. This frame denoted  $B$  is pushed onto the stack after that for  $A$ . In addition to a separate copy of the local variables for the routine, it contains a pointer to the next line code that  $A$  must execute when  $B$  returns. When  $B$  returns, its stack frame is popped and  $A$  continues to execute at the line of code that had been indicated within  $B$ . When  $A$  completes it too is popped off the stack.

**Silly Example:** The following is a silly example that demonstrates how difficult it is to trace out the full stack-frame tree, yet how easy it is to determine the output using the friends/strong-induction method.

```

algorithm Fun( $n$ )
  <pre-cond>:  $n$  is an integer.
  <post-cond>: Outputs a silly string.

begin
  if(  $n > 0$  ) then
    if(  $n = 1$  ) then
      put "X"
    else if(  $n = 2$  ) then
      put "Y"
    else
      put "A"
      Fun( $n - 1$ )
      Put "B"
      Fun( $n - 2$ )
      Put "C"
    end if
  end if
end algorithm

```

**Exercise 14.5.1** Attempt to trace out the tree of stack frames for this algorithm for  $n = 5$ .

**Exercise 14.5.2** (See solution in Section V) Now try the following simpler approach. What is the output with  $n = 1$ ? What is the output with  $n = 2$ ? Trust the answers to all previous questions; do not recalculate them. (Assume a trusted friend gave you the answer.) Now, what is the output with  $n = 3$ ? Repeat this approach for  $n = 4, 5$ , and  $6$ .

# Chapter 15

## Some Simple Examples of Recursive Algorithms

I will now give some simple examples of recursive algorithms. Even if you have already seen them before, study them again, keeping the abstractions, techniques, and theory learned in this chapter in mind. For each example, look for the key steps of the friend paradigm. What are the subinstances given to the friend? What is the size of an instance? Does it get smaller? How are the friend's solutions combined to give your solution? As well, what does the tree of stack frames look like? What is the time complexity of the algorithm?

### 15.1 Sorting and Selecting Algorithms

The classic divide and conquer algorithms are Merge Sort and Quick Sort. They both have the following basic structure.

#### General Recursive Sorting Algorithm:

- Take the given list of objects to be sorted (numbers, strings, student records, etc.)
- Split the list into two sublists.
- Recursively have friends sort each of the two sublists.
- Combine the two sorted sublists into one entirely sorted list.

This process leads to four different algorithms, depending on whether you:

**Sizes:** Split the list into two sublists each of size  $\frac{n}{2}$  or one of size  $n - 1$  and the one of size one.

**Work:** Put minimal effort into splitting the list but put lots of effort into recombining the sublists or put lots of effort into splitting the list but put minimal effort into recombining the sublists.

**Exercise 15.1.1** (*See solution in Section V*) Consider the algorithm that puts minimal effort into splitting the list into one of size  $n - 1$  and the one of size one, but put lots of effort into recombining the sublists. Also consider the algorithm that puts lots of effort into splitting the list into one of size  $n - 1$  and the one of size one, but put minimal effort into recombining the sublists. What are these two algorithms?

**Merge Sort (Minimal work to split in half):** This is the classic recursive algorithm.

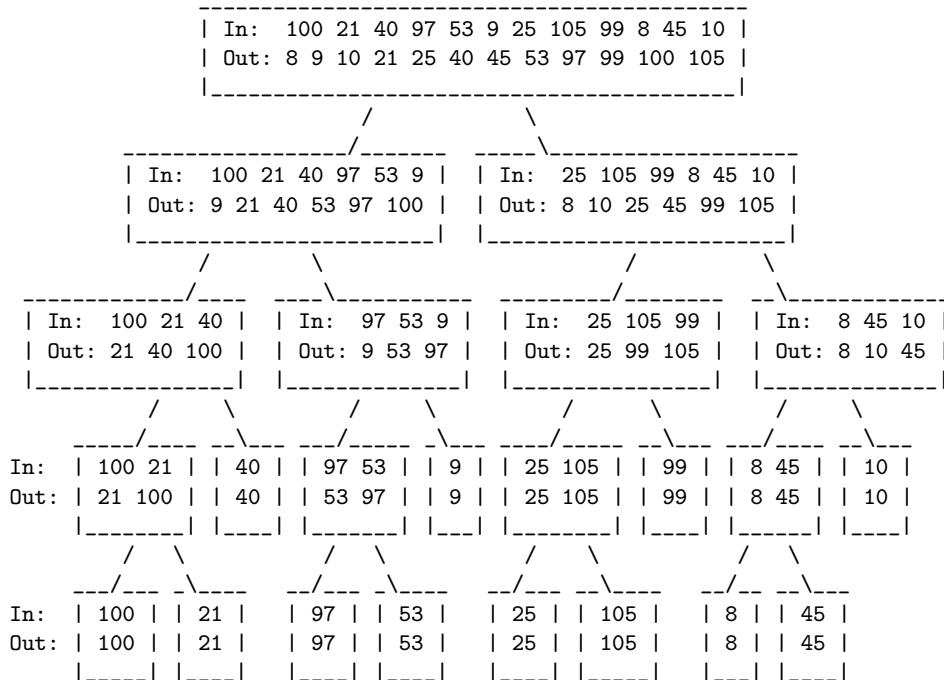
**Friend's Level of Abstraction:** Recursively give one friend the first half of the input to sort and another friend the second half to sort. You then combine these two sorted sublists into one completely sorted list. This combining process is referred to as *merging*. A simple linear time algorithm for it can be found in Section 10.3.

**Size:** The size of an instance is the number of elements in the list. If this is at least two, then the sublists are smaller than the whole list. Hence, it is valid to recurse on them with the reassurance that your friends will do their parts correctly. On the other hand, if the list contains only one element, then by default it is already sorted and nothing needs to be done.

**Generalizing the Problem:** If the input is assumed to be received in an array indexed from 1 to  $n$ , then the second half of the list is not a valid instance. Hence, we redefine the preconditions of the sorting problem to require as input both an array  $A$  and a subrange  $[i, j]$ . The postcondition is that the specified sublist be sorted in place.

**Running Time:** Let  $T(n)$  be the total time required to sort a list of  $n$  elements. This total time consists of the time for two subinstances of half the size to be sorted, plus  $\Theta(n)$  time for merging the two sublists together. This gives the recurrence relation  $T(n) = 2T(n/2) + \Theta(n)$ . See Section 6 to learn how to solve recurrence relations like these. In this example,  $\frac{\log a}{\log b} = \frac{\log 2}{\log 2} = 1$  and  $f(n) = \Theta(n^1)$  so  $c = 1$ . Because  $\frac{\log a}{\log b} = c$ , the technique concludes that time is dominated by all levels and  $T(n) = \Theta(f(n) \log n) = \Theta(n \log n)$ .

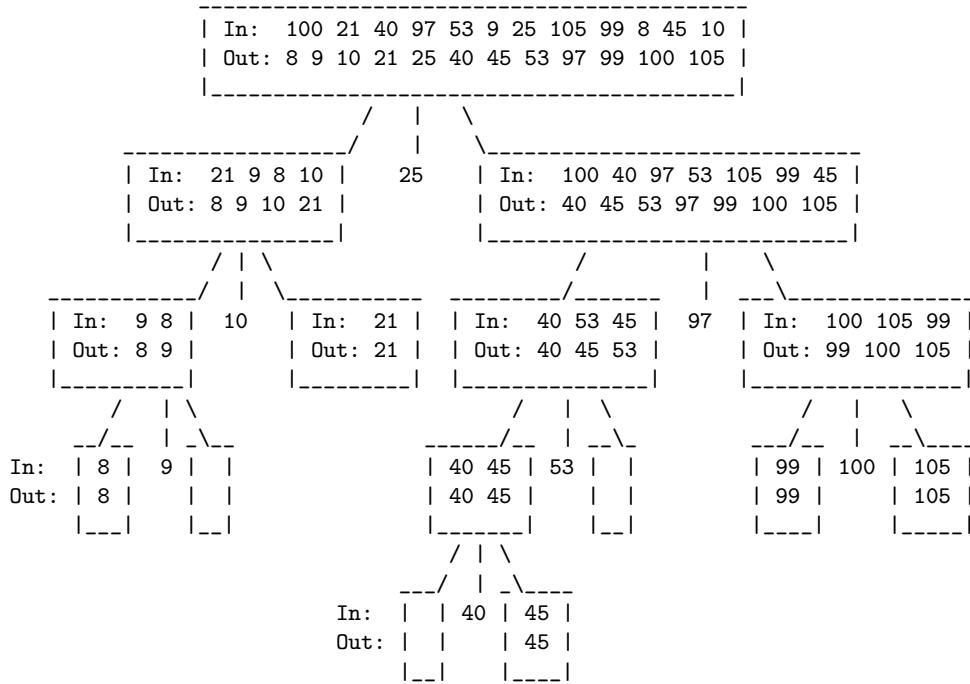
**Tree of Stack Frames:** The following is a tree of stack frames for a concrete example.



**Quick Sort (Minimal work to recombine the halves):** The following is one of the fastest sorting algorithms. Hence, the name.

**Friend's Level of Abstraction:** The first step in the algorithm is to choose one of the elements to be the “pivot element.” How this is to be done is discussed below. The next step is to *partition* the list into two sublists using the Partition routine defined below. This routine rearranges the elements so that all the elements that are less than or equal to the pivot element are to the left of the pivot element and all the elements that are greater than it are to the right of it. (There are no requirements on the order of the elements in the sublists.) Next, recursively have a friend sort those elements before the pivot and those after it. Finally, (without effort) put the sublists together, forming one completely sorted list.

**Tree of Stack Frames:** The following is a tree of stack frames for a concrete example.



**Running Time:** The computation time depends on the choice of the pivot element.

**Median:** If we are lucky and the pivot element is close to being the median value, then the list will be split into two sublists of size approximately  $n/2$ . We will see that partitioning the array according to the pivot element can be done in time  $\Theta(n)$ . In this case, the timing is  $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$ .

**Reasonable Split:** The above timing is quite robust with respect to the choice of the pivot. For example, suppose that the pivot always partitions the list into one sublist of  $\frac{1}{5}th$  the original size and one of  $\frac{4}{5}th$  the size. The total time is then the time to partition plus the time to sort the sublists of these sizes. This gives  $T(n) = T(\frac{1}{5}n) + T(\frac{4}{5}n) + \Theta(n)$ . Because  $\frac{1}{5} + \frac{4}{5} = 1$ , this evaluates to  $T(n) = \Theta(n \log n)$ . (See Section 6.)

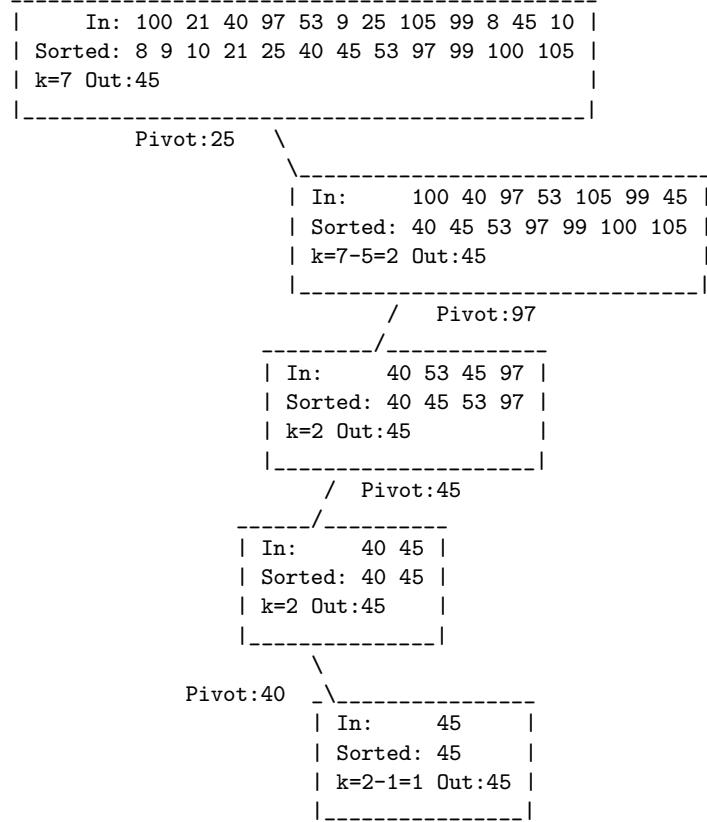
**Worst Case:** On the other hand, suppose that the pivot always splits the list into one of size  $n - 1$  and one of size 1. In this case,  $T(n) = T(n - 1) + T(1) + \Theta(n)$ , which evaluates to  $T(n) = \Theta(n^2)$ . This is the worst case scenario.

We will return to Quick Sort after considering the following related problem.

**Finding the  $k^{th}$  Smallest Element:** Given an unsorted list and an integer  $k$ , this problem finds the  $k^{th}$  smallest element from the list. It is not clear at first that there is an algorithm for doing this that is any faster than sorting the entire list. However, it can be done in linear time using the subroutine Pivot.

**Friend's Level of Abstraction:** The algorithm is like that for binary search. Ignoring input  $k$ , it proceeds just like quick sort. A pivot element is chosen randomly, and the list is split into two sublists, the first containing all elements that are all less than or equal to the pivot element and the second those that are greater than it. Let  $\ell$  be the number of elements in the first sublist. If  $\ell \geq k$ , then we know that the  $k^{th}$  smallest element from the entire list is also the  $k^{th}$  smallest element from the first sublist. Hence, we can give this first sublist and this  $k$  to a friend and ask him to find it. On the other hand, if  $\ell < k$ , then we know that the  $k^{th}$  smallest element from the entire list is the  $(k-\ell)^{th}$  smallest element from the second sublist. Hence, giving the second sublist and  $k-\ell$  to a friend, he can find it.

**Tree of Stack Frames:** The following is a tree of stack frames for a concrete example.



**Running Time:** Again, the computation time depends on the choice of the pivot element.

**Median:** If we are lucky and the pivot element is close to being the median value, then the list will be split into two sublists of size approximately  $n/2$ . Because the routine recurses on only one of the halves, the timing is  $T(n) = T(n/2) + \Theta(n) = \Theta(n)$ .

**Reasonable Split:** If the pivot always partitions the list so that the larger half is at most  $\frac{4}{5}n$ , then the total time is at most  $T(n) = T(\frac{4}{5}n) + \Theta(n)$ , which is still linear time,  $T(n) = \Theta(n)$ .

**Worst Case:** In the worst case, the pivot splits the list into one of size  $n - 1$  and one of size 1. In this case,  $T(n) = T(n - 1) + \Theta(n)$ , which is  $T(n) = \Theta(n^2)$ .

**Choosing the Pivot:** In the last two algorithms, the timing depends on choosing a good pivot element quickly.

**Fixed Value:** If you knew that you were sorting elements that are numbers within the range  $[1..100]$ , then it reasonable to partition these elements based on whether they are smaller or larger than 50. This is often referred to as *bucket sort*. See below. However, there are two problems with this technique. The first is that in general we do not know what range the input elements will lie. The second is that at every level of recursion another pivot value is needed with which to partition the elements. The solution is to use the input itself to choose the pivot value.

**Use  $A[1]$  as the Pivot:** The first thing one might try is to let the pivot be the element that happens to be first in the input array. The problem with this is that if the input happens to be sorted (or almost sorted) already, then this first element will split the list into one of size zero and one of size  $n - 1$ . This gives the worst case time of  $\Theta(n^2)$ . Given random data, the algorithm will execute quickly. On the other hand, if you forget that you sorted the data and you run it a second time, then second time will take a long time to complete.

**Use  $A[\frac{n}{2}]$  as the Pivot:** Motivated by the last attempt, one might use the element that happens to be located in the middle of the input array. For all practical purposes, this would likely work great. It would work exceptionally well when the list is already sorted. However, there are some strange inputs cooked ups for the sole purpose of being nasty to this particular implementation of the algorithm on which the algorithm runs in  $\Theta(n^2)$  time. The adversary will provide such an input giving a worst case time complexity of  $\Theta(n^2)$ .

**A Randomly Chosen Element:** In practice, what is often done is to choose the pivot element randomly from the input elements. See Section 27.2. The advantage of this is that the adversary who is choosing the worst case input instance, knows the algorithm, but does not know the random coin tosses. Hence, all input instances are equivalently good and equivalently bad.

We will prove that the expected computation time is  $\Theta(n \log n)$ . What this means is that if you ran the algorithm 1000000 times on the same input, then the average running time would be  $\Theta(n \log n)$ .

**Intuition:** One often gains good intuition by assuming that what we expect to happen happens reasonably often. Recall that if the pivot always partitions the list into one sublist of  $\frac{1}{5}th$  the original size and one of  $\frac{4}{5}th$  the size, then the total time is  $T(n) = T(\frac{1}{5}n) + T(\frac{4}{5}n) + \Theta(n) = \Theta(n \log n)$ . When a pivot is chosen randomly, the probability that it partitions the list at least this well is  $\frac{3}{5}$ . When a partition is worse than this, it is not a big problem. We just say that no significant progress is made and we try again. After all, we expect to make progress approximately three every five partitions.

**More Formal:** Formally, we set up and solve a difficult recurrence relation. Suppose that the randomly chosen pivot element happens to be the  $i^{th}$  smallest element. This splits the list into one of size  $i$  and one of size  $n - i$ , in which case the running

time is  $[T(i) + T(n-i) + \Theta(n)]$ . Averaging this over all possible values of  $i$ , gives the recursive relation  $T(n) = \text{Avg}_{i \in [0..n]} [T(i) + T(n-i) + \Theta(n)]$ . With a little work, this evaluates to  $\Theta(n \log n)$ .

**Randomly Choose 3 Elements:** Another option is to randomly select three elements from the input list and use the middle one as the pivot. Doing this greatly increases the probability that the pivot is close to the middle and hence decreases the probability of the worst case occurring. However, doing so also takes time. All in all, the expected running time is worse.

**A Deterministic Algorithm:** Though in practice the above probabilistic algorithm is easy to code and works well, theoretical computer scientists like to find a deterministic algorithm that is guaranteed to run quickly.

The following is a deterministic method of choosing the pivot that leads to the worst case running time of  $\Theta(n)$  for finding  $k^{th}$  smallest element. First group the  $n$  elements into  $\frac{n}{5}$  groups of 5 elements each. Within each group of 5 elements, do  $\Theta(1)$  work to find the median of the group. Let  $S_{\text{median}}$  be the set of  $\frac{n}{5}$  elements that is the median from each group. Recursively ask a friend to find the median element from the set  $S_{\text{median}}$ . This element will be used as our pivot.

We claim that this pivot element has at least  $\frac{3}{10}n$  elements that are less than or equal to it and another  $\frac{3}{10}n$  elements that are greater or equal to it. The proof of the claim is as follows. Because the pivot is the median within  $S_{\text{median}}$ , there are  $\frac{1}{10}n = \frac{1}{2}|S_{\text{median}}|$  elements within  $S_{\text{median}}$  that are less than or equal to the pivot. Consider any such element  $x_i \in S_{\text{median}}$ . Because  $x_i$  is the median within its group of 5 elements, there are 3 elements within this group (including  $x_i$  itself) that are less than or equal to  $x_i$  and hence in turn less than or equal to the pivot. Counting all these gives  $3 \cdot \frac{1}{10}n$  elements. A similar argument counts this many that are greater or equal to the pivot.

The algorithm to find the  $k^{th}$  largest element proceeds as stated originally. A friend is either asked to find the  $k^{th}$  smallest element within all elements that are less than or equal to the pivot or the  $(k-\ell)^{th}$  smallest element from all those that are greater than it. The claim insures that the size of the sublist given to the friend is at most  $\frac{7}{10}n$ .

Unlike the first algorithm for the finding  $k^{th}$  smallest element, this algorithm recurses twice. Hence, one would initially assume that the running time is  $\Theta(n \log n)$ . However, careful analysis shows that it is only  $\Theta(n)$ . Let  $T(n)$  denote the running time. Finding the median of each of the  $\frac{1}{5}n$  groups takes  $\Theta(n)$  time. Recursively finding the median of  $S_{\text{median}}$  takes  $T(\frac{1}{5}n)$  time. Recursing on the remaining at most  $\frac{7}{10}n$  elements takes at most  $T(\frac{7}{10}n)$  time. This gives a total of  $T(n) = T(\frac{1}{5}n) + T(\frac{7}{10}n) + \Theta(n)$  time. Because  $\frac{1}{5} + \frac{7}{10} < 1$ , this evaluates to  $T(n) = \Theta(n)$ . (See Section 6).

A deterministic Quick Sort algorithm can use this deterministic  $\Theta(n)$  time algorithm for the finding  $k^{th}$  smallest element, to find the median of the list to be the pivot. Because partitioning the elements according to the pivot already takes  $\Theta(n)$  time, the timing is still  $T(n) = 2T(\frac{n}{2}) + \Theta(n) = \Theta(n \log n)$ .

**Partitioning According To The Pivot Element:** The input consists of a list of elements  $A[I], \dots, A[J]$  and a pivot element. The output consists of the rearranged elements and an index  $i$ , such that the elements  $A[I], \dots, A[i-1]$  are all less than or equal to the pivot element,  $A[i]$  is the pivot element, and the elements  $A[i+1], \dots, A[J]$  are all greater than it.

The loop invariant is that there are indices  $I \leq i \leq j \leq J$  for which

1. The values in  $A[I], \dots, A[i - 1]$  are less than or equal to the pivot element.
2. The values in  $A[j + 1], \dots, A[J]$  are greater than the pivot element.
3. The pivot element has been removed and is on the side, leaving an empty entry either at  $A[i]$  or at  $A[j]$ .
4. The other elements in  $A[i], \dots, A[j]$  have not been considered.

The loop invariant is established by setting  $i = I$  and  $j = J$ , making  $A[i]$  empty by putting the element in  $A[i]$  where the pivot element is and putting the pivot element aside.

If the loop invariant is true and  $i < j$ , then there are four possible cases:

**Case A)  $A[i]$  is empty and  $A[j] \leq \text{pivot}$ :**  $A[j]$  belongs on the left, so move it to the empty  $A[i]$ .  $A[j]$  is now empty. Increase the left side by increasing  $i$  by one.

**Case B)  $A[i]$  is empty and  $A[j] > \text{pivot}$ :**  $A[j]$  belongs on the right and is already there. Increase the right side by decreasing  $j$  by one.

**Case C)  $A[j]$  is empty and  $A[i] \leq \text{pivot}$ :**  $A[i]$  belongs on the left and is already there. Increase the left side by increasing  $i$  by one.

**Case D)  $A[j]$  is empty and  $A[i] > \text{pivot}$ :**  $A[i]$  belongs on the right, so move it to the empty  $A[j]$ .  $A[i]$  is now empty. Increase the right side by decreasing  $j$  by one.

In each case, the loop invariant is maintained. Progress is made because  $j - i$  decreases.

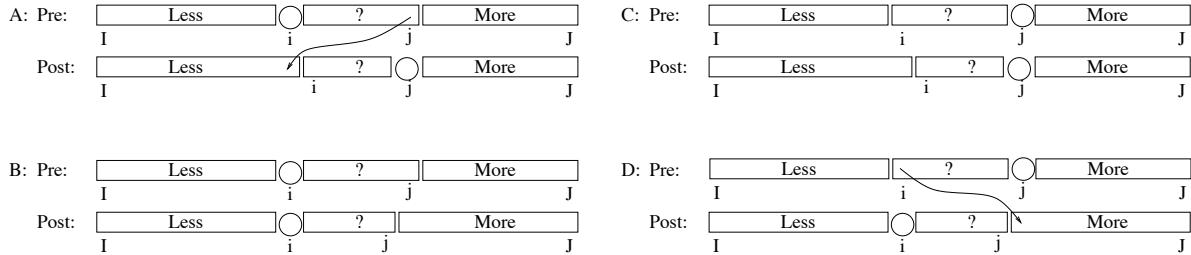


Figure 15.1: The four cases of how to iterate are shown.

When  $i = j$ , the list is split as needed, leaving  $A[i]$  empty. Put the pivot there. The postcondition follows.

**Sorting By Hand:** As a professor, I often have to sort a large stack of student's papers by the last name. Surprisingly enough, the algorithm that I use is an iterative version of quick sort. See 12.1.

## 15.2 Operations on Integers

Raising an integer to a power  $b^N$ , multiplying  $x \times y$ , and matrix multiplication each have surprising divide and conquer algorithms.

**$b^N$ :** Suppose that you are given two integers  $b$  and  $N$  and want to compute  $b^N$ .

**The Iterative Algorithm:** The obvious iterative algorithm simply multiplies  $b$  together  $N$  times. The obvious recursive algorithm recurses with  $\text{Power}(b, N) = b \times \text{Power}(b, N-1)$ . This requires the same  $N$  multiplications.

**The Straightforward Divide and Conquer Algorithm:** The obvious divide and conquer technique cuts the problem into two halves using the property that  $b^{\lceil \frac{N}{2} \rceil} \times b^{\lfloor \frac{N}{2} \rfloor} = b^{\lceil \frac{N}{2} \rceil + \lfloor \frac{N}{2} \rfloor} = b^N$ . This leads to the recursive algorithm  $\text{Power}(b, N) = \text{Power}(b, \lceil \frac{N}{2} \rceil) \times \text{Power}(b, \lfloor \frac{N}{2} \rfloor)$ . Its recurrence relation gives  $T(N) = 2T(\frac{N}{2}) + 1$  multiplications. The technique in Section 6 notes that  $\frac{\log a}{\log b} = \frac{\log 2}{\log 2} = 1$  and  $f(N) = \Theta(N^0)$  so  $c = 0$ . Because  $\frac{\log a}{\log b} > c$ , the technique concludes that time is dominated by the base cases and  $T(N) = \Theta(N^{\frac{\log a}{\log b}}) = \Theta(N)$ . This is no faster than the standard iterative algorithm.

**Reducing the Number of Recursions:** This algorithm can be improved by noting that the two recursive calls are almost the same and hence need only to be called once. The new recurrence relation gives  $T(N) = 1T(\frac{N}{2}) + 1$  multiplications. Here  $\frac{\log a}{\log b} = \frac{\log 1}{\log 2} = 0$  and  $f(N) = \Theta(N^0)$  so  $c = 0$ . Because  $\frac{\log a}{\log b} = c$ , we conclude that time is dominated by all levels and  $T(N) = \Theta(f(N) \log N) = \Theta(\log N)$  multiplications.

**algorithm**  $\text{Power}(b, N)$

**$\langle \text{pre-cond} \rangle$ :**  $N \geq 0$  ( $N$  and  $b$  not both 0)

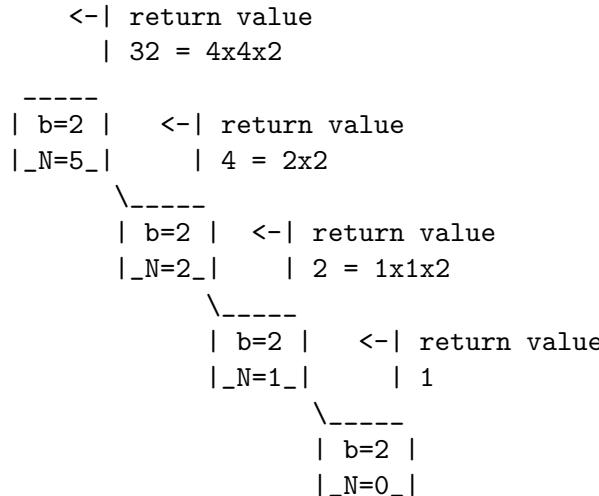
**$\langle \text{post-cond} \rangle$ :** Outputs  $b^n$ .

begin

```

    if(  $N = 0$  ) then
        result 1
    else
        half =  $\lfloor \frac{N}{2} \rfloor$ 
        p =  $\text{Power}(b, \text{half})$ 
        if(  $2 \cdot \text{half} = N$  ) then
            result(  $p \cdot p$  ) % if  $N$  is even,  $b^N = b^{N/2} \cdot b^{N/2}$ 
        else
            result(  $p \cdot p \cdot b$  ) % if  $N$  is odd,  $b^N = b \cdot b^{N/2} \cdot b^{N/2}$ 
        end if
    end if
end algorithm
```

Tree of Stack Frames:



## Running Time:

**Input Size:** One is tempted to say that the first two  $\Theta(N)$  algorithms require a linear number of multiplications and that the last  $\Theta(\log N)$  one requires a logarithmic number. However, in fact the first two require exponential  $\Theta(2^n)$  number and the last a linear  $\Theta(n)$  number in the “size” of the input, which is typically the number of bits  $n = \log N$  to represent the number.

**Operation:** Is it fair to count the number of multiplications and not bit operations in this case? I say not. The output  $b^N$  contains  $\Theta(N \log b) = 2^{\Theta(n)}$  bits and hence it will take this many bit operations to simply output the answer. Given this, it is not really fair to say that the time complexity is only of  $\Theta(n)$ .

**$x \times y$ :** The time complexity of the last algorithm was measured in terms of the number of multiplications. This begs the question of how quickly one can multiply.

The input for the next problem consists of two strings of  $n$  digits each. These are viewed as two integers  $x$  and  $y$  either in binary or in decimal notation. The problem is to multiply them.

**The Iterative Algorithm:** The standard elementary school algorithm considers each pair of digits, one from  $x$  and the other from  $y$ , and multiplies them together. These  $n^2$  products are shifted appropriately and summed. The total time is  $\Theta(n^2)$ . It is hard to believe that one could do faster.

$$\begin{array}{r}
 & 8 & 2 & 7 \\
 & 5 & 9 & 6 \\
 \hline
 & 4 & 2 \\
 & 1 & 2 \\
 4 & 8 & & \\
 6 & 3 & & \\
 1 & 8 & & \\
 7 & 2 & & \\
 3 & 5 & & \\
 1 & 0 & & \\
 \hline
 4 & 0 & 2 & 8 & 9 & 2
 \end{array}$$

**The Straightforward Divide and Conquer Algorithm:** Let us see how well the divide and conquer technique can work. Split each sequence of digits in half and consider each half as an integer. This gives  $x = x_1 \cdot 10^{\frac{n}{2}} + x_0$  and  $y = y_1 \cdot 10^{\frac{n}{2}} + y_0$ . Multiplying these symbolically gives

$$\begin{aligned}
 x \times y &= (x_1 \cdot 10^{\frac{n}{2}} + x_0) \times (y_1 \cdot 10^{\frac{n}{2}} + y_0) \\
 &= (x_1 y_1) \cdot 10^n + (x_1 y_0 + x_0 y_1) \cdot 10^{\frac{n}{2}} + (x_0 y_0)
 \end{aligned}$$

The obvious divide and conquer algorithm would recursively compute the four subproblems  $x_1 y_1$ ,  $x_1 y_0$ ,  $x_0 y_1$ , and  $x_0 y_0$ , each of  $\frac{n}{2}$  digits. This would take  $4T(\frac{n}{2})$  time. Then these four products are shifted appropriately and summed. Note that additions can be done in  $\Theta(n)$  time. See Section 9.2. Hence, the total time is  $T(n) = 4T(\frac{n}{2}) + \Theta(n)$ . Here  $\frac{\log a}{\log b} = \frac{\log 4}{\log 2} = 2$  and  $f(n) = \Theta(n^1)$  so  $c = 1$ . Because  $\frac{\log a}{\log b} > c$ , the technique concludes that time is dominated by the base cases and  $T(n) = \Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^2)$ . This is no improvement in time.

**Reducing the Number of Recursions:** Suppose that we could find a trick so that we only need to recurse three times instead of four. One’s intuition might be that this would only provide a linear time savings, but in fact the savings is much more.  $T(n) = 3T(\frac{n}{2}) + \Theta(n)$ . Now  $\frac{\log a}{\log b} = \frac{\log 3}{\log 2} = 1.58..$ , which is still bigger than  $c = 1$ . Hence, time is still dominated

by the base cases, but now this is  $T(n) = \Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^{1.58..})$ . This is a significant improvement from  $\Theta(n^2)$ .

The trick for requiring only three recursive multiplications is as follows. The first step is to multiply  $x_1y_1$  and  $x_0y_0$  recursively as required. This leaves us only one more recursive multiplication.

If you review the symbolic expansion of  $x \times y$ , you will see that we do not actually need to know the value of  $x_1y_0$  and  $x_0y_1$ . We only need to know their sum. Symbolically, we can observe the following.

$$\begin{aligned} & x_1y_0 + x_0y_1 \\ &= [x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0] - x_1y_1 - x_0y_0 \\ &= [(x_1 + x_0)(y_1 + y_0)] - x_1y_1 - x_0y_0 \end{aligned}$$

Hence, the sum  $x_1y_0 + x_0y_1$  that we need can be computed by adding  $x_1$  to  $x_0$  and  $y_1$  to  $y_0$ ; multiplying these sums; and subtracting off the values  $x_1y_1$  and  $x_0y_0$  that we know from before. This requires only one additional recursive multiplication. Again we use the fact that additions are fast, requiring only  $\Theta(n)$  time.

**algorithm** *Multiply*( $x, y$ )

*{pre-cond}*:  $x$  and  $y$  are two integers represented as an array of  $n$  digits

*{post-cond}*: The output consists of their product represented as an array of  $n + 1$  digits

```

begin
  if(n=1) then
    result(  $x \times y$  ) % product of single digits
  else
     $\langle x_1, x_0 \rangle$  = high and low order  $\frac{n}{2}$  digits of x
     $\langle y_1, y_0 \rangle$  = high and low order  $\frac{n}{2}$  digits of y
    A = Multiply( $x_1, y_1$ )
    C = Multiply( $x_0, y_0$ )
    B = Multiply( $x_1 + x_0, y_1 + y_0$ ) - A - C
    result(  $A \cdot 10^n + B \cdot 10^{\frac{n}{2}} + C$  )
  end if
end algorithm

```

It is surprising that this trick reduces the time from  $\Theta(n^2)$  to  $\Theta(n^{1.58})$ .

**Dividing into More Parts:** The next question is whether the same trick can be extended to improve the time even further. Instead of splitting each of  $x$  and  $y$  into two pieces, let's split them each into  $d$  pieces. The straightforward method recursively multiplies each of the  $d^2$  pairs of pieces together, one from  $x$  and one from  $y$ . The total time is  $T(n) = d^2T(\frac{n}{d}) + \Theta(n)$ . Here  $a = d^2$ ,  $b = d$ ,  $c = 1$ , and  $\frac{\log d^2}{\log d} = 2 > c$ . This gives  $T(n) = \Theta(n^2)$ . Again, we are back where we began.

**Reducing the Number of Recursions:** The trick now is to do the same with fewer recursive multiplications. It turns out it can be done with only  $2d - 1$  of them. This gives time of only  $T(n) = (2d - 1)T(\frac{n}{d}) + \Theta(n)$ . Here  $a = 2d - 1$ ,  $b = d$ ,  $c = 1$ , and  $\frac{\log(2d-1)}{\log(d)} \approx \frac{\log(d)+1}{\log(d)} = 1 + \frac{1}{\log(d)} \approx c$ . By increasing  $d$ , the time for the top stack frame and

for the base cases becomes closer and closer to being equal. Recall that when this happens, we must add an extra  $\Theta(\log n)$  factor to account for the  $\Theta(\log n)$  levels of recursion. This gives  $T(n) = \Theta(n \log n)$ , which is a surprising running time for multiplication.

**Fast Fourier Transformations:** We will not describe the trick for reducing the number of recursive multiplications from  $d^2$  to only  $2d - 1$ . Let it suffice that it involves thinking of the problem as the evaluation and interpolation of polynomials. When  $d$  becomes large, other complications arise. These are solved by using the  $2d$ -roots of unity over a finite field. Performing operations over this finite field require  $\Theta(\log \log n)$  time. This increases the total time from  $\Theta(n \log n)$  to  $\Theta(n \log n \log \log n)$ . This algorithm is used often for multiplication and many other applications such as signal processing. It is referred to as *Fast Fourier Transformations*.

**Exercise 15.2.1** Design the algorithm and compute the running time when  $d = 3$ .

**Strassen's Matrix Multiplication:** The next problem is to multiply two  $n \times n$  matrices.

**The Iterative Algorithm:** The obvious iterative algorithm computes the  $\langle i, j \rangle$  entry of the product matrix by multiplying the  $i^{th}$  row of the first matrix with the  $j^{th}$  column of the second. This requires  $\Theta(n)$  scalar multiplications. Because there are  $n^2$  such entries, the total time is  $\Theta(n^3)$ .

**The Straightforward Divide and Conquer Algorithm:** When designing a divide and conquer algorithm, the first step is to divide these two matrices into four submatrices each. Multiplying these symbolically gives the following.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} = \begin{pmatrix} ae + bf & ag + bh \\ ce + df & cg + dh \end{pmatrix}$$

Computing the four  $\frac{n}{2} \times \frac{n}{2}$  submatrices in this product in this way requires recursively multiplying eight pairs of  $\frac{n}{2} \times \frac{n}{2}$  matrices. The total computation time is given by the recurrence relation  $T(n) = 8T(n/2) + \Theta(n^2) = \Theta(n^{\frac{\log 8}{\log 2}}) = \Theta(n^3)$ . As seen before, this is no faster than the standard iterative algorithm.

**Reducing the Number of Recursions:** Strassen found a way of computing the four  $\frac{n}{2} \times \frac{n}{2}$  submatrices in this product using only seven such recursive calls. This gives  $T(n) = 7T(n/2) + \Theta(n^2) = \Theta(n^{\frac{\log 7}{\log 2}}) = \Theta(n^{2.8073})$ . We will not include the details of the algorithm.

### 15.3 Ackermann's Function

If you are wondering just how slowly a program can run, consider the code below. Assume the input parameters  $n$  and  $k$  are natural numbers.

**Code:**

```
algorithm A(k, n)
if( k = 0) then
    return( n+1+1 )
```

```

else
    if( n = 0) then
        if( k = 1) then
            return( 0 )
        else
            return( 1 )
    else
        return( A(k - 1, A(k, n - 1)))
    end if
end if
end algorithm

```

**Recurrence Relation:** Let  $T_k(n)$  denote the value returned by  $A(k, n)$ . This gives  $T_0(n) = 2 + n$ ,  $T_1(0) = 0$ ,  $T_k(0) = 1$  for  $k \geq 2$ , and  $T_k(n) = T_{k-1}(T_k(n-1))$  for  $k > 0$  and  $n > 0$ .

**Solving:**

$$T_0(n) = 2 + n$$

$$T_1(n) = T_0(T_1(n-1)) = 2 + T_1(n-1) = 4 + T_1(n-2) = 2^i + T_1(n-i) = 2n + T_1(0) = 2n.$$

$$T_2(n) = T_1(T_2(n-1)) = 2 \cdot T_2(n-1) = 2^2 \cdot T_2(n-2) = 2^i \cdot T_2(n-i) = 2^n \cdot T_2(0) = 2^n$$

$$T_3(n) = T_2(T_3(n-1)) = 2^{T_2(n-1)} = 2^{2^{T_3(n-2)}} = \underbrace{2^{2^{2^{\dots^2}}}}_{i}^{T_3(n-i)} = \underbrace{2^{2^{2^{\dots^2}}}}_{n}^{T_3(0)} = \underbrace{2^{2^{2^{\dots^2}}}}_{n}$$

$$T_4(0) = 1. \quad T_4(1) = T_3(T_4(0)) = T_3(1) = \underbrace{2^{2^{2^{\dots^2}}}}_1 = 2.$$

$$T_4(2) = T_3(T_4(1)) = T_3(2) = \underbrace{2^{2^{2^{\dots^2}}}}_2 = 2^2 = 4.$$

$$T_4(3) = T_3(T_4(2)) = T_3(4) = \underbrace{2^{2^{2^{\dots^2}}}}_4 = 2^{2^{2^2}} = 2^{2^4} = 2^{16} = 65,536.$$

Note  $\underbrace{2^{2^{2^{\dots^2}}}}_5 = 2^{65,536} \approx 10^{21,706}$ , while the number of atoms in the universe is less than  $10^{100}$ .

$$T_4(4) = T_3(T_4(3)) = T_3(65,536) = \underbrace{2^{2^{2^{\dots^2}}}}_{65,536}.$$

Ackermann's function is defined to be  $A(n) = T_n(n)$ . As seen  $A(4)$  is bigger than any number in the natural world.  $A(5)$  is unimaginable.

**Running Time:** The only way that the program builds up a big number is by continually incrementing it by one. Hence, the number of times one is added is at least as huge as the value  $T_k(n)$  returned.

**Crashing:** Programs can stop at run-time because of: 1) overflow in an integer value; 2) running out of memory; 3) running out of time. Which is likely to happen first? If the machine's integers are 32 bits, then they hold a value that is about  $10^{10}$ . Incrementing up to this value will take a long time. However, much worse than this, each two increments needs another recursive call creating a stack of about this many recursive stack frames. The machine is bound to run out of memory first.

# Chapter 16

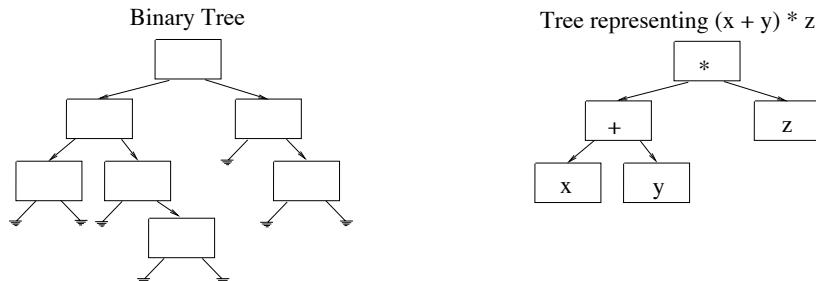
## Recursion on Trees

One key application of recursive algorithms is to perform actions on trees. The reason is that trees themselves have a recursive definition.

**Recursive Definition of Tree:** A tree is either:

- an empty tree (zero nodes) or
- a root node with some subtrees as children.

A *binary tree* is a special kind of tree where each node has a right and a left subtree.



### 16.1 Abstractions, Techniques, and Theory

To demonstrate the ideas given in Section 14, we will now develop a recursive algorithm that will compute the number of nodes in a binary tree.

**The Steps:**

**Specifications:**

**Preconditions:** The input is any binary tree. Note that trees with an empty subtree are valid trees. So are trees consisting of a single node and the empty tree.

**Postconditions:** The output is the number of nodes in the tree.

**Size:** The “size” of an instance is the number of nodes in it.

**General Input:** Consider a large binary tree with two complete subtrees.

**Magic:** We assume that by “magic” a friend is able to count the number of nodes in any tree that is strictly smaller than ours.

**Subinstances:** The subinstances of our instance tree will be the tree's left and its right subtree. These are valid instances that are strictly smaller than ours because the root (and the other subtree) has been removed.

**Subsolutions:** We ask one friend to (recursively) count the number of nodes in the left subtree and another friend for that in the right subtree.

**Solution:** The number of nodes in our tree is the number in its left subtree plus the number in its right subtree plus one for the root.

**Other Instances:** Above we considered a large binary tree with two complete subtrees. Now we will try having the instance be a tree with the right subtree missing. Surprisingly, the above algorithm still works. The number of nodes in our tree's right subtree is zero. This is the answer that our friend will return. Hence, the above algorithm returns the number in its left subtree plus zero plus one for the root. This is the correct answer. Similarly, the above algorithm works when the left subtree is empty or when the instance consists of a single leaf node. The remaining instance is the empty tree. The above algorithm does not work for it, because it does not have any subtrees. Hence, the algorithm can handle all trees except the empty tree with one piece of code.

**Base Cases:** The empty tree is sufficiently small that we can solve it in a brute force way. The number of nodes in it is zero.

**The Tree of Stack Frames:** There is one recursive stack frame for each node in the tree and the tree of stack frames directly mirrors the structure of the tree.

**Running Time:** Because there is one recursive stack frame for each node in the tree and each stack frame does a constant amount of work, the total time is linear in the number of nodes in the input tree. The recurrence relation is  $T(n) = T(n_{left}) + T(n_{right}) + \Theta(1)$ . Plugging the guess  $T(n) = cn$ , gives  $cn = cn_{left} + cn_{right} + \Theta(1)$ , which is correct because  $n = n_{left} + n_{right} + 1$ .

**Code:**

```

algorithm NumberNodes(tree)
  <pre-cond>: tree is a binary tree.
  <post-cond>: Returns the number of nodes in the tree.

  begin
    if( tree = emptyTree ) then
      result( 0 )
    else
      result( NumberNodes(leftSub(tree))
              +NumberNodes(rightSub(tree)) + 1 )
    end if
  end algorithm

```

Note, that we ensured that the algorithm developed works for every valid input instance.

**Common Bugs with Base Cases:** Many people are tempted to use trees with a single node as the base case. A minor problem with this is that it means that the routine no longer works for the empty tree, i.e., the tree with zero nodes. A bigger problem is that the routine no longer works for trees that contain a node with a left child but no right child, or visa versa. This tree is not a base case because it has more than one node. However, when the routine

recurses on the right subtree, the new subinstance consists of the empty tree. The routine, however, no longer works for this tree.

Another common bug that people make is to provide the wrong answer for the empty tree. When in doubt as to what answer should be given for the empty tree, consider an instance with the left or right subtree empty. What answer do you need to receive from the empty tree to make this tree's answer correct.

**Height:** For example, a tree with one node can either be defined to have height 0 or height 1. It is your choice. However, if you say that it has height 0, then be careful when defining the height of the empty tree.

**IsBST:** Another example is that people often say that the empty tree is not a binary search tree (Section 10.5). However, it is. A binary tree fails to be a binary search tree when certain relationships between the nodes exist. Because the empty tree has no nodes, none of these violating conditions exist. Hence, by default it is a binary search tree.

**Max:** What is the maximum value within an empty list of values? One might think 0 or  $\infty$ . However, a better answer is  $-\infty$ . When adding a new value, one uses the code  $newMax = \max(oldMax, newValue)$ . Starting with  $oldMax = -\infty$ , gives the correct answer when the first value is added.

**Exercise 16.1.1** *Many texts that present recursive algorithms for trees do not consider the empty tree to be a valid input instance. This seems to lack the ability to think abstractly. By not considering empty trees the algorithm requires many more cases. Redesign the above algorithm to return the number of nodes in the input tree. However, now do it without considering the empty tree.*

## 16.2 Simple Examples

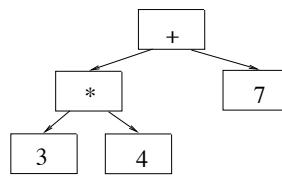
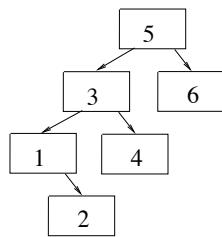
Here is a list of problems involving binary trees. Before looking at the recursive algorithms given below, try developing them yourself.

1. Printing the contents of each node in prefix, infix, and postfix order.
2. Returns the sum of the values within the nodes.
3. Returns the maximum of data fields of nodes.
4. Returns the height of tree.
5. Returns the number of leaves in the tree. (A harder one.)
6. Copy Tree.
7. Deallocate Tree.
8. Searches for a key within a binary search tree.

Think of your own.

The Solutions for these Problems are as follows.

Printing:



```

algorithm PreFix(treeObj tree)
  if tree not= emptyTree then
    put rootInfo(tree)
    PreFix(leftSub(tree))
    PreFix(rightSub(tree));
  end if
end PreOrder
Output: 531246
      +*347
  
```

```

algorithm PostFix(treeObj tree)
  if tree not= emptyTree then
    PostFix(leftSub(tree))
    PostFix(rightSub(tree))
    put rootInfo(tree)
  end if
end PostOrder
Output: 214365
      34*7+
  
```

```

algorithm InFix(treeObj tree)
  if tree not= emptyTree then
    InFix(leftSub(tree))
    put rootInfo(tree)
    InFix(rightSub(tree))
  end if
end InOrder
Output: 123456
      3*4+7
  
```

*PreFix* visits the nodes in the same order that a depth-first search finds the nodes. See Section 20.4 for the iterative algorithm for doing depth-first search of a more general graph and Section 20.5 for the recursive version of the algorithm.

Sum:

```

algorithm Sum(tree)
<pre-cond>: tree is a binary tree.
<post-cond>: Returns the sum of data fields of nodes.
  
```

```

begin
  if( tree = emptyTree ) then
    result( 0 )
  else
    result( Sum(leftSub(tree)) + Sum(rightSub(tree)) + rootData(tree) )
  end if
end algorithm
  
```

### Maximum:

**algorithm** *Max(tree)*

*⟨pre-cond⟩:* *tree* is a binary tree.

*⟨post-cond⟩:* Returns the maximum of data fields of nodes.

```
begin
    if( tree = emptyTree ) then
        result(  $-\infty$  )
    else
        result( max(Max(leftSub(tree)), Max(rightSub(tree)), rootData(tree)) )
    end if
end algorithm
```

### Height:

**algorithm** *Height(tree)*

*⟨pre-cond⟩:* *tree* is a binary tree.

*⟨post-cond⟩:* Returns the height of the tree measured in nodes.

Eg. a tree with one node has height 1.

```
begin
    if( tree = emptyTree ) then
        result( 0 )
    else
        result( max(Height(leftSub(tree)), Height(rightSub(tree))) + 1 )
    end if
end algorithm
```

**algorithm** *Height(tree)*

*⟨pre-cond⟩:* *tree* is a binary tree.

*⟨post-cond⟩:* Returns the height of the tree measured in edges.

Eg. a tree with one node has height 0.

```
begin
    if( tree = emptyTree ) then
        result( -1 )
    else
        result( max(Height(leftSub(tree)), Height(rightSub(tree))) + 1 )
    end if
end algorithm
```

**Number of Leaves:** This problem is harder than previous ones.

**algorithm** *NumberLeaves(tree)*

*(pre-cond)*: *tree* is a binary tree.

*(post-cond)*: Returns the number of leaves in the tree.

begin

```

    if( tree = emptyTree ) then
        result( 0 )
    else if( leftSub(tree) = emptyTree and rightSub(tree) = emptyTree ) then
        result( 1 )
    else
        result( NumberLeaves(leftSub(tree)) + NumberLeaves(rightSub(tree)) )
    end if
end algorithm

```

**Output Values:** Each entry gives value returned by stated subroutine given subtree as input.

Items in tree	# nodes	# leaves	Height	sum
3	10	5	5	26
/ \	/ \	/ \	/ \	/ \
/ \	/ \	/ \	/ \	/ \
2 1	3 6	2 3	2 4	6 17
/ \ / \	/ \ / \	/ \ / \	/ \ / \	/ \ / \
3 1 0 7	1 4 1 1	1 2 1 1	1 3 1 1	3 1 9 7
/ \	/ \	/ \	/ \	/ \
3 2	1 2	1	1 2	3 6
/	/	/	/	/
4	1	1	1	4

**Copy Tree:** If one wants to make a copy of a tree, one might be tempted to use the code *treeCopy* = *tree*. However, the effect of this will only be that both the variables *treeCopy* and *tree* refer to the same tree data structure that *tree* originally did. This is sufficient if one only wants to have read access to the data structure from both variables. However, if one wants to modify one of the copies, then one needs to have a completely separate copy. To obtain this, the copy routine must allocate memory for each of the nodes in the tree, copy over the information in each node, and link the nodes together in the appropriate way. The following simple recursive algorithm, *treeCopy* = *Copy(tree)*, accomplishes this.

**algorithm** *Copy(tree)*

*(pre-cond)*: *tree* is a binary tree.

*(post-cond)*: Returns a copy of the tree.

begin

```

    if( tree = emptyTree ) then
        result( emptyTree )

```

```

    else
        treeCopy = allocate memory for one node
        rootInfo(treeCopy) = rootInfo(tree)      % copy overall data in root node
        leftSub(treeCopy) = Copy(leftSub(tree))% copy left subtree
        rightSub(right) = Copy(rightSub(tree)) % copy right subtree
        result( treeCopy )
    end if
end algorithm

```

**Deallocate Tree:** If the computer system does not have garbage collection, then it is the responsibility of the programmer to deallocate the memory used by all the nodes of a tree when the tree is discarded. The following recursive algorithm, *Deallocate(tree)*, accomplishes this.

**algorithm** *Deallocate(tree)*  
***⟨pre-cond⟩:*** *tree* is a binary tree.  
***⟨post-cond⟩:*** The memory used by the tree has been deallocated.

```

begin
    if( tree ≠ emptyTree ) then
        Deallocate(leftSub(tree))
        Deallocate(rightSub(tree))
        deallocate root node pointed to by tree
    end if
end algorithm

```

**Exercise 16.2.1** In the above *Copy* and *Deallocate* routines, how much freedom is there in the order of the lines of the code?

**Search Binary Search Tree:** A binary search tree is a data structure used to store keys along with associated data. For example, the key could be a student number and the data could contain all the student's marks. Each key is stored in a different node of the binary tree. They are ordered such that for each node all the keys in its left subtree are smaller than its key and all those in the right are larger. This problem searches for a key within a binary search tree. The following recursive algorithm for it directly mirrors the iterative algorithm for it given in Section 10.5.

**algorithm** *SearchBST(tree, keyToFind)*  
***⟨pre-cond⟩:*** *tree* is a binary tree whose nodes contain key and data fields. *keyToFind* is a key.  
***⟨post-cond⟩:*** If there is a node with this key is in the tree, then the associated data is returned.

```

begin
    if( tree = emptyTree ) then
        result "key not in tree"

```

```

else if( keyToFind < rootKey(tree) ) then
    result( SearchBST(leftSub(tree), keyToFind) )
else if( keyToFind = rootKey(tree) ) then
    result( rootData(tree) )
else if( keyToFind > rootKey(tree) ) then
    result( SearchBST(rightSub(tree), keyToFind) )
end if
end algorithm

```

### 16.3 Generalizing the Problem Solved

Sometimes when writing a recursive algorithm for a problem it is easier to solve a more general version of the problem, providing more information about the original instance or asking for more information about subinstance. Remember, however, that anything that you ask your friend to do, you must be able to do yourself.

**Example - Is Binary Search Tree:** This problem returns whether or not the given tree is a binary search tree.

#### An Inefficient Algorithm:

```

algorithm IsBSTtree(tree)
⟨pre-cond⟩: tree is a binary tree.
⟨post-cond⟩: The output indicates whether it is a binary search tree.

begin
    if(tree = emptyTree) then
        return Yes
    else if( IsBSTtree(leftSub(tree)) and IsBSTtree(rightSub(tree))
            and Max(leftSub(tree)) ≤ rootKey(tree) ≤ Min(rightSub(tree)) ) then
        return Yes
    else
        return No
    end if
end algorithm

```

**Running Time:** For each node in the input tree, the above algorithm computes the minimum or the maximum value in the node's left and right subtrees. Though these operations are relatively fast for binary search trees, doing it for each node increases the time complexity of the algorithm. The reason is that each node may be traversed by either the *Min* or *Max* routine many times. Suppose for example that the input tree is completely unbalanced, i.e., a single path. For node  $i$ , computing the max of its subtree involves traversing to the bottom of the path and takes time  $n - i$ . Hence, the total running time is  $T(n) = \sum_{i=1..n} n - i = \Theta(n^2)$ . This is far too slow.

**Ask for More Information About Subinstance:** It is better to combine the *IsBSTtree* and the *Min/Max* routines into one routine so that the tree only needs to be traversed once.

In addition to whether or not the tree is a BST tree, the routine will return the minimum and the maximum value in the tree. If our instance tree is the empty tree, then we return that it is

a BST tree with minimum value  $\infty$  and with maximum value  $-\infty$ . (See Common Bugs with Base Cases Chapter 16.) Otherwise, we ask one friend about the left subtree and another about the right. They tell us the minimum and the maximum values of these and whether they are BST trees. If both subtrees are BST trees and  $leftMax \leq rootKey(tree) \leq rightMin$ , then our tree is a BST. Our minimum value is  $\min(leftMin, rightMin, rootKey(tree))$  and our maximum value is  $\max(leftMax, rightMax, rootKey(tree))$ .

**algorithm** *IsBSTtree*(*tree*)  
***⟨pre-cond⟩:*** *tree* is a binary tree.  
***⟨post-cond⟩:*** The output indicates whether it is a binary search tree.  
 It also gives the minimum and the maximum values in the tree.

```

begin
    if(tree = emptyTree) then
        return ⟨Yes,  $\infty$ ,  $-\infty$ ⟩
    else
        ⟨leftIs, leftMin, leftMax⟩ = IsBSTtree(leftSub(tree))
        ⟨rightIs, rightMin, rightMax⟩ = IsBSTtree(rightSub(tree))
        min =  $\min(leftMin, rightMin, rootKey(tree))$ 
        max =  $\max(leftMax, rightMax, rootKey(tree))$ 
        if( leftIs and rightIs and leftMax  $\leq rootKey(tree) \leq rightMin$  ) then
            isBST = Yes
        else
            isBST = No
        end if
        return ⟨isBST, min, max⟩
    end if
end algorithm

```

One might ask why the left friend provides the minimum of the left subtree even though it is not used. There are two related reasons. The first reason is because the postconditions requires that he does so. You can change the postconditions if you like, but whatever contract is made, everyone needs to keep it. The other reason is that the left friend does not know that he is the “left friend.” All he knows is that he is given a tree as input. The algorithm designer must not assume that the friend knows anything about the context in which he is solving his problem other than what he is passed within the input instance.

**Provide More Information about the Original Instance:** Another elegant algorithm for the *IsBST* problem generalizes the problem in order to provide your friend more information about your subinstance. Here the more general problem, in addition to the tree, will provide in range of values  $[min, max]$  and ask whether the tree is a binary search tree with values within this range. The original problem is solved using *IsBSTtree*(*tree*,  $[-\infty, \infty]$ ).

**algorithm** *IsBSTtree*(*tree*,  $[min, max]$ )  
***⟨pre-cond⟩:*** *tree* is a binary tree. In addition,  $[min, max]$  is a range of values.  
***⟨post-cond⟩:*** The output indicates whether it is a binary search tree with values within this range.

```

begin
    if(tree = emptyTree) then
        return Yes
    else if(   rootKey(tree) ∈ [min, max] and
              IsBSTtree(leftSub(tree), [min, rootKey(tree)]) and
              IsBSTtree(rightSub(tree), [rootKey(tree), max]) then
        return Yes
    else
        return No
    end if
end algorithm

```

See Section 16.6 for another example.

## 16.4 Heap Sort and Priority Queues

Heap Sort is a fast sorting algorithm that is easy to implement. Like Quick Sort, it has the advantage of being done “in place” in memory, while Merge and Radix/Counting Sorts require an auxiliary array of memory to transfer the data to. We include Heap Sort within the chapter because Heap Sort is implemented using recursion within a tree data structure.

**Completely Balanced Binary Tree:** We will visualize the values being sorted stored in a binary tree that is completely balanced, namely every level of the tree is completely full except for the bottom level, which is filled in from the left.

**Array Implementation of Balanced Binary Tree:** Because the tree always has this balanced shape, we do not have to bother with the overhead of having nodes with pointers. In actuality, the values are stored in an simple array  $A[1, n]$ . The mapping between the visualized tree structure and the actual array structure is done by index the nodes of the tree  $1, 2, 3, \dots, n$  starting with the root of the tree and filling each level in from left to right.

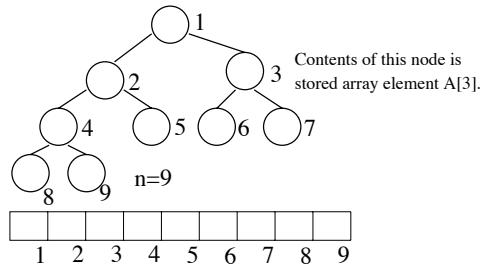


Figure 16.1: The mapping between the nodes in a balanced binary tree and the elements of an array.

- The root is stored in  $A[1]$
- The parent of  $A[i]$  is  $A[\lfloor \frac{i}{2} \rfloor]$ .
- The left child of  $A[i]$  is  $A[2 \cdot i]$ .

- The right child of  $A[i]$  is  $A[2 \cdot i + 1]$ .
- The node in the far right of the bottom level is stored in  $A[n]$ .
- If  $2i + 1 > n$ , then the node does not have a right child.

**Definition of a Heap:** A heap imposes a partial order (see Section 20.6) on the set of values requiring that the value of each node is greater or equal to that of each of the node's children. There are no rules about whether the left or right child is larger.

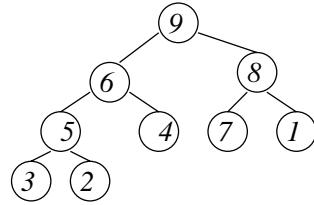


Figure 16.2: An example of nodes ordered into a heap.

**Maximum at Root:** An implication of the heap rules is that the root contains the maximum value. The maximum may appear repeatedly in other places as well.

**Exercise 16.4.1** (*See solution in Section V*) Consider a heap storing the values  $1, 2, 3, \dots, 15$ . Prove the following:

- Where in the heap can the value 1 go?
- Which values can be stored in entry  $A[2]$ ?
- Where in the heap can the value 15 go?
- Where in the heap can the value 6 go?

### The Heapify Problem:

#### Specifications:

**Precondition:** The input is a balanced binary tree such that its left and right subtrees are heaps. (I.e., it is a heap except that its root might not be larger than that of its children.)

**Postcondition:** Its values are rearranged in place to make it complete heap.

**Recursive Algorithm:** Because the left and right subtrees are heaps, the maximums of these trees are at their roots. Hence, the maximum of the *entire* tree is either at the root, its left child, or its right child. Find the maximum between these three. If the maximum is at the root, then we are finished. Otherwise, swap this maximum value with that of the root. The subtree that has a new root now has the property of its left and right subtrees being heaps. Hence, we can recurse to make it into a heap. The entire tree is now a heap.

#### Code:

**algorithm** *Heapify(r)*

***⟨pre-cond⟩:*** The balanced binary tree rooted at  $A[r]$  is such that its left and right subtrees are heaps.

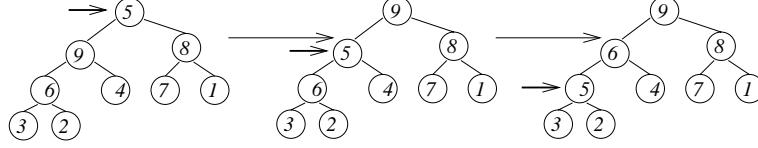


Figure 16.3: An example computation of Heapify.

```

<post-cond>: Its values are rearranged in place to make it complete heap.
begin
    if( $A[\text{rightchild}(r)]$  is max of  $\{A[r], A[\text{rightchild}(r)], A[\text{leftchild}(r)]\}$  ) then
        swap( $A[r], A[\text{rightchild}(r)]$ )
        Heapify(rightchild(r))
    elseif( $A[\text{leftchild}(r)]$  is max of  $\{A[r], A[\text{rightchild}(r)], A[\text{leftchild}(r)]\}$  ) then
        swap( $A[r], A[\text{leftchild}(r)]$ )
        Heapify(leftchild(r))
    else %  $A[r]$  is max of  $\{A[r], A[\text{rightchild}(r)], A[\text{leftchild}(r)]\}$ 
        exit
    end if
end algorithm

```

**Running Time:**  $T(n) = 1 \cdot T(n/2) + \Theta(1)$ . The technique in Section 6 notes that  $\frac{\log a}{\log b} = \frac{\log 1}{\log 2} = 0$  and  $f(n) = \Theta(n^0)$  so  $c = 0$ . Because  $\frac{\log a}{\log b} = c$ , we conclude that time is dominated by all levels and  $T(n) = \Theta(f(n) \log n) = \Theta(\log n)$ .

Because this algorithm recurses only once per call, it is can easily be made into an iterative algorithm.

**Iterative Algorithm:** A good loop invariant would be “The entire tree is a heap except that node  $i$  might not be greater or equal to both of its children. As well, the value of  $i$ ’s parent is at least the value of  $i$  and of  $i$ ’s children.” When  $i$  is the root, this is the precondition. The algorithm proceeds as in the recursive algorithm. Node  $i$  follows one path down the tree to a leaf. When  $i$  is a leaf, the whole tree is a heap.

**Code:**

```
algorithm Heapify(r)
```

***<pre-cond>*:** The balanced binary tree rooted at  $A[r]$  is such that its left and right subtrees are heaps.

***<post-cond>*:** Its values are rearranged in place to make it complete heap.

```
begin
```

```
     $i = r$ 
```

```
loop
```

***<loop-invariant>*:** The entire tree rooted at  $A[r]$  is a heap except that node  $i$  might not be greater or equal to both of its children. As well, the value of  $i$ ’s parent is at least the value of  $i$  and of  $i$ ’s children.

exit when  $i$  is a leaf

```
    if( $A[\text{rightchild}(i)]$  is max of  $\{A[i], A[\text{rightchild}(i)], A[\text{leftchild}(i)]\}$  ) then
```

```
        swap( $A[i], A[\text{rightchild}(i)]$ )
```

```
         $i = \text{rightchild}(i)$ 
```

```
    elseif( $A[\text{leftchild}(i)]$  is max of  $\{A[i], A[\text{rightchild}(i)], A[\text{leftchild}(i)]\}$  ) then
```

```

        swap( $A[i], A[\text{leftchild}(i)]$ )
         $i = \text{leftchild}(i)$ 
    else %  $A[i]$  is max of  $\{A[i], A[\text{rightchild}(i)], A[\text{leftchild}(i)]\}$ 
        exit
    end if
end loop
end algorithm

```

**Running Time:**  $T(n) = \Theta(\text{the height of tree}) = \Theta(\log n)$ .

### The MakeHeap Problem:

#### Specifications:

**Precondition:** The input is an array of numbers, which can be viewed as a balanced binary tree of numbers.

**Postcondition:** Its values are rearranged in place to make it heap.

**Recursive Algorithm:** The obvious recursive algorithm is to recursively make  $\lceil \frac{n-1}{2} \rceil$  of the numbers into a heap, make another  $\lfloor \frac{n-1}{2} \rfloor$  into a heap, and put the remaining number at the root of a tree with these two heaps as children. This now meets the precondition for Heapify, which turns the whole thing into a heap.

**Running Time:**  $T(n) = 2T(\frac{n}{2}) + \Theta(\log n)$ . The technique in Section 6 notes that  $\frac{\log a}{\log b} = \frac{\log 2}{\log 2} = 1$  and  $f(n) = \Theta(n^0 \log n)$  so  $c = 0$ . Because  $\frac{\log a}{\log b} > c$ , we conclude that time is dominated by the base cases and  $T(n) = \Theta(n^{\frac{\log a}{\log b}}) = \Theta(n)$ .

Because the structure of the recursive tree for this algorithm is so predictable, it can easily be made into an iterative algorithm, which calls Heapify on exactly the same nodes though in a slightly different order.

**Iterative Algorithm:** The loop invariant is that all subtrees of height  $i$  are heaps. Initially, the leaves of height  $i = 1$  are already heaps. Suppose that all subtrees of height  $i$  are heaps. The subtrees of height  $i + 1$  have the property that their left and right subtrees are heaps. Hence, we can use Heapify to make them into heaps. This maintains the loop invariant while increasing  $i$  by one. The postcondition clearly follows from the loop invariant and the exit condition that  $i = \log n$ .

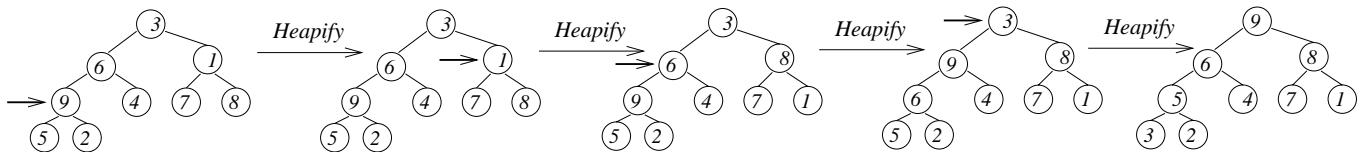


Figure 16.4: An example computation of MakeHeap.

#### Code:

**algorithm** *MakeHeap()*

***⟨pre-cond⟩:*** The input is an array of numbers, which can be viewed as a balanced binary tree of numbers.

***⟨post-cond⟩:*** Its values are rearranged in place to make it heap.

```

begin
    loop  $k = \lceil \frac{n}{2} \rceil, \lceil \frac{n}{2} \rceil - 1, \lceil \frac{n}{2} \rceil - 2, \dots, 2, 1$ 
        Heapify(k)
    end loop
end algorithm

```

**Running Time:** The number of subtrees of height  $i$  is  $2^{(\log n)-i}$ , because each such tree has its root at level  $(\log n) - i$  in the tree. Each take  $\Theta(i)$  to heapify. This gives a total time of  $T(n) = \sum_{i=1}^{\log n} (2^{(\log n)-i}) i$ . This sum is geometric. Hence, its total is theta of its max term. The first term with  $i = 1$  is  $(2^{(\log n)-1}) 1 = \Theta(2^{\log n}) = \Theta(n)$ . The last term with  $i = \log n$  is  $(2^{(\log n)-\log n}) \log n = (2^0) \log n = \log n$ . The first term is the biggest, giving a total time of  $\Theta(n)$ .

### The HeapSort Problem:

#### Specifications:

**Precondition:** The input is an array of numbers.

**Postcondition:** Its values are rearranged in place to be in sorted order.

**Algorithm:** The loop invariant is that for some  $i \in [0, n]$ , the  $n - i$  largest elements have been removed and are sorted on the side and the remaining  $i$  elements form a heap. The loop invariant is established for  $i = n$  by forming a heap from the numbers using the MakeHeap algorithm. When  $i = 0$ , the values are sorted.

Suppose that the loop invariant is true for  $i$ . The maximum of the remaining values is at the root of the heap. Remove it and put it in its sorted place on the left end of the sorted list. Take the bottom right-hand element of the heap and fill the newly created hole at the root. This maintains the correct shape of the tree. The tree now has the property that its left and right subtrees are heaps. Hence, you can use Heapify to make it into a heap. This maintains the loop invariant while decreasing  $i$  by one.

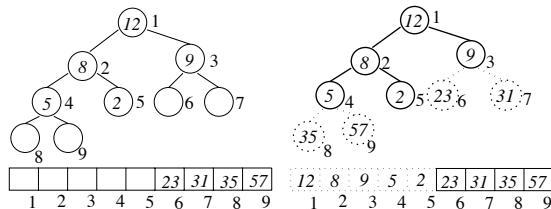


Figure 16.5: The left figure shows the loop invariant with  $n - i = 9 - 5 = 4$  of the largest elements in the array and the remaining  $i = 5$  elements forming a heap. The right figure emphasizes the fact that though a heap is viewed as being stored in a tree, it is actually implemented in an array. When some of the elements are in still in the tree and some are in the array, these views overlap.

**Array Implementation:** The heap sort can occur in place within the array. As the heap gets smaller, the array entries on the right become empty. These can be used to store the sorted list that is on the side. Putting the root element where it belongs, putting the bottom left element at the root, and decreasing the size of the heap can be accomplished by swapping the elements at  $A[1]$  and at  $A[i]$  and decrementing  $i$ .

#### Code:

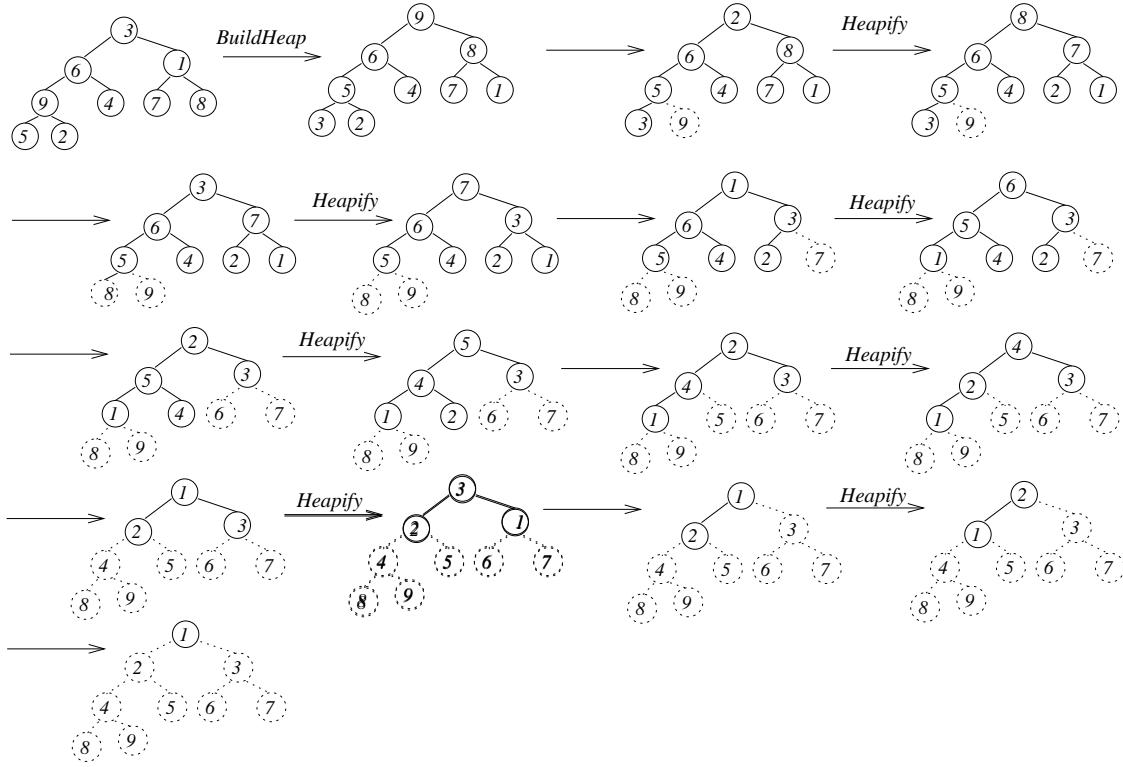


Figure 16.6: An example computation of Heap Sort.

**algorithm** *HeapSort()*

***⟨pre-cond⟩:*** The input is an array of numbers.

***⟨post-cond⟩:*** Its values are rearranged in place to be in sorted order.

begin

*MakeHeap()*

*i* = *n*

    loop

***⟨loop-invariant⟩:*** The  $n - i$  largest elements have been removed and are sorted in  $A[i + 1, n]$  and the remaining *i* elements form a heap in  $A[1, i]$ .

        exit when *i* = 1

        swap( $A[\text{root}]$ ,  $A[i]$ )

*i* = *i* - 1

*Heapify(root)*     % On a heap of size *i*.

    end loop

end algorithm

**Running Time:** *MakeHeap* takes  $\Theta(n)$  time, heapifying a tree of size *i* takes time  $\log(i)$ , for a total of  $T(n) = \Theta(n) + \sum_{i=n}^1 \log i$ . This sum behaves like an arithmetic sum. Hence, its total is *n* times its maximum value, i.e.,  $\Theta(n \log n)$ .

**Common Mistakes when Describing These Algorithms:** A loop invariant describes what the data structure looks like at each point in the computation and express how much work has been completed. It should flow smoothly from the beginning to the end of the algorithm.

At the beginning, it should follow easily from the preconditions. It should progress in small natural steps. Once the exit condition has been met, the postconditions should follow easily from the loop invariant. Precondition and postcondition are not loop invariants. Statements that are *always* true, such as  $1 + 1 = 2$  or “The root is the max of any heap”, give no information about the state of the program within the loop. For Heapify, “The left subtree and the right subtree of the current node are heaps” is useful. However, in the end the subtree becomes a leaf, at which point this loop invariant does not tell you that the whole tree is a heap. For Heap Sort, “The tree is a heap” is great, but how do you get a sorted list from this in the end? Do not run routines without making sure that their preconditions are met, such as having Heap Sort call Heapify without being sure that the left and right subtrees of the given node are heaps.

**Priority Queues:** Like stacks and queues, priority queues are an important ADT.

**Definition:** A *priority queue* consists of:

**Data:** A set of elements. Each element of which is associated with an integer that is referred to as the *priority* of the element.

#### Operations:

**Insert Element:** An element, along with its priority, is added to the queue.

**Change Priority:** The priority of an element already in the queue is changed. The routine is passed a pointer to the element within the priority queue and its new priority.

**Remove an Element:** Removes and returns an element of the highest priority from the queue.

#### Implementations:

Implementation	Insert Time	Change Time	Remove Time
Sorted in an array or linked list by priority	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Unsorted in an array or linked list	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Separate queue for each priority level (to add, go to correct queue; to delete, find first non-empty queue)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\#\text{of priorities})$
Heaps	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

**Heap Implementation:** The elements of a priority queue are stored in a heap ordered according to the priority of the elements.

#### Operations:

**Remove an Element:** The element of the highest priority is at the top of the heap. It can be removed and the heap reheapified as done in Heap Sort.

**Insert Element:** Place the new element in the lower right corner of the heap and then bubble it up the heap until it finds the correct place according to its priority.

**Exercise 16.4.2** Design this algorithm.

**Change Priority:** The routine is passed a pointer to the element within the priority. After making the change, this element is either bubbled up or down the heap, depending on whether the priority has increased or decreased.

**Exercise 16.4.3** Design this algorithm.

## 16.5 Representing Expressions with Trees

We will now consider how to represent multivariate equations using binary trees. We will develop the algorithms to evaluate, copy, differentiate, simplify, and print such an equation. Though these are seemingly complex problems, they have simple recursive solutions.

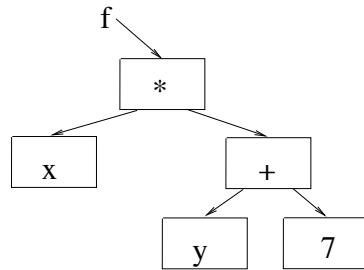
**Recursive Definition of an Expression:** An Expression is either:

- Single variables “x”, “y”, and “z” and single real values are themselves examples of equations.
- If f and g are equations then  $f+g$ ,  $f-g$ ,  $f*g$ , and  $f/g$  are also equations.

**Tree Data Structure:** Note that the above recursive definition of an expression directly mirrors that of a binary tree. Because of this, a binary tree is a natural data structure for storing an equation. (Conversely, you can use an equation to represent a binary tree.)

**Evaluate Equation:** This routine evaluates an equation that is represented by a tree.

**Example:**  $f = x * (y + 7)$ ,  $xvalue = 2$ ,  $yvalue = 3$ ,  $zvalue = 5$ , and returns  $2 * (3 + 7) = 20$ .



**Code:**

```

algorithm Eval(f, xvalue, yvalue, zvalue)
  <pre-cond>: f is an equation whose only variables are x, y, and z. xvalue, yvalue, and zvalue are the three real values to assign to these variables.
  <post-cond>: The returned value is the evaluation of the equation at these values for x, y, and z. The equation is unchanged.

begin
  if( f = a real value ) then
    result( f )
  else if( f = "x" ) then
    result( xvalue )
  else if( f = "y" ) then
    result( yvalue )
  else if( f = "z" ) then
    result( zvalue )
  else if( rootOp(f) = "+" ) then
    result( Eval(leftSub(tree), xvalue, yvalue, zvalue)
           +Eval(rightSub(tree), xvalue, yvalue, zvalue) )
  else if( rootOp(f) = "-" ) then
    result( Eval(leftSub(tree), xvalue, yvalue, zvalue)
           -Eval(rightSub(tree), xvalue, yvalue, zvalue) )
  else
    result( Eval(leftSub(tree), xvalue, yvalue, zvalue)
           *Eval(rightSub(tree), xvalue, yvalue, zvalue) )
end
  
```

```

        -Eval(rightSub(tree), xvalue, yvalue, zvalue) )
else if( rootOp(f) = "*" ) then
    result( Eval(leftSub(tree), xvalue, yvalue, zvalue)
            ×Eval(rightSub(tree), xvalue, yvalue, zvalue) )
else if( rootOp(f) = "/" ) then
    result( Eval(leftSub(tree), xvalue, yvalue, zvalue)
            /Eval(rightSub(tree), xvalue, yvalue, zvalue) )
end if
end algorithm

```

**Differentiate Equation:** This routine computes the derivative of a given equation with respect to an indicated variable.

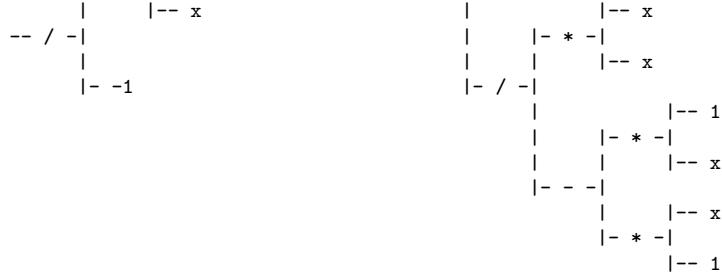
### Specification:

**Preconditions:** The input consists of  $\langle f, x \rangle$ , where  $f$  is an equation represented by a tree and  $x$  is a string giving the name of a variable.

**Postconditions:** The output is the derivative  $d(f)/d(x)$ . This derivative should be an equation represented by a tree whose nodes are separate from those of  $f$ . The data structure  $f$  should remain unchanged.

**Examples:** Rotate the page clockwise  $90^\circ$  to read these equations.

$f = x+y$  -- y --- + -   --- x	$f' = d(f)/d(x)$  -- 0 --- + -   --- 1
$f = x*y$  -- y --- * -   --- x	$f' = d(f)/d(x)$  -- 0  - * -     --- x --- + -     --- y  - * -   --- 1
$f = x/y$  -- y --- / -   --- x	$f' = d(f)/d(x)$  -- y  - * -     --- y --- / -     --- 0    --- x  --- - -     --- y  - * -   --- 1
$f = (x/x)/x$ $i.e. = 1/x$  -- x --- / -     --- x	$f' = d(f)/d(x)$  -- x  - * -     --- x --- / -     --- 1    --- x  --- - -     --- x    --- / -       --- x
<b>Simplify <math>f'</math>:</b>  --- x  - * -	



**Exercise 16.5.1** (See solution in Section V) *Describe the algorithm for Derivative. Do not give the complete code. Only give the key ideas.*

**Exercise 16.5.2** *Trace out the execution of Derivative on the instance  $f = (x/x)/x$  given above. In other words, draw a tree with a box for each time a routine is called. For each box, include only the function  $f$  passed and derivative returned.*

**Simplify Equation:** This routine simplifies a given equation.

**Specification:**

**Preconditions:** The input consists of an equation  $f$  represented by a tree.

**Postconditions:** The output is another equation that is a simplification of  $f$ . Its nodes should be separate from those of  $f$  and  $f$  should remain unchanged.

**Examples:** The equation created by Derivative will not be in the simplest form. For example, the derivative of  $x*y$  with respect to  $x$  will be computed to be:  $1*y + x*0$ . This should be simplified to  $y$ . See above for another example.

**Code:**

```

algorithm Simplify(f)
  <pre-cond>: f is an equation.
  <post-cond>: The output is a simplification of this equation.

begin
  if( f = a real value or a single variable ) then
    result( Copy(f) )
  else % f is of the form (g op h)
    g = Simplify(leftSub(f))
    h = Simplify(rightSub(f))
    if( one of the following forms apply
        1 * h = h      g * 1 = g      0 * h = 0      g * 0 = 0
        0 + h = h      g + 0 = g      g - 0 = g      x - x = 0
        0/h = 0        g/1 = g      g/0 = infinity  x/x = 1
        6 * 2 = 12     6/2 = 3      6 + 2 = 8      6 - 2 = 4      ) then
      result( the simplified form )
    else
      result( Copy(f) )
    end if
  end if
end algorithm

```

**Exercise 16.5.3** Trace out the execution of *Simplify* on the derivative  $f'$  given above, where  $f = (x/x)/x$ . In other words, draw a tree with a box for each time a routine is called. For each box, include only the function  $f$  passed and simplified expression returned.

## 16.6 Pretty Tree Print

The *PrettyPrint* problem is to print the contents of a binary tree in ASCII in a format that looks like a tree.

### Specification:

**Precondition:** The input consists of an equation  $f$  represented by a tree.

**Postcondition:** The tree representing  $f$  is printed sideways on the page. Rotate the page clockwise  $90^\circ$ . Then the root of the tree is at the top and the equation can be read from the left to the right. To make the task of printing more difficult, the program does not have random access to the screen. Hence, the output text must be printed in the usual character by character fashion.

**Examples:** For example, consider the tree representing the equation  $[17 + [Z \times X]] \times [[Z \times Y]/[X + 5]]$ . The *PrettyPrint* output for this tree is:

```

          |--- 5
          |--- +
          |   |--- x
          |--- /
          |   |--- y
          |   |--- *
          |       |--- z
-- * ---+
          |       |--- x
          |       |--- *
          |       |--- z
          |--- +
          |   |--- 17

```

**First Attempt:** The first thing to note is that there is a line of output for each node in the tree and that these appear in the reverse order from the standard *infix* order. See Section 16. We reverse the order by switching the left and right subroutine calls.

**algorithm** *PrettyPrint*( $f$ )

*⟨pre-cond⟩*:  $f$  is an equation.

*⟨post-cond⟩*: The equation is printed sideways on the page.

```

begin
  if(  $f$  = a real value or a single variable ) then
    put  $f$ 
  else
    PrettyPrint(rightSub( $f$ ))
    put rootOp( $f$ )
    PrettyPrint(leftSub( $f$ ))
  end if
end algorithm

```

The second observation is that the information for each node is indented four spaces for each level of recursion.

**Exercise 16.6.1** (*See solution in Section V*) Change the above algorithm so that the information for each node is indented four spaces for each level of recursion.

What remains is to determine how to print the branches of the tree.

**Generalizing the Problem Solved:** Consider the example instance  $[17 + [Z \times X]] \times [[Z \times Y] / [X + 5]]$  given above. One stack frame (friend) during the execution will be given the subtree  $[Z \times Y]$ . The task of this stack frame is more than that specified by the postconditions of PrettyPrint. It must print the following lines of the larger image.

```

|     |     |-- y
|     |     * -|
|     |     |-- z

```

We will break this subimage into three blocks.

**PrettyPrint Image:** The right most part is the output of PrettyPrint for the give subinstance  $[Z \times Y]$ .

```

      |-- y
- * -|
      |-- z

```

**Branch Image:** The column of characters to the left of this PrettyPrint tree consists of a branch of the tree. This branch goes to the right (up on the page) if the given subtree is the left child of its parent and to the left (down on the page) if the subtree is the right child. Including this branch gives the following image.

```

|     |-- y
|- * -|
      |-- z

```

Whether this subinstance is the left or the right child of its parent will be passed as an extra input parameter,  $dir \in \{root, left, right\}$ .

**Left Most Block:** To the left of the column containing a branch is another block of the image. This block consists of the branches within the larger tree that cross over the PrettyPrint of the subtree. Again this image depends on the ancestors of the subtree  $[Z \times Y]$  within the original tree. After playing with a number of examples, one can notice that this block of the image has the interesting property that it consists of the same string of characters repeated each line. The extra input parameter  $prefix$  will simply be the string of characters contained in this string. In this example, the string is “bbbbbb|bbbbbb”. Here the character ‘b’ is used to indicate a blank.

**GenPrettyPrint:** This routine is the generalization of the PrettyPrint routine.

#### Specification:

**Precondition:** The input consists of  $\langle prefix, dir, f \rangle$  where  $prefix$  is a string of characters,  $dir \in \{root, left, right\}$ , and  $f$  is an equation represented by a tree.

**Postcondition:** The output is an image printed in the usual character by character fashion. The resulting image consists of following three blocks

**PrettyPrint Image:** First the expression given by  $f$  is printed as required for PrettyPrint.

**Branch Image:** If  $dir = left$ , then the subtree  $f$  is the left child of its parent and a branch is added to the image extending from the root of the PrettyPrint image to the right (up on the page). If  $dir = right$ , then its the left child and this branch extends to the left (down on the page). If  $dir = root$ , its the root of the tree and no additional branches are needed.

**Left Most Block:** Finally, each line of the resulting image is prefixed with the string given in  $prefix$ .

**Examples:**

Input <‘‘aaaa’’,root,[y*z]>	Input <‘‘aaaa’’,left,[y*z]>	Input <‘‘aaaa’’,right,[y*z]>
Output aaaa  -- z aaaa-- * -  aaaa  -- y	Output aaaa   -- z aaaa - * -  aaaa  -- y	Output aaaa  -- z aaaa - * -  aaaa   -- y

**Code for PrettyPrint:**

```
algorithm PrettyPrint(f)
⟨pre-cond⟩:  $f$  is an equation.
⟨post-cond⟩: The equation is printed sideways on the page.
```

```
begin
    GenPrettyPrint( “”, root,  $f$  )
end algorithm
```

**Subinstances of GenPrettyPrint:** As the routine  $GenPrettyPrint$  recurses, the tree  $f$  within the instance gets smaller and the string  $prefix$  gets longer. In addition to our  $prefix$ , our friends will be asked to print six extra characters on each line which produces the branch to the left or right.

**Code for GenPrettyPrint:**

```
algorithm GenPrettyPrint(prefix,dir,f)
⟨pre-cond⟩: prefix is a string of characters, dir is one of {root,left,right},
            and  $f$  is an equation.
⟨post-cond⟩: The image is printed as described above.

% Determine the character in the ‘‘branch’’
if( dir=root ) then           if( dir=left ) then           if( dir=right) then
    branch_right = ‘‘ ’          branch_right = ‘‘|’           branch_right = ‘‘ ’
    branch_root  = ‘‘-’           branch_root  = ‘‘|’           branch_root  = ‘‘|’
    branch_left   = ‘‘ ’           branch_left   = ‘‘ ’           branch_left   = ‘‘|’
end if                         end if                         end if

if( $f$ = a real value or a single variable ) then
    put prefix + branch_root + ‘‘-- ‘‘ + f
else
    GenPrettyPrint(prefix + branch_right + ‘‘bbbbbb’’, right, rightSub(f))
    put             prefix + branch_root + ‘‘-b’’ + rootOp(f) + ‘‘b-’’ + ‘‘|’’
    GenPrettyPrint(prefix + branch_left + ‘‘bbbbbb’’, left, leftSub(f))
end if
```

**Exercise 16.6.2** (*See solution in Section V*) Trace out the execution of *PrettyPrint* on the instance  $f = 5 + [1 + 2/4] \times 3$ . In other words, draw a tree with a box for each time a routine is called. For each box, include only the values of *prefix* and *dir* and what output is produced by the execution starting at that stack frame.

# Chapter 17

## Recursive Images

Recursion can be used to construct very complex and beautiful pictures. We begin by combining the same two fixed images recursively over and over again. This produces fractal like images those substructures are identical to the whole. Next we will generate random mazes by use randomness to slightly modify these two images so that the substructures are not identical.

### 17.1 Drawing a Recursive Image from a Fixed Recursive and Base Case Images

**Drawing An Image:** An image is specified by a set of lines, circles, and arcs and by two points  $A$  and  $B$  that are referred to as the “handles.” Before such an image can be drawn on the screen, its location, size, and orientation on the screen need to be specified. We will do this by specifying two points  $A$  and  $B$  on the screen. Then a simple program is able to translate, rotate, scale, and draw the image on the screen in such a way that the two handle points of the image land on these two specified points on the screen.

**Specifying A Recursive Image:** A recursive image is specified by the following.

1. a “base case” image
2. a “recurse” image
3. a set of places within the recurse image to “recurse”
4. the two points  $A$  and  $B$  on the screen at which the recursive image should be drawn.
5. an integer  $n$

**The Base Case:** If  $n = 1$ , then the base case image is drawn.

**Recursing:** If  $n > 1$ , then the recurse image is drawn on the screen at the location specified. Included in the recurse image are a number of “places to recurse.” These are depicted by an arrow “ $\rightarrow >$ ”. When the recurse image is translated, rotated, scaled, and drawn on the screen these arrows are located some where on the screen. The arrows themselves are not drawn. Instead, the same picture is drawn recursively at these locations but with the value  $n - 1$ .

**Examples:**

**Man Recursively Framed:** See Figure 17.1a. The base case for this construction consists of a happy face. Hence, when “ $n = 1$ ”, this face is drawn. The recurse image consists of a man holding a frame. There is one place to recurse within the frame. Hence, when  $n = 2$ , this man is drawn with the  $n = 1$  happy face inside of it. For  $n = 3$ , the man is holding a frame containing the  $n = 2$  image of a man holding a framed  $n = 1$  happy face. The recursive image provided is with  $n = 5$ . It consists of a man holding a picture of a man holding a picture of a man holding a picture of ... a face. In general, the recursive image for  $n$  contains  $R(n) = R(n - 1) + 1 = n - 1$  men and  $B(n) = B(n - 1) = 1$  happy faces.

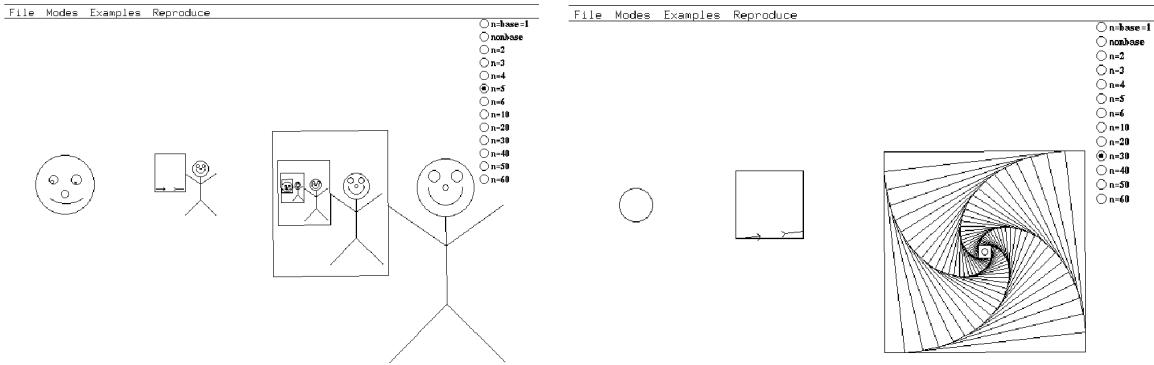


Figure 17.1: a) Man Recursively Framed, b)Rotating Square

**Rotating Square:** See Figure 17.1b. This image is similar to the “Man Recursively Framed” construction. Here, however, the  $n = 1$  base case consists of a circle. The recurse image consists of a single square with the  $n - 1$  image shrunk and rotated within it. The squares continue to spiral inward until the base case is reached.

**Birthday Cake:** See Figure 17.2. The birthday cake recursive image is different in that it recurses in two places. The  $n = 1$  base case consists of a single circle. The recursive image consists of a single line with two smaller copies of  $n - 1$  drawn above it. In general, the recursive image for  $n$  contains  $R(n) = 2R(n - 1) + 1 = 2^{n-1} - 1$  lines from the recurse image and  $B(n) = 2B(n - 1) = 2^{n-1}$  circles from the base case image.

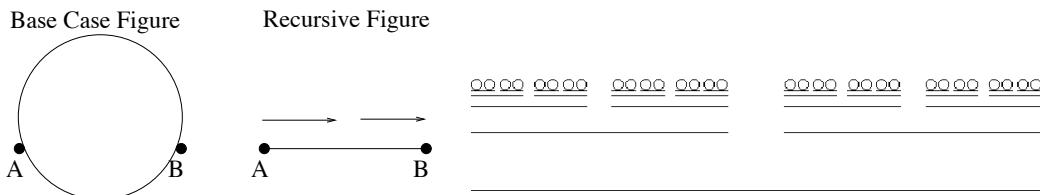


Figure 17.2: Birthday Cake

**Leaf:** See Figure 17.3. A leaf consists of a single stem plus eight sub-leaves along it. Each sub-leaf is an  $n - 1$  leaf. The base case image is empty and the recurse image consists of the stem plus the eight places to recurse. Hence, the  $n = 1$  image is blank. The  $n = 2$  image consists of a lone stem.  $n = 3$  is a stem with eight stems for leaves and so on. In general, the recursive image for  $n$  contains  $R(n) = 8R(n - 1) + 1 = \frac{1}{7}(8^{n-1} - 1)$  stems from the recurse image.

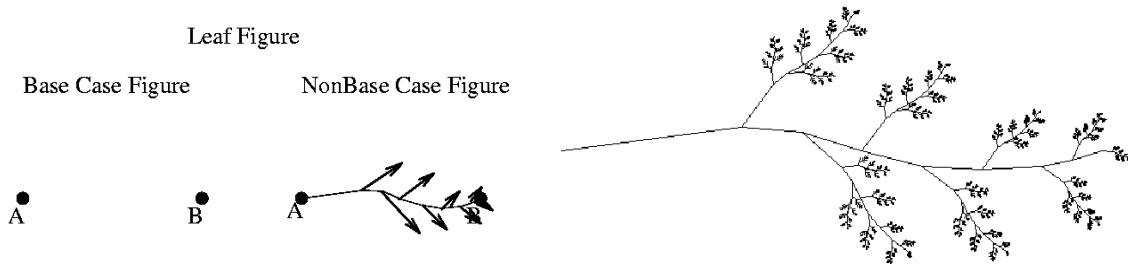


Figure 17.3: Leaf

**Fractal:** See Figure 17.4. This recursive image is a classic. The base case is a single line.

The recurse image is empty except for four places to recurse. Hence,  $n = 1$  consists of the line.  $n = 2$  consists of four lines, forming a line with an equilateral triangle jutting out of it. As  $n$  becomes large, the image becomes a snowflake. It is a fractal in that every piece of it looks like a copy of the whole.

The classic way to construct it is slightly different than done here. In the classical method, one is allowed the following operation on a line. Given a line, it is divided into three equal parts. The middle part is replaced with the two equal length line segments forming an equal lateral triangle. Starting with a single line, the fractal is constructed by repeatedly applying this operation over and over again to all the lines that appear.

In general, the recursive image for  $n$  contains  $B(n) = 4B(n - 1) = 4^{n-1}$  base case lines. The length of each of these lines is  $L(n) = \frac{1}{3}L(n - 1) = \left(\frac{1}{3}\right)^{n-1}$ . The total length of all these lines is  $B(n) \cdot L(n) = \left(\frac{4}{3}\right)^{n-1}$ . Note that as  $n$  approaches infinity, the fractal becomes a curve of infinite length.

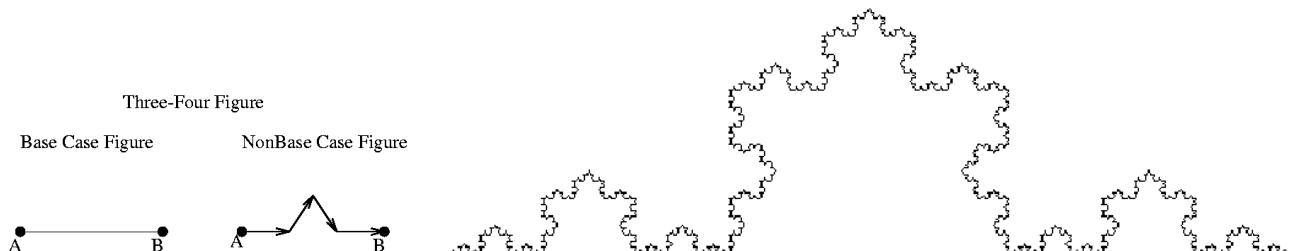


Figure 17.4: Fractals

**Exercise 17.1.1** (See solution in Section V) See Figure 17.5.a. Construct the recursive image that arises from the base case and recurse image for some large  $n$ . Describe what is happening.

**Exercise 17.1.2** (See solution in Section V) See Figure 17.5.b. Construct the recursive image that arises from the base case and recurse image for some large  $n$ . Note that one of the places to recurse is pointing in the other direction. To line the image up with these arrows, the image must be rotated  $180^\circ$ . The image cannot be flipped.

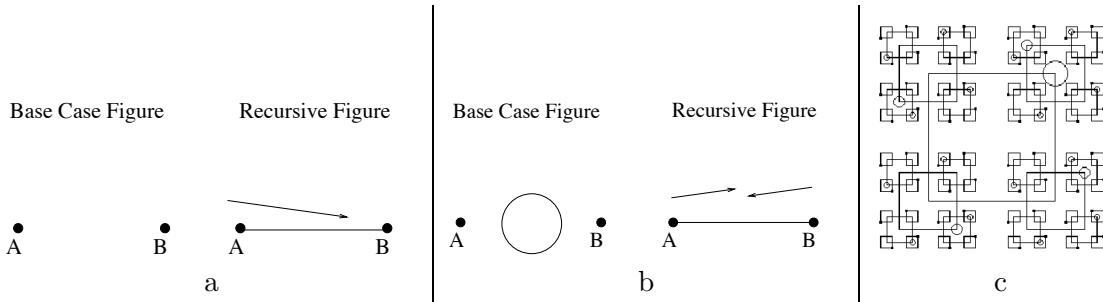


Figure 17.5: Three more examples

**Exercise 17.1.3** (See solution in Section V) See Figure 17.5.c. This construction looks simple enough. The difficulty is keeping track of at which corners the circle is. Construct the base case and the recurse image from which the following recursive image arises. Describe what is happening.

## 17.2 Randomly Generating a Maze

We will use similar methods to generate a random maze. The maze  $M$  will be represented by an  $n \times m$  two dimensional array with entries from  $\{\text{brick}, \text{floor}, \text{cheese}\}$ . Walls consist of lines of bricks. A mouse will be able to move along floor squares in any of the eight directions. The maze generated will not contain corridors as such, but only many small rectangular rooms. Each room will either have one door in one corner of the room or two doors in opposite corners. The *cheese* will be placed in a room that is chosen randomly from among the rooms that are “far” from the start location.

**Precondition:** The routine *AddWalls* is passed a matrix representing the maze as constructed so far and the coordinates of a room within it. The room will have a surrounding wall except for one door in one of its corners. It will be empty of walls. The routine is also passed a flag indicating whether or not cheese should be added somewhere in the room.

**Postcondition:** The output is the same maze with a randomly chosen sub-maze added within the indicated room and cheese added as appropriate.

**Initial Conditions:** To meet the preconditions of *AddWalls*, the main routine first constructs the four outer walls with the top right corner square left as a floor tile to act as a door into the maze and as the start square for the mouse. Calling *AddWalls* on this single room completes the maze.

**SubInstances:** If the indicated room has height and width of at least 3, then the routine AddWalls will choose a single location  $(i, j)$  uniformly at random from all those in the room that are not right next to one of its outer walls. (The  $(i, j)$  chosen by the top stack frame in the above example maze is indicated.) A wall is added within the room all the way across row  $i$  and all the way down column  $j$  subdividing the room into 4 smaller rooms. To act as a door connecting these four rooms, the square at location  $(i, j)$  remains a floor tile. Then four friends are asked to fill in a maze into each of these four smaller rooms. If our room is to have cheese then one of the three rooms not containing the door to our room is selected to contain the cheese.

```

*
*****
*   Friend1's   * Friend2's *
*   Room       * Room   *
***** *           *****
*           *(i,j)   *
*   Friend3's   * Friend4's *
*   Room       * Room   *
***** *           *****

```

**Running Time:** The time required to construct an  $n \times n$  maze is  $\Theta(n^2)$ . This can be seen two ways. For the easy way, note that a brick is added at most once to any entry of the matrix and that there are  $\Theta(n^2)$  entries. The hard way solves the recurrence relation  $T(n) = 4T(n/2) + \Theta(n) = \Theta(n^2)$ .

**Searching The Maze:** One way of representing a maze is by a graph. Section 20 presents a number of iterative algorithms for searching a graph. Section 20.5 presents the recursive version of the depth-first search algorithm. All of these could be used by a mouse to find the cheese.

## Chapter 18

# Parsing with Context-Free Grammars

An important computer science problem is to be able to parse a string according a given context-free grammar. A *context-free grammar* is a means of describing which strings of characters are contained within a particular language. It consists of a set of rules and a start *non-terminal* symbol. Each rule specifies one way of replacing a non-terminal symbol in the current string with a string of terminal and non-terminal symbols. When the resulting string consists only of terminal symbols, we stop. We say that any such resulting string has been *generated* by the grammar.

Context-free grammars are used to understand both the syntax and the semantics of many very useful languages, such as mathematical expressions, JAVA, and English. The *syntax* of a language indicates which strings of tokens are valid sentences in that language. The *semantics* of a language involves the meaning associated with strings. In order for a compiler or natural language “recognizers” to determine what a string means, it must *parse* the string. This involves deriving the string from the grammar and, in doing so, determining which parts of the string are “noun phrases”, “verb phrases”, “expressions”, and “terms.”

Usually, the first algorithmic attempts to parse a string from a context-free grammar requires  $2^{\Theta(n)}$  time. However, there is an elegant dynamic-programming algorithm given in Section 24.10 that parses a string from any context-free grammar in  $\Theta(n^3)$  time. Although this is impressive, it is much too slow to be practical for compilers and natural language recognizers. Some context-free grammars have a property called *look ahead one*. Strings from such grammars can be parsed in linear time by what I consider to be one of the most amazing and magical recursive algorithms. This algorithm is presented in this chapter. It demonstrates very clearly the importance of working within the friends level of abstraction instead of tracing out the stack frames: Carefully write the specifications for each program, believe by magic that the programs work, write the programs calling themselves as if they already work, and make sure that as you recurse the instance being inputted gets “smaller.”

**The Grammar:** As an example, we will look at a very simple grammar that considers expressions over  $\times$  and  $+$ .

```
exp ⇒ term
      ⇒ term + term + ... + term
          i.e., an expression is one or more terms added together.
term⇒ fact
      ⇒ fact * fact * ... * fact
          i.e., a term is one or more factors multiplied together.
```

fact  $\Rightarrow$  int  
 $\Rightarrow$  ( exp )  
i.e., a factor is either a simple integer or a more complex expression within brackets.

## A Derivation of a String:

```

s = 6 * 8 + ( ( 2 + 42 ) * ( 5 + 12 ) + 987 * 7 * 123 + 15 * 54 )
      |-----exp-----|
      |-----t-----| + |-----term-----|
      f * f      |-----fact-----|
      6     8   ( |-----exp-----| )
                  |-----term-----| + |-----term-----| + |-----term-----|
                  |-----fact---| * |-----fact---|      f * f *      f      f * f
                  ( |-----ex-| )   ( |-----ex-| )   987    7    123    15    54 )
                  t + t          t + t
                  f    f          f + f
                  2    42          5    12
s = 6 * 8 + ( ( 2 + 42 ) * ( 5 + 12 ) + 987 * 7 * 123 + 15 * 54 )

```

**A Parsing of an Expression:** The following are different forms that the parsing of an expression could take:

- A binary-tree data structure with each internal node representing either '\*' or '+' and leaves representing integers.
  - A text-based picture of the tree described above.

```

s = ( ( 2 + 42 ) * ( 5 + 12 ) + 987 * 7 * 123 + 15 * 54 ) =
p =
          |-- 2
          |+ -
          | |-- 42
          |- *
          | |-- 5
          | |+ -
          | | |-- 12
          -- +
          | |-- 987
          | |- *
          | | |-- 7
          | | |- *
          | | | |-- 123
          |- +
          | | |-- 15
          |- *
          | |-- 54

```

- A string with more brackets indicating the internal structure.

```
s = ((2+42) * (5+12)) + (987*(7*123)) + (15*54)
p = (((2+42) * (5+12)) + ((987*(7*123)) + (15*54)))
```

- An integer evaluation of the expression.

```
s = ( ( 2 + 42 ) * ( 5 - 12 ) + 987 * 7 * 123 + 15 * 54 )
p = 851365
```

**The Parsing Abstract Data Type:** The following is an example of where it is useful not to give the full implementation details of an abstract data type. In fact, we will even leave the specification of parsing structure open for the implementer to decide.

For our purposes, we will only say the following: When  $p$  is a variable of type parsing, we will use “ $p=5$ ” to indicate that it is assigned a parsing of the expression “5”. We will go on to *overload* the operations  $*$  and  $+$  as operations that join two parsings into one. For example, if  $p_1$  is a parsing of the expression “ $2*3$ ” and  $p_2$  of “ $5*7$ ”, then we will use  $p = p_1 + p_2$  to denote a parsing of expression “ $2*3 + 5*7$ ”.

The implementer defines the structure of a parsing by specifying in more detail what these operations do. For example, if the implementer wants a parsing to be a binary tree representing the expression, then  $p_1 + p_2$  would be the operation of constructing a binary tree with the root being a new ‘+’ node, the left subtree being the binary tree  $p_1$ , and the right subtree being the binary tree  $p_2$ . On the other hand, if the implementer wants a parsing to be simply an integer evaluation of the expression, then  $p_1 + p_2$  would be the integer sum of the integers  $p_1$  and  $p_2$ .

**The Specifications:** The parsing algorithm has the following specs:

**Precondition:** The input consists of a string of tokens  $s$ . The possible tokens are the characters ‘\*’ and ‘+’ and arbitrary integers. The tokens are indexed as  $s[1], s[2], s[3], \dots, s[n]$ .

**Postcondition:** If the input is a valid “expression” generated by the grammar, then the output is a “parsing” of the expression. Otherwise, an error message is given.

The algorithm consists of one routine for each *non-terminal* of the grammar: *GetExp*, *GetTerm*, and *GetFact*. The specs for *GetExp* are the following:

**Precondition:** The input of *GetExp* consists of a string of tokens  $s$  and an index  $i$  that indicates a starting point within  $s$ .

**Output:** The output consists of a parsing of the longest substring  $s[i], s[i + 1], \dots, s[j - 1]$  of  $s$  that starts at index  $i$  and is a valid expression. The output also includes the index  $j$  of the token that comes immediately after the parsed expression.

If there is no valid expression starting at  $s[i]$ , then an error message is given.

The specs for *GetTerm* and *GetFact* are the same, except that they return the parsing of the longest term or factor starting at  $s[i]$  and ending at  $s[j - 1]$ .

### Examples:

**GetExp:**

```

s = ( ( 2 * 8 + 42 * 7 ) * 5 + 8 )
      i                               j   p = ( ( 2 * 8 + 42 * 7 ) * 5 + 8 )
          i                           j   p =   ( 2 * 8 + 42 * 7 ) * 5 + 8
              i           j           p =     2 * 8 + 42 * 7
                  i       j           p =         42 * 7
                      i       j       p =             5 + 8

```

**GetTerm:**

```

s = ( ( 2 * 8 + 42 * 7 ) * 5 + 8 )
      i                               j   p = ( ( 2 * 8 + 42 * 7 ) * 5 + 8 )
          i                           j   p =   ( 2 * 8 + 42 * 7 ) * 5
              i           j           p =     2 * 8
                  i       j       p =         42 * 7

```

i	j	p =	5
<b>GetFact:</b>			
s = ( ( 2 * 8 + 42 * 7 ) * 5 + 8 )			
i		j	p = ( ( 2 * 8 + 42 * 7 ) * 5 + 8 )
i			p = ( 2 * 8 + 42 * 7 )
i j			p = 2
i j			p = 42
i j			p = 5

**Intuitive Reasoning for *GetExp* and for *GetTerm*:** Consider some input string  $s$  and some index  $i$ . The longest substring  $s[i], \dots, s[j - 1]$  that is a valid expression consists of some number of terms added together. In all of these case, it begins with a term. By magic, assume that the *GetTerm* routine already works. Calling  $\text{GetTerm}(s, i)$  will return  $p_{term}$  and  $j_{term}$ , where  $p_{term}$  is the parsing of this first term and  $j_{term}$  indexes the token immediately after this term. Specifically, if the expression has another term then  $j_{term}$  indexes the '+' that is between these terms. Hence, we can determine whether there is another term by checking  $s[j_{term}]$ . If  $s[j_{term}] = '+'$ , then *GetExp* will call *GetTerm* again to get the next term. If  $s[j_{term}]$  is not a '+' but some other character, then *GetExp* is finished reading in all the terms. *GetExp* then constructs the parsing consisting of all of these terms added together.

The intuitive reasoning for *GetTerm* is just the same.

#### *GetExp* Code:

**algorithm** *GetExp* ( $s, i$ )

***⟨pre-cond⟩:***  $s$  is a string of tokens and  $i$  is an index that indicates a starting point within  $s$ .

***⟨post-cond⟩:*** The output consists of a parsing  $p$  of the longest substring  $s[i], s[i + 1], \dots, s[j - 1]$  of  $s$  that starts at index  $i$  and is a valid expression. The output also includes the index  $j$  of the token that comes immediately after the parsed expression.

```

begin
    if ( $i > |s|$ ) return "Error: Expected characters past end of string." end if
     $\langle p_{\langle term,1 \rangle}, j_{\langle term,1 \rangle} \rangle = \text{GetTerm}(s, i)$ 
     $k = 1$ 
    loop
        loop-invariant: The first  $k$  terms of the expression have been read.
        exit when  $s[j_{\langle term,k \rangle}] \neq '+'$ 
         $\langle p_{\langle term,k+1 \rangle}, j_{\langle term,k+1 \rangle} \rangle = \text{GetTerm}(s, j_{\langle term,k \rangle} + 1)$ 
         $k = k + 1$ 
    end loop
     $p_{exp} = p_{\langle term,1 \rangle} + p_{\langle term,2 \rangle} + \dots + p_{\langle term,k \rangle}$ 
     $j_{exp} = j_{\langle term,k \rangle}$ 
    return  $\langle p_{exp}, j_{exp} \rangle$ 
end algorithm

```

#### *GetTerm* Code:

```

algorithm GetTerm (s, i)
  <pre-cond>: s is a string of tokens and i is an index that indicates a starting point within s.
  <post-cond>: The output consists of a parsing p of the longest substring s[i], s[i + 1], ..., s[j - 1] of s that starts at index i and is a valid term. The output also includes the index j of the token that comes immediately after the parsed term.

begin
  if (i > |s|) return “Error: Expected characters past end of string.” end if
   $\langle p_{\langle \text{fact},1 \rangle}, j_{\langle \text{fact},1 \rangle} \rangle = \text{GetFact}(s, i)$ 
  k = 1
  loop
    <loop-invariant>: The first k facts of the term have been read.
    exit when s[jfact,k] ≠ ‘*’
     $\langle p_{\langle \text{fact},k+1 \rangle}, j_{\langle \text{fact},k+1 \rangle} \rangle = \text{GetFact}(s, j_{\langle \text{fact},k \rangle} + 1)$ 
    k = k + 1
  end loop
  pterm = pfact,1 * pfact,2 * ... * pfact,k
  jterm = jfact,k
  return  $\langle p_{\text{term}}, j_{\text{term}} \rangle$ 
end algorithm

```

**Intuitive Reasoning for *GetFact*:** The longest substring *s*[*i*], ..., *s*[*j* - 1] that is a valid factor has one of the following two forms:

```

fact  $\Rightarrow$  int
fact  $\Rightarrow$  ( exp )

```

Hence, we can determine which form the factor has by testing *s*[*i*].

If *s*[*i*] is an integer, then we are finished. *p<sub>fact</sub>* is a parsing of this single integer *s*[*i*] and *j<sub>fact</sub>* = *i* + 1. Note that the +1 moves the index past the integer.

If *s*[*i*] = ‘(’, then to be a valid factor there must be a valid expression starting at *j<sub>term</sub>* + 1, followed by a closing bracket ‘)’. We can parse this expression with *GetExp*(*s*, *j<sub>term</sub>* + 1), which returns *p<sub>exp</sub>* and *j<sub>exp</sub>*. The closing bracket after the expression must be in *s*[*j<sub>exp</sub>*]. Our parsed factor will be *p<sub>fact</sub>* = (*p<sub>exp</sub>*) and *j<sub>fact</sub>* = *j<sub>exp</sub>* + 1. Note that the +1 moves the index past the ‘)’.

If *s*[*i*] is neither an integer nor a ‘(’, then it cannot be a valid factor. Give a meaningful error message.

#### ***GetFact* Code:**

```

algorithm GetFac (s, i)
  <pre-cond>: s is a string of tokens and i is an index that indicates a starting point within s.
  <post-cond>: The output consists of a parsing p of the longest substring s[i], s[i + 1], ..., s[j - 1] of s that starts at index i and is a valid factor. The output also includes the index j of the token that comes immediately after the parsed factor.

```

```

begin
    if ( $i > |s|$ ) return “Error: Expected characters past end of string.” end if
    if ( $s[i]$  is an int)
         $p_{fact} = s[i]$ 
         $j_{fact} = i + 1$ 
        return  $\langle p_{fact}, j_{fact} \rangle$ 
    else if ( $s[i] = '('$ )
         $\langle p_{exp}, j_{exp} \rangle = GetExp(s, i + 1)$ 
        if ( $s[j_{exp}] = ')'$ )
             $p_{fact} = (p_{exp})$ 
             $j_{fact} = j_{exp} + 1$ 
            return  $\langle p_{fact}, j_{fact} \rangle$ 
        else
            Output “Error: Expected ')' at index  $j_{exp}$ ”
        end if

    else
        Output “Error: Expected integer or '(' at index  $i$ ”
    end if
end algorithm

```

**Exercise 18.0.1** (See solution in Section V) Consider  $s = “( ( 1 ) * 2 + 3 ) * 5 * 6 + 7 ”$ .

1. Give a derivation of the expression  $s$  as done above.
2. Draw the tree structure of the expression  $s$ .
3. Trace out the execution of your program on  $GetExp(s, 1)$ . In other words, draw a tree with a box for each time a routine is called. For each box, include only whether it is an expression, term, or factor and the string  $s[i], \dots, s[j - 1]$  that is parsed.

**Proof of Correctness:** To prove that a recursive program works, we must consider the “size” of an instance. The routine needs only consider the postfix  $s[i], s[i + 1], \dots$ , which contains  $(|s| - i + 1)$  characters. Hence, we will define the size of instance  $\langle s, i \rangle$  to be  $|\langle s, i \rangle| = |s| - i + 1$ .

Let  $H(n)$  be the statement “Each of  $GetFac$ ,  $GetTerm$ , and  $GetExp$  work on instances  $\langle s, i \rangle$  when  $|\langle s, i \rangle| = |s| - i + 1 \leq n$ .” We prove by way of induction that  $\forall n \geq 0$ ,  $H(n)$ .

If  $|\langle s, i \rangle| = 0$ , then  $i > |s|$ : There is not a valid expression/term/factor starting at  $s[i]$ , and all three routines return an error message. It follows that  $H(0)$  is true.

If  $|\langle s, i \rangle| = 1$ , then there is one remaining token: For this to be a factor, term, or expression, this token must be a single integer.  $GetFac$  is written to give the correct answer in this situation.  $GetTerm$  gives the correct answer, because it calls  $GetFac$ .  $GetExp$  gives the correct answer, because it calls  $GetTerm$  which in turn calls  $GetFac$ . It follows that  $H(1)$  is true.

Assume  $H(n - 1)$  is true, i.e., that “Each of  $GetFac$ ,  $GetTerm$ , and  $GetExp$  work on instances of size at most  $n - 1$ .”

Consider  $GetFac(s, i)$  on an instance of size  $|s| - i + 1 = n$ . It makes at most one subroutine call,  $GetExp(s, i+1)$ . The size of this instance is  $|s| - (i+1) + 1 = n - 1$ . Hence, by assumption

this subroutine call returns the correct answer. Because all of  $GetFac(s, i)$ 's subroutine calls return the correct answer, the above intuition proves that  $GetFac(s, i)$  works on all instances of size  $n$ .

Now consider  $GetTerm(s, i)$  on an instance of size  $|s| - i + 1 = n$ . It calls  $GetFac$  some number of times. The input instance for the first call  $GetFac(s, i)$  still has size  $n$ . Hence, the induction hypothesis  $H(n - 1)$  does NOT claim that it works. However, the previous paragraph proves that this routine does in fact work on instances of size  $n$ . The remaining calls are on smaller instances.

Finally, consider  $GetExp(s, i)$  on an instance  $\langle s, i \rangle$  of size  $|s| - i + 1 = n$ . We use the previous paragraph to prove that its first subroutine call  $GetTerm(s, i)$  works.

In conclusion, all three work on all instances of size  $n$  and hence on  $H(n)$ . This completes the induction step.

**Look Ahead One:** A grammar is said to be *look ahead one* if, given any two rules for the same non-terminal, the first place that the rules differ is a difference in a terminal. This feature allows the above parsing algorithm to look only at the next token in order to decide what to do next.

An example of a good set of rules would be:

$$\begin{aligned} A &\Rightarrow B 'b' C 'd' E \\ A &\Rightarrow B 'b' C 'e' F \\ A &\Rightarrow B 'c' G H \end{aligned}$$

An example of a bad set of rules would be:

$$\begin{aligned} A &\Rightarrow B C \\ A &\Rightarrow D E \end{aligned}$$

With such a grammar, you would not know whether to start parsing the string as a B or a D. If you made the wrong choice, you would have to back up and repeat the process.

# **Part IV**

# **Optimization Problems**

## Chapter 19

# Definition of Optimization Problems

Many important and practical problems can be expressed as an *optimization problem*. Such problems involve finding the best of an exponentially large set of solutions. It can be like finding a needle in a haystack. The obvious algorithm, considering each of the solutions, takes too much time because there are so many solutions. Some of these problems can be solved in polynomial time using network flow, linear programming, greedy algorithms, or dynamic programming. When not, recursive backtracking can sometimes find an optimal solution for some instances for some practical applications. Random algorithms provide approximately optimal solutions for other problems. However, for the most of optimization problems, the best known algorithm require  $2^{\Theta(n)}$  time on the worst case input instances. The commonly held belief is that there are not polynomial time algorithms for them (though we may be wrong). NP-completeness helps to justify this belief by showing that some of these problems are universally hard amongst this class problems. We now formally define this class of problems.

**Ingredients:** An optimization problem is specified by defining instances, solutions, and costs.

**Instances:** The *instances* are the possible inputs to the problem.

**Solutions for Instance:** Each instance has an exponentially large set of *solutions*. A solution is *valid* if it meets a set of criteria determined by the instance at hand.

**Cost of Solution:** Each solution has an easy to compute *cost* or *measure of success*.

### Specification of an Optimization Problem:

**Preconditions:** The input is one instance.

**Postconditions:** The output is one of the valid solutions for this instance with optimal (minimum or maximum as the case may be) cost. (The solution to be outputted might not be unique.)

### Examples:

**Longest Common Subsequence:** This is an example for which we have polynomial time algorithm.

**Instances:** An instance consists of two sequences, e.g.,  $X = \langle A, B, C, B, D, A, B \rangle$  and  $Y = \langle B, D, C, A, B, A \rangle$ .

**Solutions:** A subsequence of a sequence is a subset of the elements taken in the same order. For example,  $Z = \langle B, C, A \rangle$  is a subsequence of  $X = \langle A, \underline{B}, \underline{C}, B, D, \underline{A}, B \rangle$ . A solution is a sequence,  $Z$ , that is a subsequence of both  $X$  and  $Y$ . For example,  $Z = \langle B, C, A \rangle$  is solution because it is a subsequence common to both  $X$  and  $Y$  ( $Y = \langle \underline{B}, D, \underline{C}, \underline{A}, B, A \rangle$ ).

**Cost of Solution:** The cost (or success) of a solution is the length of the common subsequence, e.g.,  $|Z| = 3$ .

**Goal:** Given two sequences  $X$  and  $Y$ , the goal is to find the longest common subsequence (LCS for short). For the example given above,  $Z = \langle B, C, B, A \rangle$  is a longest common subsequence.

**Course Scheduling:** This is an example for which we do not have polynomial time algorithm.

**Instances:** An instance consists of the set of courses specified by a university, the set of courses that each student requests, and the set of time slots in which courses can be offered.

**Solutions:** A solution for an instance is a schedule which assigns each course a time slot.

**Cost of Solution:** A conflict occurs when two courses are scheduled at the same time even though a student requests them both. The cost of a schedule is the number of conflicts that it has.

**Goal:** Given the course and student information, the goal is to find the schedule with the fewest conflicts.

# Chapter 20

## Graph Search Algorithms

An optimization problem requires finding the best of a large number of solutions. This can be compared to a mouse finding cheese in a maze. Graph search algorithms provide a way of systematically searching through this maze of possible solutions.

Another example of an optimization problem is finding the shortest path between two nodes in a graph. There may be an exponential number of paths between these two nodes. It would take too much time to consider each such path. The algorithms used to find a shortest one demonstrate many of the principles that will arise when solving harder optimization problems.

A surprisingly large number of problems in computer science can be expressed as a graph theory problem. In this chapter, we will first learn a generic search algorithm in which the algorithm finds more and more of the graph by following arbitrary edges from nodes that have already been found. We also consider the more specific orders of depth first and breadth-first search to traverse the graph. Using these ideas, we are able to discover shortest paths between pairs of nodes and learn information about the structure of the graph.



### 20.1 A Generic Search Algorithm

**Specifications of the Problem:** *Reachability-from-single-source  $s$*

**Preconditions:** The input is a graph  $G$  (either directed or undirected) and a source node  $s$ .

**Postconditions:** The output consists of all the nodes  $u$  that are reachable by a path in  $G$  from  $s$ .

**Basic Steps:** Suppose you know that node  $u$  is reachable from  $s$  (denoted as  $s \rightarrow u$ ) and that there is an edge from  $u$  to  $v$ . Then you can conclude that  $v$  is reachable from  $s$  (i.e.,  $s \rightarrow u \rightarrow v$ ). You can use such steps to build up a set of reachable nodes.

- $s$  has an edge to  $v_4$  and  $v_9$ . Hence,  $v_4$  and  $v_9$  are reachable.
- $v_4$  has an edge to  $v_7$  and  $v_3$ . Hence,  $v_7$  and  $v_3$  are reachable.
- $v_7$  has an edge to  $v_2$  and  $v_8$ . ...

### Difficulties:

- How do you keep track of all this?
- How do you know that you have found all the nodes?
- How do you avoid cycling, as in  $s \rightarrow v_4 \rightarrow v_7 \rightarrow v_2 \rightarrow v_4 \rightarrow v_7 \rightarrow v_2 \rightarrow v_4 \rightarrow v_7 \rightarrow v_2 \rightarrow v_4 \dots$  forever?

### Ingredients of the Loop Invariant:

**Found:** If you trace a path from  $s$  to a node, then we will say that the node has been *found*.

**Handled:** At some point in time after node  $u$  has been found, you will want to follow all the edges from  $u$  and find all the nodes  $v$  that have edges from  $u$ . When you have done that for node  $u$ , we say that it has been *handled*.

**Data Structure:** You must maintain (1) the set of nodes *foundHandled* that have been found and handled and (2) the set of nodes *foundNotHandled* that have been found but *not* handled.

### The Loop Invariant:

**LI1:** For each found node  $v$ , we know that  $v$  is reachable from  $s$  because we have traced out a path  $s \rightarrow v$  from  $s$  to it.

**LI2:** If a node has been handled, then all of its neighbors have been found.

These loop invariants are simple enough that establishing and maintaining them should be easy. But do they suffice to prove the postcondition? We will see.

**Body of the Loop:** A reasonable step would be:

- Choose some node  $u$  from *foundNotHandled* and handle it. This involves following all the edges from  $u$ .
- Newly-found nodes are now added to the set *foundNotHandled* (if they have not been found already).
- $u$  is moved from *foundNotHandled* to *foundHandled*.

### Code:

```
algorithm Search( $G, s$ )
```

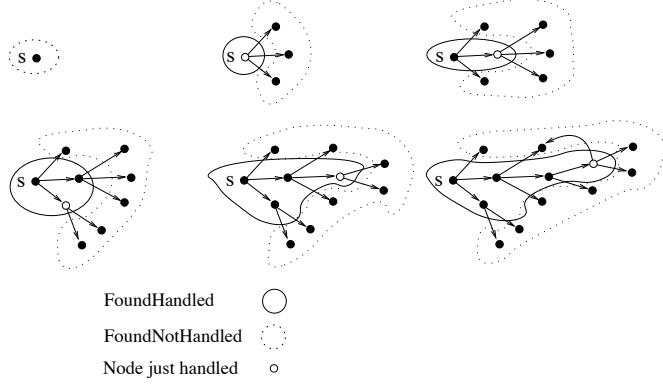


Figure 20.1: The generic search algorithm *handles* one found node at a time by *finding* their neighbors.

***(pre-cond)*:**  $G$  is a (directed or undirected) graph and  $s$  is one of its nodes.

***(post-cond)*:** The output consists of all the nodes  $u$  that are reachable by a path in  $G$  from  $s$ .

```

begin
    foundHandled = ∅
    foundNotHandled = {s}
    loop
        (loop-invariant): See above.
        exit when foundNotHandled = ∅
        let  $u$  be some node from foundNotHandled
        for each  $v$  connected to  $u$ 
            if  $v$  has not previously been found then
                add  $v$  to foundNotHandled
            end if
        end for
        move  $u$  from foundNotHandled to foundHandled
    end loop
    return foundHandled
end algorithm

```

**Maintaining the Loop Invariant (i.e.,  $\langle LI' \rangle$  & not  $\langle \text{exit} \rangle$  &  $\text{code}_{\text{loop}} \rightarrow \langle LI'' \rangle$ ):**

Suppose that  $LI'$  which denotes the statement of the loop invariant before the iteration is true, the exit condition  $\langle \text{exit} \rangle$  is not, and we have executed another iteration of the algorithm.

**Maintaining LI1:** After the iteration, the node  $v$  is considered found. Hence, in order to maintain the loop invariant, we must be sure that  $v$  is reachable from  $s$ . Because  $u$  was in  $foundNotHandled$ , the loop invariant assures us that we have traced out a path  $s \rightarrow u$  to it. Now that we have traced the edge  $u \rightarrow v$ , we have traced a path  $s \rightarrow u \rightarrow v$  to  $v$ .

**Maintaining LI2:** Node  $u$  is designated handled only after ensuring that all its neighbors have been found.

**The Measure of Progress:** The measure of progress requires the following three properties:

**Progress:** We must guarantee that our measure of progress increases by at least one every time around the loop. Otherwise, we may loop forever, making no progress.

**Bounded:** There must be an upper bound on the progress required before the loop exits. Otherwise, we may loop forever, increasing the measure of progress to infinity.

**Conclusion:** When sufficient progress has been made to exit, we must be able to conclude that the problem is solved.

An obvious measure would be the number of found nodes. The problem is that when handling a node, you may only find nodes that have already been found. In such a case, no progress is actually made.

A better measure of progress is the number of nodes that have been handled. We can make progress simply by handling a node that has not yet been handled. We also know that if the graph  $G$  has only  $n$  nodes, then this measure cannot increase past  $n$ .

**Exit Condition:** Given our measure of progress, when are we finished? We can only handle nodes that have been found and not handled. Hence, when all the nodes that have been found have also been handled, we can make no more progress. At this point, we must stop.

**Initial Code (i.e.,  $\langle pre-cond \rangle \& code_{pre-loop} \Rightarrow \langle loop-invariant \rangle$ ):** Initially, we know only that  $s$  is reachable from  $s$ . Hence, let's start by saying that  $s$  is found but not handled and that all other nodes have not yet been found.

**Exiting Loop (i.e.,  $\langle LI \rangle \& \langle exit \rangle \rightarrow \langle post \rangle$ ):** Our output will be the set of found nodes. The postcondition requires the following two claims to be true.

**Claim:** Found nodes are reachable from  $s$ .

This is clearly stated in the loop invariant.

**Claim:** Every reachable node has been found. A logically equivalent statement is that every node that has not been found is not reachable.

**One Proof:**

- Draw a circle around the nodes of the graph  $G$  that have been found.
- If there are no edges going from the inside of the circle to the outside of the circle, then there are no paths from  $s$  to the nodes outside of the circle. Hence, we can claim we have found all the nodes reachable from  $s$ .
- How do we know that this circle has no edges leaving it?
  - Consider a node  $u$  in the circle. Because  $u$  has been found and  $foundNotHandled = \emptyset$ , we know that  $u$  has also been handled.
  - By the loop invariant LI2, if  $\langle u, v \rangle$  is an edge, then  $v$  has been found and thus is in the circle as well.
  - Hence, if  $u$  is in the circle and  $\langle u, v \rangle$  is an edge, then  $v$  is in the circle as well (i.e., no edges leave the circle).
  - This is known as a *closure property*. See Section 23.4.3 for more information on this property.

**Another Proof:** Proof by contradiction.

- Suppose that  $w$  is reachable from  $s$  and that  $w$  has not been found.
- Consider a path from  $s$  to  $w$ .
- Because  $s$  has been found and  $w$  has not, the path starts in the set of found nodes and at some point leaves it.
- Let  $\langle u, v \rangle$  be the first edge in the path for which  $u$  but not  $v$  has been found.
- Because  $u$  has been found and  $foundNotHandled = \emptyset$ , it follows that  $u$  has been handled.
- Because  $u$  has been handled,  $v$  must be found.
- This contradicts the definition of  $v$ .

### Running Time:

**A Simple but False Argument:** For every iteration of the loop, one node is handled and no node is handled more than once. Hence, the measure of progress (the number of nodes handled) increases by one with every loop.  $G$  only has  $|V| = n$  nodes. Hence, the algorithm loops at most  $n$  times. Thus, the running time is  $\mathcal{O}(n)$ .

This argument is false, because while handling  $u$  we must consider  $v$  for every edge coming out of  $u$ .

**Overestimation:** Each node has at most  $n$  edges coming out of it. Hence, the running time is  $\mathcal{O}(n^2)$ .

**Correct Complexity:** Each edge of  $G$  is looked at exactly twice, once from each direction.

The algorithm's time is dominated by this fact. Hence, the running time is  $\mathcal{O}(|E|)$ , where  $E$  is the set of edges in  $G$ .

**The Order of Handling Nodes:** This algorithm specifically did not indicate which node  $u$  to select from  $foundNotHandled$ . It did not need to, because the algorithm works no matter how this choice is made. We will now consider specific orders in which handle the nodes and specific applications of these orders.

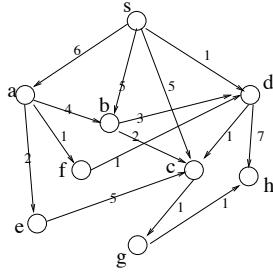
**Queue/Breadth-First Search:** One option is to handle nodes in the order they are found.

This treats  $foundNotHandled$  as a queue: “first in, first out.” The effect is that the search is *breadth first*, meaning that all nodes at distance 1 from  $s$  are handled first, then all those at distance two, and so on. A byproduct of this is that we find for each node  $v$  a shortest path from  $s$  to  $v$ . See Section 20.2.

**Priority Queue/Shortest (Weighted) Paths:** Another option calculates for each node  $v$  in  $foundNotHandled$  the minimum weighted distance from  $s$  to  $v$  along any path seen so far. It then handles the node that is closest to  $s$  according to this approximation. Because these approximations change throughout time,  $foundNotHandled$  is implemented using a priority queue: “highest current priority out first.” Like breadth-first search, the search handles nodes that are closest to  $s$  first, but now the length of a path is the sum of its edge weights. A by-product of this method is that we find for each node  $v$  the shortest weighted path from  $s$  to  $v$ . See Section 20.3.

**Stack/Depth-First Search:** Another option is to handle the node that was found most recently. This method treats  $foundNotHandled$  as a stack: “last in, first out.” The effect is that the search is *depth-first*, meaning that a particular path is followed as deeply as possible into the graph until a dead-end is reached, forcing the algorithm to backtrack. See Section 20.4.

**Exercise 20.1.1** Try searching the following graph in the above three ways.



## 20.2 Breadth-First Search/Shortest Paths

We will now develop an algorithm for the *shortest-paths problem*. The algorithm uses a *breadth-first search*. This algorithm is a less generic version of the algorithm in Section 20.1, because the order in which the nodes are handled is now specified more precisely. The loop invariants are strengthened in order to solve the shortest-paths problem.

**Two Versions of the Problem:**

**The Shortest-Path  $st$  Problem:** Given a graph  $G$  (either directed or undirected) and specific nodes  $s$  and  $t$ , the problem is to find one of the shortest paths from  $s$  to  $t$  in  $G$ .

**Instances:** An instance  $\langle G, s, t \rangle$  consists of a graph  $G$  and specific nodes  $s$  and  $t$ .

**Solutions for Instance:** A solution for instance  $\langle G, s, t \rangle$  is a path  $\pi$  from  $s$  to  $t$ .

**Cost of Solution:** The length (or cost) of a path  $\pi$  is the number of edges in the path.

**Goal:** Given an instance  $\langle G, s, t \rangle$ , the goal is to find an optimal solution, i.e., a shortest path from  $s$  to  $t$  in  $G$ .

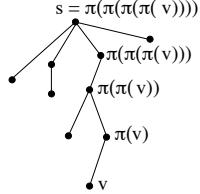
These are the ingredients of an optimization problem. See Section 19 for a formal definition. As often is the case, the set of solutions for an instance may well be exponential. We do not want to check them all.

**The Shortest Path - Single Source  $s$  Multiple Sink  $t$ :**

**Preconditions:** The input is a graph  $G$  (either directed or undirected) and a source node  $s$ .

**Postconditions:** The output consists of a  $d$  and a  $\pi$  for each node of  $G$ . It has the following properties:

1. For each node  $v$ ,  $d(v)$  gives the length  $\delta(s, v)$  of the shortest path from  $s$  to  $v$ .
2. The *shortest paths* or *breadth-first search tree* is defined using  $\pi$  as follows:  $s$  is the root of the tree.  $\pi(v)$  is the parent of  $v$  in the tree. For each node  $v$ , one of the shortest paths from  $s$  to  $v$  is given backward, with  $v, \pi(v), \pi(\pi(v)), \pi(\pi(\pi(v))), \dots, s$ . A recursive definition is that this shortest path from  $s$  to  $v$  is the given shortest path from  $s$  to  $\pi(v)$ , followed by the edge  $\langle \pi(v), v \rangle$ .



We will solve this version of the problem.

**Prove Path Is Shortest:** In order to claim that the shortest path from  $s$  to  $v$  is of some length  $d(v)$ , you must do two things:

**Not Further:** You must produce a suitable path of this length. We call this path a *witness* of the fact that the distance from  $s$  to  $v$  is at most  $d(v)$ . In *finding* a node, we trace out a path from  $s$  to it. If we have already traced out a shortest path from  $s$  to  $u$  with  $d(u)$  edges in it and we trace an edge from  $u$  to  $v$ , then we have traced a path from  $s$  to  $v$  with  $d(v) = d(u) + 1$  edges in it. In this path from  $s$  to  $v$ , the node preceding  $v$  is  $\pi(v) = u$ .

**Not Closer:** You must prove that there are no shorter paths. This is harder. Other than checking an exponential number of paths, how can you prove that there are no shorter paths? We will do it using the following trick: Suppose we can ensure that the order in which we find the nodes is according to the length of the shortest path from  $s$  to them. Then, when we find  $v$ , we know that there isn't a shorter path to it or else we would have found it already.

**Definition  $V_j$ :** Let  $V_j$  denote the set of nodes at distance  $j$  from  $s$ .

### The Loop Invariant:

**LI1:** For each found node  $v$ ,  $d(v)$  and  $\pi(v)$  are as required, i.e., they give the shortest length and a shortest path from  $s$  to the node.

**LI2:** If a node has been handled, then all of its neighbors have been found.

**LI3:** So far, the order in which the nodes have been found is according to the length of the shortest path from  $s$  to it, i.e. the nodes in  $V_j$  before those in  $V_{j+1}$ .

**Order to Handle Nodes:** The only way in which we are changing the general search algorithm is by being more careful in our choice of which node from *foundNotHandled* to handle next. According to LI3, the nodes that were found earlier are closer to  $s$  than those that are found later. The closer a node is to  $s$ , the closer are its neighbors. Hence, in an attempt to find close nodes, the algorithm will next handle the earliest found node. This is accomplished by treating the set *foundNotHandled* as a queue, “first in, first out.”

**Body of the Loop:** Remove the first node  $u$  from the *foundNotHandled* queue and handle it as follows. For every neighbor  $v$  of  $u$  that has not been found,

- add the node to the queue,
- let  $d(v) = d(u) + 1$ ,
- let  $\pi(v) = u$ , and
- consider  $u$  to be handled and  $v$  to be *foundNotHandled*.

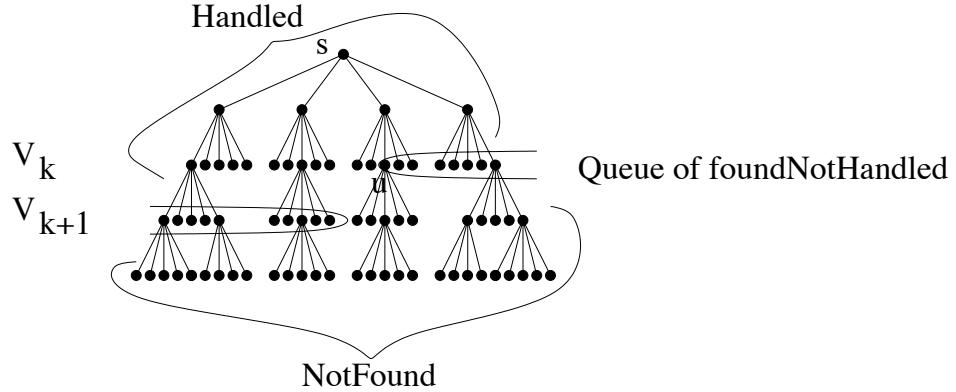


Figure 20.2: Breadth-First Search Tree: We cannot assume that the graph is a tree. Here I have presented only the tree edges given by  $\pi$ . The figure helps to explain the loop invariant, showing which nodes have been found, which found but not handled, and which handled.

**Code:**

```

algorithm ShortestPath ( $G, s$ )
  <pre-cond>:  $G$  is a (directed or undirected) graph and  $s$  is one of its nodes.
  <post-cond>:  $\pi$  specifies a shortest path from  $s$  to each node of  $G$  and  $d$  specifies their
    lengths.

  begin
     $foundHandled = \emptyset$ 
     $foundNotHandled = \{s\}$ 
     $d(s) = 0, \pi(s) = \epsilon$ 
    loop
      <loop-invariant>: See above.
      exit when  $foundNotHandled = \emptyset$ 
      let  $u$  be the node in the front of the queue  $foundNotHandled$ 
      for each  $v$  connected to  $u$ 
        if  $v$  has not previously been found then
          add  $v$  to  $foundNotHandled$ 
           $d(v) = d(u) + 1$ 
           $\pi(v) = u$ 
        end if
      end for
      move  $u$  from  $foundNotHandled$  to  $foundHandled$ 
    end loop
    (for unfound  $v, d(v) = \infty$ )
    return  $\langle d, \pi \rangle$ 
  end algorithm

```

**Maintaining the Loop Invariant (i.e.,  $\langle LI' \rangle$  & not  $\langle exit \rangle$  &  $code_{loop} \rightarrow \langle LI'' \rangle$ ):**

Suppose that  $LI'$  which denotes the statement of the loop invariant before the iteration is true, the exit condition  $\langle exit \rangle$  is not, and we have executed another iteration of the algorithm.

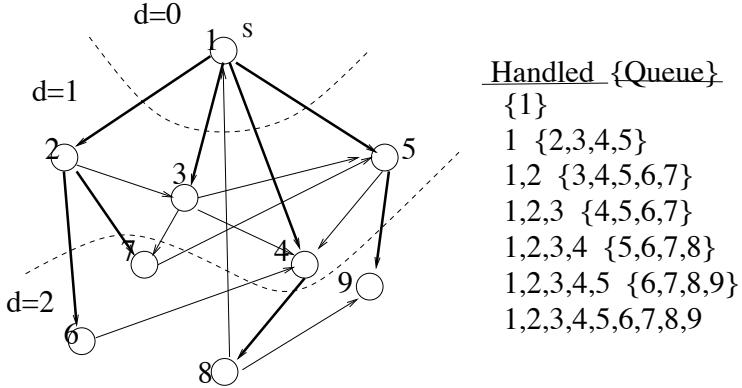


Figure 20.3: Breadth-First Search of a Graph. The numbers show the order in which the nodes were found. The contents of the queue are given at each step. The tree edges are darkened.

**Closer Nodes Have Already Been Found:** We will need the following claim twice.

**Claim:** If the first node in the queue  $\text{foundNotHandled}$ , i.e.,  $u$ , is in  $V_k$ , then

1. all the nodes in  $V_0, V_1, V_2, \dots, V_{k-1}$  have already been found and handled
2. and all the nodes in  $V_k$  have already been found.

**Proof of Claim 1:** Let  $u'$  denote any node in  $V_0, V_1, V_2, \dots, V_{k-1}$ . Because LI3' ensures that nodes have been found in the order of their distance and because  $u'$  is closer to  $s$  than  $u$ ,  $u'$  must have been found earlier than  $u$ . Hence,  $u'$  cannot be in the queue  $\text{foundNotHandled}$  or else it would be earlier in the queue than  $u$ , yet  $u$  is first. This proves that  $u'$  has been handled.

**Proof of Claim 2:** Consider any node  $v$  in  $V_k$  and any path of length  $k$  to it. Let  $u'$  be the previous node in this path. Because the subpath to  $u'$  is of length  $k-1$ ,  $u'$  is in  $V_{k-1}$ , and hence by claim 1 has already been handled. Therefore, by LI2', the neighbors of  $u'$ , of which  $v$  is one, must have been found.

**Maintaining LI1:** During this iteration, all the neighbors  $v$  of node  $u$  that had not been found are now considered found. Hence, their  $d(v)$  and  $\pi(v)$  must now give the shortest length and a shortest path from  $s$ . The code sets  $d(v)$  to  $d(u) + 1$  and  $\pi(v)$  to  $u$ . Hence, we must prove that the neighbors  $v$  are in  $V_{k+1}$ . As said above in the general steps, there are two steps to do this.

**Not Further:** There is a path from  $s$  to  $v$  of length  $k+1$ : follow the path of length  $k$  to  $u$  and then take edge to  $v$ . Hence, the shortest path to  $v$  can be no longer than this.

**Not Closer:** We know that there isn't a shorter path to  $v$  or it would have been found already. More formally, the claim states that all the nodes in  $V_0, V_1, V_2, \dots, V_k$  have already been found. Because  $v$  has not already been found, it cannot be one of these.

**Maintaining LI2:** Node  $u$  is designated handled only after ensuring that all its neighbors have been found.

**Maintaining LI3:** By the claim, all the nodes in  $V_0, V_1, V_2, \dots, V_k$  have already been found and hence have already been added to the queue. Above, we proved that the node  $v$  being found is in  $V_{k+1}$ . It follows that the order in which the nodes are found continues to be according to their distance from  $s$ .

**Initial Code (i.e.,  $\langle \text{pre} \rangle \rightarrow \langle \text{LI} \rangle$ ):** The initial code puts the source  $s$  into  $\text{foundNotHandled}$  and sets  $d(s) = 0$  and  $\pi(s) = \epsilon$ . This is correct, given that initially  $s$  has been found but not handled. The other nodes have not been found and hence their  $d(v)$  and  $\pi(v)$  are irrelevant. The loop invariants follow easily.

**Exiting Loop (i.e.,  $\langle \text{LI} \rangle \& \langle \text{exit} \rangle \rightarrow \langle \text{post} \rangle$ ):** The general-search postconditions prove that all reachable nodes have been found. LI1 states that for these nodes the values of  $d(v)$  and  $\pi(v)$  are as required.

For the nodes that are unreachable from  $s$ , you can set  $d(v) = \infty$  or you can leave them undefined. In some applications (such as the world wide web), you have no access to unreachable nodes. An advantage of this algorithm is that it never needs to know about a node unless it has been found.

**Exercise 20.2.1** (*See solution in Section V*) Suppose  $u$  is being handled,  $u \in V_k$ , and  $v$  is a neighbor of  $u$ . For each of the following cases, explain which  $V_{k'} v$  might be in:

- $\langle u, v \rangle$  is an undirected edge, and  $v$  has been found before.
- $\langle u, v \rangle$  is an undirected edge, and  $v$  has not been found before.
- $\langle u, v \rangle$  is a directed edge, and  $v$  has been found before.
- $\langle u, v \rangle$  is a directed edge, and  $v$  has not been found before.

### 20.3 Dijkstra's Shortest-Weighted Paths Algorithm

We will now make the shortest-paths problem more general by allowing each edge to have a different weight (length). The length of a path from  $s$  to  $v$  will be the sum of the weights on the edges of the path. This makes the problem harder because the shortest path to a node may wind deep into the graph along many short edges instead of along a few long edges. Despite this, only small changes need to be made to the algorithm. This algorithm is called *Dijkstra's algorithm*.

**Specifications of the Problem:** **Name:** The *shortest-weighted-path* problem.

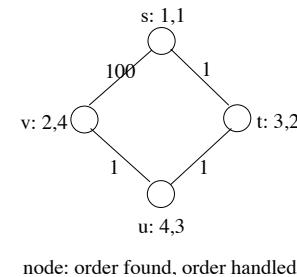
**Preconditions:** The input is a graph  $G$  (either directed or undirected) and a source node  $s$ . Each edge  $\langle u, v \rangle$  is allocated a non-negative weight  $w_{\langle u, v \rangle}$ .

**Postconditions:** The output consists of  $d$  and  $\pi$ , where for each node  $v$  of  $G$ ,  $d(v)$  gives the length  $\delta(s, v)$  of the shortest-weighted path from  $s$  to  $v$  and  $\pi$  defines a *shortest-weighted-paths tree*. (See Section 20.2.)

**Prove Path Is Shortest:** As before, proving that the shortest path from  $s$  to  $v$  is of some length  $d(v)$  involves producing a suitable path of this length and proving that there are no shorter paths.

**Not Further:** As before a witness that there is such a path is produced by tracing it out. The only change is that when we find a path  $s \rightarrow u \rightarrow v$ , we compute its length to be  $d(v) = d(u) + w_{\langle u, v \rangle}$  instead of only  $d(v) = d(u) + 1$ .

**Not Closer:** Unlike in the BFS shortest paths algorithm from Section 20.2, the algorithm does not find the nodes in the order of length of the shortest path to it from  $s$ . In the example on the right, when handling  $s$  we first find  $v$  and then  $t$ , even though  $v$  has the furthest distance, 3, from  $s$ . Instead, the algorithm handles the nodes in this desired order, namely  $s$ ,  $t$ ,  $u$ , and  $v$ . Because of this, when we handle a node, we know that there is no shorter path to it because otherwise we would have handled it already.



**The Next Node To Handle:** The algorithm must choose which of the unhandled nodes to handle next. The previous intuition says to handle the one that is the next closest to  $s$ . The difficulty is that initially we do not know the length of the shortest paths. Instead, the node chosen will be the one that is closest to  $s$  according to our current approximation. In the above example, after handling  $s$  our best approximation of the distance to  $v$  is 100 and to  $t$  is only 1. Hence, handle  $t$  next.

**An Adaptive Greedy Criteria:** This choice amounts to an *Adaptive Greedy Criteria*. See Chapter 22 for more on greedy algorithms.

**Growing a Tree One Node at a Time:** It turns out that the next node to be handled will always be only one edge from a previously handled node. Hence, the tree of handled nodes expands out, one node at a time.

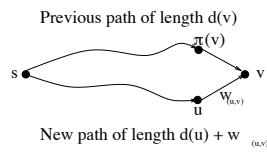
**Approximation of Shortest Distances:** For every node  $v$ , before getting its shortest overall distance, we will maintain  $d(v)$  and  $\pi(v)$  to be the shortest length and path from  $s$  to  $v$  from among those paths that we have *handled* so far.

**Updating:** This information is continuously updated as we find shorter paths to  $v$ . For example, if we find  $v$  when handling  $u$ , then we update these values as follows:

```

 $foundPathLength = d(u) + w_{(u,v)}$ 
if  $d(v) > foundPathLength$  then
     $d(v) = foundPathLength$ 
     $\pi(v) = u$ 
end if

```



When handling  $s$  in our example,  $d(v)$  is set of  $d(s) + w_{(s,v)} = 0 + 100 = 100$ . Later when handling  $u$ , it is updated to  $d(u) + w_{(u,v)} = 2 + 1 = 3$ .

**Definition of A Handled Path:** We say that a path has been handled if it contains only handled edges. Such paths start at  $s$ , visit as many number of handled nodes, and then follow one last edge to a node that may or may not be handled. (See the solid path to  $u$  in Figure 20.4.a.)

**Priority Queue:** The next node to be handled is the one with the smallest  $d(u)$  value. Searching the set of unhandled nodes each iteration for this node would be too time consuming. Resorting the nodes each iteration as the  $d(u)$  values change would also be too time consuming. A more efficient implementation uses a priority queue to hold the unhandled nodes prioritized according to their current  $d(u)$  value. This can be

implemented using a Heap. (See Section 16.4.) We will denote this priority queue by *notHandled*.

**Consider All Nodes “Found”:** No path has yet been handled to any node that has not yet been found, and hence  $d(v) = \infty$ . If we add these nodes to the queue, they will be selected last. Therefore, there is no harm in adding them. Hence, we will distinguish only between those nodes that have been handled and those that have not.

### The Loop Invariant:

**LI1:** For each handled node  $v$ , the values of  $d(v)$  and  $\pi(v)$  give the shortest length and a shortest path from  $s$  (and this path contain only handled nodes).

**LI2:** For each of the unhandled nodes  $v$ , the values of  $d(v)$  and  $\pi(v)$  give the shortest length and path from among those paths that have been *handled*.

**Body of Loop:** Take the next node  $u$  from the priority queue *notHandled* and handle it. This involves handling all edges  $\langle u, v \rangle$  out of  $u$ . Handling edge  $\langle u, v \rangle$  involves updating the  $d(v)$  and  $\pi(v)$  values. The priorities of these nodes are changed in the priority queue as necessary.

### Code:

```

algorithm ShortestWeightedPath ( $G, s$ )
⟨pre-cond⟩:  $G$  is a weighted (directed or undirected) graph and  $s$  is one of its nodes.
⟨post-cond⟩:  $\pi$  specifies a shortest weighted path from  $s$  to each node of  $G$  and  $d$  specifies
their lengths.

begin
     $d(s) = 0, \pi(s) = \epsilon$ 
    for other  $v$ ,  $d(v) = \infty$  and  $\pi(v) = \text{nil}$ 
     $handled = \emptyset$ 
     $notHandled =$  priority queue containing all nodes. Priorities given by  $d(v)$ .
    loop
        ⟨loop-invariant⟩: See above.
        exit when  $notHandled = \emptyset$ 
        let  $u$  be a node from notHandled with smallest  $d(u)$ 
        for each  $v$  connected to  $u$ 
             $foundPathLength = d(u) + w_{\langle u, v \rangle}$ 
            if  $d(v) > foundPathLength$  then
                 $d(v) = foundPathLength$ 
                 $\pi(v) = u$ 
                (update the notHandled priority queue)
            end if
        end for
        move  $u$  from notHandled to handled
    end loop
    return  $\langle d, \pi \rangle$ 
end algorithm

```

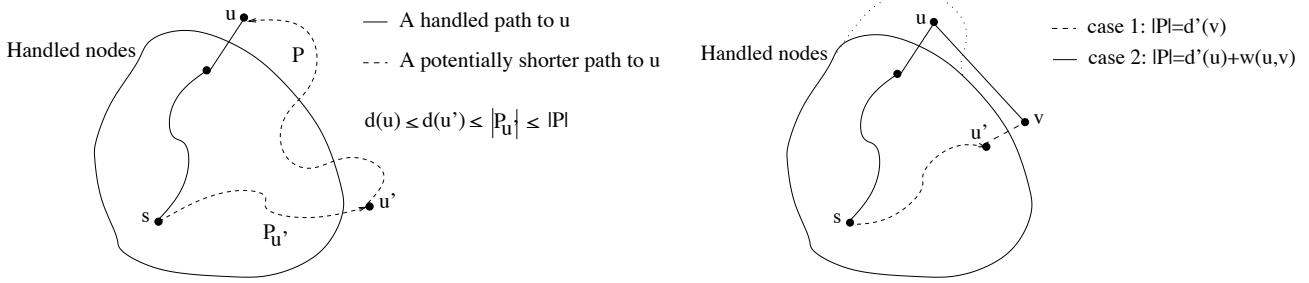


Figure 20.4: The left figure shows a handled path to node  $u$  and is used to maintain LI1. The right figure is used to maintain LI2.

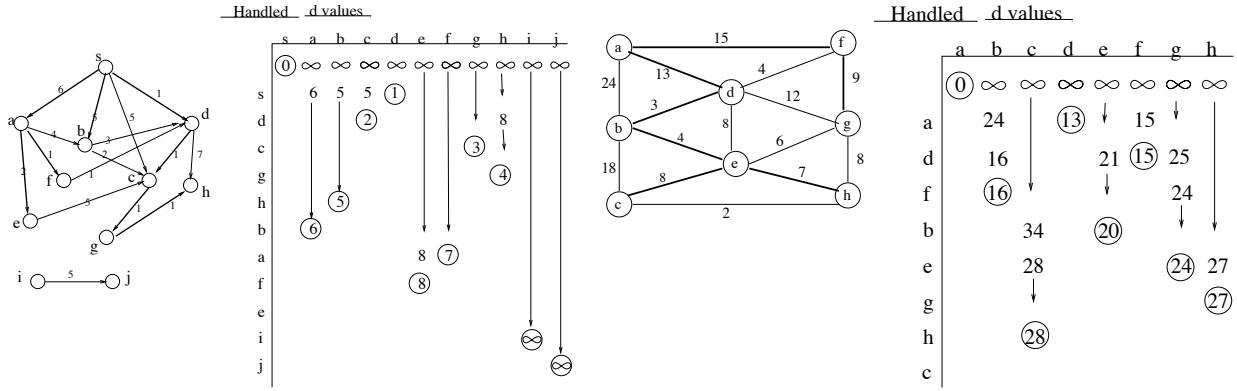


Figure 20.5: Dijkstra's Algorithm. The  $d$  value at each step is given for each node. The tree edges are darkened.

**Maintaining LI1 (i.e.,  $\langle LI1', LI2' \rangle$  & not  $\langle \text{exit} \rangle$  &  $\text{code}_{\text{loop}} \rightarrow \langle LI1'' \rangle$ ):** The loop handles a node  $u$  with smallest  $d(u)$  from  $\text{notHandled}$ . Hence to maintain LI1, we must ensure that its  $d(u)$  and  $\pi(u)$  values give an overall shortest path to  $u$ . Consider some other path  $P$  to  $u$ . We will see that it is no shorter. See Figure 20.4.a. Because the path  $P$  starts at the handled node  $s$  and ends at the previously unhandled node  $u$ , there has to be some node  $u'$  which is the first previously unhandled node along  $P$ . (It is possible that  $u' = u$ .) By the choice of  $u$ ,  $u$  has the smallest  $d(u)$  from  $\text{notHandled}$ . Hence,  $d(u) \leq d(u')$ . Let  $P_{u'}$  be the part the path that goes from  $s$  to  $u'$ . This is a previously handled path. Hence, by LI2',  $d(u') \leq |P_{u'}|$ . Being a subpath and there being no negative weights,  $|P_{u'}| \leq |P|$ . Combining these gives  $d(u) \leq |P|$ . In conclusion,  $d(u)$  is the length of the shortest path to  $u$  and hence LI1 has been maintained.

**Maintaining LI2 (i.e.,  $\langle LI1', LI2' \rangle$  & not  $\langle \text{exit} \rangle$  &  $\text{code}_{\text{loop}} \rightarrow \langle LI2'' \rangle$ ):** Setting  $d''(v)$  to  $\min\{d'(v), d'(u) + w_{\langle u,v \rangle}\}$  ensures that there is a handled path with this length to  $v$ . To maintain LI2, we must prove that there does not exist a shorter one from among those paths that now are considered handled. Note that such paths can now include the newly handled node  $u$ . Let  $P$  be a shortest one. See Figure 20.4.b. Let  $u'$  be the second last node in  $P$ . Because  $P$  is a handled path  $u'$  must be a handled node. There are two cases:

**$u \neq u'$ :** If  $u'$  is a previously handled node, then by the second part of LI1', the shortest path to it does not need to contain the newly handled node  $u$ . It follows that this

path  $P$  to  $v$  is a previously handled path. Hence, its length is at least the length of the shortest previously handled path to  $v$ , which by LI2', is  $d'(v)$ . This in turn is at least  $\min\{d'(v), d'(u) + w_{\langle u,v \rangle}\} = d''(v)$ .

**$u = u'$ :** If the second last node in  $P$  is the newly handled node  $u$ , then its length is the length of the shortest path to  $u$ , which we now know is  $d'(u)$ , plus the weight of the edge  $\langle u, v \rangle$ . It follows that  $|P| \geq \min\{d'(v), d'(u) + w_{\langle u,v \rangle}\} = d''(v)$ .

Either way, shortest path  $P$  to  $v$  that is now considered to be handled has length at least  $d''(v)$ . Hence, LI2 is maintained.

**Initial Code (i.e.,  $\langle \text{pre} \rangle \rightarrow \langle \text{LI} \rangle$ ):** The initial code is the same as that for the BFS shortest paths algorithm from Section 20.2, that is  $s$  is found but not handled with  $d(s) = 0$ ,  $\pi(s) = \epsilon$ . Initially no paths to  $v$  have been *handled* and hence the length of the shortest handled path to  $v$  is  $d(v) = \infty$ . This satisfies all three loop invariants.

**Exiting Loop (i.e.,  $\langle \text{LI} \rangle \& \langle \text{exit} \rangle \rightarrow \langle \text{post} \rangle$ ):** See the shortest path algorithm.

**Running Time:** The general search algorithm takes time  $\mathcal{O}(|E|)$ , where  $E$  are the edges in  $G$ . This algorithm is the same, except for handling the priority queue. A node  $u$  is removed from the priority queue  $|V|$  times and a  $d(v)$  value is updated at most  $|E|$  times, once for each edge. Section 16.4 covered how priority queues can be implemented using a heap so that deletions and updates each take  $\Theta(\log(\text{size of the priority queue}))$  time. Hence, the running time of this algorithm is  $\Theta(|E|\log(|V|))$ .

**Exercise 20.3.1** (*See solution in Section V*) Given a graph where each edge weight is one, compare and contrast the computation of the BFS shortest paths algorithm from Section 20.2 and that of this Dijkstra shortest-weighted paths algorithm. How do their choices of the next node to handle and their loop invariants compare?

## 20.4 Depth-First Search

We have considered breadth-first search that first visits nodes at distance 1 from  $s$ , then those at distance 2, and so on. We will now consider a *depth-first search*, which continues to follow some path as deeply as possible into the graph before it is forced to backtrack.

**Changes to the Generic Search Algorithm:** The next node  $u$  we handle is the one most recently found. *foundNotHandled* will be implemented as a stack. At each iteration, we pop the most recently pushed node and handle it. Try this out on a graph (or on a tree). The pattern in which nodes are found consists of a single path with single edges hanging off it. See Figure 20.6a.

In order to prevent the single edges hanging off the path from being searched, we make a second change to the original searching algorithm: we no longer completely handle one node before we start handling edges from other nodes. From  $s$ , an edge is followed to one of its neighbors  $v_1$ . Before visiting the other neighbors of  $s$ , the current path to  $v_1$  is extended to  $v_2, v_3, \dots$  (See Figure 20.6b.) We keep track of what has been handled by storing an integer  $i_u$  for each node  $u$ . We maintain that for each  $u$ , the first  $i_u$  edges of  $u$  have already been handled.

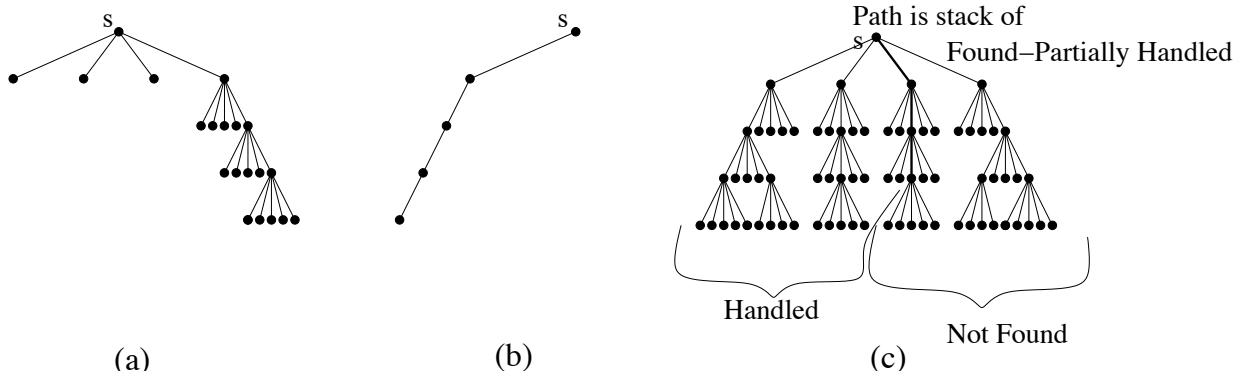


Figure 20.6: If the next node in the stack was completely handled, then the initial order in which nodes are found is given in (a). If the next node is only partially handled, then this initial order is given in (b). (c) presents more of the order in which the nodes are found. Though the input graph may not be a tree. The figure only shows the tree edges given by  $\pi$ .

`foundNotHandled` will be implemented as a stack of tuples  $\langle v, i_v \rangle$ . At each iteration, we pop the most recently pushed tuple  $\langle u, i_u \rangle$  and handle the  $(i_u + 1)^{st}$  edge from  $u$ . Try this out on a graph (or on a tree). Figure 20.6c shows the pattern in which nodes are found.

## Loop Invariants:

- LI1:** The nodes in the stack  $foundNotHandled$  are ordered such that they define a path starting at  $s$ .

**LI2:**  $foundNotHandled$  is a stack of tuples  $\langle v, i_v \rangle$  such that for each  $v$ , the first  $i_v$  edges of  $v$  have been handled. (Each node  $v$  appears no more than once.)

## Code:

**algorithm** *DepthFirstSearch* ( $G, s$ )

***⟨pre-cond⟩*:**  $G$  is a (directed or undirected) graph and  $s$  is one of its nodes.

$\langle \text{post-cond} \rangle$ : The output is a depth-first search tree of  $G$  rooted at  $s$ .

```

begin
  foundHandled =  $\emptyset$ 
  foundNotHandled =  $\{\langle s, 0 \rangle\}$ 
  time = 0 % Used for Time Stamping. See below.
loop
  loop-invariant: See above.
  exit when foundNotHandled =  $\emptyset$ 
  pop  $\langle u, i \rangle$  off the stack foundNotHandled
  if u has an  $(i+1)^{st}$  edge  $\langle u, v \rangle$ 
    push  $\langle u, i + 1 \rangle$  onto foundNotHandled
    if v has not previously been found then
       $\pi(v) = u$ 
       $\langle u, v \rangle$  is a tree edge
      push  $\langle v, 0 \rangle$  onto foundNotHandled

```

```

 $s(v) = \text{time}; \text{time} = \text{time} + 1$ 
else if  $v$  has been found but not completely handled then
     $\langle u, v \rangle$  is a back edge
else ( $v$  has been completely handled)
     $\langle u, v \rangle$  is a forward or cross edge
end if
else
    move  $u$  to  $\text{foundHandled}$ 
     $f(v) = \text{time}; \text{time} = \text{time} + 1$ 
end if
end loop
return  $\text{foundHandled}$ 
end algorithm

```

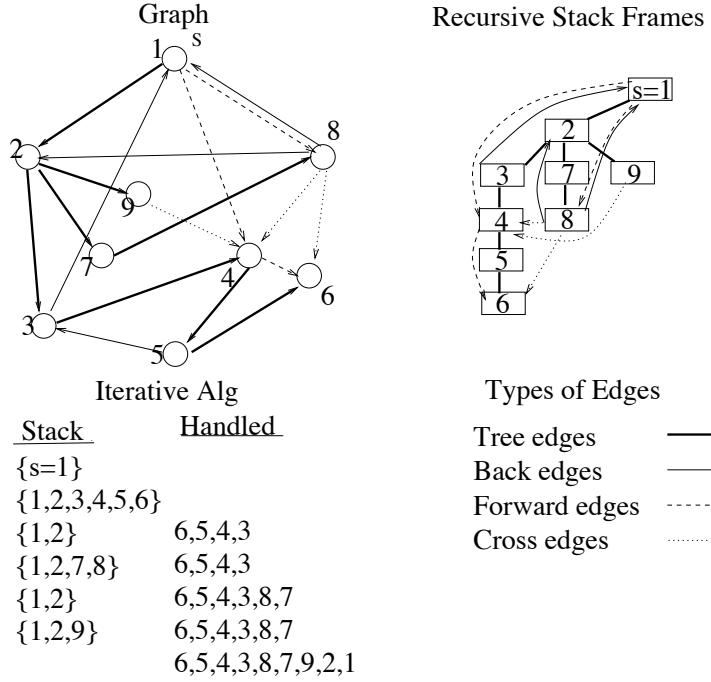


Figure 20.7: Depth-First Search of a Graph. The numbers give the order in which the nodes are found. The contents of the stack are given at each step.

**Establishing and Maintaining the Loop Invariant:** It is easy to see that with  $\text{foundNotHandled} = \{\langle s, 0 \rangle\}$ , the loop invariant is established. If the stack does contain a path from  $s$  to  $u$  and  $u$  has an unhandled edge to  $v$ , then  $u$  is kept on the stack and  $v$  is pushed on top. This extends the path from  $u$  onward to  $v$ . If  $u$  does not have an unhandled edge, then  $u$  is popped off the stack. This decreases the path from  $s$  by one.

**Classification of Edges:** The depth-first search algorithm can be used to classify edges.

**Tree Edges:** Tree edges are the edges  $\langle u, v \rangle$  in the depth-first search tree. When such edges are handled,  $v$  has not yet been found.

**Back Edges:** Back edges are the edges  $\langle u, v \rangle$  such that  $v$  is an ancestor of  $u$  in the depth-first search tree. When such edges are handled,  $v$  is in the stack, i.e., found but not completely handled.

**Cyclic:** A graph is cyclic if and only if it has a back-edge.

Proof ( $\Leftarrow$ ): The loop invariant of the depth-first search algorithm ensures that the contents of the stack forms a path from  $s$  through  $v$  and onward to  $u$ . Adding on the edge  $\langle u, v \rangle$  creates a cycle back to  $v$ .

Proof ( $\Rightarrow$ ): Later we prove that if the graph has no back edges then there is a total ordering of the nodes respecting the edges and hence the graph has no cycles.

**Bipartite:** A graph is bipartite if and only if there is no back-edge between any two nodes with the same level-parity, i.e., iff it has no odd length cycles.

**Forward Edges and Cross Edges:** Forward edges are the edges  $\langle u, v \rangle$  such that  $v$  is a descendant of  $u$  in the depth-first search tree.

Cross edges  $\langle u, v \rangle$  are such that  $u$  and  $v$  are in different branches of the depth-first search tree (i.e., are neither ancestors or descendants of each other) and  $v$ 's branch is traversed before (to the “left” of)  $u$ 's branch.

When forward edges and cross edges are handled,  $v$  has been completely handled. The depth-first search algorithm does not distinguish between forward edges and cross edges.

**Exercise 20.4.1** Prove that when doing DFS on undirected graphs there are never any forward or cross edges.

**Time Stamping:** Some implementations of depth-first search time stamp each node  $u$  with a start time  $s(u)$  and a finish time  $f(u)$ . Here “time” is measured by starting a counter at zero and incrementing it every time a node is found for the first time or a node is completely handled.  $s(u)$  is the time at which node  $u$  is first found and  $f(u)$  is the time at which it is completely handled. The time stamps are useful in the following way.

- $v$  is a descendant of  $u$  if and only if the time interval  $[s(v), f(v)]$  is completely contained in  $[s(u), f(u)]$ .
- If  $u$  and  $v$  are neither ancestor or descendant of each other, then the time intervals  $[s(u), f(u)]$  and  $[s(v), f(v)]$  are completely disjoint.

Using the time stamps, this can be determined in constant time.

## 20.5 Recursive Depth-First Search

We now present a recursive implementation of depth-first search algorithm, which directly mirrors the iterative version. The only difference is that the iterative version uses a stack to keep track of the route back to the start node, while the recursive version uses the stack of recursive stack frames.

**Code:**

**algorithm** *DepthFirstSearch* ( $s$ )

***pre-cond*:** An input instance consists of a (directed or undirected) graph  $G$  with some of its nodes marked *found* and a source node  $s$ .

***(post-cond)***: The output is the same graph  $G$  except all nodes  $v$  reachable from  $s$  without passing through a previously found node are now also marked as being found. The graph  $G$  is an global variable Abstract Data Type which is assumed to be both input and output to the routine.

```

begin
  if  $s$  is marked as found then
    do nothing
  else
    mark  $s$  as found
    for each  $v$  connected to  $s$ 
      DepthFirstSearch ( $v$ )
    end for
  end if
end algorithm

```

**Exercise 20.5.1** Trace out the iterative and the recursive algorithms on the same graph and see how they compare.

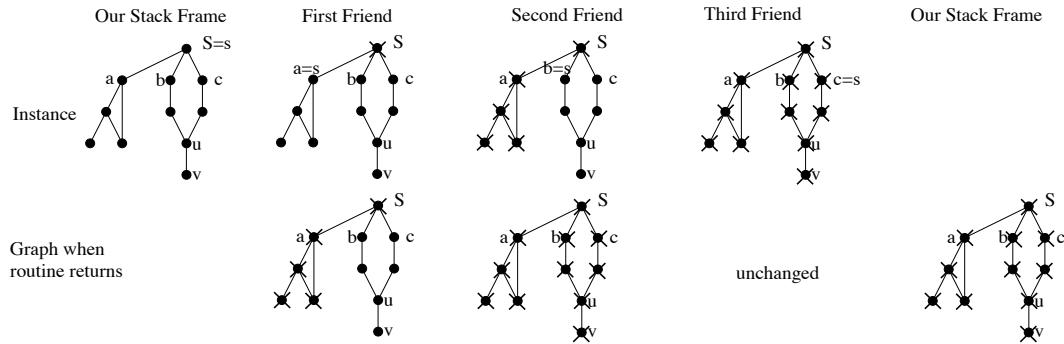


Figure 20.8: An Example Instance Graph

**Example:** Consider the instance graph in Figure 20.8.

**Pruning Paths:** There are two obvious paths from node  $S$  to node  $v$ . However, there are actually an infinite number of such paths. One path of interest is the one that starts at  $S$ , traverses around past  $u$  up to  $c$  and then down to  $v$ . All of these equally valued paths will be pruned from consideration, except the one that goes from  $S$  through  $b$  and  $u$  directly to  $v$ .

**Three Friends:** Given this instance, we first mark our source node  $S$  with an  $x$  and then we recurse three times, once from each of  $a$ ,  $b$ , and  $c$ .

**Friend a:** Our first friend marks all nodes that are reachable from its source node  $a = s$  without passing through a previously marked node. This includes only the nodes in the left most branch, because when we marked our source  $S$ , we block his route to the rest of the graph.

**Friend b:** Our second friend does the same. He finds, for example, the path that goes from  $b$  through  $u$  directly to  $v$ . He also finds and marks the nodes back around to  $c$ .

**Friend c:** Our third friend is of particular interest. He finds that his source node,  $c$ , has already been marked. Hence, he returns without doing anything. This prunes off this entire branch of the recursion tree. The reason that he can do this is because for any path to a node that he would consider, another path to the same node has already been considered.

**Achieving The Postcondition:** Consider the component of the graph reachable from our source  $s$  without passing through a previously marked nodes. (Because our instance has no marked nodes, this includes all the nodes.) To mark the nodes within this component, we do the following. First, we mark our source  $s$ . This partitions our component of reachable nodes into subcomponents that are still reachable from each other. Each such subcomponent has at least one edge from  $s$  into it. When we traverse the first such edge, this friend marks all the nodes within this subcomponent.

**Running Time:** Marking a node before it is recursed from insures that each node is recursed from at most once. Recursing from a node involves traversing each edge from it. Hence, each edge is traversed at most twice: once from each direction. Hence, the running time is linear in the number of edges.

## 20.6 Linear Ordering of a Partial Order

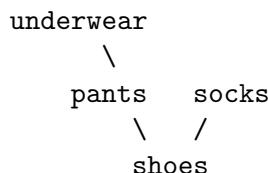
Finding a linear order consistent with a given partial order is one of many applications of a depth-first search. Hint: If a question ever mentions that a graph is directed acyclic always start by running this algorithm.

### Total and Partial Orders:

**Def<sup>n</sup>:** A *total order* of a set of objects  $V$  specifies for each pair of objects  $u, v \in V$  either (1) that  $u$  is *before*  $v$  or (2) that  $v$  is *before*  $u$ . It must be *transitive*, in that if  $u$  is before  $v$  and  $v$  is before  $w$ , then  $u$  is before  $w$ .

**Def<sup>n</sup>:** A *partial order* of a set of objects  $V$  supplies only some of the information of a total order. For each pair of objects  $u, v \in V$ , it specifies either that  $u$  is before  $v$ , that  $v$  is before  $u$ , or that the order of  $u$  and  $v$  is undefined. It must also be transitive.

For example, you must put on your underwear before your pants, and you must put on your shoes *after* both your pants and your socks. According to transitivity, this means you must put your underwear on *before* your shoes. However, you do have the freedom to put your underwear and your socks on in either order. My son, Josh, when six mistook this partial order for a total order and refuses to put on his socks before his underwear. When he was eight, he explained to me that the reason that he could get dressed faster than me was that he had a “short-cut”, consisting of putting his socks on before his pants. I was thrilled that he had at least partially understood the idea of a partial order.



A partial order can be represented by a directed acyclic graph  $G$ . The vertices consist of the objects  $V$ , and the directed edge  $\langle u, v \rangle$  indicates that  $u$  is before  $v$ . It follows from transitivity that if there is a directed path in  $G$  from  $u$  to  $v$ , then we know that  $u$  is before  $v$ . A cycle in  $G$  from  $u$  to  $v$  and back to  $u$  presents a contradiction because  $u$  cannot be both before and after  $v$ .

**Specifications of the Problem:** *Topological Sort:*

**Preconditions:** The input is a directed acyclic graph  $G$  representing a partial order.

**Postconditions:** The output is a total order consistent with the partial order given by  $G$ , i.e.,  $\forall$  edges  $\langle u, v \rangle \in G$ ,  $u$  appears before  $v$  in the total order.

**An Easy but Slow Algorithm:**

**The Algorithm:** Start at any node  $v$  of  $G$ . If  $v$  has an outgoing edge, walk along it to one of its neighbors. Continue walking until you find a node  $t$  that has no outgoing edges. Such a node is called a *sink*. This process cannot continue forever because the graph has no cycles.

The sink  $t$  can go after every node in  $G$ . Hence, you should put  $t$  last in the total order, delete  $t$  from  $G$ , and recursively repeat the process on  $G - v$ .

**Running Time:** It takes up to  $n$  time to find the first sink,  $n - 1$  to find the second, and so on. The total time is  $\Theta(n^2)$ .

**Algorithm Using a Depth-First Search:** Start at any node  $s$  of  $G$ . Do a depth-first search starting at node  $s$ . After this search completes, nodes that are considered found will *continue* to be considered found, so should not be considered again. Let  $s'$  be any unfound node of  $G$ . Do a depth-first search starting at node  $s'$ . Repeat the process until all nodes have been found.

Use the time stamp  $f(u)$  to keep track of the order in which nodes are *completely handled*, i.e., removed from the stack. Output the nodes in the *reverse* order.

If you ever find a back edge, then stop and report that the graph has a cycle.

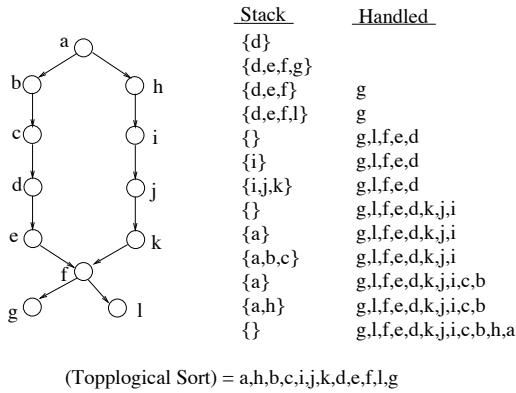


Figure 20.9: A Topological Sort Is Found Using a Depth-First Search.

**Proof of Correctness:**

**Lemma:** For every edge  $\langle u, v \rangle$  of  $G$ , node  $v$  is completely handled before  $u$ .

**Proof of Lemma:** Consider some edge  $\langle u, v \rangle$  of  $G$ . Before  $u$  is completely handled, it must be put onto the stack  $foundNotHandled$ . At this point in time, there are three cases:

**Tree Edge:**  $v$  has not yet been found. Because  $u$  has an edge to  $v$ ,  $v$  is put onto the top of the stack above  $u$  before  $u$  has been completely handled. No more progress will be made towards handling  $u$  until  $v$  has been completely handled and removed from the stack.

**Back Edge:**  $v$  has been found, but not completely handled, and hence is on the stack somewhere below  $u$ . Such an edge is a back edge. This contradicts the fact that  $G$  is acyclic.

**Forward or Cross Edge:**  $v$  has already been completely handled and removed from the stack. In this case, we are done:  $v$  was completely handled before  $u$ .

**Exercise 20.6.1** *Prove that the lemma is sufficient to prove that the reverse order in which the nodes were completely handled is a correct topological sort.*

**Running Time:** As with the depth-first search, no edge is followed more than once. Hence, the total time is  $\Theta(|E|)$ .

**Shortest-Weighted Path:** Suppose you want to find the shortest-weighted path for a graph  $G$  that you know is acyclic. You could use Dijkstra's algorithm from Section 20.3. However, as hinted above, whenever a question mentions that a graph is acyclic always start by finding a linear order consistent with the edges of the graph. Once this has been completed, you can handle the nodes (as done in Dijkstra's algorithm) in this linear order of the nodes.

**Proof of Correctness:** The shortest path to node  $v$  will not contain any nodes  $u$  that appear after it in the total order because by the requirements of the total order there is not path from  $u$  to  $v$ . Hence, it is fine to handle  $v$ , committing to a shortest path to  $v$ , before considering node  $u$ . Hence, it is fine to handle the nodes in the order given by the total order.

**Running Time:** The advantage of this algorithm is that you do not need to maintain a priority queue, as done in Dijkstra's algorithm. This decreases the time from  $\Theta(|E| \log |V|)$  to  $\Theta(|E|)$ .

# Chapter 21

## Network Flows

Network flow is a classic computational problem. Suppose that a given directed graph is thought of as a network of pipes starting at a source node  $s$  and ending at a sink node  $t$ . Through each pipe water can flow in one direction at some rate up to some maximum capacity. The goal is to find the maximum total rate at which water can flow from the source node  $s$  to the sink node  $t$ . If you physically had this network, you could determine the answer simply by pushing as much water through as you could. However, achieving this algorithmically is more difficult than you might first think because the exponentially many paths from  $s$  to  $t$  overlap winding forward and backwards in complicated ways. Being able to solve this problem, on the other hand, has a surprisingly large number of applications.

**Formal Specification:** Network Flow is another example of an optimization problem which involves searching for a best solution from some large set of solutions. See Section 19 for a formal definition.

**Instances:** An instance  $\langle G, s, t \rangle$  consists of a directed graph  $G$  and specific nodes  $s$  and  $t$ . Each edge  $\langle u, v \rangle$  is associated with a positive capacity  $c_{\langle u, v \rangle}$ .

**Solutions for Instance:** A solution for the instance is a *flow*  $F$  which specifies the flow  $F_{\langle u, v \rangle}$  through each edge of the graph. The requirements of a flow are as follows.

**Unidirectional Flow:** For any pair of nodes, it is easiest to assume that flow does not go in both directions between them. Hence, we will require that at least one of  $F_{\langle u, v \rangle}$  and  $F_{\langle v, u \rangle}$  is zero and that neither are negative.

**Edge Capacity:** The flow through any edge cannot exceed the capacity of the edge, namely  $F_{\langle u, v \rangle} \leq c_{\langle u, v \rangle}$ .

**No Leaks:** No water can be added at any node other than the source  $s$  and no water can be drained at any node other than the sink  $t$ . At each other node the total flow into the node equals the total flow out, namely for all nodes  $u \notin \{s, t\}$ ,  $\sum_v F_{\langle v, u \rangle} = \sum_u F_{\langle u, v \rangle}$ .

For example, see the left of Figure 21.1.

**Cost of Solution:** Typically in an optimization problem, each solution is assigned a “cost” that either must be maximized or minimized. For this problem, the cost of a flow  $F$ , denoted  $rate(F)$ , is the total rate of flow from the source  $s$  to the sink  $t$ . We will define this to be the total that leaves  $s$  without coming back, namely  $rate(F) = \sum_v [F_{\langle s, v \rangle} - F_{\langle v, s \rangle}]$ . Agreeing with our intuition, we will later prove that because no

flow leaks or is created in between  $s$  and  $t$ , this flow equals that flowing into  $t$  without leaving it, namely  $\sum_v [F_{\langle v,t \rangle} - F_{\langle t,v \rangle}]$ .

**Goal:** Given an instance  $\langle G, s, t \rangle$ , the goal is to find an optimal solution, i.e., a maximum flow.

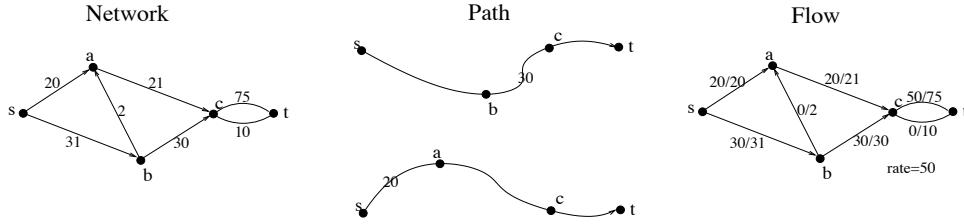


Figure 21.1: A network, with its edge capacities labeled, is given on the left. In the middle are two paths through which flow can be pushed. On the right, the resulting flow is given. The first value associated with each edge is its flow and the second is its capacity. The total rate of the flow is 50.

**Exercise 21.0.2** Some texts ensure that flow goes in only one direction between any two nodes, not by requiring that the flow in one direction  $F_{\langle v,u \rangle}$  is zero, but by requiring that  $F_{\langle v,u \rangle} = -F_{\langle u,v \rangle}$ . We do not do this in order to be more consistent with intuition and to emphasize the subtleties. The advantage, however, is that this change simplifies many of the equations. For example, the no leak requirement simplifies to  $\sum_v F_{\langle u,v \rangle} = 0$ . This exercise is to explain this change and then to redo all of the other equations in this section in a similar way.

In Section 21.1, we will design an algorithm for the network flow problem. We will see that this algorithm is an example of a *hill climbing algorithm* and that it does not necessarily work because it may get stuck in a *small local maximum*. In Section 21.2, we will modify the algorithm and use the *primal-dual* method to guarantee that it will find a global maximum. The problem is that the running time may be exponential. In Section 21.3, we prove that the *steepest ascent* version of this hill climbing algorithm runs in polynomial time. Finally, Section 21.4, relates these ideas to another more general problem called *linear programming*.

## 21.1 A Hill Climbing Algorithm with a Small Local Maximum

**Basic Steps:** The first step in designing an algorithm is to see what basic operations might be performed.

**Push from Source:** The first obvious thing to try is to simply start pushing water out of  $s$ . If the capacities of the edges near  $s$  are large then they can take lots of flow. Further down the network, however, the capacities may be smaller, in which case the flow that we started will get stuck. To avoid causing capacity violation or leaks, we will have to back off the flow that we started. Even further down the network, an edge may fork into edges with larger capacities, in which case we will need to decide in which direction to route the flow. Keeping track of this could be a headache.

**Plan Path For A Drop of Water:** A solution to both of the problem of flow getting stuck and the problem of routing flow along the way is to first find an entire path from  $s$  to

$t$  through which flow can take. In example from Figure 21.1, water can flow along the path  $\langle s, b, c, t \rangle$ . See the top middle of Figure 21.1. We then can push as much as possible through this path. It is easy to see that the bottle neck is edge  $\langle b, c \rangle$  with capacity 30. Hence, we add a flow of 30 to each edge along this path. That working well, we can try adding more water through another path. Let us try the path  $\langle s, a, c, t \rangle$ . The first interesting thing to note is that the edge  $\langle c, t \rangle$  in this path already has flow 30 through it. Because this edge has a capacity of 75, the maximum flow that can be added to it is  $75 - 30 = 40$ . This, however, turns out not to be the bottle neck because edge  $\langle s, a \rangle$  has capacity 20. Adding a flow of 20 to each edge along this path gives the flow shown on the right of Figure 21.1. For each edge, the left value gives its flow and the right gives its capacity. There being no more paths forward from  $s$  to  $t$ , we are now stuck. Is this the maximum flow?

**A Winding Path:** Water has a funny way of seeping from one place to another. It does not need to only go forward. Though the path  $\langle s, b, a, c, t \rangle$  winds backwards, more flow can be pushed through it. Another way to see that the concept of “forward” is not relevant to this problem, note that Figure 21.3 gives another layout of the exact same graph except that in this layout this path moves only forward. The bottle neck in adding flow through this path is edge  $\langle a, c \rangle$ . Already having a flow 20, its flow can only increase by 1. Adding a flow of 1 along this path gives the flow shown on the bottom left in Figure 21.2. Though this example reminds us that we need to consider all viable paths from  $s$  to  $t$ , we know that finding paths through a graph is easy using either breadth-first or depth-first search (Sections 20.2 and 20.4). However, in addition, we want to make sure that the path we find is such that we can add a non-zero amount of flow through it. For this, we introduce the idea of an *augmentation graph*.

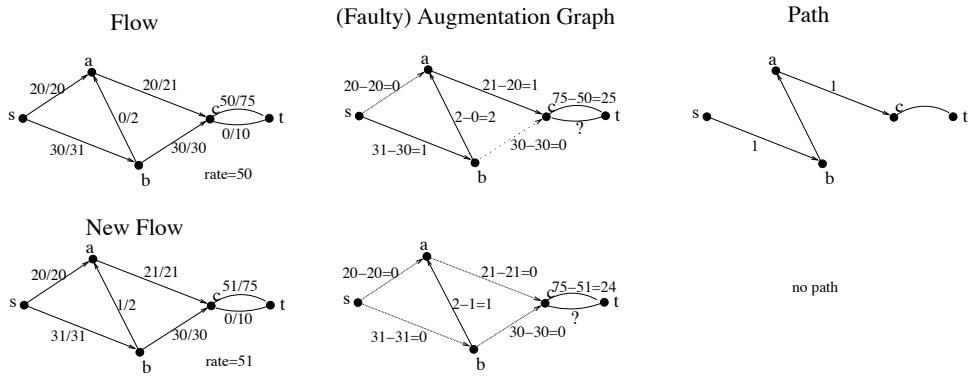


Figure 21.2: The top left is the same flow given in Figure 21.1, the first value associated with each edge being its flow and the second being its capacity. The top middle is a first attempt at an augmentation graph for this flow. Each edge is labeled with the amount of more flow that it can handle, namely  $c_{\langle u,v \rangle} - F_{\langle u,v \rangle}$ . The top right is the path in this augmentation graph through which flow is augmented. The bottom left is the resulting flow. The bottom middle is its (faulty) augmentation graph. No more flow can be added through it.

**The (faulty) Augmentation Graph:** Before we can find a path through which more flow can be added, we need to compute for each edge the amount of flow that can be added through this edge. To keep track of this information, we construct from the current flow  $F$  a graph denoted by  $G_F$  and called an *augmentation graph*. (Augment means

to add on. Augmentation is the amount you add on.) This graph initially will have the same directed edges as our network,  $G$ . Each of these edges is labeled with the amount by which its flow can be increased. We will call this the edge's *augment capacity*. Assuming that the current flow through the edge is  $F_{\langle u,v \rangle}$  and its capacity is  $c_{\langle u,v \rangle}$ , this augment capacity is given by  $c_{\langle u,v \rangle} - F_{\langle u,v \rangle}$ . Any edge for which this capacity is zero is deleted from the augmentation graph. Because of this, non-zero flow can be added along any path found from  $s$  to  $t$  within this augmentation graph. The path chosen will be called the *augmentation path*. The minimum augmentation capacity of any of its edges is the amount by which the flow in each of its edges is augmented. For an example, see Figure 21.2. In this case, the only path happens to be the path  $\langle s, b, a, c, t \rangle$ , which is the path that we used. Its path is augmented by a flow of 1.

We have now defined the basic step of the algorithm.

**The (faulty) Algorithm:** We can now easily fill in the remaining detail of the algorithm.

**The Loop Invariant:** The most obvious loop invariant is that at the top of the main loop we have a legal flow. It is possible that some more complex invariant will be needed, but for the time being this seems to be enough.

**The Measure of Progress:** The obvious measure of progress is how much flow the algorithm has managed to get between  $s$  and  $t$ , i.e., the rate  $rate(F)$  of the current flow.

**The Main Steps:** Given some current legal flow  $F$  through the network  $G$ , the algorithm improves the flow as follows: It constructs the augmentation graph  $G_F$  for the flow; finds an augmentation path from  $s$  to  $t$  through this graph using breadth-first or depth-first search; finds the edge in the path whose augmentation capacity is the smallest; and increases the flow by this amount through each edge in the path.

**Maintaining the Loop Invariant:** We must prove that the newly created flow is a legal flow in order to prove that  $\langle loop-invariant' \rangle \& \neg \langle exit-cond \rangle \& code_{loop} \Rightarrow \langle loop-invariant'' \rangle$ .

**Edge Capacity:** We are careful never to increase the flow of any edge by more than the amount  $c_{\langle u,v \rangle} - F_{\langle u,v \rangle}$ . Hence, its flow never increases beyond its capacity  $c_{\langle u,v \rangle}$ .

**No Leaks:** We are careful to add the same amount to every edge along a path from  $s$  to  $t$ . Hence, for any node  $u$  along the path there is one edge  $\langle v, u \rangle$  into the node whose flow changes and one edge  $\langle u, v' \rangle$  out of the node whose flow changes. Because these change by the same amount, the flow into the node remains equal to that out, namely for all nodes  $u \notin \{s, t\}$ ,  $\sum_v F_{\langle v, u \rangle} = \sum_v F_{\langle u, v \rangle}$ . In this way, we maintain the fact that the current flow has no leaks.

**Making Progress:** Because the edges whose flows could not change were deleted from the augmenting graph, we know that the flow through the path that was found can be increased by a positive amount. This increases the total flow. Because the capacities of the edges are integers, we can prove inductively that the flows are always integers and hence the flow increases by at least one. (Having fractions as capacities is fine, but having irrationals as capacities can cause the algorithm to run forever.)

**Initial Code:** We can start with a flow of zero through each edge. This establishes the loop invariant because this is a legal flow.

**Exit Condition:** At the moment, it is hard to imagine how we will know whether or not we have found the maximum flow. However, it is easy to see what will cause our algorithm to get stuck. If the augmenting graph for our current flow is such that there is no path in it from  $s$  to  $t$  then unless we can think of something better to do, we must exit.

**Termination:** As usual, we prove that this iterative algorithm eventually terminates because every iteration the rate of flow increases by at least one and because the total flow certainly cannot exceed the sum of the capacities of all the edges.

This completely defines an algorithm.

**Types of Algorithms:** At this point, we will back up and consider how the algorithm that we have just developed fits into three of the classic types of algorithms.

**Hill Climbing:** This algorithm is an example of an algorithmic technique known as hill climbing. Hiking at the age of seven, my son Josh stated that the way to find the top of the hill is simply to keep walking in a direction that takes you up and you know you are there when you cannot go up any more. Little did he know that this is also a common technique for finding the best solution for many optimization problems. The algorithm maintains one solution for the problem and repeatedly makes one of a small set of prescribed changes to this solution in a way that makes it a better solution. It stops when none of these changes is able to make a better solution. There are two problems with this technique. First, it is not necessarily clear how long it will take until the algorithm stops. The second is that sometimes it finds a small local maximum, i.e., the top of a small hill, instead of the overall global maximum. There are many hill climbing algorithms that are used extensively even though they are not guaranteed to work, because in practice they seem to work well.



**Greedy vs Backtracking:** Chapter 22 describes a class of algorithms known as greedy algorithms in which no decision that is made is revoked. Chapter 23 describes another class of algorithms known as recursive backtracking algorithms that continually notices that it has made a mistake and backtracks trying other options until a correct sequence of choices is made.

The network flow algorithm just developed could be considered to be greedy because once the algorithm decides to put flow through an edge it may later add more, but it never removes flow. Given that our goal is to get as much flow from  $s$  to  $t$  as possible and that it does not matter how that flow gets there, it makes sense that such a greedy approach would work.

We will now return to the algorithm that we have developed and determine whether or not it works.

**A Counterexample:** Proving that a given algorithm works for every input instance can be a major challenge. However, in order to prove that it does not work, we only need to give one input instance in which it fails. Figure 21.3 gives such an example. It traces out the algorithm on the same instance from Figure 21.1 that we did before. However, this time the algorithm happens to choose different paths. First it puts a flow of 2 through the path  $\langle s, b, a, c, t \rangle$ , followed by a flow of 19 through  $\langle s, a, c, t \rangle$ , followed by a flow of 29 through  $\langle s, b, c, t \rangle$ . At this point, we are stuck because the augmenting graph does not contain a path from  $s$  to  $t$ . This is a problem because the current flow is only 50, while we have already seen that the flow for this network can be 51. In hill climbing terminology, this flow is a small local maximum because we cannot improve it using the steps that we have allowed, but it is not a global maximum because there is a better solution.

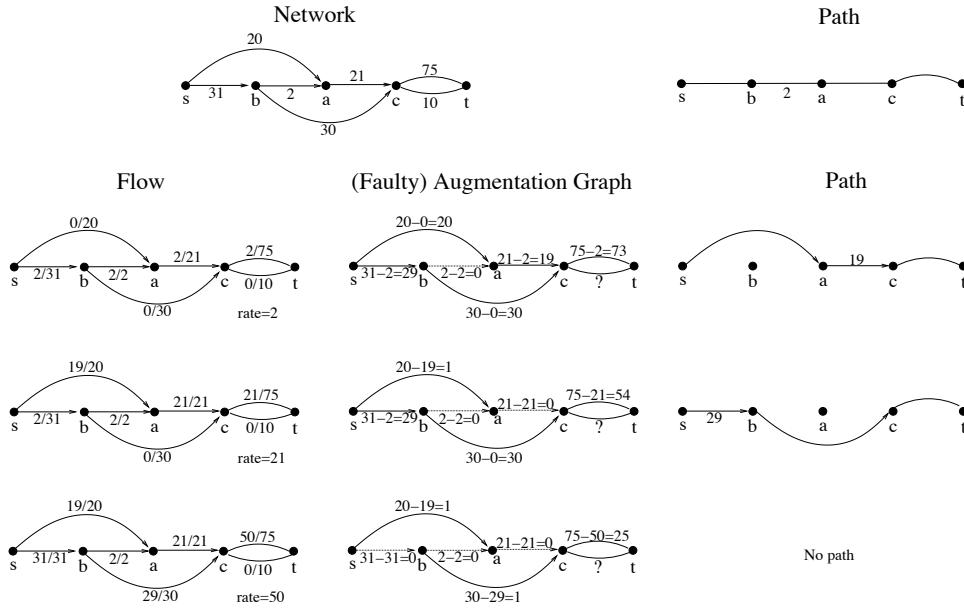


Figure 21.3: The faulty algorithm is traced on the instance from Figure 21.1. The nodes in this graph are laid out differently to emphasize the first path chosen. The current flow is given on the left, the corresponding augmentation graph in the middle, the augmenting path on the right, and the resulting flow on the next line. The algorithm gets stuck in a suboptimal local maximum.

**Where We Went Wrong:** From a hill climbing perspective, we took a step in an arbitrary direction that takes us up but with our first attempt we happened to head up the big hill and in the second we happened to head up the small hill. The flow of 51 that we obtained first turns out to be the unique maximum solution (often there are more than one possible maximum solutions). Hence, we can compare it to our present solution to see where we went wrong. In the first step, we put a flow of 2 through the edge  $\langle b, a \rangle$ , however, in the end it turns out that putting more than 1 through it is a mistake.

**Fixing the Algorithm:** The following are possible ways of fixing bugs like this one.

**Make Better Decisions:** We have seen that if we start by putting flow through the path  $\langle s, b, c, t \rangle$  then the algorithm works but if we start with the path  $\langle s, b, a, c, t \rangle$  it does not. One way of fixing the bug is finding some way to choose which path to add flow to next so that we do not get stuck in this way. From the greedy algorithm's perspective, if we are going to commit to a choice then we better make a good one. I know of no way to fix the network flows algorithm in this way.

**Backtrack:** If we make a decision that is bad, then we must back track and change it. In this example, we need to find a way of decreasing the flow through the edge  $\langle b, a \rangle$  from 2 down to 1. A general danger of backtracking algorithms over greedy algorithms is that the algorithm will have a much longer running time if it keeps changing its mind. From both a iterative algorithm and a hill climbing perspective, you cannot have an iteration that decreases your measure of progress by taking a step down the hill or else there is a danger that the algorithm runs for ever.

**Take Bigger Steps:** One way of avoiding getting stuck at the top of a small hill is to take a step that is big enough so that you step over the valley onto the slope of the bigger hill and a little higher up. Doing this requires redefine your definition of a “step.” This is the approach that we will take. We need to find a way of decreasing the flow through the edge  $\langle b, a \rangle$  from 2 down to 1 while maintaining the loop invariant that we have a legal flow and increasing the overall flow from  $s$  to  $t$ . The place in the algorithm in which we consider how the flow through an edge is allowed to change is when we define the augmenting graph. Hence, let us reconsider its definition.

## 21.2 The Primal-Dual Hill Climbing Method

We will now define a larger “step” that the hill climbing algorithm may take in hopes to avoid local maximum.

**The (correct) Algorithm:**

**The Augmentation Graph:** As before, the augmentation graph expresses how the flow in each edge is able to change.

**Forward Edges:** As before when an edge  $\langle u, v \rangle$  has flow  $F_{\langle u, v \rangle}$  and capacity  $c_{\langle u, v \rangle}$ , we put the corresponding edge  $\langle u, v \rangle$  in the augmenting graph with augment capacity  $c_{\langle u, v \rangle} - F_{\langle u, v \rangle}$  to indicate that we are allowed to add this much flow from  $u$  to  $v$ .

**Reverse Edges:** Now we see that there is a possibility that we might want to decrease the flow from  $u$  to  $v$ . Given that its current flow is  $F_{\langle u, v \rangle}$ , this is the amount that it can be decreased by. Effectively this is the same as increasing the flow from  $v$  to  $u$  by this same amount. Moreover, however, if the reverse edge  $\langle v, u \rangle$  is also in the graph and has capacity  $c_{\langle v, u \rangle}$ , then we are able to increase the flow from  $v$  to  $u$  by this second amount  $c_{\langle v, u \rangle}$  as well. Therefore, when the edge  $\langle u, v \rangle$  has flow  $F_{\langle u, v \rangle}$  and the reverse edge  $\langle v, u \rangle$  has capacity  $c_{\langle v, u \rangle}$ , we also put the reverse edge  $\langle v, u \rangle$  in the augmenting graph with augment capacity  $F_{\langle u, v \rangle} + c_{\langle v, u \rangle}$ . For more intuition see Figure 21.4 and for an example see edge  $\langle c, t \rangle$  in the second augmenting graph in Figure 21.5.

If instead the reverse edge  $\langle v, u \rangle$  has the flow in  $F$ , then we do the reverse. If neither edges have flow, then both edges with their capacities are added to the augmenting graph.

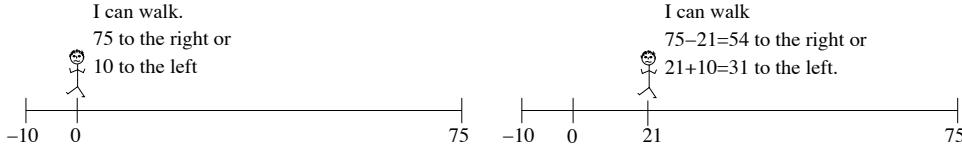


Figure 21.4: Suppose that from my home, I can walk 75 km to the right or 10 to the left. Then if I am already 21 km to the right, then I can walk  $75 - 21 = 54$  km to the right or  $21 + 10 = 31$  to the left. Suppose that my bank account can only hold \$75 or go into over draft of up to \$10. If I already have \$21 in the account, then I am able to add  $75 - 21 = \$54$  or remove  $21 + 10 = \$31$ .

**The Main Steps:** Little else changes in the algorithm. Given some current legal flow  $F$  through the network  $G$ , the algorithm improves the flow as follows: It constructs the augmentation graph  $G_F$  for the flow; finds an augmentation path from  $s$  to  $t$  through this graph using breadth-first or depth-first search; finds the edge in the path whose augmentation capacity is the smallest; and increases the flow by this amount through each edge in the path. If the edge in the augmenting graph is in the opposite direction as that in the flow graph then this involves decreases its flow by this amount. Recall that this is because increasing flow from  $v$  to  $u$  is effectively the same as decreasing it from  $u$  to  $v$ .

**Continuing The Example:** Figure 21.5 traces this new algorithm on the same example as in Figure 21.3. Note how the new augmenting graphs include edges in the reverse direction. Each step is the same as that in Figure 21.3, until the last step, in which these reverse edges provide the path  $\langle s, a, b, c, t \rangle$  from  $s$  to  $t$ . The bottle neck in this path is 1. Hence, we increase the flow by 1 in each edge in the path. The effect is that the flow through the edge  $\langle b, a \rangle$  decreases from 2 to 1 giving the optimal flow that we had obtained before.

**Bigger Step:** The reverse edges that have been added to the augmentation graph may well not be needed. They do, after all, undo flow that has already been added through an edge. On the other hand, having more edges in the augmentation graph can only increase the possibility of there being a path from  $s$  to  $t$  through it.

**Maintaining the Loop Invariant and Making Progress:** Little changes in the proof that these steps increase the total flow without violating any edge capacities or creating leaks at nodes, except that now one need to be a little more careful with your plus and minus signs. This will be left as an exercise.

**Exercise 21.2.1** *Prove that the new flow is legal.*

**Exit Condition:** As before the algorithm exits when it gets stuck because the augmenting graph for our current flow is such that there is no path in it from  $s$  to  $t$ . However, with more edges in our augmenting graph this may not occur as soon.

**Minimum Cut:** We will define this later.

**Code:**

**algorithm** *NetworkFlow* ( $G, s, t$ )

***{pre-cond}*:**  $G$  is a network given by a directed graph with capacities on the edges.  
 $s$  is the source node.  $t$  is the sink.

***{post-cond}*:**  $F$  specifies a maximum flow through  $G$  and  $C$  specifies a minimum cut.

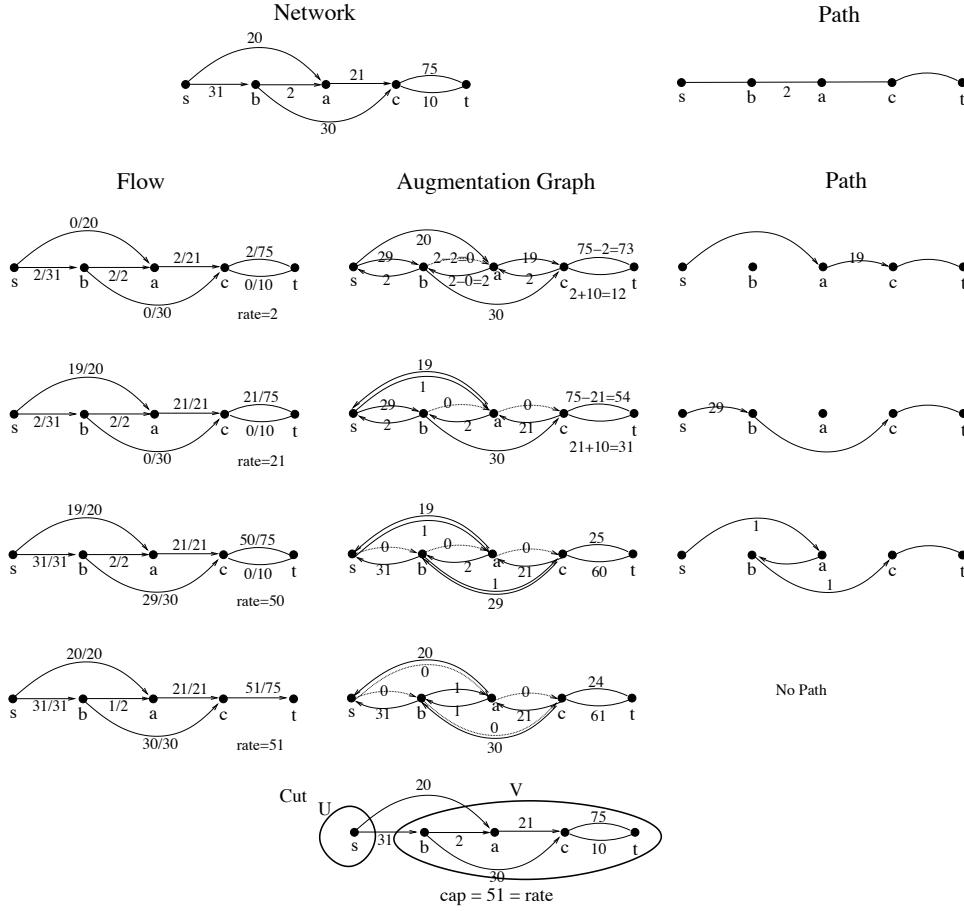


Figure 21.5: The correct algorithm is traced as was done in Figure 21.3. The current flow is given on the left, the corresponding augmentation graph in the middle, the augmenting path on the right, and the resulting flow on the next line. The optimal flow is obtained. The bottom figure is a minimum cut  $C = \langle U, V \rangle$ .

```

begin
   $F =$  the zero flow
  loop ⟨loop-invariant⟩:  $F$  is a legal flow.
     $G_F =$  the augmenting graph for  $F$ , where
      edge  $\langle u, v \rangle$  has augment capacity  $c_{\langle u, v \rangle} - F_{\langle u, v \rangle}$  and
      edge  $\langle v, u \rangle$  has augment capacity  $c_{\langle v, u \rangle} + F_{\langle v, u \rangle}$ .
    exit when  $s$  is not connected to  $t$  in  $G_F$ 
     $P =$  a path from  $s$  to  $t$  in  $G_F$ 
     $w =$  the minimum augmenting capacity in  $P$ 
    Add  $w$  to the flow  $F$  in every edge in  $P$ 
  end loop
   $U =$  nodes reachable from  $s$  in  $G_F$ 
   $V =$  nodes not reachable from  $s$  in  $G_F$ 
   $C = \langle U, V \rangle$ 

```

```

    return(  $F,C$ )
end algorithm

```

**Ending:** The next step in proving that this improved algorithm works correctly is to prove that it always finds a global maximum without getting stuck in a small local maximum. Using the notation of iterative algorithms, we must prove that  $\langle \text{loop-invariant} \rangle \& \langle \text{exit-cond} \rangle \& \text{code}_{\text{post-loop}} \Rightarrow \langle \text{post-cond} \rangle$ . From the loop invariant we know that the algorithm has a legal flow. Because we have exited, we know that the augmenting graph does not contain a path from  $s$  to  $t$  and hence we are stuck at the top of a local maximum. We must prove that there are no small local maximum and hence we must be at a global maximum and hence have an optimal flow. The method used is called the *primal-dual method*.

**Primal-Dual Hill Climbing** As an analogy suppose that over the hills on which we are climbing there is an exponential number of roofs one on top of the other. As before our problem is to find a place to stand on the hills that has maximum height. We call this the *primal* optimization problem. An equally challenging problem is to find the lowest roof. We call this the *dual* optimization problem. One thing that is easy to prove is that each roof is above each place to stand. It follows trivially that lowest and hence optimal roof is above the highest and hence optimal place to stand, but off hand we do not know how much above it is.

We say that a hill climbing algorithm gets stuck when it is unable to step in a way that moves it to a higher place to stand. A primal-dual hill climbing algorithm is able to prove that the only reason for getting stuck is because the place it is standing is pressed up against a roof. This is proved by proving that from any location, it can either step to higher location or specify a roof to which this location is adjacent. We will now see how these conditions are sufficient for proving what we want.

**Lemma: Finds Optimal.** A primal-dual hill climbing algorithm is guaranteed to find an optimal solution to both the primal and the dual optimization problems.

**Proof:** By the design of the algorithm, it only stops when it has a location  $L$  and a roof  $R$  with matching heights, i.e.,  $\text{height}(L) = \text{height}(R)$ . This location must be optimal because every other location  $L'$  must be below this roof and hence cannot be higher than this location, i.e.,  $\forall L', \text{height}(L') \leq \text{height}(R) = \text{height}(L)$ . We say that this dual solution  $R$  *witnesses* the fact that the primal solution  $L$  is optimal. Similarly, this primal solution  $L$  witnesses the fact that the dual solution  $R$  is optimal, namely  $\forall R', \text{height}(R') \geq \text{height}(L) = \text{height}(R)$ . This is called the *duality principle*.

**Cuts As Upper Bounds:** In order to apply these ideas to the network flow problem, we must find some upper bounds on the flow between  $s$  and  $t$ . Through a single path, the capacity of each edge acts as upper bound because the flow through the path cannot exceed the capacity of any of its edges. The edge with the smallest capacity, being the lowest upper bound, is the bottleneck. In a general network, a single edge cannot act as bottleneck because the flow might be able to go around this edge via other edges. A similar approach, however, works. Suppose that we wanted to bound the traffic between Toronto and Berkeley. We know that any such flow must cross the Canadian/US border. Hence, there is no need to worry about what the flow might do within Canada or within the US. We can safely say that the flow from Toronto to Berkeley is bounded above by the sum of the capacities of all the border crossings. Of course, this does not mean that this flow can be achieved. Other upper bounds can be obtained by summing up

the border crossing for other regions. For example, you could bound the traffic leaving Toronto, leaving Ontario, entering California, or entering Berkeley. This brings us to the following definition.

**Cut of a Graph:** A cut  $C = \langle U, V \rangle$  of a graph is a partitioning of the nodes of the graph into two sets  $U$  and  $V$  such that the source  $s$  is in  $U$  and the sink  $t$  is in  $V$ . The capacity of a cut is the sum of the capacities of all edges from  $U$  to  $V$ , namely  $\text{cap}(C) = \sum_{u \in U} \sum_{v \in V} c_{\langle u, v \rangle}$ .

One thing to note is that because the nodes in a graph do not have a “location” as cities do, there is no reason for the partition of the nodes to be “geographically contiguous.” Anyone of the exponential number of partitions will do.

**Flow Across a Cut:** To be able to compare the rate of flow from  $s$  to  $t$  with the capacity of a cut, we will first need to define the flow across a cut.

**rate( $F, C$ ):** Define  $\text{rate}(F, C)$  to be the current flow  $F$  across the cut  $C$ , which is the total of all flow in edges that cross from  $U$  to  $V$  minus the total of all the flow that comes back, namely  $\text{rate}(F, C) = \sum_{u \in U} \sum_{v \in V} [F_{\langle u, v \rangle} - F_{\langle v, u \rangle}]$ .

**rate( $F$ ) = rate( $F, \langle \{s\}, G - \{s\} \rangle$ ):** Recall that the flow from  $s$  to  $t$  was defined to be the total flow that leaves  $s$  without coming back, namely  $\text{rate}(F) = \sum_v [F_{\langle s, v \rangle} - F_{\langle v, s \rangle}]$ . You will note that this is precisely the equation for the flow across the cut that puts  $s$  all by itself, namely that  $\text{rate}(F) = \text{rate}(F, \langle \{s\}, G - \{s\} \rangle)$ .

**Lemma:  $\text{rate}(F, C) = \text{rate}(F)$ .** Intuitively this makes sense. Because no water leaks or is created between the source  $s$  and the sink  $t$ , the flow out of  $s$  equals the flow across any cut between  $s$  and  $t$ , which in turn equals the flow into  $t$ . It is because these are the same that we simply call this the flow from  $s$  to  $t$ . The intuition for the proof is that because the flow into a node is the same as that out of the node, if you move the node from one side of the cut to the other this does not change the total flow across the cut. Hence we can change the cut one node at a time from being the one containing only  $s$  to being the cut that we are interested in.

More formally this is done by induction on the size of  $U$ . For the base case,  $\text{rate}(F) = \text{rate}(F, \langle \{s\}, G - \{s\} \rangle)$  gives us that our hypothesis  $\text{rate}(F, C) = \text{rate}(F)$  is true for every cut which has only one node in  $U$ . Now suppose that by way of induction, we assume that it is true for every cut which has  $i$  nodes in  $U$ . We will now prove it for those cuts that have  $i + 1$  nodes in it. Let  $C = \langle U, V \rangle$  be any such cut. Choose one node  $x$  (other than  $s$ ) from  $U$  and move it across the boarder. This gives us a new cut  $C' = \langle U - \{x\}, V \cup \{x\} \rangle$ , where the side  $U - \{x\}$  contains only  $i$  nodes. Our assumption then gives us that the flow across this cut is equal to the flow of  $F$ , i.e.,  $\text{rate}(F, C') = \text{rate}(F)$ . Hence, in order to prove that  $\text{rate}(F, C) = \text{rate}(F)$ , we only need to prove that  $\text{rate}(F, C) = \text{rate}(F, C')$ . We will do this by proving that the difference between these is zero. By definition

$$\begin{aligned} & \text{rate}(F, C) - \text{rate}(F, C') \\ &= \left[ \sum_{u \in U} \sum_{v \in V} F_{\langle u, v \rangle} - F_{\langle v, u \rangle} \right] - \left[ \sum_{u \in U - \{x\}} \sum_{v \in V \cup \{x\}} F_{\langle u, v \rangle} - F_{\langle v, u \rangle} \right] \end{aligned}$$

Figure 21.6 shows the terms that do not cancel.

$$\begin{aligned}
&= \left[ \sum_{v \in V} F_{\langle x, v \rangle} - F_{\langle v, x \rangle} \right] - \left[ \sum_{u \in U} F_{\langle u, x \rangle} - F_{\langle x, u \rangle} \right] \\
&= \left[ \sum_{v \in V} F_{\langle x, v \rangle} - F_{\langle v, x \rangle} \right] + \left[ \sum_{v \in U} F_{\langle x, v \rangle} - F_{\langle v, x \rangle} \right] \\
&= \left[ \sum_v F_{\langle x, v \rangle} - F_{\langle v, x \rangle} \right] = 0
\end{aligned}$$

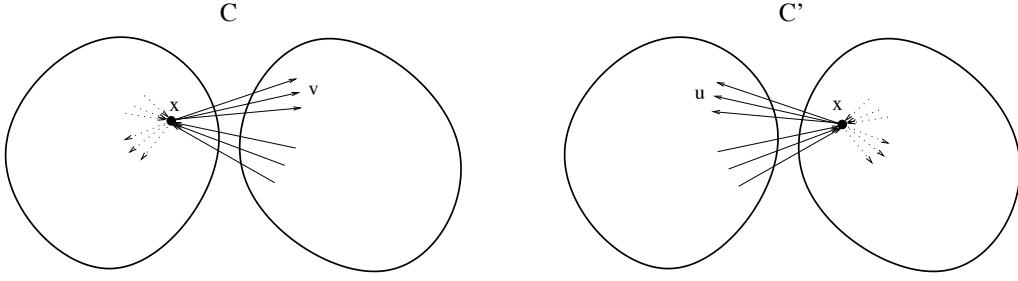


Figure 21.6: The edges across the cut that do not cancel in  $\text{rate}(F, C) - \text{rate}(F, C')$  are shown.

This last value is the total flow out of the node  $x$  minus the total flow into the node which is zero by the requirement of the flow  $F$  that no node leaks. This proves that  $\text{rate}(F, C) = \text{rate}(F)$  for every cut which has  $i + 1$  nodes in  $U$ . By way of induction, it then is true for all cuts for every size of  $U$ . In conclusion, this formally proves that given any flow  $F$ , its flow is the same across any cut  $C$ .

**Lemma:**  $\text{rate}(F) \leq \text{cap}(C)$ : It is now easy to prove that the rate  $\text{rate}(F)$  of any flow  $F$  is at most the capacity  $\text{cap}(C)$  of any cut  $C$ . In the primal-dual analogy, this proves that each roof is above each place to stand. Given  $\text{rate}(F) = \text{rate}(F, C)$ , it is sufficient to prove that the flow across a cut is at most the capacity of the cut. This follows easily from the definitions.

$\text{rate}(F, C) = \sum_{u \in U} \sum_{v \in V} [F_{\langle u, v \rangle} - F_{\langle v, u \rangle}] \leq \sum_{u \in U} \sum_{v \in V} [F_{\langle u, v \rangle}]$ , because having positive flow backwards across the cut from  $V$  to  $U$  only decreases the flow. Then this sum is at most  $\sum_{u \in U} \sum_{v \in V} [c_{\langle u, v \rangle}]$ , because no edge can have flow exceeding its capacity. Finally, this is the definition of the capacity  $\text{cap}(C)$  of the cut. This proves the required  $\text{rate}(F) \leq \text{cap}(C)$ .

**Take a Step or Find a Cut:** The primal-dual method requires that from any location, one can either step to higher location or specify a roof to which this location is adjacent. In the network flow problem, this translates to: given any legal flow  $F$ , being able to find either a better flow or a cut whose capacity is equal to the rate of the current flow. Doing this is quite intuitive. The augmenting graph  $G_F$  includes those edges through which the flow rate can be increased. Hence, the nodes reachable from  $s$  in this graph are the nodes to which more flow could be pushed. Let  $U$  denote this set of nodes. In contrast, the remaining set of nodes, which we will denote  $V$ , are those to which more flow cannot be pushed. See the cut at the bottom of Figure 21.5. No flow can be pushed across the border between  $U$  and  $V$  because all the edges crossing over are at capacity.

If  $t$  is in  $U$ , then there is a path from  $s$  to  $t$  through which the flow can be increased. On the other hand, if  $t$  is in  $V$ , then  $C = \langle U, V \rangle$  is a cut separating  $s$  and  $t$ . What remains is to formalize the proof that the capacity of this cut is equal to rate of the current flow.

**Lemma:  $\text{cap}(C) = \text{rate}(F)$ :** Above we proved  $\text{rate}(F, C) = \text{rate}(F)$ , i.e., that the rate of the current flow is the same as that across the cut  $C$ . It then remains only to prove  $\text{rate}(F, C) = \text{cap}(C)$ , i.e., that the current flow across the cut  $C$  is equal to the capacity of the cut.

**Lemma:  $\text{rate}(F, C) = \text{cap}(C)$ :** To prove this, it is sufficient to prove that every edge  $\langle u, v \rangle$  crossing from  $U$  to  $V$  has flow in  $F$  at capacity, i.e.  $F_{\langle u, v \rangle} = c_{\langle u, v \rangle}$  and every edge  $\langle v, u \rangle$  crossing back from  $V$  to  $U$  has zero flow in  $F$ . These give that  $\text{rate}(F, C) = \sum_{u \in U} \sum_{v \in V} [F_{\langle u, v \rangle} - F_{\langle v, u \rangle}] = \sum_{u \in U} \sum_{v \in V} [c_{\langle u, v \rangle} - 0] = \text{cap}(C)$ .

**$F_{\langle u, v \rangle} = c_{\langle u, v \rangle}$ :** Consider any edge  $\langle u, v \rangle$  crossing from  $U$  to  $V$ . If  $F_{\langle u, v \rangle} < c_{\langle u, v \rangle}$  then the edge  $\langle u, v \rangle$  with augment capacity  $c_{\langle u, v \rangle} - F_{\langle u, v \rangle}$  would be added to the augmenting graph. However, having such an edge in the augmenting graph contradicts the fact that  $u$  is reachable from  $s$  in the augmenting graph and  $v$  is not.

**$F_{\langle v, u \rangle} = 0$ :** If  $F_{\langle v, u \rangle} > 0$  then the edge  $\langle u, v \rangle$  with augment capacity  $c_{\langle u, v \rangle} + F_{\langle v, u \rangle}$  would be added to the augmenting graph. Again having such an edge is a contradiction.

This proves that  $\text{rate}(F, C) = \text{cap}(C)$ .

Having proved  $\text{rate}(F, C) = \text{cap}(C)$  and  $\text{rate}(F, C) = \text{rate}(F)$  gives us the required statement  $\text{cap}(C) = \text{rate}(F)$  that the flow we have found equals the capacity of the cut.

**Ending:** The conclusion of the above proofs is that this improved network flows algorithm always finds a global maximum without getting stuck in a small local maximum. Each iteration it either finds a path in the augmenting graph through which it can improve the current flow or it finds a cut that witnesses the fact that there are no better flows.

**Max Flow, Min Cut Duality Principle:** By accident, when proving that our network flow algorithm works, we proved two interesting things about a completely different computational problem, *the min cut problem*. This problem, given a network  $\langle G, s, t \rangle$  finds a cut  $C = \langle U, V \rangle$  whose capacity  $\text{cap}(C) = \sum_{u \in U} \sum_{v \in V} c_{\langle u, v \rangle}$  is minimum. The first interesting thing is that we have proved is that the maximum flow through this graph is equal to its minimum cut. The second interesting thing is that this algorithm to find a maximum flow also finds a minimum cut.

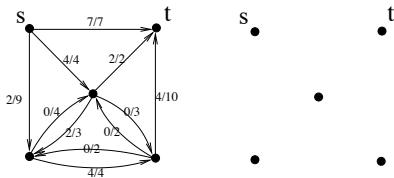
**Credits:** This algorithm was developed by Ford and Fulkerson in 1962.

**Running Time:** Above we proved that this algorithm eventually terminates because every iteration the rate of flow increases by at least one and because the total flow certainly can never exceed the sum of the capacities of all the edges. Now we must bound its running time.

**Exponential?:** Suppose that the network graph has  $m$  edges each with a capacity that is represented by an  $\mathcal{O}(\ell)$  bit number. Each capacity could be as large as  $\mathcal{O}(2^\ell)$  and the total maximum flow could be as large as  $\mathcal{O}(m \cdot 2^\ell)$ . Starting out as zero and increasing by about one each iteration, the algorithm would need  $\mathcal{O}(m \cdot 2^\ell)$  iterations until the maximum flow is found. This running time is polynomial in the number of edges  $m$ .

Recall, however, that the running time of an algorithm is expressed not as a function the number of values in the input nor as a function of the values themselves, but as a function of the size of the input instance, which in this case is the number of bits (or digits) needed to represent all of the values, namely  $\mathcal{O}(m \cdot \ell)$ . If  $\ell$  is large, then the number of iterations,  $\mathcal{O}(m \cdot 2^\ell)$ , is exponential in this size. This is a common problem with hill climbing algorithms.

**Exercise 21.2.2** Starting with the flow given below, complete the network flow algorithm.



**Exercise 21.2.3** In hill climbing algorithms there are steps that make lots of progress and those that make very little progress. For example, the first iteration on the input given in Figure 21.1 might find a path through the augmentation graph through which a flow of 30 can be added. It might, however, find the path through which only a flow of 2 can be added. The purpose of this question is to determine how bad the running time might be when the computation is unlucky enough to always take the worst legal step allowed by the algorithm. Start by taking the step that increases the flow by 2 thought the input given in Figure 21.1. Then continue to take the worst possible step. You could draw out each and every step, but it is better to use this opportunity to use loop invariants. What does the flow look like after  $i$  iterations? Repeat this process on the same graph except that the four edges forming the square now have capacities 1,000,000,000,000,000 and the cross over edge has capacity one. (Also move  $t$  to  $c$  or give that last edge a large capacity.)

1. What is the worst case number of iteration of this network flow algorithm as a function of the number of edges  $m$  in the input network?
2. What is the official “size” of a network?
3. What is the worst case number of iteration of this network flow algorithm as a function of the “size” of the input network.

**Exercise 21.2.4** If all the capacities in the given network are integers, prove that the algorithm always returns a solution in which the flow through each edge is an integer. For some applications, this fact is crucial.

## 21.3 The Steepest Assent Hill Climbing Algorithm

We have all experienced that climbing a hill can take a long time if you wind back and forth barely increasing your height at all. In contrast, you get there much faster if energetically you head straight up the hill. This method, which is call the method of *steepest assent*, is to always take the step that increases your height by the most. If you already know that the hill climbing algorithm in which you take any step up the hill works, then this new more specific algorithm also works. However, if we are lucky it finds the optimal solution faster.

In our network flow algorithm, the choice of what step to take next involves choosing which path in the augmenting graph to take. The amount the flow increases is the smallest augmentation capacity of any edge in this path. It follows that the choice that would give us the biggest improvement is the path whose smallest edge is the largest for any path from  $s$  to  $t$ . Our steepest ascent network flow algorithm will augment such a best path each iteration. What remains to be done is to give an algorithm that finds such a path and to prove that doing this, a maximum flow is found within a polynomial number of iterations.

**Finding The Augmenting Path With The Biggest Smallest Edge:** The new problem to be solved is as follows. The input consists of a directed graph with positive edge weights and with special nodes  $s$  and  $t$ . The output consists of a path from  $s$  to  $t$  through this graph whose smallest weighted edge is as big as possible.

**Easier Problem:** Before attempting to develop an algorithm for this, let us consider an easier but related problem. In addition to the directed graph, the input to the easier problem provides a weight denoted  $w_{min}$ . It either outputs a path from  $s$  to  $t$  whose smallest weighted edge is at least as big as  $w_{min}$  or states that no such path exists.

**Using the Easier Problem:** Assuming that we can solve this easier problem, we solve the original problem by running the first algorithm with  $w_{min}$  being every edge weight in the graph, until we find the weight for which there is a path with such a smallest weight, but there is not a path with a bigger smallest weight. This is our answer. (See Exercise 21.3.1 on the possibility of using binary search on the weights  $w_{min}$  to find the critical weight.)

**Solving the Easier Problem:** A path whose smallest weighted edge is at least as big as  $w_{min}$  will obviously not contain any edge whose weight is smaller than  $w_{min}$ . Hence, the answer to this easier problem will not change if we delete from the graph all edges whose weight is smaller. Any path from  $s$  to  $t$  in the remaining graph will meet our needs. If there is no such path then we also know there is no such path in our original graph. This solves the problem.

**Implementation Details:** In order to find a path from  $s$  to  $t$  in a graph, the algorithm branches out from  $s$  using breadth-first or depth-first search marking every node reachable from  $s$  with the predecessor of the node in the path to it from  $s$ . If in the process  $t$  is marked, then we have our path. (See Section 20.1.) It seems a waste of time to have to redo this work for each  $w_{min}$ . A standard algorithmic technique in such a situation is to use an iterative algorithm. The loop invariant will be that the work for the previous  $w_{min}$  has been done and is stored in a useful way. The main loop will then complete the work for the current  $w_{min}$  reusing as much of the previous work as possible. This can be implemented as follows. Sort the edges from biggest to smallest (breaking ties arbitrarily). Consider them one at a time. When considering  $w_i$ , we must construct the graph formed by deleting all the edges with weights smaller than  $w_i$ . Denote this  $G_{w_i}$ . We must mark every node reachable from  $s$  in this graph. Suppose that we have already done these things in the graph  $G_{w_{i-1}}$ . We form  $G_{w_i}$  from  $G_{w_{i-1}}$  simply by adding the single edge with weight  $w_i$ . Let  $\langle u, v \rangle$  denote this edge. Nodes are reachable from  $s$  in  $G_{w_i}$  that were not reachable in  $G_{w_{i-1}}$  only if  $u$  was reachable and  $v$  was not. This new edge then allows  $v$  to be reachable. In addition, other nodes may be reachable from  $s$  via  $v$  through other edges that we had added before. These can all be marked reachable simply by starting a depth first search from  $v$ , marking all those nodes that are now reachable that have not been marked reachable before. The algorithm will stop at the

first edge that allows  $t$  to be reached. The edge with the smallest weight in this path to  $t$  will be the edge with weight  $w_i$  added during this iteration. There is not a path from  $s$  to  $t$  in the input graph with a larger smallest weighted edge because  $t$  was not reachable when only the larger edges were added. Hence, this path is a path to  $t$  in the graph whose smallest weighted edge is the largest. This is the required output of this subroutine.

**Running Time:** Even though the algorithm for finding the path with the largest smallest edge runs depth-first search for each weight  $w_i$ , because the work done before is reused, no node in the process is marked reached more than once and hence no edge is traversed more than once. It follows that this process requires only  $\mathcal{O}(m)$  time, where  $m$  is the number of edges. This time, however, is dominated by the time  $\mathcal{O}(m \log m)$  to sort the edges.

**Code:**

```

algorithm LargestShortestWeight( $G, s, t$ )
  <pre-cond>:  $G$  is a weighted directed (augmenting) graph.  $s$  is the source node.  $t$  is the sink.
  <post-cond>:  $P$  specifies a path from  $s$  to  $t$  whose smallest edge weight is as large as possible.  $\langle u, v \rangle$  is its smallest weighted edge.

  begin
    Sort the edges by weight from largest to smallest
     $G'$  = graph with no edges
    mark  $s$  reachable
    loop
      <loop-invariant>: Every node reachable from  $s$  in  $G'$  is marked reachable.
      exit when  $t$  is reachable
       $\langle u, v \rangle$  = the next largest weighted edge in  $G$ 
      Add  $\langle u, v \rangle$  to  $G'$ 
      if(  $u$  is marked reachable and  $v$  is not ) then
        Do a depth-first search from  $v$  marking all reachable nodes not marked before.
      end if
    end loop
     $P$  = path from  $s$  to  $t$  in  $G'$ 
    return(  $P, \langle u, v \rangle$  )
  end algorithm

```

**Binary Search: Exercise 21.3.1** Could we use binary search on the weights  $w_{min}$  to find the critical weight (see Section 11.1) and if so would it be faster? Why?

**Running Time of Steepest Ascent:** What remains to be done is to determine how many times the network flow algorithm must augment the flow in a path when the path chosen is that whose augmentation capacity is the largest possible.

**Decreasing the Remaining Distance by Constant Factor:** The flow starts out as zero and may need to increase be as large as  $\mathcal{O}(m \cdot 2^\ell)$  when there are  $m$  edges with  $\ell$  bit capacities. We would like the number of steps to be not exponential but linear in  $\ell$ . One way to achieve this is to ensure that the current flow doubles each iteration. This, however, is likely not to happen. Another possibility is to turn the measure of progress

around. After the  $i^{th}$  iteration, let  $R_i$  denote the remaining amount that the flow must increase. More formally, suppose that the maximum flow is  $rate_{max}$  and that the rate of the current flow is  $rate(F)$ . The remaining distance is then  $R_i = rate_{max} - rate(F)$ . We will show that the amount  $w_{min}$  by which the flow increases is at least some constant fraction of  $R_i$ .

**Bounding The Remaining Distance:** The funny thing about this measure of progress, is that the algorithm does not know what the maximum flow  $rate_{max}$  is. However, it is only needed as part of the analysis. For this, we must bound how big the remaining distance,  $R_i = rate_{max} - rate(F)$ , is. Recall that the augmentation graph for the current flow is constructed so that the augmentation capacity of each edge gives the amount that the flow through this edge can be increased by. Hence, just as the sum of the capacities of the edges across any cut  $C = \langle U, V \rangle$  in the network, acts as an upper bound to the total flow possible, the sum of the augmentation capacities of the edges across any cut  $C = \langle U, V \rangle$  in the augmentation graph, acts as an upper bound to the total amount that the current flow can be increased.

**Choosing a Cut:** To do this analysis, we need to choose which cut we will use. (This is not part of the algorithm.) As before, the natural cut to use comes out of the algorithm that finds the path from  $s$  to  $t$ . Let  $w_{min} = w_i$  denote the smallest augmentation capacity in the path whose smallest augmentation capacity is largest. Let  $G_{w_{i-1}}$  be the graph created from the augmenting graph by deleting all edges whose augmentation capacities are smaller or equal to  $w_{min}$ . Note that this is the last graph that the algorithm which finds the augmenting path considers before adding the edge with weight  $w_{min}$  that connects  $s$  and  $t$ . We know that there is not a path from  $s$  to  $t$  in  $G_{w_{i-1}}$  or else there would be an path in the augmenting graph whose smallest augmenting capacity was larger then  $w_{min}$ . Form the cut  $C = \langle U, V \rangle$  by letting  $U$  be the set of all the nodes reachable from  $s$  in  $G_{w_{i-1}}$  and letting  $V$  be those that are not. Now consider any edge in the augmenting graph that crosses this cut. This edge cannot be in the graph  $G_{w_{i-1}}$  or else it would be crossing from a node in  $U$  that is reachable from  $s$  to a node that is not reachable from  $s$ , which is a contradiction. Because this edge has been deleted in  $G_{w_{i-1}}$ , we know that its augmentation capacity is at most  $w_{min}$ . The number of edges across this cut is at most the number of edges in the network, which has be denoted by  $m$ . It follows that the sum of the augmentation capacities of the edges across this cut  $C = \langle U, V \rangle$  is at most  $m \cdot w_{min}$ .

**Bounding The Increase,  $w_{min} \geq \frac{1}{m}R_i$ :** We have determined that  $R_i = rate_{max} - rate(F)$ , which is the remaining amount that the flow needs to be increased, is at most the sum of the augmentation capacities across the cut  $C$ , which is at most  $m \cdot w_{min}$ , i.e.,  $R_i \leq m \cdot w_{min}$ . Rearranging this gives that  $w_{min} \geq \frac{1}{m}R_i$ . It gives that  $w_{min}$ , which is the amount that the flow does increase by this iteration, is at least  $\frac{1}{m}R_i$ .

**The Number of Iterations:** We have determined that the flow increases each iteration by at least  $\frac{1}{m}$  times the remaining amount  $R_i$  that it must be increased. This, of course, decreases the remaining amount, giving that  $R_{i+1} \leq R_i - \frac{1}{m}R_i$ . You might think that it follows that the maximum flow is obtained in only  $m$  iterations. This would be true if  $R_{i+1} \leq R_i - \frac{1}{m}R_0$ . However, it is not because the smaller  $R_i$  gets, the smaller it decreases by. One way to bound the number of iterations needed is to note that  $R_i \leq (1 - \frac{1}{m})^i R_0$  and then to either bound logarithms base  $(1 - \frac{1}{m})$  or to know that  $\lim_{m \rightarrow \infty} (1 - \frac{1}{m})^m = \frac{1}{e} \approx \frac{1}{2.17}$ . However, I think that the following method is more

intuitive. As long as  $R_i$  is big, we know that it decreases by a lot. To make this concrete, let's consider what happens after some  $I^{th}$  iteration and say that  $R_i$  is still relatively big when it is still at least  $\frac{1}{2}R_I$ . As long as this is the case,  $R_i$  decrease by at least  $\frac{1}{m}R_i \geq \frac{1}{2m}R_I$ . After  $m$  such iterations,  $R_i$  would decrease from  $R_I$  to  $\frac{1}{2}R_I$ . The only reason that it would not continue to decrease this fast is if it already had decreased this much. Either way, we know that every  $m$  iterations,  $R_i$  decreases by a factor of two. This process may make you think of what is known as zeno's paradox. If you cut the remaining distance in half and then in half again and so on, then though you get very close very fast, you never actually get there. However, if all the capacities are integers then all values will be integers and hence when  $R_i$  decreases to be less than one, it must in fact be zero, giving us the maximum flow.

Initially, the remaining amount  $R_i = rate_{max} - rate(F)$  is at most  $\mathcal{O}(m \cdot 2^\ell)$ . Hence, if it decreases by at least a factor of two each  $m$  iterations, then after  $mj$  iterations, this amount is at most  $\mathcal{O}(\frac{m \cdot 2^\ell}{2^j})$ . This reaches one when  $j = \mathcal{O}(\log_2(m \cdot 2^\ell)) = \mathcal{O}(\ell + \log m)$  or  $\mathcal{O}(m\ell + m \log m)$  iterations. If your capacities are real numbers, then you will be able to approximate the maximum flow to within  $\ell'$  bits of accuracy in another  $m\ell'$  iterations.

**Bounding the Running Time:** We have determined that each iteration takes  $m \log m$  time and that only  $\mathcal{O}(m\ell + m \log m)$  iterations are required. It follows that this steepest assent network flow algorithm runs in time  $\mathcal{O}(\ell m^2 \log m + m^2 \log^2 m)$ .

**Fully Polynomial Time:** A lot of work was spent finding an algorithm that is what is known as *fully polynomial*. This requires that the number of iterations be polynomial in the number of values and does not depend at all on the values themselves. Hence, if you charge only one time step for addition and subtraction, even if the capacities are strange things like  $\sqrt{2}$ , then the algorithm gives the exact answer (at least symbolically) in polynomial time. My father, Jack Edmonds, and a colleague, Richard Karp, developed such an algorithm in 1972. It is version of the original Ford-Fulkerson algorithm. However, in this, each iteration, the path from  $s$  to  $t$  in the augmenting graph with the smallest number of edges is augmented. This algorithm iterates at most  $\mathcal{O}(nm)$  times, where  $n$  is the number of nodes and  $m$  the number of edges. In practice, this is slower than the  $\mathcal{O}(m\ell)$  time steepest assent algorithm.

## 21.4 Linear Programming

When I was an undergraduate, I had a summer job with a food company. Our goal was to make cheap hot dogs. Every morning we got the prices of thousands of ingredients: pig hearts, sawdust, .... Each ingredient has an associated variable indicating how much of it to add to the hot dogs. There are thousands of linear constraints on these variables: so much meat, so much moisture, .... Together these constraints specify which combinations of ingredients constitute a "hot dog." The cost of the hot dog is a linear function of what you put into it and their costs. The goal is to determine what to put into the hot dogs that day to minimize the cost. This is an example of a general class of problems referred to as *linear programs*.

**Formal Specification:** A linear program is an optimization problem whose constraints and objective functions are linear functions.

**Instances:** An input instance consists of (1) a set of linear constraints on a set of variables and (2) a linear objective function.

**Solutions for Instance:** A solution for the instance is a setting of all the variables that satisfies the constraints.

**Cost of Solution:** The cost of a solutions is given by the objective function.

**Goal:** The goal is to find a setting of these variables that optimizes the cost, while respecting all of the constraints.

**Examples:**

**Concrete Example:**

maximize

$$7x_1 - 6x_2 + 5x_3 + 7x_4$$

subject to

$$\begin{aligned} 3x_1 + 7x_2 + 2x_3 + 9x_4 &\leq 258 \\ 6x_1 + 3x_2 + 9x_3 - 6x_4 &\leq 721 \\ 2x_1 + 1x_2 + 5x_3 + 5x_4 &\leq 524 \\ 3x_1 + 6x_2 + 2x_3 + 3x_4 &\leq 411 \\ 4x_1 - 8x_2 - 4x_3 + 4x_4 &\leq 685 \end{aligned}$$

**Matrix Representation:** A linear program can be expressed very compactly using matrix algebra. Let  $n$  denote the number of variables and  $m$  the number of constraints. Let  $a$  denote the row of  $n$  coefficients in the objective function,  $M$  denote the matrix with  $m$  rows and  $n$  columns of coefficients on the left hand side of the constraints, let  $b$  denote the column of  $m$  coefficients on the right hand side of the constraints, and finally let  $x$  denote the column of  $n$  variables. Then the goal of the linear program is to maximize  $a \cdot x$  subject to  $M \cdot x \leq b$ .

**Network Flows:** The network flows problem can be expressed as instances of linear programming.

**Exercise 21.4.1** *Given a network flow instance, express it as a linear program.*

**The Euclidean Space Interpretation:** Each possible solution, giving values to the variables  $x_1, \dots, x_n$ , can be viewed as a point in  $n$  dimensional space. This space is easiest to view when there are only two or three dimensions, but the same ideas hold for any number of solutions.

**Constraints:** Each constraint specifies a boundary in space, on one side of which a valid solution must lie. When  $n = 2$ , this constraint is a one-dimensional line. See Figure 21.7. When  $n = 3$ , it is a two-dimensional plane, like the side of a box. In general, it is an  $n - 1$  dimensional space. The space bounded by all of the constraints is called a *polyhedral*.

**The Objective Function:** The objective function gives a direction in Euclidean space. The goal is to find a point in the bounded polyhedral that is the furthest in this direction. The best way to visualize this is to rotate the Euclidean space so that the objective function points straight up. For Figure 21.7, rotate the book so that the big arrow points upwards. Given this, the goal is to find a point in the bounded polyhedral that is as high as possible.

**A Vertex is an Optimal Solution:** You may recall that  $n$  linear equations with  $n$  unknowns are sufficient to specify a unique solution. Because of this,  $n$  constraints, when met with equality, intersect at one point. This is called a *vertex*. For example, you can

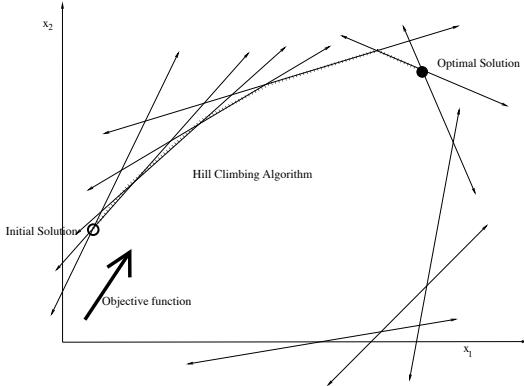


Figure 21.7: The Euclidean space representation of a linear program with  $n = 2$ .

see in Figure 21.7 how for  $n = 2$ , two lines defines a vertex. You can also see how for  $n = 3$ , three sides of a box defines a vertex.

As you can imagine from looking at Figure 21.7, if there is a unique solution, it will be at a vertex where  $n$  constraints meet. If there is a whole region of equivalently optimal solutions, then at least one of them will be a vertex. The advantage of this knowledge is that our search for an optimal solution will focus on these vertices.

**The Hill Climbing Algorithm:** Once the book is rotated to point the objective function in Figure 21.7 upwards, the obvious algorithm simply climbs the hill formed by the outside of the bounded polyhedral until the top is reached. Recall that hill climbing algorithms maintain a valid solution and each iteration take a “step”, replacing it with a better solution, until there is no better solution that can be obtained in one such “step.” The only things remaining in defining a hill climbing algorithm for linear programming is to devise a way to find an initial valid solution and to define what constitutes a “step” to a better solution.

**A Step:** Suppose by the loop invariant, we have a solution that in addition to being valid, it is also a vertex of the bounding polyhedral. More formally, the solution satisfying all of the constraints and meets  $n$  of the constraints with equality. A step will involve sliding along the edge (one dimensional line) between two adjacent vertices. This involves *relaxing* one of the constraints that is met with equality so that it no longer is met with equality and *tightening* one of the constraints that was not met with equality so that it now is met with equality. This is called *pivoting* out one equation and in another. The new solution will be the unique solution that satisfies with equality the  $n$  presently selected equations. Of course, each iteration such a step can be taken only if it continues to satisfy all of the constraints and improves the objective function. There are fast ways of finding a good step to take. However, even if you do not know these, there are only  $n \cdot m$  choices of “steps” to try, when there are  $n$  variables and  $m$  equations.

**Finding an Initial Valid Solution:** If we are lucky, the origin is a valid solution. However, in general finding some valid solution is itself a challenging problem. Our algorithm to do so will be an iterative algorithm that includes the constraints one at a time. Suppose by the loop invariant, we have vertex solution that satisfies the first  $i$  of the equations. For example, that we have a vertex solution that satisfies all of the constraints in our above concrete example except the last one, which happens to be  $4x_1 - 8x_2 - 4x_3 + 4x_4 \leq$

685. We will then treat the negative of this next constraint as the objective function, namely  $-4x_1 + 8x_2 + 4x_3 - 4x_4$ . We will run our hill climbing algorithm, starting with the vertex we have until, we have a vertex solution that maximizes this new objective function subject to the first  $i$  equations. This is equivalent to minimizing the objective  $4x_1 - 8x_2 - 4x_3 + 4x_4$ . If this minimum is less than 685, then we have found a vertex solution that satisfies the first  $i + 1$  equation. If not, then we determined that no such solution exists.

**No Small Local Maximum:** To prove that the above algorithm eventually finds a global maximum, we must prove that it will not get stuck in a small local maximum.

**Convex:** The intuition is straightforward. Because the bounded polyhedral is the intersection of straight cuts, it is what we call *convex*. More formally, this means that the line between any two points in the polyhedral are also in the polyhedral. This means that there cannot be two local maximum points, because between these two hills there would need to be a valley and a line between two points across this valley would be outside the polyhedral.

**The Primal-Dual Method:** As done with the network flow algorithm, the primal dual method formally proves that a global maximum will be found. Given any linear program, defined by an optimization function and a set constraints, there is a way of forming its *dual* minimization linear program. Each solution to this dual acts as a roof or upper bound on how high the primal solution can be. Then each iteration either finds a better solution for the primal or providing a solution for the dual linear program with a matching value. This dual solution witnesses the fact no primal solution is bigger.

**Forming the Dual:** If the primal linear program is to maximize  $a \cdot x$  subject to  $Mx \leq b$ , then the dual is to minimize  $b^T \cdot y$  subject to  $M^T \cdot y \geq a^T$ . Where  $b^T$ ,  $M^T$ , and  $a$  are the transposes formed by flipping the vector or matrix along the diagonal. The dual of the concrete example given above is

minimize

$$258 + 721y_2 + 524y_3 + 411y_4 + 685y_5$$

subject to

$$3y_1 + 6y_2 + 2y_3 + 3y_4 + 4y_5 \geq 7$$

$$7y_1 + 3y_2 + 1y_3 + 6y_4 - 8y_5 \geq -6$$

$$2y_1 + 9y_2 + 5y_3 + 2y_4 - 4y_5 \geq 5$$

$$9y_1 - 6y_2 + 5y_3 + 3y_4 + 4y_5 \geq 7$$

The dual will have a variable for each constraint in the primal and a constraint for each of its variables. The coefficients of the objective function becomes the numbers on the right hand side of the inequalities and the numbers on the right hand side of the inequalities becomes the coefficients of the objective function. Finally, the maximize becomes a minimize. Another interesting thing is that the dual of the dual is the same as the original primal.

**Upper Bound:** We prove that the value of any solution to the primal linear program is at most the value of any solution to the dual linear program as flows. The value of the primal solution  $x$  is  $a \cdot x$ . The constraints  $M^T \cdot y \geq a^T$  can be turned around to give  $a \leq y^T \cdot M$ . This gives that  $a \cdot x \leq y^T \cdot M \cdot x$ . Using the constraints  $Mx \leq b$ , this is at most  $y^T \cdot b$ . This can be turned around to give  $b^T \cdot y$ , which is value of the dual solution  $y$ .

**Running Time:** The primal-dual hill climbing algorithm is guaranteed to find the optimal solution. In practice, it works quickly (though for my summer job, the computers would crank for hours.) However, there is no known hill climbing algorithm that is guaranteed to run in polynomial time.

There is another algorithm that solves this problem, called the *Ellipsoid Method*. Practically, it is not as fast, but theoretically it provably runs in polynomial time.

# Chapter 22

## Greedy Algorithms

Every two year old knows the greedy algorithm. In order to get what you want, just start grabbing what looks best.



### 22.1 Abstractions, Techniques and Theory

**Optimization Problems:** As mentioned already, *optimization problems* requires finding the best of an exponentially large set of solutions. (See Section 19.) A very select number of these problems can be solved using a greedy algorithm. Most of these have the following form.

**Instances:** An instance consists of a set of objects and a relationship between them. Think of the objects as being prizes that you must choose between.

**Solutions for Instance:** A solution requires the algorithm to make a choice about each of the objects in the instance. Sometimes, this choice is more complex, but usually it is simply whether or not to keep it. In this case, a solution is the subset of the objects that you have kept. The catch is that some subsets are not allowed because these objects conflict somehow with each other.

**Cost of Solution:** Each solution is assigned a cost (or measure of success). Often when a solution consists of a non-conflicting subset of the objects, this cost is the number of objects in the subset or the sum of the costs of its individual objects. Sometimes the cost is a more complex function.

**Goal:** Given an instance, the goal is to find one of the valid solutions for this instance with optimal (minimum or maximum as the case may be) cost. (The solution to be outputted might not be unique.)

### **The Brute Force Algorithm (Exponential Time):**

The brute force algorithm for an optimization problem considers each possible solution for the given instance, computes its cost, and outputs the cheapest. Because each instance has an exponential number of solutions, this algorithm takes exponential time.



**The Greedy Choice:** The greedy step is the first that would come to mind when designing an algorithm for a problem. Given the set of objects specified in the input instance, the greedy step chooses and commits to one of these objects because, according to some simple criteria, it seems to be the “best.” When proving that the algorithm works, we must be able to prove that this locally greedy choice does not have negative global consequences.

**The Game Show Example:** Suppose the instance specifies a set of prizes and an integer  $m$  and allows you to choose  $m$  of the prizes. The criteria, according to which some of these prizes appear to be “better” than others, may be its dollar price, the amount of joy it would bring you, how practical it is, or how much it would impress the neighbors. At first it seems obvious that you should choose your first prize using the greedy approach. However, some of these prizes conflict with each other and as is often the case in life there are compromises that need to be made. For example, if you take the pool, then your yard is too full to be able to take many of the other prizes. Or if you take the lion, then are many other animals that would not appreciate living together with it. As is also true in life, it is sometimes hard to look into the future and predict the ramifications of the choices made today.



**Making Change Example:** The goal of this optimization problem is to find the minimum number of quarters, dimes, nickels, and pennies that total to a given amount. The above format states that an instance consists of a set objects and a relationship between them. Here, the set is a huge pile of coins and the relationship is that the chosen coins must total to the given amount. The cost of a solution, which is to be minimized, is the number of coins in the solution.

**The Greedy Choice:** The coin that appear to be best to take is a quarter, because it makes the most progress towards making our required amount while only incurring a cost of one.

**A Valid Choice:** Before committing to a quarter, we must make sure that it does not conflict with the possibility of arriving at a valid solution. If the sum to be obtained happens to be less than \$0.25, then this quarter should be rejected, even though it appears at first to be best. On the other hand, if the amount is at least \$0.25, then we can commit to the quarter without invalidating the solution we are building.

**Leading to an Optimal Solution:** A much more difficult and subtle question is whether or not committing to a quarter leads us towards obtaining an optimal solution. In this case, it happens that it does, though this not at all obvious.

**Going Wrong:** Suppose that the previous Making Change Problem is generalized to include as part of the input the set of coin denominations available. This problem is identical to Integer-Knapsack Problem given in Section 24.5. The greedy algorithm does not work. For example, suppose we have 4, 3, and 1 cent coins. If the given amount is 6, than the optimal solution contains two 3 cent coins. One goes wrong by greedily committing to a 4 cent coin.

**Have Not Gone Wrong:** Committing to the pool, to the lion, or to the 4 cent coin in the previous examples, though they locally appear to be the “best” objects, do not lead to an optimal solution. However, for some problems and for some definitions of “best”, the greedy algorithm does work. Before committing to the seemingly “best” object, we need to prove that we do not go wrong by doing it.

**“The” Optimal Solution Contains the “Best” Object:** The first attempt at proving this might try to prove that for every set of objects that might be given as an instance, the “best” of these objects is definitely in its optimal solution. The problem with this is that there may be more than one optimal solution. It might not be the case that all of them contain the chosen object. For example, if all the objects were the same, then it would not matter which subset of objects were chosen.

**At Least One Optimal Solution Remaining:** Instead of requiring all optimal solutions to contain the “best” object, what needs to be proven is that at least one does. The effect of this is that though committing to the “best” object may eliminate the possibility of some of the optimal solutions, it does not eliminate all of them. There is the saying, “Do not burn your bridges behind you.” The message here is slightly different. It is okay to burn a few of your bridges as long as you do not burn all of them.

**The Second Step:** After the “best” object has been chosen and committed to, the algorithm must continue and choose the remaining objects for the solution. There are two different abstractions within which one can think about this process, iterative and recursive. Though the resulting algorithm is (usually) the same, having the different paradigms at your disposal can be helpful.

**Iterative:** In the iterative version, there is a main loop. Each iteration, the “best” is chosen from amongst the objects that have not yet been considered. The algorithm then commits to some choice about this object. Usually, this involves deciding whether to commit to putting this chosen object in the solution or to commit to rejecting it.

**A Valid Choice:** The most common reason for rejecting an object is that it conflicts with the objects committed to previously. Another reason for rejecting an object is that the object fills no requirements that are not already filled by the objects already committed to.

**Cannot Predict the Future:** At each step, the choice that is made can depend on the choices that were made in the past, but it cannot depend on the choices that will be made in the future. Because of this, no backtracking is required.

**Making Change Example:** The greedy algorithm for finding the minimum number of coins summing to a given amount is as follows. Commit to quarters until the next quarter increases your current sum above the required amount. Then reject the remaining quarters. Then do the same with the dimes, the nickels, and pennies.

**Recursive:** A recursive greedy algorithm makes a greedy first choice and then recurses once or twice in order to solve the remaining subinstance.

**Making Change Example:** After committing to a quarter, we could subtract \$0.25 from the required amount and ask a friend to find the minimum number of coins to make this new amount. Our solution, will be his solution plus our original quarter.

**Binary Search Tree Example:** The recursive version of a greedy algorithm is more useful when you need to recurse more than once. For example, suppose you want to construct a binary search tree for a set of keys that minimizes the total height of the tree, i.e., a balanced tree. The greedy algorithm will commit to the middle key being at the root. Then it will recurse once for the left subtree and once for the right.

To learn more about how to recurse after the greedy choice has been made see recursive backtracking algorithms in Chapter 23.

**Proof of Correctness:** Greedy algorithms themselves are very easy to understand and to code. If your intuition is that it should not work, then your intuition is correct. For most optimization search problems, all greedy algorithms tried do not work. By some miracle, however, for some problems there is a greedy algorithm that works. The proof that they work, however, is very subtle and difficult. As with all iterative algorithms, we prove that it works using loop invariants.

**A Formal Proof:** Section 8.5.2 proves that an iterative algorithm works if the loop invariant can be established and maintained and from it the postcondition can be proved. The loop invariant here is that the algorithm has not gone wrong, there is at least one optimal solution consistent with the choices made so far. This is established,  $\langle \text{pre} \rangle \rightarrow \langle LI \rangle$ , by noting that initially no choices have been made and hence all optimal solutions are consistent with these choices. The loop invariant is maintained,  $\langle LI' \rangle \wedge \neg \langle \text{exit} \rangle \wedge \text{code}_{\text{loop}} \rightarrow \langle LI'' \rangle$ , as follows. If it is true when at the top of the loop, then let  $optS_{LI}$  denote one such solution.  $\text{code}_{\text{loop}}$  during the next iteration either commits or rejects the next best object. The proof describes a method for massaging  $optS_{LI}$  into  $optS_{\text{ours}}$  and proves that this is a valid solution, is consistent both with the choices made previously by the algorithm and with this new choice, and is optimal. The existence of such a  $optS_{\text{ours}}$  proves that the loop invariant has been maintained. The last step is to prove that in the end, the algorithm has a concrete optimal solution,  $\langle LI \rangle \wedge \langle \text{exit} \rangle \rightarrow \langle \text{post} \rangle$ . Progress is made each step by committing to or rejecting another object. When each object has been considered the algorithm exits. These choices specify a solution. The loop invariant states there is an optimal solution consistent with these choices. Hence, this solution obtained must be optimal.

We will now redo the proof in a more intuitive, fun, and detailed way.

**The Loop Invariant:** The loop invariant maintained is that we have not gone wrong. There is at least one optimal solution consistent with the choices made so far, i.e., containing the objects committed to so far and not containing the objects rejected so far.

**Three Players:** To help understand this proof, we will tell a story involving three characters: the algorithm, the prover, and a fairy god mother.



**The Algorithm:** Each iteration, the algorithm chooses the “best” object from amongst those not considered so far and either commits to it or rejects it.

**The Prover:** The prover’s task is to prove that the loop invariant is maintained. Having a separate prover emphasizes that fact that his actions are not a part of the algorithm and hence do not need to be coded or executed.

**The Fairy God Mother:** Instead of the prover pretending that he has and is manipulating a hypothetical optimal solution  $optS_{LI}$ , he can pretend that he has a fairy god mother who holds and manipulates one for him. We say that this solution *witnesses* the fact such a solution exists. Having a separate fairy god mother emphasizes that neither the algorithm nor the prover actually know the solution.

**Initially (i.e.,  $\langle pre \rangle \rightarrow \langle LI \rangle$ ):** Initially, the algorithm has made no choices, neither committing to nor rejecting any objects. The prover then establishes the loop invariant as follows. Assuming that there is at least one legal solution, he knows that there must be an optimal solution. He goes on to note that this optimal solution by default is consistent with the choices made so far, because no choices have been made so far. Knowing that such a solution exists, the prover kindly asks his fairy god mother to find one. She being all powerful has no problem doing this. If there are more than one equally good optimal solutions, then she chooses one arbitrarily.

**Maintaining the Loop Invariant (i.e.,  $\langle LI' \rangle \& \text{ not } \langle exit \rangle \& \text{ code}_{loop} \rightarrow \langle LI'' \rangle$ ):**  
Now consider an arbitrary iteration.

**What We Know:** At the beginning of this iteration, the algorithm has a set *Commit* of objects committed to so far and the set of objects rejected so far. The prover knows that the loop invariant is true, i.e., that there is at least one optimal solution consistent with these choices made so far, however, he does not know one. Witnessing that there is one, the fairy god mother is holding one such optimal solution. We will use  $optS_{LI}$  to denote the solution that she holds. In addition to containing those objects in *Commit* and not those in *Reject*, this solution may contain objects that the algorithm has not considered yet.

**Taking a Step:** During the iteration, the algorithm proceeds to choose the “best” object from amongst those not considered so far and either commits to it or rejects

it. In order to prove that the loop invariant has been maintained, the prover must prove that there is at least one optimal solution consistent with both the choices made previously and with this new choice. He is going to accomplish this by getting his fairy god mother to witness this fact by switching to such an optimal solution.

**Weakness in Communication:** It would be great if the prover could simply ask the fairy god mother whether such a solution exists. However, he cannot ask her to find such a solution if he is not already confident that it exists, because he does not want to ask her to do anything that is impossible.

**Massage Instructions:** The prover accomplishes his task by giving his fairy god mother detailed instructions. He starts by saying, “If it happens to be the case that the optimal solution that you hold is consistent with this new choice that was made, then we are done, because this will witness the fact that there is at least one optimal solution consistent with both the choices made previously and with this new choice.” “Otherwise,” he says, “you must massage (modify) the optimal solution that you have in the following ways.” The fairy god mother follows the detailed instructions that he gives her, but gives him no feedback as to how they go. We will use  $optS_{ours}$  to denote what she constructs.

**Making Change Example:** If the remaining amount required is at least \$0.25, then the algorithm commits to another quarter. Excluding the committed to coins, the fairy god mother’s optimal solution  $optS_{LI}$  must be making up the same remaining amount. This amount must contain either an additional quarter; three dimes; two dimes and a nickel; one dime and three nickels; five nickels; or combinations with at least five pennies. The prover tells her to replace the three dimes with the newly committed to quarter and a nickel and the other options with just the quarter. If the algorithm, on the other hand, rejects the next (and later all remaining) quarters because the remaining amount required is less than \$0.25, the prover is confident that optimal solution held by his fairy god mother cannot contain additional quarters either.

**Proving That She Has A Witness:** It is the job of the prover to prove that the thing  $optS_{ours}$  that his fairy god mother now holds is a valid, consistent, and optimal solution.

**Proving A Valid Solution:** Because he knows that what she had been holding,  $optS_{LI}$ , at the beginning of the iteration was a valid solution, he knows that the objects in it did not conflict in any way. Hence, all he needs to do is to prove that he did not introduce any conflicts that he did not fix.

**Making Change Example:** The prover was careful that the changes he made did not change the total amount that she was holding.

**Proving Consistency:** He must also prove that the solution she is now holding is consistent with both the choices made previously by the algorithm and with this new choice. Because he knows that what she had been holding was consistent with the previous choices, he only needs to prove that he modified it to be consistent with the new choices without messing this earlier fact.

**Making Change Example:** Though the prover may have removed some of the coins that the algorithm has not considered yet, he was sure not to have her remove any of the previously committed to coins. He also managed to add the newly committed to quarter.

**Proving Optimal:** One might think that proving that the solution  $optS_{ours}$  is optimal would be hard, given that we do not even know the cost of an optimal solution. However, the prover can be assured that it is optimal as long as its cost is the same as the optimal solution,  $optS_{LI}$ , from which it was derived. If there is case in which prover manages to improve the solution, then this contradicts the fact that  $optS_{LI}$  is optimal. This contradiction only proves that such a case will not occur. However, the prover does not need to concern himself with this problem.

**Making Change Example:** Each change that the prover instructs his fairy god mother to make either keeps the number of coins the same or decreases the number. Hence, because  $optS_{LI}$  is optimal,  $optS_{ours}$  is as well.

This completes the prover's proof that his fairy god mother now has an optimal solution consistent with both the previous choices and with the latest choice. This witnesses the fact that such a solution exists.

This proves that the loop invariant has been maintained.

**Continuing:** This completes everybody's requirements for this iteration. This is all repeated over and over again. Each iteration, the algorithm commits to more about the solution and the fairy god mother's solution is changed to be consistent with these commitments.

**Exiting Loop (i.e.,  $\langle LI \rangle \& \langle exit \rangle \rightarrow \langle post \rangle$ ):** After the algorithm has considered every object in the instance and each has either been committed to or rejected, the algorithm exits. We still know that the loop invariant is true. Hence, the prover knows that there is an optimal schedule  $optS_{LI}$  consistent with all of these choices. Previously, this optimal solution was only imaginary. However, now we concretely know that this imagined solution consists of those objects committed to. Hence, the algorithm can return this set as the solution.

**Running Time:** Greedy algorithms are very fast because they take only a small amount of time per object in the instance.

**Exercise 22.1.1** Above we proved that the greedy algorithm does not work for the making change problem, when the denominations of the coins are 4, 3, and 1 cent, but it does work when the denominations are 25, 10, 5, and 1. Does it work when the denominations are 25, 10, and 1, with no nickels?

**Exercise 22.1.2** Suppose the coin denominations are  $c_1 > c_2 > \dots > c_r$  (order taken by the greedy algorithm). An interesting problem is determining whether the greedy algorithm works. A complete answer to this question is too hard. However, what restrictions on the coin denominations are sufficient to ensure that the greedy algorithm works?

- Suppose for each  $i$ , each coin  $c_i$  is an integer multiple of the next smaller  $c_{i+1}$ , e.g. 120, 60, 12, 3, 1. If this is true do we know that greedy algorithm works? If it is not true, do we know that the greedy algorithm does not work?
- Suppose for each  $i$ , each coin is more than twice the previous, i.e.  $c_i \geq 2c_{i+1}$ , do we know that greedy algorithm works? If this is not true do we know that the greedy algorithm does not work?
- Other interesting characteristics?

**Fixed vs Adaptive Priority:** Iterative greedy algorithms come in two flavors, *fixed priority* and *adaptive priority*.

**Fixed Priority:** A fixed priority greedy algorithm begins by sorting the objects in the input instance from best to worst according to a fixed greedy criteria. For example, it might sort the objects based on the cost of the object or the arrival time of the object. The algorithm then considers the objects one at a time in this order.

**Adaptive Priority:** In an adaptive priority greedy algorithm, the greedy criteria is not fixed, but depends on which objects have been committed to so far. At each step, the next “best” object is chosen according to the current greedy criteria. Blindly searching the remaining list of objects each iteration for the next best object would be too time consuming. So would re-sorting the objects each iteration according to the new greedy criteria. A more efficient implementation uses a priority queue to hold the remaining objects prioritized according to the current greedy criteria. This can be implemented using a Heap. (See Section 16.4.)

**Code:**

```
algorithm AdaptiveGreedy( set of objects )
```

*⟨pre-cond⟩:* The input consists of a set of objects.

*⟨post-cond⟩:* The output consists of an optimal subset of them.

begin

    Put objects in a Priority Queue according to the initial greedy criteria

    Commit =  $\emptyset$  % set of objects previously committed to

    loop   *⟨loop-invariant⟩:* There is at least one optimal solution consistent with

        the choices made so far

        exit when the priority queue is empty

        Remove “best” object from priority queue

        If this object does not conflict with those in Commit and is needed then

            Add object to Commit

        end if

        Update the priority queue to reflect the new greedy criteria.

            This is done by changing the priorities of the objects effected.

    end loop

    return (Commit)

end algorithm

**Example:** Dijkstra’s shortest weighted path algorithm can be considered to be a greedy algorithm with an adaptive priority criteria. See Section 20.3. It chooses the next edge to include in the optimal *shortest-weighted-paths tree* based on which node currently seems to be the closest to  $s$ . Those yet to be chosen are organized in a priority queue. Even Breadth-First and Depth-First Search can be considered to be adaptive greedy algorithms. In fact, they very closely resemble Prim’s Minimal Spanning Tree algorithm, Section 22.2.3, in how a tree is grown from a source node. They are adaptive because as the algorithm proceeds, the set from which the next edge is chosen changes.

## 22.2 Examples of Greedy Algorithms

This completes the presentation of the general techniques and the theory behind greedy algorithms. We now will provide a few examples.

### 22.2.1 A Fixed Priority Greedy Algorithm for the Job/Event Scheduling Problem

Suppose that many people want to use your conference room for events and you must schedule as many of these as possible. (The version in which some events are given a higher priority is considered in Section 24.4.)

**The Event Scheduling Problem:**

**Instances:** An instance is  $\langle\langle s_1, f_1 \rangle, \langle s_2, f_2 \rangle, \dots, \langle s_n, f_n \rangle\rangle$ , where  $0 \leq s_i \leq f_i$  are the starting and finishing times for the  $i^{th}$  event.

**Solutions:** A solution for an instance is a schedule  $S$ . This consists of a subset  $S \subseteq [1..n]$  of the events that don't conflict by overlapping in time.

**Cost of Solution:** The cost (or success)  $C(S)$  of a solution  $S$  is the number of events scheduled, i.e.,  $|S|$ .

**Goal:** Given a set of events, the goal of the algorithm is to find an *optimal solution*, i.e., one that maximizes the number of events scheduled.

**Possible Criteria for Defining “Best”:**

**The Shortest Event  $f_i - s_i$ :** It seems that it would be best to schedule short events first because they increase the number of events scheduled without booking the room for a long period of time. This greedy approach does not work.

**Counterexample:** Suppose that the following lines indicate the starting and completing times of three events to schedule.

We would be going wrong to schedule the short event in the middle, because the only optimal schedule does not include it.

**The Earliest Starting Time  $s_i$  or the Latest Finishing Time  $f_i$ :** First come first serve, which is a common scheduling algorithm, does not work either.

**Counterexample:**

The long event is both the earliest and the latest. Committing to scheduling it would be a mistake.

**Event Conflicting with the Fewest Other Events:** Scheduling an event that conflicts with other events prevents you from scheduling these events. Hence a reasonable criteria would be to first schedule the event with the fewest conflicts.

**Counterexample:** In the following example, the middle event would be committed to first. This eliminates the possibility of scheduling four events.



**Earliest Finishing Time  $f_i$ :** This criteria may seem a little odd at first, but it too makes sense. It says to schedule the event who will free up your room for someone else as soon as possible. We will see that this criteria works for every set of events.

**Exercise 22.2.1** See how it works on the above three examples and on the example from Figure 22.1.

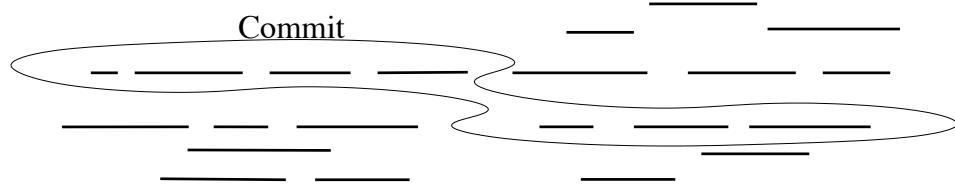


Figure 22.1: A set of events and those committed to by the Earliest Finishing Time First Greedy algorithm

**Code:** The resulting greedy algorithm for the Event Scheduling problem is as follows:

**algorithm** *Scheduling* ( $\langle\langle s_1, f_1 \rangle, \langle s_2, f_2 \rangle, \dots, \langle s_n, f_n \rangle\rangle$ )

***⟨pre-cond⟩:*** The input consists of a set of events.

***⟨post-cond⟩:*** The output consists of a schedule that maximizes the number of events scheduled.

begin

    Sort the events based on their finishing times  $f_i$

$Commit = \emptyset$  % The set of events committed to be in the schedule

    loop  $i = 1 \dots n$  % Consider the events in sorted order.

        if( event  $i$  does not conflict with an event in  $Commit$  ) then

$Commit = Commit \cup \{i\}$

        end loop

        return( $Commit$ )

    end algorithm

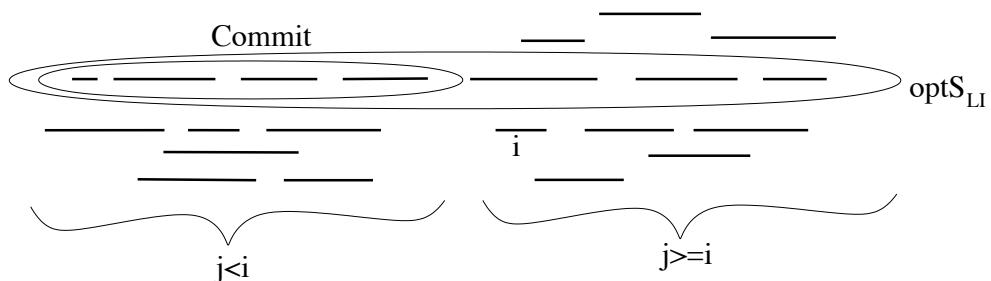


Figure 22.2: A set of events, those committed to at the current point in time, those rejected, those in the optimal solution assumed to exist, and the next event to be considered

**The Loop Invariant:** The loop invariant is that we have not gone wrong. There is at least one optimal solution consistent with the choices made so far, i.e., containing the objects committed to so far and not containing the objects rejected so far.

**Initial Code (i.e.,  $\langle \text{pre} \rangle \rightarrow \langle \text{LI} \rangle$ ):** Initially, no choices have been made and hence trivially all optimal solutions are consistent with these choices.

**Maintaining the Loop Invariant (i.e.,  $\langle \text{LI}' \rangle \& \text{ not } \langle \text{exit} \rangle \& \text{ code}_{\text{loop}} \rightarrow \langle \text{LI}'' \rangle$ ):**

**Hypothetical Optimal Solution:** Let  $\text{optS}_{\text{LI}}$  denote one of the hypothetical optimal schedules assumed to exist by the loop invariant.

**Algorithm's Actions:** If event  $i$  conflicts with an event in  $\text{Commit}$ , then the algorithm rejects it.  $\text{optS}_{\text{LI}}$  contains all the events in  $\text{Commit}$  and hence cannot contain event  $i$  either. Hence, the loop invariant is maintained. From here on, let us assume that  $i$  does not conflict with any event in  $\text{Commit}$  and hence will be added to it.

**Massaging Optimal Solutions:** If we are lucky, the schedule  $\text{optS}_{\text{LI}}$  already contains event  $i$ . In this case, we are done. Otherwise, we will massage the schedule  $\text{optS}_{\text{LI}}$  into another schedule  $\text{optS}_{\text{ours}}$  by adding  $i$  and removing any events that conflict with it. Note that this massaging is NOT part of the algorithm, as we do not actually have an optimal schedule yet.

**A Valid Solution:** Our massaged set  $\text{optS}_{\text{ours}}$  contains no conflicts, because  $\text{optS}_{\text{LI}}$  contained none and we were careful to introduce none.

**Consistent with Choices Made:**  $\text{optS}_{\text{LI}}$  was consistent with the previous choices. We added event  $i$  to make  $\text{optS}_{\text{ours}}$  consistent with this choices. Finally note that we did not remove any events from  $\text{Commit}$  because these do not conflict with event  $i$ .

**Optimal:** To prove that  $\text{optS}_{\text{ours}}$  has the optimal number of events in it, we need only to prove that it has at least as many as  $\text{optS}_{\text{LI}}$ . We added one event to the schedule. Hence, we must prove that we have not removed more than one. Let  $j$  denote a deleted event. Being in  $\text{optS}_{\text{LI}}$  and not in  $\text{Commit}$  it must be an event not yet considered. Because the events are sorted based on their finishing time,  $j > i$  implies that event  $j$  finishes after event  $i$  finishes, i.e.,  $f_j \geq f_i$ . If event  $j$  conflicts with event  $i$ , it follows that it also starts before it finishes, i.e.,  $s_j \leq f_i$ . (In Figure 22.2, there are three future events conflicting with  $i$ .) Combining  $f_j \geq f_i$  and  $s_j \leq f_i$ , gives that such an event  $j$  is running at the finishing time  $f_i$  of event  $i$ . Hence, any two such events  $j$  conflict with each other. Therefore, they cannot both be in the schedule  $\text{optS}_{\text{LI}}$  because it contains no conflicts.

**Loop Invariant Has Been Maintained:** In conclusion, we constructed a valid optimal schedule  $\text{optS}_{\text{ours}}$  that contains the events in  $\text{Commit} \cup \{i\}$  and no rejected events. This proves that the loop invariant is maintained.

**Exiting Loop (i.e.,  $\langle \text{LI} \rangle \& \langle \text{exit} \rangle \rightarrow \langle \text{post} \rangle$ ):** By LI, there is an optimal schedule  $\text{optS}_{\text{LI}}$  containing the events in  $\text{Commit}$  and not containing the previous events not in  $\text{Commit}$ . Because all events are previous events, it follows that  $\text{Commit} = \text{optS}_{\text{LI}}$  is in an optimal schedule for our instance.

**Running Time:** The loop is iterated once for each of the  $n$  events. The only work is determining whether event  $i$  conflicts with a event within  $\text{Commit}$ . Because of the ordering of the events, event  $i$  finishes after all the events in  $\text{Commit}$ . Hence, it conflicts with an event in  $\text{Commit}$

if and only if it starts before the last finishing time of an event in it. It is easy to remember this last finishing time, because it is simply the finishing time of the last event to be added to *Commit*. Hence, the main loop runs in  $\Theta(n)$  time. The total time of the algorithm then is dominated by the time to sort the events.

### 22.2.2 An Adaptive Priority Greedy Algorithm for the Interval Cover Problem

#### The Interval Cover Problem:

**Instances:** An instance consists a set of points  $P$  and a set of intervals  $I$  on the real line.  
An interval consists of a starting and a finishing time  $(s_i, f_i)$ .

**Solutions:** A solution for an instance is a subset  $S$  of the intervals that covers all the points.  
It is fine if the intervals overlap.

**Cost of Solution:** The cost  $C(S)$  of a solution  $S$  is the number of intervals required, i.e.,  $|S|$ . Having longer or shorter intervals does not matter. Covering the points more than once does not matter. Covering parts of the line without points does not matter. Only the number of intervals matters.

**Goal:** The goal of the algorithm is to find an *optimal cover*, i.e., a subset of the intervals that covers all the points and that contains the minimum number of intervals.

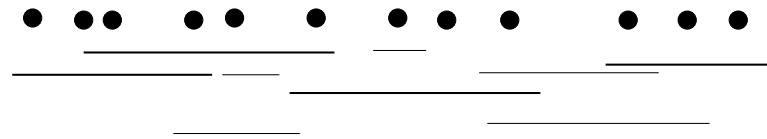


Figure 22.3: An example instance of the interval cover problem is given. The intervals in one optimal solution are highlighted.

**Exercise 22.2.2** (*See solution in Section V*) Consider the greedy criteria that selects the interval that covers the largest number of uncovered points. Does this work? If not give a counter example.

**The Adaptive Greedy Criteria:** The algorithm sorts the points and covers them in order from left to right. If the intervals committed to so far, i.e., those in *Commit*, cover all of the points in  $P$ , then the algorithm stops. Otherwise, let  $P_i$  denote the left most point in  $P$  that is not covered by *Commit*. The next interval committed to must cover this next uncovered point,  $P_i$ . Of the intervals that start to the left of the point, the algorithm greedily takes the one that extends as far to the right as possible. The hope in doing so is that the chosen interval, in addition to covering  $P_i$ , will cover as many other points as possible. Let  $I_j$  denote this interval. If there is no such interval  $I_j$  or it does not extend to the right far enough to cover the point  $P_i$ , then no interval covers this point and the algorithm reports that no subset of the intervals covers all of the points. Otherwise, the algorithm commits to this interval by adding it *Commit*. Note that this greedy criteria with which to select the next interval changes as the point  $P_i$  to be covered changes.

**The Loop Invariant:** The loop invariant is that we have not gone wrong. There is at least one optimal solution consistent with the choices made so far, i.e., containing the objects committed to so far and not containing the objects rejected so far.

**Maintaining the Loop Invariant (i.e.,  $\langle LI' \rangle$  & not  $\langle exit \rangle$  &  $code_{loop} \rightarrow \langle LI'' \rangle$ ):**

Assume that we are at the top of the loop and that the loop invariant is true, i.e., there exists an optimal cover that contains all of the intervals in  $Commit$ . Let  $optS_{LI}$  denote such a cover that is assumed to exist. If we are lucky and  $optS_{LI}$  already contains the interval  $I_j$  being committed to this iteration, then we automatically know that there exists an optimal cover that contains all of the intervals in  $Commit \cup \{I_j\}$  and hence the loop invariant has been maintained. If on the other hand, the interval  $I_j$  being committed to is not in  $optS_{LI}$ , then we must massage this optimal solution into another optimal solution that does contain it.

**Massaging  $optS_{LI}$  into  $optS_{ours}$ :** The optimal solution  $optS_{LI}$  must cover point  $P_i$ . Let

$I_{j'}$  denote one of the intervals in  $optS_{LI}$  that covers  $P_i$ . Our solution  $optS_{ours}$  is the same as  $optS_{LI}$  except  $I_{j'}$  is removed and  $I_j$  is added. We know that  $I_{j'}$  is not in  $Commit$ , because the point  $P_i$  is not covered by  $Commit$ . Hence as constructed  $optS_{ours}$  contains all the intervals in  $Commit \cup \{I_j\}$ .

**$optS_{ours}$  is an Optimal Solution:** Because  $optS_{LI}$  is an optimal cover, we can prove that  $optS_{ours}$  is an optimal cover, simply by proving that it covers all of the points covered by  $optS_{LI}$  and that it contains the same number of intervals. (The latter is trivial.)

The algorithm considered the point  $P_i$  because it is the left most uncovered point. It follows that the intervals in  $Commit$  cover all the points to the left of point  $P_i$ .

The interval  $I_{j'}$ , because it covers point  $P_i$ , must start to the left of point  $P_i$ . Hence, the algorithm must have considered it when it chose  $I_j$ .  $I_j$ , being the interval that extends as far to the right as possible of those that start to the left of point  $P_i$ , must extend at least as far to the right as  $I_{j'}$  and hence  $I_j$  covers as many points to the right as  $I_{j'}$  covers. It follows that  $optS_{ours}$  covers all of the points covered by  $optS_{LI}$ .

Because  $optS_{ours}$  is an optimal solution containing  $Commit \cup \{I_j\}$ , we have proved such a solution exists. Hence, the loop invariant has been maintained.

**Maintaining the Greedy Criteria:** As the point  $P_i$  to be covered changes, the greedy criteria according to which the next interval is chosen changes. Blindly searching for the interval that is best according to the current greedy criteria would be too time consuming. The following data structures help to make the algorithm more efficient.

**An Event Queue:** The progress of the algorithm can be viewed as an *event marker* moving along the real line. An *event* in the algorithm occurs when this marker either reaches the start of an interval or a point to be covered. This is implemented not with a marker, but by *event queue*. The queue is constructed initially by sorting the intervals according to their start time, the points according to their position, and merging these two lists together. The algorithm removes and processes these events one at a time.

**Additional Loop Invariants:** The following additional loop invariants relate the current position event marker with the current greedy criteria.

**LI1 Points Covered:** All the points to the left of the event marker have been covered by the intervals in  $Commit$ .

**LI2 The Priority Queue:** A priority queue contains all intervals (except possibly those in  $Commit$ ) that start to the left of the event marker. The priority according to which they are organized is how far  $f_j$  to the right the interval extends. This priority queue can be implemented using a Heap. (See Section 16.4.)

**LI3 Last Place Covered:** A variable  $last$  indicates the right most place covered by an interval in  $Commit$ .

### Maintaining the Additional Loop Invariants:

**A Start Interval Event:** When the event marker passes the starting time  $s_j$  of an interval  $I_j$ , this interval from then on will start to the left of the event marker and hence is added to the priority queue with its priority being its finishing time  $f_j$ .

**An End Interval Event:** When the event marker passes the finishing time  $f_j$  of an interval  $I_j$ , we learn that this interval will not be able to cover future points  $P_i$ . Though the algorithm no longer wants to consider this interval, the algorithm will be lazy and leave it in the priority queue. Its priority will be lower than those actually covering  $P_i$ . There being nothing useful to do, the algorithm does not consider these events.

**A Point Event:** When the event marker reaches a point  $P_i$  in  $P$ , the algorithm uses  $last \geq P_i$  to check whether the point is already covered by an interval in  $Commit$ . If it is covered, then nothing needs to be done. If not, then LI1 assures us that this point is the left most uncovered point. LI2 ensures that the priority queue is organizing the intervals according to the current greedy criteria, namely it contains all intervals that start to the left of the point  $P_i$  sorted according to how far to the right the interval extends. Let  $I_j$  denote the highest priority interval in the priority queue. Assuming that it covers  $P_i$ , the algorithm commits to it. As well, if it covers  $P_i$ , then it must extend further to the right than other intervals in  $Commit$  and hence  $last$  is updated to  $f_j$ . (The algorithm can either remove the interval  $I_j$  committed to from the priority queue or not. It will not extend far enough to the right to cover the next uncovered point, and hence its priority will be low in the queue.)

### Code:

**algorithm** *IntervalPointCover* ( $P, I$ )

***⟨pre-cond⟩:***  $P$  is a set of points and  $I$  is a set of intervals on a line.

***⟨post-cond⟩:*** The output consists of the smallest set of intervals that covers all of the points.

begin

```

Sort  $P = \{P_1, \dots, P_n\}$  in ascending order of  $p_i$ 's.
Sort  $I = \{\langle s_1, f_1 \rangle, \dots, \langle s_m, f_m \rangle\}$  in ascending order of  $s_j$ 's.
Events = Merge( $P, I$ )      % sorted in ascending order
consideredI =  $\emptyset$         % the priority queue of intervals being considered
Commit =  $\emptyset$             % solution set: covering subset of intervals
last =  $-\infty$              % rightmost point covered by intervals in Commit
for each event  $e \in Events$ , in ascending order do
    if( $e = \langle s_j, f_j \rangle$ ) then
        Insert interval  $\langle s_j, f_j \rangle$  into the priority queue  $consideredI$  with priority  $f_j$ 
    else ( $e = P_i$ )
        if( $P_i > last$ ) then                      %  $P_i$  is not covered by Commit
             $\langle s_j, f_j \rangle = ExtractMax(consideredI)$  %  $f_j$  is max in  $consideredI$ 
            last =  $P_i$ 
    end if
end for

```

```

if(consideredI was empty or  $P_i > f_j$ ) then
    return ( $P_i$  cannot be covered)
else
     $Commit = Commit \cup \{j\}$ 
     $last = f_j$ 
end if
end if
end if
end for
return ( $Commit$ )
end algorithm

```

**Running Time:** The initial sorting takes  $\mathcal{O}((n + m) \log(n + m))$  time. The main loop iterates  $n + m$  times, once per event. Since  $H$  contains a subset of  $I$ , the priority queue operations *Insert* and *ExtractMax* each takes  $\mathcal{O}(\log m)$  time. The remaining operations of the loop take  $\mathcal{O}(1)$  time per iteration. Hence, the loop takes a total of  $\mathcal{O}((n + m) \log m)$  time. Therefore, the running time of the algorithm is  $\mathcal{O}((n + m) \log(n + m))$ .

### 22.2.3 The Minimum-Spanning-Tree Problem

Suppose that you are building network of computers. You need to decide between which pairs of computers to run a communication line. You want all of the computers to be connected via the network. You want to do it in a way that minimizes your costs. This is an example of the Minimum Spanning Tree Problem.

**Definitions:** Consider a subset  $S$  of the edges of an undirected graph  $G$ .

**A Tree:**  $S$  is said to be *tree* if it contains no cycles and is connected.

**Spanning Set:**  $S$  is said to *span* the graph iff every pair nodes is connected by a path through edges in  $S$ .

**Spanning Tree:**  $S$  is said to be a *spanning tree* of  $G$  iff it is a tree that spans the graph. Note cycles will cause redundant paths between pairs of nodes.

**Minimal Spanning Tree:**  $S$  is said to be a *minimal spanning tree* of  $G$  iff it is a spanning tree with minimal total edge weight.

**Spanning Forest:** If the graph  $G$  is not connected then it cannot have a spanning tree.  $S$  is said to be a *spanning forest* of  $G$  iff it is a collection of trees that spans each of the connected components of the graph. In other words, pairs of nodes that are connected by a path in  $G$  are still connected by a path when considering only the edges in  $S$ .

#### The Minimum Spanning Tree (Forest) Problem:

**Instances:** An instance consists of an undirected graph  $G$ . Each edge  $\{u, v\}$  is labeled with a real-valued (possibly negative) weight  $w_{\{u,v\}}$ .

**Solutions:** A solution for an instance is a spanning tree (forest)  $S$  of the graph  $G$ .

**Cost of Solution:** The cost  $C(S)$  of a solution  $S$  is sum of its edge weights.

**Goal:** The goal of the algorithm is to find a minimum spanning tree (forest) of  $G$ .

## Possible Criteria for Defining “Best”:

**Cheapest Edge (Kruskal’s Algorithm):** The obvious greedy algorithm simply commits to the cheapest edge that does not create a cycle with the edges committed to already.

**Code:** The resulting greedy algorithm is as follows.

```

algorithm KruskalMST ( $G$ )
⟨pre-cond⟩:  $G$  is an undirected graph.
⟨post-cond⟩: The output consists of a minimal spanning tree.
begin
    Sort the edges based on their weights  $w_{\{u,v\}}$ .
    Commit =  $\emptyset$  % The set of edges committed to
    loop  $i = 1 \dots m$  % Consider the edges in sorted order.
        if( edge  $i$  does not create a cycle with the edges in Commit ) then
            Commit = Commit  $\cup \{i\}$ 
        end loop
        return(Commit)
    end algorithm

```

**Checking For a Cycle:** One task that this algorithm must be able to do quickly is to determine whether the new edge  $i$  creates a cycle with the edges in *Commit*. As a task in itself, this would take a while. However, if we maintain an extra data structure, this task can be done very quickly.

**Connected Components of Commit:** We partition the nodes in the graph into sets so that between any two nodes in the same set, *Commit* provides a path between them and between any two nodes in the different sets, *Commit* does not connect them. These sets are referred to as *components* of the subgraph induced by the edges in *Commit*. The algorithm can determine whether the new edge  $i$  creates a cycle with the edges in *Commit* by checking whether the endpoints of the edge  $i = \{u, v\}$  are contained in the same component. The required operations on components are handled by the following *Union-Find* data structure.

**Union-Find Set System:** This data structure, created for this purpose, maintains a number of disjoint sets of elements and allows three operations: 1) *Makeset(v)* which creates a new set containing the specified element  $v$ ; 2) *Find(v)* which determines the “name” of the set containing a specified element; and 3) *Union( $u, v$ )* which merges the sets containing the specified elements  $u$  and  $v$ . On average, for all practical purposes, each of these operations can be completed in a constant amount of time. See Section 10.5.

**Code:** The union-find data structure is integrated into the MST algorithm as follows.

```

algorithm KruskalMST ( $G$ )
⟨pre-cond⟩:  $G$  is an undirected graph.
⟨post-cond⟩: The output consists of a minimal spanning tree.
begin
    Sort the edges based on their weights  $w_{\{u,v\}}$ .
    Commit =  $\emptyset$  % The set of edges committed to
    for each  $v$ ,

```

```

% With no edges in Commit, each node is in a component by itself.
MakeSet(v)
end for
loop  $i = 1 \dots m$  % Consider the edges in sorted order.
     $u$  and  $v$  are end points of edge  $i$ .
    if( Find( $u$ )  $\neq$  Find( $v$ ) ) then
        % The end points of edge  $i$  are in different components and
        % hence do not create a cycle with edges in Commit.
        Commit = Commit  $\cup \{i\}$ 
        Union( $u, v$ ) % Edge  $i$  connects the two components, hence
                           % they are merged into one component.
    end if
end loop
return(Commit)
end algorithm

```

**Running Time:** The initial sorting takes  $\mathcal{O}((m) \log(m))$  time when  $G$  has  $m$  edges.

The main loop iterates  $m$  times, once per edge. Checking for a cycle takes time  $\alpha(n) \leq 4$  on average. Therefore, the running time of the algorithm is  $\mathcal{O}(m \log m)$ .

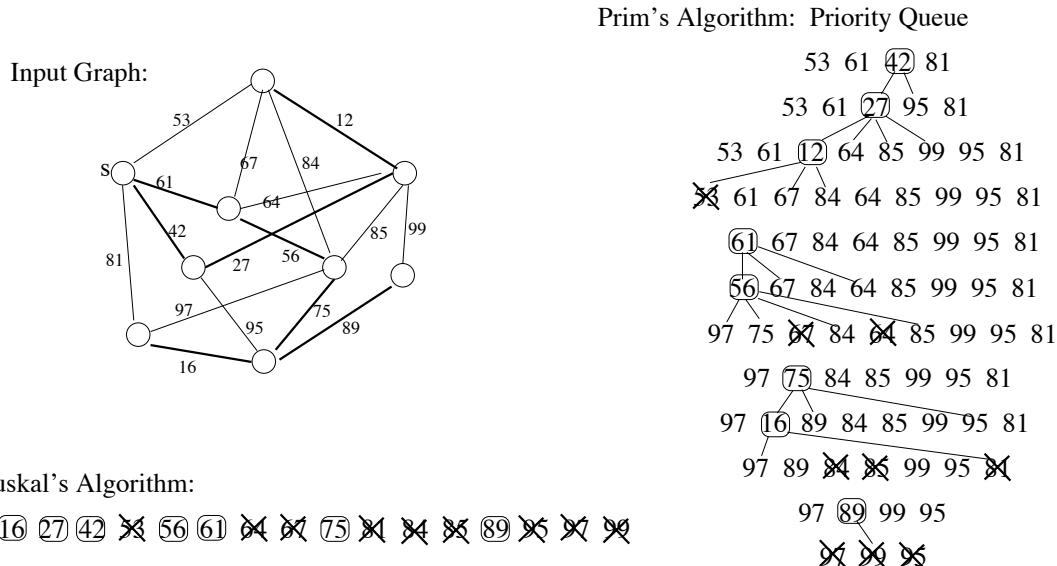


Figure 22.4: Both Kruskal's and Prim's algorithms are run on the given graph. For Kruskal's algorithm the sorted order of the edges is shown. For Prim's algorithm the running contents of the priority queue is shown (edges are in no particular order). Each iteration, the best edge is considered. If it does not create a cycle, it is add to the MST. This is shown by circling the edge weight and darkening the graph edge. For Prim's algorithm, the lines out of the circles indicate how the priority queue is updated. If the best edge does create a cycle, then the edge weight is Xed.

**Cheapest Connected Edge (Prim's Algorithm):** The following greedy algorithm expands out a tree of edges from a source node as done in the generic search algorithm of Section 20.1. Each iteration it commits to the cheapest edge of those that expand this tree, i.e., cheapest from amongst those edges that are connected to the tree *Commit*

and yet does not create a cycle with it.

**Advantage:** If you are, for example, trying to find a MST of the world wide web, then you may not know about an edge until you have expanded out to it.

**Adaptive:** Note that this is an adaptive greedy algorithm, because which edges are considered change as the algorithm proceeds. A priority queue is maintained with the allowed edges with priorities given by their weights  $w_{\{u,v\}}$ . Because the allowed edges are those that are connected to the tree *Commit*, when a new edge  $i$  is added to *Commit*, the edges connected to  $i$  are added to the queue.

**Code:**

```

algorithm PrimMST ( $G$ )
  <pre-cond>:  $G$  is an undirected graph.
  <post-cond>: The output consists of a minimal spanning tree.
  begin
    Let  $s$  be the start node in  $G$ .
     $Commit = \emptyset$  % The set of edges committed to
     $Queue =$  edges adjacent to  $s$  % Priority Queue
    Loop until  $Queue = \emptyset$ 
       $i =$  cheapest edge in  $Queue$ 
      if( edge  $i$  does not create a cycle with the edges in  $Commit$  ) then
         $Commit = Commit \cup \{i\}$ 
        Add to  $Queue$  edges adjacent to edge  $i$  that have not been added before
      end if
    end loop
    return( $Commit$ )
  end algorithm

```

**Checking For a Cycle:** Because one tree is grown, like in the generic search algorithm of Section 20.1, one end of the edge  $i$  will have been found already. It will create a cycle iff the other end has also been found already.

**Running Time:** The main loop iterates  $m$  times, once per edge. The priority queue operations *Insert* and *ExtractMax* each takes  $\mathcal{O}(\log m)$  time. Therefore, the running time of the algorithm is  $\mathcal{O}(m \log m)$ .

**A More General Algorithm:** When designing an algorithm it is best to leave as many implementation details unspecified as possible. One reason is that this gives more freedom to anyone who may want to implement or to modify your algorithm. Another reason is that it provides better intuition as to why the algorithm works. The following is a greedy criteria that is quite general.

**Cheapest Connected Edge of Some Component:** Partition the nodes of the graph  $G$  into connected *components* of nodes that are reachable from each other only through the edges in *Commit*. Nodes with no adjacent edges will be in a component by themselves. Each iteration, the algorithm is free to choose however it likes one of these components. Denote this component with  $C$ . (If it prefers  $C$  can be the union of a number of different components.) Then the algorithm greedily commits to the cheapest edge of those that expand this component, i.e., cheapest from amongst those edges that are connected to the component and yet do not create a cycle with it. When ever it likes, the algorithm also has the freedom to throw away uncommitted edges that create a cycle with the edges in *Commit*.

**Generalizing:** This greedy criteria is general enough to include both Kruskal's and Prim's algorithms. Therefore, if we prove that this greedy algorithm works no matter how it is implemented, then we automatically prove that both Kruskal's and Prim's algorithms work.

**Cheapest Edge (Kruskal's Algorithm):** This general algorithm may choose the component that is connected to the cheapest uncommitted edge that does not create a cycle. Then when it chooses the cheapest edge out of this component, it gets the overall cheapest edge.

**Cheapest Connected Edge (Prim's Algorithm):** This general algorithm may always choose the component that contains the source node  $s$ . This amounts to Prim's Algorithm.

**The Loop Invariant:** The loop invariant is that we have not gone wrong. There is at least one optimal solution consistent with the choices made so far, i.e., containing the objects committed to so far and not containing the objects rejected so far.

**Maintaining the Loop Invariant (i.e.,  $\langle LI' \rangle \& \text{not } \langle \text{exit} \rangle \& \text{code}_{\text{loop}} \rightarrow \langle LI'' \rangle$ ):** If we are unlucky and the optimal MST  $optS_{LI}$  that is assumed to exist by the loop invariant does not contain the edge  $i$  being committed to this iteration, then we must massage this optimal solution  $optS_{LI}$  into another one  $optS_{ours}$  that contains all of the edges in  $Commit \cup \{i\}$ . This proves that the loop invariant has been maintained.

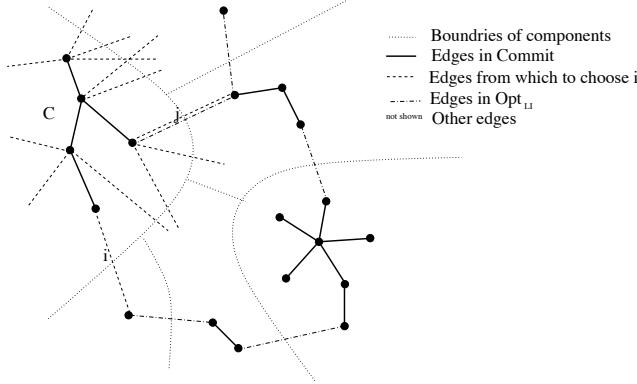


Figure 22.5: Let  $C$  be one of the components of the graph induced by the edges in  $Commit$ . Edge  $i$  is chosen to be the cheapest out of  $C$ . An optimal solution  $optS_{LI}$  is assumed to exist that contains the edges in  $Commit$ . Let  $j$  be any edge in  $optS_{LI}$  that is out of  $C$ . Form  $optS_{ours}$  by removing  $j$  and adding  $i$ .

**Massaging  $optS_{LI}$  into  $optS_{ours}$ :** We add the edge  $i = \{u, v\}$  being committed to this iteration to the optimal solution  $optS_{LI}$ . Because  $optS_{LI}$  spans the graph  $G$ , there is some path  $P$  within it from node  $u$  to node  $v$ . This path along with the edge  $i = \{u, v\}$  creates a cycle, which must be broken by removing one edge from  $P$ . Let  $C$  denote the component of  $Commit$  that the general greedy algorithm chooses the edge  $i$  from. Because  $i$  expands the component without creating a cycle with it, one and only one of the edge  $i$ 's nodes  $u$  and  $v$  are within the component. Hence, this path  $P$  starts within  $C$  and ends outside of  $C$ . Hence, there must be some edge  $j$  in the path that leaves  $C$ . We will delete this edge from our optimal MST, namely  $optS_{ours} = optS_{LI} \cup \{i\} - \{j\}$ .

$optS_{ours}$  is an Optimal Solution:

**$optS_{ours}$  has no cycles:** Because  $optS_{LI}$  has no cycles, we create only one cycle by adding edge  $i$  and we destroyed this cycle by deleting edge  $j$ .

**$optS_{ours}$  Spans  $G$ :** Because  $optS_{LI}$  spans  $G$ ,  $optS_{ours}$  spans it as well. Any path between two nodes that goes through edge  $j$  in  $optS_{LI}$  will now follow the remaining edges of path  $P$  together with edge  $i$  in  $optS_{ours}$ .

**$optS_{ours}$  has minimum weight:** Because  $optS_{LI}$  has minimum weight, it is sufficient to prove that edge  $i$  is at least as cheap as edge  $j$ . Note that by construction edge  $j$  leaves component  $C$  and does not create a cycle with it. Because edge  $i$  was chosen because it was the cheapest (or at least one of the cheapest) such edges, it is true that edge  $i$  is at least as cheap as edge  $j$ .

**Exercise 22.2.3** In Figure 22.4, suppose we decide to change the weight 61 to any real number from  $-\infty$  to  $+\infty$ . What is the interval of values that it could be changed to, for which the MST remains the same? Explain your answer. Similarly for the weight 95.

**Exercise 22.2.4** A favorite game of my kids has been Magic, and now Jugio. One aspect of the game is as follows. You have  $n$  defense cards, the  $i^{\text{th}}$  of which, denoted  $D_i$ , has worth  $w_i$  and defense ability  $d_i$ . Your opponent has  $n$  attack cards, the  $j^{\text{th}}$  of which, denoted  $A_j$ , has attack ability  $a_j$ . You can see all of the cards. Your task is to define a 1-1 matching between the attacking cards and the defending cards, i.e. each attack card is allocated to a unique defense card. If card  $D_i$  is defending against  $A_j$  and  $d_i < a_j$ , then he dies. Your goal is to maximize the sum of the worth  $w_i$  of your cards that live. Give your algorithm. Then prove that it works.

## Chapter 23

# Recursive Backtracking and Dynamic Programming Algorithms

Recursive backtracking and dynamic programming are powerful tools for solving *optimization problems*, which as said, requires finding the best of an exponentially large set of solutions. (See Section 19.) The key component of both of these techniques is a recurrence relation which says how to find an optimal solution for one instance of the problem from optimal solutions for some number of smaller instances to the same problem.

*Recursive backtracking* recursively solves each of these smaller instances. Effectively, the recursive backtracking algorithm enumerates all options trying as many as required and pruning as many as possible. In practice, if the instance that one needs to solve is sufficiently small and has enough structure that a lot of pruning is possible, then an optimal solution can be found for the instance reasonably quickly. However, for large worst case instances, the running time is still exponential.

Instead of recursing on these subinstances, *Dynamic Programming* iteratively fills in a table with an optimal solution for each so that each only needs to be solved once. As such, dynamic programming provides polynomial time algorithms for many important and practical problem.

Personally, I do not like the name “Dynamic Programming.” Though, it is true that dynamic programming algorithms have a “program” of subinstances to solve. But these subinstances are not chosen “dynamically,” but in a fixed pre-scheduled order. In contrast, in recursive backtracking algorithms, the subinstances are constructed dynamically.

One way to design a dynamic programming algorithm is to start by guessing the set of subinstances that need to be solved. However, we feel that it is easier to start by designing the recurrence relation and the easiest way to do this is to first design a recursive backtracking algorithm for the problem. Once this has been done, a technique referred to as *memoization* can be used to mechanically convert this recursive back tracking algorithm into a dynamic programming algorithm. Hence, we begin this chapter with a step by step way to design a recursive backtracking algorithm.

### 23.1 Recursive Backtracking Algorithms

**An Algorithm as a Sequence of Decisions:** An algorithm for finding an optimal solution for your instance must make a sequence of small decisions about the solution, “Do we include the first object in the solution or not?”, “Do we include the second?”, “the third?, … or “At the first fork in the road, do we go left or right?” “At the second fork which direction do we go?”, “at the third?”, .... As one stack in the recursive algorithm, our task is to deal only

with the first of these decisions. A recursive friend will deal with the rest. We have seen that greedy algorithms make decisions simply by committing to the option that looks the best at the moment. However, this usually does not work. Often, in fact, we have no inspirational technique to know how to make each decision in a way that leads to an optimal (or even a sufficiently good) solution. The difficulty is that it is hard to see the global consequences of the local choices that we make. Sometimes a local initial sacrifice can globally lead to a better overall solution. Instead, we use perspiration. We try all options.

**Searching A Maze:** When we come to a fork in the road, we try them all. For each, we get a friend to search exhaustively, backtrack to the fork, and report the highlights. Our task is to determine which of these answers is best overall. Note that our friends will have their own forks to deal with. However, it is best not to worry about this, since their path is their responsibility not ours.



**Code:** The following is the basic structure that the code will take.

**algorithm**  $\text{Alg}(I)$

**$\langle \text{pre-cond} \rangle$ :**  $I$  is an instance to the problem.

**$\langle \text{post-cond} \rangle$ :**  $\text{optSol}$  is one of the optimal solutions for the instance  $I$  and  $\text{optCost}$  is its cost.

begin

  if(  $I$  is small ) then  
    return( brute force answer )

  else

    % Deal with the first decision by trying each of the  $K$  possibilities.

    for  $k = 1$  to  $K$

      % Temporarily, commit to making the first decision in the  $k^{\text{th}}$  way.

$\langle \text{optSol}_k, \text{optCost}_k \rangle$  = Recursively deal with all the remaining decisions and in doing so find the best solution  $\text{optSol}_k$  for our instance  $I$  from amongst those consistent with this  $k^{\text{th}}$  way of making the first decision.  $\text{optCost}_k$  is its cost.

```

    end for
    % Having the best,  $optSol_k$ , for each possibility  $k$ , we keep the best of these best.
     $k_{min}$  = “a  $k$  that minimizes  $optCost_k$ ”
     $optSol = optSol_{k_{min}}$ 
     $optCost = optCost_{k_{min}}$ 
    return  $\langle optSol, optCost \rangle$ 
end if
end algorithm

```

**Searching For the Best Animal:** Suppose instead of searching through a structured maze, we are searching through a large set of objects, say for the best animal at the zoo. Again we break the search into smaller searches each of which we delegate to a friend. We might, for example, ask one friend for the best vertebrate and another the best invertebrate. We will take the best of these best as our answer. This algorithm is recursive. The friend with the vertebrate task asks a friend to find the best mammal, another for the best bird, and another for the best reptile.

**A Classification Tree of Solutions:** This algorithm unwinds into the tree of stack frames that directly mirrors the taxonomy tree that classifies animals. Each solution is identified with a leaf.

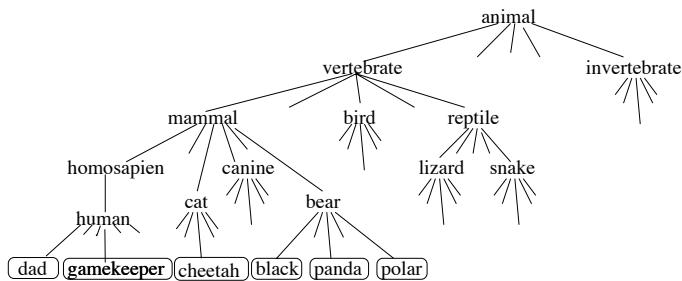


Figure 23.1: Classification Tree of Animals

**Iterating Through the Solutions to Find the Optimal One:** This algorithm amounts to using depth-first search to traverse this classification tree, iterating through all the solutions associated with the leaves. Though this algorithm may seem complex, it is often the easiest way to iterate through all solutions.

**Speeding Up the Algorithm:** This algorithm is not any faster than the *brute force* algorithm that simply compares each animal. However, the structure that the recursive backtracking adds can possibly be exploited to speed up the algorithm. A branch of the tree can be pruned off when we know that this does not eliminate all optimal solutions. Greedy algorithms prune off all branches except one path down the tree. In addition, memoization uses the optimal solution from one subtree to find the optimal in another.

**The Little Bird Abstraction:** Personally, I like to use a “little bird” abstraction to help focus on two of the most difficult and creative parts of designing a recursive backtracking algorithm. Doing so certainly is not necessary. It is up to you.



**What Question to Ask:** The key difference between searching a maze and searching for the best animal is that in the first the forks are fixed by the problem but in the second the algorithm designer is able to choose them. Instead of forking on vertebrates vs invertebrates, we could fork on brown animals vs green animals. This choice is a difficult and creative part of the algorithm design process. It dictates the entire structure of the algorithm, which in turns dictates how well the algorithm can be sped up. I like to view this process of forking as asking *a little bird* a question, “Is the best animal a vertebrate or an invertebrate?”, “Is the best vertebrate, a mammal, a bird, a reptile, or a fish?” The classification tree becomes a strategy for the game of 20 questions. Each sequence of possible answers vertebrate-mammal-cat-cheetah uniquely specifies an animal. Worrying, however, only about the top level of recursion, the algorithm designer must formulate one small question about the optimal solution that is being searched for. The question should be such that having a correct answer greatly reduces your search.

**Constructing a Subinstance for a Friend:** The second creative part of designing a recursive backtracking algorithm is how to express the problem “Find the best mammal” as a smaller instance to the same search problem. The little bird abstraction helps again. We pretend that she answered “mammal.” Trusting (at least temporarily) in her answer, helps us focus, on the fact that we are now only considering mammals and this helps us to design a subinstance that asks for the best one. Another difficulty is that a solution to this subinstance needs to be translated before it is in the correct form to be a solution to our instance.

**A Flock of Stupid Birds vs a Wise Little Bird:** The following two ways of thinking about the algorithm are equivalent. The second is slightly less complicated, but requires further abstraction.

**A Flock of Stupid Birds:** Suppose that our question about whether the optimal solution is a mammal or a reptile has  $K$  different answers. For each, we pretend that a bird gave us this answer. By giving her the benefit of doubt, we ask a friend to give us the optimal solution from amongst those that are consistent with this answer. At least one of these birds must have been telling us the truth. We find it by taking the best of the optimal solutions obtained in this way.

**A Wise Little Bird:** If we had a little bird who would answer our questions correctly, designing an algorithm would be a lot easier: We ask the little bird “Is the best animal, a bird, mammal, reptile, or fish?” She tells us mammal. We ask our friend for the best mammal. Trusting the little bird and the friend, we give this as the best animal. Just as Non-Deterministic Finite Automaton (NFA) and Non-Deterministic

Turing Machines can be viewed as a higher power that provide help, our little bird can be viewed as a limited higher power. She is limited in that we can only ask her questions that do not have too many possible answers, because in reality we must try all these possible answers.

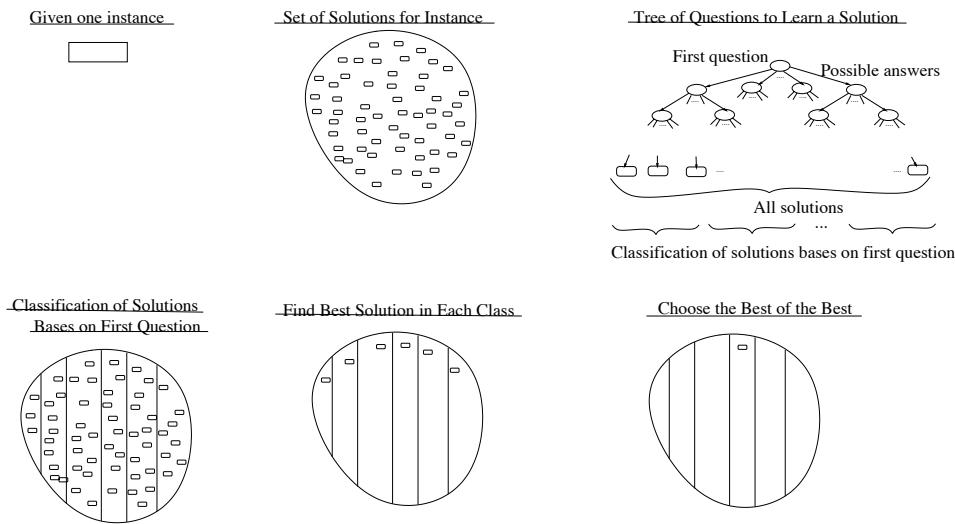


Figure 23.2: Classifying solutions and taking the best of the best

Little blue bird on my shoulder. It's the truth. It's actual. Every thing is factual.  
Zippedy do dah, zippedy ay; wonderful feeling, wonderful day.  
From 1959 Disney movie "Sound of the South",

## 23.2 The Steps in Developing a Recursive Backtracking

This section presents the steps that I recommend using when developing a recursive backtracking algorithm.

**1) Specification:** The first step in designing an algorithm for a problem is to be very clear about what the problem is that needs to be solved. For an optimization problem, we need to be clear about what the set of instances is, for each instance what its set of solutions is, and for each solution what its cost is.

**Shortest Weighted Path within a Directed Leveled Graph:** As a running example, we will develop an algorithm for a version of the shortest-weighted paths problem from Section 20. We use it because it nicely demonstrates the concepts in a graphical way. We generalize the problem by allowing negative weights on the edges and simplify it by requiring the input graph to be leveled.

**Instances:** An instance (input) consists of  $\langle G, s, t \rangle$ , where  $G$  is a weighted directed layered graph  $G$ ,  $s$  is a specified source node and  $t$  is a specified destination node. See Figure 23.3.a. The graph  $G$  has  $n$  nodes. Each node has maximum in and out degree  $d$ . Each edge  $\langle v_i, v_j \rangle$  is labeled with a real-valued (possibly negative)

weight  $w_{\langle v_i, v_j \rangle}$ . The nodes are partitioned into levels so that each edge is directed from some node to a node in a lower level. (This prevents cycles.) It is easiest to assume that the nodes are ordered such that an edge can only go from node  $v_i$  to node  $v_j$  if  $i < j$ .

**Solutions:** A solution for an instance is a path from the source node  $s$  to destination node  $t$ .

**Cost of Solution:** The cost of a solution is the sum of the weights of the edges within the path.

**Goal:** Given a graph  $G$ , the goal is to find a path with minimum total weight from  $s$  to  $t$ .

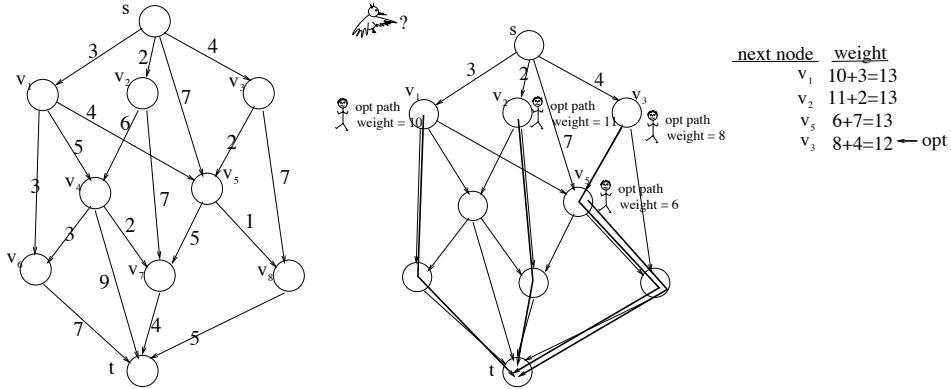


Figure 23.3: a: The directed leveled weighted graph  $G$  used as a running example. b: The recursive backtracking algorithm

**Brute Force Algorithm:** The problem with simply trying all paths is that there may be an exponential number of them.

**Exercise 23.2.1** Give a directed leveled graph on  $n$  nodes that has a small number of edges and as many paths from  $s$  to  $t$  as possible.

**2) Design a Question for the “Little Bird” & Its Answers:** Suppose the little bird knows one of the optimal solutions for our instance. The algorithm designer must formulate a small question about this solution and the list of its possible answers.

**Question About The End Of The Solution:** The question should be such that having a correct answer greatly reduces the search. Generally, we ask for the last “part” of the solution.

**Leveled Graph:** Not knowing yet why we ask about the last part of the solution, we will design this algorithm by asking instead about the first part. Given a graph and nodes  $s$  and  $t$ , I ask “Which edge should we take first to form an optimal path to  $t$ ?“ She assures me that taking edge  $\langle s, v_1 \rangle$  is good.

**The Possible Bird Answers:** Together with your question, you provide the little bird with a list  $A_1, A_2, \dots, A_K$  of possible answers and the little bird answers simply by returning the index  $k \in [1..K]$  of her answer.

**Count The Bird Answers:** You can only ask the bird a little question. (It is only a *little* bird.) By little, I mean that the number  $K$  of different answers that the bird might give must be small. The smaller  $K$  is, the more efficient the final algorithm will be.

**Leveled Graph:** The specification of the problem gives that there are at most  $d$  edges out of any node. Hence, this is a bound on the number  $K$  of different answers.

- 3) Constructing Subinstances:** Suppose that the little bird gives us the  $k^{th}$  of his answers. This giving us some of the solution, we want to ask the friend for the rest of the solution. A friend, however being a recursive call, must be given a smaller instance to the same search problem. The algorithm designer must formulate subinstance  $subI$  for the friend so that he returns to us the information that we desire.

**Leveled Graph:** I start by (at least temporarily) trusting the little bird and take step along the edge  $\langle s, v_1 \rangle$ . Standing at  $v_1$ , the natural question to ask my friend is “Which is the best path from  $v_1$  to  $t$ ?.” Expressed as  $\langle G, v_1, t \rangle$ , this is a subinstance to the same computational problem.

- 4) Constructing a Solution for My Instance:** Suppose that the friend gives you an optimal solution  $optSubSol$  for his instance  $subI$ . How do you produce an optimal solution  $optSol$  for your instance  $I$  from the bird’s answer  $k$  and the friend’s solution  $optSubSol$ ?

**Leveled Graph:** My friend faithfully will give me the path  $optSubSol = \langle v_1, v_6, t \rangle$ , this being a best path from  $v_1$  to  $t$ . The difficulty is that this is not a solution for my instance  $\langle G, t, t \rangle$ , because it is not, in itself, a path from  $s$  to  $t$ . The path from  $s$  is formed by first taking the step from  $s$  to  $v_i$  and then following the best path from there to  $t$ , namely  $optSol = \langle \text{bird answer} \rangle + optSubSol = \langle s, v_1 \rangle + \langle v_1, v_6, t \rangle = \langle s, v_1, v_6, t \rangle$ .

We can trust the friend to give provide an optimal solution to the subinstance  $subI$ , because he is really a smaller recursive version of ourselves. Recall, in Section 14.4, we used strong induction to prove that we can trust our recursive friends.

- 5) Costs of Solutions and Subsolutions:** We must also return the cost  $optCost$  of our solution  $optSol$ . How do you determine it from the bird’s  $k$  and the cost  $optSubCost$  of the friend’s  $optSubSol$ ?

**Leveled Graph:** The cost of the entire path from  $s$  to  $t$  is the cost of the edge  $\langle s, v_1 \rangle$  plus the cost of the path from  $v_1$  to  $t$ . Luckily, our friend gives the latter.  $optCost_k = w_{\langle s, v_1 \rangle} + optSubCost = 3 + 10 = 13$ .

- 6) Best of the Best:** Try all bird’s answers and take best of best.

**Leveled Graph:** If we trust both the bird and the friend, we conclude that this path from  $s$  to  $t$  is a best path. It turns out that because our little bird gave us the wrong first edge, then this might not be the best path from  $s$  to  $t$ . However, our work was not wasted, because we did succeed in finding the best path from amongst those that start with the edge  $\langle s, v_1 \rangle$ . Not trusting the little bird, we repeat this process finding a best path starting with each of  $\langle s, v_2 \rangle$ ,  $\langle s, v_5 \rangle$  and  $\langle s, v_3 \rangle$ . At least one of these four paths must be an overall best path. We give the best of these best as the overall best path.

Let  $optSol_k$  and  $optCost_k$  denote the optimal solution for our instance  $I$  and its cost that we formed when temporarily trusting the  $k^{th}$  bird's answer. Search through this list of costs  $optCost_1, optCost_2, \dots, optCost_K$  finding one of those that is the cheapest. Denote the index of the chosen one with  $k_{min}$ . The optimal solution that we will return is then  $optSol = optSol_{k_{min}}$  and its cost is  $optCost = optCost_{k_{min}}$ .

- 7) Base Cases:** The base case instances are instances to your problem that are small enough that they cannot be solved using the above method, but they can be solved easily in a brute force way. What are these base cases and what are their solutions?

**Leveled Graph:** The only base case is finding a best path from  $s$  to  $t$  when  $s$  and  $t$  are the very same node. In this case, the bird would be unable to give the first edge in the best path because it contains no edges. The optimal solution is the empty path and its cost is zero.

- 8) Code:** From the above pieces, the code can always be put together using the same basic structure.

**algorithm** *LeveledGraph* ( $G, t, t$ )

***⟨pre-cond⟩:***  $G$  is a weighted directed layered graph and  $s$  and  $t$  are nodes.

***⟨post-cond⟩:***  $optSol$  is a path with minimum total weight from  $s$  to  $t$  and  $optCost$  is its weight.

```

begin
    % Base Case: The only base case is for the best path from  $t$  to  $t$ . It's
    % solution is the empty path with cost zero.
    if( $s = t$ ) then
        return  $\langle \emptyset, 0 \rangle$ 
    else
        % General Case:
        % Try each possible bird answer.
        for each of the  $d$  edges  $\langle s, v_k \rangle$ 
            % The bird and Friend Alg: The bird tells us that the first edge in an optimal
            % path from  $s$  to  $t$  is  $\langle s, v_k \rangle$ . We ask the friend for an optimal path from  $v_k$ 
            % to  $t$ . He solves this recursively giving us  $optSubSol$ . To this, we add the
            % bird's edge giving us  $optSol_k$ . This  $optSol_k$  is a best path from  $s$  to  $t$  from
            % amongst those paths consistent with the bird's answer.
             $\langle optSubSol, optSubCost \rangle = LeveledGraph((G, v_k, t))$ 
             $optSol_k = \langle s, v_k \rangle + optSubSol$ 
             $optCost_k = w_{\langle s, v_k \rangle} + optSubCost$ 
        end for
        % Having the best,  $optSol_k$ , for each bird's answer  $k$ , we keep the best of these best.
         $k_{min} = \text{"a } k \text{ that minimizes } optCost_k"$ 
         $optSol = optSol_{k_{min}}$ 
         $optCost = optCost_{k_{min}}$ 
        return  $\langle optSol, optCost \rangle$ 
    end if
end algorithm

```

**Running Time:** Recursive backtracking algorithm faithfully enumerates all solutions for your instance and hence requires exponential time. Though this algorithm may seem complex, how else would you iterate through all paths? We will now use memoization techniques to mechanically convert this algorithm into a dynamic programming algorithm that runs in polynomial time.

**Exercise 23.2.2** (*See solution in Section V*) *An instance may have many optimal solutions with exactly the same cost. The postcondition of the problem allows anyone of these to become output. Which line of code in the above algorithm chooses which of these optimal solutions will be selected?*

### 23.3 The Steps in Developing a Dynamic Programming Algorithm

We will now describe how to use *memoization* to mechanically convert a recursive backtracking algorithm into a dynamic programming algorithm. Memoization speeds up such a recursive algorithm by saving the result for each subinstance it encounters so that it does not need to be recomputed again. Dynamic programming takes the idea of memoization one step further. Instead of traversing the tree of recursive stack frames, keeping track of which friends are waiting for answers from which friends, it first determines the complete set of subinstances for which solutions are needed and then computes them in an order such that no friend must wait. As it goes, it fills out a table containing an optimal solution for each subinstance. The technique for finding an optimal solution for a given subinstance is identical to the technique used in the recursive backtracking algorithm. The only difference is that instead of recursing to solve its subsubinstances, an optimal solution for it, having been found earlier, can be found in the table. When entire table has been completed, the last entry will contain an optimal solution for the original instance.

**1) The Set of Subinstances:** Obtain the set of subinstances that need to be solved by the dynamic programming algorithm by tracing the Recursive Back-Tracking Algorithm through the tree of stack frames starting with the given instance  $I$ . The set consists of the initial instance  $I$ , its subinstances, their subinstances, and so one. Ensure that this set contains all the required subinstances by making sure that is closed under this “sub”-operator, or equivalently that no subinstance is lonely because if its included than so are all its friends. See Section 23.4.3. Also ensure that all (or at least most) of these subinstances are needed.

#### Leveled Graph:

**Include  $\langle G, v_7, t \rangle$ :** On  $\langle G, s, t \rangle$ , the little bird, among other things, suggests taking the edge  $\langle s, v_1 \rangle$  leading to the subinstance  $\langle G, v_1, t \rangle$ . On this subinstance, the little bird, among other things, suggests taking the edge  $\langle v_1, v_4 \rangle$  leading to the subinstance  $\langle G, v_4, t \rangle$ . On this subinstance, the little bird, among other things, suggests taking the edge  $\langle v_4, v_7 \rangle$  leading as said to the subinstance  $\langle G, v_7, t \rangle$ . Hence, this subinstance must be considered by the dynamic programming algorithm. See Figure 23.4.a,  $\langle G, v_7, t \rangle$ .

**Exclude  $\langle G, v_{81}, t \rangle$ :** Given the same instance  $I$ ,  $\langle G, v_{81}, t \rangle$  is not a subinstance, because node  $v_{81}$  is not a node in the graph so never arises.

**Exclude  $\langle G, v_1, v_8 \rangle$ :** Neither is  $\langle G, v_1, v_8 \rangle$  a required subinstance, because each subinstance that arises is looking for a path that ends in the node  $t$ .

**Guess the Set:** Starting with the instance  $\langle G, s, t \rangle$ , the complete set of subinstances called will be  $\{\langle G, v_i, t \rangle \mid \text{for each node } v_i \text{ above } t\}$ , namely, the task of finding the

best path from some new source node  $v_i$  to the original destination node  $t$ . See Figure 23.4.b.

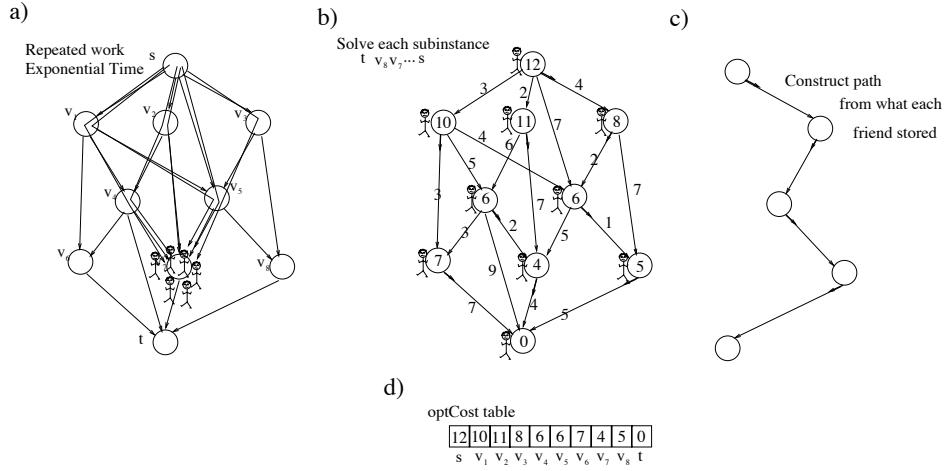


Figure 23.4: a) The recursive algorithm is exponential because different “friends” are assigned the same task. b) The dynamic programming algorithm: For each node  $v_i$ , there is a subinstance  $\langle G, v_i, t \rangle$  asking for an optimum path from  $v_i$  to  $t$ . Each is assigned a friend. The little arrow out of node  $v_i$  indicates the first edge in an optimal path from  $v_i$  to  $t$  and the value within the circle of node gives the cost of this path. c) Gives the optimal path from  $s$  to  $t$ . It can be found by following the little arrows. d) Gives the contents of the  $optCost$  table.

**Redundancy:** We can speed up the recursive backtracking algorithm only when it solves the same subinstance many times.

**Leveled Graph:** The recursive backtracking algorithm designed above traverses each of the exponentially many paths from  $s$  to  $t$ . The good news is that within this exponential amount of work, there is a great deal of redundancy. Different “friends” are assigned the exact same task. In fact, for each path from  $s$  to  $v_i$ , some friend is asked to solve the subinstance  $\langle G, v_i, t \rangle$ . See Figure 23.4.a.

**One Friend Per Subinstance:** To save time, the dynamic programming algorithm solves each of these subinstances only once. I like to imagine that we allocate one “friend” to each of these subinstances whose job it is to find an optimal solution for it and to provide this solution to other friend who needs it.

**2) Count The Subinstances:** The running time of the dynamic programming algorithm is proportional to the number of subinstances. At this point in the design of the algorithm, the program designer should count how many subinstances an instance  $I$  has as a function of the size  $n = |I|$  of the instance. If there are too many of them, then start at the very being designing a new recursive backtracking algorithm with a different question for the little bird.

**Leveled Graph:** The number of subinstances in the set  $\{\langle G, v_i, t \rangle \mid \text{for each node } v_i \text{ above } t\}$  is  $n$ , the number of nodes in the graph  $G$ .

- 3) Construct a Table Indexed by Subinstances:** The algorithm designer constructs a table. It must have one entry for each subinstance. Generally, the table will have one dimension for each “parameter” used to specify a particular subinstance. Each entry in the table is used to store an optimal solution for the subinstance along with its cost. Often we split this table into two tables: one for the solution and one for the cost.

**Leveled Graph:** The single parameter used to specify a particular subinstance is  $i$ . Hence, suitable tables would be  $optSol[0..n]$  and  $optCost[0..n]$ , where  $optSol[i]$  will store the best path from node  $v_i$  to node  $t$  and  $optCost[i]$  will store its cost. See Figure 23.4.d.

- 4) Solution from Subsolutions:** The dynamic programming algorithm for finding an optimal solution to a given instance from an optimal solution to a subinstance is identical to that within the recursive backtracking algorithm.

**Leveled Graph:** Though the recursive backtracking algorithm solves  $\langle G, s, t \rangle$ ,  $Friend_i$  in the dynamic programming algorithm is solving  $\langle G, v_i, t \rangle$ , looking for a best path from  $v_i$  to  $t$ . He does this as follows. He asks the little bird for the first edge in his path and tries each of her possible answers. When the bird suggests the edge  $\langle v_i, v_k \rangle$ , he asks a friend for a best path from  $v_k$  to  $t$ , tacks the bird’s edge  $\langle v_i, v_k \rangle$  onto this path from  $v_k$  to  $t$  giving a path that is the best path from  $v_i$  to  $t$  amongst those consistent with the bird’s answer  $\langle v_i, v_k \rangle$ . After trying each bird’s answer,  $Friend_i$  saves the the best of these best paths in the table.

The only change to the recursive backtracking algorithm is that instead of recursing to solve a subinstance, its optimal solution is found in the table.

**Leveled Graph:** When  $Friend_i$  asks a friend for a best path from  $v_k$  to  $t$ , this task is the subinstance  $\langle G, v_k, t \rangle$ , which has been allocated to  $Friend_k$  and its solution has been stored in  $optSol[k]$ . The recursive backtracking code

$$\begin{aligned} \langle optSubSol, optSubCost \rangle &= LeveledGraph(\langle G, v_k, t \rangle) \\ optSol_k &= \langle s, v_k \rangle + optSubSol \\ optCost_k &= w_{\langle s, v_k \rangle} + optSubCost \end{aligned}$$

is changed to simply

$$\begin{aligned} optSol_k &= \langle v_i, v_k \rangle + optSol[k] \\ optCost_k &= w_{\langle v_i, v_k \rangle} + optCost[k] \end{aligned}$$

- 5) Base Cases:** The base case instances are exact same as with the recursive backtracking algorithm. The dynamic programming algorithm starts its computation by storing in the table an optimal solution for each of these and their costs.

**Leveled Graph:** The only base case is finding a best path from  $s$  to  $t$  when  $s$  and  $t$  are the very same node. This only occurs with the subinstance  $\langle G, v_n, t \rangle$ , where  $v_n$  is another name for  $t$ . The recursive backtracking code

```

if( $s = t$ ) then
    return  $\langle \emptyset, 0 \rangle$ 

```

is changed to simply

```

% Base Case:
optSol[n] =  $\emptyset$ 
optCost[n] = 0

```

- 6) The Order in which to Fill the Table:** When a friend in the recursive backtracking algorithm needs help from a friend the algorithm recurses and the stack frame for the first friend waits until the stack frame for the second friend returns. This forms a tree of recursive stack frames, keeping track of which friends are waiting for answers from which friends. In contrast, in a dynamic programming algorithm, the friends solve their subinstances in an order so that nobody has to wait. Every recursive algorithm must guarantee that it recurses only on “smaller” instances. Hence, if the dynamic-programming algorithm fills in the table from smaller to larger instances, then when an instance is being solved, the solution for each of its subinstances is already available. Alternatively, the algorithm designer can simply choose any order to fill the table that respects the dependencies between the instances and their subinstances.

**Leveled Graph:**  $Friend_i$  with instance  $\langle G, v_i, t \rangle$ , depends on  $Friend_k$  when there is an edge  $\langle v_i, v_k \rangle$ . From the precondition of the problem, we know that each edge must go from a higher level to a lower one and hence we know that  $k > i$ . Filling the table in the order  $t = v_n, v_{n-1}, v_{n-2}, \dots, v_2, v_1, v_0 = s$  ensures that when  $Friend_i$  does his work,  $Friend_k$  has already stored his answer in the table. The subinstance  $\langle G, v_n, t \rangle$ , however, has been solved already. The following loop is put around the general case code of the recursive backtracking algorithm.

```

% General Cases: Loop over subinstances in the table.
for  $i = n - 1$  to 0
    % Solve instance  $\langle G, v_i, t \rangle$  and fill in table entry  $\langle i \rangle$ .

```

Viewing the dynamic programming algorithm as an iterative algorithm, the loop invariant when working on a particular subinstance is that all “smaller” subinstances that will be needed have been solved. Each iteration maintains the loop invariant while making progress by solving this next subinstance.

- 7) The Final Solution:** The original instance will be the last subinstance to be solved. When complete the dynamic program simply returns this answer.

**Leveled Graph:** The original instance  $\langle G, s, t \rangle$  is the same as the subinstance  $\langle G, v_0, t \rangle$ , where  $v_0$  is another name for  $s$ . The dynamic program ends with the code

```
return  $\langle optSol[0], optCost[0] \rangle$ 
```

**8) Code:** From the above pieces, the code can always be put together using the same basic structure.

**algorithm** *LeveledGraph* ( $G, s, t$ )

***⟨pre-cond⟩:***  $G$  is a weighted directed layered graph and  $s$  and  $t$  are nodes.

***⟨post-cond⟩:***  $optSol$  is a path with minimum total weight from  $s$  to  $t$  and  $optCost$  is its weight.

begin

% Table:  $optSol[i]$  stores an optimal path from  $v_i$  to  $t$  and  $costSol[i]$  its cost.

$table[0..n]$   $optSol, optCost$

% Base Case: The only base case is for the best path from  $t$  to  $t$ .

It's solution is the empty path with cost zero.

$optSol[n] = \emptyset$

$optCost[n] = 0$

% General Cases: Loop over subinstances in the table.

for  $i = n - 1$  to 0

% Solve instance  $\langle G, v_i, t \rangle$  and fill in table entry  $\langle i \rangle$ .

% Try each possible bird answers.

for each of the  $d$  edges  $\langle v_i, v_k \rangle$

% The bird and Friend Alg: The bird tells us that the first edge in an optimal path from  $v_i$  to  $t$  is  $\langle v_i, v_k \rangle$ . We ask the friend for an optimal path from  $v_k$  to  $t$ . He gives us  $optSol[k]$  which he had stored in the table. To this we add the bird's edge. This gives us  $optSol_k$  which is a best path from  $v_i$  to  $t$  from amongst those paths consistent with the bird's answer.

$optSol_k = \langle v_i, v_k \rangle + optSol[k]$

$optCost_k = w_{\langle v_i, v_k \rangle} + optCost[k]$

end for

% Having the best,  $optSol_k$ , for each bird's answer  $k$ , we keep the best of these best.

$k_{min} = \text{"a } k \text{ that minimizes } optCost_k \text{"}$

$optSol[i] = optSol_{k_{min}}$

$optCost[i] = optCost_{k_{min}}$

end for

return  $\langle optSol[0], optCost[0] \rangle$

end algorithm

Be sure to include the following withing the code:

- When specifying the pre and post condition of the problem, be clear what a subinstance, a solution, and the cost of a solution are.
- The table must be indexed by subinstances. When allocating the table, be clear what subinstance each entry of the table represents.
- When looping through the subinstances, be clear which subinstance is the current one to be solved.

- Out line the bird and friend algorithm:
  - When looping through the bird answers, be clear which bird answer is currently being tried.
  - When getting help from a friend, be clear what subinstance you give him.
  - Be clear how you form your solution from the bird’s answer and from the friend’s solution. Similarly, costs.
- Be clear what the base case subinstances are and what are their solutions and costs are.
- Be clear how the solution and cost for the original instance are returned.

**9) Running Time:** One can see that the code loops over each subinstance and for each loops over each bird answer. From this, the running time is seems to be the number of subinstances in the table times the number  $K$  of answers to the bird’s question. We will see in Section 23.4.4 that actually the running time of this version of the algorithm is a factor of  $n$  time bigger than this. The same section will tell how to remove this extra factor of  $n$ .

**Leveled Graph:** The running time of this algorithm is now polynomial. There are only  $n$  friends, one for each node in the graph. For instance,  $\langle G, v_i, t \rangle$ , there is a bird answer for each edge out of his source node  $v_i$ . There are at most  $d$  of these. The running time is then only  $\mathcal{O}(n \cdot d)$  times this extra factor of  $n$ .

## 23.4 More Theory

We will now go over some of the more subtle points.

### 23.4.1 The Question For the Little Bird

The designer of a recursive backtracking algorithm, a dynamic programming algorithm, or a greedy algorithm must decide which question to ask the little bird. If you do not like this analogy, you can instead say that the algorithm designer must decide which sequence of decision will specify the solution constructed by the algorithm or can say that he must decide which things the algorithm will try before backtracking to try something else. Either way this is one of the main creative steps in designing the algorithm. This section examines some more advanced techniques that might be used.

**Local vs Global Considerations:** One of the reasons that optimization problems are difficult is that, although we are able to make what we call *local* observations and decisions, it is hard to see the *global* consequences of these decisions.

**Leveled Graph:** Which edge out of  $s$  is cheapest is a local question. Which path is the overall cheapest is a global question. We were tempted to follow the cheapest edge out of the source  $s$ . However, this greedy approach does not work. Sometimes one can arrive at a better overall path by starting with a first edge that is not the cheapest. This local *sacrifice* could globally lead to a better overall solution.

**Ask About A Local Property:** The question that we ask the bird is about some *local* property of the solution. For example:

- If the solution is a sequence of objects, a good question would be, “What is the first object in the sequence?”

**Example:** If the solution is a path though a graph, we might ask, “What is the first edge in the path?”

- If the instance is a sequence of objects and a solution is a subset of these object, a good question would be, “Is the first object of the instance included in the optimal solution?”

**Example:** In the Longest Common Subsequence problem, Section 24.2, we will ask, “Is the first character of either  $X$  or  $Y$  included in  $Z$ ? ”

- If a solution is a binary tree of objects, a good question would be, “What object is at the root of the tree?”

**Example:** In the Best Binary Search Tree problem, Section 24.7, we will ask, “Which key is at the root of the tree?”

In contrast, asking the bird for the number of edges in the best path in the leveled graph is a global not a local question.

**The Number  $K$  of Different Bird Answers:** You can only ask the bird a little question. (It is only a *little* bird.) By little, I mean the following: together with your question, you provide the little bird with a list  $A_1, A_2, \dots, A_K$  of possible answers and the little bird answers simply by returning the index  $k \in [1..K]$  of her answer. In a little question, the number  $K$  of different answers that the bird might give must be small. The smaller  $K$  is, the more efficient the final algorithm will be.

**Leveled Graph:** When asking for an edge out of  $s$ , the number of answers  $K$  is the degree of the node.

**Brute Force:** The obvious question to ask the little bird is for her to tell you an entire optimal solution. However, the number of solutions for your instance  $I$  is likely exponential; each solution is a possible answer. Hence,  $K$  would be exponential. After getting rid of the bird, the resulting algorithm would be the usual brute force algorithm.

**Repeated Questions:** Although you want to avoid thinking about it, each of your recursive friends will have to ask his little bird a similar question. Hence, you should choose a question that provides a reasonable follow-up question of a similar form. For example:

- “What is the second object in the sequence?”
- “Is the second object of the instance included in the optimal solution?”
- “What is the root in the left/right subtree?”

In contrast, asking the bird for the number of edges in the best path in the leveled graph does not have good follow up question.

**Reversing the Order** This change is purely for aesthetic reasons. Note how the dynamic program loops backwards through the nodes of the graph,  $t = v_n, v_{n-1}, v_{n-2}, \dots, v_2, v_1, v_0 = s$ . The standard way to do it is to work forward. The recursive backtracking algorithm worked forward from  $s$ . The Dynamic Programming technique reverses the recursive backtracking algorithm by completing the subinstances from smallest to largest. In order to have the final algorithm move forward, the recursive backtracking algorithm needs to go backwards. To do

this, the little bird should ask something about the end of the solution and not about the beginning.

- I ask the little bird not for the first but for the last edge in the optimal path.
- “What is the last object in the sequence?”
- “Is the last object of the instance included in the optimal solution?”
- We still ask about the root. It is not useful to ask about the leaves.

**Exercise 23.4.1** *Start over and redevelop the dynamic programming algorithm with this new question for the little bird so that the work is completed starting at the top of the graph. Once you have done it yourself, see Section 23.4.6 for the resulting code.*

**The Required Creativity:** Choosing the question to ask the little bird requires some creativity. Like any creative skill, it is learned by looking at other people’s examples and trying a lot of your own. On the other hand, there are not that many different questions that you might ask the bird. Hence, you can design an algorithm using each possible question and use the best of these.

**Common Pitfalls:** In one version of the scrabble game, an input instance consists of a set of letters and a board and the goal is to find a word that returns the most points. A student described the following recursive backtracking algorithm for it. The bird provides the best word out of the list of letters. The friend provides the best place on the board to put the word.

**Bad Question for Bird:** What is being asked of the bird is not a “little question.” The bird is doing most of the work for you.

**Bad Question for Friend:** What is being asked of the friend needs to be a subinstance of the same problem as that of the given instance, not a subproblem of the given problem.

### 23.4.2 Subinstances and Subsolutions

Getting a trustworthy answer from the little bird, narrows our search problem down to the task of finding the best solution from among those solutions consistent with this answer. It would be great if we could simply ask a friend to find us such a solution, however, we are only allowed to ask our friend to solve subinstances of the original computational problem. Our task within this section is to formulate a subinstance to our computational problem such that the search for its optimal solutions somehow parallels our narrowed search task.

**The Recursive Structure of the Problem:** In order to be able to design a recursive backtracking algorithm for a optimization problem, the problem needs to have a recursive structure. The key property is that in order for a solution of the instance to be optimal, some part of the solution must itself be optimal. The computational problem has a recursive structure if the task of finding an optimal way to construct this part of the solution is a subinstance of the same computational problem.

**Leveled Graph:** For a path from  $s$  to  $t$  to be optimal, the subpath from some  $v_i$  to some  $v_j$  along the path must itself be an optimal path between these nodes. The computational problem has a recursive structure because the task of finding an optimal way to construct this part of the path is a subinstance of the same computational problem.

**Question from Answer:** Sometimes it is a challenge to know what subinstance to ask our friend. It turns out that it is easier to know what answer (subsolution) that we want from him. Knowing the answer we want will be a huge hint as to what the question should be.

**Each Solution as a Sequence of Answers:** One task that you as the algorithm designer must do is to organize the information needed to specify a solution into a sequence of fields,  $sol = \langle field_1, field_2, \dots, field_m \rangle$ .

**Best Animal:** In the zoo problem, each solution consists of an animal, which we will identify with the sequence of answers to the little bird's questions,  $sol = \langle \text{vertebrate}, \text{mammal}, \text{cat}, \text{cheetah} \rangle$ .

**Leveled Graph:** In the Leveled Graph Problem, a solution consists of a path,  $\langle s, v_1, v_6, t \rangle$ , which we will identify with the sequence of edges  $sol = \langle \langle s, v_1 \rangle, \langle v_1, v_6 \rangle, \langle v_6, t \rangle \rangle$ .

**Bird's Question and Remaining Task:** The algorithm asks the little bird for the first field  $field_1$  of one of the instance's optimal solutions and asks the friend for the remaining fields  $\langle field_2, \dots, field_m \rangle$ . We will let  $k$  denote the answer provided by the bird and  $optSubSol$  that provided by the friend. Given both, the algorithm formulates the final solution by simply concatenating these two parts together, namely  $optSol = \langle k, optSubSol \rangle = \langle field_1, \langle field_2, \dots, field_m \rangle \rangle$ .

**Leveled Graph:** Asking for the first field of an optimal solution  $optSol = \langle \langle s, v_1 \rangle, \langle v_1, v_6 \rangle, \langle v_6, t \rangle \rangle$  amounts to asking for the first edge that the path should take. The bird answers  $k = \langle s, v_1 \rangle$ . The friend provides  $optSubSol = \langle \langle v_1, v_6 \rangle, \langle v_6, t \rangle \rangle$ . Concatenating these forms our solution.

**Formulating the Subinstance:** We need to find an instance of the computational problem whose optimal solution is  $optSubSol = \langle field_2, \dots, field_m \rangle$ . If one wants to get formal, the instance is that whose set of valid solutions is  $setSubSol = \{subSol \mid \langle k, subSol \rangle \in setSol\}$ .

**Leveled Graph:** The instance whose solution is  $optSubSol = \langle \langle v_1, v_6 \rangle, \langle v_6, t \rangle \rangle$  is  $\langle G, v_1, t \rangle$  asking for the optimal solution from  $v_1$  to  $t$ .

**Costs Solutions:** In addition to finding an optimal solution  $optSol = \langle field_1, \langle field_2, \dots, field_m \rangle \rangle$  for the instance  $I$ , the algorithm must also produce the cost of this solution. To be helpful, the friend provides the cost of his solution,  $optSubSol$ . Due to the recursive structure of the problem, the costs of these solutions  $optSol = \langle field_1, \langle field_2, \dots, field_m \rangle \rangle$  and  $optSubSol = \langle field_2, \dots, field_m \rangle$  usually differ in some uniform way. For example, often the cost is the sum of the costs of the fields, i.e.,  $cost(optSol) = \sum_{i=1}^m cost(field_i)$ . In this case we have that  $cost(optSol) = cost(field_1) + cost(optSubSol)$ .

**Leveled Graph:** The cost of of a path from  $s$  to  $t$  is the cost of the first edge plus the cost of the rest of the path.

### Formal Proof of Correctness:

**Recursive Structure of Costs:** In order for this recursive back tracking method to solve an optimization problem, the costs that the problem allocates to the solutions must

have the following recursive structure. Consider two solutions  $sol = \langle k, subSol \rangle$  and  $sol' = \langle k, subSol' \rangle$  both consistent with the same bird's answer  $k$ . If the given cost function dictates that the solution  $sol$  is better than the solution  $sol'$ , then the subsolution  $subSol$  of  $sol$  will also be better than the subsolution  $subSol'$  of  $sol'$ . This ensures that any optimal subsolution of the subinstance leads to an optimal solution of the original instance.

**Theorem:** The solution  $optSol$  returned is a best solution for  $I$  from amongst those that are consistent with the information  $k$  provided by the bird.

**Proof:** By way of contradiction, assume not. Then, there must be another solution  $betterSol$  consistent with  $k$  whose cost is strictly better than that for  $optSol$ . From the way we constructed our friend's subinstance  $subI$ , this better solution must have the form  $betterSol = \langle k, betterSubSol \rangle$  where  $betterSubSol$  is a solution for  $subI$ . We ensured that the costs are such that because the cost of  $betterSol$  is better than that of  $optSol$ , it follows that the cost of  $betterSubSol$  is better than that of  $optSubSol$ . This contradicts the fact that  $optSubSol$  is an optimal solution for the subinstance  $subI$ . Recall, we proved in Section 14.4 using strong induction that we can trust the friend to provide an optimal solution to the subinstance  $subI$ . The conclusion of this proof by contradiction is that  $optSol$  must be at least as good of a solution as  $betterSol$ .

**Size of an Instance:** In order to avoid recursing indefinitely, the subinstance that you give your friend must be *smaller* than your own instance according to some measure of size. By the way that we formulated the subinstance, we know that its valid solutions  $subSol = \langle field_2, \dots, field_m \rangle$  are shorter than the valid solutions  $sol = \langle field_1, field_2, \dots, field_m \rangle$  of the instance. Hence, a reasonable measure of the size of an instance is the length of its longest valid solution. This measure only fails to work when an instance has valid solutions that are infinitely long.

**Leveled Graph:** The size of the instance  $\langle G, s, t \rangle$  is the length of the longest path or simply the number of levels between  $s$  and  $t$ . Given this, the size of the subinstance  $\langle G, v_i, t \rangle$ , which is the number of levels between  $v_i$  and  $t$ , is smaller.

**Each Solution as a Tree of Answers:** Though most recursive backtracking, dynamic programming, and greedy algorithms fit into the structure defined above, a few have the following more complex structure. In these, the fields specifying a solution are organized into a tree instead of a sequence. For example, if the problem is to find the best binary search tree, then it is quite reasonable that the fields are the nodes of the tree and these fields are organized as the tree itself. The algorithm asks the little bird to tell it the field at the root of one of the instance's optimal solutions. One friend is asked to fill in the left subtree and another the right. See Sections 24.7 and 24.8.

### 23.4.3 The Set of Subinstances

**Can Be Difficult:** When using the memoization technique to mechanically convert a recursive algorithm into an iterative algorithm, the most difficult step is determining for each input instance the complete set of subinstances that will get called by the recursive algorithm, starting with this instance.

**Leveled Graph:** One might speculated that a subinstance for the Leveled Graph Problem consists of  $\{\langle G, v_i, v_j \rangle \mid \text{for each pair of nodes } v_i \text{ and } v_j\}$ , namely, the task of finding the best path between each pair of nodes. Later by tracing the recursive algorithm, we saw that these subinstances are not all needed, because the subinstances called always search for a path ending in the same fixed node  $t$ .

**Guess and Check:** The technique to find the set of subinstances is to first try to trace out the recursive algorithm on a small example and guess what the set of subinstances will be. Then the lemma below can be used to check if this set is big enough and not too big.

**A Set Being Closed under an Operation:** The following mathematical concepts will help you.

We say that the set of even integers is *closed under* addition and multiplication because the sum and the product of any two even numbers is even. In general, we say a set is closed under an operation if applying the operation to any elements in the set results in an element that is also in the set.

**The Construction Game:** Consider the following game: I give you the integer 2. You are allowed to construct new objects by taking objects you already have and either adding them or multiplying them. What is the complete set of number that you are able to construct?

**Guess a Set:** You might guess that you are able to construct the set of positive even integers. How do you know that this set is big enough and not too big?

**Big Enough:** Because the set of positive even integers is closed under addition and multiplication, we claim you will never construct an object that is not a positive even number.

**Proof:** We prove by induction on  $t \geq 0$  that after  $t$  steps you only have positive even numbers. This is true for  $t = 0$ , because initially you only have the positive even integer 2. If it is true for  $t$ , the object constructed in step  $t + 1$  is either the sum or the product of previously constructed objects, which are all positive even integers. Because the set of positive even integers is closed under these operations, the resulting object must also be positive even. This completes the inductive step.

**Not Too Big:** Every positive even integer can be generated by this game.

**Proof:** Consider some positive even number  $i = 2j$ . Initially, we have only 2. We construct  $i$  by adding  $2 + 2 + 2 + \dots + 2$  a total of  $j$  times.

**Conclusion:** The set of positive even integers accurately characterizes which numbers can be generated by this game, no less and no more.

**Lemma:** The set  $\mathcal{S}$  will be the complete set of subinstances called starting from our initial instance  $I_{start}$  iff

1.  $I_{start} \in \mathcal{S}$ .
2.  $\mathcal{S}$  is closed under the “sub”-operator. ( $\mathcal{S}$  is big enough.)

The sub-operator is defined as follows: Given a particular instance to the problem, applying the sub-operator produces all the subinstances constructed from it by a single stack frame of the recursive algorithm.

3. Every subinstance  $I \in \mathcal{S}$  can be generated from  $I_{start}$  using the sub-operator. ( $\mathcal{S}$  is not too big.)

This ensures that there are not any instances in  $\mathcal{S}$  that are not needed. The dynamic-programming algorithm will work fine if your set of subinstances contains subinstances that are not called. However, you do not want the set too much larger than necessary, because the running time depends on its size.

### Examples:

**The Wedding Invitation List:** One of the nightmares when getting married is deciding upon the invitation list. The goal is to make everyone there happy while keeping the number of people small. The three rules in the lemma also apply here.

**$I_{start} \in \mathcal{S}$ :** Clearly the Bride and Groom need to be invited.

**Closure Insures that Everyone is Happy:** If you invite aunt Hilda, then in order to keep her happy, you need to invite her obnoxious son. Similarly, you need to keep all of your subinstances happy, by being sure that for every subinstance included, its immediate friends are also included. In the Wedding List problem, you will quickly invite the entire world. The “Six Degrees of Separation” phenomenon states that the set consisting of your friend’s friend’s friend’s friend’s friend’s friends includes everyone. Luckily, this does not occur here, because like aunt Hilda’s obnoxious son, the base case subinstances do not have any friends.

**Everyone Needed:** We see that everyone on the list must be invited. Even the obnoxious son must come. Because the bride must come, her mother must come. Because her mother must come, aunt Hilda must come, and hence the son.

### Leveled Graph:

**Guess a Set:** The guessed set is  $\{\langle G, v_i, t \rangle \mid \text{for each node } v_i \text{ above } t\}$ .

**Closed:** Consider an arbitrary subinstance  $\langle G, v_i, t \rangle$  from this set. The sub-operator considers some edge  $\langle v_i, v_k \rangle$  and forms the subinstance  $\langle G, v_k, t \rangle$ . This is contained in the stated set of subinstances.

**Generating:** Consider an arbitrary subinstance  $\langle G, v_i, t \rangle$ . It will be called by the recursive algorithm if and only if it can be reached from the original destination source  $s$ . Suppose there is a path  $\langle s, v_{k_1}, v_{k_2}, \dots, v_{k_r}, v_i \rangle$ . Then we demonstrate that the instance  $\langle G, v_i, t \rangle$  is called by the recursive algorithm as follows: The initial stack frame on instance  $\langle G, s, t \rangle$ , among other things, recurses on  $\langle G, v_{k_1}, t \rangle$ , which recurses on  $\langle G, v_{k_2}, t \rangle, \dots$ , which recurses on  $\langle G, v_{k_r}, t \rangle$ , which recurses on  $\langle G, v_i, t \rangle$ .

If the node  $v_i$  cannot be reached from node  $s$ , then the subinstance  $\langle G, v_i, t \rangle$  will never be called by the recursive algorithm. Despite this, we will include it in our dynamic program, because this is not known about  $v_i$  until after the algorithm has run.

**The Number of Subinstances:** A dynamic-programming algorithm is fast only if the given instance does not have many subinstances.

**Subinstance is a Subsequence:** A common reason for the number subinstances of a given instance being polynomial is that the instance consists of a *sequence* of things rather than a *set* of things. In such a case, each subinstance can be a contiguous (continuous) subsequence of the things rather than an arbitrary subset of them. There are only  $\mathcal{O}(n^2)$  contiguous subsequences of a sequence of length  $n$ , because one can be specified by specifying the two end points. Even better, there are even few subinstances if it is

defined to be a prefix of the sequence. There are only  $n$  prefixes, because one can be specified by specifying the one end point. On the other hand, there are  $2^n$  subsets of a set, because for each object you must decide whether or not to include it.

**Leveled Graph:** As stated in the definition of the problem, it is easiest to assume that the nodes are ordered such that an edge can only go from node  $v_i$  to node  $v_j$  if  $i < j$  in this order. We initially guessed that the  $\mathcal{O}(n^2)$  subinstances consisting of subsequences between two nodes  $v_i$  and  $v_j$ . We then decreased this to only the  $\mathcal{O}(n)$  postfixes from some node  $v_i$  to the end  $t$ . Note that subinstances cannot be subsequences of the instance if the input graph is not required to be leveled.

**Sat:** In contrast, the satisfiability problem of Section 25.2, an instance is a set of constraints and any subset of them might be a subinstance. The problem is that there are exponential number,  $2^n$ , of such subinstances. Thus the obvious dynamic programming algorithm does not run in polynomial time. It is widely believed that this problem has no polynomial time algorithm exists.

#### 23.4.4 Decreasing Time and Space

We will now consider a technique for decreasing time and space used by the dynamic-programming algorithm by a factor of  $n$ .

**Recap of a Dynamic Programming Algorithm:** A dynamic programming algorithm has two nested loops (or sets of loops). The first iterates through all the subinstances represented in the table finding an optimal solution for each. When finding an optimal solution for the current subinstance, the second loop iterates through the  $K$  possible answers to the little bird's question, trying each of them. Within this inner loop, the algorithm must find a best solution for the current subinstance from amongst those consistent with the current bird's answer. This step seems to require only a constant amount of work. It involves looking up in the table an optimal solution for a sub-subinstance of the current subinstance and using this to construct a solution for the current subinstance.

**Running Time?:** From the recap of the algorithm, the running time is clearly the number of subinstances in the table times the number  $K$  of answers to the bird's question times what appears (falsely) to be constant time.

**Friend to Friend Information Transfer:** In both a recursive backtracking and a dynamic programming algorithm, information is transferred from sub-friend to friend. In a recursive backtracking, this information is transferred by returning it from a subroutine call. In a dynamic programming algorithm, this information is transferred by having the sub-friend store the information in the table entry associated with his subinstance and having the friend looking this information up from the table. The information transferred is an optimal solution and its cost. The cost, being only an integer, is not a big deal. However, an optimal solution generally requires  $\Theta(n)$  characters to write down. Hence, transferring this information requires this much time.

**Leveled Graph:** In the line of code “ $optSol_k = \langle v_i, v_k \rangle + optSol[k]$ ,”  $Friend_i$  asks  $Friend_k$  for his best path. This path may contain  $n$  nodes. Hence, it could take  $Friend_i$   $\mathcal{O}(n)$  time steps simply to transfer the answer from  $Friend_k$ .

**Time and Space Bottleneck:** Being within these two nested loops, this information transfer is the bottleneck on the running time of the algorithm. In a dynamic programming algorithm, this information for each subinstance is stored in the table for the duration of the algorithm. Hence, this bottleneck on the memory space requirements of the algorithm.

**Leveled Graph:** The total time is  $\mathcal{O}(n \cdot d \cdot n)$ . The total space is  $\Theta(n \cdot n)$  space,  $\mathcal{O}(n)$  for each of the  $n$  table entries. These can be improved to  $\Theta(nd)$  time and  $\Theta(n)$  space.

**A Faster Dynamic Programming Algorithm:** We will now modify the dynamic programming algorithm to decrease its time and space requirements. The key idea is to reduce the amount of information transferred.

**Cost from Sub-Cost:** The sub-friends do not need to provide an optimal sub-solution in order to find the cost of an optimal solution to the current subinstance. The sub-friends need only provide the cost of this optimal sub-solution. Transferring only the costs, speeds up the algorithm.

**Leveled Graph:** In order for  $Friend_i$  to find the *cost* of a best path from  $v_i$  to  $t$ , he need receive only the best *sub-cost* from his friends. (See the numbers within the circles in Figure 23.4.b.) For each of the edges  $\langle v_i, v_k \rangle$  from his source node  $v_i$ , he learns from  $Friend_k$  the cost of a best path from  $v_k$  to  $t$ . He adds the cost of the edge  $\langle v_i, v_k \rangle$  to this to determine the cost of a best path from  $v_i$  to  $t$  from amongst those that take this edge. Then he determines the cost of an overall best path from  $v_i$  to  $t$  by taking the best of these best costs.

Note that this algorithm requires  $\mathcal{O}(n \cdot d)$  not  $\mathcal{O}(n \cdot d \cdot n)$  time because this best cost can be transferred from  $Friend_k$  to  $Friend_i$  in constant time. However, this algorithm finds the cost of the best path, but does not find a best path.

### The Little Bird's Advice:

**Definition of Advice:** A friend trying to find an optimal solution to his subinstance asks the little bird a question about this optimal solution. The answer, usually denoted  $k$ , to this question classifies the solutions. If this friend had an all powerful little bird, then she could advise him which class of solutions to search in to find an optimal solution. Given that he does not have such a bird, he must simply try all  $K$  of the possible answers and determine himself which answer is best. Either way we will refer to this best answer as *the little bird's advice*.

**Leveled Graph:** The bird's advice to  $Friend_i$  who is trying to find a best path from  $v_i$  to  $t$  is which edge to take first. This edge for each friend is indicated by the little arrows in Figure 23.4.b.)

**Advise from Cost:** Consider again the algorithm that transfers only the cost of an optimal solution. Within this algorithm, each friend is able to determine the little bird's advice to him.

**Leveled Graph:**  $Friend_i$  determines for each of the edges  $\langle v_i, v_k \rangle$  from his node the cost of a best path from  $v_i$  to  $t$  from amongst those that take this edge and then determines which of these is best. Hence, though this  $Friend_i$  never learns a best path from  $v_i$  to  $t$  in its entirety, he does learn which edge is taken first. In other

words, he does determine, even without help from the little bird, what the little bird's advice would be.

**Transferring the Bird's Advice:** The little bird's advice does not need to be transferred from  $Friend_k$  to  $Friend_i$  because  $Friend_i$  does not need it. However,  $Friend_k$  will store this advice in the table so that it can be used at the end of the algorithm. This advice can usually be stored in constant space. Hence, it can be stored along with the best cost in constant time without slowing down the algorithm.

**Leveled Graph:** The advice indicates a single edge. Theoretically, taking  $\mathcal{O}(\log n)$  bits, this takes more than constant space, however, practically it can be stored using two integers.

**Information Stored in Table:** The key change in order to make the dynamic programming algorithm faster is that the information stored in the table will no longer be an optimal solution and its cost. Instead, only the cost of an optimal solution and the little bird's advice  $k$  are stored.

**Leveled Graph:** The above code for *LeveledGraph* remains unchanged except for two changes. The line “ $optSol_k = < v_i, v_k > + optSol[k]$ ,” within the inner loop and the line “ $optSol[i] = optSol_{k_{min}}$ ” which stores the optimal solution are commented out (I recommend leaving them in as comments to add clarity for the reader). The second of these is replaced with the line “ $birdAdvice[i] = k_{min}$ ” which stores the bird's advice. See Section 23.4.6 for the new code.

**Time and Space Requirements:** The running time of the algorithm computing the costs and the bird's advice is:

$$\begin{aligned} \text{Time} = & \text{ "the number of subinstances indexing your table"} \\ & \times \text{ "the number of different answers } K \text{ to the bird's question"} \end{aligned}$$

The space requirement is:

$$\text{Space} = \text{ "the number of subinstances indexing your table"}$$

**Constructing an Optimal Solution:** With the above modifications, the algorithm no longer constructs an optimal solution. An optimal solution for the original instance is required, but not for the other subinstance. We construct an optimal solution for the instance using a separate algorithm that is run after the above faster dynamic programming algorithm fills the table in with costs and bird's advice. This new algorithm starts over from the beginning, solving the optimization problem. However, now we know what answer the little bird would give for every subinstance considered. Hence we can simply run the bird-friend algorithm.

**A Recursive Bird-Friend Algorithm:** The second run of the algorithm will be identical to the recursive algorithm, except now we only need to follow one path down the recursion tree. Each stack frame, instead of branching for each of the  $K$  answers that the bird might give, recurses only on the single answer given by the bird. This algorithm runs very quickly. Its running time is proportional to the number of fields needed to represent the optimal solution.

**Leveled Graph:** A best path from  $s = v_0$  to  $t = v_n$  is found as follows. Each friend knows the first edge taken out of his node. What remains is to put these pieces together by walking forward through the graph following the indicated directions. See Figure 23.4.c.

$Friend_s$  knows that the first edge is  $\langle s, v_3 \rangle$ .  $Friend_3$  knows that the next edge is  $\langle v_3, v_5 \rangle$ .  $Friend_5$  knows the edge  $\langle v_8, t \rangle$ . This completes the path.

**algorithm**  $LeveledGraphWithAdvice (\langle G, v_i, t \rangle, birdAdvice)$

$\langle pre \& post-cond \rangle$ : Same as  $LeveledGraph$  except with advice.

```

begin
    if( $v_i = t$ ) then return( $\emptyset$ )
     $k_{min} = birdAdvice[i]$ 
     $optSubSol = LeveledGraphWithAdvice (\langle G, v_{k_{min}}, t \rangle, birdAdvice)$ 
     $optSol = \langle v_i, v_k \rangle + optSubSol$ 
    return  $optSol$ 
end algorithm

```

**Iterative Solution:** Because this last algorithm only recurses once per stack frame, it is easy to turn it into an iterative algorithm. The algorithm is a lot like a greedy algorithm found in Chapter 22 because the algorithm always knows which greedy choice to make. I leave this, however, for you to do as an exercise.

**Exercise 23.4.2** Design this iterative algorithm.

### 23.4.5 Counting The Number of Solutions

The dynamic programming algorithms considered returns one of the possibly many optimal solutions for the given instance. It does not return all of them because there may be an exponential number of them. In this section, we describe how to change the algorithm so that it also outputs how many possible optimal solutions there are.

**Counting Fruit:** If I want to count the number of fruit in a bowl, I can ask one friend to count for me the number of red fruit and another the number of green fruit and another the number of orange. My answer is the sum of these three. If all the orange fruit is rotten, then I might not include the number of orange fruits in my sum.

**Double Counting:** To be sure that we do not double count some optimal solutions we need that the set of solutions consistent with one bird answer is disjoint from those consistent with another bird answer.

**Fruit:** For example, if some fruits are half red and half green, these might get double counted. We want to make sure that my question “colour” disjointly partitions the fruit according to the answer given.

**Leveled Graph:** If the bird’s answer tells us the first edge of the optimal path, then those paths beginning in the first edge are clearly different than the paths ending in the second.

**Longest Common Subsequence:** On the other hand, for the longest common subsequence problem in Section 24.2, the bird’s answers were “the last letter of  $X$  is excluded” and “the last letter of  $Y$  is excluded”. There are solutions for which both are true. This causes

a problem when counting solutions. Hence, we need to change the dynamic programming algorithm so that each solution only has one valid bird answer. For example, with the longest common subsequence problem, we could change the bird's answers to being either “the last letter of  $X$  is excluded” or “the last letter of  $X$  is included”.

**Computing the Count:** In the new dynamic programming algorithm, each friend, in addition to the cost of the optimal solution for his subinstance and the bird's advice, stores the number of optimal solutions for the subinstance. This is computed as follow. Consider me to be one of these friends with one of these Subinstances. I try each of the  $K$  possible bird's answers. When trying  $k$ , I find the best solution to my instance from amongst its solutions that are consistent with this bird's answer  $k$ . I do this by asking a friend of mine to solve some subinstance. This friend tells me the number of optimal solutions to this subinstance. Let  $num_k$  be this number. Generally, there is a one-to-one mapping between my friend's optimal solutions and solutions to my instance that are consistent with this bird's answer  $k$ . Hence, I know that there are  $num_k$  of these. As before let  $optCost_k$  be the cost of the best solution to my instance from amongst its solutions that are consistent with this bird's answer  $k$ . I compute the cost of my optimal solution simply by  $optCost = \max_{k \in [K]} optCost_k$ . There may be a number of bird's answers that lead to this same optimal cost  $optCost$ . Each of these lead to optimal solutions. We were careful that the set of solutions consistent with one bird answer is disjoint from that consistent with another bird answer. Hence, the total number of optimal solutions to my instance is  $num = \sum_{k \in \{k \mid optCost_k = optCost\}} num_k$ . See Section 23.4.6 for the new Leveled Graph code.

**Bounding the Number of Solutions:** Let  $Num(n)$  denote the maximum number of possible optimal solutions that any dynamic program as described above might output given an instance of size  $n$ . The number  $num_k$  of optimal solutions reported by my friend is at most  $Num(n - 1)$ , because he is a subinstance that has size at most  $n - 1$ . There are at most  $K$  bird answers. Hence,  $num = \sum_{k \in \{k \mid optCost_k = optCost\}} num_k$  can be at most  $Num(n) = \sum_{k \in [1..K]} Num(n - 1) = K \times Num(n - 1) = K^n$ . Note it would take exponential time to output them these optimal solutions. However, the number of bits to represent this number is only  $\log_2(K^n) = \log_2(K) \times n = \Theta(n)$ . Hence, this number can be outputted.

### 23.4.6 The New Code

**Three Changes:** The dynamic programming algorithm for the Leveled Graph Problem developed in Section 23.3 has been changed in three ways.

**Reversing the Order:** As described in Section 23.4.1, the algorithm was redeveloped with the little bird being asked for the last edge in the path instead of for the first edge. This has the aesthetic advantage of having the algorithm now branch out forward from  $s$  instead of backwards from  $t$ .

**Storing Bird's Advice Instead of Solution:** As described in Section 23.4.4, the algorithm now stores the little bird's advice instead of the optimal solution. This is done to decrease the time and space used by the algorithm by a factor of  $n$ .

**Counting The Number of Solution:** As described in Section 23.4.5, the algorithm, in addition to one of the many optimal solutions for the given instance, now also outputs the number of possible optimal solutions.

**Code:**

```

algorithm LeveledGraph ( $G, s, t$ )
<pre-cond>:  $G$  is a weighted directed layered graph and  $s$  and  $t$  are nodes.
<post-cond>:  $optSol$  is a path with minimum total weight from  $s$  to  $t$ , and  $optCost$  is it's
    weight, and  $optNum$  is the number of possible optimal solutions.

begin
    % Table:  $optSol[i]$  would store an optimal path from  $s$  to  $v_i$ ,
    % but actually we store only the bird's advice for the subinstance
    % and the cost of its solution.
    table[0.. $n$ ] birdAdvice, optCost, optNum

    % Base Case: The only base case is for the best path from  $s$  to  $s$ .
    It only has one optimal solution which is the empty path with cost zero.
    %  $optSol[0] = \emptyset$ 
    optCost[0] = 0
    birdAdvice[0] =  $\emptyset$ 
    optNum[0] = 1

    % General Cases: Loop over subinstances in the table.
    for  $i = 1$  to  $n$ 
        % Solve instance  $\langle G, s, v_i \rangle$  and fill in table entry  $\langle i \rangle$ .
        % Try each possible bird answers.
        for each of the  $d$  edges  $\langle v_k, v_i \rangle$ 
            % The bird and Friend Alg: The bird tells us that the last edge in an optimal
            % path from  $s$  to  $v_i$  is  $\langle v_k, v_i \rangle$ . We ask the friend for an optimal path from  $s$ 
            % to  $v_k$ . He gives us  $optSol[k]$  which he had stored in the table. To this we
            % add the bird's edge. This gives us  $optSol_k$  which is a best path from  $s$  to  $v_i$ 
            % from amongst those paths consistent with the bird's answer.
            %  $optSol_k = optSol[k] + \langle v_k, v_i \rangle$ 
            optCost $k$  = optCost[ $k$ ] +  $w_{\langle v_k, v_i \rangle}$ 
        end for
        % Having the best,  $optSol_k$ , for each bird's answer  $k$ , we keep the best of these
        % best.
         $k_{min} = \text{"a } k \text{ that minimizes } optCost_k"$ 
        %  $optSol[i] = optSol_{k_{min}}$ 
        optCost[ $i$ ] = optCost $k_{min}$ 
        birdAdvice[ $i$ ] =  $k_{min}$ 
        optNum[ $i$ ] =  $\sum_{k \in \{k \mid optCost_k = optCost_{k_{min}}\}} optNum[k]$ .
    end for
    optSol = LeveledGraphWithAdvice ( $\langle G, s, v_n \rangle, birdAdvice$ )
    return  $\langle optSol, optCost[n], optNum[n] \rangle$ 
end algorithm

```

## Chapter 24

# Examples of Dynamic Programs

This completes the presentation of the general techniques and the theory behind dynamic programming algorithms. We will now develop algorithms for other optimization problems.

### 24.1 The Printing Neatly Problem

Consider the problem of printing a paragraph neatly on a printer. The input text is a sequence of  $n$  words with lengths  $l_1, l_2, \dots, l_n$ , measured in characters. Each printer line can hold a maximum of  $M$  characters. Our criterion for “neatness” is for there to be as few spaces on the ends of the lines as possible.

#### Printing Neatly:

**Instances:** An instance  $\langle M; l_1, \dots, l_n \rangle$  consists of the line and the word lengths. Generally,  $M$  will be thought of as a constant, so we will leave it out when it is clear from the context.

**Solutions:** A solution for an instances is a list giving the number of words for each line,  $\langle k_1, \dots, k_r \rangle$ .

**Cost of Solution:** Given the number of words in each line, the cost of this solution is the sum of the cubes of the number of blanks on the end of each line, (including for now the last line).

**Goal:** Given the line and word lengths, the goal is to split the text into lines in a way that minimizes the cost.

**Example:** Suppose that one way of breaking the text into lines gives 10 blanks on the end of one of the lines, while another way gives 5 blanks on the end of one line and 5 on the end of another. Our sense of esthetics dictates that the second way is “neater.” Our cost heavily penalizes having a large number of blanks on a single line by cubing the number. The cost of the first solution is  $10^3 = 1,000$  while the cost of the second is only  $5^3 + 5^3 = 250$ .

**Local vs Global Considerations:** We are tempted to follow a greedy algorithm and put as many words on the first line as possible. However, a local *sacrifice* of putting few words on this line may lead globally to a better overall solution.

**The Question For the Little Bird:** We will ask for the number of words  $k$  to put on the *last* line. For each of the possible answers, the best solution is found that is consistent with this answer and then the best of these best solutions is returned.

**Your Instance Reduced to a Subinstance:** If the bird told you that an optimal number of words to put on the last line is  $k$ , then an optimal printing of the words is an optimal printing of first  $n - k$  words followed by the remaining  $k$  words on a line by themselves. Your friend can find an optimal way of printing these first words by solving the subinstance  $\langle M; l_1, \dots, l_{n-k} \rangle$ . All you need to do then is to add the last  $k$  words, namely  $optSol = \langle optSubSol, k \rangle$ .

**The Cost:** The total cost of an optimal solution for the given instance  $\langle M; l_1, \dots, l_n \rangle$  is the cost of the optimal solution for the subinstance  $\langle M; l_1, \dots, l_{n-k} \rangle$ , plus the cube of the number of blanks on the end of the line that contains the last  $k$  words, namely  $optCost = optSubCost + (M - k + 1 - \sum_{j=n-k+1}^n l_j)^3$ .

**The Set of Subinstances:** By tracing the recursive algorithm, we see that the set of subinstance used consists only of prefixes of the words, namely  $\{\langle M; l_1, \dots, l_i \rangle \mid i \in [0, n]\}$ .

**Closed:** We know that this set contains all subinstances generated by the recursive algorithm because it contains the initial instance and is closed under the sub-operator. Consider an arbitrary subinstance  $\langle M; l_1, \dots, l_i \rangle$  from this set. Applying the sub-operator constructs the subinstances  $\langle M; l_1, \dots, l_{i-k} \rangle$  for  $1 \leq k \leq i$ , which are contained in the stated set of subinstances.

**Generating:** Consider the arbitrary subinstance  $\langle M; l_1, \dots, l_i \rangle$ . We demonstrate that it is called by the recursive algorithm as follows: The initial stack frame on instance  $\langle M; l_1, \dots, l_n \rangle$ , among other things, sets  $k$  to be 1 and recurses on  $\langle M; l_1, \dots, l_{n-1} \rangle$ . This stack frame also sets  $k$  to be 1 and recurses on  $\langle M; l_1, \dots, l_{n-2} \rangle$ . This continues  $n - i$  times, until the desired  $\langle M; l_1, \dots, l_i \rangle$  is called.

**Constructing a Table Indexed by Subinstances:** We now construct a table having one entry for each subinstance. The single parameter used to specify a particular subinstance is  $i$ . Hence, suitable tables would be  $birdAdvice[0..n]$  and  $cost[0..n]$ .

**The Order in which to Fill the Table:** The “size” of subinstance  $\langle M; l_1, \dots, l_i \rangle$  is simply the number of words  $i$ . Hence, the table is filled in by looping with  $i$  from 0 to  $n$ .

**Code:**

**algorithm** *PrintingNeatly*( $\langle M; l_1, \dots, l_n \rangle$ )

***pre-cond*:**  $\langle l_1, \dots, l_n \rangle$  are the lengths of the words and  $M$  is the length of each line.

***post-cond*:**  $optSol$  splits the text into lines in an optimal way and  $cost$  is its cost.

begin

% Table:  $optSol[i]$  would store an optimal way to print the first  $i$  words of the input,  
but actually we store only the bird’s advice for the subinstance  
and the cost of its solution.

$table[0..n]$   $birdAdvice$ ,  $cost$

% Base Case: The only base case is for the best printing of the first zero words.

It’s solution is the empty printing with cost zero.

%  $optSol[0] = \emptyset$

$cost[0] = 0$

$birdAdvice[0] = \emptyset$

```

% General Cases: Loop over subinstances in the table.
for  $i = 1$  to  $n$ 
    % Solve instance  $\langle M; l_1, \dots, l_i \rangle$  and fill in table entry  $\langle i \rangle$ .
    K = "maximum number  $k$  such that the words of length  $l_{i-k+1}, \dots, l_i$  fit on a single line."
    % Try each possible bird answers.
    for  $k = 1$  to  $K$ 
        % The bird and Friend Alg: The bird tells us to put  $k$  words on the last line.
        We ask the friend for an optimal printing of the first  $i - k$  words. He gives
        us  $optSol[i - k]$  which he had stored in the table. To this we add the bird's
         $k$  words on a new last line. This gives us  $optSol_k$  which is a best printing
        of the first  $i$  words from amongst those printings consistent with the bird's
        answer.
        %  $optSol_k = \langle optSub[i - k], k \rangle$ 
         $cost_k = cost[i - k] + (M - k + 1 - \sum_{j=i-k+1}^i l_j)^3$ 
    end for
    % Having the best,  $optSol_k$ , for each bird's answer  $k$ , we keep the best of these best.
     $k_{min} = \text{"a } k \text{ that minimizes } cost_k"$ 
     $birdAdvice[i] = k_{min}$ 
    %  $optSol[i] = optSol_{k_{min}}$ 
end for
 $optSol = PrintingNeatlyWithAdvice (\langle M; l_1, \dots, l_n \rangle, birdAdvice)$ 
return  $\langle optSol, cost[n] \rangle$ 
end algorithm

```

### An Example:

**Our Instance:** This question is represented as the following Printing Neatly instance,  
 $\langle M; l_1, \dots, l_n \rangle = \langle 11; 4, 4, 3, 5, 5, 2, 2, 2 \rangle$ .

**Possible Solutions:** Three of the possible ways to print this text are as follows.

$\langle k_1, k_2, \dots, k_r \rangle = \langle 2, 2, 2, 2 \rangle$		$\langle k_1, k_2, \dots, k_r \rangle = \langle 1, 2, 2, 3 \rangle$		$\langle k_1, k_2, \dots, k_r \rangle = \langle 2, 2, 1, 3 \rangle$	
Love.life..	$2^3$	Love.....	$7^3$	Love.life..	$2^3$
man.while..	$2^3$	life.man...	$3^3$	man.while..	$2^3$
there.as...	$3^3$	while.there	$0^3$	there.....	$6^3$
we.be.....	$6^3$	as.we.be...	$3^3$	as.we.be...	$3^3$
Cost	= 259	Cost	= 397	Cost	= 259

Of these three, the first and the last are the cheapest and are likely the cheapest of all the possible solutions.

**The Table:** The  $birdAdvice[0..n]$  and  $cost[0..n]$  tables for our "love life" example are given in the following chart. The first and second columns in the table below are used to index into the table. The solutions to the subinstances appear in the third column even though they are not actually a part of the algorithm. The bird's advice and the costs themselves are given in the fourth and fifth columns. Note the original instance, its solution, and its cost are in the bottom row.

i	Subinstance	Sub Solution	Bird's Advice	Sub Cost
0	$\langle 11; \emptyset \rangle$	$\emptyset$		0
1	$\langle 11; 4 \rangle$	$\langle 1 \rangle$	1	$0 + 7^3 = 343$
2	$\langle 11; 4, 4 \rangle$	$\langle 2 \rangle$	2	$0 + 2^3 = 8$
3	$\langle 11; 4, 4, 3 \rangle$	$\langle 1, 2 \rangle$	2	$343 + 3^3 = 370$
4	$\langle 11; 4, 4, 3, 5 \rangle$	$\langle 2, 2 \rangle$	2	$8 + 2^3 = 16$
5	$\langle 11; 4, 4, 3, 5, 5 \rangle$	$\langle 2, 2, 1 \rangle$	1	$16 + 6^3 = 232$
6	$\langle 11; 4, 4, 3, 5, 5, 2 \rangle$	$\langle 2, 2, 2 \rangle$	2	$16 + 3^3 = 43$
7	$\langle 11; 4, 4, 3, 5, 5, 2, 2 \rangle$	$\langle 2, 2, 3 \rangle$	3	$16 + 0^3 = 16$
8	$\langle 11; 4, 4, 3, 5, 5, 2, 2, 2 \rangle$	$\langle 2, 2, 1, 3 \rangle$	3	$232 + 3^3 = 259$

The steps in filling in the last entry are as follows.

k	Solution	Cost
1	$\langle \langle 2, 2, 3 \rangle, 1 \rangle$	$16 + 9^3 = 745$
2	$\langle \langle 2, 2, 2 \rangle, 2 \rangle$	$43 + 6^3 = 259$
3	$\langle \langle 2, 2, 1 \rangle, 3 \rangle$	$232 + 3^3 = 259$
4	Does not fit	$\infty$

Either  $k = 2$  or  $k = 3$  could have been used. We used  $k = 3$ .

**Constructing an Optimal Solution:** The above algorithm finds the optimal cost and the bird's advice for each subinstance, but does not construct the optimal solution. With the bird's advice, one can quickly run the algorithm again to construct the optimal solution.

**algorithm**  $PrintingNeatlyWithAdvice(\langle M; l_1, \dots, l_i \rangle, birdAdvice)$

**$\langle pre \& post-cond \rangle$ :** Same as  $PrintingNeatly$  except with advice.

```

begin
    if( $i = 0$ ) then
         $optSol = \emptyset$ 
        return  $optSol$ 
    else
         $k_{min} = birdAdvice[i]$ 
         $optSubSol = PrintingNeatlyWithAdvice(\langle M; l_1, \dots, l_{i-k_{min}} \rangle, birdAdvice)$ 
         $optSol = \langle optSubSol, k_{min} \rangle$ 
        return  $optSol$ 
    end if
end algorithm

```

**Continuing the Example:** The iterative version of the algorithm will be presented by going through its steps. The solution is constructed backwards.

1. We want a solution for  $\langle 11; 4, 4, 3, 5, 5, 2, 2, 2 \rangle$ . Indexing into the table we get  $k = 3$ . We put the last  $k = 3$  words on the last line, forming the solution  $\langle k_1, k_2, \dots, k_r \rangle = \langle ???, 3 \rangle$ .
2. Now we want a solution for  $\langle 11; 4, 4, 3, 5, 5 \rangle$ . Indexing into the table we get  $k = 1$ . We put the last  $k = 1$  words on the last line, forming the solution  $\langle k_1, k_2, \dots, k_r \rangle = \langle ???, 1, 3 \rangle$ .

3. Now we want a solution for  $\langle 11; 4, 4, 3, 5 \rangle$ . Indexing into the table we get  $k = 2$ . We put the last  $k = 2$  words on the last line, forming the solution  $\langle k_1, k_2, \dots, k_r \rangle = \langle ???, 2, 1, 3 \rangle$ .
4. Now we want a solution for  $\langle 11; 4, 4 \rangle$ . Indexing into the table we get  $k = 2$ . We put the last  $k = 2$  words on the last line, forming the solution  $\langle k_1, k_2, \dots, k_r \rangle = \langle ???, 2, 2, 1, 3 \rangle$ .
5. Now we want a solution for  $\langle 11; \emptyset \rangle$ . We know that the optimal solutions to this is  $\emptyset$ . Hence, our solution  $\langle k_1, k_2, \dots, k_r \rangle = \langle 2, 2, 1, 3 \rangle$ .

**Time and Space Requirements:** The running time is the number of subinstances times the number of possible bird answers and the space is the number of subinstances. There are  $\Theta(n)$  subinstances in the table and the number of possible bird answers is  $\Theta(n)$  because she has the option of telling you pretty well any number of words to put on the last line. Hence, the total running time is  $\Theta(n) \cdot \Theta(n) = \Theta(n^2)$  and the space requirements are  $\Theta(n)$ .

**Reusing the Table:** Sometimes you can solve many related instances of the same problem using the same table.

**The New Problem:** When actually printing text neatly, it does not matter how many spaces are on the end of the very last line. Hence, the cube of this number should not be included in the cost.

**Algorithm:** For  $k = 1, 2, 3, \dots$ , we find the best solution with  $k$  words on the last line and then take the best of these best. How to print all but the last  $k$  words is an instance of the original Printing Neatly Problem because we charge for all of the remaining lines of text. Each of these takes  $\mathcal{O}(n^2)$  time so the total time will be  $\mathcal{O}(n \cdot n^2)$ .

**Reusing the Table:** Time can be saved by filling in the table only once. One can get the costs for these different instances off this single table. After determining which is best, call *PrintingNeatlyWithAdvice* once to construct the solution for this instance. The total time is reduced to only  $\mathcal{O}(n^2)$ .

## 24.2 The Longest Common Subsequence Problem

With so much money in things like genetics, there is a big demand for algorithms that find patterns in strings. The following optimization problem is called the *longest common subsequence*.

**Longest Common Subsequence:**

**Instances:** An instance consists of two sequences:  $X = \langle A, B, C, B, D, A, B \rangle$  and  $Y = \langle B, D, C, A, B, A \rangle$ .

**Solutions:** A subsequence of a sequence is a subset of the elements taken in the same order. For example,  $Z = \langle B, C, A \rangle$  is a subsequence of  $X = \langle A, \underline{B}, \underline{C}, B, D, \underline{A}, B \rangle$ . A solution is a subsequence  $Z$  that is common to both  $X$  and  $Y$ . For example,  $Z = \langle B, C, A \rangle$  is a solution because it is a subsequence common to both  $X$  and  $Y$  ( $Y = \langle \underline{B}, D, \underline{C}, \underline{A}, B, A \rangle$ ).

**Cost of Solution:** The cost (or success) of a solution is the length of the common subsequence, e.g.,  $|Z| = 3$ .

**Goal:** Given two sequences  $X$  and  $Y$ , the goal is to find the longest common subsequence (LCS). For the example given above,  $Z = \langle B, C, B, A \rangle$  would be the longer common subsequence.

**Greedy Algorithm:** Suppose  $X = \langle A, B, C, D \rangle$  and  $Y = \langle B, C, D, A \rangle$ . A greedy algorithm may commit to matching the two  $A$ 's. However, this would be a mistake because the optimal answer is  $Z = \langle B, C, D \rangle$ .

**Possible Little Bird Answers:** Typically, the question asked of the little bird is for some detail about the end of an optimal solution.

**Case  $x_n \neq z_l$ :** Suppose that the bird assures us that the last character of  $X$  is not the last character of at least one longest common subsequence  $Z$  of  $X$  and  $Y$ . We could simply ignore this last character of  $X$ . We could ask a friend to give us a longest common subsequence of  $X' = \langle x_1, \dots, x_{n-1} \rangle$  and  $Y$  and this would be a longest common subsequence of  $X$  and  $Y$ .

**Case  $y_m \neq z_l$ :** Similarly, if we are told that the last character of  $Y$  is not used, than we could ignore it.

**Case  $x_n = y_m = z_l$ :** Suppose that the little bird tells us that the last character of both  $X$  and  $Y$  is the last character of an optimal  $Z$ . This, of course, requires that the last characters of  $X$  and  $Y$  are the same. In this case, we could simply ignore this last character of both  $X$  and  $Y$ . We could ask a friend to give us a longest common subsequence of  $X' = \langle x_1, \dots, x_{n-1} \rangle$  and  $Y' = \langle y_1, \dots, y_{m-1} \rangle$ . A longest common subsequence of  $X$  and  $Y$  would be the same, except with the character  $x_n = y_m$  tacked on to the end, i.e.,  $Z = Z'x_n$ . On the other hand, if the little bird answers this case  $x_n = y_m = z_l$  and we on our own observe that  $x_n \neq y_m$ , then we know that the little bird is wrong.

**Case  $z_l = ?$ :** Even more extreme, suppose that the little bird goes as far as to tell us the last character of a longest common subsequence  $Z$ . We could then delete the last characters of  $X$  and  $Y$  up to and including the last occurrence of this character. A friend could give us a longest common subsequence of the remaining  $X$  and  $Y$  and then we could add on the known character to give us  $Z$ .

**Case  $x_n = z_l \neq y_m$ :** Suppose that the bird assures us that the last character of  $X$  is the last character of an optimal  $Z$ . This would tell us the last character of  $Z$  and hence the last case would apply.

**The Question For the Little Bird:** Above we gave a number of different answers that the little bird might give, each of which would help us find an optimal  $Z$ . We could add even more possible answers to the list. However, the larger the number  $K$  of possible answers is, the more work our algorithm will have to do. Hence, we want to narrow this list of possibilities down as far as possible. We will consider only the first three bird answers. This is sufficient because for every possible solution at least one of these is true, i.e. either 1)  $x_n \neq z_l$ , 2)  $y_m \neq z_l$ , or 3)  $x_n = y_m = z_l$ . Of course, it may be the case that  $x_n \neq z_l$  and  $y_m \neq z_l$  but if this is true than the bird has a choice whether to answer (1) or (2).

**The Set of Subinstances:** We guess that the set of subinstances of the instance  $\langle\langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_m \rangle\rangle$  is  $\{\langle\langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle\rangle \mid i \leq n, j \leq m\}$ .

**Closed:** We know that this set contains all subinstances generated by the recursive algorithm because it contains the initial instance and is closed under the sub-operator. Consider an arbitrary subinstance,  $\langle\langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle\rangle$ . Applying the sub-operator constructs the subinstances  $\langle\langle x_1, \dots, x_{i-1} \rangle, \langle y_1, \dots, y_{j-1} \rangle\rangle$ ,  $\langle\langle x_1, \dots, x_{i-1} \rangle, \langle y_1, \dots, y_j \rangle\rangle$ ,

and  $\langle\langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_{j-1} \rangle\rangle$ , which are all contained in the stated set of subinstances.

**Generating:** We know that the specified set of subinstances does not contain subinstances not called by the recursive program because we can construct any arbitrary subinstance from the set with the sub-operator. Consider an arbitrary subinstance  $\langle\langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle\rangle$ . The recursive program on instance  $\langle\langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_m \rangle\rangle$  can recurse repeatedly on the first option  $n - i$  times and then repeatedly on the second option  $m - j$  times. This results in the subinstance  $\langle\langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle\rangle$ .

**Constructing a Table Indexed by Subinstances:** We now construct a table having one entry for each subinstance. It will have a dimension for each of the parameters  $i$  and  $j$  used to specify a particular subinstance. The tables will be  $cost[0..n, 0..m]$  and  $birdAdvice[0..n, 0..m]$ .

**Base Cases:** The subinstance represented by  $cost[0, j]$  is  $\langle\emptyset, \langle y_1, \dots, y_m \rangle\rangle$  and has no characters in  $X$ . Hence, it is not reasonable to ask the bird about the last character of  $X$ . Besides, this is an easy case to handle as a base case. The only subsequence of the empty string is the empty string. Hence, the longest common subsequence is the empty string. The cost of this solution is  $cost[0, j] = 0$ .

**Order in which to Fill the Table:** The official order in which to fill the table with subinstances is from “smaller” to “larger.” The “size” of the subinstance  $\langle\langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle\rangle$  is  $i + j$ . Thus, you would fill in the table along the diagonals. However, the obvious order of looping for  $i = 0$  to  $n$  and  $j = 0$  to  $m$  also respects the dependencies between the instances and thus could be used instead.

**Code:**

```

algorithm LCS( $\langle\langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_m \rangle\rangle$ )
⟨pre-cond⟩: An instance consists of two sequences
⟨post-cond⟩: optSol is a longest common subsequence and optCost is its length.

begin
    % Table: optSol[ $i, j$ ] would store an optimal LCS for  $\langle\langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle\rangle$ , but
    % actually we store only the bird's advice for the subinstance and the cost of its
    % solution.
    table[0.. $n$ , 0.. $m$ ] birdAdvice, cost

    % Base Cases: The base cases consist of when one string or the other is empty, i.e.
    % when  $i = 0$  or when  $j = 0$ .
    For each, the solution is the empty string with cost zero.
    for  $j = 0$  to  $m$ 
        % optSol[0,  $j$ ] =  $\emptyset$ 
        cost[0,  $j$ ] = 0
        birdAdvice[0,  $j$ ] =  $\emptyset$ 
    end for
    for  $i = 0$  to  $n$ 
        % optSol[ $i$ , 0] =  $\emptyset$ 
        cost[ $i$ , 0] = 0

```

```

birdAdvice[i, 0] = ∅
end for

% General Cases: Loop over subinstances in the table.
for i = 1 to n
    for j = 1 to m
        % Solve instance  $\langle\langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle\rangle$  and fill in table entry  $\langle i, j \rangle$ .
        % The bird and Friend Alg: The bird tells us either (1)  $x_i \neq z_l$ , (2)  $y_j \neq z_l$ ,
        % or (3)  $x_i = y_j = z_l$ . We remove this last letter (1)  $x_n$ , (2)  $y_m$ , or (3) both
        % and ask the friend for an optimal LCS for the remaining words. He gives
        % us (1)  $optSol[i - 1, j]$ , (2)  $optSol[i, j - 1]$ , or (3)  $optSol[i - 1, j - 1]$  which
        % he had stored in the table. For cases 1 & 2, we leave this the same. For
        % case 3, we add one the bird's letter, assuming that  $x_i = y_j$ . This gives us
        %  $optSol_k$  which is a LCS for  $\langle i, j \rangle$  from amongst those printings consistent
        % with the bird's answer.
        % Try each possible bird answers.
        % case k = 1)  $x_n \neq z_l$ 
        %   optSol1 = optSol[i - 1, j]
        %   cost1 = cost[i - 1, j]
        % case k = 2)  $y_m \neq z_l$ 
        %   optSol2 = optSol[i, j - 1]
        %   cost2 = cost[i, j - 1]
        % case k = 3)  $x_n = y_m = z_l$ 
        if  $x_i = y_j$  then
            % optSol3 = optSol[i - 1, j - 1] +  $x_i$ 
            % cost3 = cost[i - 1, j - 1] + 1
        else
            % Bird was wrong
            % optSol3 = ?
            % cost3 = -∞
        end if
    % end cases
    % Having the best,  $optSol_k$ , for each bird's answer  $k$ ,
    % we keep the best of these best.
    kmax = "a  $k \in [1, 2, 3]$  that maximizes  $cost_k$ "
    % optSol[i, j] = optSolkmax
    cost[i, j] = costkmax
    birdAdvice[i, j] = kmax
end for
end for
optSol = LCSWithAdvice ( $\langle\langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_m \rangle\rangle$ , birdAdvice)
return  $\langle optSol, cost[n, m] \rangle$ 
end algorithm

```

**Using Information About the Subinstance:** This algorithm only has three different bird answers. Surely, this is good enough. However, the number of the possible answers can be narrowed even further. We know the instance is  $\langle\langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle\rangle$ , hence without asking the bird, we know whether or not the last characters are the same, i.e.,  $x_n = y_m$ . If

$x_n \neq y_m$ , the third case with  $x_n = y_m = z_l$  is clearly not possible. One might also wonder whether the case  $x_n \neq z_l$  will lead to an optimal solution when  $x_n = y_m$ . The next exercise shows that we only need to consider the third case, i.e., as in a greedy algorithm, we know the answer even before asking the question.

**Exercise 24.2.1** (*See solution in Section V*) Prove that if  $x_n = y_m$ , then we only need to consider the third case and if  $x_n \neq y_m$ , then we only need to consider the first two cases.

The loop over subinstances is then changed as follows.

```
% Solve instance  $\langle\langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle\rangle$  and fill in table entry  $\langle i, j \rangle$ .
if  $x_i = y_j$  then
    birdAdvice[i, j] = 3
    % optSol[i, j] = optSol[i - 1, j - 1] + x_i
    cost[i, j] = cost[i - 1, j - 1] + 1
else
    % Try possible bird answers.
    % cases k = 1, 2
    % optSol_1 = optSol[i - 1, j]
    cost_1 = cost[i - 1, j]
    % optSol_2 = optSol[i, j - 1]
    cost_2 = cost[i, j - 1]
% end cases
% Having the best, optSol_k, for each bird's answer k,
we keep the best of these best.
k_max = "a  $k \in [1, 2]$  that maximizes cost_k"
% optSol[i, j] = optSol_{k_max}
cost[i, j] = cost_{k_max}
birdAdvice[i, j] = k_max
end if
```

### Constructing an Optimal Solution:

**algorithm**  $LCSWithAdvice(\langle\langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle\rangle, birdAdvice)$

***⟨pre & post-cond⟩:*** Same as  $LCS$  except with advice.

```
begin
    if ( $i = 0$  or  $j = 0$ ) then
        optSol =  $\emptyset$ 
        return optSol
    end if
    k_max = birdAdvice[i, j]
    if  $k_{max} = 1$  then
        optSubSol =  $LCSWithAdvice(\langle\langle x_1, \dots, x_{i-1} \rangle, \langle y_1, \dots, y_{j-1} \rangle\rangle, birdAdvice)$ 
        optSol =  $\langle optSubSol, x_i \rangle$ 
    else if  $k_{max} = 2$  then
```

```

 $optSubSol = LCSWithAdvice(\langle\langle x_1, \dots, x_{i-1} \rangle, \langle y_1, \dots, y_j \rangle\rangle, birdAdvice)$ 
 $optSol = optSubSol$ 
else if  $k_{max} = 3$  then
     $optSubSol = LCSWithAdvice(\langle\langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_{j-1} \rangle\rangle, birdAdvice)$ 
     $optSol = optSubSol$ 
end if
return  $optSol$ 
end algorithm

```

**Time and Space Requirements:** The running time is the number of subinstances times the number of possible bird answers and the space is the number of subinstances. The number of subinstances is  $\Theta(n^2)$  and the bird has  $K=$ three possible answers for you. Hence, the time and space requirements are both  $\Theta(n^2)$ .

**Example:**

j	0	1	2	3	4	5	6	7	8
y <sub>j</sub>	0	1	2	0	1	1	0	1	0
i	x <sub>i</sub>	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0
1	1	0	0	1	1	1	1	1	1
2	0	0	1	1	2	2	2	2	2
3	0	0	1	1	2	2	3	3	3
4	1	0	1	2	2	3	3	4	4
5	0	0	1	2	3	3	3	4	5
6	1	0	1	2	3	4	4	5	5
7	0	0	1	2	3	4	4	5	6

Figure 24.1: Consider the instance  $X = 1001010$  and  $Y = 01011010$ . The tables generated are given. Each number is  $cost[i, j]$ , which is the length of the longest common subsequence of the first  $i$  characters of  $X$  and the first  $j$  characters of  $Y$ . The arrow indicates whether the bird's advice is to include  $x_i = y_j$ , to exclude  $x_i$ , or to exclude  $y_j$ . The circled digits of  $X$  and  $Y$  give an optimal solution.

### 24.3 More of the Input Iterative Loop Invariant Perspective

Dynamic programs can be thought of from two very different perspectives: as an optimized recursive back tracking algorithm and as an iterative loop invariant algorithm that fills in a table. It is important to understand both perspectives. From the perspective of an iterative algorithm, the loop invariant maintained is that the previous table entries have been filled in correctly. Progress is made while maintaining this loop invariant by filling in the next entry. This is accomplished using the solutions stored in the previous entries.

In *More of the Input* type of iterative algorithms (see Section 8.4), the subinstances are prefixes of the instance and hence the algorithm iterates through the subinstances by iterating in some way through the elements of the instance.

**Longest Increasing Contiguous SubSequence:** To begin understanding dynamic programming from the iterative loop invariant perspective, let us consider a very simple example.

Suppose that the input consists of a sequence  $A[1, n]$  of integers and we want to find the longest contiguous subsequence  $A[k_1, k_2]$  such that the elements are monotonically increasing. For example, the optimal solution for  $[5, 3, 1, 3, 7, 9, 8]$  is  $[1, 3, 7, 9]$ .

**Longest Block of Ones:** The problem is very similar to the Longest Block of Ones problem given in Section 9.2.

**Deterministic Non-Finite Automata:** The algorithm will read the input characters one at a time. Let  $A[1, i]$  denote the subsequence read so far. The loop invariant will be that some information about this prefix  $A[1, i]$  is stored. From this information about  $A[1, i]$  and the element  $A[i + 1]$ , the algorithm must be able to determine the required information about the prefix  $A[1, i + 1]$ . In the end, the algorithm must be able to determine the solution, from this information about the entire sequence  $A[1, n]$ . Such an algorithm is called a Deterministic Finite Automata (DFA) if only a constant amount of information is stored at each point in time. (See Section 9.2.) However, in this chapter more memory than this will be required.

**The Algorithm:** After reading  $A[1, i]$ , remember the longest increasing contiguous subsequence  $A[k_1, k_2]$  read so far and its size. In addition, so that you know whether the current increasing contiguous subsequence gets to be longer than the previous one, save the longest one ending in the value  $A[i]$ , and its size.

If you have this information about  $A[1, i - 1]$ , then you can learn it about  $A[1, i]$  as follows. If  $A[i - 1] \leq A[i]$ , then the longest increasing contiguous subsequence ending in the current value increases in length by one. Otherwise, it shrinks to being only the one element  $A[i]$ . If this subsequence increases to be longer than our previously longest, then it replaces the previous longest. In the end, we know the longest increasing contiguous subsequence.

The running time is  $\Theta(n)$ . This is not a DFA because the amount of space to remember an index and a count is  $\Theta(\log n)$ .

**Longest Increasing SubSequence:** The following is a harder problem. Again the input consists of a sequence  $A$  of integers of size  $n$ . However now we want to find the longest (not necessarily contiguous) subsequence  $S \subseteq [1, n]$  such that the elements, in the order that they appear in  $A$ , are monotonically increasing. For example, an optimal solution for  $[5, 1, 5, 7, 2, 4, 9, 8]$  is  $[1, 5, 7, 9]$  and so is  $[1, 2, 4, 8]$ .

**Dynamic Programming Deterministic Non-Finite Automata:** Again the algorithm will read the input characters one at a time. But now the algorithm will store suitable information, not only about the current subsequence  $A[1, i]$ , but also about each previous subsequence  $\forall j \leq i, A[1, j]$ . Each of these subsequences  $A[1, j]$  will be referred to as a *subinstance* of the original instance  $A[1, n]$ .

**The Algorithm:** As before, we will store both the longest increasing sequence seen so far and the longest one(s) that we are currently growing.

**The Loop Invariant:** Suppose that the subsequence read so far is  $10, 20, 1, 30, 40, 2, 50$ . Then  $10, 20, 30, 40, 50$  is the longest increasing subsequence so far. A shorter one is  $1, 2, 50$ . The problem is that these end in a large number, so we may not be able to extend them further. If the rest of the string is  $3, 4, 5, 6, 7, 8$ , we will have to have remembered that  $1, 2$  is the longest increasing subsequence that ends in the value  $2$ .

In fact, for many values  $v$ , we need to remember the longest increasing subsequence ending in this value  $v$  (or smaller), because in the end, it may be that the many of the remaining elements increase starting from this value. We only need to do this for values  $v$  that have been seen so far in the array. Hence, one possibility is to store, for each  $j \leq i$ , the longest increasing subsequence in  $A[1, j]$  that ends with the value  $A[j]$ .

**Maintaining the Loop Invariant:** If you have this information for each  $j \leq i - 1$ , then we learn it about  $A[i]$  as follows. For each  $j \leq i - 1$ , if  $A[j] \leq A[i]$ , then  $A[i]$  can extend the subsequence ending with  $A[j]$ . Given this construction, the maximum length for  $i$  would then be one more than that for  $j$ . We get the longest one for  $i$  by taking the best of these overall such  $j$ . If there is no such  $j$ , then the count for  $i$  will be 1, namely simply  $A[i]$  itself.

**Final Solution:** In the end, the solution is the increasing subsequence ending in  $A[j]$  where  $j \in [1, n]$  is that for which this count is the largest.

**Running Time:** The time for finding this best subsequence ending in  $A[j]$  from which to extend to  $A[i]$  would take  $\Theta(i)$  time if each  $j \in [1, i - 1]$  needed to be checked. However, by storing this information in a heap, the best  $j$  can be found in  $\Theta(\log i)$  time. This gives a total time of  $\Theta(\sum_{i=1}^n \log i) = \Theta(n \log n)$  for this algorithm.

**Recursive Backtracking:** We can also understand this same algorithm from the recursive backtracking perspective. Given the goal of finding the longest increasing subsequence of  $A[1, i]$ , one might ask the “little bird” whether or not the last element  $A[i]$  should be included. Both options need to be tried. If  $A[i]$  is not to be included, then the remaining subtask is to find the longest increasing subsequence of  $A[1, i - 1]$ . This is clearly a subinstance of the same problem. However, if  $A[i]$  is to be included, then the remaining subtask is to find the longest increasing subsequence of  $A[1, i - 1]$  that ends in a value that is smaller or equal to this last value  $A[i]$ . This is another way of seeing that we need to learn both the longest increasing sequence of the prefix  $A[1, i]$  and the longest one ends in  $A[i]$ .

## 24.4 A Greedy Dynamic Program: The Weighted Job/Activity Scheduling Problem

**The Weighted Event Scheduling Problem:** Suppose that many events want to use your conference room. Some of these events are given a higher priority than others. Your goal is to schedule the room in the optimal way. A greedy algorithm was given for the unweighted version in Section 22.2.1.

**Instances:** An instance is  $\langle\langle s_1, f_1, w_1 \rangle, \langle s_2, f_2, w_2 \rangle, \dots, \langle s_n, f_n, w_n \rangle\rangle$ , where  $0 \leq s_i \leq f_i$  are the starting and finishing times and  $w_i$  the priority weight for  $n$  events.

**Solutions:** A solution for an instance is a schedule  $S$ . This consists of a subset  $S \subseteq [1..n]$  of the events that don't conflict by overlapping in time.

**Cost of Solution:** The cost (or success)  $C(S)$  of a solution  $S$  is the sum of the weights of the events scheduled, i.e.,  $\sum_{i \in S} w_i$ .

**Goal:** The goal of the algorithm is to find the *optimal solution*, i.e., the one that maximizes the total scheduled weight.

## Failed Algorithms:

**Greedy Earliest Finishing Time:** The greedy algorithm used in Section 22.2.1 for the unweighted version greedily selects the event with the earliest finishing time  $f_i$ . This algorithm fails, when the events have weights. The following is a counterexample.

$$\frac{1}{\underline{\hspace{1cm} \hspace{1cm} 1000}}$$

The specified algorithm schedules the top event for a total weight of 1. The optimal schedule schedules the bottom event for a total weight of 1000.

**Greedy Largest Weight:** Another greedy algorithm selects the first event using the criteria of the largest weight  $w_i$ . The following is a counterexample for this.

$$\frac{2}{\underline{\hspace{1cm} \hspace{1cm} 1 \hspace{1cm} 1}}$$

The top event has weight 2 and the bottom ones each have weight 1. The specified algorithm schedules the top event for a total weight of 2. The optimal schedule schedules the bottom events for a total weight of 9.

**Dynamic Programming:** The obvious dynamic programming algorithm is for the little bird to tell you whether or not to schedule event  $J_n$ .

**Bird & Friend Algorithm:** Consider an instance  $J = \langle \langle s_1, f_1, w_1 \rangle, \langle s_2, f_2, w_2 \rangle, \dots, \langle s_n, f_n, w_n \rangle \rangle$ . The little bird considers an optimal schedule. We ask the little bird whether or not to schedule event  $J_n$ . If she says yes, then the remaining possible events to schedule are those in  $J$  excluding event  $J_n$  and excluding all events that conflict with event  $J_n$ . We ask a friend to schedule these. Our schedule is his with event  $J_n$  added. If instead the bird tells us not to schedule event  $J_n$ , then the remaining possible events to schedule are those in  $J$  excluding event  $J_n$ .

**The Set of Subinstances:** When tracing the recursive algorithm on small examples, we see that the set of subinstance used can be exponentially large. See the left side of Figure 24.2 for example in which this is the case. The events in the instance are paired so that for  $i \in [1.. \frac{n}{2}]$ , job  $J_i$  conflicts with job  $J_{\frac{n}{2}+i}$ , but jobs between pairs do not conflict. After the little bird tells you whether or not to schedule jobs  $J_{\frac{n}{2}+i}$  for  $i \in [1.. \frac{n}{2}]$ , job  $J_i$  will remain in the subinstance if and only if job  $J_{\frac{n}{2}+i}$  was not scheduled. This results in at least  $2^{n/2}$  different paths down the tree of stack frames in the recursive backtracking algorithm, each leading to a different subinstance. In contrast, look at the instance on the right of Figure 24.2. It only has a linear number of subinstances. The only difference between these examples is the order of the events. This example motivates the next algorithm.

**The Greedy Dynamic Programming:** First sort the events according to their finishing times  $f_i$  (a greedy thing to do). Then run the same dynamic programming algorithm in which the little bird tells you whether or not to schedule event  $J_n$ .

**The Set of Subinstances:** When tracing the recursive algorithm on small examples, it looks hopeful that the set of subinstance used is  $\{\langle \langle s_1, f_1, w_1 \rangle, \langle s_2, f_2, w_2 \rangle, \dots, \langle s_i, f_i, w_i \rangle \rangle \mid i \in [0..n]\}$ . If, so the algorithm is polynomial time. The danger is that when excluding those events that conflict with

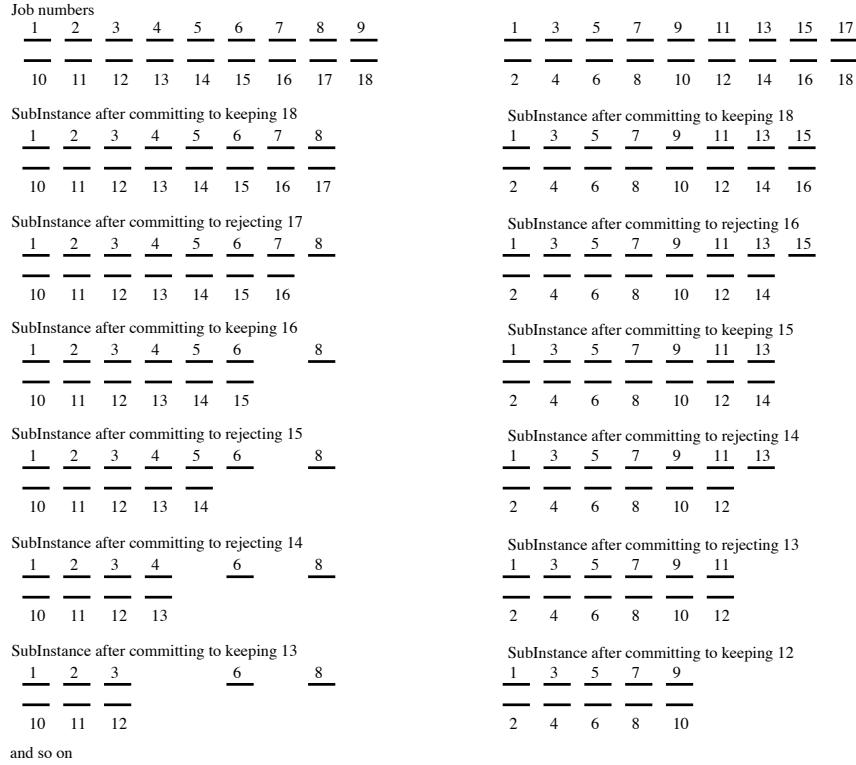


Figure 24.2: Two examples of the subinstances formed from the recursive backtracking algorithm. The first instance will have an exponential number of subinstances depending on how the bird answers because each subinstance is a subset of the events. The second instance will have a linear number of subinstances no matter how the bird answers because each subinstance is a prefix of the events.

event  $J_n$ , the subinstance created will no longer have a contiguous prefix of the events. For example, if event  $J_2$  conflicts with  $J_n$  but event  $J_3$  does not, then the new subinstance will need to be  $\langle \langle s_1, f_1, w_1 \rangle, \langle s_3, f_3, w_3 \rangle, \dots, ?? \rangle$ , which is not included. Needing to solve this subinstance as well may make the running time exponential.

**Closed:** For this algorithm, more than those previous, we must be careful to show that this set contains all subinstances generated by the recursive algorithm by showing that it closed under the sub-operator. Consider an arbitrary subinstance  $\langle \langle s_1, f_1, w_1 \rangle, \langle s_2, f_2, w_2 \rangle, \dots, \langle s_i, f_i, w_i \rangle \rangle$  in the set. If we delete from this event  $J_i$  and all events that conflict with it, we must show that this new subinstance is again in our set. Let  $i' \in [0..i - 1]$  be the largest index such that  $f_{i'} \leq s_i$ . Because the events have been sorted by their finishing time, we know that all events  $J_k$  in  $\langle \langle s_1, f_1, w_1 \rangle, \langle s_2, f_2, w_2 \rangle, \dots, \langle s_{i'}, f_{i'}, w_{i'} \rangle \rangle$  also have  $f_k \leq s_i$  and hence do not conflict with  $J_i$ . All events  $J_k$  in  $\langle \langle s_{i'+1}, f_{i'+1}, w_{i'+1} \rangle, \dots, \langle s_i, f_i, w_i \rangle \rangle$  have  $s_i < f_k \leq f_i$  and hence conflict with  $J_i$ . It follows that the resulting subinstance is  $\langle \langle s_1, f_1, w_1 \rangle, \langle s_2, f_2, w_2 \rangle, \dots, \langle s_{i'}, f_{i'}, w_{i'} \rangle \rangle$ , which is our set of subinstances. If only the other hand, only event  $J_i$  is deleted, then the resulting subinstance is  $\langle \langle s_1, f_1, w_1 \rangle, \dots, \langle s_{i-1}, f_{i-1}, w_{i-1} \rangle \rangle$ , which is obviously in our set of subinstances. It is because this works out, that the algorithm is polynomial time.

**Generating:** Consider the arbitrary subinstance  $\langle\langle s_1, f_1, w_1 \rangle, \langle s_2, f_2, w_2 \rangle, \dots, \langle s_i, f_i, w_i \rangle\rangle$ . It is generated by the recursive algorithm when the little bird states that none of the later events are included in the solution.

**The Table:** The dynamic programming table is a one dimensional array indexed by  $i \in [0..n]$ . The order to fill it in is with increasing  $i$ . As in the greedy algorithm, the events are being considered ordered by earliest finishing time first. The  $i$  entry is filled in by trying each of the two answers the bird might give.

**Time and Space Requirements:** Generally, the running time is the number of subinstances times the number of possible bird answers and the space is the number of subinstances. This would give  $T = \Theta(n \times 2)$  and  $S = \Theta(n)$ . However, in this case, the running time is larger than this. The reason is that when the event  $J_i$  is to be included it takes  $\mathcal{O}(\log n)$  time to do a binary search to find which earlier events conflict with it and hence need to be deleted. This gives a running time of  $\mathcal{O}(n \log n)$ , apart from the initial  $\mathcal{O}(n \log n)$  time sorting of the activities by finishing time.

**Exercise 24.4.1** Write out the pseudo code for this algorithm.

## 24.5 Exponential Time? The Integer-Knapsack Problem

Another example of an optimization problem is the *integer-knapsack problem*. For the problem in general, no polynomial algorithm is known. However, if the volume of the knapsack is a small integer, then dynamic programming provides a fast algorithm.



### Integer-Knapsack Problem:

**Instances:** An instance consists of  $\langle V, \langle v_1, p_1 \rangle, \dots, \langle v_n, p_n \rangle \rangle$ . Here,  $V$  is the total volume of the knapsack. There are  $n$  objects in a store. The volume of the  $i^{th}$  object is  $v_i$ , and its price is  $p_i$ .

**Solutions:** A solution is a subset  $S \subseteq [1..n]$  of the objects that fit into the knapsack, i.e.,  $\sum_{i \in S} v_i \leq V$ .

**Cost of Solution:** The cost (or success) of a solution  $S$  is the total value of what is put in the knapsack, i.e.,  $\sum_{i \in S} p_i$ .

**Goal:** Given a set of objects and the size of the knapsack, the goal is fill the knapsack with the greatest possible total price.

**Failed Greedy Algorithms:** Three obvious greedy algorithms take first the most valuable object,  $\max_i p_i$ , the smallest object,  $\min_i v_i$ , or the object with the highest value per volume,  $\max_i \frac{p_i}{v_i}$ . However, they do not work.

**Exercise 24.5.1** For each of this greedy algorithms, give an instance on which it does not work.

**The Question to Ask the Little Bird:** Consider a particular instance,  $\langle V, \langle v_1, p_1 \rangle, \dots, \langle v_n, p_n \rangle \rangle$  to the knapsack problem. The little bird might tell us whether or not an optimal solution for this instance contains the  $n^{th}$  item from the store. For each of these  $K = 2$  possible answers, the best solution is found that is consistent with this answer and then the best of these best solutions is returned.

**Reduced to Subinstance:** Either way, our search for an optimal packing is simplified. If an optimal solution does *not* contain the  $n^{th}$  item, then we simply delete this last item from consideration. This leaves us with the smaller instance,  $\langle V, \langle v_1, p_1 \rangle, \dots, \langle v_{n-1}, p_{n-1} \rangle \rangle$ . On the other hand, if an optimal solution *does* contain the  $n^{th}$  item, then we can take this last item and put it into the knapsack first. This leaves a volume of  $V - v_n$  in the knapsack. We determine how best to fill the rest of the knapsack with the remaining items by looking at the smaller instance  $\langle V - v_n, \langle v_1, p_1 \rangle, \dots, \langle v_{n-1}, p_{n-1} \rangle \rangle$ .

**The Set of Subinstances:** By tracing the recursive algorithm, we see that the set of subinstance used is  $\{\langle V', \langle v_1, p_1 \rangle, \dots, \langle v_i, p_i \rangle \rangle \mid V' \in [0..V], i \in [0..n]\}$ . Note that the items considered are a contiguous prefix of the original items, indicating a polytime algorithm. However, in addition we are considering every possible smaller knapsack size.

**Closed:** Applying the sub-operator to an arbitrary subinstance  $\langle V', \langle v_1, p_1 \rangle, \dots, \langle v_i, p_i \rangle \rangle$  from this set constructs subinstances  $\langle V', \langle v_1, p_1 \rangle, \dots, \langle v_i, p_{i-1} \rangle \rangle$  and  $\langle V' - v_i, \langle v_1, p_1 \rangle, \dots, \langle v_i, p_{i-1} \rangle \rangle$ , which are contained in the stated set of subinstances. Therefore, this set contains all subinstances generated by the recursive algorithm.

**Generating:** For some instances, these subinstances might not get called in the recursive program for every possible value of  $V'$ . However, as an exercise you could construct instances for which each such subinstance was called.

**Exercise 24.5.2** Construct an instance for which each of its subinstances are called.

**Constructing a Table Indexed by Subinstances:** The table indexed by the above set of subinstances will have a dimension for each of the parameters  $i$  and  $V'$  used to specify a particular subinstance. The tables will be  $cost[0..n, 0..V]$  and  $birdAdvice[0..V, 0..n]$ .

**Code:**

```

algorithm Knapsack ( $\langle V, \langle v_1, p_1 \rangle, \dots, \langle v_n, p_n \rangle \rangle$ )
<pre-cond>:  $V$  is the volume of the knapsack.  $v_i$  and  $p_i$  are the volume and the price of the  $i^{th}$  objects in a store.
<post-cond>:  $optSol$  is a way to fill the knapsack with the greatest possible total price.
 $optCost$  is its price.

begin
    % Table:  $optSol[V', i]$  would store an optimal way to fill a knapsack with volume  $V'$ 
    % with the first  $i$  objects,
    but actually we store only the bird's advice for the subinstance
    and the cost of its solution.

```

```

i^{th} item from the knapsack or (2) include it. Either way, we remove this last object, but in case (2) we decrease the size of the knapsack by the space needed for this item. Then we ask the friend for an optimal packing of the resulting subinstance. He gives us (1)  $optSol[V', i - 1]$  or (2)  $optSol[V' - v_i, i - 1]$  which he had stored in the table. If the bird had said we were to include the  $i^{th}$  item, then we add this item to the friend's solution. This gives us  $optSol_k$  which is a best packing for  $\langle V', i \rangle$  from amongst those consistent with the bird's answer.
        % Try each possible bird answers.
        % cases k = 1, 2 where 1=exclude 2=include
            % optSol1 = optSol[V', i - 1]
            cost1 = cost[V', i - 1]
            if(V' - vi ≥ 0) then
                % optSol2 = optSol[V' - vi, i - 1] ∪ i
                cost2 = cost[V' - vi, i - 1] + pi
            else
                % Bird was wrong
                % optSol2 = ?
                cost2 = -∞
            end if
        % end cases
        % Having the best,  $optSol_k$ , for each bird's answer  $k$ ,
        % we keep the best of these best.
        kmax = "a  $k$  that minimizes  $cost_k$ "
        % optSol[V', i] = optSolkmax
        cost[V', i] = costkmax
        birdAdvice[V', i] = kmax
    end for
end for
optSol = KnapsackWithAdvice (⟨V, ⟨v1, p1⟩, ..., ⟨vn, pn⟩⟩, birdAdvice)
return ⟨optSol, cost[V, n]⟩
end algorithm

```

### Constructing an Optimal Solution:

**algorithm**  $KnapsackWithAdvice(\langle V', \langle v_1, p_1 \rangle, \dots, \langle v_i, p_i \rangle \rangle, birdAdvice)$

**$\langle pre \& post-cond \rangle$ :** Same as  $Knapsack$  except with advice.

```

begin
    if ( $i = 0$ ) then
         $optSol = \emptyset$ 
        return  $optSol$ 
    end if
     $k_{max} = birdAdvice[V', i]$ 
    if  $k_{max} = 1$  then
         $optSubSol = KnapsackWithAdvice(\langle V', \langle v_1, p_1 \rangle, \dots, \langle v_{i-1}, p_{i-1} \rangle \rangle, birdAdvice)$ 
         $optSol = optSubSol$ 
    else
         $optSubSol = KnapsackWithAdvice(\langle V' - v_i, \langle v_1, p_1 \rangle, \dots, \langle v_{i-1}, p_{i-1} \rangle \rangle, birdAdvice)$ 
         $optSol = optSubSol \cup i$ 
    end if
    return  $optSol$ 
end algorithm

```

**Time and Space Requirements:** The number of subinstances is  $\Theta(V \cdot n)$  and the bird chooses between two options: to include or not to include the object. Hence, the running and the space requirements are both  $\Theta(V \cdot n)$ . However, this running time should be expressed as a function of input size. The number of bits needed to represent the instance  $\langle V', \langle v_1, p_1 \rangle, \dots, \langle v_n, p_n \rangle \rangle$  is  $N = |V| + n \cdot (|v| + |p|)$ , where  $|V|$ ,  $|v|$ , and  $|p|$  are the the numbers of bits needed to represent  $V$ ,  $v_i$ , and  $p_i$ . Expressed in these terms, the running time is  $T(|\text{instance}|) = \Theta(nV) = \Theta(n2^{|V|})$ . This is quicker than the brute force algorithm because its running time is polynomial in the number of items  $n$ . In the worst case, however,  $V$  is large and the time can be exponential in the number of bits  $N$ . I.e., if  $|V| = \Theta(N)$ , then  $T = \Theta(2^N)$ . In fact, the knapsack problem is one of the classic NP complete problems, which means that it is generally believed that not polynomial time algorithm exists for it.

**Exercise 24.5.3** *The fractional-knapsack problem is the same as the integer-knapsack problem except that there is a given amount of each object and any fractional amount of each it can be put into the knapsack. Develop a quick greedy algorithm for this problem.*

## 24.6 The Solution Viewed as a Tree: Chains of Matrix Multiplications

The next example will be our first example in which the fields of information specifying a solution are organized into a tree instead of into a sequence. See Section 23.4.2. The algorithm asks the little bird to tell it the field at the root of one of the instance's optimal solutions and then a separate friend will be asked for each of the solution's subtrees.

The optimization problem determines how to optimally multiplying together a chain of matrices. Multiplying an  $a_1 \times a_2$  matrix by a  $a_2 \times a_3$  matrix requires  $a_1 \cdot a_2 \cdot a_3$  scalar multiplications.

Matrix multiplication is commutative, meaning that  $(M_1 \cdot M_2) \cdot M_3 = M_1 \cdot (M_2 \cdot M_3)$ . Sometimes different bracketing of a sequence of matrix multiplications can lead to the total number of scalar multiplications being very different. For example,

$$(\langle 5 \times 1,000 \rangle \cdot \langle 1,000 \times 2 \rangle) \cdot \langle 2 \times 2,000 \rangle = \langle 5 \times 2 \rangle \cdot \langle 2 \times 2,000 \rangle = \langle 5 \times 2,000 \rangle$$

requires  $5 \times 1,000 \times 2 + 5 \times 2 \times 2,000 = 10,000 + 20,000 = 30,000$  scalar multiplications. However,

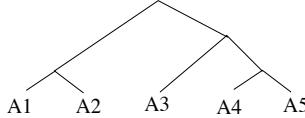
$$\langle 5 \times 1,000 \rangle \cdot (\langle 1,000 \times 2 \rangle \cdot \langle 2 \times 2,000 \rangle) = \langle 5 \times 1,000 \rangle \cdot \langle 1,000 \times 2,000 \rangle = \langle 5 \times 2,000 \rangle$$

requires  $1,000 \times 2 \times 2,000 + 5 \times 1,000 \times 2,000 = 4,000,000 + 10,000,000 = 14,000,000$ . The problem considered here is to find how to bracket a sequence of matrix multiplications in order to minimize the number of scalar multiplications.

### Chains of Matrix Multiplications:

**Instances:** An instance is a sequence of  $n$  matrices  $\langle A_1, A_2, \dots, A_n \rangle$ . (A precondition is that for each  $k \in [1..n - 1]$   $\text{width}(A_k) = \text{height}(A_{k+1})$ .)

**Solutions:** A solution is a way of bracketing the matrices, e.g.,  $((A_1 A_2)(A_3(A_4 A_5)))$ . A solution can equivalently be viewed as a binary tree with the matrices  $A_1, \dots, A_n$  at the leaves. The binary tree would give the order in which to multiply the matrices.



**Cost of Solution:** The cost of a solution is the number of scalar multiplications needed to multiply the matrices according to the bracketing.

**Goal:** Given a sequence of matrices, the goal is to find a bracketing that requires the fewest multiplications.

**A Failed Greedy Algorithm:** An obvious greedy algorithm selects where the last multiplication will occur according the criteria of which is cheapest. We can prove that any such simple greedy algorithm will fail, even when the instance contains only three matrices. Let the matrices  $A_1, A_2$ , and  $A_3$  have height and width  $\langle a_0, a_1 \rangle, \langle a_1, a_2 \rangle$ , and  $\langle a_2, a_3 \rangle$ . There are two orders in which these can be multiplied. Their costs are as follows.

$$\text{cost}((A_1 \cdot A_2) \cdot A_3) = a_0 a_1 a_2 + a_0 a_2 a_3$$

$$\text{cost}(A_1 \cdot (A_2 \cdot A_3)) = a_1 a_2 a_3 + a_0 a_1 a_3$$

Consider the algorithm that uses the method whose last multiplication is the cheapest. Let us assume that the algorithm uses the first method. This gives that

$$\text{a)} a_0 a_2 a_3 < a_0 a_1 a_3$$

However, we want the algorithm to give the wrong answer. Hence, we want the second method to be the cheapest. This gives that

$$\text{b)} a_0 a_1 a_2 + a_0 a_2 a_3 > a_1 a_2 a_3 + a_0 a_1 a_3$$

Simplifying line (a) gives  $a_2 < a_1$ . Plugging line (a) into line (b) gives  $a_0 a_1 a_2 >> a_1 a_2 a_3$ . Simplifying this gives  $a_0 >> a_3$ . Let us now assign simple values meeting  $a_2 < a_1$  and  $a_0 >> a_3$ . Say  $a_0 = 1000$ ,  $a_1 = 2$ ,  $a_2 = 1$ , and  $a_3 = 1$ . Plugging these in gives

$$\text{cost}((A_1 \cdot A_2) \cdot A_3) = 1000 \cdot 2 \cdot 1 + 1000 \cdot 1 \cdot 1 = 2000 + 1000 = 3000$$

$$\text{cost}(A_1 \cdot (A_2 \cdot A_3)) = 2 \cdot 1 \cdot 1 + 1000 \cdot 2 \cdot 1 = 2 + 2000 = 2002$$

This is an instance in which the algorithm gives the wrong answer. Because  $1000 < 2000$  it uses the first method. However, the second method is cheaper.

**Exercise 24.6.1** Give the steps in the technique above to find a counter example for the greedy algorithm that multiplies the cheapest pair together first.

**A Failed Dynamic Programming Algorithm:** An obvious question to ask the little bird would be which pair of consecutive matrices to multiply together first. Though this algorithm works, it has exponential running time. The problem is that if you trace the execution of the recursive algorithm, there is an exponential number of different subinstances. Consider paths down the tree of stack frames in which for each pair  $A_{2i}$  and  $A_{2i+1}$ , the bird either gets us to multiply them together or does not. This results in  $2^{n/2}$  different paths down the tree of stack frames in the recursive backtracking algorithm, each leading to a different subinstance.

**The Question to Ask the Little Bird:** A better question is to ask the little bird to give us the splitting  $k$  so that the last multiplication multiplies the product of  $\langle A_1, A_2, \dots, A_k \rangle$  and of  $\langle A_{k+1}, \dots, A_n \rangle$ . This is equivalent to asking for the root of the binary tree. For each of the possible answers, the best solution is found that is consistent with this answer and then the best of these best solutions is returned.

**Reduced to Subinstance:** With this advice, our search for an optimal bracketing is simplified. We need only solve two subinstances: finding an optimal bracketing of  $\langle A_1, A_2, \dots, A_k \rangle$  and of  $\langle A_{k+1}, \dots, A_n \rangle$ .

**Recursive Structure:** An optimal bracketing of the matrices  $\langle A_1, A_2, \dots, A_n \rangle$  multiplies the sequence  $\langle A_1, \dots, A_k \rangle$  with its optimal bracketing, multiplies  $\langle A_{k+1}, \dots, A_n \rangle$  with its optimal bracketing, and then multiplies these two resulting matrices together, i.e.,  $optSol = (optLeft)(optRight)$ .

**The Cost of the Optimal Solution Derived from the Cost for Subinstances:** The total number of scalar multiplications used in this optimal bracketing is the number used to multiply  $\langle A_1, \dots, A_k \rangle$ , plus the number for  $\langle A_{k+1}, \dots, A_n \rangle$ , plus the number to multiply the final two matrices.  $\langle A_1, \dots, A_k \rangle$  evaluates to a matrix whose height is the same as that of  $A_1$  and whose width is that of  $A_k$ . Similarly,  $\langle A_{k+1}, \dots, A_n \rangle$  becomes a  $height(A_{k+1}) \times width(A_n)$  matrix. (Note that  $width(A_k) = height(A_{k+1})$ .) Multiplying these requires  $height(A_1) \times width(A_k) \times width(A_n)$  number of scalar multiplications. Hence, in total,  $cost = costLeft + costRight + height(A_1) \times width(A_k) \times width(A_n)$ .

**The Set of Subinstances Called:** The set of subinstances of the instance  $\langle A_1, A_2, \dots, A_n \rangle$  is  $\langle A_i, A_{i+1}, \dots, A_j \rangle$  for every choice of end points  $1 \leq i \leq j \leq n$ . This set of subinstances contains all the subinstances called, because it is closed under the sub-operator. Applying the sub-operator to an arbitrary subinstance  $\langle A_i, A_{i+1}, \dots, A_j \rangle$  from this set constructs subinstances  $\langle A_i, \dots, A_k \rangle$  and  $\langle A_{k+1}, \dots, A_j \rangle$  for  $i \leq k < j$ , which are contained in the stated set of subinstances. Similarly, the set does not contain subinstances *not* called by the recursive program, because we easily can construct any arbitrary subinstance in the set with the sub-operator. For example,  $\langle A_1, \dots, A_n \rangle$  sets  $k = j$  and calls  $\langle A_1, \dots, A_j \rangle$ , which sets  $k = i + 1$  and calls  $\langle A_i, \dots, A_j \rangle$ .

**Constructing a Table Indexed by Subinstances:** The table indexed by the above set of subinstances will have a dimension for each of the parameters  $i$  and  $j$  used to specify a particular subinstance. The tables will be  $cost[1..n, 1..n]$  and  $birdAdvice[1..n, 1..n]$ . See Figure 24.3.

**Order in which to Fill the Table:** The size of a subinstance is the number of matrices in it. We will fill the table in this order.

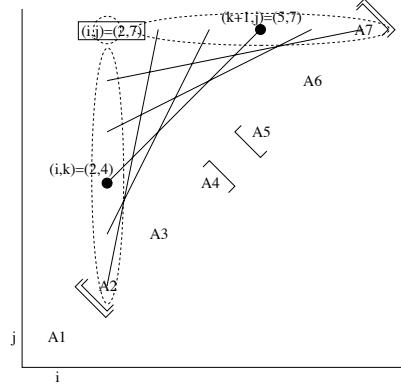


Figure 24.3: The table produced by the dynamic-programming solution for Choose. When searching for the optimal bracketing of  $A_2, \dots, A_7$ , one of the methods to consider is  $[A_2, \dots, A_4][A_5, \dots, A_7]$ .

**Exercise 24.6.2** (See solution in Section V) Use a picture to make sure that when  $\langle i, j \rangle$  is filled,  $\langle i, k \rangle$  and  $\langle k + 1, j \rangle$  are already filled for all  $i \leq k < j$ . Give two other orders that work.

**algorithm** *MatrixMultiplication* ( $\langle A_1, A_2, \dots, A_n \rangle$ )

***pre-cond***: An instance is a sequence of  $n$  matrices.

***post-cond***: *optSol* is a bracketing that requires the fewest multiplications and *optCost* is this number.

```

begin
    % Table: optSol[i, j] would store an optimal way of bracketing the matrices
    %  $\langle A_i, A_{i+1}, \dots, A_j \rangle$ ,
    % but actually we store only the bird's advice for the subinstance
    % and the cost of its solution.
    table[1..n, 1..n] birdAdvice, cost

    % Base Cases: The base cases are when there is only one matrix, i.e.  $\langle A_i \rangle$ .
    For each, the solution is the empty bracketing with cost zero.
    for  $i = 1$  to  $n$ 
        % optSol[i, i] =  $A_i$ 
        cost[i, i] = 0
        birdAdvice[i, i] =  $\emptyset$ 
    end for

    % General Cases: Loop over subinstances in the table.
    for size = 2 to  $n$ 
        for  $i = 1$  to  $n - size + 1$ 
            j= i+size-1
            % Solve instance  $\langle i, j \rangle$  and fill in the table.
            % Try each possible bird answers.
            for  $k = i$  to  $j - 1$ 

```

```

% The bird and Friend Alg: The bird give us the splitting  $k$  so that
% the last multiplication multiplies the product of  $\langle A_i, A_{i+1}, \dots, A_k \rangle$ 
% and of  $\langle A_{k+1}, \dots, A_j \rangle$ . One friend gives us  $optSol[i, k]$ , an optimal
% bracketing of  $\langle A_i, \dots, A_k \rangle$  and another gives us  $optSol[k + 1, j]$ , an
% optimal bracketing of  $\langle A_{k+1}, \dots, A_j \rangle$ . We combine these friends
% and bird's information giving us  $optSol_k$  which is a best bracketing
% for  $\langle i, j \rangle$  from amongst those consistent with the bird's answer.
% Get help from friend
%  $optSol_k = ( optLeft ) ( optRight )$ 
cost $k$  = cost[i, k] + cost[k + 1, j] + height(Ai) × width(Ak) × width(Aj)
end for
% Having the best,  $optSol_k$ , for each bird's answer  $k$ ,
we keep the best of these best.
kmin = "a  $k$  that minimizes cost $k$ "
% optSol[i, j] = optSolkmin
cost[i, j] = costkmin
birdAdvice[i, j] = kmin
end for
end for
optSol = MatrixMultiplicationWithAdvice ( $\langle A_1, A_2, \dots, A_n \rangle$ , birdAdvice)
return  $\langle optSol, cost[1, n] \rangle$ 
end algorithm

```

### Constructing an Optimal Solution:

```

algorithm MatrixMultiplicationWithAdvice ( $\langle A_i, A_2, \dots, A_j \rangle$ , birdAdvice)
<pre & post-cond>: Same as MatrixMultiplication except with advice.

```

```

begin
if ( $i = j$ ) then
    optSol = Ai
    return optSol
end if
kmin = birdAdvice[i, j]
optLeft = MatrixMultiplicationWithAdvice ( $\langle A_1, \dots, A_{k_{min}} \rangle$ , birdAdvice)
optRight = MatrixMultiplicationWithAdvice ( $\langle A_{k_{min}+1}, \dots, A_j \rangle$ , birdAdvice)
optSol = ( optLeft ) ( optRight )
return optSol
end algorithm

```

**Time and Space Requirements:** The running time is the number of subinstances times the number of possible bird answers and the space is the number of subinstances. The number of subinstances is  $\Theta(n^2)$  and the bird chooses one of  $\Theta(n)$  places to split the sequence of matrices. Hence, the running time is  $\Theta(n^3)$  and the space requirements are  $\Theta(n^2)$ .

## 24.7 Generalizing the Problem Solved: Best AVL Tree

As discussed in Section 14.3, it is sometimes useful to generalize the problem solved so that you can either give or receive more information from your friend in a recursive algorithm. This was demonstrated in Section 16 with a recursive algorithm for determining whether or not a tree is an AVL tree. This same idea is useful for dynamic programming. We will now demonstrate this by giving an algorithm for finding the best AVL tree. To begin, we will develop the algorithm for the *Best Binary Search Tree* Problem introduced at the beginning of this chapter.

### The Best Binary Search Tree Problem:

**Instances:** An instance consists of  $n$  probabilities  $p_1, \dots, p_n$  to be associated with the  $n$  keys  $a_1 < a_2 < \dots < a_n$ . The values of the keys themselves do not matter. Hence, can assume that  $a_i = i$ .

**Solutions:** A solution for an instance is a binary search tree containing the keys. A binary search tree is a binary tree such that the nodes are labeled with the keys and for each node all the keys in its left subtree are smaller and all those in the right are larger.

**Cost of Solution:** The cost of a solution is the expected depth of a key when choosing a key according to the given probabilities, namely  $\sum_{i \in [1..n]} [p_i \cdot \text{depth of } a_i \text{ in tree}]$ .

**Goal:** Given the keys and the probabilities, the goal is to find a binary search tree with minimum expected depth.

**Expected Depth:** The time required to searching for a key is proportional to the depth of the key in the binary search tree. Finding the root is fast. Finding a deep leaf takes much longer. The goal is to design the search tree so that the keys that are searched for often are closer to the root. The probabilities  $p_1, \dots, p_n$  given as part of the input specify the frequency with which each key is searched for, eg.  $p_3 = \frac{1}{8}$  means that key  $a_3$  is search for on average one out of every eight times.

One minimizes the depth of a binary search tree by having it be completely balanced. Having it balanced, however, dictates the location of each key. Although having the tree partially unbalanced increases its overall height, it may allow for the keys that are searched for often to be placed closer to the top.

We will manage to put some of the nodes close to the root and others we will not. The standard mathematical way of measuring the overall successes of putting more likely keys closer to the top is the expected depth of a key when the key is chosen randomly according to the given probability distribution. It is calculated by  $\sum_{i \in [1..n]} p_i \cdot d_i$ , where  $d_i$  is the depth of  $a_i$  in the search tree.

One way to understand this is to suppose that we needed to search for a billion keys. If  $p_3 = \frac{1}{8}$ ,  $a_3$  is searched for on average one out of every eight times. Because we are searching for so many keys, it is almost certain that the number of times we search for this key is very close to  $\frac{1}{8}$  billion. In general, the number of times we search for  $a_i$  is  $p_i \times$  billion. To compute the average depth of these billion searches, we sum up the depth of each them and divide by a billion, namely  $\frac{1}{\text{billion}} \sum_{k \in [1..\text{billion}]} [\text{depth of } k^{\text{th}} \text{ search}] = \frac{1}{\text{billion}} \sum_{i \in [1..n]} (p_i \times \text{billion}) \cdot d_i = \sum_{i \in [1..n]} p_i \cdot d_i$ .

**Bird and Friend Algorithm:** I am given an instance consisting of  $n$  probabilities  $p_1, \dots, p_n$ . I ask the bird which key to put at the root. She answer  $a_k$ . I ask one friend for the best binary

search tree for the keys  $a_1, \dots, a_{k-1}$  and its expected depth. I ask another friend for the best tree of the specified height for  $a_{k+1}, \dots, a_n$  and its expected depth. I build the tree with  $a_k$  at the root and these as the left and right subtrees.

**Generalizing the Problem Solved:** A set of probabilities  $p_1, \dots, p_n$  defining a probability distribution should have the property that  $\sum_{i \in [1..n]} p_i = 1$ . However, we will generalize the problem by removing this restriction. This will allow us to ask our friend to solve subinstances that are not officially legal. Note that the probabilities given to the friends in the above algorithm do not sum to 1.

**The Cost of an Optimal Solution Derived from the Costs for the Subinstances:** The expected depth of my tree is computed from that given by my friend as follows.

$$Cost = \sum_{i \in [1..n]} [p_i \cdot \text{depth of } a_i \text{ in tree}] \quad (24.1)$$

$$= \sum_{i \in [1..k-1]} [p_i \cdot (\text{depth of } a_i \text{ in left subtree}) + 1] + [p_k \cdot 1] \quad (24.2)$$

$$+ \sum_{i \in [k+1..n]} [p_i \cdot (\text{depth of } a_i \text{ in right subtree}) + 1] \quad (24.3)$$

$$= Cost_{left} + \left[ \sum_{i \in [1..k-1]} p_i \right] + p_k + Cost_{right} + \left[ \sum_{i \in [k+1..n]} p_i \right] \quad (24.4)$$

$$= Cost_{left} + \left[ \sum_{i \in [1..n]} p_i \right] + Cost_{right} \quad (24.5)$$

$$= Cost_{left} + Cost_{right} + 1 \quad (24.6)$$

**The Complete Set of Subinstances that Will Get Called:** The complete set of subinstances is  $S = \{\langle a_i, \dots, a_j; p_i, \dots, p_j \rangle \mid 1 \leq i \leq j \leq n\}$ . The table is 2-dimensional  $\Theta(n \times n)$ .

**Running Time:** The table has size  $\Theta(n \times n)$ . The bird can give  $n$  different answers. Hence, the time is  $\Theta(n^3)$ .

We now change the above problem so that it is looking for the best AVL Search Tree.

### The Best AVL Tree Problem:

**Instances:** An instance consists of  $n$  probabilities  $p_1, \dots, p_n$  to be associated with the  $n$  keys  $a_1 < a_2 < \dots < a_n$ .

**Solutions:** A solution for an instance is an AVL tree containing the keys. An AVL tree is a binary search tree with the property that every node has a balance factor of -1, 0, or 1, where its balance factor is the difference between the height of its left and its right subtrees.

**Cost of Solution:** The cost of a solution is the expected depth of a key  $\sum_{i \in [1..n]} [p_i \cdot \text{depth of } a_i \text{ in } T]$ .

**Goal:** Given the keys and the probabilities, the goal is to find an AVL tree with minimum expected depth.

**Generalizing the Problem Solved:** Recall Section 16 we wrote a recursive program to determine whether a tree is an AVL tree. We needed to know the height of the left and the right subtree. Therefore, we generalized the problem so that it also returned the height. We will do a similar thing here.

**Insuring the Balance Between Heights of Left and Right SubTrees:** Let us begin by trying the algorithm for the general binary search tree. We ask the bird which key to put at the root. She answer  $a_k$ . We ask friends to build the left and the right sub-AVL trees. They could even tell us how high they are. What do we do, however, if the difference in their heights is greater than one?

We would like to ask the friends to build their AVL trees so that the difference in their heights is at most one. This, however, requires the friends to coordinate. This would be hard.

Another option is to ask the bird what height the left and right subtree should be. The bird will give you heights that are within one of each other. Then we could separately ask each friend to give the best AVL tree of the given height.

**The New Problem:** An instance consists of the keys, the probabilities, and a required height. The goal is to find the best AVL tree with the given height.

**Bird and Friend Algorithm:** An AVL tree of height  $h$  has left and right subtrees of heights either  $\langle h - 2, h - 1 \rangle$ ,  $\langle h - 1, h - 1 \rangle$ , or  $\langle h - 1, h - 2 \rangle$ . The bird tells me which of these three is the case and which value  $a_k$  will be at the root. I can then ask the friends for the best left and right subtrees of the specified height.

**The Complete Set of Subinstances that Will Get Called:** Recall in Section 16, we proved that the minimum height of an AVL tree with  $n$  nodes is  $h = \log_2 n$  and that its maximum height is  $h = 1.455 \log_2 n$ . Hence, the complete set of subinstances is  $S = \{\langle h; a_i, \dots, a_j; p_i, \dots, p_j \rangle \mid 1 \leq i \leq j \leq n, h \in [\log_2(j - i + 1)..1.455 \log_2(j - i + 1)]\}$ . The table is a 3-dimensional  $\Theta(n \times n \times \log n)$  box.

**Running Time:** The table has size  $\Theta(n \times n \times \log n)$ . The bird can give  $3 \cdot n$  different answers. Hence, the time is  $\Theta(n^3 \log n)$ .

**Solving the Original Problem:** In the original problem, the height was not fixed. To solve this problem, we could simply run the previous algorithm for each  $h$  and take the best of these AVL trees. There is a faster way.

**Reusing the Table:** We do not need to run the previous algorithm more than once. After running it once, the table already contains the cost of the best AVL for each of the possible heights  $h$ . To find the best overall AVL tree, we need only compare these listed in the table.

## 24.8 All Pairs Shortest Paths with Negative Cycles

The algorithm given here has three interesting features: 1) The question asked of the little bird is interesting; 2) Instead of organizing each solution into a sequence of answers or into a binary tree of answers, something in between is done; 3) It stores something other than the bird's advice in the table. This algorithm is due to Floyd-Warshall and Johnson.

**All Pairs Shortest Weighted Paths with Possible Negative Cycles:** Like the problem solved by Dijkstra's algorithm in Section 20.3 and by dynamic programming for leveled graphs in Section 23.2, the problem is to find a shortest-weighted path between two nodes. Now, however, we consider the possibility of cycles with negative weights and hence the

possibility of optimal solutions that are infinitely long paths with infinitely negative weight. Also the problem is extended to ask for the shortest path between every pair of nodes.

**Instances:** An instance (input) consists a weighted graph  $G$  (either directed or undirected).

Each edge  $\langle u, v \rangle$  is allocated with a possibly negative weight  $w_{\langle u, v \rangle} \in [-\infty, \infty]$ .

**Solutions:** A solution consists of a data structure,  $\langle \pi, cycleNode \rangle$ , that specifies a path from  $v_i$  to  $v_j$  for every pair of nodes.

**Cost of Solution:** The cost of a path is the sum of the weights of the edges within the path.

**Goal:** Given a graph  $G$ , the goal is to find, for each pair of nodes, a path with minimum total weight between them.

**Types of Paths:** Here we discuss the types of paths that arise within this algorithm and how they are stored.

**Negative Cycles:** As an example, find a minimum weighted path from  $v_i$  to  $v_j$  in the graph Figure 24.4. The path  $\langle v_i, v_c, v_d, v_e, v_j \rangle$  has weight  $1+2+2+1 = 6$ . The path  $\langle v_i, v_a, v_b, v_j \rangle$  has weight  $1 + 2 + 4 = 7$ . This is not as good as the previous. However, we can take a detour in this path by going around the cycle, giving the path  $\langle v_i, v_a, v_b, v_m, v_a, v_b, v_j \rangle$ . Because the cycle has negative weight, the new weight is better,  $1+2+(3+(-6)+2)+4 = 6$ . That working well, why not go around the cycle again, bring the weight down to 5. The more we go around, the better the path. In fact, there is no reason that we cannot go around an infinite number of times, producing a path with weight  $-\infty$ . We have no requirement that the paths have a finite number of edges, so this is a legal path. There may or may not be other paths with  $-\infty$  weight, but none can beat this. Hence this must be a path with minimum weight.

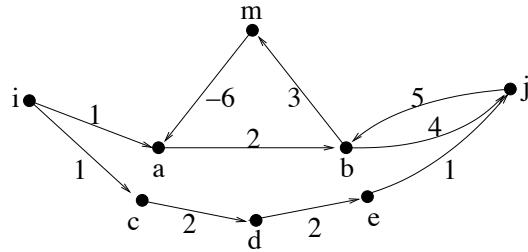


Figure 24.4: An infinite path around a negative cycle.

**Definition of Simple Paths:** A path is said to be *simple* if no node is repeated.

**The  $\pi$  All Paths Data Structure:** The problem asks for an optimal path between each pair of nodes  $v_i$  and  $v_j$ . Section 20.2 introduces a good data structure,  $\pi[j]$ , for storing an optimal path from one fixed node  $s$  to all nodes  $v_j$ . It can be modified to store an optimal path between every pair of nodes. Here  $\pi[i, j]$  stores the second last node in the path from  $v_i$  to  $v_j$ . A problem, however, with this data structure is that it is only able to store simple paths.

**Simple Paths:** Consider the simple path in Figure 24.5. The entire path is defined recursively to be the path from  $v_i$  to the node given as  $\pi[i, j]$ , followed by the last edge  $\langle \pi[i, j], v_j \rangle$  to  $v_j$ . More concretely, the nodes in the path walking backwards

from  $v_j$  to  $v_i$  are  $v_j$ ,  $\pi[i, j]$ ,  $\pi[i, \pi[i, j]]$ ,  $\pi[i, \pi[\pi[i, j]]]$ ,  $\dots$ , until the node  $v_i$  is reached.

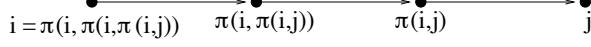


Figure 24.5: The  $\pi$  all paths data structure for simple paths.

**Code for Walking Backwards:** The following code, traces the path backwards from  $v_j$  to  $v_i$ .

**algorithm** *PrintSimplePath* ( $i, j, \pi$ )

***⟨pre-cond⟩:***  $\pi$  specifies a path between every pair of nodes.

***⟨post-cond⟩:*** Prints the path from  $v_i$  to  $v_j$  backwards.

begin

```

    walk = j
    print  $v_j$ 
    while( walk  $\neq i$  )
        walk =  $\pi[i, walk]$ 
        print  $v_{walk}$ 
    end while
end algorithm

```

**Cannot Store Paths with Cycles:** The data structure  $\pi$  is unable to store paths with cycles for the following reason. Suppose that the path from  $v_i$  to  $v_j$  contains the node  $v_m$  more than once.  $\pi$  gives how to back up from  $v_j$  along this path. After reaching  $v_m$ ,  $\pi$  gives how to back up further until  $v_m$  is reached a second time. Backing up further,  $\pi$  will follow the same cycle back to  $v_m$  over and over again.  $\pi$  is unable to describe how to deviate off this cycle in order continue on to  $v_i$ .

**Storing Optimal Paths:** For each pair of nodes  $v_i$  and  $v_j$ , an optimal path is stored.

**Finite Paths:** If the minimum weight of a path from  $v_i$  to  $v_j$  is not  $-\infty$ , then we know that there is an optimal path that is simple, because if a cycle has a positive weight it should not be taken and if it has a negative weight it should be taken again. Such a simple path is traced backwards by  $\pi$ .

**Infinite Paths:** If the minimum weight of a path from  $v_i$  to  $v_j$  is  $-\infty$ , then  $\pi$  cannot trace this path backwards, because it contains a negative weighted cycle. Instead, an infinite optimal path from  $v_i$  to  $v_j$  is stored as follows.

**Node on Cycle  $cycleNode[i, j]$ :** The data structure  $cycleNode[i, j]$  will store one node  $v_m$  on one such negative cycle along the path.

**The Path in Three Segments:** The path from  $v_i$  to  $v_j$  that will be stored by the algorithm follows a simple path from  $v_i$  to  $v_m$ , followed an infinite number of times by a negative weighted simple path from  $v_m$  back to  $v_m$ , ending with a simple path from  $v_m$  to  $v_j$ . For the example in Figure 24.4, these paths are  $\langle v_i, v_a, v_b, v_m, \rangle$ ,  $\langle v_m, v_a, v_b, v_m \rangle$ , and  $\langle v_m, v_a, v_b, v_j \rangle$ . Each of these three simple paths is stored by being traced backwards by  $\pi$ . More concretely, the infinite path from  $v_i$  to  $v_j$  is traced backwards as follows.

$$v_j, \pi[m, j], \pi[m, \pi[m, j]], \pi[m, \pi[m, \pi[m, j]]], \dots, v_m$$

```

infinite cycle begins
 $\pi[m, m], \pi[m, \pi[m, m]], \pi[m, \pi[m, \pi[m, m]]], \dots, v_m$ 
infinite cycle ends
 $\pi[i, m], \pi[i, \pi[i, m]], \pi[i, \pi[i, \pi[i, m]]], \dots, v_i$ 

```

The following code traces this path.

```

algorithm PrintPath ( $i, j, \pi, cost, cycleNode$ )
<pre-cond>:  $\pi$ ,  $cost$ , and  $cycleNode$  are as given by the
    AllPairsShortestPaths algorithm.  $v_i$  and  $v_j$  are two nodes.
<post-cond>: Prints the path from  $v_i$  to  $v_j$  backwards.
begin
    if(  $cost[i, j] \neq -\infty$  ) then
        PrintSimplePath ( $i, j, \pi$ )
    else
        m = cycleNode[i,j]
        PrintSimplePath ( $m, j, \pi$ )
        print “infinite cycle begins”
        PrintSimplePath ( $m, \pi[m, m], \pi$ )
        print “infinite cycle ends”
        PrintSimplePath ( $i, \pi[i, m], \pi$ )
    end if
end algorithm

```

**Additional Simple Paths:** The infinite path from  $v_i$  to  $v_j$  required three simple paths, from  $v_i$  to  $v_m$ ,  $v_m$  to  $v_m$ , and  $v_m$  to  $v_j$ . These paths need not be optimal, but they do need to be simple.

**Complications in Find Simple Paths:** Having negative cycles around, greatly complicates the algorithm’s task of ensuring that the paths traced back by  $\pi$  are in fact simple paths. For example, if one were looking for a minimal weighted simple path from  $v_i$  to  $v_m$ , one would be tempted to include the negative cycle from  $v_m$  back to  $v_m$ . If it was included, however, the path would not be simple and  $\pi$  would not be able to trace backwards from  $v_m$  to  $v_i$ .

**Non-Optimal Simple Paths:** Luckily, there is no requirement on these three simple paths being optimal in any way. After all, the combined path, having weight  $-\infty$ , must be a path with minimum total weight. Therefore, to simplify the algorithm, the algorithm’s only goal is to store some simple path in each of these three cases.

**A Simple Path from  $v_i$  to  $v_j$ :** Just as the infinite path from  $v_i$  to  $v_j$  required three simple paths,  $v_i$  to  $v_m$ ,  $v_m$  to  $v_m$ , and  $v_m$  to  $v_j$ , other pairs of nodes may require a simple path from  $v_i$  to  $v_j$ . Just in case it is needed, the algorithm will ensure that  $\pi$  traces some non-optimal simple path backwards from  $v_j$  to  $v_i$ . In order to avoid the negative cycle, this path from will avoid the node  $v_m$ . For the example in Figure 24.4, the path stored will be  $\langle v_i, v_c, v_d, v_e, v_j \rangle$ . This is accomplished by having  $\pi[i, j] = v_e$ , while  $\pi[m, j] = v_b$ .

This completes the discussion of the types of paths that arise within this algorithm and how they are stored.

### The Question For the Little Bird:

**The Last Edge:** The algorithm in Section 23.2 asked the little bird for the last edge in an optimal path from  $v_i$  to  $v_j$  (equivalently, we could ask for the second last node  $\pi[i, j]$ ) and then deleted this edge from consideration. One problem with this approach is that the optimal path may be infinitely long and deleting one edge does not make it significantly shorter. Hence, no “progress” is made. Another problem is that this edge may appear many times within the optimal solution. Hence, we cannot stop considering it so quickly.

**Number of Times Node  $v_n$  Appears:** Section 23.4.1 suggests that if the instance is a sequence of objects and a solution is a subset of these object, than a good question may be whether the last object of the instance is included in an optimal solution? Here an instance is a set of nodes, but we can order them in some arbitrary way. Then we can ask how many times the node  $v_n$  is included internally within an optimal path from  $v_i$  to  $v_j$ . There are only  $K = 3$  possible answers:  $k = 0, 1$ , and  $\infty$ . Note that  $v_n$  will not be included in the path only twice, because if this cycle from  $v_n$  back to  $v_n$  has a positive weight, why take the cycle, and if the cycle has a negative weight, why not go around again. For each of these  $K = 3$  possible answers that the little bird may give, the best solution is found that is consistent with this answer and then the best of these best solutions is returned.

**Recursive Structure:** An optimal path from  $v_i$  to  $v_j$  that contains the node  $v_n$  only once consists of an optimal path from  $v_i$  to  $v_n$  that includes node  $v_n$  only at the end followed by an optimal one from  $v_n$  to  $v_j$  that includes node  $v_n$  only at the beginning.

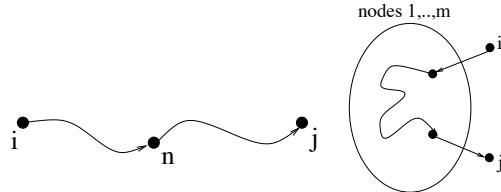


Figure 24.6: Generalizing the problem

**Generalizing the Problem:** This recursive structure motivates wanting to search for a shortest path between two specified nodes that excludes a specified set of nodes except possibly at the beginning and at the end of the path. By tracing the recursive algorithm, we will see that the set of subinstance used is  $\{\langle G_m, i, j \rangle \mid i, j \in [1..n] \text{ and } m \in [0..n]\}$ , where the goal of the instance  $\langle G_m, i, j \rangle$  is to find a path with minimum total weight from  $v_i$  to  $v_j$  that includes only the nodes  $v_1, v_2, \dots, v_m$  except possibly at its beginning or end. We allow the ends of the paths to be outside of this range of nodes because the specified nodes  $v_i$  and  $v_j$  might be outside of this range. To simplify the problem, the algorithm must return only  $cost_m[i, j]$  and  $\pi_m[i, j]$  which are the total weight of the path found and the second last node in it. When this cost is  $-\infty$ , then we must also store a node  $cycleNode[i, j]$  on the negative cycle within this infinite path. More formally,

**Preconditions:** For instance  $\langle G_m, i, j \rangle$ ,  $G$  is a weighted graph (directed or undirected) with possibly negative weights and  $v_m, v_i$ , and  $v_j$  are nodes.

**Postconditions:** An optimal path from  $v_i$  to  $v_j$  from amongst those paths that includes only the nodes  $v_1, v_2, \dots, v_m$  except possibly at its beginning or end is returned along with its cost  $\text{cost}_m[i, j]$ . This is stored as follows.  $\pi_m$  specifies a *simple* path between every pair of nodes  $v_i$  and  $v_j$ . If  $\text{cost}_m[i, j] \neq -\infty$ , then this path is optimal. If  $\text{cost}_m[i, j] = -\infty$ , then the path  $v_i, \dots, (\text{cycleNode}[i, j], \dots)^*, \dots, v_j$  is an optimal infinitely long path with a negative cycle.

**Dijkstra's Algorithm:** This is very similar to the second loop invariant LI2 for Dijkstra's algorithm in Section 20.3, namely that for each node  $v$ , the algorithm has a shortest path from the fixed source node  $s$  to  $v$  from among those paths that have been *handled*. Handled paths contain only previously handled nodes except possibly at its beginning or end. In Dijkstra's algorithm the order to "handle" the nodes is decided dynamically, while in this All Pairs Shortest Paths algorithm the nodes are handled in order. At this point in time,  $v_1, v_2, \dots, v_m$  have been handled.

**Base Cases and The Original Instance:** Note that the new instance  $\langle G_n, s, t \rangle$  is our original instance asking for an optimal path from  $v_i$  to  $v_j$  including any node in the middle. Also note that the new instance  $\langle G_0, i, j \rangle$  asks for an optimal path from  $v_i$  to  $v_j$  but does not allow any nodes to be included in the middle. Hence, the only valid solution is the single edge from  $v_i$  to  $v_j$ . If there is such an edge then  $\text{cost}_0[i, j]$  is the weight of this edge and  $\pi_0[i, j] = i$ . Otherwise,  $\text{cost}_0[i, j] = \infty$  and  $\pi_0[i, j]$  is nil.

**Your Instance Reduced to a Subinstance:** Given an instance  $\langle G_m, i, j \rangle$ , the last node in the graph of consideration is  $v_m$  not  $v_n$ . Hence, we will ask the little bird how many times the node  $v_m$  is included internally within an optimal path from  $v_i$  to  $v_j$ . Recall that there are only  $K = 3$  possible answers,  $k = 0, 1$ , or  $\infty$ .

**$v_m$  is Included Internally  $k = 0$  Times:** Suppose that the bird assures us that the node  $v_m$  does not appear internally within at least one of the shortest weighted paths from  $v_i$  to  $v_j$ .

**One Friend:** We could simply ignore the node  $v_m$  and ask a friend to give us a shortest weighted path from  $v_i$  to  $v_j$  that does not include node  $v_m$ . This amounts to the subinstance  $\langle G_{m-1}, i, j \rangle$ . See the left path from  $v_i$  to  $v_j$  in Figure 24.7.

**$\text{cost}_m[i, j]$  and  $\pi_m[i, j]$ :** The path, and hence its weight and second last node, have not changed, giving  $\text{cost}_m[i, j] = \text{cost}_{m-1}[i, j]$  and  $\pi_m[i, j] = \pi_{m-1}[i, j]$ .

**$v_m$  is Included Internally  $k = 1$  Times:** On the other hand, suppose that we are told that node  $v_m$  appears exactly once in least one of the shortest weighted paths from  $v_i$  to  $v_j$ .

**Two Friends:** We know that there is a shortest paths from  $v_i$  to  $v_j$  that is composed of the following two parts: a shortest weighted path from  $v_i$  to  $v_m$  that includes node  $v_m$  only at the end and a shortest one from  $v_m$  to  $v_j$  that includes node  $v_m$  only at the beginning. We can ask friends to find each of these two subpaths by giving them the subinstances  $\langle G_{m-1}, i, m \rangle$  and  $\langle G_{m-1}, m, j \rangle$ . We combine their answers to obtain our path. See the right path from  $v_i$  to  $v_j$  in Figure 24.7 excluding the dotted part.

**$\text{cost}_m[i, j]$  and  $\pi_m[i, j]$ :** The weight of our combined path will be the sum of the weights given to us by our friends, namely  $\text{cost}_m[i, j] = \text{cost}_{m-1}[i, m] + \text{cost}_{m-1}[m, j]$ .

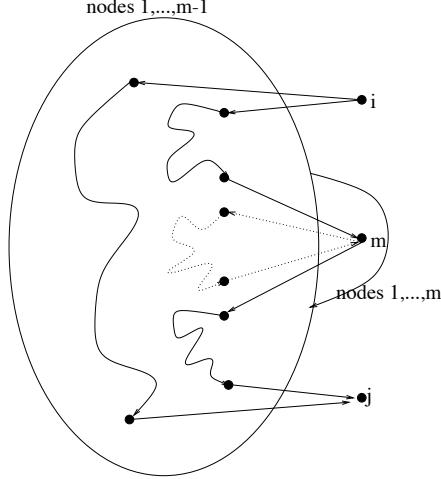


Figure 24.7: Updating the path from  $v_i$  to  $v_j$  when node  $v_m$  is included

The second last node in our path from  $v_i$  to  $v_j$  is the second last node in the path from  $v_m$  to  $v_j$  given to us by our second friend, namely  $\pi_m[i, j] = \pi_{m-1}[m, j]$ .

**$v_m$  is Included Internally  $k = \infty$  Times:** Finally, suppose that we are told that  $v_m$  appears infinitely often in least one of the shortest weighted paths from  $v_i$  to  $v_j$ .

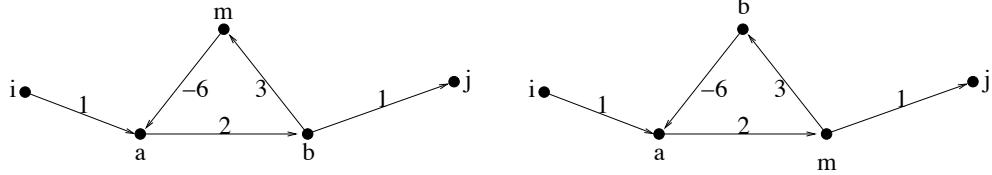
**Three Friends:** The only reason for an optimal path to return to  $v_m$  more than once is because there is an negative weighted cycle including  $v_m$ . Though there may be more than one such negative weighted cycle, there is no advantage in an optimal path taking different ones. Hence, there is an optimal path from  $v_i$  to  $v_j$  with the following form: an optimal path from  $v_i$  to  $v_m$  that includes node  $v_m$  only at the end; an infinitely repeating negatively weighted optimal path from  $v_m$  back to  $v_m$  that includes node  $v_m$  only at the ends; finally an optimal path from  $v_m$  to  $v_j$  that includes node  $v_m$  only at the beginning. We can ask friends to find each of these three subpaths by giving them the subinstances  $\langle G_{m-1}, i, m \rangle$ ,  $\langle G_{m-1}, m, m \rangle$ , and  $\langle G_{m-1}, m, j \rangle$ . We combine their answers to obtain our optimal path. See the right most path from  $v_i$  to  $v_j$  in Figure 24.7 including the dotted part.

**$cost_m[i, j]$ :** Again the weight of our combined path will be the sum of the weights given to us by our friends, namely  $cost_m[i, j] = cost_{m-1}[i, m] + \infty \cdot cost_{m-1}[m, m] + cost_{m-1}[m, j]$ . For this weight to be  $-\infty$  as desired, there are three requirements. The cycle needs to have a negative weight, namely  $cost_{m-1}[m, m] < 0$ . Also there must actually be a path from  $v_i$  to the cycle, namely  $cost_{m-1}[i, m] \neq \infty$ , and then from the cycle onto  $v_j$ , namely  $cost_{m-1}[m, j] \neq \infty$ . If one of these is missing, then the bird is mistaken to tell us to visit  $v_m$  more than once. We strongly discourage this options by setting  $cost_m[i, j] = +\infty$ .

**Node on Cycle  $cycleNode[i, j]$ :** Recall that when there is the optimal path from  $v_i$  to  $v_j$  is infinitely long, the data structure  $cycleNode[i, j]$  is to store one node  $v_m$  on one such negative cycle along the path. This is done simply by setting  $cycleNode[i, j] = m$ .

**$\pi_m[i, j]$ :** In this case, the path traced backwards by  $\pi$  does not need to be an optimal path. The only requirement is that it is an simple path. There are two cases to consider.

**The Simple Path Excluding  $v_m$ :** The  $k = 0$  friend, given the subinstance  $\langle G_{m-1}, i, j \rangle$ , can give us a simple path from  $v_i$  to  $v_j$  that does not contain  $v_m$ . This path would suit our purposes fine, assuming that it exists. We can test this by whether  $\text{cost}_{m-1}[i, j] \neq \infty$ . To take this path, we set  $\pi_m[i, j] = \pi_{m-1}[i, j]$ . Note that in the left graph below such a path exists, but that in the right one it does not.



**The Simple Path Including  $v_m$ :** Now suppose that a path from  $v_i$  to  $v_j$  excluding  $v_m$  does not exist. We do know that there is one that includes  $v_m$  an infinite number of times, hence there is also a truncated version of this path that includes it only once. The  $k = 1$  friends, with the subinstances  $\langle G_{m-1}, i, m \rangle$  and  $\langle G_{m-1}, m, j \rangle$ , can give this path. This path would also suit our purposes fine, assuming that it does not contain a cycle. Note that in the left graph above this path does contain a cycle, but that in the right one it does not. We have handled the left example, using the  $k = 0$  path. For the right example, we use the  $k = 1$  path. To take this path, we set  $\pi_m[i, j] = \pi_{m-1}[m, j]$ . Closer examination reveals that these are the only two cases.

This completes the steps for solving an individual instance  $\langle G_m, i, j \rangle$ .

#### Filling in The Table:

**The Set of Subinstances:** The claim was that the set of subinstance used is  $\{\langle G_m, i, j \rangle \mid i, j \in [1..n] \text{ and } m \in [0..n]\}$ .

**Closed:** We know that this set contains all subinstances generated by the recursive algorithm because it contains the initial instance and is closed under the sub-operator. Consider an arbitrary subinstance  $\langle G_m, i, j \rangle$  from this set. Applying the sub-operator constructs the subinstances  $\langle G_{m-1}, i, j \rangle$ ,  $\langle G_{m-1}, i, m \rangle$ ,  $\langle G_{m-1}, m, m \rangle$ , and  $\langle G_{m-1}, m, j \rangle$ , all of which are contained in the stated set of subinstances.

**Generating:** Starting for the original instance  $\langle G_n, s, t \rangle$ , the only subinstances  $\langle G_m, i, j \rangle$  that will get called by the recursive algorithm are those for which  $v_i$  and  $v_j$  are either  $v_s$  and  $v_t$  or within the range  $v_m, v_{m+1}, \dots, v_n$ . However, if we want to solve the all pairs shortest paths problem, then all of these subinstances will be needed.

**What is Saved in the Table:** As indicated in the introduction for this problem, the algorithm stores something other than the bird's advice in the table  $\text{birdAdvice}_m[i, j]$ . The purpose of storing the bird's advice for each subinstance is so that in the end the optimal solution can be constructed from piecing together the bird's advice for each of these subinstances. The Bird's advice is the number of times  $k = 0, 1$ , or  $\infty$  that the node  $v_m$  appears in an optimal path for the instance. Pieced together, these fields of information are organized into a strange tree structure in which each field has one, two, or three children depending on whether the node  $v_m$  breaks the path into one, two, or three subpaths. See Section 23.4.2. Though this would work, it is a strange data

structure to store the optimal paths. Instead, the tables store  $cost_m[i, j]$ ,  $\pi_m[i, j]$ , and  $cycleNode[i, j]$ .

**The Order in which to Fill the Table:** The “size” of subinstance  $\langle G_m, i, j \rangle$  is the number of nodes  $m$  being considered in the middle of the paths. The tables will be filled in order of  $m = 0, 1, 2, \dots, n$ .

**Code:**

**algorithm** *AllPairsShortestPaths* ( $G$ )

***{pre-cond}*:**  $G$  is a weighted graph (directed or undirected) with possibly negative weights.

***{post-cond}*:** For each pair of nodes  $v_i$  and  $v_j$ , an optimal path from  $v_i$  to  $v_j$  and its cost  $cost[i, j]$  are returned. The path is stored as follows.  $\pi$  specifies a *simple* path between every pair of nodes  $v_i$  and  $v_j$ . This is done by having  $\pi[i, j]$  specify the second last node in the path. If  $cost[i, j] \neq -\infty$ , then this simple path is optimal. If  $cost[i, j] = -\infty$ , then the path  $v_i, \dots, (cycleNode[i, j], \dots)^*, \dots, v_j$  is an optimal infinitely long path with a negative cycle.

begin

% Table:  $optSol_m[i, j]$  would store an optimal solution for the subinstance  $\langle G_m, i, j \rangle$ , namely a minimum weight path from  $v_i$  to  $v_j$  whose internal nodes include only the nodes  $v_1, v_2, \dots, v_m$ . Instead, the paths are stored, as described in the postcondition, with  $\pi_m[i, j]$ ,  $cost_m[i, j]$ , and  $cycleNode_m[i, j]$ .  
 $table[0..n, 1..n, 1..n]$   $\pi, cost, cycleNode$

% Base Cases: The base cases are when  $m = 0$ , i.e. no internal nodes are allowed in the path. Hence, the path either is a single node, a single edge, or does not exist.

for  $i = 1$  to  $n$

    for  $j = 1$  to  $n$

        if(  $i = j$  ) then

$\pi_0[i, j] = \text{nil}$

$cost_0[i, j] = 0$

        else if(  $\langle v_i, v_j \rangle$  is an edge ) then

$\pi_0[i, j] = i$

$cost_0[i, j] = w_{\langle v_i, v_j \rangle}$

        else

$\pi_0[i, j] = \text{nil}$

$cost_0[i, j] = \infty$

        end if

    end for

end for

% General Cases: Loop over subinstances in the table.

for  $m = 1$  to  $n$

    for  $i = 1$  to  $n$

        for  $j = 1$  to  $n$

            % Solve the subinstance  $\langle G_m, i, j \rangle$

            % The bird and Friend Alg: The bird tells us whether the node  $v_m$  appears

$k = 0, k = 1$ , or  $k = \infty$  times in the optimal path.

```

% Try each possible bird answers.
% cases k = 0
    % Because  $v_m$  does not appear in the path,  $optSol_m[i, j]$  is simply  $optSol_{m-1}[i, j]$ .
     $cost_{k=0} = cost_{m-1}[i, j]$ 
     $\pi_{k=0} = \pi_{m-1}[i, j]$ 
% cases k = 1
    % Because  $v_m$  appears once in the path,  $optSol_m[i, j]$  is the concatenation of  $optSol_{m-1}[i, m]$  and  $optSol_{m-1}[m, j]$ .
     $cost_{k=1} = cost_{m-1}[i, m] + cost_{m-1}[m, j]$ 
     $\pi_{k=1} = \pi_{m-1}[m, j]$ 
% cases k =  $\infty$ 
    % Because  $v_m$  appears infinitely often in the path,
    %  $optSol_m[i, j]$  is the concatenation of  $optSol_{m-1}[i, m]$ ,
    % the cycle  $optSol_{m-1}[m, m]$  infinitely often, ending with
    %  $optSol_{m-1}[m, j]$ .
    if(  $cost_{m-1}[i, m] \neq \infty$  and  $cost_{m-1}[m, m] < 0$  and  $cost_{m-1}[m, j] \neq \infty$  ) then
        %  $v_m$  is on a negative cycle
         $cost_{k=\infty} = -\infty$ 
         $cycleNode[i, j] = m$ 
        if(  $cost_{m-1}[i, j] \neq \infty$ )
             $\pi_{k=\infty} = \pi_{m-1}[i, j]$ 
        else
             $\pi_{k=\infty} = \pi_{m-1}[m, j]$ 
        end if
    else
        % Little bird made a mistake
         $cost_{\infty} = +\infty$ 
    end if
% end cases
% Having the best,  $optSol_k$ , for each bird's answer  $k$ ,
    we keep the best of these best. (Given a tie, take  $k = 0$  over  $k = 1$  over
     $k = \infty$ .)
     $k_{min} = \text{"a } k \in [0, 1, \infty] \text{ that minimizes } cost_k"$ 
    %  $optSol_m[i, j] = optSol_{k_{min}}$ 
     $\pi_m[i, j] = \pi_{k_{min}}$ 
     $cost[i, j] = cost_{k_{min}}$ 
end for
end for
end for
return  $\langle \pi_n, cost_n \rangle$ 
end algorithm

```

**Time and Space Requirements:** The running time is the number of subinstances times the number of possible bird answers and the space is the number of subinstances. The number of subinstances is  $\Theta(n^3)$  and the bird has  $K=$ three possible answers for you. Hence, the time and space requirements are both  $\Theta(n^3)$ .

**Saving Space:** Memory space can be saved by observing that to solve the subinstance  $\langle G_m, i, j \rangle$  only the values for the previous  $m$  are needed. Hence the same table can be reused, by constructing the current table from the previous table and then coping the current to the previous and repeating. Even more space and coding hassles can be saved by not having one set of tables for the current value of  $m$  and another for the previous value, by simply updating the values in place. However, before this is done, more care is needed to make sure that this simplified algorithm still works.

**Example:**

Graph with weighted edges:				
$cost_0 =$				
a	0	40	1	60
b	6	0	10	4
c	20	$\infty$	0	2
d	$\infty$	4	$\infty$	0

$\Pi_0 =$				
$cost_a =$				
a	0	40	1	60
b	6	0	7	4
c	20	60	0	2
d	$\infty$	4	$\infty$	0

$\Pi_a =$				
$cost_b =$				
a	0	40	1	44
b	6	0	7	4
c	20	60	0	2
d	10	4	11	0

$cost_b =$				
$\Pi_b =$				
a	a	b	c	d
b	b	a	a	b
c	c	a	c	c
d	b	d	a	

$cost_c =$				
$\Pi_c =$				
a	0	40	1	3
b	6	0	7	4
c	20	60	0	2
d	10	4	11	0

$cost_d =$				
$\Pi_d =$				
a	0	7	1	3
b	6	0	7	4
c	12	6	0	2
d	10	4	11	0

Shorten *PrintPath* to simply *P*.  
Then  $P(a, b) = P(a, d), b = P(a, c), d, b = P(a, a), c, d, b = a, c, d, b$ .

**Exercise 24.8.1** Trace the *AllPairsShortestPaths* and *PrintPath* on the graph in Figure 24.4. Consider the nodes ordered alphabetically.

## 24.9 All Pairs Using Matrix Multiplication

There is another dynamic programming algorithm that also finds the shortest that between every pair of nodes. It is similar in some ways to the Floyd-Warshall and Johnson algorithm, but it is fun because it can be viewed as matrix multiplication.

**Exercise 24.9.1** Let  $G = (V, E)$  be a (directed or undirected) graph and  $k \leq n$  some integer. Let  $M^k$  be a matrix with a both a row and a column for each node in the graph such that for each pair of nodes  $u, v \in V$   $M^k[u, v]$  gives the number of distinct paths from  $u$  to  $v$  that contains exactly  $k$  edges. Here a path may visit a node more than once. Note that  $M^1[u, v]$  is 1 if there is an edge  $\langle u, v \rangle$  or is zero and  $M^1[u, u] = 1$  because there is a path of length zero from  $u$  to  $u$ . Prove that  $M^{i+j} = M^i \times M^j$ , where “ $\times$ ” is standard matrix multiplication, i.e  $M^{i+j}[u, v] = M^i[u, v] \times M^j[u, v] = \sum_w [M^i[u, w] \cdot M^j[w, v]]$ .

**Exercise 24.9.2** Now let the graph  $G = (V, E)$  have weights  $w_{u,v}$  (positive or negative) on each edge. Redefine  $\widehat{M}^k[u, v]$  to give the weight of the shortest path from  $u$  to  $v$  with the smallest total weight from amongst these paths that contains exactly  $k$  edges. Note that  $\widehat{M}^1[u, v]$  is the weight of the edge  $w_{u,v}$  (or infinity) and  $\widehat{M}^1[u, u] = 0$  because there is a path of length zero from  $u$  to

*u. Prove that  $\widehat{M}^{i+j} = \widehat{M}^i \times \widehat{M}^j$ , where “ $\times$ ” is standard matrix multiplication except that scalar multiplication is changed to “ $+$ ” and “ $+$ ” is changed to  $\text{Min}$ , i.e  $\widehat{M}^{i+j}[u, v] = \widehat{M}^i[u, v] \times \widehat{M}^j[u, v] = \text{Min}_w [\widehat{M}^i[u, w] + \widehat{M}^j[w, v]]$ .*

**Exercise 24.9.3** *If all the edge weights are positive, then the shortest weighted path contains at most  $n - 1$  edges. Hence,  $\widehat{M}^N[u, v]$  for  $N \geq n - 1$  gives the overall shortest weighted path from  $u$  to  $v$ . Given  $\widehat{M}^1$ , what is the fastest way of computing  $\widehat{M}^N$  for some  $N \geq n$ ?*

**Exercise 24.9.4** *If there is a path from  $u$  to  $v$  containing a negative weighted cycle, then this negative cycle can be repeated infinitely often giving a path with negative infinity weight and infinite edges. To detect this, compute  $\widehat{M}^N[u, v]$  and  $\widehat{M}^{2N}[u, v]$  for some large  $N$  and see if they are different. The question is how large does  $N$  need to be. You would think that  $N = n - 1$  would be sufficient, but it is not. Give a graph with  $n$  edges, each with positive or negative  $\ell$  bit integer weights, for which  $\widehat{M}^k[u, v] = \widehat{M}^1[u, v]$  for  $k \in [1, N]$  for some very large  $N$  but  $\widehat{M}^{N+1}[u, v]$  is smaller.*

**Exercise 24.9.5** *The standard algorithm for standard matrix multiplication takes  $\Theta(n^3)$  time. Strassen’s algorithm Section 15.2 is able to do it in  $\Theta(n^{2.8073})$  time. Does this same algorithm work for this strange multiplication? The following,  $x \cdot (y + z) = x \cdot y + x \cdot z$  and  $(x - y) + y = x$  are true for all real numbers. Are the same true when replacing  $\cdot$  with  $+$  and replacing  $+$  with  $\text{Min}$ ?*

## 24.10 Parsing with Context-Free Grammars

Recall the problem of parsing a string according to a given context-free grammar. Section 18 developed an elegant recursive algorithm which works only for look-ahead-one grammars. We now present a dynamic programming algorithm that works for any context-free grammar.

Given a grammar  $G$  and a string  $s$ , the first step in parsing is to convert the grammar into one in **Chomsky normal form**, which is defined below. (Although, a dynamic program could be written to work directly for any context-free grammar, it runs much faster if the grammar is converted first.)

### The Parsing Problem:

**Instance:** An instance consists of  $\langle G, T_{\text{start}}, s \rangle$ , where  $G$  is a grammar in Chomsky normal form,  $T_{\text{start}}$  is the non-terminal of  $G$  designated as the start symbol, and  $s$  is the string  $\langle a_1, \dots, a_n \rangle$  of terminal symbols to be generated. The grammar  $G$  consists of a set of non-terminal symbols  $V = \langle T_1, \dots, T_{|V|} \rangle$  and a set of rules  $\langle r_1, \dots, r_m \rangle$ . The definition of Chomsky normal form is that each rule  $r_q$  has one of the following three forms:

- $A_q \Rightarrow B_q C_q$ , where  $A_q$ ,  $B_q$ , and  $C_q$  are non-terminal symbols.
- $A_q \Rightarrow b_q$ , where  $b_q$  is a terminal symbol.
- $T_{\text{start}} \Rightarrow \epsilon$ , where  $T_{\text{start}}$  is the start symbol and  $\epsilon$  is the empty string. This rule can only be used to parse the string  $s = \epsilon$ . It may not be used within the parsing of a larger string.

**Solution:** A solution is a partial parsing  $P$ , consisting of a tree. Each internal node of the tree is labeled with a non-terminal symbol, the root with the specified symbol  $T_{\text{start}}$ . Each internal node must correspond to a rule of the grammar  $G$ . For example, for rule  $A \Rightarrow BC$ , the node is labeled  $A$  and its two children are labeled  $B$  and  $C$ . In a complete

parsing, each leaf of the tree is labeled with a terminal symbol. (In a partial parsing, some leaves may still be labeled with non-terminals.)

**Cost of Solution:** A parsing  $P$  is said to generate the string  $s$  if the leaves of the parsing in order forms  $s$ . The cost of  $P$  will be zero if it generates the string  $s$  and will be infinity otherwise.

**Goal:** The goal of the problem is, given an instance  $\langle G, T_{start}, s \rangle$ , to find a parsing  $P$  that generates  $s$ .

**Not Look Ahead One:** The grammar  $G$  might not be *look ahead one*. For example, in

$$\begin{aligned} A &\Rightarrow B \ C \\ A &\Rightarrow D \ E \end{aligned}$$

you do not know whether to start parsing the string as a B or a D. If you make the wrong choice, you have to back up and repeat the process. However, this problem is a perfect candidate for a dynamic-programming algorithm.

**The Parsing Abstract Data Type:** We will use the following abstract data type to represent parsings. Suppose that there is a rule  $r_q = "A_q \Rightarrow B_q C_q"$  that generates  $B_q$  and  $C_q$  from  $A_q$ . Suppose as well that the string  $s_1 = \langle a_1, \dots, a_k \rangle$  is generated starting with the symbol  $B_q$  using the parsing  $P_1$  ( $B_q$  is the root of  $P_1$ ) and that  $s_2 = \langle a_{k+1}, \dots, a_n \rangle$  is generated from  $C_q$  using  $P_1$ . Then we say that the string  $s = s_1 \circ s_2 = \langle a_1, \dots, a_n \rangle$  is generated from  $A_q$  using the parsing  $P = \langle A_q, P_1, P_2 \rangle$ .

**The Number of Parses:** Usually, the first algorithmic attempts at parsing are some form of brute force algorithm. The problem is that there are an exponential number of parses to try. This number can be estimated roughly as follows. When parsing, the string of symbols needs to increase from being of size 1 (consisting only of the start symbols) to being of size  $n$  (consisting of  $s$ ). Applying a rule adds only one more symbol to this string. Hence, rules need to be applied  $n - 1$  times. Each time you apply a rule, you have to choose which of the  $m$  rules to apply. Hence, the total number of choices may be  $\Theta(m^n)$ .

**The Question to Ask the Little Bird:** Given an instance  $\langle G, T_{start}, s \rangle$ , we will ask the little bird a question that contains two sub-questions about a parsing  $P$  that generates  $s$  from  $T_{start}$ .

The first sub-question is the index  $q$  of the rule  $r_q = "T_{start} \Rightarrow B_q C_q"$  that is applied first to our start symbol  $T_{start}$ . Although this is useful information, I don't see how it alone could lead to a subinstance.

We don't know  $P$ , but we do know that  $P$  generates  $s = \langle a_1, \dots, a_n \rangle$ . It follows that, for some  $k \in [1..n]$ , after  $P$  applies its first rule  $r_q = "T_{start} \Rightarrow B_q C_q"$ , it then generates the string  $s_1 = \langle a_1, \dots, a_k \rangle$  from  $B_q$  and the string  $s_2 = \langle a_{k+1}, \dots, a_n \rangle$  from  $C_q$ , so that overall it generates  $s = s_1 \circ s_2 = \langle a_1, \dots, a_n \rangle$ . Our second sub-question asked of the bird is to tell us this  $k$  that splits the string  $s$ .

**Help From Friend:** What we do not know about of the parsing tree  $P$  is how  $B_q$  generates  $s_1 = \langle a_1, \dots, a_k \rangle$  and how  $C_q$  generates  $s_2 = \langle a_{k+1}, \dots, a_n \rangle$ . Hence, we ask our friends for optimal parses for the subinstances  $\langle G, B_q, s_1 \rangle$  and  $\langle G, C_q, s_2 \rangle$ . They respond with the parses  $P_1$  and  $P_2$ . We conclude that  $P = \langle T_{start}, P_1, P_2 \rangle$  generates  $s = s_1 \circ s_2 = \langle a_1, \dots, a_n \rangle$

from  $T_{start}$ . If either friend gives us a parsing with infinite cost, then we know that no parsing consistent with the information provided by the bird is possible. The cost of our parsings in this case is infinity as well. This can be achieved by setting the cost of the new parsing to be the maximum of that for  $P_1$  and for  $P_2$ . The line of code will be  $cost_{\langle q,k \rangle} = \max(cost[B_q, 1, k], cost[C_q, k + 1, n])$ .

**The Set of Subinstances:** The set of subinstances that get called by the recursive program consisting of you, your friends, and their friends is  $\{\langle G, T_h, a_i, \dots, a_j \rangle \mid h \in V, 1 \leq i \leq j \leq n\}$ .

**Closed:** We know that this set contains all subinstances generated by the recursive algorithm because it contains the initial instance and is closed under the sub-operator. Consider an arbitrary subinstance  $\langle G, T_h, a_i, \dots, a_j \rangle$  in the set. Its subinstances are  $\langle G, B_q, a_i, \dots, a_k \rangle$  and  $\langle G, C_q, a_{k+1}, \dots, a_j \rangle$ , which are both in the set.

**Generating:** Some of these subinstances will not be generated. However, most of our instances will.

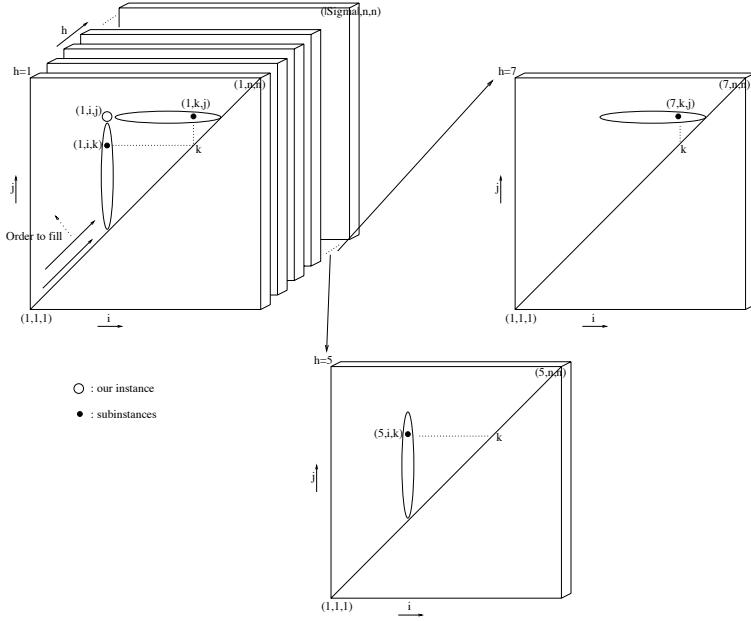


Figure 24.8: The dynamic-programming table for parsing is shown. The table entry corresponding to the instance  $\langle G, T_1, a_i, \dots, a_j \rangle$  is represented by the little circle. Using the rule  $T_1 \Rightarrow T_1 T_1$ , the subinstances  $\langle G, T_1, a_i, \dots, a_k \rangle$  and  $\langle G, T_1, a_{k+1}, \dots, a_j \rangle$  are formed. Using the rule  $T_1 \Rightarrow T_5 T_7$ , the subinstances  $\langle G, T_5, a_i, \dots, a_k \rangle$  and  $\langle G, T_7, a_{k+1}, \dots, a_j \rangle$  are formed. The table entries corresponding to these subinstances are represented by the dot within the ovals.

**Constructing a Table Indexed by Subinstances:** The table would be three dimensional. The solution for subinstance  $\langle G, T_h, a_i, \dots, a_j \rangle$  would be stored in entry  $Table[h, i, j]$  for  $h \in V$  and  $1 \leq i \leq j \leq n$ .

**The Order in which to Fill the Table:** The size of the subinstance  $\langle G, T_h, a_i, \dots, a_j \rangle$  is the length of the string to be generated, i.e.,  $j - i + 1$ . We will consider longer and longer strings.

**Base Cases:** One base case is the subinstance  $\langle G, T_{start}, \epsilon \rangle$ . This empty string  $\epsilon$  is parsed with the rule  $T_{start} \Rightarrow \epsilon$ , assuming that this is a legal rule. The other base cases are the subinstances  $\langle G, A_q, b_q \rangle$ . This string consisting of the single character  $b_q$  is parsed with the rule  $A_q \Rightarrow b_q$ , assuming that this is a legal rule.

**Code:**

```

algorithm Parsing( $\langle G, T_{start}, a_1, \dots, a_n \rangle$ )
<pre-cond>:  $G$  is a Chomsky normal form grammar,  $T_{start}$  is a non-terminal, and  $s$  is the
string  $\langle a_1, \dots, a_n \rangle$  of terminal symbols.
<post-cond>:  $P$ , if possible, is a parsing that generates  $s$  starting from  $T_{start}$  using  $G$ .

begin
    % Table:  $optSol[h, i, j]$  would store an optimal solution for  $\langle G, T_h, a_i, \dots, a_j \rangle$ , namely
    % a parsing for  $a_i, \dots, a_j$  starting with non-terminal  $T_h$ . Instead, we store only the
    % bird's advice for the subinstance and the cost of its solution.
     $table[|V|, n, n]$  birdAdvice, cost

    % The case  $s = \epsilon$  is handled separately
    if  $n = 0$  then
        if  $T_{start} \Rightarrow \epsilon$  is a rule then
             $P$  = the parsing applies this one rule.
        else
             $P = \emptyset$ 
        end if
        return( $P$ )
    end if

    % Base Cases: The base cases are when the string to parse only one character  $a_i$ .
    For  $i = 1$  to  $n$ 
        For each non-terminal  $T_h$ 
            If there is a rule  $r_q = "A_q \Rightarrow b_q"$ , where  $A_q$  is  $T_h$  and  $b_q$  is  $a_i$  then
                 $birdAdvice[h, i, i] = \langle q, ? \rangle$ 
                 $cost[h, i, i] = 0$ 
            else
                 $birdAdvice[h, i, i] = \langle ?, ? \rangle$ 
                 $cost[h, i, i] = \infty$ 
            end if
        end loop
    end loop

    % General Cases: Loop over subinstances in the table.
    for  $size = 2$  to  $n$  % length of substring  $\langle a_i, \dots, a_j \rangle$ 
        for  $i = 1$  to  $n - size + 1$ 
             $j = i + size - 1$ 
            For each non-terminal  $T_h$ , i.e.,  $h \in [1..|V|]$ 
                % Solve instance  $\langle G, T_h, a_i, \dots, a_j \rangle$  and fill in table entry  $\langle h, i, j \rangle$ 
                % Loop over possible bird answers.

```

```

for each rule  $r_q = "A_q \Rightarrow B_q C_q"$  for which  $A_q$  is  $T_h$ 
    for each split in the string  $k = i$  to  $j - 1$ 
        % Ask friend if you can generate  $\langle a_i, \dots, a_k \rangle$  from  $B_q$ 
        % Ask another friend if you can generate  $\langle a_{k+1}, \dots, a_j \rangle$  from  $C_q$ 
         $cost_{\langle q, k \rangle} = \max(cost[B_q, i, k], cost[C_q, k + 1, j])$ 
    end for
    end for
    % Take the best bird answer, i.e., one of cost zero if  $s$  can be generated.
     $\langle q_{min}, k_{min} \rangle = "a \langle q, k \rangle \text{ that minimizes } cost_{\langle q, k \rangle}"$ 
     $birdAdvice[h, i, j] = \langle q_{min}, k_{min} \rangle$ 
     $cost[h, i, j] = cost_{\langle q_{min}, k_{min} \rangle}$ 
end for
end for
end for

% Constructing the solution  $P$ 
if( $cost[1, 1, n] = 0$ ) then % i.e. if  $s$  can be generated from  $T_{start}$ 
     $P = ParsingWithAdvice(\langle G, T_{start}, a_1, \dots, a_n \rangle, birdAdvice)$ 
else
     $P = \emptyset$ 
end if
return( $P$ )
end algorithm

```

### Constructing an Optimal Solution:

**algorithm**  $ParsingWithAdvice(\langle G, T_h, a_i, \dots, a_j \rangle, birdAdvice)$   
**⟨pre & post-cond⟩:** Same as  $Parsing$  except with advice.

```

begin
     $\langle q, k \rangle = birdAdvice[h, i, j]$ 
    if( $i = j$ ) then
        Rule  $r_q$  must have the form " $A_q \Rightarrow b_q$ ", where  $A_q$  is  $T_h$  and  $b_q$  is  $a_i$ 
        Parsing  $P = \langle T_h, a_i \rangle$ 
    else
        Rule  $r_q$  must have the form " $A_q \Rightarrow C_q B_q$ ", where  $A_q$  is  $T_h$ 
         $P_1 = ParsingWithAdvice(\langle G, B_q, a_i, \dots, a_k \rangle, birdAdvice)$ 
         $P_2 = ParsingWithAdvice(\langle G, C_q, a_{k+1}, \dots, a_j \rangle, birdAdvice)$ 
        Parsing  $P = \langle T_h, P_1, P_2 \rangle$ 
    end if
    return( $P$ )
end algorithm

```

**Time and Space Requirements:** The running time is the number of subinstances times the number of possible bird answers and the space is the number of subinstances. The number of subinstances indexing your table is  $\Theta(|V|n^2)$ , namely  $Table[h, i, j]$  for  $h \in V$  and  $1 \leq i \leq j \leq n$ . The number of answers that the bird might give you is at most  $\mathcal{O}(mn)$ , namely  $\langle q, k \rangle$

for each of the  $m$  rules  $r_q$  and split  $k \in [1..n - 1]$ . This gives time =  $\mathcal{O}(|V|n^2 \cdot mn)$ . If the grammar  $G$  is fixed, then the time is  $\Theta(n^3)$ .

A tighter analysis would note that the bird would only answer  $q$  for rules  $r_q = "A_q \Rightarrow B_q C_q"$ , for which the left-hand side  $A_q$  is the non-terminal  $T_h$  specified in the instance. Let  $m_{T_h}$  be the number of such rules. Then the loop over non-terminals  $T_h$  and the loop over rules  $r_q$  would not require  $|V|m$  time, but  $\sum_{T_h \in V} m_{T_h} = m$ . This gives a total time of  $\Theta(n^3m)$ .

## 24.11 More Examples

**Exercise 24.11.1** *The problem is to schedule a tour trying to attend as many important events as possible. The input includes of an undirected graph  $G = \langle V, E \rangle$  with  $V$  being a set of  $L$  locations and  $d_{u,v}$  being the time required to travel from location  $u$  to location  $v$  for  $\{u, v\} \in E$ . Let  $D$  be the maximum degree of a node in  $G$ . The input includes a range of times  $[1, T]$ . Finally, the input includes a list of  $n$  events. The event  $E_i$  occurs at location  $v_{l_i} \in V$ , at time  $t_i \in [1, T]$ , and is worth  $w_i$  dollars. Events may occur at the same locations and/or the same time. The events are sorted by their time  $t_i$ . Let  $E_0$  denote the start event, which occurs at node  $v_{l_0}$  and time  $t_0 = 0$ . Its worth  $w_0$  is always zero. A solution is a subset the events that can be attended starting with the start event, i.e. if event  $E_i$  and  $E_j$  are both attended, then the shortest distance between their locations is at most the time between them, i.e.  $|v_{l_j} - v_{l_i}| \leq |t_j - t_i|$ . The worth of a solution is the total of the worths  $w_i$  of the events attended. The goal is to maximize the worth of the schedule. Here are a few bird questions:*

1. From our current space/time location  $\langle v_{l_0}, t_0 \rangle$  which edge in  $G$  should we take?
2. Which event should we attend first?
3. Should we attend the first event  $E_1$ ?
4. Should we attend the last event  $E_n$ ? For this one, I can think of two different ways of proceeding with friends.
  - (a) Include in an instance a finish location. The new problem is to attend as many events starting with the start event and ending with the ending event.
  - (b) Another option is to try to keep the finishing location unspecified. If the bird says to attend event  $E_n$ , then when deleting event  $E_n$  from the friend's list, one way to ensure that this event can be visited is to but to also delete any event  $E_i$  from which event  $E_n$  can't be reached in time.

All but one of these leads to a reasonable dynamic programming algorithm. For each, give the bird-friend algorithm, the set of subinstances that need to be solved by the recursive backtracking algorithm, the table layout, the order to fill the table, and the running time. Compare their running times.

# Chapter 25

## Examples of Recursive Backtracking

Recursive backtracking algorithms can only be converted into dynamic programming algorithms only when the set of subinstances to be solved is small and predictable. If this is not the case, the recursive backtracking algorithm can still be an effective algorithmic technique. To begin, if one is content to do a brute force search through all solutions, then recursive backtracking provides an systematic way to do this. More over, often the algorithm can often be sped up by pruning off entire branches of the recursive tree. In practice, if the instance that one needs to solve is sufficiently small and has enough structure that a lot of pruning is possible, then an optimal solution can be found for the instance reasonably quickly. However, for large worst case instances, the running time is still exponential.

### 25.1 Pruning Branches

The following are typical reasons why an entire branch of the solution classification tree can be pruned off.

**Not Valid Solutions:** Recall that in a recursive backtracking algorithm, the “little bird” tells the algorithm something about the solution and then the recursive backtracking algorithm recurses by asking a friend a question. Then this friend gets more information about the solution from his “little bird” and so on. Hence, following a path down the recursive tree specifies more and more about a solution until a leaf of the tree fully specifies one particular solution. Sometimes it happens that part way down the tree, the algorithm has already received enough information about the solution to determine that it contains a conflict or defect making any such solution invalid. The algorithm can stop recursing at this point and backtrack. This effectively prunes off the entire subtree of solutions rooted at this node in the tree.

**The Longest Common Subsequence Problem:** There have been a number of examples already in which the algorithm knew to prune the recursive search. For example, in the Longest Common Subsequence Problem in Section 24.2 if the bird says that both of the last two characters are included in the solution, but the last two characters are different, then the algorithm knows that the bird was wrong.

**Not Highly Valued Solutions:** Similarly, when the algorithm arrives at the root of a subtree it might realize that all solutions within this subtree are not rated sufficiently high to be

optimal. Perhaps, because the algorithm has already found a solution provably better than all of these. Again, the algorithm can prune this entire subtree from its search.

**Greedy Algorithms:** Greedy algorithms are effectively recursive backtracking algorithms with extreme pruning. Whenever the algorithm has a choice as to which “little bird” answer to take, i.e. which path down the recursive tree to take, instead of iterating through all of the options, it goes only for the one that looks best according to some greedy criteria. In this way the algorithm follows only one path down the recursive tree. Greedy algorithms are covered in Chapter 22.

**Massaging Solutions:** Let us recall why greedy algorithm are able to prune so that we can use the same reasoning with recursive backtracking algorithms. In each step in a greedy algorithm, the algorithm commits to some decision about the solution. This effectively burns some of his bridges because it eliminates some solutions from consideration. However, this is fine as long as he does not burn all his bridges. The prover proves that there is an optimal solution consistent with the choices made by massaging any possible solution that is not consistent with the latest choice into one that has at least as good value and is consistent with this choice. Similarly, a recursive backtracking algorithm can prune of branches in its tree when it know that this does not eliminate all remaining optimal solutions.

**Depth First Search:** Recursive depth first search, Section 20.5, is a recursive backtracking algorithm. A solution to the “searching a maze for cheese” optimization problem is a path in the graph starting from  $s$ . The value of a solution is the weight of the node at the end of the path. The algorithm marks nodes that it has visited. Then when the algorithm revisits a node, it knows that it can prune this subtree in this recursive search because it knows that any node reachable from the current node has already been reached. In Figure 20.8, the path  $\langle s, c, u, v \rangle$  is pruned because it can be massaged into the path  $\langle s, b, u, v \rangle$  which is just as good.

**The Longest Common Subsequence Problem:** If the instance to the Longest Common Subsequence Problem is  $X = \langle A, B, C, B \rangle$  and  $Y = \langle A, D, B \rangle$ , then the improved version of the algorithm (See Exercise 24.2.1 and its proof.), seeing that the last two characters are the same, does not consider the bird answers that the last character of  $X$  or  $Y$  is not include. This prunes off the branch of the recursive tree which includes the optimal solution  $Z = \langle A, B \rangle$  as  $X = \langle \underline{A}, \underline{B}, C, B \rangle$ . This is fine because this optimal solution can be massaged into the optimal solution  $Z = \langle A, B \rangle$  as  $X = \langle \underline{A}, B, C, \underline{B} \rangle$ .

## 25.2 Satisfiability

A famous optimization problem is called *satisfiability* or *SAT* for short. It is one of the basic problems arising in many fields. The recursive backtracking algorithm given here is referred to the *Davis Putnum* algorithm. It is an example of an algorithm whose running time is exponential time for worst case inputs, yet in many practical situations can work well. This algorithm is one of basic algorithms underlying automated theorem proving and robot path planning among other things.

**The Satisfiability Problem:**

**Instances:** An instance (input) consists of a set of constraints on the assignment to the binary variables  $x_1, x_2, \dots, x_n$ . A typical constraint might be  $\langle x_1 \text{ or } \overline{x}_3 \text{ or } x_8 \rangle$ , meaning

$(x_1 = 1 \text{ or } x_3 = 0 \text{ or } x_8 = 1)$  or equivalently that either  $x_1$  is true,  $x_3$  is false, or  $x_8$  is true. More generally an instance could be a more general circuit built with *AND*, *OR*, and *NOT* gates, but we will leave this until Section 26.1.

**Solutions:** Each of the  $2^n$  assignments is a possible solution. An assignment is valid for the given instance, if it satisfies all of the constraints.

**Cost of Solution:** An assignment is assigned the value one if it satisfies all of the constraints and the value zero otherwise.

**Goal:** Given the constraints, the goal is to find a satisfying assignment.

**Iterating Through the Solutions:** The brute force algorithm simply tries each of the  $2^n$  assignments of the variables. Before reading on, think about how you would non-recursively iterate through all of these solutions. Even this simplest of examples is surprisingly hard.

**Nested Loops:** The obvious algorithm is to have  $n$  nested loops each going from 0 to 1. However, this requires knowing the value of  $n$  before compile time, which is not too likely.

**Incrementing Binary Numbers:** Another option is to treat the assignment as an  $n$  bit binary number and then loop through the  $2^n$  assignments by incrementing this binary number each iteration.

**Recursive Algorithm:** The recursive backtracking technique is able to iterate through the solutions with much less effort in coding. First the algorithm commits to assigning  $x_1 = 0$  and recursively iterates through the  $2^{n-1}$  assignments of the remaining variables. Then the algorithm backtracks repeating these steps with the choice  $x_1 = 1$ . Viewed another way, the first little bird question about the solutions is whether the first variable  $x_1$  is set to zero or one, the second question asks about the second variable  $x_2$ , and so on. The  $2^n$  assignments of the variables  $x_1, x_2, \dots, x_n$  are associated with the  $2^n$  leaves of the complete binary tree with depth  $n$ . A given path from the root to a leaf commits each variable  $x_i$  to being either zero or one by having the path turn to either the left or to the right when reaching the  $i^{\text{th}}$  level.

**Instances and Subinstances:** Given an instance, the recursive algorithm must construct two subinstances for his friend to recurse with. There are two techniques for doing this.

**Narrowing the Class of Solutions:** Associated with each node of the classification tree is a subinstance defined as follows. The set of constraints remains unchanged except the solutions considered must be consistent in the variables  $x_1, x_2, \dots, x_r$  with the assignment given by the path to the node. Traversing a step further down the classification tree further narrows the set of solutions.

**Reducing the Instance:** Given an instance consisting of a number of constraints on  $n$  variables, we first try assigning  $x_1 = 0$ . The subinstance to be given to the first friend will be the constraints on remaining variables given that  $x_1 = 0$ . For example, if one of our original constraints is  $\langle x_1 \text{ or } \overline{x}_3 \text{ or } x_8 \rangle$ , then after assigning  $x_1 = 0$ , the reduced constraint will be  $\langle \overline{x}_3 \text{ or } x_8 \rangle$ . This is because it is no longer possible for  $x_1$  to be true, leaving that one of  $\overline{x}_3$  or  $x_8$  must be true. On the other hand, after assigning  $x_1 = 1$ , the original constraint is satisfied independent of the values of the other variables, and hence this constraint can be removed.

**Pruning:** This recursive backtracking algorithm for *SAT* can be sped up. This can either be viewed globally as a pruning off of entire branches of the classification tree or locally as seeing that some subinstances after they have been sufficiently reduced are trivial to solve.

**Pruning Off Branches The Tree:** Consider the node of the classification tree arrived at down the subpath  $x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 0, \dots, x_8 = 0$ . All of the assignment solutions consistent with this partial assignment fail to satisfy the constraint  $\langle x_1 \text{ or } \overline{x}_3 \text{ or } x_8 \rangle$ . Hence, this entire subtree can be pruned off.

**Trivial Subinstances:** When the algorithm tries to assign  $x_1 = 0$ , the constraint  $\langle x_1 \text{ or } \overline{x}_3 \text{ or } x_8 \rangle$  is reduced to  $\langle \overline{x}_3 \text{ or } x_8 \rangle$ . Assigning  $x_2 = 1$ , does not change this particular constraint. Assigning  $x_3 = 1$ , reduces this constraint further to simply  $\langle x_8 \rangle$  stating that  $x_8$  must be true. Finally, when the algorithm is considering the value for  $x_8$ , it sees from this constraint that  $x_8$  is *forced* to be one. Hence, the  $x_8 = 1$  friend is called, but the  $x_8 = 0$  friend is not.

**Stop When an Assignment is Found:** The problem specification only asks for one satisfying assignment. Hence, the algorithm can stop when one is found.

**Davis Putnum:** The above algorithm branches on the values of each variable,  $x_1, x_2, \dots, x_n$ , in order. However, there is no particular reason that this order needs to be fixed. Each branch of the recursive algorithm can dynamically use some heuristic to decide which variable to branch on next. For example, if there is a variable like  $x_8$  above whose assignment is forced by some constraint, then clearly this assignment should be done immediately. Doing so removes this variable from all the other constraints, simplifying the instance. Moreover, if the algorithm branched on  $x_4, \dots, x_7$  before the forcing of  $x_8$ , then this same forcing would need to be repeated within all  $2^4$  of these branches.

If there are no variables to force, a common strategy is to branch on the variable that appears in the largest number of constraints. The thinking is that the removal of this variable may lead to the most simplification of the instance.

An example of how different branches may set the variables in a different order is the following. Suppose that  $\langle x_1 \text{ or } x_2 \rangle$  and  $\langle \overline{x}_1 \text{ or } x_3 \rangle$  are two of the constraints. Assigning  $x_1 = 0$  will simplify the first constraint to  $\langle x_2 \rangle$  and remove the second constraint. The next step would be to force  $x_2 = 1$ . On the other hand, assigning  $x_1 = 1$  will simplify the second constraint to forcing  $x_3 = 1$ .

**Code:**

```

algorithm DavisPutnum (c)
  <pre-cond>: c is a set of constraints on the assignment to  $\vec{x}$ .
  <post-cond>: If possible, optSol is a satisfying assignment and optCost is one. Otherwise
    optCost is zero.

begin
  if( c has no constraints or no variables ) then
    % c is trivially satisfiable
    return  $\langle \emptyset, 1 \rangle$ 
  else if( c has both a constraint forcing a variable  $x_i$  to 0 and one forcing the same variable to 1 ) then
    % c is trivially not satisfiable
  
```

```

        return  $\langle \emptyset, 0 \rangle$ 
else
    for any variable forced by a constraint to some value
        substitute this value into  $c$ .
    let  $x_i$  be the variable that appears the most often in  $c$ 
    % Loop over the possible bird answers
    for  $k = 0$  to 1 (unless a satisfying solution has been found)
        % Get help from friend
        let  $c'$  be the constraints  $c$  with  $k$  substituted in for  $x_i$ 
         $\langle optSubSol, optSubCost \rangle = DavisPutnum(c')$ 
         $optSol_k = \langle \text{forced values}, x_i = k, optSubSol \rangle$ 
         $optCost_k = optSubCost$ 
    end for
    % Take the best bird answer.
     $k_{max} = \text{"a } k \text{ that maximizes } optCost_k"$ 
     $optSol = optSol_{k_{max}}$ 
     $optCost = optCost_{k_{max}}$ 
    return  $\langle optSol, optCost \rangle$ 
end if
end algorithm

```

**Running Time:** If no pruning is done, then clearly the running time is  $\Omega(2^n)$  as all  $2^n$  assignments are tried. Considerable pruning needs to occur to make the algorithm polynomial time. Certainly in the worst case, the running time is  $2^{\Theta(n)}$ . In practice, however, the algorithm can be quite fast. For example, suppose that the instance is chosen randomly by choosing  $m$  constraints, each of which is the *or* of three variables or there negations, eg.  $\langle x_1 \text{ or } \overline{x}_3 \text{ or } x_8 \rangle$ . If few constraints are chosen, say  $m$  is less than about  $3n$ , then with very high probability there are many satisfying assignments and the algorithm quickly finds one of these assignments. If a lot of constraints are chosen, say  $m$  is at least  $n^2$ , then with very high probability there are many conflicting constraints preventing there from being any satisfying assignments and the algorithm quickly finds one of these contradictions. On the other hand, if the number of constraints chosen is between these thresholds, then it has been proven that the Davis Putnum algorithm takes exponential time.

### 25.3 Scrabble

Consider the following scrabble problem. An instance consists of a set of letters and a dictionary. A solution consists of a permutation of a subset of the given letters. A solution is valid if it is in the dictionary. The value of a solution is given by its placement on the board. The goal is to find a highest point word that is in the dictionary.

The simple brute force algorithm searches the dictionary for each permutation of each subset of the letters. The back-tracking algorithm tries all of the possibilities for the first letter and then recurses. Each of these stack frames tries all of the remaining possibilities for the second letter, and so on. This can be pruned by observing that if the word constructed so far, eg. “xq”, does not match the first letters of any word in the dictionary, then there is no need for this stack frame to recurse any further. (Another improvement on the running time ensures that the words are searched for in the dictionary in alphabetical order.)

## 25.4 Queens

The following is a fun exercise to help you trace through a recursive backtracking algorithm. It is called the Queen's Problem. Physically get yourself (or make on paper) a chess board and eight tokens to act as queens. The goal is to place all eight queens on the board in a way such that no pieces moving like queen along a row, column, or diagonal is able to capture any other piece.

The recursive backtracking algorithm is as follows. First it observes that each of the eight rows can have at most one queen or else they will capture each other. Hence each row must have one of the eight queens. Given a placement of queens in the first few rows, a stack frame tries each of the legal placements of a queen in the next row. For each such placements, the algorithm recurses.

**Code:**

```
algorithm Queens ( $C, row$ )
⟨pre-cond⟩:  $C$  is a chess board containing a queen on the first  $row - 1$  rows in such a way
    that no two can capture each other. The remaining rows have no queen.
⟨post-cond⟩: All legal arrangements of the 8 queens consistent with this initial placement
    are printed out.

begin
    if(  $row > 8$  ) then
        print( $C$ )
    else
        loop  $col = 1 \dots 8$ 
            Place a queen on location  $C(row, col)$ 
            if( this creates a conflict) then
                Do not pursue this option further. Do not recurse.
                Note this prunes off this entire branch of the recursive tree
            else
                Queens ( $C, row + 1$ )
            end if
            backtrack removing the queen from location  $C(row, col)$ 
        end loop
    end if
end algorithm
```

Trace this algorithm. It is not difficult to do because there is an active stack frame for each queen currently on the board. You start by placing a queen on each row, one at a time, in the left most legal position until you get stuck. Then whenever a queen cannot be placed on a row or moves off the right of a row, you move the queen on the row above until it is in the next legal spot.

**Exercise 25.4.1** (*See solution in Section V*) *Trace this algorithm. What are the first dozen legal outputs for the algorithm. To save time record the positions stated in a given output by the vector  $\langle c_1, c_2, \dots, c_8 \rangle$  where for each  $r \in [1..8]$  there is a queen at location  $C(r, c_r)$ . To save more time, note that the first two or three queens do not move so fast. Hence, it might be worth it to draw a board with all squares conflicting with these crossed out.*

**Exercise 25.4.2** (*See solution in Section V*) *Consider the same Queens algorithm placing  $n$  queens on an  $n$  by  $n$  board instead of only 8. Give a reasonable upper and lower bound on the running time of this algorithm after all the pruning occurs.*

# Chapter 26

## Reductions

A giraffe with its long neck is a very different beast than a mouse, which is different from a snake. However, Darwin and gang observed that the first two have some key similarities, both being social, nursing their young, and having hair. The third is completely different in these ways. Studying similarities and differences between things can reveal subtle and deep understandings of their underlining nature that would not have been noticed by studying them one at a time. Sometimes things that at first appear to be completely different, when viewed slightly differently, turn out to be the same except for superficial cosmetic differences. This section will teach how use reductions to discover these similarities between different optimization problems.

**Reduction  $P_1 \leq_{poly} P_2$ :** A reduction consists of writing a polynomial ( $n^{\Theta(1)}$ ) time algorithm for one of the problems  $P_1$  using a supposed algorithm for the other problem  $P_2$  as a subroutine. (Note we may or may not actually have an algorithm for  $P_2$ .) The standard notation for this is  $P_1 \leq_{poly} P_2$ .

**Conclusions:** A reduction lets us compare the time complexities and underlying structures of these two problems.

**Upper Bounds:** From the reduction  $P_1 \leq_{poly} P_2$  alone, we cannot conclude that there is a polynomial time algorithm for  $P_1$ . But it does tell us that if there is a polynomial time algorithm for  $P_2$ , then there is one for  $P_1$ . This is useful in two ways. First, it allows us to construct algorithms for new problems from known algorithms for other problems. Moreover, it tells us that  $P_1$  is “at least as easy as”  $P_2$ .

**Hotdogs  $\leq_{poly}$  Linear Programming:** Section 21.4 describes how to solve the problem of making a cheap hotdog using an algorithm for solving linear programming.

**Bipartite Matching  $\leq_{poly}$  Network Flows:** We will give an algorithm for Bipartite Matching in Section 26.5 that uses the network flows algorithm.

**Lower Bounds:** The counter positive of the last statement is that if there is not a polynomial time algorithm for  $P_1$ , then there cannot be one for  $P_2$  (otherwise there would be one for  $P_1$ .) This tells us that  $P_2$  is “at least as hard as”  $P_1$ . In particular, if we already suspect that there is no polynomial time algorithm for  $P_1$ , then it follows that there is not likely one for  $P_2$ , either.

**(Any Optimization Problem)  $\leq_{poly}$  SAT:** We already suspect that there is at least one optimization problem that is hard. This gives strong evidence that SAT is hard. See Section 26.1.

**SAT  $\leq_{poly}$  3-COL:** This gives evidence that 3-Colouring is also hard. See Section 26.3.

**3-COL  $\leq_{poly}$  Course Scheduling,**

**3-COL  $\leq_{poly}$  Independent Set,**

**3-COL  $\leq_{poly}$  3-SAT:**

These give evidence that Course Scheduling, Independent Set, and 3-SAT are hard.

See Sections 26.2 and 26.3.

**Factoring  $\leq_{poly}$  Decoding:** We already suspect that Factoring is hard. This gives evidence that Decoding encrypted messages is hard. See Section 26.4.

**Halting Problem  $\leq_{poly}$  (What Does This TM Do):** It can be proved that the Halting Problem (given a Turing Machine  $M$  and an input  $I$ , does the  $M$  halt on  $I$ ) is undecidable (no algorithm can always answer correctly in finite time). Such a reduction can be used to prove that most any problem asking what the computation of a given Turing Machine does is also undecidable.

**Reverse Reductions:** Knowing  $P_1 \leq_{poly} P_2$  and knowing that there is not a polynomial time algorithm for  $P_2$  does not tell us anything about the whether there is a polynomial time algorithm for  $P_1$ . Though it does tell us that the algorithm for  $P_1$  given in the reduction does not work, there well may be another completely different algorithm for  $P_1$ . Similarly, knowing that there is a polynomial time algorithm for  $P_1$  does not tell us anything about the whether there one for  $P_2$ . To make these two conclusions, you must prove the reverse reduction  $P_2 \leq_{poly} P_1$ .

**The Same Problem except for Superficial Differences:** More than just being able to compare their time complexities, knowing proving  $P_1 \leq_{poly} P_2$  and  $P_2 \leq_{poly} P_1$  reveals that the two problems are some how fundamentally at their core the same problem, asking the same types of questions. Sometimes this similarity is quite superficial. They simply use different vocabulary. However, at other times this connection between the problems is quite surprising, revealing a deeper understanding about each of the problems. One way in which we can make a reduction even more striking is by restricting the algorithm for the one to call the algorithm for the other only once. Then the mapping between them is even more direct.

**NP-Complete:** The above statements prove that the problems SAT, 3-Colouring, Course Scheduling, Independent Set, and 3-SAT are all fundamentally the same problem. ((Any Optimization Problem)  $\leq_{poly}$  SAT gives that each is  $\leq_{poly}$  SAT and then transitivity gives that if  $P_1 \leq_{poly} P_2$  and  $P_2 \leq_{poly} P_3$ , then  $P_1 \leq_{poly} P_3$ .) In fact there are thousands of very different problems that are equivalent to these. These problems are said to be *NP-Complete*.

## 26.1 Satisfiability Is At Least As Hard As Any Optimization Problem

We have already seen that *optimization problems* form an important and practical class of computational problems. These involve searching through the exponential set of solutions for the instance to find one with optimal cost. See Section 19 for a more formal definition. Though there are quick algorithms for some of these problems, for most of them the best known algorithms require  $2^{\Theta(n)}$  time on the worst case input instances and it is strongly believed that there are not polynomial time algorithms for them. The main reason for this belief is that many smart people have devoted

many years of research into looking for fast algorithms and have not found them. To help justify this and to unify their efforts, this section uses reductions to prove that some of these optimization problems are universally hard or *complete* amongst the class of optimization problems because if you could design an algorithm to solve such a problem quickly, then you could translate this algorithm into one that solves any optimization problem quickly. Conversely, (and more likely) if there is even one optimization problem that cannot be solved quickly, then none of these complete problems can be either. This can be useful for you. If your boss gives you a problem for which you cant find an quick solution then perhaps you can prove that it is at least as hard as all these other problems that no one has yet been able to solve. We will now demonstrate how such reductions are done.

**(Any Optimization Problem)  $\leq_{poly}$  SAT:** This reduction will prove the satisfiability problem is complete for the class of optimization problems, meaning that is universally hard for this class.

**The Satisfiability Problem:** The famous computational problem, *satisfiability (SAT)*, is required to find a satisfying assignment for a given circuit. Section 25.2 gives a recursive backtracking algorithm for this problem, but in the worst case its running time is  $2^{\Theta(n)}$ .

**Circuit:** A circuit is a both a useful notation for describing an algorithm in detail and a practical thing built in silicon in your computer.

**Construction:** It is built with *AND*, *OR*, and *NOT* gates. At the top are  $n$  wires labeled with the binary variables  $x_1, x_2, \dots, x_n$ . To specify the circuit's input, each of these will take on either 1 or 0, *true* or *false*, 5 volts or 0 volts. Each *AND* gate has two wires coming into it, either from an input  $x_i$  or from the output of another gate. An *AND* gate outputs *true* if both of its inputs are *true*. Similarly, each *OR* gate outputs *true* if at least one of its inputs is *true* and each *NOT* gate outputs *true* if its single input is *false*. We will only consider circuits that have no cycles, so these *true/false* values percolate down to the output wires. There will be a single output wire if the circuit computes a *true/false* function of its input  $x_1, x_2, \dots, x_n$  and will have  $m$  output wires if it outputs an  $m$  bit string which can be used to encode some required information.

**Compute Any Function:** Given any function  $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$ , a circuit can compute it with a most  $\Theta(nm \cdot 2^n)$  gates as follows. For any fixed input instance  $\langle x_1, x_2, \dots, x_n \rangle = \langle 1, 0, \dots, 1 \rangle$ , a circuit can say “the input is this fixed instance” simply by computing  $[(x_1 = 1) \text{ AND } NOT(x_2 = 1) \text{ AND } \dots \text{ AND } (x_n = 1)]$ . Then the circuit computes the  $i^{th}$  bit of the function's output by outputting 1 if [“the input is this fixed instance” *OR* “this instance” *OR* … *OR* “this instance”], where each instance is listed for which the  $i^{th}$  bit of the function's output is 1.

**Poly Size for Poly Time:** More importantly, given any algorithm whose Turing Machine's running time is  $T(n)$  and given any fixed integer  $n$ , there is an easily constructed circuit with at most  $\Theta(T(n)^2)$  gates that computes the output of the algorithm given any  $n$  bit input instance. Change the definition of a Turing machine slightly so that each cell is big enough so that the cell currently being pointed to by the head can store not only its current cell's contents but also the current state of the machine. This cell contents can be encoded with  $\Theta(1)$  bits. Because the Turing machine uses only  $T(n)$  time, it can use at most the first  $T(n)$  cells of memory. For

each of the  $T(n)$  steps of the algorithm, the circuit will have a row of  $\Theta(1) \cdot T(n)$  wires whose values encode the contents of memory of these  $T(n)$  cells during this time step. The gates of the circuit between these rows of wires compute the next contents of memory from the previous. Because the contents of cell  $i$  at time  $t$  depends only on the contents of cells  $i - 1$ ,  $i$ , and  $i + 1$  at time  $t - 1$  and each of these is only a  $\Theta(1)$  number of bits, this dependency can be computed using a circuit with  $\Theta(1)$  gates. This is repeated in a matrix of  $T(n)$  time steps and  $T(n)$  cells for a total of  $\Theta(T(n)^2)$  gates. At the bottom, the circuit computes the output of the function from the contents of memory of the Turing machine at time  $T(n)$ .

**Satisfiability Specification:** The Satisfiability problem takes as input a circuit with a single *true/false* output and return an assignment to the variables  $x_1, x_2, \dots, x_n$  for which the circuit gives *true*, if such an assignment exists.

**Optimization Problems:** This reduction will select a generic optimization problem and show that Satisfiability is at least as hard as it is. To do this, we need to have a clear definition of what a generic optimization problem looks like.

**Definition:** Each such problem has a set instances that might be given as input, each instance has a set of potential solutions some of which are valid, and each solution has a cost. The goal, given an instance, is to find one of its valid solutions with optimal cost. An important feature of an optimization problem is that there are polynomial time algorithms for the following.

**Valid( $I, S$ ):** Given an instance  $I$  and a potential solution  $S$ , there is an algorithm  $Valid(I, S)$  running in time  $|I|^{\mathcal{O}(1)}$  that determines if  $I$  is a valid instance for the optimization problem and that  $S$  is a valid solution for  $I$ .

**Cost( $S$ ):** Given a valid solution  $S$ , there is an algorithm  $Cost(S)$  running in time  $|I|^{\mathcal{O}(1)}$  that computes the cost of the solution  $S$ .

#### Examples:

**Graph Colouring:** Given a graph, colour the nodes of the graph so that two nodes do not have the same colour if they have an edge between them. Use as few colours as possible.

**Course Scheduling:** Given the set of courses requested by each student and the set of time slots available, find a schedule which minimizes the number of conflicts.

**Independent Set:** Given a graph, find a largest subset of the nodes for which there are no edges between any pair in the set.

**Airplane:** Given the requirements of a plane, design it optimizing its performance.

**Business:** Given a description of the business, make a business plan to maximize its profits.

**Factoring:** Given an integer, factor it, eg.  $6 = 2 \times 3$ .

**Cryptography:** Given an encrypted message, decode it.

**Exercise 26.1.1** For each of these problems, define  $I$ ,  $S$ ,  $Valid(I, S)$  and  $Cost(S)$ .

**Alg easier for the Optimization Problem:** We prove that the problem Satisfiability is “at least as hard as” the optimization problem as follows. Given a fast algorithm  $Alg_{harder}$  for Satisfiability and the descriptions  $Valid(I, S)$  and  $Cost(S)$  of an optimization problem, we will now design a fast algorithm  $Alg_{easier}$  for the optimization problem.

**Binary Search for Cost:** Given some instance  $I_{easier}$  to the optimization problem,  $Alg_{easier}$ 's first task is to determine the cost  $c_{opt}$  of the optimal solution for  $I$ .  $Alg_{easier}$  starts by determining whether or not there is a valid solution for  $I$  that has cost at least  $c = 1$ . If it does,  $Alg_{easier}$  repeats this with  $c = 2, 4, 8, 16, \dots$ . If it does not,  $Alg_{easier}$  tries  $c = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \dots$ , until it finds  $c_1$  and  $c_2$  between which it knows the cost of an optimal solution lies. Then it does binary search to find  $c_{opt}$ . The last step is to find a solution for  $I$  that has this optimal cost.

**Finding a Solution with given Cost:**  $Alg_{easier}$  determines whether  $I$  has a solution  $S$  with cost at least  $c$  or finds a solution with cost  $c_{opt}$  as follows.  $Alg_{easier}$  constructs a circuit  $C$  and calls the algorithm  $Alg_{harder}$ , which provides a satisfying assignment to  $C$ .  $Alg_{easier}$  wants the satisfying assignment that  $Alg_{harder}$  provides to be the solution  $S$  that it needs. Hence,  $Alg_{easier}$  designs  $C$  to be satisfied by the assignment  $S$  only if  $S$  is a solution  $S$  for  $I$  with a cost as required, i.e.,  $C(S) \equiv [Valid(I, S) \text{ and } Cost(S) \geq c]$ . Because there are polynomial time algorithms for  $Valid(I, S)$ , for  $Cost(S)$ , and for  $\geq$ ,  $Alg_{easier}$  can easily construct such a circuit  $C(S)$ . If a such solution  $S$  satisfying  $C$  exists, then  $Alg_{harder}$  kindly provides one.

This completes the reduction from Satisfiable to any Optimization problem.

## 26.2 Steps to Prove NP-Completeness

This section defines the class NP and gives steps for proving that a computational problem is NP-complete.

**Complete for Non-Deterministic Polynomial Time Decision Problems:** The set of computational problems that are complete (universally hard) for optimization problems is extremely rich and varied. Studying them has become a fascinating field of research. We will review the definitions.

**NP Decision Problems:** Theoreticians generally only consider a subclass of the optimization problems referred to as the class of *Non-Deterministic Polynomial Time* Problems (NP).

**One Level of Cost:** Instead of worrying about whether one solution has a better cost than another, we will completely drop the notion of the cost of a solution.  $S$  will only be considered to be a “valid solution” for the instance  $I$  if it is a solution with a sufficiently good cost. This is not a big restriction, because if you want to consider solutions with different costs, you can always do binary search as done above for the cost of the optimal solution.

**Decision Problems:** Theoreticians classically take this idea a step further by not be expecting the algorithm to return a solution for an instance but only decide whether *Yes* the instances does have a solution or *No* it does not. We, on the other hand, in order to help with our understanding, have our algorithms return the solution as well.

**Witness:** A solution for an instance is often referred to as a *witness*, because though it may take exponential time to find it, if it were provided by a (non-deterministic) fairy god mother, then it can be used in polynomial time to witness the fact that the answer for this instance is yes. In this way, NP problems are asymmetrical because

there does not seem to be a witness that quickly proves that an instances does not have solution.

**Formal Definition:** We say that such a computational problem  $P$  is in the class of *Non-Deterministic Polynomial Time* Problems (NP) if there is a polynomial time algorithm  $Valid(I, S)$  that specifies *Yes* when  $S$  is a (sufficiently good) solution for the instance  $I$  and *No*, if not. More formally,  $P$  can be defined as follows.

$$P(I) \equiv [\exists S, Valid(I, S)]$$

**Examples:**

**Satisfiability (SAT):** Satisfiability was initially defined as a decision problem.

Given a circuit, determine whether there is an assignment that satisfies it.

**Graph 3-Colouring (3-COL):** Given a graph, determine whether its nodes can be coloured with three colours so that two nodes do not have the same colour if they have an edge between them.

**Course Scheduling:** Given the set of courses requested by each student, the set of time slots available, and an integer  $K$ , determine whether there is schedule with at most  $K$  conflicts.

**Cook vs Karp Reductions:** Stephen Cook first proved that Satisfiability is complete for the class of NP-problems. His definition of a reduction  $P_{easier} \leq_{poly} P_{harder}$  is, as said above, that one can write an algorithm  $Alg_{easier}$  for the problem  $P_{easier}$  using an algorithm  $Alg_{harder}$  for the problem  $P_{harder}$  as a subroutine. In general, this algorithm  $Alg_{easier}$  could call  $Alg_{harder}$  as many times as it likes and do anything it wants with the answers that it receives. Richard Karp later observed that when the problems  $P_{easier}$  and  $P_{harder}$  are more similar in nature then the algorithm  $Alg_{easier}$  used in the reduction need only call  $Alg_{harder}$  once and answers *Yes* if and only if  $Alg_{harder}$  answers *Yes*. These two definitions of reductions are referred to as *Cook* and *Karp* reductions. Though we defined Cook reductions above because they more natural, only Karp reductions will be considered from here on.

**NP-Complete:** We say that a computational problem  $P_{harder}$  is *NP-complete* if

1. it is in NP and
  2. every language in NP can be polynomially reduced to it using a Karp reduction.
- More formally,

$$\forall \text{ Optimization Problems } P_{easier}, P_{easier} \leq_{poly} P_{harder}.$$

To prove this, it is sufficient to prove that that it is at least as hard as some problem already known to be NP-complete. For example, because we now know that Satisfiability is NP-complete, it is sufficient to prove that  $SAT \leq_{poly} P_{harder}$ .

**The Steps to Prove NP-Complete:** Proving problems to be NP-complete is a bit of an art, but once you get the hang of it, they can be quite fun problems to solve. We will now carefully lay out the steps needed.

**Course Scheduling is NP-Complete:** As as simple running example, we will prove that  $Alg_{harder} = \text{Course Scheduling problem}$  is NP-complete.

- 0)  $P_{harder} \in \text{NP}$ : The first step is to prove that problem  $P_{harder}$  is in NP by providing the polynomial time algorithm  $\text{Valid}(I_{harder}, S_{harder})$  that specifies whether  $S_{harder}$  is a valid solution for instance  $I_{harder}$ .

**Course Scheduling:** It is not hard to determine in polynomial time whether the instance  $I_{harder}$  and the solution  $S_{harder}$  are properly defined and to check that within this schedule  $S_{harder}$ , the number of times that a student wants to take two courses that are offered at the same time is at most  $K$ .

- 1) **What To Reduce It To:** An important and challenging step in proving that a problem is NP-complete is deciding which NP-complete problem to reduce it to.

**3-COL  $\leq_{poly}$  Course Scheduling:** We will reduce Course Scheduling to 3-COL, namely prove the reduction  $3\text{-COL} \leq_{poly} \text{Course Scheduling}$ . We will save the proof that 3-COL is NP-complete for our next example because it is much harder.

**Hint:** You want to choose a problem that is “similar” in nature to yours. In order to have more to choose from, it helps to know a large collection of problems that are NP-complete. There are entire books devoted to this. When in doubt 3-SAT and 3-Col are good problems to use.

- 2) **What is What:** It is important to remember what everything is.

**3-COL  $\leq_{poly}$  Course Scheduling:**

$P_{easier} = \text{3COL}$  is the Graph 3-Colouring problem.

$I_{easier} = I_{graph}$ , an instance to it, is an undirected graph.

$S_{easier} = S_{colouring}$ , a potential solution, is a colouring of each of its nodes with either red, blue or green. It is a valid solution if no edge has two nodes with the same colour.

$\text{Alg}_{easier}$  is an algorithm that takes graph  $I_{graph}$  as input and determines whether it has a valid colouring.

$P_{harder} = \text{Course Scheduling}$

$I_{harder} = I_{courses}$ , an instance to it, is the set of courses requested by each student, the set of time slots available, and the integer  $K$ .

$S_{harder} = S_{schedule}$ , a potential solution, is a schedule assigning courses to time slots. It is a valid solution if it has at most  $K$  conflicts.

$\text{Alg}_{harder}$  is an algorithm that takes  $I_{courses}$  as input and determines whether it has a valid schedule.

Note that before proved as such, such instances which may or may be satisfiable and such potential solutions may or may not be valid.

**Warning:** At least a quarter of the incorrect reductions that I have marked are confused about what is an instance and what is a solution for each of the two problems.

- 3) **Direction of Reduction and Code:** Another common source of mistakes is doing the reduction in the wrong direction. I recommend not memorizing this direction, but working it out each time. Our goal is to prove that the problem  $P_{harder}$  is at least as hard as  $P_{easier}$ , namely  $P_{easier} \leq_{poly} P_{harder}$ . We prove that  $P_{easier}$  is relatively easy by designing a fast algorithm  $\text{Alg}_{easier}$  for it using a supposed fast algorithm  $\text{Alg}_{harder}$  for  $P_{harder}$ . (But there is likely not a fast algorithm for  $P_{easier}$  and hence there is not likely one for  $P_{harder}$ .) The code for our algorithm for  $P_{easier}$  will be as follows.

**algorithm**  $\text{Alg}_{easier}(I_{easier})$

**$\langle \text{pre-cond} \rangle$ :**  $I_{easier}$  is an instance of  $P_{easier}$ .

**$\langle \text{post-cond} \rangle$ :** Determine whether  $I_{easier}$  has a solution  $S_{easier}$  and if so returns it.

begin

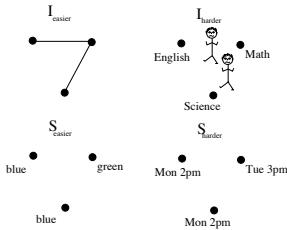
```

 $I_{harder} = InstanceMap(I_{easier})$ 
 $\langle ans_{harder}, S_{harder} \rangle = Alg_{harder}(I_{harder})$ 
if(  $ans_{harder} = Yes$  ) then
     $ans_{easier} = Yes$ 
     $S_{easier} = SolutionMap(S_{harder})$ 
else
     $ans_{easier} = No$ 
     $S_{easier} = nil$ 
end if
return( $\langle ans_{harder}, S_{harder} \rangle$ )
end algorithm

```

- 4) **Look For Similarities:** Though the problems  $P_{easier}$  and  $P_{harder}$  may appear to be very different, the goal in this step is to look for underlying similarities. Compare how their solutions,  $S_{easier}$  and  $S_{harder}$ , are formed out of their instances,  $I_{easier}$  and  $I_{harder}$ . Generally, the instances can be thought of as sets of elements with a set of constraints between them. Can you view their solutions as subsets of these elements or as a labeling of them? What allows a solution to form and what constrains how it is formed. Can you talk about the two problems using the same language? For example, a subset of elements can be viewed as a labeling of the elements with zero and one. Similarly, a labeling of each element  $e$  with  $\ell_e \in [1..L]$  can be viewed as a subset of the pairs  $\langle e, \ell_e \rangle$ .

**3-COL  $\leq_{poly}$  Course Scheduling:** A solution  $S_{colouring}$  is a colouring, which assigns a colour to each node. A solution  $S_{schedule}$  is a schedule, which assigns a time slot to each course. This similarity makes it clear that there is a similarity between the roles of the nodes of  $I_{graph}$  and of the courses of  $I_{courses}$  and between the colours of  $S_{colouring}$  and the time slots of  $S_{schedule}$ . Each colouring conflict arise from a edge between nodes and each scheduling conflicts arises from a student wanting two courses. This similarity makes it clear that there is a similarity between the roles of the edges of  $I_{graph}$  and of the course requests of  $I_{courses}$ .



- 5) **InstanceMap:** You must define a polynomial time algorithm  $InstanceMap(I_{easier})$  that, given an instance  $I_{easier}$  of  $P_{easier}$ , constructs an instance  $I_{harder}$  of  $P_{harder}$  that has “similar” sorts of solutions. The main issue is that the constructed instance  $I_{harder}$  has a solution if and only if the given instance  $I_{easier}$  has a solution, i.e. *Yes* instances get mapped to *Yes* instances and *No* to *No*.

**3-COL  $\leq_{poly}$  Course Scheduling:** Given a graph  $I_{graph}$  to be coloured, we design an instance  $I_{courses} = InstanceMap(I_{graph})$  to be scheduled. Using the similarities observed in step 4, our mapping we will have one course for each node of the graph and one time slot for each of the three colours green, red, and blue. For each edge between nodes  $u$  and  $v$  in the graph, we will have a student who requests both course  $u$  and course  $v$ . The colouring problem does not allow any conflicts. Hence, we set  $K = 0$ .

**Not Onto or 1-1:** It is important that each instance  $I_{easier}$  is mapped to some instance  $I_{harder}$ , but it is not important whether an instance  $P_{harder}$  is mapped to more than once or not at all. Above, for example, we never mention instances to be scheduled that have more than three time slots or that allow  $K > 0$  conflicts.

**Warning:** Be sure to do this mapping in the correct direction. The first step in designing an algorithm  $Alg_{easier}$  is to suppose that you have been given an input  $I_{easier}$  for it. Before your algorithm can call the algorithm  $Alg_{harder}$  as a subroutine, your must construct an instance  $I_{easier}$  to give to it.

**Warning:** A common mistake is to define the mapping only for the *Yes* instances, i.e., those with solutions or to use a solution  $S_{easier}$  for  $I_{easier}$  for determining the instance  $I_{harder}$  mapped to. Recall that the algorithm  $Alg_{easier}$  that you are designing is given an instance  $I_{easier}$ , but it does not know whether or not it has a solution and it certainly does not have a solution. The whole point of this to give an argument that finding a solution may take exponential time. It is safer when defining the mapping  $InstanceMap(I_{easier})$  not to even mention whether the instance  $I_{easier}$  as a solution or what that solution might be.

**6) SolutionMap:** You must also define a polynomial time algorithm  $SolutionMap(S_{harder})$  mapping each valid solution  $S_{harder}$  for the instance  $I_{harder} = InstanceMap(I_{easier})$  you just constructed to a valid solution  $S_{easier}$  for instance  $I_{easier}$  that was given as input. The difficulty is that valid solutions can be subtle and the instance  $I_{harder}$  may have some that you had not intended when you constructed it. One way to help avoid missing some is to throw a much wider net by considering all “potential solutions.” In this step, for each potential solution  $S_{harder}$  for  $I_{harder}$ , you must either give a reason why it is not a valid solution or map it to a solution  $S_{easier} = SolutionMap(S_{harder})$  for  $I_{easier}$ . It is fine if some of the solutions that you map happen not to be valid.

**3-COL  $\leq_{poly}$  Course Scheduling:** Given a schedule  $S_{schedule}$  assigning course  $u$  to time slots  $c$ , we define  $S_{colouring} = SolutionMap(S_{schedule})$  to be the colouring that colours node  $u$  with colour  $c$ .

**Warning:** When the instance  $I_{harder}$  you constructed has solutions that you did not expect, there are two problems. First, the unknown algorithm  $Alg_{harder}$  may give you one of these unexpected solutions. Second, there is a danger that  $I_{harder}$  has solutions but your given instance  $I_{easier}$  does not. For example, if in step 5, our  $I_{harder}$  allowed more than three time slots or more than  $K = 0$  conflicts, then the instance may have many unexpected solutions. In such cases, you may have to redo step 5 adding extra constraints to the instance  $I_{harder}$  so that it no longer has these solutions.

**7) Valid to Valid:** In order to prove that the algorithm  $Alg_{easier}(I_{easier})$  works, you must prove that if  $S_{harder}$  is a valid solution for  $I_{harder} = InstanceMap(I_{easier})$ , then  $S_{easier} = SolutionMap(S_{harder})$  is a valid solution for  $I_{easier}$ .

**3-COL  $\leq_{poly}$  Course Scheduling:** Supposing that the schedule is valid, we prove that the colouring is valid as follows. The instance to be scheduled is constructed so that, for each edge of the given graph, there is a student who requests the courses  $u$  and  $v$  associated with the nodes of this edge. Because the schedule is valid, there are  $K = 0$  course conflicts, and hence these courses are all scheduled at different time slots. The constructed colouring, therefore allocates different colours to these nodes.

**Warning:** Before proving step 7, be sure that the mappings from steps 5 and 6 are not ambiguous, otherwise, there is the danger of interpreting them one way in one part of the proof and in another way in another part. In order to help avoiding this, I strongly recommend separating all of the steps into clearly separated paragraphs. I have read too many student's solutions in which the steps are mixed together making it harder to follow and much more prone to problems.

- 8) **ReverseSolutionMap:** Though we do not need it for the code, for the proof you must define an algorithm  $ReverseSolutionMap(S'_{easier})$  mapping in the reverse direction from each “potential” solution  $S'_{easier}$  for instance  $I_{easier}$  to a potential solution  $S'_{harder}$  for instance  $I_{harder}$ .

**3-COL  $\leq_{poly}$  Course Scheduling:** Given a colouring  $S'_{colouring}$  colouring node  $u$  with colour  $c$ , we define  $S'_{schedule} = ReverseSolutionMap(S'_{colouring})$  to be the schedule assigning course  $u$  to time slots  $c$ .

**Warning:**  $ReverseSolutionMap(S'_{easier})$  does not need to be the inverse map of  $SolutionMap(S_{harder})$ . In fact, it is important that you define the mapping  $ReverseSolutionMap(S_{easier})$  for every possible solution  $S'_{easier}$ , not just those mapped to by  $SolutionMap(S_{harder})$ . Otherwise, there is the danger is that  $I_{easier}$  has solutions but your constructed instance  $I_{harder}$  does not.

- 9) **Reverse Valid to Valid:** You must also prove the reverse direction that if  $S'_{easier}$  is a valid solution for  $I_{easier}$ , then  $S'_{harder} = ReverseSolutionMap(S'_{easier})$  is a valid solution for  $I_{harder} = InstanceMap(I_{easier})$ .

**3-COL  $\leq_{poly}$  Course Scheduling:** Supposing that the colouring is valid, we prove that the schedule is valid as follows. The instance to be scheduled is constructed so that each student requests the courses  $u$  and  $v$  associated with nodes of some edge. Because the colouring is valid, these nodes have been allocated different colours and hence the courses are all scheduled different time slots. Hence, there will be  $K = 0$  course conflicts.

- 10) **Working Algorithms:** Given the above steps, it is now possible to prove that if the supposed algorithm  $Alg_{harder}$  correctly solves  $P_{harder}$ , then our algorithm  $Alg_{easier}$  correctly solves  $P_{easier}$ .

**Yes to Yes:** We start by proving that  $Alg_{easier}$  answers *Yes* when given an instance for which the answer is *Yes*. If  $I_{easier}$  is a *Yes* instance, then by the definition of the problem  $P_{easier}$ , it must have a valid solution. Let us denote by  $S'_{easier}$  one such valid solution. Then by step 9, it follows that  $S'_{harder} = ReverseSolutionMap(S'_{easier})$  is a valid solution for  $I_{harder} = InstanceMap(I_{easier})$ . This witnesses the fact that  $I_{harder}$  has a valid solution and hence  $I_{harder}$  is an instance for which the answer is *Yes*. If  $Alg_{harder}$  works correctly as supposed, then it returns *Yes* and a valid solution  $S_{harder}$ . Our code for  $Alg_{easier}$  will then return the correct answer *Yes* and  $S_{easier} = SolutionMap(S_{harder})$ , which by step 7 is a valid solution for  $I_{easier}$ .

**No to No:** We must now prove the reverse, that if the instance  $I_{easier}$  given to  $\text{Alg}_{easier}$  is a *No* instance, then  $\text{Alg}_{easier}$  answers *No*. The problem with *No* instances is that they have no witness to prove that they are *No* instances. Luckily, to prove something, it is sufficient to prove the contrapositive. Namely instead of proving  $A \Rightarrow B$ , where  $A = "I_{easier} \text{ is a } No \text{ instance}"$  and  $B = "\text{Alg}_{easier} \text{ answers } No"$ , we will prove that  $\neg B \Rightarrow \neg A$ , where  $\neg B = "\text{Alg}_{easier} \text{ answers } Yes"$  and  $\neg A = "I_{easier} \text{ is a } Yes \text{ instance.}"$  Convince yourself that this is equivalent.

If  $\text{Alg}_{easier}$  is given the instance  $I_{easier}$  and answers *Yes*, our code is such that  $\text{Alg}_{harder}$  must have returned *Yes*. If  $\text{Alg}_{harder}$  works correctly as supposed, the instance  $I_{harder} = \text{InstanceMap}(I_{easier})$  that it was given must be a *Yes* instance. Hence,  $I_{harder}$  must have a valid solution. Let us denote by  $S_{harder}$  one such valid solution. Then by step 7,  $S_{easier} = \text{SolutionMap}(S_{harder})$  is a valid solution for  $I_{easier}$ , witnessing  $I_{easier}$  being a *Yes* instance. This is the required conclusion  $\neg A$ .

This completes the proof that if the supposed algorithm  $\text{Alg}_{harder}$  correctly solves  $P_{harder}$ , then our algorithm  $\text{Alg}_{easier}$  correctly solves  $P_{easier}$ .

- 11) Running Time:** The remaining step is to prove that the constructed algorithm  $\text{Alg}_{easier}$  runs in polynomial ( $|I_{easier}|^{\Theta(1)}$ ) time. Steps 5 and 6 require that both  $\text{InstanceMap}(I_{easier})$  and  $\text{SolutionMap}(S_{harder})$  work in polynomial time. Hence, if  $P_{harder}$  can be solved “quickly”, then  $\text{Alg}_{easier}$  runs in polynomial time. Typically for reductions people assume that  $\text{Alg}_{harder}$  is an *oracle* meaning that it solves its problem in one time step. Exercise 26.2.5 explores the issue of running time further.

This concludes the proof that  $P_{harder} = \text{Course Scheduling}$  is NP-complete, (assuming of course that that  $P_{easier} = 3\text{-COL}$  has already been proven to be NP-complete.)

**Exercise 26.2.1** We began this section by proving (Any Optimization Problem)  $\leq_{poly} SAT$ . To make this proof more concrete redo it completing each of the above steps specifically for  $3\text{-COL} \leq_{poly} SAT$ . Hint: The circuit  $I_{harder} = \text{InstanceMap}(I_{easier})$  should have a variable  $x_{\langle u, c \rangle}$  for each pair  $\langle u, c \rangle$ .

**Exercise 26.2.2**  $3\text{-SAT}$  is a subset of the Satisfiability problem in which the input circuit must be a big AND of clauses, each clause must be the OR of at most three literals, and each literal is either a variable or its negation. Prove that  $3\text{-SAT}$  is NP-complete by proving that  $3\text{-COL} \leq_{poly} 3\text{-SAT}$ . Hint: The answer is almost identical to that for Exercise 26.2.1.

**Exercise 26.2.3** Let  $\overline{SAT}$  be the complement of the Satisfiability problem, namely the answer is Yes if and only if the input circuit is not satisfiable. Can you prove  $SAT \leq_{poly} \overline{SAT}$  using Cook reductions. Can you prove it using Karp reductions?

- Answer: Using Cook reductions it is easy.  $\text{InstanceMap}(I)$  simply maps each instance  $I$  of SAT to itself,  $I$ , which is valid instance of  $\overline{SAT}$ . Then  $\text{Alg}_{SAT}$  says *Yes* if and only if  $\text{Alg}_{\overline{SAT}}$  says *No*. Such a reduction is not believed to be possible using Karp reductions. The problem  $\overline{SAT}$  is not believed to be in NP because while a satisfying assignment can act as a witness to the fact that a circuit is satisfiable, there are no known “witnesses” to prove that a circuit is not satisfiable.

**Exercise 26.2.4** Suppose problem  $P_1$  is a restricted version of  $P_2$ , in that they are the same except  $P_1$  is defined on a subset  $\mathcal{I}_1 \subseteq \mathcal{I}_2$  of the instances that  $P_2$  is defined on. For example, 3-SAT is a restricted version of SAT because both determine whether a given circuit has a satisfying assignment, however, 3-SAT only considers special types of circuits with clauses of three literals. How hard is it to prove  $P_1 \leq_{\text{poly}} P_2$ ? How hard is it to prove  $P_2 \leq_{\text{poly}} P_1$ ?

- Answer: The first is easy.  $\text{InstanceMap}(I_1)$  simply maps each instance  $I_1 \in S_1 \subseteq S_2$  of  $P_1$  to itself,  $I_1$ , which is valid instance of  $P_2$ . The second is much harder because  $\text{InstanceMap}(I_2)$  must map each instance  $I_2 \in S_2$  of  $P_2$  to some instance  $I_1$  within the restricted set  $S_1$ .

**Exercise 26.2.5** Suppose that when proving  $P_{\text{easier}} \leq_{\text{poly}} P_{\text{harder}}$ , the routines  $\text{InstanceMap}(I_{\text{easier}})$  and  $\text{SolutionMap}(S_{\text{harder}})$  each run in  $\mathcal{O}(|I_{\text{easier}}|^3)$  time and that the mapping  $\text{InstanceMap}(I_{\text{easier}})$  constructs from the instance  $I_{\text{easier}}$  an instance  $I_{\text{harder}}$  that is much bigger, namely,  $|I_{\text{harder}}| = |I_{\text{easier}}|^2$ . Given the following two running times of the algorithm  $\text{Alg}_{\text{harder}}$ , determine the running time of the algorithm  $\text{Alg}_{\text{easier}}$ . (Careful.)

1.  $\text{Time}(\text{Alg}_{\text{harder}}) = \Theta(2^{n^{\frac{1}{3}}})$
2.  $\text{Time}(\text{Alg}_{\text{harder}}) = \Theta(n^c)$  for some constant  $c$ .

- Answer: The running time  $\text{Time}(\text{Alg}_{\text{harder}})$  is measured as a function of its own input size, namely  $\Theta(2^{|I_{\text{harder}}|^{\frac{1}{3}}})$ . But because  $|I_{\text{harder}}| = |I_{\text{easier}}|^2$ , this same time is  $\Theta(2^{|I_{\text{easier}}|^{\frac{2}{3}}})$ , in terms of  $\text{Alg}_{\text{easier}}$ 's input size. The extra  $\mathcal{O}(|I_{\text{easier}}|^3)$  time for the mappings is not substantial. Hence,  $\text{Alg}_{\text{easier}}$ 's total running time is  $\Theta(2^{n^{\frac{2}{3}}})$ . Similarly, if  $\text{Alg}_{\text{harder}}$  runs in polynomial time, namely  $\text{Time}(\text{Alg}_{\text{harder}}) = \Theta(|I_{\text{harder}}|^c)$ , then so does  $\text{Alg}_{\text{easier}}$ , namely  $\text{Time}(\text{Alg}_{\text{easier}}) = \Theta(|I_{\text{easier}}|^{2c} + |I_{\text{easier}}|^3)$ , but note it is a different polynomial.

## 26.3 3-Colouring is NP-Complete

We will now repeat the above steps again in order to prove that 3-Colouring is NP-complete.

- 0) **In NP:** The problem 3-COL is in NP because given an instance graph  $I_{\text{graph}}$  and a solution colouring  $S_{\text{colouring}}$ , it is easy to have an algorithm  $\text{Valid}(I_{\text{graph}}, S_{\text{colouring}})$  check that each node is coloured with one of three colours and that the nodes of each edge have different colours.
- 1) **What To Reduce It To:** We will reduce 3-COL to SAT by proving  $\text{SAT} \leq_{\text{poly}} 3\text{-COL}$ . Above we proved (Any Optimization Problem)  $\leq_{\text{poly}}$  SAT and that  $3\text{-COL} \leq_{\text{poly}}$  Course Scheduling. Together, these give us that SAT, 3-COL, and Course Scheduling are each NP-Complete problems.
- 2) **What is What:**

$P_{\text{easier}}$  is the Satisfiability problem (SAT).

$I_{\text{circuit}}$ , an instance to it, is a circuit.

$S_{\text{assignment}}$ , a potential solution, is an assignment to the circuit variables  $x_1, x_2, \dots, x_n$ .

$P_{harder}$  is the Graph 3-Colouring problem (3-COL).

$I_{graph}$ , an instance to it, is a graph.

$S_{colouring}$ , a potential solution, is an colouring of the nodes of the graph with 3 colours.

- 3) **Direction of Reduction and Code:** To prove 3-COL is “at least as hard”, we must prove that SAT is “at least as easy”, namely  $SAT \leq_{poly} 3\text{-COL}$ . To do this, we must design an algorithm for SAT given an algorithm for 3-COL. The code will be identical to that above.
- 4) **Look For Similarities:** An assignment allocates *True/False* values to each variable, which in turn induces *True/False* values to the output of each gate. A colouring allocates one of three colours to each node. This similarity hints at mapping the variables and outputs of each gate to nodes in the graph and mapping *True* to one colour and *False* to another. With these ideas in mind, Steven Rudich had a computer search for the smallest graph that behaves like an *or* gate when coloured with three colours. The graph found is the top left in Figure 26.1. He calls it an *OR Gadget*.

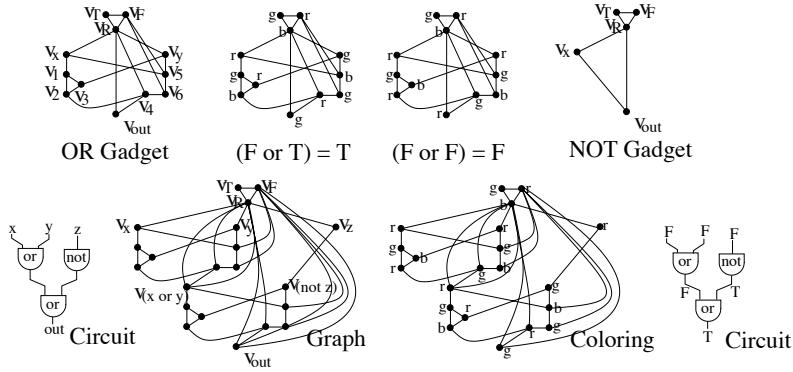


Figure 26.1: On the top, the first figure is the *or* gadget. The next two are colourings of this gadget demonstrating  $(False \text{ or } True) = True$  and  $(False \text{ or } False) = False$ . Note that within these colourings, the colour green indicates true and red indicates false. The top right figure is the *not* gadget. On the bottom, the first figure is the circuit given as an instance to SAT. The next is graph that it is translated into. The next is a 3-colouring of this graph. The last is the assignment for the circuit obtained from the colouring.

**Translating between Colours and True/False:** The three nodes  $v_T$ ,  $v_F$ , and  $v_R$  in the *or* gadget are referred to as the *pallet*. Because of the edges between them, when the gadget is properly coloured, these nodes need to be assigned different colours. Whatever colour is assigned to the node  $v_T$ , we will say that it is the colour indicating *true*, that colouring  $v_F$  the colour indicating *false*, and that colouring  $v_R$  the *remaining* colour. For example, in all the colourings in Figure 26.1, the colour green indicates *true*, red indicates *false*, and blue is the remaining colour.

**Input and Output Values:** Nodes  $v_x$  and  $v_y$  in the *or* gadget act as the gadget’s inputs and the node  $v_{out}$  as its output. Because each of these nodes has an edge to node  $v_R$ , they cannot be coloured with the *remaining* colour. The node will be said to “have the value *true*”, if it is assigned the same colour as  $v_T$  and *false* if the same as  $v_F$ . The colouring in the second figure in Figure 26.1 sets  $x = \text{false}$ ,  $y = \text{true}$ , and the output

$= \text{true}$ . The colouring in the third figure sets  $x = \text{false}$ ,  $y = \text{false}$ , and the output  $= \text{false}$ .

**Theorem:** Rudish's *or* gadget acts like an *or* gate, in that it always can be and always must be coloured so that the value of its output node  $v_{out}$  is the *or* of the values of its two input nodes  $v_x$  and  $v_y$ .

**Proof:** There are four input instances to the gate to consider.

**(False or True) = True:** If node  $v_x$  is coloured false and  $v_y$  is coloured true, then because  $v_5$  has an edge to each, it must be coloured the remaining colour.  $v_6$ , with edges to  $v_F$  and  $v_5$  must be coloured true.  $v_4$ , with edges to  $v_R$  and  $v_6$  must be coloured false.  $v_{out}$ , with edges to  $v_R$  and  $v_4$  must be coloured true. The colouring in the second figure in Figure 26.1 proves that such a colouring is possible.

**(False or False) = False:** If node  $v_x$  and  $v_y$  are both coloured false, then neither nodes  $v_1$  nor  $v_3$  can be coloured false. Because of the edge between them, one of them must be true and the other the remaining colour. Because  $v_2$  has an edge to each of them, it must be coloured false.  $v_4$ , with edges to  $v_R$  and  $v_2$  must be coloured true.  $v_{out}$ , with edges to  $v_R$  and  $v_4$  must be coloured false. The colouring in the third figure in Figure 26.1 proves that such a colouring is possible.

**Exercise 26.3.1** Complete the proof of the theorem by proving the cases  $(\text{True or True}) = \text{True}$  and  $(\text{True or False}) = \text{True}$ .

**Exercise 26.3.2** Prove a similar theorem for the *not* gadget. See the top right figure in Figure 26.1.

**5) InstanceMap, Translating the Circuit into a Graph:** Our algorithm for the SAT takes as input a circuit  $I_{\text{circuit}}$  to be satisfied and in order to receive help from the 3-COL algorithm constructs from it a graph  $I_{\text{graph}} = \text{InstanceMap}(I_{\text{circuit}})$  to be coloured. See the first two figure on the bottom of Figure 26.1. The graph will have one pallet of nodes  $v_T$ ,  $v_F$ , and  $v_R$  with which to define the true and the false colour. For each variable  $x_i$  of the circuit, it will have one node labeled  $x_i$ . It will also have one node labeled  $x_{out}$ . For each *or* gate and *not* gate in the circuit, the graph will have one copy of the *or gadget* or the *not gadget*. The *and* gates could be translated into a similar *and* gadget or translated to  $[x \text{ and } y] = [\text{not}(\text{not}(x) \text{ or } \text{not}(y))]$ . All of these gadgets share the same three pallet nodes. If in the circuit the output of one gate is the input of another then the corresponding nodes in the graph are the same. Finally, one extra edge is added to the graph from the  $v_F$  node to the  $v_{out}$  node.

**6) SolutionMap, Translating a Colouring into an Assignment:** When the supposed algorithm finds a colouring  $S_{\text{colouring}}$  for the graph  $I_{\text{graph}} = \text{InstanceMap}(I_{\text{circuit}})$ , our algorithm must translate this colouring into an assignment  $S_{\text{assignment}} = \text{SolutionMap}(S_{\text{colouring}})$  of the variables  $x_1, x_2, \dots, x_n$  for circuit. See the last two figure on the bottom of Figure 26.1. The translation is accomplished by setting  $x_i$  to true if node  $v_{x_i}$  is coloured the same colour as node  $v_T$  and false if the same as  $v_F$ . If node  $v_{x_i}$  has the same colour node  $v_R$ , then this is not a valid colouring because there is an edge in the graph from node  $v_{x_i}$  to node  $v_R$  and hence need not be considered.

**Warning:** Suppose that the graph constructed had a separate node for each time that the circuit used the variable  $x_i$ . The statement “set  $x_i$  to true when the node  $v_{x_i}$  has some

colour” would then be ambiguous, because the different nodes representing  $x_i$  may be given different colours. Similar mistakes are often made.

- 7) Valid to Valid:** Here we must prove that if the supposed algorithm gives us a valid colouring  $S_{\text{colouring}}$  for the graph  $I_{\text{graph}} = \text{InstanceMap}(I_{\text{circuit}})$ , then  $S_{\text{assignment}} = \text{SolutionMap}(S_{\text{colouring}})$  is an assignment that satisfies the circuit. By the gadget theorem, each gadget in the graph must be coloured in a way that acts like the corresponding gate. Hence, when we apply the assignment to the circuit, the output of each gate will have the value corresponding to the colour of corresponding node. It is as if the colouring of the graph is performing the computation of the circuit. It follows that the output of the circuit will have the value corresponding to the colour of node  $v_{\text{out}}$ . Because node  $v_{\text{out}}$  has an edge to  $v_R$  and an extra edge to  $v_F$ ,  $v_{\text{out}}$  must be coloured true. Hence, the assignment is one for which the output of the circuit is true.

**Exercise 26.3.3** Verify that each edge in the graph  $I_{\text{graph}} = \text{InstanceMap}(I_{\text{circuit}})$  is needed by showing that if it was not there then it would be possible for the graph to have a valid colouring even when the circuit is not satisfied.

- 8) ReverseSolutionMap:** For the proof we must also define the reverse mapping from each assignment  $S_{\text{assignment}}$  to a colouring  $S_{\text{colouring}} = \text{ReverseSolutionMap}(S_{\text{assignment}})$ . Start by colouring the pallet nodes true, false, and the remaining colour. Colour each node  $v_{x_i}$  true or false according to the assignment. Then the theorem states that no matter how the inputs nodes to a gadget is coloured, the entire gadget can be coloured with the output node having the colour as indicated by the output of the corresponding gate.

- 9) Reverse Valid to Valid:** Now we prove that if the assignment  $S_{\text{assignment}}$  satisfies the circuit, then the colouring  $S_{\text{colouring}} = \text{ReverseSolutionMap}(S_{\text{assignment}})$  is valid. The theorem ensured that each edge in each gadget has two different colours. The only edge remaining to consider is the extra edge. As the colours percolate down the graph node  $v_{\text{out}}$  must have colour corresponding to the output of the circuit, which must be the true colour because the assignment is satisfies the circuit. This ensures that even the extra edge from node  $v_F$  to  $v_{\text{out}}$  is coloured with two different colours.

- 10 & 11:** These steps are always the same.  $\text{InstanceMap}(I_{\text{circuit}})$  maps Yes circuit instances to Yes 3-COL instances and No to No. Hence, if the supposed algorithm 3-COL works correctly in polynomial time, then our designed algorithm correctly solves SAT in polynomial time. It follows that  $\text{SAT} \leq_{\text{poly}} \text{3-COL}$ . In conclusion, 3-Colouring is NP-complete.

**Exercise 26.3.4** Prove that Independent Set is NP-complete by proving that  $\text{Time(3-COL)} \leq \text{Time(Independent Set)} + n^{\Theta(1)}$ . Hint: A 3-coloring for the graph  $G_{\text{COL}}$  can be thought of as a subset of the pairs  $\langle u, c \rangle$  where  $u$  is a node of  $G_{\text{COL}}$  and  $c$  is a color. An independent set of the graph  $G_{\text{Ind}}$  selects a subset of its nodes. Hence, a way to construct the graph  $G_{\text{Ind}} = \text{InstanceMap}(G_{\text{COL}})$  would be to having a node for each pair  $\langle u, c \rangle$ . Be careful when defining the edges for the graph  $G_{\text{Ind}} = \text{InstanceMap}(G_{\text{COL}})$  so that each valid independent set of size  $n$  in the constructed graph corresponds to a valid three coloring of the original graph. If the constructed graph has unexpected independent sets, you may need to add more edges to the graph.

## 26.4 Integer Factorization and Cryptography are Difficult

In this section, we justify our belief that decoding cryptographic messages is a hard computational problem by reducing it to factoring, i.e., Factoring  $\leq_{poly}$  Decoding.

### Primes and Factoring:

**Factoring is Difficult:** Given an integer  $x$ , the factoring problem must find factors  $a, b \neq 1$  such that  $x = a \times b$  or even better decomposes it into its prime factors, eg.  $1176 = 2^3 \times 3^1 \times 5^0 \times 7^2$ . The obvious algorithm attempts to divide  $2, 3, 4, 5, 6, \dots$  into  $x$ . See Section 4.2 for an explanation why this algorithm takes  $2^{\Theta(n)}$  time when  $x$  is an  $n$  digit number. The best known algorithms take time  $???$ . It is strongly believed that one cannot factor significantly quicker.

**Prime or Composite:** An integer  $x$  is said to be composite if it has factors other than one and itself. Otherwise, it is prime. For example,  $6 = 2 \times 3$  is composite and  $2, 3, 5, 7, 11, 13, 17, \dots$  are prime. Section 27.2 sketches a randomized algorithm that is able to determine whether an  $n$  digit number is prime or not in time  $\Theta(n)$ . A major break through in 2002 by  $???$  was to find a deterministic algorithm solving the problem running in the same time.

**Finding a Big Prime Number:** Suppose you want to find a random  $n$  digit prime. A fast method is to randomly choose any  $n$  digit number  $x$  and then test whether  $x, x+1, x+2, \dots, x+\mathcal{O}(n)$  is prime stopping when the first is found. The distribution of primes is such that the difference between consecutive  $n$  digit primes is approximately  $\Theta(n)$ .

**Search Hard, Decision Easy:** For most optimization problems, determining whether or not an instance has a solution is no easier than finding a solution. We have just seen that the problem of factoring is a major exception to this, because Factoring is believed to be difficult, but knowing whether an integer has factors is easy. It is this rare combination that is the basis for cryptography.

**Cryptography:** Cryptography is an extremely useful tool in this information age for sending secure messages and for verifying ones true identity. The basis of one of the main strategies uses that fact that large primes are easy to find, but factoring is hard.

**Message Passing:** If you want to receive messages then you can randomly choose two large primes  $p_1$  and  $p_2$  as described above. You multiply them together giving you  $x = p_1 \times p_2$ . You publish this product to the world. Anyone wanting to send you a message can encode this message using your product  $x$ . You, knowing  $p_1$  and  $p_2$  can decode the message. Anyone else, knowing  $x$  but not knowing  $p_1$  and  $p_2$ , cannot decode your message without first factoring  $x$ . Factoring is believed to be hard. Hence, decrypting someone else's message is hard.

**Identity Verification:** You can verify to anyone that you are you by having them encode a message for you and you correctly decoding it.

**Encryption Keys:** When you are doing on-line banking or the like and it speaks of 100 digit encryption keys. What they are talking about are 100 digit products of primes  $x = p_1 \times p_2$ .

**Breaking Codes:** The military has huge computers cranking all day long trying to factor products of primes so that they can decode messages sent by their “enemy.” One of the reasons that the German’s lost the second world war is because mathematicians were able to break their code.

## 26.5 An Algorithm for Bipartite Matching using the Network Flow Algorithm

Above we were justifying our belief that certain computational problems are difficult by reducing them to other problems believed to be difficult. Here, we will give an example of the reverse, by proving that the problem *Bipartite Matching* can be solved easily by reducing it to the Network Flows problem, which we already know is easy because we gave an polynomial time algorithm for it in Section 21.

**Bipartite Matching:** Bipartite matching is a classic optimization problem useful for deciding who should marry who. As always, we define the problem by given a set of instances, a set of solutions for each instance, and a cost for each solution.

**Instances:** An input instance to the problem is a bipartite graph. A bipartite graph is a graph whose nodes are partitioned into two sets  $U$  and  $V$  and all edges in the graph go between  $U$  and  $V$ . See the first figure in Figure 26.2. For example,  $U$  might represent the set of boys,  $V$  the set of girls, and the graph *Loves* puts an edge between boy  $u \in U$  and girl  $v \in V$  if  $u$  and  $v$  love each other.

**Solutions for Instance:** Given an instance, a solution is a matching. A matching is a subset  $M$  of the edges so that no node appears more than once in  $M$ . See the last figure in Figure 26.2. This might indicate which boy should marry which girl. The matching requirement is that nobody marries more than one person. (Sorry for the restriction to monogamous heterosexual marriages.)

**Cost of a Solution:** The cost (or success) of a matching is the number of pairs matched. It is said to be a perfect matching if every node is matched.

**Goal:** Given a bipartite graph, the goal of the problem is to find a matching that matches as many pairs as possible.

**Network Flows:** Network Flow is another example of an optimization problem which involves searching for a best solution from some large set of solutions. See Section 19 for a formal definition.

**Instances:** An instance  $\langle G, s, t \rangle$  consists of a directed graph  $G$  and specific nodes  $s$  and  $t$ . Each edge  $\langle u, v \rangle$  is associated with a positive capacity  $c_{\langle u, v \rangle}$ .

**Solutions for Instance:** A solution for the instance is a *flow*  $F$  which specifies a flow  $F_{\langle u, v \rangle} \leq c_{\langle u, v \rangle}$  through each edges of the network with no leaking or additional flow at any node.

**Cost of Solution:** The cost (or success) of a flow is the the amount of flow out of node  $s$ .

**Goal:** Given an instance  $\langle G, s, t \rangle$ , the goal is to find an optimal solution, i.e., a maximum flow.

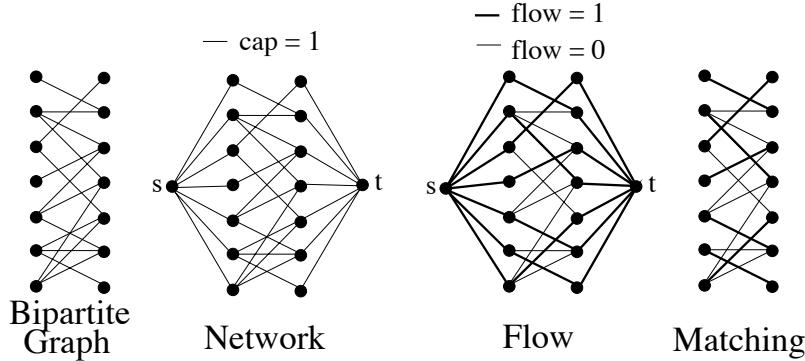


Figure 26.2: The first figure is the bipartite graph given as an instance to Bipartite matching. The next is the network that it is translated into. The next is a flow through this network. The last is the matching obtained from the flow.

**Bipartite Matching  $\leq_{poly}$  Network Flows:** We go through the same steps as before.

- 3) **Direction of Reduction and Code:** We will now design an algorithm for Bipartite Matching given an algorithm for Network Flows.
- 4) **Look For Similarities:** A matching decides which edges to keep and a flow decides which edges to put flow through. This similarity suggests keeping the edges that have flow through them.
- 5) **InstanceMap, Translating the Bipartite Graphs into a Network:** Our algorithm for Bipartite Matching takes as input a bipartite graph  $G_{bipartite}$ . The first step is to translate this into a network  $G_{network} = InstanceMap(G_{bipartite})$  as follows. See the first two figures in Figure 26.2. The network will have the nodes  $U$  and  $V$  from the bipartite graph and for each edge  $\langle u, v \rangle$  in the bipartite graph, the network has a directed edge  $\langle u, v \rangle$ . In addition, the network will have a source node  $s$  with a directed edge from  $s$  to each node  $u \in U$ . It will also have a sink node  $t$  with a directed edge from each node  $v \in V$  to  $t$ . Every edge in the network will have capacity one.
- 6) **SolutionMap, Translating a Flow into an Matching:** When the Network Flows algorithm finds a flow  $S_{flow}$  through the network, our algorithm must translate this flow into a matching  $S_{matching} = SolutionMap(S_{flow})$ . See the last two figures in Figure 26.2.

**SolutionMap:** The translation puts the edge  $\langle u, v \rangle$  in the matching if there is a flow of one through the corresponding edge in the network and not if there is no flow in the edge.

**Mistake:** One must be careful to map *every* possible flow to a matching. The above mapping is ill defined when there is a flow of  $\frac{1}{2}$  through an edge. This needs to be fixed and could be quite problematic.

**Integer Flow:** Luckily, Exercise 21.2.4 proves that if all the capacities in the given network are integers, then the algorithm always returns a solution in which the flow through each edge is an integer. Given that our capacities are all one, each edge will either have a flow of zero or of one. Hence, in our translation, it is well-defined whether to include the edge  $\langle u, v \rangle$  in the matching or not.

**7) Valid to Valid:** Here we must prove that if the flow  $S_{flow}$  is valid than the matching  $S_{matching}$  is also valid.

**Each  $u$  Matched At Most Once:** Consider a node  $u \in U$ . The flow into  $u$  can be at most one because there is only one edge into it and it has capacity one. For the flow to be valid, the flow out of this node must equal that in. Hence, it too can be at most one. Because each edge out of  $u$  either has flow zero or one, it follows that at most one edge out of  $u$  has flow. We can conclude that  $u$  is matched to at most one node  $v \in V$ .

**Each  $v$  Matched At Most Once:**

**Exercise 26.5.1** *Give a similar proof for this case.*

**Cost to Cost:** To be sure that the matching we obtain contains the maximum number of edges, it is important that the cost of the matching  $S_{matching} = SolutionMap(S_{flow})$  equals the cost of the flow. The cost of the flow is the amount of flow out of node  $s$ , which equals the flow across the cut  $\langle U, V \rangle$ , which equals the number of edges  $\langle u, v \rangle$  with flow of one, which equals the number of edges in the matching, which equals the cost of the matching.

**8) ReverseSolutionMap:** The reverse mapping from each matching  $S_{matching}$  to a valid flow  $S_{flow} = ReverseSolutionMap(S_{matching})$  is straight forward. If edge  $\langle u, v \rangle$  is in the matching, then put a flow of one from the source  $s$ , along the edge  $\langle s, u \rangle$  to node  $u$ , across the corresponding edge  $\langle u, v \rangle$ , and then on through the edge  $\langle v, t \rangle$  to  $t$ .

**9) Reverse Valid to Valid:** We must also prove that if the matching  $S_{matching}$  is valid then the flow  $S_{flow} = ReverseSolutionMap(S_{matching})$  is also valid.

**Flow in Equals Flow Out:** Because the flow is the sum of paths, we can be assured that the flow in equals the flow out of every node except for the source and the sink. Because the matching is valid, each  $u$  and each  $v$  is matched either zero or once. Hence the flows through the edges  $\langle s, u \rangle$ ,  $\langle u, v \rangle$ , and  $\langle v, t \rangle$  will be at most their capacity one.

**Cost to Cost:** Again, we need to prove that the cost of the flow  $S_{flow} = ReverseSolutionMap(S_{matching})$  is the same as the cost of the matching. This will be left as an exercise.

**10 & 11:** These steps are always the same.  $InstanceMap(G_{bipartite})$  maps bipartite graph instances to network flow instances  $G_{flow}$  with the same cost. Hence, because algorithm  $Alg_{flow}$  correctly solves network flows quickly, our designed algorithm correctly solves bipartite matching quickly.

In conclusion, bipartite matching can be solved in the same time that network flows is solved.

## Chapter 27

# Randomized Algorithms

For some computational problems, allowing the algorithm to flip coins makes for a simpler, faster, easier to analyze algorithm. The following are the three main reasons.

**Hiding the Worst Cases from the Adversary:** The “running time” of a randomized algorithms is analyzed in a different way than that of a deterministic algorithm. At times, this way is more fair and more in line with how the algorithm actually performs in practice. Suppose, for example, that a deterministic algorithm quickly gives the correct answer on most input instances, yet is very slow or gives the wrong answer on a few instances. Its running time and its correctness is generally measured to be that on these worst case instances. A randomized algorithm might also sometimes be very slow or gives the wrong answer. However, we accept this, as long as on every input instance, the probability of doing so (over the choice of random coins) is small.

**Probabilistic Tools:** The field of probabilistic analysis has many useful techniques and lemmas that can make the analysis of the algorithm simple and elegant.

**Solution has a Random Structure:** When the solution that we are attempting to construct has a random structure, a good way to construct it is to simply flip coins to decide how to built each part. Sometimes we are then able to prove that with high probability the solution obtained this way has better properties than any solution we know how to construct deterministically. More over, if we can prove that the solution constructed randomly has extremely good properties with some very small but non-zero probability, eg.  $prob = 10^{-100}$ , then this proves the existence of such a solution even though we have no reasonably quick way of finding one. Another interesting situation is when the randomly constructed solution very likely has the desired properties, eg. with probability 0.999999, however, there is no quick way of testing whether what we have produced has the desired properties.

This chapter considers these ideas further.

### 27.1 Introduction to Probability Theory

Probability theory is a means of measuring the likelihood of different *events* occurring when conducting some well-defined *experiment*.

**Experiments:** An experiment might be as simple as flipping a coin and observing whether the event heads or the event tails occurs. It might consist of buying a lottery ticket and observing

how much money is gained or lost. It might also consist of executing a randomized algorithm on a given input instance and observing how long it executes and whether it gives the correct output.

**Probability of an Event:** The probability of event  $A$  is a real number  $p = \Pr[A] \in [0, 1]$  that measures the fraction of times that the event occurs.

**Definition:** You can determine this probability either by repeating the experiment or by looking into how the experiment works. Either way, it involves counting.

**Running Many Times:** If you repeat the experiment “independently” some very large number of times  $N$ , then  $p$  is defined to be the fraction of those times in which it likely occurs, i.e.,  $pN$  times out of the  $N$  trials.

$$p = \Pr[A] = \lim_{N \rightarrow \infty} \frac{\text{The \# of times event } A \text{ occurs in } N \text{ trials}}{N}$$

**Inner Workings:** One can also look into the workings of the experiment to determine the probability of an event.

**Underlying Coin Flips:** Suppose, for example, that the randomness for the experiment comes from flipping a fair coin a fixed number of times. We will use  $r$  to denote the outcomes of the coin flips. For example,  $r$  might be  $\langle \text{heads}, \text{tails}, \text{heads}, \text{heads}, \dots, \text{tails} \rangle$ . Each is equally likely to occur. Knowing this outcome  $r$  completely determines what occurs during the experiment. The probability  $p$  of event  $A$  can then be defined to be

$$p = \Pr[A] = \frac{\text{The \# of } r \text{ for which event } A \text{ occurs}}{\text{The \# of } r}$$

**Random Real in  $[0, 1]$ :** Computers can not actually flip coins. Instead, your program can call a system routine which tries to return some thing that is *pseudo random*. A common routine *rand* returns a random real value  $x$  between zero and one. You can use this to simulate other random distributions. For example, one can simulate a 6-sided dice as follows. If  $x \in [0, \frac{1}{6})$  pretend that you rolled a one. If  $x \in [\frac{1}{6}, \frac{2}{6})$ , pretend you rolled a two and so on. This works because for any  $0 \leq a \leq b \leq 1$ ,  $\Pr[x \in [a, b]] = b - a$ .

### Examples:

**Coin Flip:** Given this definition, it is easy to see that the probability of the event *heads* when flipping a coin is  $p = \frac{1}{2}$ .

**Lotteries:** The probability of your ticket winning when there are 10,000,000 tickets is  $p = \frac{1}{10,000,000}$ .

**Dieing:** To help get perspective on the probability  $p = \frac{1}{10,000,000}$ , it is approximately the probability of dying in the next five minutes, because people generally live at most 90 years which is  $90 \cdot 365 \cdot 24 \cdot 60/5 \approx 10,000,000$  blocks of 5 minutes and we approximate that you die in a random one of these.

**Ven Diagrams:** A useful way to visualize probabilities is with *Ven Diagrams*. Draw a square with area one. Let each point in it represent one outcome  $r = \langle \text{heads}, \text{tails}, \text{heads}, \text{heads}, \dots, \text{tails} \rangle$  of the coin flips. For each event, circle those outcomes that lead to the event occurring. The area of the circled region is the probability

of the event. For example, Figure 27.1.1 represents the fact that event  $A$  occurs with probability  $p = \frac{1}{3}$  and fails to occur with probability  $1-p = \frac{2}{3}$ .

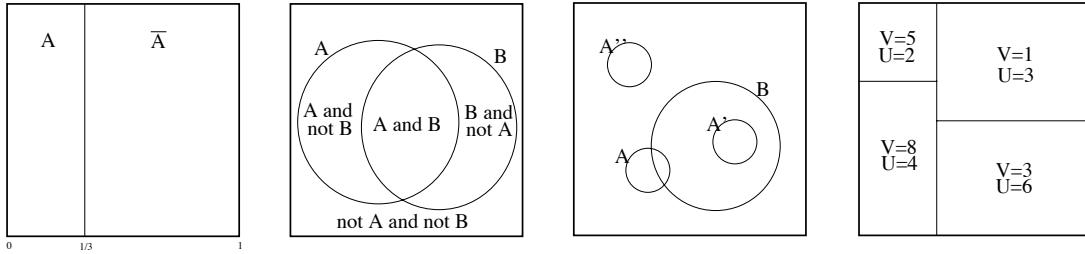


Figure 27.1: The four ven diagrams. The first shows that the probability of event  $A$  is  $p = \frac{1}{3}$ . The second shows the probability of events  $A$  and  $B$  happening simultaneously. The third demonstrates events  $A$  and  $B$  being independent, positively dependent, or negatively dependent. The last shows a random variable  $V$  with  $\Pr[V = 5] = \frac{1}{9}$ .

**Dependencies Between Events:** When you have more than one event, the dependencies between them can be complicated.

**Ven Diagrams:** Ven diagrams are useful for visualizing these dependencies. Figure 27.1.2 represents the event when both event  $A$  and event  $B$  both occur simultaneously, when only one or the other occurs, and when neither occurs.

**Probability of  $A$  given  $B$ :** The probability of an event  $A$  is also related to the extent of our knowledge about whether the event will occur. If all coin flip outcomes  $r$  are equally likely, then  $\Pr[A]$  tells us the likelihood of event  $A$  happening. But suppose now, that we knew that event  $B$  happened. This narrows the possible coin flip outcomes  $r$  to only those for which  $B$  occurs. See the  $B$  circle in Figure 27.1.2. If only these are equally likely, then the fraction of times that  $A$  will happen is

$$\Pr[A|B] = \frac{\text{The } \# \text{ of } r \text{ for which both } A \text{ and } B \text{ occur}}{\text{The } \# \text{ of } r \text{ for which event } B \text{ occurs}} = \frac{\Pr[A \text{ and } B]}{\Pr[B]}$$

**Independence and Dependence:** Events can be dependent in different ways. Figure 27.1.3 gives examples.

**Independent Events:** Events  $A$  and  $B$  are said to be *independent* if knowing that  $B$  occurs, does not give you any information about whether  $A$  occurs. For example, when flipping two coins, their outcomes are independent. The formal definition is

$$\Pr[A|B] = \Pr[A].$$

An equivalent definition is that events  $A$  and  $B$  are independent if and only if

$$\Pr[A \text{ and } B] = \Pr[A] \cdot \Pr[B].$$

**Exercise 27.1.1** (*See solution in Section V*) Prove that these two definitions are equivalent.

**Exercise 27.1.2** (*See solution in Section V*) Compute  $\Pr[A \text{ and not } B]$  and  $\Pr[\text{not } A \text{ and not } B]$  in terms of  $\Pr[A]$  and  $\Pr[B]$ .

**Positively Dependent:** Events  $A'$  and  $B$  are said to be *positively dependent* if they are more likely to occur together, i.e.  $\Pr[A'|B] > \Pr[A']$  and  $\Pr[A' \text{ and } B] > \Pr[A'] \cdot \Pr[B]$ . (Note that in Figure 27.1.3,  $\Pr[A'|B] = 1$ .) This may occur because  $B$  “causes”  $A$  to happen, because  $A$  “causes”  $B$  to happen, or because some event  $C$  “causes” both  $A$  and  $B$  to happen. A butterfly flapping its wings in Africa and a storm in Toronto are likely independent events, but they say that in this interconnected chaotic world, these events may be dependent.

**Negatively Dependent:** Events  $A''$  and  $B$  are said to be *negatively dependent* if they are less likely to occur together, i.e.  $\Pr[A''|B] < \Pr[A'']$  and  $\Pr[A'' \text{ and } B] > \Pr[A''] \cdot \Pr[B]$ . (Note that in Figure 27.1.3,  $\Pr[A''|B] = 0$ .)

**Random Variables:** Some experiments result in a value, like your winnings at gambling or the running time of a randomized algorithm. The resulting value  $V$  is referred to as a *random variable*, as it takes on different values with different probabilities.

### Examples:

**Ven Diagram:** In Figure 27.1.4,  $\Pr[V = 5] = \frac{1}{9}$  and  $\Pr[V = 1] = \frac{1}{3}$ .

**Running Time:** The running time  $T$  of a randomized algorithm is a random variable.

**Number of Heads:** If you flip a coin  $n$  times, the number of times that you get a head is a random variable.

**Indicator Variables:** An *indicator variable*  $I_A$  is a random variable which is 1 when the event  $A$  being indicated occurs and zero when it does not.

**Expected Value:** The *expected value* of a random variable is not the value that you expect, but is average value if you were to repeat it many times.

**Definition:** The following are three equivalent definitions.

**Average:** Suppose again that the randomness from flipping a fair coin a fixed number of times and let  $V_r$  denote the value of  $V$  when that the outcomes of the coin flips is  $r$ . Each  $r$  is equally likely to occur. The expected value of  $V$  is its average value.

$$\text{Exp}_r[V] = \frac{\sum_r V_r}{\text{The \# of different } r}$$

**Value:** A more standard definition considers separately each value  $v$  that  $V$  might take on.

$$\text{Exp}_r[V] = \sum_{\text{values } v} \Pr[V = v] \cdot v$$

**Disjoint Events:** Sometimes it is easier to partition all possible outcomes into a set of events.

$$\text{Exp}_r[V] = \sum_{\text{disjoint events } A} \Pr[A] \cdot [\text{value of } V \text{ during event } A]$$

**Exercise 27.1.3** (*See solution in Section V*) Prove that these three definitions of  $\text{Exp}[V]$  are equivalent.

### Examples:

**Coin Flip:** If you get  $V = 1$  for a head and  $V = -1$  for a tail, then the expected amount is  $\text{Exp}[V] = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot (-1) = 0$ .

**Ven Diagram:** In Figure 27.1.4, the expected value of  $V$  is  $\text{Exp}[V] = \sum_v \Pr[V = v] \cdot v = \frac{1}{9} \cdot 5 + \frac{1}{3} \cdot 1 + \frac{2}{9} \cdot 8 + \frac{1}{3} \cdot 3 = 3\frac{2}{3}$ .

**Lotteries:** If you pay \$5 for a lottery ticket and with probability  $p = \frac{1}{10,000,000}$  you win \$25,000,000, then your expected winnings is  $(1 - \frac{1}{10,000,000}) \cdot 0 + \frac{1}{10,000,000} \cdot 25,000,000 = \$2.50$ . But you paid \$5. Hence, you expect to lose half your money. This is a little misleading, because I expect you will lose all of your money.

**Expected Happiness:** Money is not everything though. What is your expected gain in happiness? I claim having \$5 given your current level of wealth adds more to your happiness than having \$5 when you already have \$25,000,000. This proves that happiness does not increase linearly with money. In fact, I would guess it is more logarithmic because no matter how much you have, if the amount you have doubles, your happiness increases by more or less a fixed amount. So let's guess that buying a roti with your \$5 would bring you one unit of happiness and winning \$25,000,000 would bring you 1,000 units of happiness. You say more? Okay, 100,000,000 units. Then your expected happiness gained by buying a ticket is  $(1 - \frac{1}{10,000,000}) \cdot (-1) + \frac{1}{10,000,000} \cdot 100,000 \approx (-1) + 0.001 \approx -1$ , i.e. you lose.

**Expected Number:** If you flip a coin  $n$  times, the expected number of times that you get a head is  $\frac{n}{2}$ .

**Indicator Variables:** The expected value of an indicator variable  $I_A$  equals the probability of the event  $A$ , i.e.  $\text{Exp}[I_A] = \Pr[A] \cdot 1 + \Pr[\text{not } A] \cdot 0 = \Pr[A]$ .

### Linearity of Expectation:

**Expectation of Sum:** A very useful fact is that the expectation of the sum is equal to the sum of the expectations. Let  $V_1, V_2, V_3, \dots, V_n$  be  $n$  random variables, which may or may not be dependent in complicated ways. If you form a new random variable denoted  $V'$  whose value on every outcome of the coins is the sum of the  $V_i$ , then

$$\text{Exp}[V'] = \text{Exp}\left[\sum_i V_i\right] = \sum_i \text{Exp}[V_i].$$

### Examples:

**Ven Diagram:** In Figure 27.1.4,

$$\text{Exp}[V] = \sum_v \Pr[V = v] \cdot v = \frac{1}{9} \cdot 5 + \frac{1}{3} \cdot 1 + \frac{2}{9} \cdot 8 + \frac{1}{3} \cdot 3 = 3\frac{2}{3}.$$

$$\text{Exp}[U] = \sum_u \Pr[U = u] \cdot u = \frac{1}{9} \cdot 2 + \frac{1}{3} \cdot 3 + \frac{2}{9} \cdot 4 + \frac{1}{3} \cdot 6 = 4\frac{1}{9}.$$

$$\text{Exp}[(V + U)] = \sum_w \Pr[(V + U) = w] \cdot w = \frac{1}{9} \cdot 7 + \frac{1}{3} \cdot 4 + \frac{2}{9} \cdot 12 + \frac{1}{3} \cdot 9 = 7\frac{7}{9}.$$

We can check that  $\text{Exp}[(V + U)] = 7\frac{7}{9} = 3\frac{2}{3} + 4\frac{1}{9} = \text{Exp}[V] + \text{Exp}[U]$ .

**Expected Number:** If you have  $n$  trials where each trial has success with probability  $p$ , the expected number of successes is  $pn$ . This is true even if the success of each trial dependent in complicated ways on each other. The simple proof is as follows.

$$\text{Exp}[\text{Numb of successes}] = \text{Exp}\left[\sum_i I_i\right] = \sum_i \text{Exp}[I_i] = \sum_i [p \cdot 1 + (1 - p) \cdot 0] = pn$$

**Exercise 27.1.4** (See solution in Section V) Prove that the expectation of the sum is equal to the sum of the expectations.

**Expectation of Product:** The same thing is true for the product of random variable if the random variables are independent and is not necessarily true if they are dependent.

**Exercise 27.1.5** (See solution in Section V) Prove that if  $V_1, V_2, V_3, \dots, V_n$  are independent random variables, then  $\text{Exp}[V'] = \text{Exp}[\prod_i V_i] = \prod_i \text{Exp}[V_i]$ .

**Exercise 27.1.6** (See solution in Section V) Prove that if the random variables are dependent than the above is not necessarily true.

**Markov's Tail Inequality:** One simple useful observation is the following. If  $V$  is a random variable that only takes on non-negative values and  $v$  is any fixed value, then

$$\Pr[V \geq v] \leq \frac{\text{Exp}[V]}{v}$$

For example, in Figure 27.1.4,  $0.2222 = \frac{2}{9} = \Pr[V \geq 8] \leq \frac{\text{Exp}[V]}{8} = \frac{3.2/3}{8} = 0.4583$ .

**Exercise 27.1.7** (See solution in Section V) Prove Markov's inequality

**Standard Deviation:**  $\text{Exp}[V]$  gives the expected or the average value of the random variable  $V$ . However, we might also want to know how likely or how much the actual value of  $V$  deviates far from this expectation, namely  $|V - \text{Exp}[V]|$ . We could compute the expected deviation, i.e.  $\text{Exp}[|V - \text{Exp}[V]|]$ , however, the absolute values makes the computations cumbersome. Hence, we compute the expected value of the square of the deviation, namely

$$\text{Variance}[V] = \text{Exp}[(V - \text{Exp}[V])^2] = \sum_v \Pr[V = v] \cdot (v - \text{Exp}[V])^2.$$

Note how the square acts like it is taking the absolute value because both negative and positive values become positive. Another effect of squaring the deviation is that large deviations like  $V - \text{Exp}[V] = 100$  when squared  $100^2 = 10,000$  become even more significant to the expectation. The next thing that we do to take the square root of this expected value. The reason is that if  $V$  takes a value with units meters, then so is  $(V - \text{Exp}[V])$ , but  $(V - \text{Exp}[V])^2$  and  $\text{Exp}[(V - \text{Exp}[V])^2]$  would have units meters squared. By taking the square root of this, the units become meters again. We call this the *standard deviation* of the random variable  $V$ .

$$\text{StandardDeviation}[V] = \sqrt{\text{Exp}[(V - \text{Exp}[V])^2]}$$

**Examples:**

**Balanced:** Suppose that  $V = 2$  with probability  $\frac{1}{2}$  and  $V = 8$  with probability  $\frac{1}{2}$ . Its expected value is  $\text{Exp}[V] = \frac{1}{2} \cdot 2 + \frac{1}{2} \cdot 8 = 5$ , its variance is  $\text{Var}[V] = \sum_v \Pr[V = v] \cdot (v - \text{Exp}[V])^2 = \frac{1}{2} \cdot (2 - 5)^2 + \frac{1}{2} \cdot (8 - 5)^2 = \frac{1}{2} \cdot (-3)^2 + \frac{1}{2} \cdot (3)^2 = 9$ , and its standard deviation is  $SD[V] = \sqrt{\text{Var}[V]} = 3$ . This makes sense because we expect  $V$  to deviate 3 from its expected value 5.

**Ven Diagram:** In Figure 27.1.4, the expected value of  $V$  is  $\text{Exp}[V] = 3\frac{2}{3}$ , its variance is  $\text{Var}[V] = \sum_v \Pr[V = v] \cdot (v - \text{Exp}[V])^2 = \frac{1}{9} \cdot (5 - 3\frac{2}{3})^2 + \frac{1}{3} \cdot (1 - 3\frac{2}{3})^2 + \frac{2}{9} \cdot (8 - 3\frac{2}{3})^2 + \frac{1}{3} \cdot (3 - 3\frac{2}{3})^2 = 6\frac{8}{9}$  and its standard deviation is  $SD[V] = \sqrt{\text{Var}[V]} = 2.624\dots$

**Chebyshev's Tail Inequality:** Another useful observation is the following. If  $V$  is a random variable (taking on positive or negative values) and  $h$  is any fixed value, then

$$\Pr[|V - \text{Exp}[V]| \geq h] \leq \frac{\text{SD}[V]^2}{h^2}$$

For example, in Figure 27.1.4,  $0.2222 = \frac{2}{9} = \Pr[V \geq 8] \leq \Pr[|V - \text{Exp}[V]| \geq 8 - 3\frac{2}{3}] \leq \frac{\text{SD}[V]^2}{h^2} = \frac{(2.624..)^2}{(8-3\frac{2}{3})^2} = 0.3666$ .

**Exercise 27.1.8** (*See solution in Section V*) Prove Chebyshev's inequality

**Chernoff's Tail Inequalities:** If you have  $n$  independent trials where each trial has success with probability  $p \leq \frac{1}{2}$ , you expect to get  $pn$  successes. The probability of deviating far from this is exponentially small. More specifically, the probability of getting fewer than  $pn-h$  successes is at most  $e^{-h^2/(2pn)}$ . For example, the probability of getting a constant factor fewer, i.e.  $h = \epsilon pn$ , is at most  $e^{-\epsilon^2 pn/2} = e^{-\Theta(n)}$ . My favorite way of expressing it is as follows. The standard deviation is  $\sqrt{pn}$ . The probability of getting  $c$  standard deviations too few, i.e.  $h = c\sqrt{pn}$ , is at most  $e^{-c^2/2}$ . This version also demonstrates that you won't likely get exactly  $pn$  successes, but within  $\sqrt{pn}$  or  $2\sqrt{pn}$  of this number.

For example, if you flip a fair coin 20,000 times, the probability of getting fewer than  $pn - 6\sqrt{np} = \frac{1}{2}20,000 - 600 = 9,400$  heads is at most  $e^{-c^2/2} = e^{-6^2/2} \approx 10^{-8}$ . Similarly, if you flip it a large  $n$  number times, then the fraction of heads is very likely at most  $\frac{n/2+6\sqrt{n/2}}{n} \approx \frac{1}{2}$ . If  $p$  is small, then the probability of getting  $h$  too many is slightly bigger, namely  $e^{-h^2/(3pn)}$ .

**Exercise 27.1.9** (*See solution in Section V*) Give the strongest bound on the probability of getting fewer than  $pn-h$  successes that you can.

**Probability of Succeeding at Least Once:** Suppose that that your experiment, say the running of an algorithm, succeeds with at least probability  $p$ . Suppose that you are able repeat the experiment independently  $N$  times and that you only need to at least one of these times to succeed over all. Finally, suppose that you want to succeed overall with probability  $1-\epsilon$  for some small  $\epsilon > 0$ . Then it is sufficient to repeat the experiment  $N = \frac{1}{p} \ln\left(\frac{1}{\epsilon}\right)$  times. The probability that you fail each of these times is at most

$$\Pr[\text{AlwaysFail}] \leq (1-p)^N \leq e^{-pN} = e^{-\ln(\frac{1}{\epsilon})} = \epsilon.$$

For example, if  $p = \frac{1}{n^2}$  and  $\epsilon = 10^{-9}$  (one in a billion), then  $N = \frac{1}{p} \ln\left(\frac{1}{\epsilon}\right) = n^2 \ln(10^9) \leq 21n^2$ . See Exercise 27.1.11.

**Probability of a Bad Event:** Suppose that there is a list of bad things that might happen. Suppose that you can prove that the probability that the  $i^{th}$  one happens is at most  $p_i$ . It follows that

$$\Pr[\text{At least one bad thing happens}] \leq \sum_i p_i$$

**Exercise 27.1.10** (*See solution in Section V*) Prove this.

## Some Useful Approximations:

$1 - p \leq e^{-p}$ : This is useful bound that we have seen already. It is very close to equality when  $p$  is close to zero, i.e.  $1 - 0 = 1 = e^0$ . Here are some other similar inequalities.

- $1 - p + \frac{p^2}{2} \geq e^{-p}$
- $1 + p \leq e^p$  and for  $p \in [0, 1]$ ,  $1 + p + p^2 \geq e^p$  and  $1 + p \geq e^{p - \frac{p^2}{3}}$ .
- $(1 - p)^n = 1 - np + \Theta(p^2)$ .

$n! \approx \left(\frac{n}{e}\right)^n$ : This is a fairly close approximation of  $n!$  which is the number of ways of arranging  $n$  objects. Stirling's approximation, which is even closer, is  $n! = \sqrt{e\pi n} \cdot \left(\frac{n}{e}\right)^n e^w$  where  $\frac{1}{12(n+5)} \leq w \leq \frac{1}{12n}$ .

$\left(\frac{n}{a}\right)^a \leq \binom{n}{a} \leq \left(\frac{en}{a}\right)^a$ : This is a fairly close approximation of  $\binom{n}{a} = \frac{n!}{a!(n-a)!}$  which is the number of subsets of size  $a$  of  $n$  objects. Another approximation, which is even closer is  $\binom{n}{rn} \approx 2^{\text{Entropy}(r) \cdot n}$ , where  $\text{Entropy}(r) = r \log_2 \frac{1}{r} + (1-r) \log_2 \frac{1}{1-r}$ .

**Exercise 27.1.11** (See solution in Section V) Prove these approximations.

## 27.2 Using Randomness to Hide The Worst Cases

The standard way of measuring the running time and correctness of a deterministic algorithm is based on the worst case input instance chosen by some nasty adversary who has studied the algorithm in detail. This is not fair if the algorithm does very well on all but a small number of very strange and unlikely input instances. On the other hand, knowing that the algorithm works well on most instances is not always satisfactory, because for some applications it is just those the hard instances that you want to solve. In such cases, it might be more comforting to use a randomized algorithm that guarantees that on every input instance, the correct answer will be obtained quickly with high probability.

A randomized algorithm is able to flip coins as it proceeds to decide what actions to take next. Equivalently, a randomized algorithm  $A$  can be thought of as a set of deterministic algorithms  $A_1, A_2, A_3, \dots$  where  $A_r$  is what algorithm  $A$  does when the outcome of the coin flips is  $r = \langle \text{heads}, \text{tails}, \text{heads}, \text{heads}, \dots, \text{tails} \rangle$ . Each such deterministic algorithm  $A_r$  will have a small set of worst case input instances on which it either gives the wrong answer or runs too slow. The idea is that these algorithms  $A_1, A_2, A_3, \dots$  have different sets of worst case instances. This randomized algorithm is good if for each input instance, the fraction of the deterministic algorithms  $A_1, A_2, A_3, \dots$  for which it is not a worst case instance is at least  $p$ . Then when one of these  $A_r$  is chosen randomly, it solves this instance quickly with probability at least  $p$ .

I sometimes find it useful to consider the analysis of randomized algorithms as a game between an algorithm designer and an adversarial chooser of the input instance. In the game, it is not always fair for the adversarial input chooser to know the algorithm first, because then it can choose the instance that is worst case for this algorithm. Similarly, it is not always fair for the algorithm designer to know the input instance first or even which instances are likely, because then it can design the algorithm to work well on these. The way we analyze the running time of randomized algorithms compromises between these two. In this game, the algorithm designer without knowing the input instance must first fix what his algorithm will do given the outcome of the coins. Knowing this, but not knowing the outcomes of the coins, the instance chooser chooses the worst case instance. We then flip coins, run the algorithm, and see how well it does.

**Three Models:** The following are formal definitions of three models.

**Deterministic Worst Case:** In a worst case analysis, a deterministic algorithm  $A$  for a computational problem  $P$  must always give the correct answer quickly.

$$\exists A, \forall I, [A(I) = P(I) \text{ and } \text{Time}(A, I) \leq T_{upper}(|I|)]$$

**Los Vegas:** The algorithm is said to be *Los Vegas* if the algorithm is guaranteed to always give the correct answer, but the running time of the algorithm depends on the outcomes of the random coin flips. The goal is to prove that on every input instance, the expected running time is small.

$$\exists A, \forall I, [\forall r, A_r(I) = P(I) \text{ and } \text{Exp}_r[\text{Time}(A_r, I)] \leq T_{upper}(|I|)]$$

**Monte Carlo:** The algorithm is said to be *Monte Carlo* if the algorithm is guaranteed to stop quickly, but it can sometimes, depending on the outcomes of the random coin flips, give the wrong answer. The goal is to prove that on every input, the probability of it giving the wrong answer is small.

$$\exists A, \forall I, [Pr_r[A_r(I) \neq P(I)] \leq p_{fails} \text{ and } \forall r, \text{Time}(A_r, I) \leq T_{upper}(|I|)]$$

The following examples demonstrate these ideas.

**Quick Sort:** Recall the quick sort algorithm from Section 15.1. The algorithm chooses a pivot element and partitions the list of numbers to be sorted into those that are smaller than the pivot and those that are larger than it. Then it recurses on each of these two parts. The running time varies from  $\Theta(n \log n)$  to  $\Theta(n^2)$  depending on the choices of pivots.

**Deterministic Worst Case:** A reasonable choice for the pivot is to always use the element that happens to be located in the middle of array to be sorted. For all practical purposes, this would likely work great. It would work exceptionally well when the list is already sorted. However, there are some strange inputs cooked up for the sole purpose of being nasty to this particular implementation of the algorithm on which the algorithm runs in  $\Theta(n^2)$  time. The adversary will provide such an input giving a worst case time complexity of  $\Theta(n^2)$ .

**Los Vegas:** In practice, what is often done is to choose the pivot element randomly from the input elements. This makes it irrelevant which order the adversary puts the elements in the input instance. The expected computation time is  $\Theta(n \log n)$ .

**The Game Show Problem:** The input  $I$  to the game show problem specifies which of  $N$  doors has prizes behind them. At least half the doors are promised to have prizes. An algorithm  $A$  is able to look behind the doors in any order that it likes, but nothing else. It solves the problem correctly when it finds a prize. The running time is the number of doors opened.

**Deterministic Worst Case:** Any deterministic algorithm fixes the order that it looks behind the doors. Knowing this order, the adversary places no prizes behind the first  $\frac{N}{2}$  doors looked at.

**Los Vegas:** In contrast, a random algorithm will look behind doors in random order. It does not matter where the adversary puts the prizes, the probability that one is not found after  $t$  doors is  $\frac{1}{2^t}$  and the expected time until a prize is found is  $\text{Exp}[T] = \sum_t \Pr[T = t] \cdot t = 2$ .

**Monte Carlo:** If the promise is that either at least half the doors have prizes or none of them do and if the algorithm stops after 10 empty doors and claims that there are no prizes, than this algorithm is always fast, but give the wrong answer with probability  $\frac{1}{2^{10}}$ .

**Randomized Primality Testing:** An integer  $x$  is said to be *composite* if it has factors other than one and itself. Otherwise, it is said to be *prime*. For example,  $6 = 2 \times 3$  is composite and  $2, 3, 5, 7, 11, 13, 17, \dots$  are prime. See Sections 4.2 and 26.4 for explanations of why it takes  $2^{\Theta(n)}$  time to factor an  $n$  bit number. A major break through in 2002 Agrawal et al. was to find a polynomial time deterministic algorithm for determining whether an  $n$  bit number is prime. Here we give an easy randomized algorithm by Rabin-Miller for this problem.

**Fermat's Little Theorem:** Don't worry about the math, but Fermat's Little Theorem says that if  $x$  is prime, then for every  $a \in [1, x - 1]$ , it is the case that  $a^{x-1} \equiv 1 \pmod{x}$ . Moreover, if  $x$  is not prime, than  $a^{x-1} \equiv 1 \pmod{x}$  is true for at most half of the values  $a \in [1, x - 1]$ . We can use this to distinguish primes from composite numbers.

**The Game Show Problem:** Given an integer  $x$ , suppose that you have one door for each  $a \in [1, x - 1]$ . We will say that there is a prize behind this door if  $a^{x-1} \not\equiv 1 \pmod{x}$ . Fermat's Little Theorem says that if  $x$  is prime, than none of the doors have prizes behind them and if it is composite than at least half the doors have prizes. The algorithm attempts to determine which is the case by opening  $t$  randomly chosen doors for some integer  $t$ .

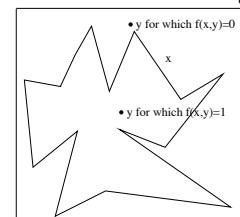
**Exercise 27.2.1** *If the algorithm finds a prize, what do you know about the integer? If it does not find a prize, what do you know?*

**Exercise 27.2.2** *If the algorithm must always give the correct answer, how many doors need to be opened in terms of the number of digits  $n$  in the instance  $x$ .*

**Exercise 27.2.3** *If  $t$  doors are open and the input instance  $x$  is a prime, what is the probability that the algorithm gives the correct answer? If the instance is composite, what is this probability?*

**Randomized Counting:** In many applications, one wants to count the number of occurrences of something. This problem can often be expressed follows. Given the input instance  $x$ , count the number of  $y$  for which  $f(x, y) = 1$ . It is likely very difficult to determine the exact number. However, a good way to approximate this number is to randomly choose some large number of values  $y$ . For each, test whether  $f(x, y) = 1$ . Then the fraction of  $y$  for which  $f(x, y) = 1$  can be approximated by [the number you found]/[the number you tried]. The number of  $y$  for which  $f(x, y) = 1$  can be approximated by [the fraction you found]  $\times$  [the total number of  $y$ ].

For example, suppose you had some strange shape and you wanted to find its area. Then  $x$  would specify the shape,  $y$  would specify some point within a surrounding box, and  $f(x, y) = 1$  if the point is within the shape. Then the number of  $y$  for which  $f(x, y) = 1$  gives you the area of your shape.



**Exercise 27.2.4** Design a randomized algorithm for separating  $n$  VLSI chips into those that are “good” and those that are “bad” by test two chips at a time and learning either that they are the same or that they are different. To help, at least half of the chips are promised to be good. Here are some hints.

- Randomly select one of the chips. What is the probability that the chip is good?
- How can you learn whether or not the selected chip is good?
- If it is good, how can you easily partition the chips into good and bad chips.
- If the chip is not good, what should your algorithm do?
- When should the algorithm stop?
- What is the expected running time of this algorithm?

### 27.3 Solutions of Optimization Problems with a Random Structure

Optimizations problems are looking for the best solution for an instance. Sometimes good solutions have a random structure. In such cases, a good way to construct it is to simply flip coins to decide how to built each part. We give two examples. The first one, *Max Cut*, being NP-complete, likely requires exponential time to find the best solution. However, in  $\mathcal{O}(n)$  time, we can find a solution which is likely to be at least half as good as optimal. The second example, *expander graphs* is even more extreme. Though there are deterministic algorithms for constructing graphs with fairly good expansion properties, a random graph almost for sure has much better expansion properties (with probability  $p \geq 0.999999$ ). A complication, however, is that there is no polynomial time algorithm which tests whether this randomly constructed graph has the desired properties. Pushing the limits further, it can be proved that the same random graph has extremely good properties with some very small but non-zero probability (eg.  $p \geq 10^{-100}$ ). Though we have no quick way to construct such a graph, this does proves that such a graph exists.

**The Max Cut Problem:** The input to the Max Cut problem is an undirected graph. The output is a partition of the nodes into two sets  $U$  and  $V$  so that the number of edges that cross over from one side to the other is as large as possible. This problem is NP-complete and hence, the best known algorithm for finding an optimal solution requires  $2^{\Theta(n)}$  time. The following randomized algorithm runs in time  $\Theta(n)$  and is expected to obtain a solution for which half the edges cross over. Note that the optimal solution cannot have more than all the edges cross over, so a randomized algorithm is expected perform at least half as well as the optimal solution can do. This algorithm is incredibly simple. It simply flips a coin for each node to decide whether to put it into  $U$  or into  $V$ . Each edge will cross over with probability  $\frac{1}{2}$ . Hence, the expected number of edges to cross over is  $\frac{|E|}{2}$ .

**Expander Graphs:** An  $n$  node degree  $d$  graph is said to be an *Expander Graph* if moving from a set of its nodes across its edges expands us out to an even larger set of nodes. More formally, for  $0 < \alpha < 1$  and  $1 < \beta < d$ , a graph  $G = \langle V, E \rangle$  is an  $\langle \alpha, \beta \rangle$ -expander if for every subset  $S \subseteq V$  of its nodes, if  $|S| \leq \alpha n$  then  $N(S) \geq \beta |S|$ . Here  $N(S)$  is the neighborhood of  $S$ , i.e. all nodes with an edge from some node in  $S$ .

**Non-Overlapping Sets of  $d$  Neighbors:** Because each node  $v \in V$  has  $d$  neighbors  $N(v)$ , a set  $S$  has  $d|S|$  edges leaving these nodes. However, if these sets  $N(v)$  of neighbors overlap a lot, then the total number of neighbors  $N(S) = \cup_{v \in S} N(v)$  of  $S$  might be very small. We can't expect  $N(S)$  to be bigger than  $d|S|$  but we do want it to have size at least  $\beta|S|$  where  $1 < \beta < d$ . If  $S$  is too big, we can't expect it to expand further. Hence, we only require this expansion property for sets  $S$  of size at most  $\alpha n$ . Because we do expect sets of size  $\alpha n$  to expand to a neighborhood of size  $\beta \alpha n$ , we do require that  $\alpha\beta < 1$ .

**Connected with Short Paths:** If  $\alpha\beta > \frac{1}{2}$ , then every pair of nodes in  $G$  is connected with a path of length at most  $\frac{2 \log(n/2)}{\log \beta}$ . The proof is as follows. Consider two nodes  $u$  and  $v$ . The node  $u$  has  $d$  neighbors,  $N(u)$ . These neighbors  $N(u)$  must have at least  $\beta|N(u)| = \beta d$  neighbors  $N(N(u))$ . These neighbors  $N(N(u))$  must have at least  $\beta^2 d$  neighbors. It follows that there are at least  $\beta^{i-1} d$  nodes with distance  $i$  from  $u$ . This continues until we have  $\beta \alpha n > \frac{n}{2}$  nodes at distance  $i = \log_\beta \frac{n}{2}$  from  $u$ . This set might not contain  $v$ . However, starting from  $v$  there is another set of more than half the nodes that are distance  $i = \log_\beta \frac{n}{2}$  from  $v$ . These two sets must overlap at some node  $w$ . Hence, there is a path from  $u$  to  $w$  to  $v$  of length at most  $\frac{2 \log(n/2)}{\log \beta}$ .

**Uses:** Expander graphs are very useful both in practice and for proving theorems.

**Fault Tolerant Networks:** As we have seen every pair of nodes in an expander graph are connected. This is still true if a large number of nodes or edges fail. Hence, this is a good pattern for wiring a communications network.

**Pseudo Random Generators:** Taking a short random walk in an expander graph quickly gets you to a random node. This is useful for generating long random looking strings from a short seed string.

**Concentrating and Recycling Random Bits:** If one has a source that has some randomness in it (say  $n$  coin tosses with an unknown probability and with unknown dependencies between the coins), expander graphs can be used to produce a string of  $m$  bits appearing to be the result of  $m$  fair and independent coins.

**Error Correcting Codes:** Expander graphs are also useful in designing ways of encoding a message into a longer code so that if any reasonable fraction of the longer code is corrupted, the original message can still be recovered. The intuition is that the faulty bits are connected by short paths to correct bits.

**If  $\alpha\beta < 1$ , then Expander Graphs Exist:** We will now prove that for any constants  $\alpha$  and  $\beta$  for which  $\alpha\beta < 1$  there is a degree  $d$ ,  $n$  node  $\langle\alpha, \beta\rangle$ -expander graph for some sufficiently big constant  $d$ . For example, if  $\alpha = \frac{1}{2}$ ,  $\beta = \frac{3}{2}$ , then  $d = 5$  is sufficient. To make the analysis easier, we will consider directed graphs where each node  $u$  is connected to  $d$  nodes chosen independently at random. (Note if you ignore the directions of the edges, then each node has average degree  $2d$  and neighborhood sets are only bigger.) We prove that the probability we do not get such an expander graph is strictly less than one. Hence, one must exist.

**Event  $E_{S,T}$ :** The graph  $G$  will not be a  $\langle\alpha, \beta\rangle$ -expander if there is some set  $S$  for which  $|S| \leq \alpha n$  and  $|N(S)| < \beta|S|$ . Hence, for each pair of sets  $S$  and  $T$ , with  $|S| \leq \alpha n$  and  $|T| < \beta|S|$ , let  $E_{S,T}$  denote the bad event that  $N(S) \subseteq T$ . Let us bound the probability of  $E_{S,T}$  when we choose  $G$  randomly. Each node in  $S$  needs  $d$  neighbors for a total of

$d|S|$  randomly chosen neighbors. The probability of a particular one of these landing in  $T$  is  $\frac{|T|}{n}$ . Because these edges are chosen more or less independently, the probability of them all landing in  $T$  is  $\left(\frac{|T|}{n}\right)^{d|S|}$ .

**Probability of Some Bad Event:** The probability that  $G$  is not an expander is the probability that at least one of these bad events  $E_{S,T}$  happens, which is at most the sum of the probabilities of these individual events.

$$\begin{aligned} \Pr[G \text{ not an expander}] &= \Pr[\text{OR}_{S,T} E_{S,T}] \leq \sum_{S,T} \Pr[E_{S,T}] \\ &= \sum_{(s \leq \alpha n)} \sum_{(S \mid |S|=s)} \sum_{(T \mid |T|=\beta s)} \Pr[E_{S,T}] = \sum_{s \leq \alpha n} \binom{n}{s} \binom{n}{\beta s} \left(\frac{|T|}{n}\right)^{d|S|} \end{aligned}$$

We now use the result that  $\binom{n}{a} \leq \left(\frac{en}{a}\right)^a$ .

$$\begin{aligned} \Pr[G \text{ not an expander}] &\leq \sum_{s \leq \alpha n} \left(\frac{en}{s}\right)^s \left(\frac{en}{\beta s}\right)^{\beta s} \left(\frac{\beta s}{n}\right)^{ds} = \sum_{s \leq \alpha n} \left[ \left(\frac{en}{s}\right) \left(\frac{en}{\beta s}\right)^\beta \left(\frac{\beta s}{n}\right)^d \right]^s \\ &\leq \sum_{s \leq \alpha n} \left[ \left(\frac{en}{\alpha n}\right) \left(\frac{en}{\beta \alpha n}\right)^\beta \left(\frac{\beta \alpha n}{n}\right)^d \right]^s = \sum_{s \leq \alpha n} \left[ \frac{e^{\beta+1}}{\alpha} \cdot (\alpha \beta)^{d-\beta} \right]^s \end{aligned}$$

The requirement is that  $\alpha \beta < 1$ . Hence, if  $d$  is sufficiently big,  $(d \geq \log\left(\frac{2e^{\beta+1}}{\alpha}\right) / \log\left(\frac{1}{\alpha \beta}\right) + \beta)$ , then the bracketed amount is at most  $\frac{1}{2}$ .

$$\Pr[G \text{ not an expander}] \leq \sum_{s \leq \alpha n} \left[\frac{1}{2}\right]^s < 1$$

It follows that  $\Pr[G \text{ is an expander}] > 0$ , meaning that there exists at least one such  $G$  which is an expander.

# Chapter 28

## Lower Bounds

This chapter focuses on determining the *time complexity* of a computation problem. This is defined to be the fastest running time of any algorithm solving the problem. To do this, the chapter must first determine what an algorithm is and what its running time is. Various models of each are presented. An *upper bound* on the time complexity of a problem consists of algorithms for it running in the stated time. A *lower bound* proves that NO algorithm exists that runs faster than the stated time. Section 28.1 reviews the definitions of deterministic worst-case time complexity. Section 28.2 gives an information theoretic lower bound for sorting. Section 27 defines randomized algorithms and gives a few upper and lower bounds. Finally, Section 26 uses reductions to both provide more upper bounds and to make strong arguments for the commonly held belief of certain lower bounds are true.

### 28.1 Deterministic Worst-Case Time Complexity

The time complexity of a deterministic algorithm and of a computational problem were defined in Section 4. We will now define this more carefully using the existential and universal quantifiers that were defined in Section 1.

**The Time Complexity of a Problem:** The time complexity of a computational problem  $P$  is the minimum time needed by an algorithm to solve it. See Section 4.

**Upper Bound:** Problem  $P$  is said to be computable in time  $T_{\text{upper}}(n)$  if there is an algorithm  $A$  which outputs the correct answer, namely  $A(I) = P(I)$ , within the bounded time, namely  $\text{Time}(A, I) \leq T_{\text{upper}}(|I|)$ , on every input instance  $I$ . The formal statement is

$$\exists A, \forall I, [A(I) = P(I) \text{ and } \text{Time}(A, I) \leq T_{\text{upper}}(|I|)]$$

$T_{\text{upper}}(n)$  is said to be only an *upper bound* on the complexity of the problem  $P$ , because there may be another algorithm that runs faster. For example,  $P = \text{Sorting}$  is computable in  $T_{\text{upper}}(n) = \mathcal{O}(n^2)$  time. It is also computable in  $T_{\text{upper}}(n) = \mathcal{O}(n \log n)$ .

**Exercise 28.1.1** (*See solution in Section V*) Does the order of the quantifiers matter? Explain for which problems  $P$  and for which time complexities  $T_{\text{upper}}$  the following statement is true:

$$\forall I, \exists A, [A(I) = P(I) \text{ and } \text{Time}(A, I) \leq T_{\text{upper}}(|I|)]$$

**Lower Bound of a Problem:** A lower bound on the time needed to solve a problem states that no matter how smart you are you cannot solve the problem faster than the stated time  $T_{lower}(n)$ , because such algorithm simply does not exist. There may be algorithms that give the correct answer or run sufficiently quickly on some inputs instance. But for every algorithm, there is at least one instance  $I$  for which either the algorithm gives the wrong answer, namely  $A(I) \neq P(I)$ , or it runs in too much time, namely  $Time(A, I) \geq T_{lower}(|I|)$ . The formal statement is the negation (except for  $\geq$  vs  $>$ ) of that for the upper bound.

$$\forall A, \exists I, [A(I) \neq P(I) \text{ or } Time(A, I) \geq T_{lower}(|I|)]$$

For example, it should be clear that no algorithm can sort  $n$  values in only  $T_{lower} = \sqrt{n}$  time, because in that much time the algorithm could not even look at all the values.

**Proofs Using the Prover/Adversary Game:** Recall the technique described in Section 1 for proving statements with existential and universal quantifiers.

**Upper Bound:** We can use the prover/adversary game to prove the upper bound statement  $\exists A, \forall I, [A(I) = P(I) \text{ and } Time(A, I) \leq T_{upper}(|I|)]$  as follows: You, the prover, provide the algorithm  $A$ . Then the adversary provides an input  $I$ . Then you must prove that your  $A$  on input  $I$  gives the correct output in the allotted time. Note this is what we have been doing throughout the book, providing algorithms and proving that they work.

**Lower Bound of a Problem:** A proof of the lower bound  $\forall A, \exists I, [A(I) \neq P(I) \text{ or } Time(A, I) \geq T_{lower}(|I|)]$  consists of a strategy that, when given an algorithm  $A$  by an adversary, you, the prover, study his algorithm and provide an input  $I$ . Then you either prove that his  $A$  on input  $I$  gives the wrong output or runs in more than the allotted time.

**Current State of the Art in Proving Lower Bounds:** Lower bounds are *very* hard to prove because you must consider *every* algorithm, no matter how strange or complex. After all, there are examples of algorithms that start out doing very strange things and then in the end magically produce the required output.

**Information Theoretic:** The technique used here to prove lower bounds does not consider the amount of work that must get done to solve the problem, but the amount of *information* that must be transferred from the input to the output. The problem with these lower bounds is that they are not bigger than linear with respect to the bit size of the input and the output.

**Restricted Model:** A common method of proving lower bounds is to consider only algorithms that have a restricted structure.

**General Model:** The theory community is just now managing to prove the first non-linear lower bounds on a general model of computation. This is quite exciting for those of us in the field.

## 28.2 Information Theoretic Lower Bound for Sorting

We have seen a number of algorithms that can sort  $N$  numbers using  $\mathcal{O}(N \log N)$  comparisons between the values, eg. Merge, Quick, and Heap Sort. In this section, we will prove that no

algorithm can sort with fewer bit operations<sup>1</sup>. Note that Section 12 presents a sorting algorithm Radix/Counting that requires only  $\Theta(n)$  bit operations when  $n$  is the total number of bits in the input instance. At first, this appears to be a contradiction to the lower bound. However, the lower bound states that  $\Omega(N \log N)$  comparisons when  $N$  is the number of numbers in the list. Assuming that the  $N$  numbers to be sorted are distinct, each needs  $\Theta(\log N)$  bits to be represented for a total of  $n = \Theta(N \log N)$  bits. Hence, the lower bound does not in fact say that more than  $\Theta(n)$  bit operations when  $n$  is the total number of bits in the input instance.

**The Yes/No Questions Game:** We start proving lower bounds by generalizing the game of guessing a number between 1 and 100 and proving a lower bound on the number of questions that need to be asked.

**Specification:** Alice, the first player in the game, chooses one object from the fixed set  $\{I_1, I_2, \dots, I_M\}$ . Bob asks yes/no questions about this object until he determines which it is.

Alice chooses a number in  $[1,8]$ .

Bob asks

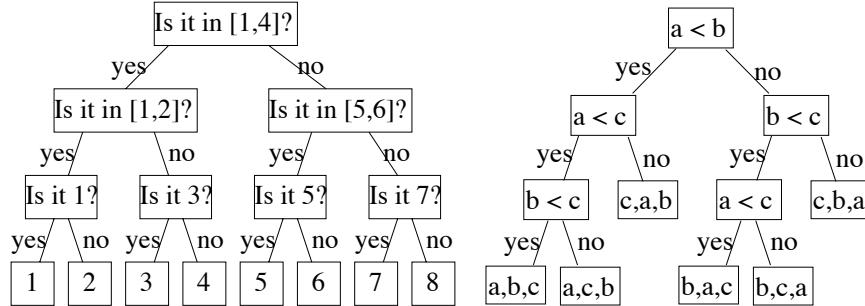


Figure 28.1: A good Yes/No algorithm for Bob to guess a number in  $[1,8]$  would be binary search. This is shown on the left. The right shows a good Yes/No algorithm for Bob to guess one of the  $M = 3!$  permutations of the letters  $a$ ,  $b$ , and  $c$ . It also represents a comparison based decision tree for sorting the three variables  $a$ ,  $b$ ,  $c$ .

**The Model of Computation (Domain of Valid Algorithms):** A yes/no question might be “Is your number less than 50?” or something stranger like “Is the third bit in it zero?” An algorithm for Bob consists of a binary tree. Each node is labeled with a yes/no question, and the two edges coming out of it are labeled with *yes* and *no*. Each leaf is labeled with one object from  $\{I_1, \dots, I_M\}$ . The game proceeds by Bob starting at the root and following a path down to a leaf. At each step, Bob asks the question labeling the node that he is on and follows the edge corresponding to the answer received. When he reaches a leaf, he answers the object given by its label.

**Lower-Bound Statement:** The lower bound is  $T_{lower}(M) = \log_2 M$  questions, where  $M$  is the number of different objects  $\{I_1, \dots, I_M\}$  that we need to distinguish between.

**Proof of Lower Bound:** Suppose the adversary provides an algorithm  $A$  for Bob. We, as the prover, provide an object  $I \in \{I_1, \dots, I_M\}$  such that when Alice chooses this object either Bob gives the wrong object or asks too many questions.

<sup>1</sup>The classic lower bound considers only comparison based algorithms, but we generalize it to consider any bit operations.

If the number of leaves in the tree for the algorithm  $A$  is less than  $M$ , then there must be some object  $I$  that is not given at any leaf. Bob clearly never gives this object as an answer and hence gives the wrong object when this is Alice's object, i.e.,  $A(I) \neq P(I)$ . On the other hand, if the number of leaves in the tree for the algorithm  $A$  is at least  $M$ , then we will find an object  $I$  for which Bob asks at least  $\log_2 M$  questions. There may be some objects for which Bob guesses quickly. For example, if his first question is "Is your number 36?" then such an algorithm works well if Alice happens to be thinking of 36. However, we are considering the worst-case input. Every binary tree with  $M$  leaves must have a leaf with depth of at least  $\log_2 M$ . The object  $I$  that leads to this leaf requires this many questions. We, as the prover in the prover/verifier game, will provide such an instance  $I$ . It follows that  $Time(A, I) \geq T_{lower}(|I|) = \log_2 M$ .

**Exercise 28.2.1** *Prove using strong induction on  $M$  that every binary tree with  $M$  leaves must have a leaf with depth of at least  $\log_2 M$ .*

**Exercise 28.2.2** *How would the lower bound change if a single operation, instead of being only a yes/no question, could be a question with at most  $r$  different answers? Here  $r$  is some fixed parameter.*

**Exercise 28.2.3** *(See solution in Section V) Recall the magic sevens card trick introduced in Section 11.3. Someone selects one of  $n$  cards and the magician must determine what it is by asking questions. Each round the magician rearranges the cards into rows and ask which of the  $r$  rows the card is in. Give an information theoretic argument to prove a lower bound on the number of rounds  $t$  that are needed.*

**Exercise 28.2.4** *(See solution in Section V) Suppose that you have  $n$  objects that are completely identical except that one is slightly heavier. The problem  $P$  is to find the heavier object. You have a scale. A single operation consists of placing any disjoint sets of the objects the two sides of the scale. If one side is heavier then the scale tips over. Give matching upper and lower bounds for this problem.*

**Communication Complexity:** Consider the following problem: Alice has some object from the  $M$  objects  $\{I_1, \dots, I_M\}$  and she must communicate which object she has to Bob by send a string of bits. The string sent will be an *identifier* for the object. The goal is to assign each object a unique identifier so that the longest one has as few bits as possible.

**Lower Bound via a Reduction:** The same lower bounds of  $T_{lower}(M) = \log_2 M$  questions applies. If by way of contradiction, Alice could distinguish the  $M$  objects with fewer bits, then Bob could play the yes/no game by asking Alice in turn for each of these bits.

**Exercise 28.2.5** *State and prove a lower bound when instead of bits Alice can send Bob letters from some fixed alphabet  $\Sigma$ .*

**Lower Bound for Sorting:** We will now prove the classic lower bound that sorting  $N$  values requires at least  $\Omega(N \log N)$  bit operations.

**Only Linear:** Assuming that the  $N$  numbers to be sorted are distinct, most need  $\Theta(\log N)$  bits to be represented for a total of  $n = \Theta(N \log N)$  bits. Hence, the lower bound only says that the number of operations need is at least linear in both the number of bits to represent the input instance and the number to represent the output.

**Comparison Algorithms:** The standard lower bound proved is that no comparison based algorithm is able to sort  $N$  values with fewer than  $\Omega(N \log N)$  comparisons. A *comparison based algorithm* is one whose only way of accessing the input values is by comparing pairs of them, i.e., queries of the form  $a_i \leq a_j$ ? All of the sorting algorithms presented in this text, except for radix/counting, are examples. We will start with this simpler lower bound.

**Decision Tree:** Though an algorithm can be written in your favorite language, one representation of comparison based algorithm is the *decision tree*. The advantage of this representation is that it ignores implementation details that are not important in our task of counting the number of comparisons needed. It ignores all issues of how the flow through the code is controlled, eg. loop and if statements, how the algorithm stores the information that it learns, and how the elements to be sorted are moved from one place to another. It focuses only on determining when *information theoretically* the algorithm has compared enough elements to determine the sorted order.

A decision tree is very similar to the tree representing an algorithm for Bob for the yes/no game. See Figure 28.1.b. It consists of a binary tree. Each node is labeled with a comparison of two elements and the two edges coming out of it are labeled with *yes* and *no*. Each leaf is labeled with one of the  $N!$  permutation of  $N$  elements. The computation proceeds starting at the root and following a path down to a leaf. At each step, the computation performs the comparison labeling the node that it is on and follows the edge corresponding to the answer received. When the computation reaches a leaf, it outputs the permutation of the elements given by its label.

**Connection to the Yes/No Game:** Note that a decision tree that correctly sorts  $N$  values could equally well be used by Bob in the Yes/No game in which Alice chooses an one of the  $M = N!$  permutations of  $N$  values  $\{1, 2, \dots, N\}$  and Bob tries to guess it by asking for the comparisons between pairs of elements of the permutation.

**Lower Bound:** We could use the lower bound for the yes/no game that we have already proved or we could prove it directly. The direct approach takes the decision tree representing some algorithm  $A$ . If the number of leaves in  $A$  is less than  $M = N!$  then we choose a permutation  $I$  that does not appear in any leaf. The sorting algorithm  $A$  will give the wrong answer on this input, i.e.,  $A(I) \neq Sort(I)$ . On the other hand, if the number of leaves is at least  $M = N!$  then there is at least one leaf at depth  $\log_2 M = \log N! = \Omega(N \log N)$ . On the permutation  $I$  that leads to this leaf,  $A$  will require at this many comparisons, i.e.,  $Time(A, I) \geq \Omega(N \log N)$ .

**Math:** Note that in  $N! = 1 \cdot 2 \cdot 3 \cdots \cdot N$ ,  $\frac{N}{2}$  of the factors are at least  $\frac{N}{2}$  and all  $N$  of the factors are at most  $N$ . Hence  $N!$  is in the range  $[\frac{N}{2}^{\frac{N}{2}}, N^N]$ . Hence  $\log N!$  is in the range  $[\frac{N}{2} \log \frac{N}{2}, N \log N]$ .

**Lower Bound for More General Algorithms:** The more general we make the domain of algorithms, the stronger our lower bound becomes, because we are proving that no algorithm even from this larger domain can solve the problem quickly. Now, we will consider algorithms  $A$  as generally as possible.

**The Model of Computation:** First we define what forms we allow our algorithm  $A$  to take and how we measure its performance.

**Language:** The algorithm  $A$  can be written in JAVA, C, or your favorite language.

**Size of an Instance:** There is no restriction on how the  $N$  values in the instance are represented. The size of the instance does not take into account the magnitude or decimal precision of the values. It is only the number of values,  $N$ .

**Single-Bit-Output Operations:** We will measure the running time of the algorithm to be the number of *single-bit-output operations*. These will not be limited to the standard operations *AND*, *OR*, and *NOT*. Each such operation can be of any computational difficulty and can take as input anything about the  $N$  values and about what has been computed so far. The restriction is that each can only output one bit. Any operation of any algorithm can be broken into a number of such operations. For example, taking the sum of  $N$  numbers can be done using one operation for each bit of the output, “What is the first bit of the sum?”, “What is the second?”, ....

**Connection to the Yes/No Game:** We will now show how to translate any sorting algorithm  $A$  into a decision tree that sorts or equivalently into an algorithm for Bob to determine which of the  $M = N!$  permutations Alice has chosen. Each of the permutations Alice may have chosen is a possible input instances to the supposed sorting algorithm  $A$ . Not knowing the input instance that Alice has chosen, Bob will imagine that the  $i^{th}$  element to be sorted is hidden in a closed black box labeled  $i$ . Then he will trace the computation of his algorithm  $A$  on this instance. The only difficulty in tracing  $A$  is that he does not know the values in the black boxes.  $A$ 's actions might, for example, depend on the outcome of an operation like “Is the value in the  $i^{th}$  black box less than that in the  $j^{th}$ ? ” or “Is the third bit of the value in the  $i^{th}$  black box one? ” Bob is able, however, to trace  $A$  until such a single-bit-output operation occurs. At each such points in time, Bob need only ask Alice, who knows the input instance, what the output of this single-bit-output operation will be. With this answer, Bob learns how to trace  $A$  further. After Bob asks for the outcome of each of  $A$ 's operations on this input instance, Bob is able to complete the execution of  $A$ . Seeing how  $A$  manipulates these black boxed values to sort them, he will learn what Alice's initial permutation had been.

**Lower Bound:** The same lower bound of  $\log_2 M = \log_2 N! = \Omega(N \log N)$  single-bit-output operations (or queries to Alice) apply to these more general decision trees as we saw with the comparison based ones. Hence, sorting  $N$  values requires at least  $\Omega(N \log N)$  of these bit operations.

**Lower Bound for Very Poor Algorithm:** Even a very poor algorithm that sorts  $N$  values correctly on only some  $p_{\text{succes}} \geq \frac{1}{2^{\Theta(N)}}$  fraction of the input instances requires at least  $T_{\text{lower}}(N) = \Omega(N \log N)$  bit operations.

**Proof:** There are  $N! = 2^{\Theta(N \log N)}$  input instances. If an algorithm works correctly on  $p_{\text{succes}} \geq \frac{1}{2^{\Theta(N)}}$  fraction, then it still works correctly on  $p_{\text{succes}} \cdot N! \geq \frac{1}{2^{\Theta(N)}} \cdot 2^{\Theta(N \log N)} = 2^{\Theta(N \log N)}$  instances. Ignore the other inputs. If the algorithm  $A$ , when viewed as a decision tree, has fewer than this number of leaves, then it cannot give the correct answer for all of these. Otherwise, its height and hence its running time must be at least  $\log_2(p_{\text{succes}} \cdot N!) = \Omega(N \log N)$ .

**A Lower Bound for Randomized Sorting:** We will now prove that even a very poor randomized algorithm is not able to do significantly better.

**Theorem:** Even a poor randomized algorithm that sorts  $N$  values correctly with probability

of only  $p_{\text{succes}} \geq \frac{1}{2^{\Omega(n)}}$ , requires at least  $T_{\text{lower}}(N) = \Omega(N \log N)$  bit operations. Stated another way, if the algorithm use less time, then it succeeds less often.

$$\forall A, \exists I, \Pr_r [A_r(I) = P(I) \text{ and } \text{Time}(A_r, I) \leq T_{\text{lower}}(|I|)] \leq p_{\text{succes}}$$

**Proof:** By way of contradiction, suppose that there is a randomized algorithm  $A$  which for each input instance  $I$  sorts correctly in this time with this probability  $p_{\text{succes}}$ . From this we will find a deterministic algorithm  $A'$  that sorts some  $p_{\text{succes}}$  fraction of the input instances in this much time. This contradicts the above information theoretic argument proving that this is impossible. Hence, such a randomized algorithm cannot exist either.

**Random Coins VS Random Instances:** Given a random algorithm that works well for every instance for most coin flips, we can construct a deterministic algorithm that works well for most instances. Namely,

$$\begin{aligned} \text{if } & \exists A, \forall I, \Pr_r [A_r(I) = P(I) \text{ and } \text{Time}(A_r, I) \leq T_{\text{lower}}(|I|)] \geq p_{\text{succes}}, \\ \text{then } & \exists A', \quad \Pr_I [A'(I) = P(I) \text{ and } \text{Time}(A', I) \leq T_{\text{lower}}(|I|)] \geq p_{\text{succes}}. \end{aligned}$$

**Proof:** Given the randomized algorithm  $A$ , construct a matrix  $b_{\langle I, r \rangle}$  with a row for each instances  $I$  and a column for each outcome  $r = \langle \text{heads}, \dots, \text{tails} \rangle$  of the coin flips and  $b_{\langle I, r \rangle} = \text{succes}$  if  $[A_r(I) = P(I) \text{ and } \text{Time}(A_r, I) \leq T_{\text{lower}}(|I|)]$ . Because  $\forall I, \Pr_r [A_r(I) = P(I) \text{ and } \text{Time}(A_r, I) \leq T_{\text{lower}}(|I|)] \geq p_{\text{succes}}$ , we know that the fraction of *succes* in each row is at least  $p_{\text{succes}}$ . It follows that the fraction of *succes* in the entire matrix is at least  $p_{\text{succes}}$ . It follows that there must be a column index by some coin flips  $r$  which has at least a  $p_{\text{succes}}$  fraction of *succes*. Deterministically fixing the coin flips to such an  $r$  gives a deterministic algorithm  $A' = A_r$  which succeeds on at least a  $p_{\text{succes}}$  fraction of the instances  $I$ .

**Theorem:** Any randomized algorithm that always sorts  $N$  values correctly but whose running time depends on the coin flips has an expected running time of at least  $T_{\text{lower}}(N) = \Omega(N \log N)$  bit operations.

$$\forall A, \exists I, \text{Exp}_r [\text{Time}(A_r, I)] \geq T_{\text{lower}}(|I|)$$

**Proof:** Consider any algorithm that always correctly sorts. The above theorem proves that the probability that it sorts faster than  $T_{\text{lower}}(N) = \Omega(N \log N)$  is at most  $p_{\text{succes}} \leq \frac{1}{2^{\Omega(n)}}$ . To be generous, let's assume that when it sorts faster, then it sorts in zero time. Even then, the expect running time is at least

$$\begin{aligned} \text{Exp}_r [\text{Time}(A_r, I)] &= \sum_r \frac{1}{\text{The } \# \text{ of different } r} \cdot \text{Time}(A_r, I) = \sum_e \Pr[e] \cdot \text{Time}_e \\ &\geq \frac{1}{2^{\Omega(n)}} \cdot 0 + \left(1 - \frac{1}{2^{\Omega(n)}}\right) \cdot \Omega(N \log N) = \Omega(N \log N). \end{aligned}$$

In conclusion, randomized sorting algorithms are not significantly faster than  $\Omega(N \log N)$ .

# **Part V**

## **Exercise Solutions**

### 3.4.1

$$\begin{aligned}
7 \cdot 2^{3n^5} + 9n^4 \log^7 n &\in 7 \cdot 2^{3n^5} + \Theta(n^4 \log^7 n) \\
7 \cdot 2^{3n^5} + n^4 \log^8 n &\in 7 \cdot 2^{3n^5} + \tilde{\Theta}(n^4) \\
7 \cdot 2^{3n^5} + n^5 &\in 7 \cdot 2^{3n^5} + n^{\Theta(1)} \\
7 \cdot 2^{3n^5} + n^{\log n} &\in (7 + o(1))2^{3n^5} \\
8 \cdot 2^{3n^5} &\in \Theta(2^{3n^5}) \\
2^{4n^5} &\in 2^{\Theta(n^5)} \\
2^{n^6} &\in 2^{n^{\Theta(1)}}
\end{aligned}$$

**5.1.1** The sums are approximated as follows.

1.  $\sum_{i=1}^n \sum_{j=1}^n i^j = \sum_{i=1}^n \Theta(i^n) = \Theta(n \cdot n^n)$
2.  $\sum_{i=1}^n \sum_{j=1}^n j^i$ . Clearly this is equal to the previous sum. But good to do again.  $= \sum_{i=1}^n \Theta(n \cdot n^i) = \Theta(n \cdot n^n)$ .
3.  $\sum_{i=1}^n \sum_{j=1}^i \frac{1}{j} = \sum_{i=1}^n \Theta(\ln(i)) = \Theta(n \ln(n))$ .
4.  $\sum_{i=1}^n \sum_{j=1}^{i^2} ij \log(i) = \sum_{j=1}^m ij \log(i) = i \log(i) \sum_{j=1}^m j = i \log(i) \Theta(m^2)$ . Hence,  $\sum_{j=1}^{i^2} ij \log(i) = \Theta(i^5 \log(i))$  and  $\sum_{i=1}^n \Theta(i^5 \log(i)) = \Theta(n^6 \log(n))$ .

**5.2.1** His first such step takes him half an hour, his second a quarter, his third an eighth, ... for a total of only  $\sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots = 1$  hour. Given that he travels one kilometer at one kilometer an hour, this is reasonable.

**5.2.2** If  $\frac{f(i+1)}{f(i)} \leq b < 1$ . Unwinding gives  $f(i) \leq b^1 f(i-1) \leq b^{i-n_0} f(n_0)$ . This gives that  $\sum_{i=1}^n f(i) = \sum_{i=1}^{n_0} f(i) + \sum_{i=n_0}^n f(i) \leq \Theta(1) + \sum_{i=n_0}^n b^{i-n_0} f(n_0) \leq \Theta(1) + f(n_0) \cdot \sum_{j=0}^n b^j$ , which we have already proved is  $\Theta(f(n_0)) = \Theta(1)$ .

**5.2.3** The problem arises when the function grows so quickly that  $f(n+1) \neq \Theta(f(n))$ . For example, let  $f(x) = e^x \cdot e^{e^x}$  (chosen because it grows very quickly and is easy to integrate).  $\sum_{i=1}^n f(i)$  is at least the last term  $e^n \cdot e^{e^n}$ ,  $\int_{x=1}^n f(x) \delta x$  is only  $e^{e^n}$  which is much smaller, and  $\int_{x=1}^{n+1} f(x) \delta x$  is  $e^{e^{n+1}}$  which is much bigger.

**6.4.1** We use the above table method to compute each of the following recursive relations.

	$T(n) = nT(n-1) + 1$	$T(n) = 2T(\sqrt{n}) + n$
a) # frames at the $i^{th}$ level?	$n \cdot (n-1) \cdots (n-i+1)$	$2^i$
b) Size of instance at the $i^{th}$ level?	$n - i$	$n^{2^{-i}}$
c) Time within one stack frame	1	$n^{2^{-i}}$
d) # levels	$n - h = 0$ $h = n$	$n^{2^{-h}} = 2$ $h = \log \log n$
e) # base case stack frames	$n!$	$2^h = 2^{\log \log n} = \log n$
f) $T(n)$ as a sum	$\sum_{i=0}^n n \cdot s \cdot (n-i+1) \cdot 1$	$\sum_{i=0}^{\log \log n} 2^i \cdot n^{2^{-i}}$
g) Dominated by?	base	top level
h) $\Theta(T(n))$	$T(n) = \Theta(n!)$	$T(n) = \Theta(n)$

**10.2.2** The tests will be executed in the order that they are listed. If  $next = nil$  is tested first and passes, then because there is an *or* between the conditions there is no need to test the second. However, if  $next.info \geq key$  was the first test and  $next$  was nil, then using  $next.info$  to retrieve the information in the node pointed to by  $next$  would cause a run-time error.

**14.4.2** To prove  $S(0)$ , let  $n = 0$  in the inductive step. There are no values  $k$  where  $0 \leq k < n$ . Hence, no assumptions are being made. Hence, your proof proves  $S(0)$  on its own.

**14.5.2**  $\text{Fun}(1) = X,$

$\text{Fun}(2) = Y,$

$\text{Fun}(3) = AYBXC,$

$\text{Fun}(4) = A AYBXC B Y C,$

$\text{Fun}(5) = A AAYBXCBYC B AYBX C,$

$\text{Fun}(6) = A AAAYBXCBYCBAYBXC B AAYBXCBYC C.$

**15.1.1** Insertion Sort and Selections Sort.

- 16.4.1**
- Where in the heap can the value 1 go? 1 must be in one of the leaves. If 1 was not at a leaf, then the nodes below it would need a smaller number, of which there are none.
  - Which values can be stored in entry  $A[2]$ ?  $A[2]$  can contain any value in the range 7-14. It can't contain 15, because 15 must go in  $A[1]$ . From above, we know that  $A[2]$  must be greater than each of the seven nodes in its subtree. Hence, it can't contain a value less than 7. For each of the other values, a heap can be constructed such that  $A[2]$  has that value.
  - Where in the heap can the value 15 go? 15 must go in  $A[1]$  (see above).
  - Where in the heap can the value 6 go? 6 can go anywhere except  $A[1]$ ,  $A[2]$ , or  $A[3]$ .  $A[1]$  must contain 15, and  $A[2]$  and  $A[3]$  must be at least 7.

**16.5.1**

```

algorithm Derivative(f,x)
<pre-cond>: f is an equation and x is a variable
<post-cond>: The derivative of f with respect to x is returned.
    if( f = "x" ) then
        result( 1, constructed by
            -- 1 )

    else if( f = a real value or a single variable other than "x" ) then
        result( 0, constructed by
            -- 0 )
    end if

    % if f is of the form (g op h)
    g = Copy( leftSub(f) ) % Copy needed for "*" and "/".
    h = Copy( rightSub(f) ) % Three copies needed for "/".
    g' = Derivative( leftSub(f), x )
    h' = Derivative( rightSub(f), x )

    if   ( f = g+h ) then
        result( g'+h', constructed by
            |-- h'
            --- + -|
            |-- g' )

    else if( f = g-h ) then
        result( g'-h', constructed by
            |-- h'
            --- - -|
```

```

else if( f = g*h ) then
  result( g'*h + g*h', constructed by
    ie           |-- h'
                  |- * -
                  |
                  |-- g
    -- + -|
                  |
                  |-- h
                  |- * -
                  |-- g' )

else if( f = g/h ) then
  result( g'*h - g*h')/(h*h), constructed by
                  |-- h
                  |- * -
                  |
                  |-- h
    -- / -|
                  |
                  |-- h'
                  |   |- * -
                  |   |
                  |   |-- h
                  |- - -|
                  |
                  |-- h
                  |- * -
                  |-- g' )

end if

```

**16.6.1** This could be accomplished by passing an integer giving the level of recursion.

**16.6.2** Though the left subtree is indicated on the left of the following tracing, it is recursed on after the right subtree.

```

PrettyPrint
+---+
| f
|_-
+---+
GenPrettyPrint
|_
+---+
|dir = root
|prefix =
|""
|print:
|   |-- 3
|   |- * -|   |-- 2
|   |   |   |- \ -|   |-- 4
|   |   |   |-- + -|
|   |-- + -|   |-- 1
|   |-- 5
+---+
|_
V_-----+-----+
|dir = left      |dir = right
|prefix =        |prefix =
| bbbbbbb       | bbbbbbb
|print:          |print:
|   |-- 5          |   |-- 3
+---+-----+-----+
|_-----+-----+-----+

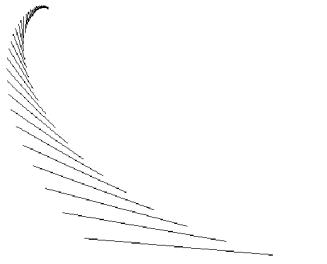
```

```

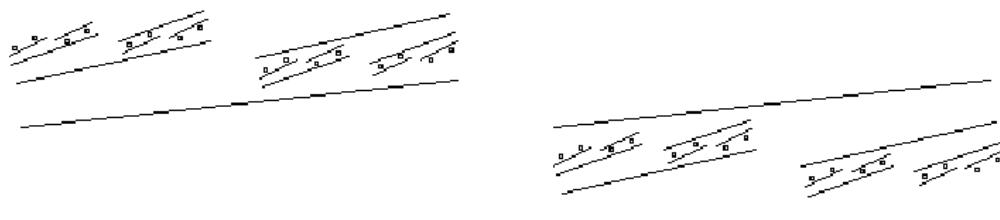
----- V ----- V -----
|dir = left      | |dir = right    |
|prefix =       | |prefix =      |
| bbbbbbb|bbbbbb | | bbbbbbbbbb|bbb
|print:        | |print:        | | | | | |
|   |   |   |-- 2 | |   |   |-- 3 |
|   |   |   | / - | |   |   |-----|
|   |   |   |-- 4 | |   |   |
|   |   | + - | |   |   |-----|
|   |   |-- 1 | |   |   |
|   |   |-----| |   |   |
|   |   |-----| |   |   |
----- V ----- V -----
|dir = left      | |dir = right    |
|prefix =       | |prefix =      |
| bbbbbbb|bbbbbb|bbb | | bbbbbbb|bbb|bbb|bbb
|print:        | |print:        | | | | | | | |
|   |   |   |-- 1 | |   |   |   |   |-- 2 |
|   |   |   |-----| |   |   |   | / - | |
|   |   |   |-----| |   |   |   |-- 4 |
|   |   |   |-----| |   |   |   |
----- V ----- V -----
|dir = left      | |dir = right    |
|prefix =       | |prefix =      |
| bbbbbbb|bbbbbb|bbb|bbb|bbb | | bbbbbbb|bbb|bbb|bbb|bbb|bbb
|print:        | |print:        | | | | | | | | | |
|   |   |   |   |-- 4 | |   |   |   |   |   |-- 2 |
|   |   |   |   |-----| |   |   |   |   |   |

```

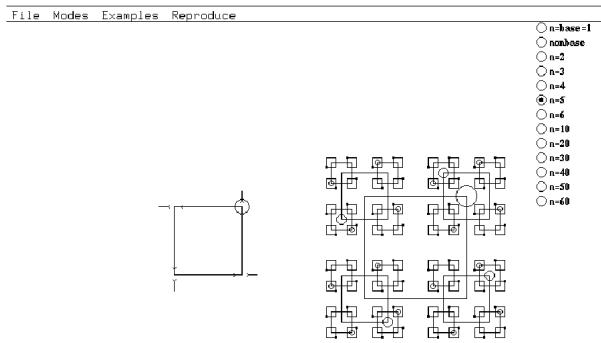
**17.1.1 Falling Line:** This construction consists of a single line with the  $n - 1$  image raised, tilted, and shrunk.



**17.1.2 Binary Tilt:** This image is the same as the birthday cake. The only differences are that the two places to recurse are tilted and one of them has been flipped upside down.



**17.1.3 Recursing at the Corners of a Square:** The image for  $n$  consists of a square with the  $n - 1$  image at each of its corners rotated in some way. There is also a circle at one of the corners. The base case is empty.



```

18.0.1 s = ( ( ( 1 ) * 2 + 3 ) * 5 * 6 + 7 )
           |-----|
           |-exp-----|
           |-term-----|
           |-fact-----|
           ( |-exp-----| )
           |-term-----| + t
           |-fact-----| * f * f   f
           ( |-exp-----| )   5   6   7
           |-t-----| + t
           |-f-| * f   f
           ( e )   2   3
           t
           f
           1
s = ( ( ( 1 ) * 2 + 3 ) * 5 * 6 + 7 )

```

**20.2.1** The node  $v$  can't be in  $V_{k'}$  for  $k' > k+1$  because there is a path of length  $k+1$  to it, namely the path to  $u$  followed by the edge  $\langle u, v \rangle$ . If  $v$  has not been found before, then we are just finding it. Its  $d(v)$  is being set to  $d(u) + 1 = k + 1$ . By LI1, this must be its distance from  $s$ . Hence,  $v$  must be in  $V_{k+1}$ . If  $v$  has been found before, it is because a shortest path has already been found to it. If the edge  $\langle u, v \rangle$  is directed, then this previous path could have any length  $k' \leq k + 1$ . However, if this edge is undirected, then there is a catch. Suppose  $v$  is in  $V_{k'}$ . Then a possible path to  $u$  is that of length  $k' + 1$  from  $s$  to  $v$  and then following the edge  $\{u, v\}$  backwards to  $u$ . Since, the shortest path to  $u$  is of length  $k$ , we have  $k' + 1 \geq k$  or  $k' \in \{k - 1, k, k + 1\}$ .

**20.3.1** Despite differences in the algorithms, on a graph with edge weights one, the BFS and Dijkstra algorithms are identical. BFS handles the first node in its queue while Dijkstra handles the node with the next smallest  $d(v)$ . However, BFS's third loop invariant assures that the nodes are found and added to the queue in the order of distance  $d(v)$ . Hence, handling the next in the queue amounts to handling the next smallest  $d(v)$ . BFS's first loop invariant states that the correct minimal distance  $d(v)$  to  $v$  is obtained when the node  $v$  is first found, while with Dijkstra we are not sure to have it until the node is handled. However, with edge weights one, when  $v$  is first found in Dijkstra's algorithm,  $d(v)$  is set to the length of the over shortest path and never changed again.

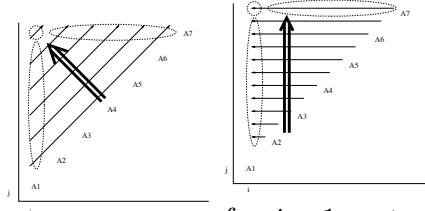
**22.2.2** In the following instance of the interval cover problem, the greedy criteria that selects the

interval that covers the largest number of uncovered points would commit to the top interval. However, the optimal solution does not contain this interval, but contains the bottom two intervals.

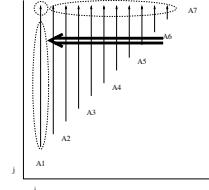


**23.2.2** The line  $k_{min}$  = “a  $k$  that maximizes  $cost_k$ .”

#### 24.6.2



According to  
size.  
for  $j = 1$  up to  $n$   
for  $i = j$  down to 1  
Solve instance  $\langle A_i, \dots, A_j \rangle$



for  $i = n$  down to 1  
for  $j = i$  up to  $n$   
% Solve instance  $\langle A_i, \dots, A_j \rangle$

**24.2.1** If  $x_n = y_m$ , then we must prove that there is at least one optimal solution that contains both of these last characters. Consider an optimal solution. It must end in this last character, otherwise it can be extended to contain it. It might not contain both of them, as in  $X = \langle \underline{A}, \underline{B}, B \rangle$ ,  $Y = \langle \underline{A}, \underline{B} \rangle$ , and optimal solution  $Z = \langle A, B \rangle$ . However, as in this case, we can just as well assume that the optimal solution takes both last characters. If  $x_n \neq y_m$ , then the optimal solution cannot take both of these last characters. Hence, it either does not take the last of  $X$  or does not take the last of  $Y$ . It might not take the last of both, but this is included in both the other two cases. See Section 25.1 for a further answer.

- 25.4.1** 1.  $\langle 1, 5, 8, 6, 3, 7, 2, 4 \rangle$  2.  $\langle 1, 6, 8, 3, 7, 4, 2, 5 \rangle$   
 3.  $\langle 1, 7, 4, 6, 8, 2, 5, 3 \rangle$  4.  $\langle 1, 7, 5, 8, 2, 4, 6, 3 \rangle$   
 5.  $\langle 2, 4, 6, 8, 3, 1, 7, 5 \rangle$  6.  $\langle 2, 5, 7, 1, 3, 8, 6, 4 \rangle$   
 7.  $\langle 2, 5, 7, 4, 1, 8, 6, 3 \rangle$  8.  $\langle 2, 6, 1, 7, 4, 8, 3, 5 \rangle$   
 9.  $\langle 2, 6, 8, 3, 1, 4, 7, 5 \rangle$  10.  $\langle 2, 7, 3, 6, 8, 5, 1, 4 \rangle$   
 11.  $\langle 2, 7, 5, 8, 1, 4, 6, 3 \rangle$  12.  $\langle 2, 8, 6, 1, 3, 5, 7, 4 \rangle$

**25.4.2** We will prove that the running time is bounded between  $(\frac{n}{2})^{\frac{n}{6}}$  and  $n^n$  and hence is  $n^{\Theta(n)} = 2^{\Theta(n \log n)}$ . Without any pruning, there are  $n$  choices on each of  $n$  rows as to where to place the row's queen. This gives  $n^n$  different placements of the queens. Each of these solutions would correspond to a leaf of the tree of stack frames. This is clearly an upper bound on number when there is pruning.

We will now give a lower bound on how many stack frames will be executed by this algorithm. Let  $j$  be one of the first  $\frac{n}{6}$  rows. I claim that each time that a stack frame is placing a queen on this row, it has at least  $\frac{n}{2}$  choices as to where to place it. The stack frame can place the queen on any of the  $n$  squares in the row as long as this square cannot be captured by one of the queens placed above it. If row  $i$  is above our row  $j$ , then the queen placed on row  $i$  can capture at most three squares of row  $j$ : one by moving on a diagonal to the left, one by moving straight down, and one by moving on a diagonal to the right. Because  $j$  is one of the first  $\frac{n}{6}$  rows, there is at most this number of rows  $i$  that are above it and hence at most  $3 \times \frac{n}{6}$

of row  $j$ 's squares can be captured. This leaves, as claimed,  $\frac{n}{2}$  squares on which the stack frame can place the queen.

From the above claim, it follows that within the tree of stack frames, each stack frame within the tree's first  $\frac{n}{6}$  levels branches out to at least  $\frac{n}{2}$  children. Hence, at the  $(\frac{n}{6})^{th}$  level of the tree there are at least  $(\frac{n}{2})^{\frac{n}{6}}$  different stack frames. Many of these will terminate without finding a complete valid placement. However, this is a lower bound on the running time of the algorithm because the algorithm recurses to each of them.

**27.1.1** Clearly, the statement  $\Pr[A|B] = \frac{\Pr[A \text{ and } B]}{\Pr[B]} = \Pr[A]$  is true if and only the statement  $\Pr[A \text{ and } B] = \Pr[A] \cdot \Pr[B]$  is true.

**27.1.2**  $\Pr[A \text{ and not } B] = \Pr[A] - \Pr[A \text{ and } B] = \Pr[A] - \Pr[A] \cdot \Pr[B] = \Pr[A] \cdot (1 - \Pr[B]).$   
 $\Pr[\text{not } A \text{ and not } B] = \Pr[\text{not } B] - \Pr[A \text{ and not } B] = (1 - \Pr[B]) - \Pr[A] \cdot (1 - \Pr[B]) = (1 - \Pr[A]) \cdot (1 - \Pr[B])$

**27.1.3**  $\text{Exp}[V] = \sum_{\text{disjoint events } A} \Pr[A] \cdot [\text{value of } V \text{ during event } A]$   
 $= \sum_v \sum_{\text{disjoint events } A \text{ for which } V = v} \Pr[A] \cdot v$   
 $= \sum_v \Pr[V = v] \cdot v.$

Obtaining the coin flips  $r$  is like an event  $A$  with  $\Pr[A] = \frac{1}{\text{The } \# \text{ of } r}$ . Hence,

$$\text{Exp}[V] = \sum_{\text{disjoint events } A} \Pr[A] \cdot [\text{value of } V \text{ during event } A] = \sum_r \frac{1}{\text{The } \# \text{ of } r} \cdot V_r.$$

**27.1.4** The proof that the expectation of the sum is equal to the sum of the expectations is as follows.  $\text{Exp}[U + V] = \sum_w \Pr[U + V = w] \cdot w = \sum_u \sum_v \Pr[U = u \text{ and } V = v] \cdot (u + v)$   
 $= [\sum_u \sum_v \Pr[U = u \text{ and } V = v] \cdot u] + [\sum_u \sum_v \Pr[U = u \text{ and } V = v] \cdot v]$   
 $= [\sum_u u \cdot [\sum_v \Pr[U = u \text{ and } V = v]]] + [\sum_v v \cdot [\sum_u \Pr[U = u \text{ and } V = v]]]$   
 $= [\sum_u u \cdot \Pr[U = u]] + [\sum_v v \cdot \Pr[V = v]] = \text{Exp}[U] + \text{Exp}[V]$

**27.1.5** The proof that if the random variables are independent than the expectation of the product is equal to the product of the expectations is as follows.

$$\begin{aligned} \text{Exp}[U \times V] &= \sum_w \Pr[U \times V = w] \cdot w = \sum_u \sum_v \Pr[U = u \text{ and } V = v] \cdot (u \times v) \\ &= \sum_u \sum_v [\Pr[U = u] \cdot \Pr[V = v]] \cdot (u \times v) \\ &= \sum_u \sum_v [\Pr[U = u] \cdot u] \times [\Pr[V = v] \cdot v] \\ &= [\sum_u \Pr[U = u] \cdot u] \times [\sum_v \Pr[V = v] \cdot v] = \text{Exp}[U] \times \text{Exp}[V] \end{aligned}$$

**27.1.6** The proof that if the random variables are dependent than it is not necessarily true that the expectation of the product is equal to the product of the expectations is as follows. Suppose that  $V_1 = V_2 = 0$  with probability  $\frac{1}{2}$  and  $V_1 = V_2 = 2$  with probability  $\frac{1}{2}$ . Then  $\text{Exp}[V_1] = \text{Exp}[V_2] = \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 2 = 1$ .  $\text{Exp}[V_1 \cdot V_2] = \frac{1}{2} \cdot (0 \cdot 0) + \frac{1}{2} \cdot (2 \cdot 2) = 2$ . This is different than  $\text{Exp}[V_1] \cdot \text{Exp}[V_2]$ .

**27.1.7** We prove Markov's tail inequality as follows. Let  $V$  be a random variable that only takes on non-negative values and  $v$  is any fixed value. Let  $X$  be the random variable which equals  $v$  if  $V \geq v$  and 0 otherwise.  $\text{Exp}[V] \geq \text{Exp}[X] = v \cdot \Pr[V \geq v]$ . Rearranging give that  $\Pr[V \geq v] \leq \frac{\text{Exp}[V]}{v}$ .

**27.1.8** We prove Chebyshev's tail inequality as follows. Let  $V$  be a random variable and  $h$  is any fixed value. Let  $Y = (V - \text{Exp}[V])^2$  be a random variable. By definition,  $\text{Exp}[Y] = \text{SD}[V]^2$ . Hence, by Markov's inequality  $\Pr[|V - \text{Exp}[V]| \geq h] = \Pr[Y \geq h^2] \leq \frac{\text{Exp}[Y]}{h^2} = \frac{\text{SD}[V]^2}{h^2}$ .

**27.1.9** We prove a weaker version of Chernoff's tail inequalities, namely that the probability of getting fewer than  $pn - h$  successes is at most  $e^{-h^2/(4pn)}$ .

**Probability  $P_x$  of  $x$  successes:** Let  $P_x$  denote the probability that there are exactly  $x$  successes when running  $n$  independent experiments each with success probability  $p$ . We will argue that

$$P_x = \binom{n}{x} p^x (1-p)^{n-x} = \frac{n!}{x!(n-x)!} \binom{n}{x} p^x (1-p)^{n-x}.$$

This is because there are  $\binom{n}{x}$  ways of “choosing”  $x$  of the  $n$  experiments to be the ones that will succeed. For each of these ways of choosing, the probability that the  $x$  chosen experiments succeed is  $p^x$  and the probability that the  $n - x$  experiments not chosen fail is  $(1-p)^{n-x}$ .

**Don't Expect the Expected:** Surprisingly, one can not expect to get exactly the expected number successes. The expected number is  $E = pn$ . The probability that there are exactly  $E$  successes is  $P_E = \binom{n}{E} \left(\frac{E}{n}\right)^E \left(\frac{n-E}{n}\right)^{n-E}$ . We will prove that  $P_E \leq \frac{e}{\sqrt{E}}$  (here  $e = 2.718..$  is the base of natural logarithms).

**The Ratio  $\frac{P_{E-h}}{P_E}$ :** Let us compare the probabilities of getting exactly the expected  $E$  and exactly  $h$  fewer than this.

$$\begin{aligned} \frac{P_{(E-h)}}{P_E} &= \frac{n!}{(E-h)!(n-E+h)!} \left(\frac{E}{n}\right)^{E-h} \left(\frac{n-E}{n}\right)^{n-E+h} \times \frac{E!(n-E)!}{n!} \left(\frac{n}{E}\right)^E \left(\frac{n}{n-E}\right)^{n-E} \\ &= \frac{E!}{(E-h)! E^h} \times \frac{(n-E)!(n-E)^h}{(n-E+h)!} \\ &= \frac{(E)}{E} \frac{(E-1)}{E} \cdots \frac{(E-h+1)}{E} \times \frac{(n-E)}{(n-E+h)} \frac{(n-E)}{(n-E+h-1)} \cdots \frac{(n-E)}{(n-E+1)} \end{aligned}$$

The first  $\frac{h}{2}$  of these factors are at most one, the next  $\frac{h}{2}$  are at most  $\frac{(E-h/2)}{E}$ . The next  $h$  are at most one. This gives

$$\frac{P_{(E-h)}}{P_E} \leq \left(\frac{E-h/2}{E}\right)^{h/2} \leq \left(1 - \frac{h}{2E}\right)^{h/2} \leq \left(e^{-\frac{h}{2E}}\right)^{h/2} = e^{-\frac{h^2}{4E}}$$

The tricky previous inequality comes from a formal definitions of the base of natural logarithms  $2.718..$ , i.e.  $\lim_{N \rightarrow \infty} (1 - \frac{1}{N})^N = e^{-1}$ . Also if you plot the two functions  $1 - x$  and  $e^{-x}$ , you can see that when  $1 - x \leq e^{-x}$ . Setting  $x = \frac{h}{2E}$  gives the required relation.

**Deviating a Standard Deviation:** When  $h \in [-\frac{1}{2}\sqrt{E}, \frac{1}{2}\sqrt{E}]$  and  $p \leq \frac{1}{2}$  a similar calculation to that above gives that  $\frac{P_{(E+h)}}{P_E} \geq e^{-\frac{h^2}{n-E}} \cdot e^{-\frac{h^2}{E}} \geq e^{-1}$ . This gives us that probability of there being  $E$  plus or minus  $\frac{1}{2}\sqrt{E}$  successes is  $1 \geq \sum_{h=-1/2\sqrt{E} \dots 1/2\sqrt{E}} P_{(E+h)} \geq \sqrt{E} \cdot e^{-1} \cdot P_E$ . This gives the result that  $P_E \leq \frac{e}{\sqrt{E}}$ .

**Bounded Tail:** The probability of getting fewer than  $E - h$  successes is

$$\sum_{h' \geq h} P_{E-h'} \leq \sum_{h' \geq h} e^{-\frac{h^2}{4E}} \cdot P_E \approx \frac{1}{\frac{h}{2E}} \cdot e^{-\frac{h^2}{4E}} \cdot \frac{e}{\sqrt{E}} \leq 5 \cdot e^{-\frac{h^2}{4E}}.$$

**27.1.10** Suppose the probability that the  $i^{th}$  bad thing happens is at most  $p_i$ . The worst case is when these bad events are disjoint so that two never occur simultaneously. Imagine a ven diagram with disjoint circles of area  $p_i$ . In this case, the probability that one happens is exactly  $\sum_i p_i$ . More formally,  $\Pr[\text{At least one bad thing happens}] = \underline{\text{The } \# \text{ of } r \text{ for which at least one bad thing happens}}$

$$\begin{aligned} &\text{The } \# \text{ of } r \\ &\leq \sum_i \frac{\text{The } \# \text{ of } r \text{ for which the } i^{th} \text{ bad thing occurs}}{\text{The } \# \text{ of } r} = \sum_i p_i. \end{aligned}$$

### 27.1.11

**$1 + p \leq e^p$  and  $1 - p \leq e^{-p}$ :** These come from two formal definitions of the base of natural logarithms 2.718.., i.e.  $\lim_{N \rightarrow \infty} (1 + \frac{1}{N})^N = e$  and  $\lim_{N \rightarrow \infty} (1 - \frac{1}{N})^N = e^{-1}$ . Also if you plot the two functions  $1 + x$  and  $e^x$  using the fact that the derivative of both  $1 + x$  and  $e^x$  at zero is 1, you can see that  $1 + x \leq e^x$  and similarly  $1 - x \leq e^{-x}$ . These approximations are very close when  $x \in o(1)$ .

**$1 + p + p^2 \geq e^p$  and  $1 - p + \frac{p^2}{2} \geq e^{-p}$ :** The Taylor expansion of a function  $f(x)$  at point  $x_0$  is a close approximation of  $f(x)$  for  $x$  close to  $x_0$ . It is defined to be  $f(x_0 + p) \approx f(x_0) + f'(x_0)p + \frac{1}{2!}f''(x_0)p^2 + \frac{1}{3!}f'''(x_0)p^3 + \dots$ . Hence,  $e^p \approx e^0 + e^0p + \frac{1}{2!}e^0p^2 + \frac{1}{3!}e^0p^3 + \dots = 1 + p + \frac{1}{2!}p^2 + \frac{1}{3!}p^3 + \dots \leq 1 + p + p^2$  when  $p \leq 1$ . Replacing  $p$  with  $-p$  gives  $e^{-p} \approx 1 + (-p) + \frac{1}{2!}(-p)^2 + \frac{1}{3!}(-p)^3 + \dots \leq 1 - p + \frac{p^2}{2}$ .

**$(1 - p)^n = 1 - np + \Theta(p^2)$ :** You know that  $(1 - p)^2 = 1 - 2p + p^2$  and  $(1 - p)^3 = 1 - 3p + 3p^2 - p^3$ . More generally,  $(a + b)^n = \sum_{i=0}^n \binom{n}{i} a^{n-i} b^i$  and hence  $(1 - p)^n = \sum_{i=0}^n \binom{n}{i} (-p)^i = 1 - np + \frac{n^2}{2}p^2 - \Theta(p^3)$ .

**$n! \approx \left(\frac{n}{e}\right)^n$ :**  $\ln(n!) = \ln(1 \cdot 2 \cdot 3 \cdot \dots \cdot n) = \ln(1) + \ln(2) + \ln(3) + \dots + \ln(n) = \sum_{i=1}^n \ln(i) \approx \sum_{i=1}^n \ln(i) = n \ln(n) - n$ . Hence,  $n! = e^{\ln(n!)} = e^{n \ln(n) - n} = \left[e^{\ln(n)}\right]^n \cdot e^{-n} = \frac{n^n}{e^n}$ .

**$\left(\frac{n}{a}\right)^a \leq \binom{n}{a} \leq \left(\frac{en}{a}\right)^a$ :**  $\binom{n}{a} = \frac{n!}{a!(n-a)!} = \frac{n(n-1)(n-2)\dots(n-a+1)}{a(a-1)(a-2)\dots 1} = \frac{n}{a} \cdot \frac{n-1}{a-1} \cdot \frac{n-2}{a-2} \dots \frac{n-a+1}{1} \geq \left(\frac{n}{a}\right)^a$ .  
 $\binom{n}{a} = \frac{n!}{a!(n-a)!} = \frac{n(n-1)(n-2)\dots(n-a+1)}{a!} \leq \frac{n^a}{a!} \approx \frac{n^a}{(a/e)^a} = \left(\frac{en}{a}\right)^a$ .

**28.1.1** It is true for any problem  $P$  and time complexities  $T_{upper}$  that give enough time to output the answer. Consider any input  $I$ .  $I$  is some fixed string.  $P$  has some fixed output  $P(I)$  on this string. Let  $A_{P(I)}$  be the algorithm that does not look at the input but simply outputs the string  $P(I)$ . This algorithm gives the correct answer on input  $I$  and runs quickly.

**28.2.1** We prove using strong induction on  $M$  that every binary tree with  $M$  leaves must have a leaf with depth of at least  $\log_2 M$ . As a base case, a tree with  $M = 1$  leaf has depth of at least  $\log_2 1 = 0$ . Suppose it was true for every  $M' < M$ . Consider a binary tree with  $M$  leaves. The number of leaves in the left subtree of the root plus the number in the right sum to  $M$ . Hence, at least one of these is at least  $\frac{M}{2}$  and hence by induction has depth of at least  $\log_2 \frac{M}{2} = (\log_2 M) - 1$ . The depth of our tree is at least one more than that of this subtree.

**28.2.3** The bound is  $n \leq r^t$ .

Each round, he selects one row, hence  $r$  possible answers. After  $t$  rounds, there are  $r^t$  combinations of answers possible.

The only information that you know is which of these combinations he gave you. Which card you produce depends deterministically (no magic) on the combination of answers given to you. Hence, depending on his answers, there are at most  $r^t$  cards that you might output.

However, there are  $n$  cards any of which may be the selected card. In conclusion,  $n \leq r^t$ .

The book has  $n = 21$ ,  $r = 3$ , and  $t = 2$ . Because  $21 = n \not\leq r^t = 3^2 = 9$ , the trick in the book does NOT work.

Two rounds is not enough. There needs to be three rounds.

**28.2.4** It is a trick question, because with a balance there are three not two outcomes and hence only  $\log_3 n$  operations are needed. Divide the objects into three piles, two of equal size and

the third as close as possible. Put the first two piles on the scale. If one is heavier, then it contains the heavier object, otherwise the third pile does. Recurse on this one pile.

# Conclusion

The overall goal of this entire course has been to teach skills in abstract thinking. I hope that it has been fruitful for you. Good luck at applying these skills to new problems that arise in other courses and in the workplace.



Figure 28.2: We say goodbye to our friends.