

Algorithms: Parallel and Sequential

Umut A. Acar and Guy E. Blelloch

February 2019

© 2019 Umut A. Acar and Guy E. Blelloch

All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the copyright holders (authors) except for the use of brief quotations in a book review.

Umut A. Acar
Carnegie Mellon University
Department of Computer Science GHC 9231
Pittsburgh PA 15213 USA

Guy E. Blelloch
Carnegie Mellon University
Department of Computer Science GHC 9211
Pittsburgh PA 15213 USA

Contents

1	Introduction	1
2	Parallelism	2
1	Parallel Hardware	2
2	Parallel Software	4
3	Work, Span, Parallel Time	5
3.1	Work and Span	5
3.2	Work Efficiency	7
3	Specification, Problem, and Implementation	8
1	Algorithm Specification	8
2	Data Structure Specification	9
3	Problem	9
4	Implementation	10
4	Genome Sequencing (An Example)	12
1	Genome Sequencing Problem	12
1.1	Background	12
1.2	Sequencing Methods	13

1.3	Genome Sequencing Problem	15
1.4	Understanding the Structure of the Problem	17
2	Algorithms for Genome Sequencing	18
2.1	Brute Force	18
2.2	Brute Force Reloaded	19
2.3	Shortest Superstrings by Algorithmic Reduction	21
2.4	Traveling Salesperson Problem	21
2.5	Reducing Shortest Superstrings to TSP	22
2.6	Greedy Algorithm	24
3	Concluding Remarks	27
 I Background		 29
5	Sets and Relations	30
1	Sets	30
2	Relations	32
6	Graph Theory	33
1	Basic Definitions	33
2	Weighted Graphs	36
3	Subgraphs	37
4	Connectivity	38
5	Graph Partition	39
6	Trees	40

CONTENTS	5
II A Language for Specifying Algorithms	42
7 Introduction	43
8 Functional Algorithms	45
1 Pure Functions	45
1.1 Safe for Parallelism	46
1.2 Persistence	48
1.3 Benign Effects	48
2 Functions as Values	49
3 Functional Algorithms	50
9 The Lambda Calculus	51
1 Syntax and Semantics	51
2 Parallelism and Reduction Order	52
10 The SPARC Language	54
1 Syntax and Semantics of SPARC	54
III Analysis of Algorithms	63
11 Introduction	64
12 Asymptotics	65
1 Basics	65
2 Big-O, big-Omega, and big-Theta	66
3 Some Conventions	69

13 Cost Models	71
1 Machine-Based Cost Models	71
1.1 RAM Model	71
1.2 PRAM: Parallel Random Access Machine	72
2 Language Based Models	73
2.1 The Work-Span Model	74
2.2 Scheduling	77
14 Recurrences	80
1 The Basics	80
2 Some conventions	81
3 The Tree Method	83
4 The Brick Method	84
5 Substitution Method	89
6 Master Method	91
IV Sequences	92
15 Introduction	93
1 Defining Sequences	94
16 The Sequence Abstract Data Type	97
1 The Abstract Data Type	97
2 Basic Functions	99
3 Tabulate	99
4 Map and Filter	100

CONTENTS	7
5 Subsequences	101
6 Append and Flatten	102
7 Update and Inject	102
8 Collect	104
9 Aggregation by Iteration	105
10 Aggregation by Reduction	107
11 Aggregation with Scan	109
17 Array Sequences	112
1 A Parametric Implementation	112
2 Implementing the Primitive Functions	115
18 Cost of Sequences	118
1 Cost Specifications	118
2 Array Sequences	119
3 Tree Sequences	125
4 List Sequences	126
19 Examples	128
1 Miscellaneous Examples	128
2 Computing Primes	132
20 Ephemeral and Single-Threaded Sequences	136
1 Persistent and Ephemeral Implementations	136
2 Ephemeral Sequences	137
3 Single-Threaded Sequences	138
3.1 Implementation	139

21 Tree Sequences	141
1 Primitive Tree Sequences	141
2 Parametric Implementation of Tree Sequences	143
V Algorithm Design And Analysis	146
22 Introduction	147
23 Basic Techniques	148
1 Algorithmic Reduction	148
2 Brute Force	149
24 Divide and Conquer	153
1 Divide and Conquer	153
2 Merge Sort	156
3 Sequence Scan	157
4 Euclidean Traveling Salesperson Problem	158
5 Divide and Conquer with Reduce	161
25 Contraction	162
1 Contraction Technique	162
2 Reduce with Contraction	163
3 Scan with Contraction	165
26 Maximum Contiguous Subsequence Sum	168
1 The Problem	168
2 Brute Force	170

CONTENTS	9
3 Applying Reduction	172
3.1 Auxiliary Problems	173
3.2 Reduction to MCSSS	175
3.3 Reduction to MCSSE	175
4 Divide And Conquer	178
4.1 A First Solution	178
4.2 Divide And Conquer with Strengthening	182
VI Probability	187
27 Introduction	188
28 Probability Spaces	189
1 Probability Spaces and Events	189
2 Properties of Probability Spaces	191
2.1 The Union Bound	191
2.2 Conditional Probability	192
2.3 Law of Total Probability	192
2.4 Independence	193
29 Random Variables	195
1 Probability Mass Function	196
2 Bernoulli, Binomial, and Geometric RVs	197
3 Functions of Random Variables	198
4 Conditioning	199
5 Independence	200

30 Expectation	201
1 Definitions	201
2 Composing Expectations	203
3 Linearity of Expectations	203
4 Conditional Expectation	205
5 Variance and Standard Deviation	205
6 Markov's Inequality	205
7 Chebyshev's Inequality	206
31 Introduction	207
1 Randomized Algorithms	207
1.1 Advantages of Randomization	208
1.2 Disadvantages of Randomization	210
2 Analysis of Randomized Algorithms	210
32 Order Statistics	212
1 The Order Statistics Problem	212
2 Randomized Algorithm for Order Statistics	212
3 Analysis	214
4 Intuitive Analysis	214
4.1 Complete Analysis	215
33 The Quick Sort Algorithm	220
1 Quicksort	220
2 Analysis of Quicksort	222
2.1 Intuition	223

CONTENTS	11
2.2 The Analysis	224
2.3 Alternative Analysis of Quicksort	229
3 Concluding Remarks	230
VII Binary Search Trees	231
34 Introduction	232
1 Motivation	232
2 Preliminaries	233
3 Searching a BST	236
4 Balancing BSTs	237
5 An Interface for Sets	238
35 Parametric BSTs	243
1 The Parametric Data Type	243
2 Algorithms based on joinMid	244
3 Parallel Functions	247
4 Cost Specification	249
4.1 Cost of Union, Intersection, and Difference	250
36 Treaps	256
1 Treap Properties	256
2 Height Analysis of Treaps	257
3 The Treap Data Structure	258
37 Augmenting Binary Search Trees	261
1 Augmenting with Values	261

2	Augmenting with Size	261
2.1	Example: Rank and Select in BSTs	263
3	Augmenting with Reduced Values	264
VIII Sets and Tables		267
38	Sets	268
1	Motivation	268
2	Sets ADT	269
3	Cost of Sets	272
39	Tables	275
1	Interface	275
2	Cost Specification for Tables	279
40	Ordering and Augmentation	280
1	Ordered Sets Interface	280
2	Cost specification: Ordered Sets	282
3	Interface: Augmented Ordered Tables	282
41	Example: Indexing and Searching	284
IX Graphs		288
42	Graphs and their Representation	289
1	Graphs and Relations	289
2	Applications of Graphs	290

CONTENTS	13	
3	Graphs Representations	292
3.1	Edge Sets	293
3.2	Adjacency Tables	294
3.3	Adjacency Sequences	295
3.4	Adjacency Matrices	297
3.5	Representing Weighted Graphs	299
43	Graph Search	301
1	Generic Graph Search	301
2	Reachability	303
3	Graph-Search Tree	304
4	Priority-First Search (PFS)	304
44	Breadth-First Search	306
1	BFS and Distances	306
2	Sequential BFS	307
2.1	Cost of Sequential BFS	308
3	Parallel BFS	310
3.1	Cost of Parallel BFS	311
4	Shortest Paths and Shortest-Path Trees	313
4.1	Cost with Sequences	316
45	Depth-First Search	318
1	DFS Reachability	318
2	DFS Trees	320
3	DFS Numbers	322

4	Cost of DFS	324
4.1	Parallel DFS	325
5	Cycle Detection	326
6	Topological Sort	327
7	Strongly Connected Components (SCC)	331
8	Discussions	335
46	Introduction	337
1	Path Weights	337
2	Shortest Path Problems	339
3	The Sub-Paths Property	339
47	Dijkstra's Algorithm	342
1	Dijkstra's Property	342
2	Dijkstra's Algorithm with Priority Queues	345
3	Cost Analysis of Dijkstra's Algorithm	349
48	Bellman-Ford's Algorithm	351
1	Graphs with Negative Edge Weights	351
2	Bellman-Ford's Algorithm	353
3	Cost Analysis	357
49	Johnson's Algorithm	360
X	Graph Contraction and Applications	364
50	Introduction	365

CONTENTS	15
1 Preliminaries	365
2 Graph Contraction	367
51 Edge Contraction	370
1 Edge Partition	370
1.1 Analysis of Parallel Edge Partition	373
2 Edge Contraction	374
52 Star Contraction	376
1 Star Partition	376
1.1 Analysis of Star Partition	381
2 Star Contraction	382
53 Graph Connectivity	385
1 Preliminaries	385
2 Algorithms for Connectivity	386
XI Minimum Spanning Trees	390
54 Introduction	391
1 Spanning Trees	391
2 Minimum Spanning Trees	392
3 Light-Edge Property	394
4 Approximating Metric TSP via MST	395
55 Sequential MST Algorithms	399
1 Prim's Algorithm	399

2	Kruskal's Algorithm	401
56	Parallel MST Algorithms	403
1	Boruvka's Algorithm	403
1.1	Algorithm Idea	404
1.2	Boruvka's Algorithm with Tree Contraction	406
1.3	Boruvka's Algorithm with Star Contraction	407
XII	Dynamic Programming	411
57	Introduction	412
58	Two Problems	418
1	Subset Sums	418
2	Minimum Edit Distance	421
59	Optimal Binary Search Trees	426
60	Implementing Dynamic Programming	431
1	Bottom-Up Method	431
2	Top-Down Method: Memoization	434
XIII	Priority Queues	437
61	Priority Queues	438
1	Implementing Priority Queues	439
2	Meldable Priority Queues	441
2.1	Leftist Heaps	444

<i>CONTENTS</i>	17
XIV Hashing	448
62 Foundations	449
1 Introduction	449
2 Hash Functions	452
3 Universal Hashing	455
63 Hash Tables	459
1 Nested Tables	460
1.1 A Parametric Design	460
1.2 Separate Chaining	462
1.3 Perfect Hashing	463
2 Flat Tables or Open Addressing	466
2.1 A Parametric Implementation of Flat Tables	466
2.2 Linear Probing	469
2.3 Quadratic Probing	470
2.4 Double Hashing	471
3 Concluding Remarks	472
XV Concurrency	473
64 Threads, Concurrency, and Parallelism	474
1 Threads	474
2 Concurrency and Parallelism	475
3 Mutable State and Race Conditions	478
65 A Practical and Efficient Implementation of Sequences	481

1	Motivation	481
2	Tabulate and Granularity Control	483
3	Scan	484
4	Filter	487
66	Critical Sections and Mutual Exclusion	489
A	Computations as Graphs	494
1	Machine-Based Cost Models	494
1.1	RAM Model	494
1.2	PRAM: Parallel Random Access Machine	495
2	Computation Graphs	497
B	Scheduling: Lower and Upper Bounds	500
1	Scheduling	500
2	Upper Bounds for Fixed Number of Processes	502
3	Upper Bounds for Dynamic Environments	504
4	Concluding Remarks	505
C	Work Stealing Algorithm	506
1	Doubly Ended Queues	507
2	Work-Stealing Algorithm	508
3	Structural Lemma	513
D	Analysis of Work Stealing	517
1	Balls and Bins Game	517
2	Bound in terms of Work and Steal Attempts	519

3 Bounding the Number of Steal Attempts	519
---	-----

Chapter 1

Introduction

This book aims to present techniques for problem solving using today's computers, including both sequentially and in parallel. For example, you might want to find the shortest path from where you are now to the nearest café by using your computer. The primary concerns will likely include correctness (the path found indeed should end at the nearest café), efficiency (that your computer consumed a relatively small amount of energy), and performance (the answer was computed reasonable quickly).

This book covers different aspects of problem solving with computers such as

- defining precisely the problem you want to solve,
- learning the different algorithm-design techniques for solving problems,
- designing abstract data types and the data structures that implement them, and
- analyzing and comparing the cost of algorithms and data structures.

Remark. We are concerned both with parallel algorithms (algorithms that can perform multiple actions at the same time) and sequential algorithms (algorithms that perform a single action at a time). In our approach, however, sequential and parallel algorithms are not particularly different.

Chapter 2

Parallelism

The term “parallelism” or “parallel computing” refers to the ability to run multiple computations (tasks) at the same time. Today parallelism is available in all computer systems, and at many different scales starting with parallelism in the nano-circuits that implement individual instructions, and working the way up to parallel systems that occupy large data centers.

1 Parallel Hardware

Multicore Chips. Since the early 2000s hardware manufacturers have been placing multiple processing units, often called “cores”, onto a single chip. These cores can be general purpose processors, or more special purpose processors, such as those found in *Graphics Processing Units* (GPUs). Each core can run in parallel with the others. Today (in year 2018), multicore chips are used in essentially all computing devices ranging from mobile phones to desktop computers and servers.

Large-Scale Parallelism. At the larger scale, many computers can be connected by a network and used together to solve large problems. For example, when you perform a simple search on the Internet, you engage a data center with thousands of computers in some part of the world, likely near your geographic location. Many of these computers (perhaps as many as hundreds, if not thousands) take up your query and sift through data to give you an accurate response as quickly as possible. Because each computer can itself be parallel (e.g., built with multicore chips), the scale of parallelism can be quite large, e.g., in the thousands.

Fundamental Reasons for Why Parallelism Matters. There are several reasons for why parallelism has become prevalent over the past decade.

First, parallel computing is simply faster than sequential computing. This is important, because many tasks must be completed quickly to be of use. For example, to be useful, an Internet search should complete in “interactive speeds” (usually below 100 milliseconds). Similarly, a weather-forecast simulation is essentially useless if it cannot be completed in time.

The second reason is efficiency in terms of energy usage. Due to basic physics, performing a computation twice as fast sequentially requires eight times as much energy (energy consumption is a cubic function of clock frequency). With parallelism we don’t need to use more energy than sequential computation, because energy is determined by the total amount of computation (work).

These two factors—time and energy—have become increasingly important in the last decade.

Example 2.1. Using two parallel computers, we can perform a computation in half the time of a sequential computer (operating at the same speed). To this end, we need to divide the computation into two parallel sub-computations, perform them in parallel and combine their results. This can require as little as half the time as the sequential computation. Because the total computation that we must do remains the same in both sequential and parallel cases, the total energy consumed is also the same.

The above reasoning holds in theory. In practice, there are overheads to parallelism: the speedup will be less than two-fold and more energy will be needed. For example, dividing the computation and combining the results could lead to additional overhead. Such overhead usually diminishes as the degree of parallelism increases but not always.

Example 2.2. As is historically popular in explaining algorithms, we can establish an analogy between parallel algorithms and cooking. As in a kitchen with multiple cooks, in parallel algorithms you can do things in parallel for faster turnaround time. For example, if you want to prepare 3 dishes with a team of cooks you can do so by asking each cook to prepare one. Doing so will often be faster than using one cook. But there are some overheads, for example, the work has to be divided as evenly as possible. Obviously, you also need more resources, e.g., each cook might need their own cooking pan.

Example 2.3 (Comparison to Sequential). One way to quantify the advantages of parallelism is to compare its performance to sequential computation. The table below illustrates the sort of performance improvements that can be achieved today. These timings are taken on a 32 core commodity server machine. In the table, the sequential timings use sequential algorithms while the parallel timings use parallel algorithms. Notice that the *speedup* for the parallel 32 core version relative to the sequential algorithm ranges from approximately 12 (minimum spanning tree) to approximately 32 (sorting).

Application	Sequential	Parallel $P = 1$	Parallel $P = 32$
Sort 10^7 strings	2.9	2.9	.095
Remove duplicates for 10^7 strings	.66	1.0	.038
Minimum spanning tree for 10^7 edges	1.6	2.5	.14
Breadth first search for 10^7 edges	.82	1.2	.046

2 Parallel Software

Challenges of Parallel Software. It would be convenient to use sequential algorithms on parallel computers, but this does not work well because parallel computing requires a different way of organizing the computation. The fundamental difference is that in parallel algorithms, computations must actually be *independent* to be performed in parallel. By independent we mean that computations do not depend on each other. Thus when designing a parallel algorithm, we have to identify the underlying dependencies in the computation and avoid creating unnecessary dependencies. This design challenge is an important focus of this book.

Example 2.4. Going back to our cooking example, suppose that we want to make a frittata in our kitchen with 4 cooks. Making a frittata is not easy. It involves cleaning and chopping vegetables, beating eggs, sauteeing, as well as baking. For the frittata to be good, the cooks must follow a specific recipe and pay attention to the dependencies between various tasks. For example, vegetables cannot be sauteed before they are washed, and the eggs cannot be fished before they are broken!

Coding Parallel Algorithms. Another important challenge concerns the implementation and use of parallel algorithms in the real world. The many forms of parallelism, ranging from small to large scale, and from general to special purpose, have led to many different programming languages and systems for coding parallel algorithms. These different programming languages and systems often target a particular kind of hardware, and even a particular kind of problem domain. As it turns out, one can easily spend weeks or even months optimizing a parallel sorting algorithm on specific parallel hardware, such as a multicore chip, a GPU, or a large-scale massively parallel distributed system.

Maximizing speedup by coding and optimizing an algorithm is not the goal of this book. Instead, our goal is to cover general design principles for parallel algorithms that can be applied in essentially all parallel systems, from the data center to the multicore chips on mobile phones. We will learn to think about parallelism at a high-level, learning general techniques for designing parallel algorithms and data structures, and learning how to approximately analyze their costs. The focus is on understanding when things can run in parallel, and when not due to dependencies. There is much more to learn about parallelism, and we hope you continue studying this subject.

Example 2.5. There are separate systems for coding parallel numerical algorithms on shared memory hardware, for coding graphics algorithms on Graphical Processing Units (GPUs),

and for coding data-analytics software on a distributed system. Each such system tends to have its own programming interface, its own cost model, and its own optimizations, making it practically impossible to take a parallel algorithm and code it once and for all possible applications. Indeed, it can require a significant effort to implement even a simple algorithm and optimize it to run well on a particular parallel system.

3 Work, Span, Parallel Time

This section describes the two measures—work and span—that we use to analyze algorithms. Together these measures capture both the sequential time and the parallelism available in an algorithm. We typically analyze both of these asymptotically, using for example the big-O notation.

3.1 Work and Span

Work. The *work* of an algorithm corresponds to the total number of primitive operations performed by an algorithm. If running on a sequential machine, it corresponds to the sequential time. On a parallel machine, however, work can be divided among multiple processors and thus does not necessarily correspond to time.

The interesting question is to what extent can the work be divided and performed in parallel. Ideally we would like to divide the work evenly. If we had W work and P processors to work on it in parallel, then even division would give each processor $\frac{W}{P}$ fraction of the work, and hence the total time would be $\frac{W}{P}$. An algorithm that achieves such ideal division is said to have *perfect speedup*. Perfect speedup, however, is not always possible.

Example 2.6. A fully sequential algorithm, where each operation depends on prior operations leaves no room for parallelism. We can only take advantage of one processor and the time would not be improved at all by adding more.

More generally, when executing an algorithm in parallel, we cannot break dependencies, if a task depends on another task, we have to complete them in order.

Span. The second measure, *span*, enables analyzing to what extent the work of an algorithm can be divided among processors. The *span* of an algorithm basically corresponds to the longest sequence of dependences in the computation. It can be thought of as the time an algorithm would take if we had an unlimited number of processors on an ideal machine.

Definition 2.1 (Work and Span). We calculate the work and span of algorithms in a very simple way that just involves composing costs across subcomputations. Basically we assume that sub-computations are either composed sequentially (one must be performed

after the other) or in parallel (they can be performed at the same time). We then calculate the work as the sum of the work of the subcomputations. For span, we differentiate between sequential and parallel composition: we calculate span as the sum of the span of sequential subcomputations or maximum of the span of the parallel subcomputations. More concretely, given two subcomputations with work W_1 and W_2 and span S_1 and S_2 , we can calculate the work and the span of their sequential and parallel composition as follows. In calculating the overall work and span, the unit cost 1 accounts for the cost of (parallel or sequential) composition.

	W (Work)	S (span)
Sequential composition	$1 + W_1 + W_2$	$1 + S_1 + S_2$
Parallel composition	$1 + W_1 + W_2$	$1 + \max(S_1, S_2)$

Note. The intuition behind the definition of work and span is that work simply adds, whether we perform computations sequentially or in parallel. The span, however, only depends on the span of the maximum of the two parallel computations. It might help to think of work as the total energy consumed by a computation and span as the minimum possible time that the computation requires. Regardless of whether computations are performed serially or in parallel, energy is equally required; time, however, is determined only by the slowest computation.

Example 2.7. Suppose that we have 30 eggs to cook using 3 cooks. Whether all 3 cooks to do the cooking or just one, the total work remains unchanged: 30 eggs need to be cooked. Assuming that cooking an egg takes 5 minutes, the total work therefore is 150 minutes. The span of this job corresponds to the longest sequence of dependences that we must follow. Since we can, in principle, cook all the eggs at the same time, span is 5 minutes.

Given that we have 3 cooks, how much time do we actually need? It should be clear that each cook can cook 10 eggs, for a total time of 50 minutes. Later we will discuss the “the greedy scheduling principle” which tells us that given a task with W work and S span, and using a greedy schedule, the time is upper bounded by $W/P + S$. In our case this would be $150/3 + 5 = 55$.

Example 2.8 (Parallel Merge Sort). As an example, consider the parallel `mergeSort` algorithm for sorting a sequence of length n . The work is the same as the sequential time, which you might know is

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

We will see that the span for `mergeSort` is

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

Thus, when sorting a million keys and ignoring constant factors, work is $10^6 \lg(10^6) > 10^7$, and span is $\lg^2(10^6) < 500$.

Parallel Time. Even though work and span, are abstract measures of real costs, they can be used to predict the run-time on any number of processors. Specifically, if for an algorithm the work dominates, i.e., is much larger than, span, then we expect the algorithm to deliver good speedups.

Exercise 2.1. How would you expect the parallel mergesort algorithm, `mergeSort`, mentioned in the example above to perform as we increase the number of processors dedicated to running it?

Solution. Recall that the work of parallel merge sort is $O(n \lg n)$, whereas the span is $O(\lg^2 n)$. Since span is much smaller than the work, we would expect to get good (close to perfect) speedups when using a small to moderate number of processors, e.g., couple of tens or hundreds, because the work term will dominate. We would expect for example the running time to halve when we double the number of processors. We should note that in practice, speedups tend to be more conservative due to natural overheads of parallel execution and due to other factors such as the memory subsystem that can limit parallelism.

3.2 Work Efficiency

If algorithm A has less work than algorithm B , but has greater span then which algorithm is better? In analyzing sequential algorithms there is only one measure so it is clear when one algorithm is asymptotically better than another, but now we have two measures. In general the work is more important than the span. This is because the work reflects the total cost of the computation (the processor-time product). Therefore typically the goal is to first reduce the work and then reduce the span by designing asymptotically work-efficient algorithms that perform no work than the best sequential algorithm for the same problem. However, sometimes it is worth giving up a little in work to gain a large improvement in span.

Definition 2.2 (Work Efficiency). We say that a parallel algorithm is *(asymptotically) work efficient*, if the work is asymptotically the same as the time for an optimal sequential algorithm that solves the same problem.

Example 2.9. The parallel `mergeSort` function described in is work efficient since it does $O(n \log n)$ work, which optimal time for comparison based sorting.

In this course we will try to develop work-efficient or close to work-efficient algorithms.

Chapter 3

Specification, Problem, and Implementation

This chapter reviews the basic concepts of specification, problem, and implementation.

Problem solving in computer science requires reasoning precisely about problems being studied and the properties of solutions. To facilitate such reasoning, we define problems by specifying them and describe the desired properties of solutions at different levels of abstraction, including the cost of the solution, and its implementation.

In this book, we are usually interested in two distinct classes of problems: algorithms or algorithmic problems and data structures problems.

1 Algorithm Specification

We specify an algorithm by describing what is expected of the algorithm via an *algorithm specification*. For example, we can specify a sorting algorithm for sequences with respect to a given comparison function as follows.

Definition 3.1 (Comparison Sort). Given a sequence A of n elements taken from a totally ordered set with comparison operator $<$, return a sequence B containing the same elements but such that $B[i] \leq B[j]$ for $0 \leq i < j < n$.

Note. The specification describes *what* the algorithm should do but it does not describe *how* it achieves what is asked. This is intentional—and is exactly the point—because there can be many algorithms that meet a specification.

A crucial property of any algorithm is its resource requirements or its *cost*. For example, of the many ways algorithms for sorting a sequence, we may prefer some over the others.

We specify the cost of class of algorithms with a *cost specification*. The following cost specification states that a particular class of parallel sorting algorithms performs $O(n \log n)$ work and $O(\log^2 n)$ span.

Cost Specification 3.2 (Comparison Sort: Efficient & Parallel). Assuming the comparison function $<$ does constant work, the cost for parallel comparison sorting a sequence of length n is $O(n \log n)$ work and $O(\log^2 n)$ span.

There can be many cost specifications for sorting. For example, if we are not interested in parallelism, we can specify the work to be $O(n \log n)$ but leave span unspecified. [The cost specification below](#) requires even smaller span but allows for more work. We usually care more about work and thus would prefer the first cost specification; there might, however, be cases where the second specification is preferable.

Cost Specification 3.3 (Comparison Sort: Inefficient but Parallel). Assuming the comparison function $<$ does constant work, the cost for parallel comparison sorting a sequence of length n is $O(n^2)$ work and $O(\log n)$ span.

2 Data Structure Specification

We specify a data structure by describing what is expected of the data structure via an *Abstract Data Type (ADT) specification*. As with algorithms, we usually give cost specifications to data structures. For example, we can specify a priority queue ADT and give it a cost specification.

Data Type 3.4 (Priority Queue). A priority queue consists of a priority queue type and supports three operations on values of this type. The operation `empty` returns an empty queue. The operation `insert` inserts a given value with a priority into the queue and returns the queue. The operation `removeMin` removes the value with the smallest priority from the queue and returns it.

Cost Specification 3.5 (Priority Queue: Basic). The work and span of a priority queue operations are as follows.

- `empty`: $O(1)$, $O(1)$.
- `insert`: $O(\log n)$, $O(\log n)$.
- `removeMin`: $O(\log n)$, $O(\log n)$.

3 Problem

A *problem* requires meeting an algorithm or an ADT specification and a corresponding cost specification. Since we allow specifying algorithms and data structures, we can distinguish between algorithms problems and data-structure problems.

Definition 3.6 (Algorithmic Problem). An *algorithmic problem* or an *algorithms problem* requires designing an algorithm that satisfies the given algorithm specification and cost specification if any.

Definition 3.7 (Data-Structures Problem). A *data-structures problem* requires meeting an ADT specification by designing a data structure that can support the desired operations with the required efficiency specified by the cost specification.

Note. The difference between an algorithmic problem and a data-structures problem is that the latter involves designing a data structure and a collection of algorithms, one for each operation, that operate on that data structure.

Remark. When we are considering a problem, it is usually clear from the context whether we are talking about an algorithm or a data structure. We therefore usually use the simpler terms *specification problem* to refer to the algorithm/ADT specification and the corresponding problem respectively.

4 Implementation

We can solve an algorithms or a data-structures problem by presenting an *implementation* or code written in some formal or semi-formal programming language. The term *algorithm* refers to an implementation that solves an algorithms problem and the term *data structure* to refer to an implementation that solves a data-structures problem.

Note. The distinction between algorithmic problems and algorithms is common in the literature but the distinction between abstract data types and data structures is less so.

Example 3.1 (Insertion Sort). In this book, we describe algorithms by using the pseudo-code notation based on SPARC, the language used in this book. For example, we can specify the classic insertion sort algorithm as follows.

```
insSort f s =
  if |s| = 0 then
    ⟨ ⟩
  else insert f s[0] (insSort f (s[1, ..., n - 1]))
```

In the algorithm, f is the comparison function and s is the input sequence. The algorithm uses a function ($\text{insert } f x s$) that takes the comparison function f , an element x , and a sequence s sorted by f , and inserts x in the appropriate place. Inserting into a sorted sequence is itself an algorithmic problem, since we are not specifying how it is implemented, but just specifying its functionality.

Example 3.2 (Cost of Insertion Sort). Considering insertion sort example, suppose that we are given a cost specification for insert : for a sequence of length n the cost of insert should be $O(n)$ work and $O(\log n)$ span. We can then determine the overall asymptotic cost of sort using our composition rules described in [Section \(Work, Span, and Parallel](#)

[Time](#)) . Since the code uses `insert` sequentially and since there are n inserts, the algorithm `insSort` has $n \times O(n) = O(n^2)$ work and $n \times O(\log n) = O(n \log n)$ span.

Example 3.3 (Priority Queues). We can give a data structure by specifying the data type used by the implementation, and the algorithms for each operation. For example, we can implement a priority queue with a binary heap data structure and describe each operation as an algorithm that operates on this data structure. In other words, a data structure can be viewed as a collection of algorithms that operate on the same organization of the data.

Remark (On the Importance of Specification). Distinguishing between specification and implementation is important due to several reasons.

First, specifications allow us to ignore the details of an implementation that we might not care to know. In many cases the specification of a problem is quite simple, but an efficient algorithm or implementation that solves it is complicated. Specifications allow us abstract from implementation details.

Second, specifications allow modularity. In computer science, it is important to improve implementations over time. As long as each implementation matches the same specification and the client only relies on the specification, then new implementations can be used without breaking things.

Third, when we compare the performance of different algorithms or data structures it is important that we do not compare apples and oranges. We have to make sure the compared objects solve the same problem—subtle differences in the problem specification can make a significant difference in how efficiently that problem can be solved.

Chapter 4

Genome Sequencing (An Example)

Sequencing of a complete human genome represents one of the greatest scientific achievements of the century. When the human genome project was first proposed in mid 1980's, the technology available could only sequence couple of hundred bases at a time. After a few decades, the efforts led to the several major landmark accomplishments. In 1996, the DNA of the first living species (fruit fly) was sequenced. This was followed, in 2001, by the draft sequence of the human genome. Finally in 2007, the full human genome diploid was sequenced. Efficient parallel algorithms played a crucial role in all these achievements. In this chapter, we review the genome-sequencing problem and discuss how the problem can be formulated as an algorithmic one.

This chapter presents an overview of the genome sequencing problem and how it may be abstracted and solved as a string problem in computer science.

1 Genome Sequencing Problem

1.1 Background

As with many "real world" applications, defining precisely the problem that models our application is interesting in itself. We therefore start with the vague and not well-defined problem of "sequencing the human genome" and convert it into a precise algorithmic problem. Our starting point is the "shotgun method" for genome sequencing, which was the method used for sequencing the human genome.

Definition 4.1 (Nucleotide). A nucleotide is the basic building block of nucleic acid polymers such as DNA and RNA. It is comprised of a number of components, which bind together to form the double-helix. The components include

- a *nitrogenous base*, one of Adenine, Cytosine, Guanine, Thymine (Uracil),
- a 5-carbon sugar molecule, and
- one or more phosphate groups.

We distinguish between four kinds of nucleotides based on their nitrogenous base and label them as 'A' (Adenine), 'C' (Cytosine), 'G' (Guanine), or 'T' (Thymine).

Definition 4.2 (Nucleic Acid). Nucleic acids are large molecules structured as linear chains of nucleotides. DNA and RNA are two important examples of nucleic acids.

Definition 4.3 (Human Genome). The *human genome* is the full nucleic acid sequence for humans consisting of nucleotide bases of A (Adenine), Cytosine (C), Guanine (G), or Thymine (T). It contains over 3 billion *base pairs*, each of which consist of two nucleotide bases bound by hydrogen bonds. It can be written as a sequence or a string of bases consisting of the letters, "A", "C", "G", and "T".

Remark. The human-genome sequence, if printed as a book, would be about as tall as the Washington Monument. The human genome is present in each cell of the human body. It appears to have all the information needed by the cells in our bodies and its deeper understanding will likely lead to insights into the nature of life.

1.2 Sequencing Methods

The challenge in sequencing the genome is that there is currently no way to read long strands with accuracy. Current DNA "reading" techniques are only capable of efficiently reading relatively short strands, e.g., 10-1000 base pairs, compared to the over three billion contained in the whole human genome. Scientists therefore cut strands into shorter fragments, sequence them, and then reassemble the sequenced fragments.

Primer Walking. A technique called *primer walking* can be used to sequence nucleic acid molecules up to 10,000 bases. The basic idea behind primer walking is to sequence a nucleic acid from each end using special molecules called *primers* that can be used to read the first 1000 bases or so. The process is then repeated to "walk" the rest of the nucleic acid by using most recently read part as a primer for the next part. Since the process requires constructing primers and since it is sequential, it is only effective for sequencing short molecules.

Fragments. To sequence a larger molecule, we can cut it into *fragments*, which can be achieved in a lab, and then use primer walking to sequence each fragment. Since each fragment can be sequenced independently in parallel, this would allow us to parallelize the hard part of sequencing. The problem though is that we don't know how to assemble them together, because the cutting process destroys the order of the fragments.

Note. The approach of dividing the genome and sequencing each piece independently is somewhat analogous to solving a jigsaw puzzle but without knowing the complete picture. It can be difficult and perhaps even impossible to solve such a puzzle.

Example 4.1. When cut, the strand `cattaggagtat` might turn into, `ag`, `gag`, `catt`, `tat`, destroying the original ordering.

The Shotgun Method. When we cut a genome into fragments we lose all the information about how the fragments should be assembled. If we had some additional information about how to assemble them, then we could imagine solving this problem. One way to get additional information on assembling the fragments is to make multiple copies of the original sequence and generate many fragments that overlap. Overlaps between fragments can then help relate and join them. This is the idea behind the shotgun (sequencing) method, which was the primary method used in sequencing the human genome for the first time.

Example 4.2. For the sequence `cattaggagtat`, we produce three copies:

```
cattaggagtat
cattaggagtat
cattaggagtat
```

We then divide each into fragments

```
catt — ag — gagtat
cat — tagg — ag — tat
ca — tta — gga — gtat
```

Note how each cut is “covered” by an overlapping fragment telling us how to patch together the cut.

Definition 4.4 (Shotgun Method). The *shotgun method* works as follows.

1. Take a DNA sequence and make multiple copies.
2. Randomly cut the sequences using a “shotgun” (in reality, using radiation or chemicals) into short fragments.
3. Sequence each fragments (possibly in parallel).
4. Reconstruct the original genome from the fragments.

Remark. Steps 1–3 of the Shotgun Method are done in a wet lab, while step 4 is the algorithmically interesting component. Unfortunately it is not always possible to reconstruct the exact original genome in step 4. For example, we might get unlucky and cut all sequences in the same location. Even if we cut them in different locations, there can be many DNA sequences that lead to the same collection of fragments. A particularly challenging problem is repetition: there is no easy way to know if repeated fragments are actual repetitions in the sequence or if they are a product of the method itself.

1.3 Genome Sequencing Problem

This section formulates the genome sequencing problem as an algorithmic problem.

Recall that using the Shotgun Method, we can construct and sequence short fragments made from copies of the original genome. Our goal is to use algorithms to reconstruct the original genome sequence from the many fragments by somehow assembling back the sequenced fragments. But there are many ways that such fragments can be assembled and we have no idea how the original genome looked like. So what can we do? In some sense we want to come up with the “best solution” that we can, given the information that we have (the fragment sequences).

Basic Terminology on Strings. From an algorithmic perspective, we can treat a genome sequence just as a sequence made up of the four different characters representing nucleotides. To study the problem algorithmically, let’s review some basic terminology on strings.

Definition 4.5 (Superstring). A string r is a *superstring* of another string s if s occurs in r as a contiguous block, i.e., s is a substring of r .

Example 4.3.

- tagg is a superstring of tag and gg but not of tg.
- gtat is a superstring of gta and tat but not of tac.

Definition 4.6 (Substring, Prefix, Suffix). A string s is a *substring* of another string r , if s occurs in r as a contiguous block. A string s is a *prefix* of another string r , if s is a substring starting at the beginning of r . A string s is a *suffix* of another string r , if s is a substring ending at the end of r .

Example 4.4.

- ag is a substring of ggag, and is also a suffix.
- gga is a substring of ggag, and is also a prefix.
- ag is not a substring of attg.

Definition 4.7 (Kleene Operators). For any set Σ , its *Kleene star* Σ^* is the set of all possible strings consisting of characters Σ , including the empty string.

For any set Σ , its *Kleene plus* Σ^+ is the set of all possible strings consisting of characters Σ , excluding the empty string.

Example 4.5. Given $\Sigma = \{a, b\}$,

$$\begin{aligned} \Sigma^* = \{ & \text{',} \\ & 'a', 'b', \\ & 'aa', 'ab', 'ba', 'bb', \\ & 'aaa', 'aab', 'aba', 'abb', \\ & 'baa', 'bab', 'bba', 'bbb', \\ & \dots \\ \} \end{aligned}$$

and

$$\begin{aligned}\Sigma^+ = \{ & 'a', 'b', \\ & 'aa', 'ab', 'ba', 'bb', \\ & 'aaa', 'aab', 'aba', 'abb', \\ & 'baa', 'bab', 'bba', 'bbb', \\ & \dots\}\end{aligned}$$

Definition 4.8 (Shortest Superstring (SS) Problem). Given an alphabet set Σ and a set of finite-length strings $A \subseteq \Sigma^*$, return a shortest string r that contains every $x \in A$ as a substring of r .

Genome Sequencing as a String Problem. We can now try to understand the properties of the genome-sequencing problem.

Exercise 4.1 (Properties of the Solution). What is a property that the result sequence needs to have in relation to the fragments sequenced in the Shotgun Method?

Solution. Because the fragments all come from the original genome, the result should contain all of them. In other words, the result is a *superstring* of the fragments.

Exercise 4.2. There can be multiple superstrings for any given set of fragments. Which superstrings are more likely to be the actual genome?

Solution. One possibly good solution is the shortest superstring. Because the Shotgun Method starts by making copies of the original sequence, by insisting on the shortest superstring, we would make sure that the duplicates would be eliminated. More specifically, if the original sequence had no duplicated fragments, then this approach would eliminate all the duplicates created by the copies that we made in the beginning.

We can abstract the genome-sequencing problem as an instance of the shortest superstring problem where $\Sigma = \{a, c, g, t\}$. We have thus converted a vague problem, sequencing the genome, into a concrete mathematical problem, the SS problem.

Remark. One might wonder why the shortest superstring is the “right answer” given that there could be many superstrings. Selecting the shortest is an instance of what is referred to as Occam’s razor—i.e., that the simplest explanation tends to be the right one. As we shall discuss more later, the SS problem is not exactly the right abstraction for the application of sequencing the genome, because it ignores some important practical factors. One issue is that the genome can contain repeated sections that are much longer than the fragments. In this case the shortest is not the right answer, but there is not enough information in the fragments to determine the right answer even given infinite computational power. One needs to add input to fix this issue. Another issue is that there can be errors in reading the short strings. This could make it impossible to find the correct or even reasonable superstring. Instead one needs to consider approximate substrings. Fortunately, the basic approach described here can be generalized to deal with these issues.

1.4 Understanding the Structure of the Problem

Let's take a closer look at the problem to make some observations.

Observation 1. We can ignore a fragment that is a substring of another fragment, because it doesn't contribute to the solution. For example, if we have gagtat, ag, and gt, we can throw out ag and gt.

Definition 4.9 (Snippets). In genome sequencing problem, we refer to the fragments that are not substrings of others as *snippets*.

Observation 2: Ordering of the snippets. Because no snippet is a substring of another, in the result superstring, snippets cannot start at the same position. Furthermore, if one starts after another, it must finish after the other. This leads to our second observation: in any superstring, the start positions of the snippets is a strict (total) order, which is the same order as their finish positions.

This observation means that to compute the shortest superstring of the snippets, it suffices to consider the permutations of snippets and compute for each permutation the corresponding superstring.

Example 4.6. In our example, we had the following fragments.

```

catt — ag — gagtat
cat — tagg — ag — tat
ca — tta — gga — gtat

```

The snippets are now: $S = \{ \text{catt}, \text{gagtat}, \text{tagg}, \text{tta}, \text{gga} \}$

The other fragments $\{ \text{cat}, \text{ag}, \text{tat}, \text{ca}, \text{gtat} \}$ are all contained within the snippets.

Consider a superstring such as cattaggagtat. The starting points for the snippets are: 0 for catt, 2 for tta, 3 for tagg, 5 for gga, and 6 for gagtat.

Shortest Superstring of a Permutation. Our second observation says that a superstring corresponds to a permutation. This leads to the natural question of how to find the shortest superstring for a given permutation. The following theorem tells us how.

Theorem 4.1 (Shortest Superstring by Overlap Removal). Given any start ordering of the snippets s_1, s_2, \dots, s_n , removing the maximum overlap between each adjacent pair of snippets (s_i, s_{i+1}) gives the shortest superstring of the snippets for that start ordering.

Proof. The theorem can be proven by induction. The base case is true since it is clearly true for a single snippet. Inductively, we assume it is true for the first i snippets, i.e., that removing the maximum overlap between adjacent snippets among these i snippets yields the shortest superstring of s_1, \dots, s_i starting in that order. We refer to this superstring as r_i . We now prove that the theorem is true for i then it is true for $i + 1$. Consider adding the snippet s_{i+1} after r_i , we know that s_{i+1} does not fully overlap with the previous snippet (s_i) by the definition of snippets. Therefore when we add it on using the maximum overlap, the resulting string r_{i+1} will be r_i with some new characters added to the end. The string r_{i+1} is a superstring of s_0, \dots, s_{i+1} because it includes r_i , which by induction is a superstring of s_0, \dots, s_i , and because it includes s_{i+1} . It is also the shortest since r_i is the shortest for s_1, \dots, s_i and a shorter string would not include s_{i+1} , because we have already eliminated the maximum overlap between s_{i+1} and r_i . \square

Example 4.7. In our running example, consider the following start ordering

```
catt tta tagg gga gagtat
```

When the maximum overlaps are removed we obtain `cattagagat`, which is indeed the shortest superstring for the given start ordering. In this case, it is also the overall shortest superstring.

Definition 4.10 (Overlap). Define the function $overlap$ as a function that computes the maximum overlap between two ordered snippets, i.e.,

$$overlap(s_i, s_j)$$

denotes the **maximum overlap** for s_i followed by s_j .

For example, for `tagg` and `gga`, $overlap(\text{tagg}, \text{gga}) = 2$.

2 Algorithms for Genome Sequencing

Overview. In this chapter, we review some of the algorithms behind the sequencing of genome, and more specifically the shortest superstring (SS) problem. The algorithms are derived by using consider three algorithm-design techniques: brute-force, reduction, and greedy.

2.1 Brute Force

Designing algorithms may appear to be an intimidating task, because it may seem as though we would need brilliant ideas that come out of nowhere. In reality, we design algorithms by starting with simple ideas based on several well-known techniques and refine them until the desired result is reached. Perhaps the simplest algorithm-design technique (and usually the least effective) is brute-force—i.e., try all solutions and pick the best.

Brute Force Algorithm 1. As applied to the genome-sequencing problem, a brute-force technique involves trying all candidate superstrings of the fragments and selecting the shortest one. Concretely, we can consider all strings $x \in \Sigma^*$, and for each x check if every fragment is a substring. Although we won't describe how here, such a check can be performed efficiently. We then pick the shortest x that is indeed a superstring of all fragments.

The problem with this brute-force algorithm is that there are an infinite number of strings in Σ^* so we cannot check them all. But, we don't need to: we only need to consider strings up to the total length of the fragments m , since we can easily construct a superstring by concatenating all fragments. Since the length of such a string is m , the shortest superstring has length at most m .

Note. Unfortunately, there are still $|\Sigma|^m$ strings of length m ; this number is not infinite but still very large. For the sequencing the genome $\Sigma = 4$ and m is in the order of billions, giving something like $4^{1,000,000,000}$. There are only about $10^{80} \approx 4^{130}$ atoms in the universe so there is no feasible way we could apply the brute force method directly. In fact we can't even apply it to two strings each of length 100.

2.2 Brute Force Reloaded

We will now design better algorithms by applying the observations from the [previous section](#). Specifically, we now know that we can compute the shortest superstring by finding the permutation that gives us the shortest superstring, and removing overlaps. Let's first design an algorithm for removing overlaps.

Algorithm 4.11 (Finding Overlap). Consider two strings s and t in that order. To find the overlap between s and t as [defined before](#), we can use the brute force-technique: consider each suffix of s and check if it is a prefix of t and select the longest such match.

The work of this algorithm is $O(|s| \cdot |t|)$, i.e., proportional to the product of the lengths of the strings since for each suffix of s we compare one to one to each character of the prefix of t . The comparisons across all suffixes and all characters within a suffix can be done in parallel in constant span. We however need to check that all characters match within a suffix, and then find the longest suffix for which this is true. These can both be done with what is called a reduce operation, which "sums" across a sequence of elements using a binary associative operator (in our case logical-and, and maximum). If implemented using a tree sum, a reduce has span that is logarithmic in the input length. Reduce will be covered extensively in later chapters. The total span is thus $O(\lg |s| + \lg |t|) \subseteq O(\lg (|s| + |t|) + \lg (|s| + |t|)) = O(\lg (|s| + |t|))$.

Algorithm 4.12 (Finding the Shortest Permutation). Using the above algorithm for finding overlaps, we can compute the shortest superstring of a permutation by eliminating the overlaps between successive strings in the permutation. We can then find the permutation with the shortest superstring by considering each permutation and selecting the permutation with the shortest superstring. Since we can consider each permutation independently of the others, this algorithm reveals parallelism.

Algorithm 4.13 (Finding the Shortest Permutation (Improved)). We can improve our algorithm for finding the shortest permutation by using a technique known as *staging*. Notice

that our algorithm repeatedly computes the overlap between the same snippets, because the permutations all belong to the same set of snippets. Since we only remove the overlap between successive snippets in a permutation, there are only $O(n^2)$ pairs to consider. We can thus stage work of the algorithm more carefully:

- First, compute the overlaps between each pair of snippets and store them in a dictionary for quick lookup.
- Second, try all permutations and compute the shortest superstring by removing overlaps as defined by the dictionary.

Note. Staging technique is inherently sequential: it creates a sequential dependency between the different stages of work that must be done. If, however, the number of stages is small (in our example, we have two), then this is harmless.

Cost Analysis. Let's analyze the work and the span of our staged algorithm. For the analysis, let W_1 and S_1 be the work and span for the first phase of the algorithm, i.e., for calculating all pairs of overlaps in our set of input snippets $s = \{s_1, \dots, s_n\}$. Let $m = \sum_{x \in S} |x|$. Using our algorithm, $overlap(x, y)$ for finding the maximum overlap between two strings x and y , we have

$$\begin{aligned}
 W_1 &\leq \sum_{i=1}^n \sum_{j=1}^n W(\text{overlap}(s_i, s_j)) \\
 &= \sum_{i=1}^n \sum_{j=1}^n O(|s_i||s_j|) \\
 &\leq \sum_{i=1}^n \sum_{j=1}^n (c_1 + c_2 |s_i||s_j|) \\
 &= c_1 n^2 + c_2 \sum_{i=1}^n \sum_{j=1}^n (|s_i||s_j|) \\
 &= c_1 n^2 + c_2 \sum_{i=1}^n \left(|s_i| \sum_{j=1}^n |s_j| \right) \\
 &= c_1 n^2 + c_2 \sum_{i=1}^n (|s_i| m) \\
 &= c_1 n^2 + c_2 m \sum_{i=1}^n |s_i| \\
 &= c_1 n^2 + c_2 m^2 \\
 &\in O(m^2) \quad \text{since } m \geq n.
 \end{aligned}$$

Since all pairs can be considered in parallel, we have for span

$$\begin{aligned}
 S_1 &\leq \max_{i=1}^n \max_{j=1}^n S(\text{overlap}(s_i, s_j)) \\
 S_1 &\leq \max_{i=1}^n \max_{j=1}^n O(\lg(|s_i| + |s_j|)) \\
 &\in O(\lg m).
 \end{aligned}$$

We therefore conclude that the first stage of the algorithm requires $O(m^2)$ work and $O(\lg m)$ span.

Moving onto the second stage, we want to compute the shortest superstring for each permutation. Given a permutation, we know that all we have to do is remove the overlaps. Since there are n overlaps to be considered and summed, this requires $O(n)$ work, assuming that we can lookup the overlaps in constant time. Since we can lookup the overlaps in parallel, the span is constant for finding the overlaps and $O(\lg n)$ for summing them, again

using a reduce. Therefore the cost for handling each permutation is $O(n)$ work and $O(\lg n)$ span.

Unfortunately, there are $n!$ permutations to consider. In fact this is another form of brute-force in which we try all possible permutations instead of all possible superstrings. Even though we have designed reasonably efficient algorithms for computing the overlaps and the shortest superstring for each permutation, there are too many permutations for this algorithm to be efficient. For $n = 10$ strings the algorithm is probably feasible, which is better than our previous brute-force algorithm, which did not even work for $n = 2$. However for $n = 100$, we'll need to consider $100! \approx 10^{158}$ permutations, which is still more than the number of atoms in the universe. Thus, the algorithm is still not feasible if the number of snippets is more than a couple dozen.

Remark. The technique of staging used above is a key technique in algorithm design and engineering. The basic idea is to identify a computation that is repeated many times and pre-compute it, storing it in a data structure for fast retrieval. Later instances of that computation can then be recalled via lookup instead of re-computing every time it is needed.

2.3 Shortest Superstrings by Algorithmic Reduction

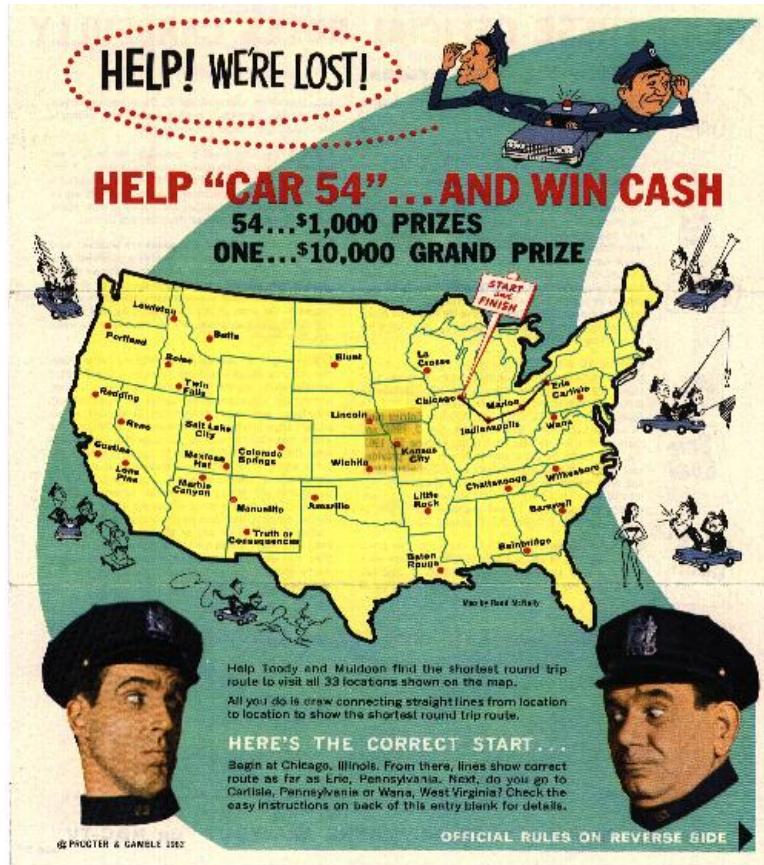
Another fundamental technique in algorithm design is to reduce one algorithms problem to another that has a known solution. It is sometimes quite surprising that we can reduce a problem to another, seemingly very different one. Here, we will reduce the shortest superstring problem to the Traveling Salesman Problem (TSP), which might appear to be quite different.

2.4 Traveling Salesperson Problem

The Traveling Salesperson Problem or TSP is a canonical problem dating back to the 1930s and has been extensively studied. The two major variants of the problem are *symmetric* TSP and *asymmetric* TSP, depending on whether the graph has undirected or directed edges, respectively. Both variants are known to be NP-hard, which indicates that it is unlikely that they have an algorithm that has polynomial work. One might ask why reduce the SS problem to a hard problem such as the TSP problem. Well it turns out that the SS problem is also NP-hard. The possible advantage of the reduction is that the TSP problem is such a well studied problem that there are many reasonably effective algorithms for the problem, even if not polynomial work.

Asymmetric TSP requires finding a Hamiltonian cycle of the graph such that the sum of the arc (directed edge) weights along the cycle is the minimum of all such cycles. A cycle is a path in a graph that starts and ends at the same vertex and a *Hamiltonian cycle* is a cycle that visits every vertex exactly once. In general graphs might not have a hamiltonian cycle, and in fact it is NP-hard to even check if a graph has such a cycle. However, in the TSP problem it is assumed that there is an edge between every pair of vertices (this is called a *complete graph*) and in this case a graph always has a hamiltonian cycle.

Example 4.8 (TSP Competition). A poster from a contest run by Proctor and Gamble in 1962 is reproduced below. The goal was to solve a 33 city instance of the TSP. Gerald Thompson, a Carnegie Mellon professor, was one of the winners.



Problem 4.3 (The Asymmetric Traveling Salesperson Problem (aTSP)). Given a weighted directed graph, find the shortest cycle that starts at some vertex and visits all vertices exactly once before returning to the starting vertex.

The symmetric version of the problem considers undirected graphs instead of directed ones—i.e., where the weight of an edge is the same in both directions.

Note. Note that the version of the problem that requires starting at a particular vertex is also NP-hard because otherwise we can solve the general problem by trying each vertex.

2.5 Reducing Shortest Superstrings to TSP

We can reduce the Shortest Superstring problem to TSP by using our second Observation, which we also used in the brute-force algorithm: the shortest superstring problem can be

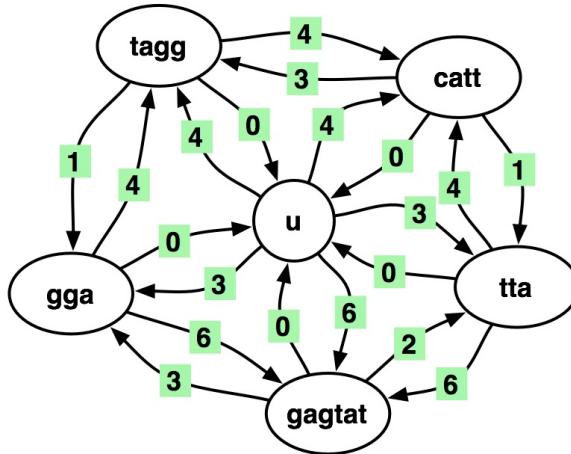
solved by trying all permutations. In particular we will make TSP try all the permutations for us.

Reduction from SS to TSP. For the reduction, we set up a graph so that each valid Hamiltonian cycle corresponds to a permutation.

We build a graph $D = (V, A)$. The reduction relies on the overlap function defined [defined earlier](#).

- The vertex set V has one vertex per snippet and a special “source” vertex u where the cycle starts and ends.
- The arc (directed edge) from s_i to s_j has weight $w_{i,j} = |s_j| - \text{overlap}(s_i, s_j)$. This quantity represents the increase in the string’s length if s_i is followed by s_j . For example, if we have tagg followed by gga, then we can generate tagga which only adds 1 character giving a weight of 1—indeed, $|\text{gga}| - \text{overlap}(\text{tagg}, \text{gga}) = 3 - 2 = 1$.
- The weights for arcs incident to source u are set as follows: $(u, s_i) = |s_i|$ and $(s_i, u) = 0$. That is, if s_i is the first string in the permutation, then the arc (u, s_i) pays for the whole length s_i . If s_i is the last string we have already paid for it, so the arc (s_i, u) is free.

Example 4.9. To see this reduction in action, the snippets in our running example, $\{\text{catt}, \text{gagtat}, \text{tagg}, \text{tta}, \text{gga}\}$ results in the graph, a subgraph of which is shown below (not all arcs are shown).



As intended, in this graph, a Hamiltonian cycle corresponds to a permutation in our second brute-force method: we start from the source and follow the cycle to produce the permutation. Furthermore, the sum of the arc weights in that cycle is equal to the length of the

superstring produced by the permutation. Since the TSP finds the minimum weight cycle, it also finds the permutation that leads to the shortest superstring. Therefore, if we could solve the TSP problem, we can solve the shortest superstring problem.

Summary. We have thus reduced the shortest-superstring problem, which is NP-hard, to another NP-hard problem: TSP. We constructed the reduction by using an insight from a brute-force algorithm: that we can solve the problem by trying out all permutations. The advantage of this reduction is that we know a lot about TSP, which can help, because for any algorithm that solves or approximates TSP, we obtain an algorithm for the shortest-superstring problem, and thus for sequencing the genome.

Remark (Hardness of Shortest Superstring). In addition to designing algorithms, reductions can be used to prove that a problem is NP-hard or NP-complete. For example, if we reduce an NP-hard (NP-complete) problem A to another problem B by performing polynomial work, and preserving the size within a polynomial factor, then B must be NP-hard (NP-complete).

For example, let's say we know the SS problem is NP hard. Since we are able to reduce it to the TSP problem in polynomial work and size using the method described above, this tells us the TSP must be NP-hard. If we wanted to go the other way and prove SS is NP-hard based on the known hardness of TSP then we would have to construct a reduction in the other direction—i.e., from the TSP to the SS. This is more challenging and we will leave it up to the most motivated students.

2.6 Greedy Algorithm

We have thus far developed a brute-force algorithm for solving the Shortest Superstring problem that requires exponential time and reduced the problem to the Traveling Salesperson Problem (TSP), which is NP-hard. We also remarked that the Shortest Superstring problem is NP-hard. Thus, we are still far away from a polynomial-work solution to the problem and we are unlikely to find one.

When a problem is NP hard, it means that there are *instances* of the problem that are difficult to solve exactly. NP-hardness doesn't rule out the possibility of algorithms that quickly compute approximate solutions or even algorithms that exactly solve many real world instances. For example the type-checking problem for strongly typed languages (e.g., the ML family of languages) is NP-hard but we use them all the time, even on large programs.

One interesting approach to overcoming difficult NP-hard problems is to use approximation. For the SS problem, we know efficient approximation algorithms that are theoretically good: they guarantee that the length of the superstring is within a constant factor of the optimal answer. Furthermore, these algorithms tend to perform even better in practice than the theoretical bounds suggest. In the rest of this unit, we discuss such an algorithm.

Greedy Algorithms. To design an approximation algorithm we will use an iterative design technique based on a *greedy heuristic*. When applying this design technique, we consider the current solution at hand and make a *greedy*, i.e., locally optimal decision to reduce the size of the problem. We then repeat the same process of making a locally optimal decision, hoping that eventually these decisions lead us to a global optimum. For example, a greedy algorithm for the TSP can visit the closest unvisited city (the locally optimal decision), removing thus one city from the problem.

Because greedy algorithms rely on a heuristic they may or may not return an optimal solution. Nevertheless, greedy algorithms are popular partly because they tend to be simple and partly because they can perform quite well. We note that, for a given problem there might be several greedy algorithms that depend on what is considered to be locally optimal.

The key step in designing a greedy algorithm is to decide the locally optimal decision. In the case of the SS problem, observe that we can minimize the length of the superstring by maximizing the overlap among the snippets. Thus, at each step of the algorithm, we can greedily pick a pair of snippets with the largest overlap and join them by placing one immediately after the other and removing the overlap. This can then be repeated until there is only one string left. This is the basic idea behind the greedy algorithm below.

Computing Joins. Let's define the function $join(x, y)$ as a function that places y after x and removes the maximum overlap.

For example,

$join(\text{tagg}, \text{gga}) = \text{tagga}$.

Algorithm 4.14 (Greedy Approximate SS).

```

greedyApproxSS (S) =
  if |S| = 1 then
    return x ∈ S
  else
    let
      T = {(overlap(x, y), x, y) : x ∈ S, y ∈ S, x ≠ y}
      (oxy, x, y) = argmax(o, x, y) ∈ T o
      z = join(x, y)
      S' = S ∪ {z} \ {x, y}
    in
      greedyApproxSS (S')
    end
  
```

The pseudocode for the greedy algorithm is shown above. Given a set of strings S , the $greedyApproxSS$ algorithm checks if the set has only 1 element, and if so returns that element. Otherwise it finds the pair of distinct strings x and y in S that have the maximum overlap. It does this by first calculating the overlap for all pairs and then picking the one of these that has the maximum overlap.

Note that T is a set of triples each corresponding to an overlap and the two strings that overlap. The notation

$$\text{argmax}_{(o, \dots) \in T^O}$$

is mathematical notation for selecting the element of T that maximizes the first element of the triple, which is the overlap.

After finding the pair (x, y) with the maximum overlap, the algorithm then replaces x and y with $z = \text{join}(x, y)$ in S to obtain the new set of snippets S' . The new set S' contains one element less than S . The algorithm recursively repeats this process on this new set of strings until there is only a single string left. It thus terminates after $|S|$ recursive calls.

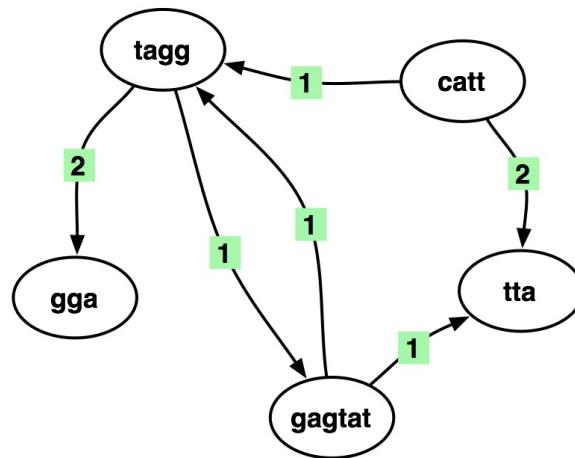
Exercise 4.4. Why is the algorithm greedy?

Solution. The algorithm is greedy because at every step it takes the pair of strings that when joined will remove the greatest overlap, a locally optimal decision. Upon termination, the algorithm returns a single string that contains all strings in the original set S . However, the superstring returned is not necessarily the shortest superstring.

Example 4.10. Consider the snippets in our running example,

$$\{ \text{catt}, \text{gagtat}, \text{tagg}, \text{tta}, \text{gga} \}.$$

The graph below illustrates the overlaps between different snippets. An arc from vertex u to v is labeled with the size of the overlap when u is followed by v . All arcs with weight 0 are omitted for simplicity.



Given these overlaps, the greedy algorithm could proceed as follows:

- join tagg and gga to obtain tagga (overlap = 2),
- join catt and tta to obtain catta (overlap = 2),

- join `gagtat` and `tagga` to obtain `gagtatagga` (overlap = 1), and
- join `gagtatagga` and `catta` to obtain `gagtataggacatta` (overlap = 0).

Note. Although the greedy algorithm merges pairs of strings one by one, we note there is still significant parallelism in the algorithm, at least as described. In particular we can calculate all the overlaps in parallel, and the largest overlap in parallel using a reduce.

Cost Analysis. From the analysis of our brute-force algorithm, we know that we can find the overlaps between the strings in $O(m^2)$ work and $O(\lg m)$ span. Thus T can be computed in $O(m^2)$ work and $O(\lg m)$ span. Finding the maximum with *argmax* can be computed by a simple reduce operation. Because $m > n$, the cost of computing T dominates. Therefore, excluding the recursive call, each call to *greedyApproxSS* costs is $O(m^2)$ work and $O(\lg m)$ span.

Observe now that each call to *greedyApproxSS* reduces the number of snippets: S' contains one fewer element than S , so there are at most n calls to *greedyApproxSS*. These calls are sequential because one call must complete before the next call can take place. Hence, the total cost for the algorithm is $O(nm^2)$ work and $O(n \lg m)$ span. The algorithm therefore has parallelism (work over span) of $O(nm^2/(n \lg m)) = O(m^2/\lg m)$ and is therefore highly parallel. There are ways to make the algorithm more efficient, but leave that as an exercise to the reader.

Approximation Quality. Since the *greedyApproxSS* algorithm does only polynomial work, and since the TSP problem is NP hard, we cannot expect it to give an exact answer on all inputs—that would imply $\mathbf{P} = \mathbf{NP}$, which is unlikely. Although *greedyApproxSS* does not return the shortest superstring, it returns a good approximation of the shortest superstring. In particular, it is known that it returns a string that is within a factor of 3.5 of the shortest; it is conjectured that the algorithm returns a string that is within a factor of 2. In practice, the greedy algorithm typically performs better than the bounds suggest. The algorithm also generalizes to other similar problems. Algorithms such as *greedyApproxSS* that solve an NP-hard problem to within a constant factor of optimal, are called *constant-factor approximation algorithms*.

3 Concluding Remarks

The *Human Genome Project* was an international scientific research project that was one of the largest collaborative projects in the human history. The project was formally launched in 1990, after several years of preparations, and pronounced complete in 2000. It costed approximately three billion dollars (in the currency of Fiscal Year 1991).

Abstracting a real-world problem such as the sequencing of the human genome, which is one of the most challenging problems that has been tackled in science, is significantly

more complex than the relatively simple abstractions that we use in this book. This section discusses some of the complexities of genome sequencing that we have not addressed.

Abstraction versus Reality. Often when abstracting a problem we can abstract away some key aspects of the underlying application that we want to solve. Indeed this is the case when using the Shortest Superstring (SS) problem for sequencing genomes. In actual genome sequencing there are two shortcomings with using the SS problem.

The first is that when reading the base pairs using a DNA sequencer there can be errors. This means the overlaps on the strings that are supposed to overlap perfectly might not. This can be dealt with by generalizing the Shortest Superstring problem to deal with approximate matching. Describing such a generalization is beyond the scope of this course, but basically one can give a score to every overlap and then pick the best one for each pair of fragments. The nice feature of this change is that the same algorithmic techniques we discussed for the SS problem still work for this generalization, only the “overlap” scores will be different.

The second shortcoming of using the SS problem by itself is that real genomes have long repeated sections, possibly much longer than the length of the fragments that are sequenced. The SS problem does not deal well with such repeats. In fact when the SS problem is applied to the fragments of an initial string with longer repeats than the fragment sizes, the repeats or parts of them are removed. One method that researchers have used to deal with this problem is the so-called *double-barrel shotgun method*. In this method strands of DNA are cut randomly into lengths that are long enough to span the repeated sections. After cutting it up one can read just the two ends of such a strand and also determine its length (approximately). By using the two ends and knowing how far apart they are it is possible to build a “scaffolding” and recognize repeats. This method can be used in conjunction with the generalization of the SS discussed in the previous paragraph. In particular the SS method allowing for errors can be used to construct from the snippets strings up to the length of the repeats, and the double barreled method can put these short strings together into longer strings, accounting for repeats.

Part I

Background

Chapter 5

Sets and Relations

This chapter presents a review of some basic definitions on sets and relations.

1 Sets

A *set* is a collection of distinct objects. The objects that are contained in a set, are called *members* or the *elements* of the set. The elements of a set must be distinct: a set may not contain the same element more than once. The set that contains no elements is called the *empty set* and is denoted by $\{\}$ or \emptyset .

Specification of Sets. Sets can be specified intentionally, by mathematically describing their members. For example, the set of natural numbers, traditionally written as \mathbb{N} , can be specified *intentionally* as the set of all nonnegative integral numbers. Sets can also be specified *extensionally* by listing their members. For example, the set $\mathbb{N} = \{0, 1, 2, \dots\}$. We say that an element x is a *member of* A , written $x \in A$, if x is in A . More generally, sets can be specified using *set comprehensions*, which offer a compact and precise way to define sets by mixing intentional and extensional notation.

Definition 5.1 (Union and Intersection). For two sets A and B , the *union* $A \cup B$ is defined as the set containing all the elements of A and B . Symmetrically, their *intersection*, $A \cap B$ is the defined as the set containing the elements that are member of both A and B . We say that A and B are *disjoint* if their intersection is the empty set, i.e., $A \cap B = \emptyset$.

Definition 5.2 (Cartesian Product). Consider two sets A and B . The *Cartesian product* $A \times B$ is the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$, i.e.,

$$A \times B = \{(a, b) : a \in A, b \in B\}.$$

Example 5.1. The Cartesian product of $A = \{0, 1, 2, 3\}$ and $B = \{a, b\}$ is

$$A \times B = \{ (0, a), (0, b), (1, a), (1, b), (2, a), (2, b), (3, a), (3, b) \}.$$

Definition 5.3 (Set Partition). Given a set A , a partition of A is a set P of non-empty subsets of A such that each element of A is in exactly one subset in P . We refer to each element of P as a *block* or a *part* and the set P as a *partition* of A . More precisely, P is a partition of A if the following conditions hold:

- if $B \in P$, then $B \neq \emptyset$,
- if $A = \bigcup_{B \in P} B$, and
- if $B, C \in P$, then $B = C$ or $B \cap C = \emptyset$.

Example 5.2. If $A = \{1, 2, 3, 4, 5, 6\}$ then $P = \{\{1, 3, 5\}, \{2, 4, 6\}\}$ is a partition of A . The set $\{1, 3, 5\}$ is a block.

The set $Q = \{\{1, 3, 5, 6\}, \{2, 4, 6\}\}$ is not a partition of A , because the element 6 is contained in multiple blocks.

Definition 5.4 (Kleene Operators). For any set Σ , its *Kleene star* Σ^* is the set of all possible strings consisting of members of Σ , including the empty string.

For any set Σ , its *Kleene plus* Σ^+ is the set of all possible strings consisting of members of Σ , excluding the empty string.

Example 5.3. Given $\Sigma = \{a, b\}$,

$$\begin{aligned} \Sigma^* = \{ & \text{ ' ',} \\ & 'a', 'b', \\ & 'aa', 'ab', 'ba', 'bb', \\ & 'aaa', 'aab', 'aba', 'abb', \\ & 'baa', 'bab', 'bba', 'bbb', \\ & \dots \\ \} \end{aligned}$$

and

$$\begin{aligned} \Sigma^+ = \{ & 'a', 'b', \\ & 'aa', 'ab', 'ba', 'bb', \\ & 'aaa', 'aab', 'aba', 'abb', \\ & 'baa', 'bab', 'bba', 'bbb', \\ & \dots \\ \} \end{aligned}$$

Exercise 5.1. Prove that Kleene star and Kleene plus are closed under string concatenation.

2 Relations

Definition 5.5 (Relation). A *(binary) relation from a set A to set B* is a subset of the Cartesian product of A and B. For a relation $R \subseteq A \times B$,

- the set $\{a : (a, b) \in R\}$ is referred to as the *domain* of R , and
- the set $\{b : (a, b) \in R\}$ is referred to as the *range* of R .

Definition 5.6 (Function). A *mapping* or *function from A to B* is a relation $R \subset A \times B$ such that $|R| = |\text{domain}(R)|$, i.e., for every a in the domain of R there is only one b such that $(a, b) \in R$. The *range* of the function is the range of R . We call the set A the *domain* and the B the *co-domain* of the function.

Example 5.4. Consider the sets $A = \{0, 1, 2, 3\}$ and $B = \{a, b\}$.

The set:

$$X = \{(0, a), (0, b), (1, b), (3, a)\}$$

is a relation from A to B since $X \subset A \times B$, but not a mapping (function) since 0 is repeated.

The set

$$Y = \{(0, a), (1, b), (3, a)\}$$

is both a relation and a function from A to B since each element only appears once on the left.

Chapter 6

Graph Theory

This chapter presents an overview of basic graph theory used throughout the book. This chapter does not aim to be complete (and is far from it) but tries to cover the most relevant material to the course. More details can be found in standard texts.

1 Basic Definitions

Definition 6.1 (Directed Graph). A *directed graph* or (*digraph*) is a pair $G = (V, A)$ where

- V is a set of *vertices*, and
- $A \subseteq V \times V$ is a set of *directed edges* or *arcs*.

Note. In a digraph, each arc is an ordered pair $e = (u, v)$. A digraph can have *self loops* (u, u) .

Definition 6.2 (Undirected graph). An *undirected graph* is a pair $G = (V, E)$ where

- V is a set of *vertices* (or nodes), and
- $E \subseteq \binom{V}{2}$ is a set of edges.

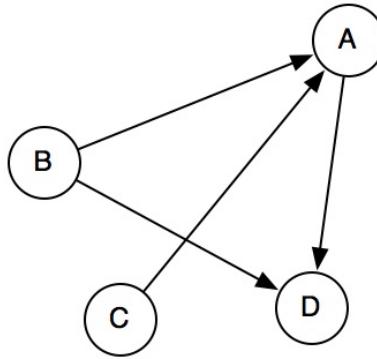
Note. Given a set V , a *k-combination* of V is a subset of k distinct elements of V . The notation

$$\binom{V}{k}$$

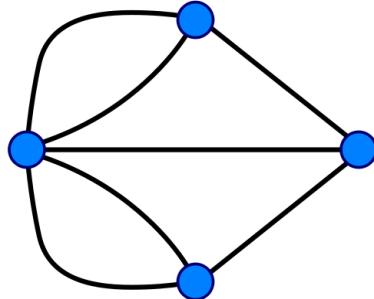
denotes the set of all k -combinations of the set V .

Therefore, in an undirected graph, each edge is an unordered pair $e = \{u, v\}$ (or equivalently $\{v, u\}$) and there cannot be self loops ($\{v, v\} = \{v\} \notin \binom{V}{2}$).

Example 6.1. An example directed graph with 4 vertices:



An undirected graph on 4 vertices, representing the Königsberg problem. (Picture Source: Wikipedia):



Remark. While directed graphs represent possibly asymmetric relationships, undirected graphs represent symmetric relationships. Directed graphs are therefore more general than undirected graphs because an undirected graph can be represented by a directed graph by replacing an edge with two arcs, one in each direction.

Graphs come with a lot of terminology, but fortunately most of it is intuitive once we understand the concept. In this section, we consider graphs that do not have any data associated with edges, such as weights. In the next section, we consider weighted graphs, where the weights on edges can represent a distance, a capacity or the strength of the connection.

Definition 6.3 (Neighbors). A vertex u is a *neighbor* of, or equivalently *adjacent* to, a vertex v in a graph $G = (V, E)$ if there is an edge $\{u, v\} \in E$. For a directed graph a vertex u is an *in-neighbor* of a vertex v if $(u, v) \in E$ and an *out-neighbor* if $(v, u) \in E$. We also say two edges or arcs are neighbors if they share a vertex.

Definition 6.4 (Neighborhood). For an undirected graph $G = (V, E)$, the *neighborhood* $N_G(v)$ of a vertex $v \in V$ is its set of all neighbors of v , i.e., $N_G(v) = \{u \mid \{u, v\} \in E\}$. For a directed

graph we use $N_G^+(v)$ to indicate the set of out-neighbors and $N_G^-(v)$ to indicate the set of in-neighbors of v . If we use $N_G(v)$ for a directed graph, we mean the out neighbors.

The neighborhood of a set of vertices $U \subseteq V$ is the union of their neighborhoods, e.g.,

- $N_G(U) = \bigcup_{u \in U} N_G(u)$, or
- $N_G^+(U) = \bigcup_{u \in U} N_G^+(u)$.

Definition 6.5 (Incidence). We say an edge is *incident* on a vertex if the vertex is one of its endpoints. Similarly we say a vertex is incident on an edge if it is one of the endpoints of the edge.

Definition 6.6 (Degree). The *degree* $d_G(v)$ of a vertex $v \in V$ in a graph $G = (V, E)$ is the size of the neighborhood $|N_G(v)|$. For directed graphs we use *in-degree* $d_G^-(v) = |N_G^-(v)|$ and *out-degree* $d_G^+(v) = |N_G^+(v)|$. We will drop the subscript G when it is clear from the context which graph we're talking about.

Definition 6.7 (Path). A *path* in a graph is a sequence of adjacent vertices. More formally for a graph $G = (V, E)$, we define the set of all paths in G , written $Paths(G)$ as

$$Paths(G) = \{P \in V^+ \mid 1 \leq i < |P|, (P_i, P_{i+1}) \in E\},$$

where V^+ indicates all positive length sequences of vertices (allowing for repeats). The length of a path is one less than the number of vertices in the path—i.e., it is the number of edges in the path. A path in a finite graph can have infinite length. A *simple path* is a path with no repeated vertices. Please see the remark below, however.

Remark. Some authors use the terms walk for path, and path for simple path. Even in this book when it is clear from the context we will sometimes drop the “simple” from simple path.

Definition 6.8 (Reachability and connectivity). In a graph G , a vertex u can *reach* another vertex v if there is a path in G that starts at u and ends at v . If u can reach v , we also say that v is *reachable* from u . We use $R_G(u)$ to indicate the set of all vertices reachable from u in G . Note that in an undirected graph reachability is symmetric: if u can reach v , then v can reach u .

We say that an undirected graph is *connected* if all vertices are reachable from all other vertices. We say that a directed graph is *strongly connected* if all vertices are reachable from all other vertices.

Definition 6.9 (Cycles). In a directed graph a *cycle* is a path that starts and ends at the same vertex. A cycle can have length one (i.e. a *self loop*). A *simple cycle* is a cycle that has no repeated vertices other than the start and end vertices being the same. In an undirected graph a simple *cycle* is a path that starts and ends at the same vertex, has no repeated vertices other than the first and last, and has length at least three. In this course we will exclusively talk about simple cycles and hence, as with paths, we will often drop simple.

Exercise 6.1. Why is it important in an undirected graph to require that a cycle has length at least three? Why is it important that we do not allow repeated vertices?

Definition 6.10 (Trees and forests). An undirected graph with no cycles is a *forest*. A forest that is connected is a *tree*. A directed graph is a forest (or tree) if when all edges are converted to undirected edges it is undirected forest (or tree). A *rooted tree* is a tree with one vertex designated as the root. For a directed graph the edges are typically all directed toward the root or away from the root.

Definition 6.11 (Directed acyclic graphs). A directed graph with no cycles is a *directed acyclic graph* (DAG).

Definition 6.12 (Distance). The *distance* $\delta_G(u, v)$ from a vertex u to a vertex v in a graph G is the shortest path (minimum number of edges) from u to v . It is also referred to as the *shortest path length* from u to v .

Definition 6.13 (Diameter). The *diameter* of a graph G is the maximum shortest path length over all pairs of vertices in G , i.e., $\max\{\delta_G(u, v) : u, v \in V\}$.

Definition 6.14 (Multigraphs). Sometimes graphs allow multiple edges between the same pair of vertices, called *multi-edges*. Graphs with multi-edges are called *multi-graphs*. We will allow multi-edges in a couple algorithms just for convenience.

Definition 6.15 (Sparse and Dense Graphs). We will use the following conventions:

$$\begin{aligned} n &= |V| \\ m &= |E| \end{aligned}$$

Note that a directed graph can have at most n^2 edges (including self loops) and an undirected graph at most $n(n - 1)/2$. We informally say that a graph is *sparse* if $m \ll n^2$ and *dense* otherwise. In most applications graphs are very sparse, often with only a handful of neighbors per vertex when averaged across vertices, although some vertices could have high degree. Therefore, the emphasis in the design of graph algorithms, at least for this book, is on algorithms that work well for sparse graphs.

Definition 6.16 (Enumerable graphs). In some cases, it is possible to label the vertices of the graphs with natural numbers starting from 0. More precisely, an *enumerable graph* is a graph $G = (V, E)$ where $V = \{0, 1, \dots, n - 1\}$. Enumerable graphs can be more efficient to represent than general graphs.

2 Weighted Graphs

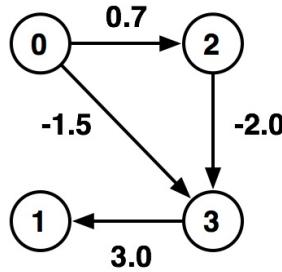
Many applications of graphs require associating weights or other values with the edges of a graph.

Definition 6.17 (Weighted and Edge-Labeled Graphs). An *edge-labeled graph* or a *weighted graph* is a triple $G = (E, V, w)$ where $w : E \rightarrow L$ is a function mapping edges or directed edges to their labels (weights), and L is the set of possible labels (weights).

In a graph, if the data associated with the edges are real numbers, we often use the term “weight” to refer to the edge labels, and use the term “weighted graph” to refer to the

graph. In the general case, we use the terms “edge label” and edge-labeled graph. Weights or other values on edges could represent many things, such as a distance, or a capacity, or the strength of a relationship.

Example 6.2. An example directed weighted graph.



Remark. As it may be expected, basic terminology on graphs defined above straightforwardly extend to weighted graphs.

3 Subgraphs

When working with graphs, we sometimes wish to refer to parts of a graph. To this end, we can use the notion of a subgraph, which refers to a graph contained in a larger graph. A subgraph can be defined as any subsets of edges and vertices that together constitute a well defined graph.

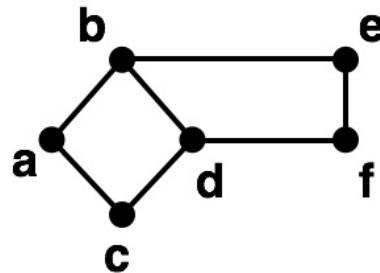
Definition 6.18 (Subgraph). Let $G = (V, E)$ and $H = (V', E')$ be two graphs. H is a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$.

Note. Note that since H is a graph, the vertices defining each edge are in the vertex set V' , i.e., for an undirected graph $E' \subseteq \binom{V'}{2}$. There are many possible subgraphs of a graph.

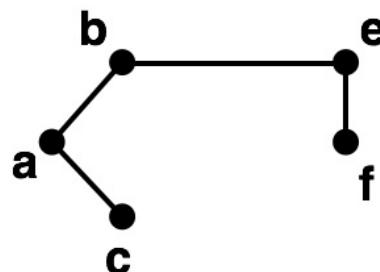
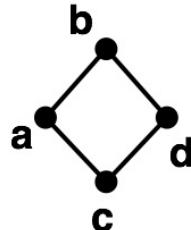
An important class of subgraphs are *vertex-induced subgraphs*, which are maximal subgraphs defined by a set of vertices. A vertex-induced subgraph is maximal in the sense that it contains all the edges that it can possibly contain. In general when an object is said to be a *maximal* “X”, it means that nothing more can be added to the object without violating the property “X”.

Definition 6.19 (Vertex-Induced Subgraph). The subgraph of $G = (V, E)$ induced by $V' \subseteq V$ is the graph $H = (V', E')$ where $E' = \{\{u, v\} \in E \mid u \in V', v \in V'\}$.

Example 6.3. Some vertex induced subgraphs:



Original graph

Induced by $\{a, b, c, e, f\}$ Induced by $\{a, b, c, d\}$

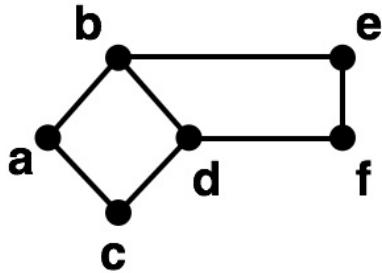
Although we will not use it in this book, it is also possible to define an induced subgraph in terms of a set of edges by including in the graph all the vertices incident on the edges.

Definition 6.20 (Edge-Induced Subgraph). The subgraph of $G = (V, E)$ induced by $E' \subseteq E$ is a graph $H = (V', E')$ where $V' = \cup_{e \in E'} e$.

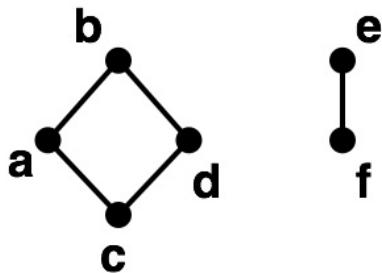
4 Connectivity

Recall that in a graph (either directed or undirected) a vertex v is reachable from a vertex u if there is a path from u to v . Also recall that an undirected graph is connected if all vertices are reachable from all other vertices.

Example 6.4. Two example graphs shown. The first is connected; the second is not.



Graph 1



Graph 2

An important subgraph of an undirected graph is a connected component of a graph, defined below.

Definition 6.21 (Connected Component). Let $G = (V, E)$ be an undirected graph. A subgraph H of G is a *connected component* of G if it is a maximally connected subgraph of G .

Here, “maximally connected component” means we cannot add any more vertices and edges from G to H without disconnecting H . In the graphs shown in the example above, the first graph has one connected component (hence it is connected); the second has two connected components.

Note. We can specify a connected component of a graph by simply specifying the vertices in the component. For example, the connected components of the second graph in the example above can be specified as $\{a, b, c, d\}$ and $\{e, f\}$.

5 Graph Partition

Recall that a partition of a set A is a set P of non-empty subsets of A such that each element of A is in exactly one subset, also called block, $B \in P$.

Definition 6.22 (Graph Partition). A *graph partition* is a partition of the vertex set of the graph. More precisely, given graph $G = (V, E)$, we define a partition of G as a set of graphs

$$P = \{G_1 = (V_1, E_1) \dots G_k = (V_k, E_k)\},$$

where $\{V_1, \dots, V_k\}$ is a (set) partition of V and G_1, \dots, G_k are vertex-induced subgraphs of G with respect to V_1, \dots, V_k respectively. As in set partitions, we use the term *part* or *block* to refer to each vertex-induced subgraph G_1, \dots, G_k .

Definition 6.23 (Internal and Cut Edges). In a graph partition, we can distinguish between two kinds of edges: internal edges and cut edges. *Internal edges* are edges that are within a block; *cut edges* are edges that are between blocks. One way to partition a graph is to make each connected component a block. In such a partition, there are no cut edges between the partitions.

6 Trees

Definition 6.24 (Tree). An undirected graph is a *tree* if it does not have cycles and it is connected. A *rooted tree* is a tree with a distinguished root node that can be used to access all other nodes. An example of a rooted tree along with the associated terminology is given in below.

Definition 6.25 (Rooted Trees). A *rooted tree* is a directed graph such that

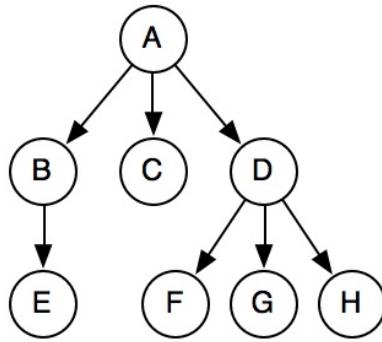
1. One of the vertices is the *root* and it has no in edges.
2. All other vertices have one in-edge.
3. There is a path from the root to all other vertices.

Rooted trees are common structures in computing and have their own dedicated terminology.

- By convention we use the term *node* instead of vertex to refer to the vertices of a rooted tree.
- A node is a *leaf* if it has no out edges, and an *internal node* otherwise.
- For each directed edge (u, v) , u is the *parent* of v , and v is a *child* of u .
- For each path from u to v (including the empty path with $u = v$), u is an *ancestor* of v , and v is a *descendant* of u .
- For a vertex v , its *depth* is the length of the path from the root to v and its *height* is the longest path from v to any leaf.
- The *height of a tree* is the height of its root.
- For any node v in a tree, the *subtree rooted at v* is the rooted tree defined by taking the induced subgraph of all vertices reachable from v (i.e. the vertices and the directed edges between them), and making v the root.

- As with graphs, an *ordered rooted tree* is a rooted tree in which the out edges (children) of each node are ordered.

Example 6.5. An example rooted tree follows.



root	:	<i>A</i>
leaves	:	<i>E, C, F, G, and H</i>
internal nodes	:	<i>A, B, and D</i>
children of <i>A</i>	:	<i>B, C and D</i>
parent of <i>E</i>	:	<i>B</i>
descendants of <i>A</i>	:	all nodes, including <i>A</i> itself
ancestors of <i>F</i>	:	<i>F, D and A</i>
depth of <i>F</i>	:	2
height of <i>B</i>	:	1
height of the tree	:	2
subtree rooted at <i>D</i>	:	the rooted tree consisting of <i>D, F, G and H</i>

Part II

A Language for Specifying Algorithms

Chapter 7

Introduction

In this book we define algorithms and data structures using nested parallelism in conjunction with a functional programming style. Our opinion is that this is the best way to capture the core ideas of algorithms and parallelism in a concise, clear, safe, and precise way. Most of the ideas we present, however, transcend the particular style of parallelism and programming we use and will be useful in a broad set of programming languages.

Nested parallelism. Nested parallelism (or nested fork-join parallelism) is a style of parallelism in which any task can fork a set of new child tasks to run in parallel. When forking, the parent task suspends, and when all the child tasks finish, they “join”, and the parent continues. Since any task can fork new tasks, the forking can be nested. Nested parallelism supports a form of parallelism in which computation can be cleanly composed either sequentially (within a task), or in parallel (among forked tasks). This in turn leads a simple cost model based on analyzing work and span.

Importantly, the model is sufficiently powerful to capture the parallelism in most of the algorithms needed for the purpose of this book. For dynamic programming, we diverge slightly from this model to a somewhat more general model.

Functional Algorithms. Functional programming is a style of programming in which functions act like mathematical functions (a mapping from domain to a codomain, and no side effects), and can be used as values (can be passed around, stored as data, and created on the fly). In this book we use this style for two important reasons.

1. Since functions have no side effects, parallelism is inherently safe and deterministic. Functions can be applied in parallel, or in different orders, without effecting each others outcomes, or the result of the final computation.

2. The ability to use functions as values allows powerful abstractions. A large fraction of the functions in many of the abstract data types we define, for example, take functions as arguments to other functions. Often these functions are created on the fly.

The functional programming style is not limited to functional programming languages. Today most programming languages support it, and in many situations the style has become dominant.

SPARC. We use a minimal, perhaps “toy”, language called SPARC to describe algorithms and data structures. It only supplies what we need for the purposes of the book, and is not meant to be fully precise. We will sometime substitute text for code. SPARC has structures for supporting nested parallelism and supports only functional programming—it does not allow for side effects.

SPARC, like many functional languages, is an extension of the lambda-calculus, which is arguably the first programming language and the basis of many ideas in modern programming languages. Chapter 9 gives a very brief overview of the lambda calculus, and Chapter 10 describes SPARC.

Function vs. Algorithm. Finally, we note that although functions in the functional programming style act like mathematical functions—i.e. a mapping from inputs to outputs—each is more than just a mathematical function. They not only embody the mapping, but they also specify the mechanism (code) by which the output is generated from the input. There can be multiple different definitions of a function that describe the same mathematical function, but compute it in different ways. Functions are therefore more accurately algorithms, and the input to output map they define is the mathematical function.

Chapter 8

Functional Algorithms

This chapter describes the idea of functional algorithms as used in this book along with some justification for the approach.

There are two aspects of functional languages that are important for our purposes. The first is that program functions are “pure”, which is to say they act like mathematical functions and have no effects beyond mapping an input to an output. The second is that functions can be treated as values, and as such can be passed as arguments to other functions, returned as results, and stored in data structures. We describe each of these two aspects in turn.

1 Pure Functions

A function in mathematics is mapping that associates each possible input x of a set X , the domain of the function, to a single output y of a set Y , the codomain of the function. When a function is applied to an element $x \in X$, the associated element y is returned. The application has no effect other than returning y .

Functions in programming languages can act like mathematical functions. For example the C “function”:

```
int double(int x) {  
    return 2 * x; }
```

acts like the mathematical function defined by the mapping

$$\{(0, 0), (-1, -2), (1, 2), (2, 4), \dots\}.$$

However, in most programming languages, including C, it is also possible to modify state that is not part of the result. This leads to the notion of side effects, or simply effects.

Definition 8.1 (Side Effects). We say that a computation has a *side effect*, if in addition to returning a value, it also performs an effect such as writing to an existing memory location (possibly part of the input), printing on the screen, or writing to a file.

Example 8.1. The C “function”

```
int double(int x) {
    y = 32;
    return 2 * x; }
```

returns twice the input x , as would a function, but also writes to the location y . It therefore has a side effect.

The process of writing over a value in a location is often called *mutation*, since it changes, or mutates, the value.

Definition 8.2 (Pure Computation). We say that a function in a program is *pure* if it doesn’t perform any side effects, and a computation is pure if all its functions are pure. Pure computations return a value without performing any side effects. In contrast an *impure* or *imperative* computation can perform side effects.

Example 8.2. The C function

```
int fib(int i) {
    if (i <= 0) return i;
    else return f(i-1) + f(i-2); }
```

is pure since it does not have any side effects. When applied it simply acts as the mathematical function defined by the mapping

$$\{(0, 0), (1, 1), (2, 1), (3, 2), (4, 3), (5, 5), \dots\}.$$

In pure computation no data can ever be overwritten, only new data can be created. Data is therefore always *persistent*—if you keep a reference to a data structure, it will always be there and in the same state as it started.

1.1 Safe for Parallelism

Pure computation is safe for parallelism. In particular, when running different components of the computation in parallel they cannot affect each other. For any two function calls $f(a)$

and $g(b)$, if we run $f(a)$ first, or $g(b)$ first, or interleave the instructions of both, the two results will always be the same. This is because f and g are (pure) functions. It means that when we specify that two components of a program can run in parallel, the run-time system is free to run them in either order on one processor, or interleave their instructions in any way on two processors, without worry.

In contrast, in imperative computation separate components, or separate function calls, can effect each other under the hood. When running such components in parallel, they can then give different results depending on the relative ordering of individual instructions of the two computations.

Definition 8.3 (Race Conditions). Side effects that alter the result of the computation based on the evaluation order (timing) are called *race conditions*.

Race conditions most often involve two components that are running in parallel, one which is writing to a location, and the other is either reading or writing the same location. Since the exact timing on real processors is highly unpredictable due to all sorts of features in the processors (caches, pipelining, interruptions, sharing of functional units, ...), it is near impossible to guarantee how instructions are interleaved between processors. Furthermore it can change every time the program is run. Hence a program with race conditions can return different results on different days. Even worse, it can return the same result for 20 years, before returning something different, and perhaps catastrophic.

Example 8.3. There are several spectacular examples of correctness problems caused by race-conditions, including for example the Northeast blackout of 2003, which affected over 50 Million people in North America.

Here are some quotes from the spokesmen of the companies involved in this event.

The first quote below describes the problem, which is a race condition (multiple computations writing to the same piece of data). "There was a couple of processes that were in contention for a common data structure, and through a software coding error in one of the application processes, they were both able to get write access to a data structure at the same time [...] And that corruption led to the alarm event application getting into an infinite loop and spinning."

The second quote describes the difficulty of finding the bug. "This fault was so deeply embedded, it took them [the team of engineers] weeks of poring through millions of lines of code and data to find it."

One approach to reason about programs with race conditions is to consider all possible interleavings of instructions on separate processors. In general, this can be very difficult since the number of interleavings is exponential in the number of instructions. However, there are structured techniques for programming with race conditions. This falls in the domain of "concurrent programs". Data structures and algorithms that reason about race conditions are typically called concurrent data structures and algorithms.

Remark (Heisenbug). Race conditions make it difficult to reason about the correctness and the efficiency of parallel algorithms. They also make debugging difficult, because each

time the code is run, it might give a different answer. For example, each time we evaluate a piece of code, we may obtain a different answer or we may obtain a correct answer 99.99% of the time but not always.

The term *Heisenbug* was coined in the early 80s to refer to a type of bug that “disappears” when you try to pinpoint or study it and “appears” when you stop studying it. They are named after the famous Heisenberg uncertainty principle which roughly says that if you localize one property, you will lose information about another complementary property. Often the most difficult Heisenbugs to find have to do with race conditions in parallel or concurrent code. These are sometimes also called concurrency bugs.

Race conditions cannot occur in pure computation, but not all impure programs have race conditions. For example, an imperative program with no parallelism has no race conditions.

1.2 Persistence

Another important benefit of pure functions is that all data is “persistent”. Since a function does not modify its input, it means that for pure functions, the input remains the same after applying the function. In other words, the input persists. This can be very useful in some applications.

Remark. The astute reader might ask the question: if the inputs persist then won’t that be a waste of memory? Indeed if a programmer keeps the input for later use, this could be a waste of memory. However, many languages have what is called a garbage collector, which will collect any data that is no longer needed. Therefore, if a function makes a call to $f(x)$, but no longer needs x , then x can be collected and its memory reclaimed. In fact, a smart compiler might recognize that x is no longer needed and not referenced by anyone else and allow it to be updated “in place” overwriting the old value. We will make use of this optimization in a later chapter that optimizes a pure implementation of sequences (Chapter 20).

1.3 Benign Effects

Benign Effects. The notion of purity can be further extended to allow for effects that are not *observable*. For example, the Fibonacci function described above may be implemented by using a mutable reference that holds some intermediate value that may be used to compute the result. If this reference is not observable (e.g., not visible to the caller of the function or any other functions), the function has no observable effect, and can thus be considered pure. Such effects are sometimes called *benign effects*. This more general notion of purity is important because it allows for example using side effects in a “responsible” fashion to improve efficiency.

Example 8.4. The C function

```
int factorial(int i) {
    int r = 1;
    for (int j = 1; j <= i; j++) r = r * i;
    return r;}
```

is not pure since it side effects (mutates) the value of `r`, but the side effect is not visible outside of the function. It is therefore a benign effect from the point of view of anyone calling `factorial`.

Important. Strictly speaking there is probably no non-trivial computation that is pure all the way to the “metal” (hardware) because almost any computation performs memory effects at the hardware level. Encapsulation of effects by observation is therefore essential for meaningful discussions of purity.

2 Functions as Values

Almost all programming languages allow applying a function to a value. Not all, however, allow passing functions as arguments, returning them from other functions, storing them in data structures and generating new functions. This ability to use functions in this way is sometimes referred to as “functions as first-class values”—i.e., functions can be treated as values.

Example 8.5 (Examples of Functions as Values). In the following definition (using SPARC)

```
f(x) =
  let g(y) = x + y
  in g end
```

$z = f(3)$

the variable z is bound to a function that adds three to its argument, i.e., the function $\{(0, 3), (1, 4), \dots\}$. We can apply $z(7)$ and it would return 10. We can also create another function $f(5)$ that adds 5 to its argument. And we can pass z , or any other function, as an argument. For example consider the definition

$g(y) = y(6)$

Now $g(z)$ returns 9 since in the body of f we apply the function z , which adds 3, to 6. And $g(f(5))$ returns 11. Finally we can store functions in data structures, as in

$\langle f(3), f(1), f(6) \rangle$

which is a sequence containing three functions, one that adds 3 one that adds 1 and one that adds 6.

Treating functions as values leads to a powerful way to code. Functions that take other functions as arguments are often called *higher-order functions*. Higher-order functions

(even in a language that is not pure) help with the design and implementation of parallel algorithm by encouraging the designer to think at a higher level of abstraction.

For example, instead of thinking about a loop that iterates over the elements of an array to generate the sum, which is completely sequential, we can define a higher-order “reduce” function. In addition to taking the array as an argument, the reduce function takes a binary associative function as another argument. It then sums the array based on that binary associative function. The advantage is that the higher-order reduce allows for any binary associative function (e.g. maximum, minimum, multiplication). By implementing the reduce function as a tree sum, which is highly parallel, we can thus perform a variety of computations in parallel rather than sequentially as a loop. In general, thinking in higher order functions encourages working at a higher level of abstraction, moving us away from the one-at-a-time (loop) way of thinking that is detrimental to code quality and to parallelism.

3 Functional Algorithms

In this book we use algorithms that use pure functions and support functions as first-class values. We refer to these as *functional algorithms*.

Remark. Coding a functional algorithm does not require a purely functional programming language. In fact, a functional algorithm can be coded in essentially any programming language—one just needs to be very careful when coding imperatively in order to avoid errors caused by sharing of state and side effects. Some imperative parallel languages such as extension to the C language, in fact, encourage programming functional algorithms. The techniques that we describe thus are applicable to imperative programming languages as well.

Chapter 9

The Lambda Calculus

This section briefly describes the lambda calculus, one of the earliest and most important contributions to computer science. It is a pure language, only supporting pure functions, and it fully supports higher-order functions.

The lambda calculus, developed by Alonzo Church in the early 30s, is arguably the first general purpose “programming language”. Although it is very simple with only three types of expressions, and one rule for “evaluation”, it captures many of the core ideas of modern programming languages. The idea of variables, functions, and function application are built in. Although conditionals and recursion are not built in, they can be easily implemented. Furthermore, although it has no primitive data types, integers, lists, trees, and other structures, can also be easily implemented. Perhaps most importantly for this book, and impressive given it was developed before computers even existed, the lambda calculus is inherently parallel. The language (pseudocode) we use in this book, SPARC, is effectively an extended and typed lambda calculus.

1 Syntax and Semantics

Definition 9.1 (Syntax of the Lambda Calculus). The lambda calculus consists of expressions e that are in one of the following three forms:

- a *variable*, such as x, y, z, \dots ,
- a *lambda abstraction*, written as $(\lambda x . e)$, where x is a variable name and e is an expression, or
- an *application*, written as $(e_1 e_2)$, where e_1 and e_2 are expressions.

A lambda abstraction $(\lambda x . e)$ defines a function where x is the argument parameter and e is the body of the function, likely containing x , possibly more than once. An application $(e_1 e_2)$ indicates that the function calculated from e_1 should be applied to the expression e_2 . This idea of function application is captured by beta reduction.

Definition 9.2 (Beta Reduction). For any application for which the left hand expression is a lambda abstraction, beta reduction “applies the function” by making the transformation:

$$(\lambda x . e_1) e_2 \longrightarrow e_1[x/e_2]$$

where $e_1[x/e_2]$ roughly means for every (free) occurrence of x in e_1 , substitute it with e_2 .

Note that this is the standard notion of function application, in which we pass in the value or the argument(s) by setting the function variables to those values.

Computation in the lambda calculus consists of using beta reduction until there is nothing left to reduce. An expression that has nothing left to reduce is in *normal form*. It is possible that an expression in the lambda calculus can “loop forever” never reducing to normal form. Indeed, the possibility of looping forever is crucial in any general (Church-Turing complete) computational model.

Exercise 9.1. Argue that the following expression in the lambda calculus never reduces to normal form, i.e., however many times beta reduction is applied, it can still be applied again.

$$((\lambda x .(x x)) (\lambda x .(x x))) .$$

Solution. A beta reduction will replace the two xs in the first lambda with the second lambda. This will generate the same expression as the original. This can be repeated any number of times, and will always come to the same point.

Church-Turing Hypothesis. In the early 30s, soon after he developed the language, Church argued that anything that can be “effectively computed” can be computed with the lambda calculus, and therefore it is a universal mechanism for computation. However, it was not until a few years later when Alan Turing developed the Turing machine and showed its equivalence to the lambda calculus that the concept of universality became widely accepted. The fact that the models were so different, but equivalent in what they can compute, was a powerful argument for the universality of the models. We now refer to the hypothesis that anything that can be computed can be computed with the lambda calculus, or equivalently the Turing machine, as the *Church-Turing hypothesis*, and refer to any computational model that is computationally equivalent to the lambda calculus as *Church-Turing complete*.

2 Parallelism and Reduction Order

Unlike the Turing machine, the lambda calculus is inherently parallel. This is because there can be many applications in an expression for which beta reduction can be applied, and the

lambda calculus allows them to be applied in any order, including a parallel order—e.g., all at once. We have to be careful, however, since the number of reductions needed to evaluate an expression (reduce to normal form) can depend significantly on the reduction order. In fact, some orders might terminate while others will not. Because of this, specific orders are used in practice. The two most prominent orders adopted by programming languages are called “call-by-value” and “call-by-need.” In both these orders lambda abstractions are considered values and beta reductions are not applied inside of them.

Definition 9.3 (Call-by-Value). In *call-by-value* evaluation order, beta reduction is only applied to $(\lambda x . e_1) e_2$ if the expression e_2 is a value, i.e., e_2 is evaluated to a value (lambda abstraction) first, and then beta reduction is applied.

Example 9.1. The ML class of languages such as Standard ML, CAML, and OCAML, all use call-by-value evaluation order.

Definition 9.4 (Call-by-Need). In *call-by-need* evaluation order, beta reduction is applied to $(\lambda x . e_1) e_2$ even if e_2 is not a value (it could be another application). If during beta reduction e_2 is copied into each variable x in the body, this reduction order is called *call-by-name*, and if e_2 is shared, it is called call-by-need.

Example 9.2. The Haskell language is perhaps the most well known example of a call-by-need (or lazy) functional language.

Since neither reduction order reduce inside of a lambda abstraction, neither of them reduce expressions to normal form. Instead they reduce to what is called “weak head normal form”. However, both reduction orders, as with most orders, remain Church-Turing complete.

Call-by-value is an inherently parallel reduction order. This is because in an expression $(e_1 e_2)$ the two subexpressions can be evaluated (reduced) in parallel, and when both are fully reduced we can apply beta reduction to the results. Evaluating in parallel, or not, has no effect on which reductions are applied, only on the order in which they are applied. On the other hand call-by-need is inherently sequential. In an expression $(e_1 e_2)$ only the first subexpression can be evaluated and when completed we can apply beta reduction to the resulting lambda abstraction by substituting in the second expression. Therefore the second expression cannot be evaluated until the first is done without potentially changing which reductions are applied.

In this book we use call-by-value.

Chapter 10

The SPARC Language

This chapter presents SPARC: a parallel and functional language used throughout the book for specifying algorithms.

SPARC is a “strict” functional language similar to the ML class of languages such as Standard ML or SML, Caml, and F#. In pseudo code, we sometimes use mathematical notation, and even English descriptions in addition to SPARC syntax. This chapter describes the basic syntax and semantics of SPARC; we introduce additional syntax as needed in the rest of the book.

1 Syntax and Semantics of SPARC

This section describes the syntax and the semantics of the core subset of the SPARC language. The term *syntax* refers to the structure of the program itself, whereas the term *semantics* refers to what the program computes. Since we wish to analyze the cost of algorithms, we are interested in not just what algorithms compute, but how they compute. Semantics that capture how algorithms compute are called *operational semantics*, and when augmented with specific costs, *cost semantics*. Here we describe the syntax of SPARC and present an informal description of its operational semantics. We will cover the cost semantics of SPARC in Chapter 13. While we focus primarily on the core subset of SPARC, we also describe some *syntactic sugar* that makes it easier to read or write code without adding any real power. Even though SPARC is a strongly typed language, for our purposes in this book, we use types primarily as a means of describing and specifying the behavior of our algorithms. We therefore do not present careful account of SPARC’s type system.

The definition below shows the syntax of SPARC. A SPARC program is an expression, whose syntax, describe the computations that can be expressed in SPARC. When evalu-

ated an expression yield a value. Informally speaking, evaluation of an expression proceeds involves evaluating its sub-expressions to values and then combining these values to compute the value of the expression. SPARC is a strongly typed language, where every closed expression, which have no undefined (free) variables, evaluates to a value or runs forever.

Identifier	<i>id</i>
Variables	<i>x</i>
Type Constructors	<i>tycon</i>
Data Constructors	<i>dcon</i>
Patterns	<i>p</i>

Types	τ
-------	--------

Data Types	<i>dty</i>
Values	<i>v</i>

Definition 10.1 (SPARC expressions).

Expression	<i>e</i>
------------	----------

Operations	<i>op</i>
Bindings	<i>b</i>

Identifiers. In SPARC, variables, type constructors, and data constructors are given a name, or an *identifier*. An identifier consist of only alphabetic and numeric characters (a-z, A-Z, 0-9), the underscore character (“_”), and optionally end with some number of “primes”. Example identifiers include, x' , x_1 , x_l , $myVar$, $myType$, $myData$, and my_data .

Program *variables*, *type constructors*, and *data constructors* are all instances of identifiers. During evaluation of a SPARC expression, variables are bound to values, which may then be used in a computation later. In SPARC, variable are *bound* during function application, as part of matching the formal arguments to a function to those specified by the application, and also by `let` expressions. If, however, a variable appears in an expression but it is not bound by the expression, then it is *free* in the expression. We say that an expression is *closed* if it has no free variables.

Types constructors give names to types. For example, the type of binary trees may be given the type constructor $btree$. Since for the purposes of simplicity, we rely on mathematical rather than formal specifications, we usually name our types behind mathematical conventions. For example, we denote the type of natural numbers by \mathbb{N} , the type of integers by \mathbb{Z} , and the type of booleans by \mathbb{B} .

Data constructors serve the purpose of making complex data structures. By convention, we will capitalize data constructors, while starting variables always with lowercase letters.

Patterns. In SPARC, variables and data constructors can be used to construct more complex *patterns* over data. For example, a pattern can be a pair (x, y) , or a triple of variables (x, y, z) , or it can consist of a data constructor followed by a pattern, e.g., $Cons(x)$ or $Cons(x, y)$. Patterns thus enable a convenient and concise way to pattern match over the data structures in SPARC.

Built-in Types. Types of SPARC include base types such as integers \mathbb{Z} , booleans \mathbb{B} , product types such as $\tau_1 * \tau_2 \dots \tau_n$, function types $\tau_1 \rightarrow \tau_2$ with domain τ_1 and range τ_2 , as well as user defined data types.

Data Types. In addition to built-in types, a program can define new *data types* as a union of tagged types, also called variants, by “unioning” them via distinct *data constructors*. For example, the following data type defines a point as a two-dimensional or a three-dimensional coordinate of integers.

```
type point  =  PointTwo of  $\mathbb{Z} * \mathbb{Z}$ 
          |  Point3D of  $\mathbb{Z} * \mathbb{Z} * \mathbb{Z}$ 
```

Recursive Data Types. In SPARC recursive data types are relatively easy to define and compute with. For example, we can define a point list data type as follows

```
type plist = Nil | Cons of point * plist.
```

Based on this definition the list

```
Cons(PointTwo(0, 0),
      Cons(PointTwo(0, 1),
            Cons(PointTwo(0, 2), Nil)))
```

defines a list consisting of three points.

Exercise 10.1 (Booleans). Some built-in types such as booleans, \mathbb{B} , are in fact syntactic sugar and can be defined by using union types as follows. Describe how you can define booleans using data types of SPARC.

Solution. Booleans can be defined as follows.

```
type myBool = myTrue | myFalse
```

Option Type. Throughout the book, we use *option* types quite frequently. Option types for natural numbers can be defined as follows.

```
type option = None | Some of N
```

Similarly, we can define option types for integers.

```
type intOption = INone | ISome of Z
```

Note that we used a different data constructor for naturals. This is necessary for type inference and type checking. Since, however, types are secondary for our purposes in this book, we are sometimes sloppy in our use of types for the sake of simplicity. For example, we use throughout *None* and *Some* for option types regardless of the type of the contents.

Values. Values of SPARC, which are the irreducible units of computation include natural numbers, integers, Boolean values `true` and `false`, unary primitive operations, such as boolean negation `not`, arithmetic negation `-`, as well as binary operations such as logical and `and` and arithmetic operations such as `+`. Values also include constant-length tuples, which correspond to product types, whose components are values. Example tuples used commonly through the book include binary tuples or pairs, and ternary tuples or triples. Similarly, data constructors applied to values, which correspond to sum types, are also values.

As a functional language, SPARC treats all function as values. The anonymous function `lambda p. e` is a function whose arguments are specified by the pattern *p*, and whose body is the expression *e*.

- Example 10.1.**
- The function `lambda x.x + 1` takes a single variable as an argument and adds one to it.
 - The function `lambda (x, y). x` takes a pairs as an argument and returns the first component of the pair.

Expressions. Expressions, denoted by e and variants (with subscript, superscript, prime), are defined inductively, because in many cases, an expression contains other expressions. Expressions describe the computations that can be expressed in SPARC. Evaluating an expression via the operational semantics of SPARC produce the value for that expression.

Infix Expressions. An *infix expression*, $e_1 \text{ op } e_2$, involve two expressions and an infix operator op . The infix operators include $+$ (plus), $-$ (minus), $*$ (multiply), $/$ (divide), $<$ (less), $>$ (greater), or , and and . For all these operators the infix expression $e_1 \text{ op } e_2$ is just syntactic sugar for $f(e_1, e_2)$ where f is the function corresponding to the operator op (see parenthesized names that follow each operator above).

We use standard precedence rules on the operators to indicate their parsing. For example in the expression

3 + 4 * 5

the $*$ has a higher precedence than $+$ and therefore the expression is equivalent to $3 + (4 * 5)$.

Furthermore all operators are left associative unless stated otherwise, i.e., that is to say that $a \text{ op}_1 b \text{ op}_2 c = (a \text{ op}_1 b) \text{ op}_2 c$ if op_1 and op_2 have the same precedence.

Example 10.2. The expressions $5 - 4 + 2$ evaluates to $(5 - 4) + 2 = 3$ not $5 - (4 + 2) = -1$, because $-$ and $+$ have the same precedence.

Sequential and Parallel Composition. Expressions include two special infix operators: $,$ and \parallel , for generating ordered pairs, or tuples, either sequentially or in parallel.

The *comma* operator or *sequential composition* as in the infix expression (e_1, e_2) , evaluates e_1 and e_2 sequentially, one after the other, and returns the ordered pair consisting of the two resulting values. Parenthesis delimit tuples.

The *parallel* operator or *parallel composition* $"\parallel"$, as in the infix expression $(e_1 \parallel e_2)$, evaluates e_1 and e_2 in parallel, at the same time, and returns the ordered pair consisting of the two resulting values.

The two operators are identical in terms of their return values. However, we will see later, their cost semantics differ: one is sequential and the other parallel. The comma and parallel operators have the weakest, and equal, precedence.

Example 10.3. • The expression

`lambda (x, y). (x * x, y * y)`

is a function that takes two arguments x and y and returns a pair consisting of the squares x and y .

• The expression

`lambda (x, y). (x * x || y * y)`

is a function that takes two arguments x and y and returns a pair consisting of the squares x and y by squaring each of x and y in parallel.

Case Expressions. A *case expression* such as

```
case e1
| Nil => e2
| Cons (x, y) => e3
```

first evaluates the expression e_1 to a value v_1 , which must return data type. It then matches v_1 to one of the patterns, *Nil* or *Cons (x, y)* in our example, binds the variable if any in the pattern to the respective sub-values of v_1 , and evaluates the “right hand side” of the matched pattern, i.e., the expression e_2 or e_3 .

Conditionals. A conditional or an *if-then-else expression*, `if e1 then e2 else e3`, evaluates the expression e_1 , which must return a Boolean. If the value of e_1 is true then the result of the if-then-else expression is the result of evaluating e_2 , otherwise it is the result of evaluating e_3 . This allows for conditional evaluation of expressions.

Function Application. A *function application*, $e_1 e_2$, applies the function generated by evaluating e_1 to the value generated by evaluating e_2 . For example, let's say that e_1 evaluates to the function f and e_2 evaluates to the value v , then we apply f to v by first matching v to the argument of f , which is pattern, to determine the values of each variable in the pattern. We then substitute in the body of f the value of each variable for the variable. To *substitute* a value in place of a variable x in an expression e , we replace each instance of x with v .

For example if function `lambda (x, y). e` is applied to the pair $(2, 3)$ then x is given value 2 and y is given value 3. Any free occurrences of the variables x and y in the expression e will now be bound to the values 2 and 3 respectively. We can think of function application as substituting the argument (or its parts) into the free occurrences of the variables in its body e . The treatment of function application is why we call SPARC a *strict* language. In strict or call-by-value languages, the argument to the function is always evaluated to a value before applying the function. In contrast non-strict languages wait to see if the argument will be used before evaluating it to a value.

Example 10.4. • The expression

$(\lambda(x, y). x/y)(8, 2)$

evaluates to 4 since 8 and 2 are bound to x and y , respectively, and then divided.

• The expression

$(\lambda(f, x). f(x, x))(plus, 3)$

evaluates to 6 because f is bound to the function $plus$, x is bound to 3, and then $plus$ is applied to the pair $(3, 3)$.

• The expression

$(\lambda x. (\lambda y. x + y)) 3$

evaluates to a function that adds 3 to any integer.

Bindings. The *let expression*,

$\text{let } b^+ \text{ in } e \text{ end,}$

consists of a sequence of bindings b^+ , which define local variables and types, followed by an expression e , in which those bindings are visible. In the syntax for the bindings, the superscript + means that b is repeated one or more times. Each binding b is either a variable binding, a function binding, or a type binding. The let expression evaluates to the result of evaluating e given the variable bindings defined in b .

A *function binding*, $x(p) = e$, consists of a function name, x (technically a variable), the arguments for the function, p , which are themselves a pattern, and the body of the function, e .

Each *type binding* equates a type to a base type or a data type.

Example 10.5. Consider the following let expression.

```
let
  x = 2 + 3
  f(w) = (w * 4, w - 2)
  (y, z) = f(x - 1)
  in
  x + y + z
  end
```

The first binding the variable x to $2 + 3 = 5$; The second binding defines a function $f(w)$ which returns a pair; The third binding applies the function f to $x - 1 = 4$ returning the pair $(4 * 4, 4 - 2) = (16, 2)$, which y and z are bound to, respectively (i.e., $y = 16$ and $z = 2$). Finally the let expressions adds x, y, z and yields $5 + 16 + 2$. The result of the expression is therefore 23.

Note. Be careful about defining which variables each binding can see, as this is important in being able to define recursive functions. In SPARC the expression on the right of each binding in a `let` can see all the variables defined in previous variable `bindings`, and can see the function name variables of all binding (including itself) within the `let`. Therefore the function binding

$$x(p) = e$$

is not equivalent to the variable binding

$$x = \text{lambda } p.e,$$

because in the prior `x` can be used in `e` and in the later it cannot. Function bindings therefore allow for the definition of recursive functions. Indeed they allow for mutually recursive functions since the body of function bindings within the same `let` can reference each other.

Example 10.6. The expression

```
let
  f(i) = if (i < 2) then i else i * f(i - 1)
  in
  f(5)
  end
```

will evaluate to the factorial of 5, i.e., $5 * 4 * 3 * 2 * 1$, which is 120.

Example 10.7. The piece of code below illustrates an example use of data types and higher-order functions.

```
let
  type point = PointTwo of  $\mathbb{Z} * \mathbb{Z}$ 
  | PointThree of  $\mathbb{Z} * \mathbb{Z} * \mathbb{Z}$ 
  injectThree (PointTwo (x, y)) = PointThree (x, y, 0)
  projectTwo (PointThree (x, y, z)) = PointTwo (x, y)
  compose f g = f g
  p0 = PointTwo (0, 0)
  q0 = injectThree p0
  p1 = (compose projectTwo injectThree) p0
  in
  (p0, q0)
  end
```

The example code above defines a `point` as a two (consisting of `x` and `y` axes) or three dimensional (consisting of `x`, `y`, and `z` axes) point in space. The function `injectThree` takes a 2D point and transforms it to a 3D point by mapping it to a point on the $z = 0$ plane. The function `projectTwo` takes a 3D point and transforms it to a 2D point by dropping its `z` coordinate. The function `compose` takes two functions `f` and `g` and composes them. The function `compose` is a higher-order function, since it operates on functions.

The point `p0` is the origin in 2D. The point `q0` is then computed as the origin in 3D. The point `p1` is computed by injecting `p0` to 3D and then projecting it back to 2D by dropping the `z` components, which yields again `p0`. In the end we thus have $p0 = p1 = (0, 0)$.

Example 10.8. The following SPARC code, which defines a binary tree whose leaves and internal nodes holds keys of integer type. The function *find* performs a lookup in a given binary-search tree *t*, by recursively comparing the key *x* to the keys along a path in the tree.

```
type tree = Leaf of  $\mathbb{Z}$  | Node of (tree,  $\mathbb{Z}$ , tree)
find (t, x) =
  case t
  | Leaf y => x = y
  | Node (left, y, right) =>
    if x = y then
      return true
    else if x < y then
      find (left, x)
    else
      find (right, x)
```

Remark.

The definition

```
lambda x . (lambda y . f(x, y))
```

takes a function *f* of a pair of arguments and converts it into a function that takes one of the arguments and returns a function which takes the second argument. This technique can be generalized to functions with multiple arguments and is often referred to as *currying*, named after Haskell Curry (1900-1982), who developed the idea. It has nothing to do with the popular dish from Southern Asia, although that might be an easy way to remember the term.

Part III

Analysis of Algorithms

Chapter 11

Introduction

Algorithm Analysis. The term *algorithm analysis* refers to mathematical analysis of algorithms with the purpose of determining their consumption of resources such as the amount of total work they perform, the energy they consume, the time to execute, and the memory or storage space that they require.

When analyzing algorithms, it is important to be *precise* so that we can compare different algorithms and assess their suitability for our purposes. It is also equally important to be *abstract* because we don't want to worry about details of compilers and computer architectures, and because we want our analysis to remain valid even as these details change over time.

To find the right balance between precision and abstraction, we rely on two levels of abstraction: asymptotic analysis and cost models.

- Asymptotic analysis enables abstracting from small factors such as the exact time a particular operation may require. Chapter 12 describes the basics of asymptotic analysis.
- Cost models specify the cost of operations available in a computational model, usually only up to the precision of the asymptotic analysis. Chapter 13 describes machine-based and language-based cost models.

Many algorithms in computer science are naturally recursive. Analyses of such algorithms typically lead us to recurrences, which are recursive mathematical relations. Solving such recurrences is a basic skill for every computer scientist. Chapter 14 covers recurrences and the basic techniques for solving them.

Chapter 12

Asymptotics

This chapter describes the asymptotic notation that is used nearly universally in computer science to analyze the resource consumption of algorithms.

1 Basics

When analyzing algorithms, we are usually interested in costs such as the total work, the running time, or space usage. In such analysis, we typically characterize the behavior of an algorithm with a *numeric function* from the domain of natural numbers (typically input sizes) to the codomain of real numbers (cost).

Example 12.1 (Numeric Functions). By analyzing the work of the algorithm A for problem P in terms of its input size n , we may obtain the numeric function

$$W_A(n) = 2n \lg n + 3n + 4 \lg n + 5.$$

By applying the analysis method to another algorithm, algorithm B , we may derive the numeric function

$$W_B(n) = 6n + 7 \lg^2 n + 8 \lg n + 9.$$

Both of these functions are numeric because their domain is the natural numbers.

When given numeric functions, how should we interpret them? Perhaps more importantly given two algorithms and their work cost as represented by two numeric functions, how should we compare them? One option would be to calculate the two functions for varying values of n and pick the algorithm that does the least amount of work for the values of n that we are interested in.

In computer science, we typically care about the cost of an algorithm for large inputs. We are therefore usually interested in the *growth* or the *growth rate* of the functions. Asymptotic analysis offers a technique for comparing algorithms by comparing the growth rate of their cost functions as the sizes get large (approach infinity).

Example 12.2 (Asymptotics). Consider two algorithms A and B for a problem P and suppose that their work costs, in terms of the input size n , are

$$W_A(n) = 2n \lg n + 3n + 4 \lg n + 5, \text{ and}$$

$$W_B(n) = 6n + 7 \lg^2 n + 8 \lg n + 9.$$

Via asymptotic analysis, we derive

$$W_A(n) \in \Theta(n \lg n), \text{ and}$$

$$W_B(n) \in \Theta(n).$$

Since $n \lg n$ grows faster than n , we would usually prefer the second algorithm, because it performs better for sufficiently large inputs.

The difference between the exact work expressions and the “asymptotic bounds” written in terms of the “Theta” functions is that the latter ignores so called *constant factors*, which are the constants in front of the variables, and *lower-order terms*, which are the terms such as $3n$ and $4 \lg n$ that diminish in growth with respect to $n \lg n$ as n increases.

Remark. In addition to enabling us to compare algorithms, asymptotic analysis also allows us to ignore certain details such as the exact time an operation may require to complete on a particular architecture. This is important because it makes it possible to apply our analysis to different architectures, where such constant may differ. Furthermore, it also enables us to create more abstract cost models: in designing cost models, we assign most operations unit costs regardless of the exact time they might take on hardware. This greatly simplifies the definition of the models.

Exercise 12.1. Comparing two algorithms that solve the same problem, one might perform better on large inputs and the other on small inputs. Can you give an example?

Solution. There are many such algorithms. A classic example is the merge-sort algorithm that performs $\Theta(n \lg n)$ work, but performs worse on smaller inputs than the asymptotically inefficient $\Theta(n^2)$ -work insertion-sort algorithm. Asymptotic notation does not help in comparing the efficiency of insertion sort and merge sort at small input sizes. For this, we need to compare their actual work functions which include the constant factors and lower-order terms that asymptotic notation ignores.

2 Big-O, big-Omega, and big-Theta

The key idea in asymptotic analysis is to understand how the growth rate of two functions compare on large input. In particular as we increase the numeric argument of both functions to infinity, does one grow faster, equally fast or slower than the other? In answering

in this question we do not care about small input and do not care about constant factors. To capture this idea, we use the following definition.

Definition 12.1 (Asymptotic dominance). Let $f(\cdot)$ and $g(\cdot)$ be two numeric functions. We say that $f(\cdot)$ *asymptotically dominates* $g(\cdot)$, if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0$,

$$g(n) \leq c \cdot f(n).$$

or, equivalently, if

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} \leq c.$$

Example 12.3. In the following examples, the function $f(\cdot)$ asymptotically dominates and thus grows at least as fast as the function $g(\cdot)$.

$$\begin{array}{ll} f(n) = 2n & g(n) = n \\ f(n) = 2n & g(n) = 4n \\ f(n) = n \lg n & g(n) = 8n \\ f(n) = n \lg n & g(n) = 8n \lg n + 16n \\ f(n) = n\sqrt{n} & g(n) = n \lg n + 2n \\ f(n) = n\sqrt{n} & g(n) = n \lg^8 n + 16n \\ f(n) = n^2 & g(n) = n \lg^2 n + 4n \\ f(n) = n^2 & g(n) = n \lg^2 n + 4n \lg n + n \end{array}$$

In the definition we ignore all n that are less than n_0 (i.e. small inputs), and we allow $g(n)$ to be some constant factor, c , larger than $f(n)$ even though $f(n)$ “dominates”. When a function $f(\cdot)$ asymptotically dominates (or dominates for short) $g(\cdot)$, we sometimes say that $f(\cdot)$ grows as least as fast as $g(\cdot)$.

Exercise 12.2. Prove that for all k , $f(n) = n$ asymptotically dominates $g(n) = \ln^k n$.

Hint: use L'Hopital's rule, which states:

$$\text{if } \lim_{n \rightarrow \infty} f(n) = \infty \text{ and } \lim_{n \rightarrow \infty} g(n) = \infty, \text{ then: } \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \lim_{n \rightarrow \infty} \frac{g'(n)}{f'(n)}.$$

Solution. We have:

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} &= \lim_{n \rightarrow \infty} \frac{\ln^k n}{n} \\ &= \left(\lim_{n \rightarrow \infty} \frac{\ln n}{n^{1/k}} \right)^k \\ &= \left(\lim_{n \rightarrow \infty} \frac{1/n}{(1/k)n^{1/k-1}} \right)^k \\ &= \left(\lim_{n \rightarrow \infty} \frac{k}{n^{1/k}} \right)^k \\ &= 0 \end{aligned}$$

We applied L'Hospital's rule from the second to the third line. Since 0 is certainly upper bounded by a constant c , we have that f dominates g .

For two functions f and g it is possible neither dominates the other. For example, for $f(n) = n \sin(n)$ and $g(n) = n \cos(n)$ neither dominates since they keep crossing. However, both f and g are dominated by $h(n) = n$.

The dominance relation defines what is called a *preorder* (distinct from “pre-order” for traversal of a tree) over numeric functions. This means that the relation is transitive (i.e., if f dominates g , and g dominates h , then f dominates h), and reflexive (i.e., f dominates itself).

Exercise 12.3. Prove that asymptotic dominance is transitive.

Solution. By the definition of dominance we have that

1. for some c_a, n_a and all $n \geq n_a$, $g(n) \leq c_a \cdot f(n)$, and
2. for some c_b, n_b and all $n \geq n_b$, $h(n) \leq c_b \cdot g(n)$.

By plugging in, we have that for all $n \geq \max(n_a, n_b)$

$$h(n) \leq c_b(c_a f(n)).$$

This satisfies the definition f dominates h with $c = c_a \cdot c_b$ and $n_0 = \max(n_a, n_b)$.

Definition 12.2 ($O, \Omega, \Theta, o, \omega$ Notation). Consider the set of all numeric functions F , and $f \in F$. We define the following sets:

Name	Definition	Intuitively
big-O	$O(f) = \{g \in F \text{ such that } f \text{ dominates } g\}$	$\leq f$
big-Omega	$\Omega(f) = \{g \in F \text{ such that } g \text{ dominates } f\}$	$\geq f$
big-Theta	$\Theta(f) = O(f) \cap \Omega(f)$	$= f$
little-o	$o(f) = O(f) \setminus \Omega(f)$	$< f$
little-omega	$\omega(f) = \Omega(f) \setminus O(f)$	$> f$

Here “\” means set difference.

Example 12.4.

$$\begin{aligned}
 f(n) &= 2n & \in O(n) \\
 f(n) &= 2n & \in \Omega(n) \\
 f(n) &= 2n & \in \Theta(n) \\
 f(n) &= 2n & \in O(n^2) \\
 f(n) &= 2n & \in o(n^2) \\
 f(n) &= 2n & \in \Omega(\sqrt{n}) \\
 f(n) &= 2n & \in \omega(\sqrt{n}) \\
 f(n) &= n \lg^8 n + 16n & \in O(n\sqrt{n}) \\
 f(n) &= n \lg^2 n + 4n \lg n + n & \in \Theta(n \lg^2 n)
 \end{aligned}$$

Exercise 12.4. Prove or disprove the following statement: if $g(n) \in O(f(n))$ and $g(n)$ is a finite function ($g(n)$ is finite for all n), then it follows that there exist constants k_1 and k_2 such that for all $n \geq 1$,

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

Solution. The statement is correct. Because $g(n) \in O(f(n))$, we know by the definition that there exists positive constants c and n_0 such that for all $n \geq n_0$, $g(n) \leq c \cdot f(n)$. It follows that for the function $k_1 \cdot f(n) + k_2$ where $k_1 = c$ and $k_2 = \sum_{i=1}^{n_0} g(i)$, we have $g(n) \leq k_1 \cdot f(n) + k_2$.

We often think of $g(n) \in O(f(n))$ as indicating that $f(n)$ is an **upper bound** for $g(n)$. Similarly $g(n) \in \Omega(f(n))$ indicates that $f(n)$ is a **lower bound** for $g(n)$, and $g(n) \in \Theta(f(n))$ indicates that $f(n)$ is a **tight bound** for $g(n)$.

3 Some Conventions

When using asymptotic notations, we follow some standard conventions of convenience.

Writing = Instead of \in . It is reasonably common to write $g(n) = O(f(n))$ instead of $g(n) \in O(f(n))$ (or equivalently for Ω and Θ). This is often considered abuse of notation since in this context the “=” does not represent any form of equality—it is not even reflexive. In this book we try to avoid using “=”, although we expect it still appears in various places.

Common Cases. By convention, and in common use, we use the following names:

<i>linear</i>	$: O(n)$
<i>sublinear</i>	$: o(n)$
<i>quadratic</i>	$: O(n^2)$
<i>polynomial</i>	$: O(n^k)$, for any constant k .
<i>superpolynomial</i>	$: \omega(n^k)$, for any constant k .
<i>logarithmic</i>	$: O(\lg n)$
<i>polylogarithmic</i>	$: O(\lg^k n)$, for any constant k .
<i>exponential</i>	$: O(a^n)$, for any constant $a > 1$.

Expressions as Sets. We typically treat expressions that involve asymptotic notation as sets. For example, in $g(n) + O(f(n))$, represents the set of functions $\{g(n) + h(n) : h(n) \in f(n)\}$. The exception is when using big-O in recurrences, which we will discuss in Chapter 14.

Subsets. We can use big-O (Ω , Θ) on both the left and right-hand sides of an equation. In this case we are indicating that one set of functions is a subset of the other. For example, consider $\Theta(n) \subset O(n^2)$. This equation indicates that the set of functions on the left-hand side is contained in the set on the right hand side. Again, sometimes “=” is used instead of “ \subset ”.

The Argument. When writing $O(n + a)$ we have to guess what the argument of the function is—is it n or is it a ? By convention we assume the letters l, m , and n are the arguments when they appear. A more precise notation would be to use $O(\lambda n. n + a)$ —after all the argument to the big-O is supposed to be a function, not an expression.

Multiple Arguments. Sometimes the function used in big-O notation has multiple arguments, as in $f(n, m) = n^2 + m \lg n$ and used in $O(f(n, m))$. In this case $f(n, m)$ asymptotically dominates $g(n, m)$ if there exists constants c and $x_0 > 0$ such that for all inputs where $n > x_0$ or $m > x_0$, $g(n, m) \leq c \cdot f(n, m)$.

Chapter 13

Cost Models

Any algorithmic analysis must assume a *cost model* that defines the resource costs required by a computation. There are two broadly accepted ways of defining cost models: machine-based and language-based cost models.

1 Machine-Based Cost Models

Definition 13.1 (Machine-Based Cost Model). A *machine-based (cost) model* takes a machine model and defines the cost of each instruction that can be executed by the machine—often unit cost per instruction. When using a machine-based model for analyzing an algorithm, we translate the algorithm so that it can be executed on the machine and then analyze the cost of the machine instructions used by the algorithm.

Remark. Machine-based models are suitable for deriving asymptotic bounds (i.e., using big-O, big-Theta and big-Omega) but not for predicting exact runtimes. The reason for this is that on a real machine not all instructions take the same time, and furthermore not all machines have the same instructions.

1.1 RAM Model

The classic machine-based model for analyzing sequential algorithms is the *Random Access Machine* or *RAM*. In this model, a machine consists of a single processor that can access unbounded memory; the memory is indexed by the natural numbers. The processor interprets sequences of machine instructions (code) that are stored in the memory. Instructions include basic arithmetic and logical operations (e.g. $+$, $-$, $*$, and, or, not), reads from and writes to arbitrary memory locations, and conditional and unconditional jumps to other locations in the code. Each instruction takes unit time to execute, including

those that access memory. The execution-time, or simply *time* of a computation is measured in terms of the number of instructions executed by the machine. Because the model is sequential (there is only one processor) time and work are the same.

Critique of the RAM Model. Most research and development of sequential algorithms has used the RAM model for analyzing time and space costs. One reason for the RAM model's success is that it is relatively easy to reason about the costs because algorithmic pseudo code can usually be translated to the model. Similarly, code in low-level sequential languages such as C can also be translated (compiled) to the RAM Model relatively easily. When using higher level languages, the translation from the algorithm to machine instructions becomes more complex and we find ourselves making strong, possibly unrealistic assumptions about costs, even sometimes without being aware of the assumptions.

More broadly, the RAM model becomes difficult to justify in modern languages. For example, in object- oriented languages certain operations may require substantially more time than others. Likewise, features of modern programming languages such as automatic memory management can be difficult to account for in analysis. Functional features such as higher-order functions are even more difficult to reason about in the RAM model because their behavior depends on other functions that are used as arguments. Such functional features, which are the mainstay of "advanced" languages such as the ML family and Haskell, are now being adopted by more mainstream languages such as Python, Scala, and even for more primitive (closer to the machine) languages such as C++. All in all, it requires significant expertise to understand how an algorithm implemented in modern languages today may be translated to the RAM model.

Remark. One aspect of the RAM model is the assumption that accessing all memory locations has the same uniform cost. On real machines this is not the case. In fact, there can be a factor of 100 or more difference between the time for accessing different locations in memory. For example, all machines today have caches and accessing the first-level cache is usually two orders of magnitude faster than accessing main memory.

Various extensions to the RAM model have been developed to account for this non-uniform cost of memory access. One variant assumes that the cost for accessing the i^{th} memory location is $f(i)$ for some function f , e.g. $f(i) = \log(i)$. Fortunately, however, most algorithms that are good in these more detailed models are also good in the RAM model. Therefore analyzing algorithms in the simpler RAM model is often a reasonable approximation to analyzing in the more refined models. Hence the RAM has served quite well despite not fully accounting for non-uniform memory costs.

The model we use in this book also does not account for non-uniform memory costs, but as with the RAM the model can be refined to account for it.

1.2 PRAM: Parallel Random Access Machine

The RAM model is sequential but can be extended to use multiple processors which share the same memory. The extended model is called the Parallel Random Access Machine.

PRAM Model. A *Parallel Random Access Machine*, or *PRAM*, consist of p sequential random access machines (RAMs) sharing the same memory. The number of processors, p , is a parameter of the machine, and each processor has a unique index in $\{0, \dots, p-1\}$, called the *processor id*. Processors in the PRAM operate under the control of a common clock and execute one instruction at each time step. The PRAM model is most usually used as a *synchronous* model, where all processors execute the same algorithm and operate on the same data structures. Because they have distinct ids, however, different processors can do different computations.

Example 13.1. We can specify a PRAM algorithm for adding one to each element of an integer array with p elements as shown below. In the algorithm, each processor updates certain elements of the array as determined by its processor id, id .

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

If the array is larger than p , then the algorithm would have to divide the array up and into parts, each of which is updated by one processor.

SIMD Model. Because in the PRAM all processors execute the same algorithm, this typically leads to computations where each processor executes the same instruction but possibly on different data. PRAM algorithms therefore typically fit into *single instruction multiple data*, or *SIMD*, programming model. Example A.1 shows an example SIMD program.

Critique of PRAM. Since the model is synchronous, and it requires the algorithm designer to map or schedule computation to a fixed number of processors, the PRAM model can be awkward to work with. Adding a value to each element of an array is easy if the array length is equal to the number of processors, but messier if not, which is typically the case. For computations with nested parallelism, such as divide-and-conquer algorithms, the mapping becomes much more complicated.

We therefore don't use the PRAM model in this book. Most of the algorithms presented in this book, however, also work with the PRAM (with more complicated analysis), and many of them were originally developed using the PRAM model.

2 Language Based Models

Language-Based Cost-Models. A *language-based model* takes a language as the starting point and defines cost as a function mapping the expressions of the language to their cost. Such a cost function is usually defined as a recursive function over the different forms of expressions in the language. To analyze an algorithm by using a language-based model,

we apply the cost function to the algorithm written in the language. In this book, we use a language-based cost model, called the work-span model.

2.1 The Work-Span Model

Our language-based cost model is based on two cost metrics: work and span. Roughly speaking, the *work* of a computation corresponds to the total number of operations it performs, and the *span* corresponds to the longest chain of dependencies in the computation. The work and span functions can be defined for essentially any language ranging from low-level languages such as C to higher level languages such as the ML family. In this book, we use the SPARC language.

Notation for Work and Span. For an expression e , or an algorithm written in a language, we write $W(e)$ for the work of e and $S(e)$ for the span of e .

Example 13.2. For example, the notation

$$W(7 + 3)$$

denotes the work of adding 7 and 3.

The notation

$$S(fib(11))$$

denotes the span for calculating the 11th Fibonacci number using some particular code for *fib*.

The notation

$$W(mySort(a))$$

denotes the work for *mySort* applied to the sequence a .

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

$$W(mySort(a)) = O(|a| \log |a|).$$

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

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

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

Definition 13.2 (SPARC Cost Model). The work and span of SPARC expressions are defined below. In the definition and throughout this book, we write $W(e)$ for the work of the expression and $S(e)$ for its span. Both work and span are cost functions that map an expression to a integer cost. As common in language-based models, the definition follows the definition of expressions for SPARC (Chapter 10). We make one simplifying assumption in the presentation: instead of considering general bindings, we only consider the case where a single variable is bound to the value of the expression.

In the definition, the notation $\text{Eval}(e)$ evaluates the expression e and returns the result, and the notation $[v/x] e$ indicates that all free (unbound) occurrences of the variable x in the expression e are replaced with the value v .

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

Example 13.3. Consider the expression $e_1 + e_2$ where e_1 and e_2 are themselves other expressions (e.g., function application). Note that this is an instance of the rule $e_1 \text{ op } e_2$, where op is a plus operation. In SPARC, we evaluate this expressions by first evaluating e_1 and then e_2 and then computing the sum. The work of the expressions is therefore

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

The additional 1 accounts for computation of the sum.

For the `let` expression, we first evaluate e_1 and assign it to x before we can evaluate e_2 . Hence the fact that the span is composed sequentially, i.e., by adding the spans.

Example 13.4. In SPARC, `let` expressions compose sequentially.

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

Note that all the rules for work and span have the same form except for parallel application, i.e., $(e_1 \parallel e_2)$. Recall that parallel application indicates that the two expressions can be evaluated in parallel, and the result is a pair of values containing the two results. In this case we use maximum for combining the span since we have to wait for the longer of the two. In all other cases we sum both the work and span. Later we will also add a parallel construct for working with sequences.

Example 13.5. The expression $(\text{fib}(6) \parallel \text{fib}(7))$ runs the two calls to `fib` in parallel and returns the pair $(8, 13)$. It does work

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

and span

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

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

Remark. When assuming purely functional programs, it is always safe to run things in parallel if there is no explicit sequencing. For the expression $e_1 + e_2$, for example, it is safe to evaluate the two expressions in parallel, which would give the rule

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

i.e., we wait for the later of the two expression to finish, and then spend one additional unit to do the addition. However, in this book we use the convention that parallelism has to be stated explicitly using \parallel .

Definition 13.3 (Average Parallelism). Parallelism, sometimes called *average parallelism*, is defined as the work over the span:

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

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

Example 13.6. For a mergesort with work $\Theta(n \log n)$ and span $\Theta(\log^2 n)$ the parallelism would be $\Theta(n / \log n)$.

Example 13.7. Consider an algorithm with work $W(n) = \Theta(n^3)$ and span $S(n) = \Theta(n \log n)$. For $n = 10,000$, $\bar{P}(n) \approx 10^7$, which is a lot of parallelism. But, if $W(n) = \Theta(n^2) \approx 10^8$ then $\bar{P}(n) \approx 10^3$, which is much less parallelism. Note that the decrease in parallelism is not because of an increase in span but because of a reduction in work.

Designing Parallel Algorithms. In parallel-algorithm design, we aim to keep parallelism as high as possible. Since parallelism is defined as the amount of work per unit of span, we can do this by decreasing span. We can increase parallelism by increasing work also, but this is usually not desirable. In designing parallel algorithms our goals are:

1. to keep work as low as possible, and
2. to keep span as low as possible.

Except in cases of extreme parallelism, where for example, we may have thousands or more processors available to use, the first priority is usually to keep work low, even if it comes at the cost of increasing span.

Definition 13.4 (Work efficiency). We say that a parallel algorithm is *work efficient* if it performs asymptotically the same work as the best known sequential algorithm for that problem.

Example 13.8. A (comparison-based) parallel sorting algorithm with $\Theta(n \log n)$ work is work efficient; one with $\Theta(n^2)$ is not, because we can sort sequentially with $\Theta(n \log n)$ work.

Note. In this book, we will mostly cover work-efficient algorithms where the work is the same or close to the same as the best sequential time. Among the algorithms that have the same work as the best sequential time, our goal will be to achieve the greatest parallelism.

2.2 Scheduling

Scheduling involves executing a parallel program by mapping the computation over the processors in such a way to minimize the completion time and possibly, the use of other resources such as space and energy. There are many forms of scheduling. This section describes the scheduling problem and briefly reviews one particular technique called greedy scheduling.

2.2.1 Scheduling Problem

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

Scheduling can be challenging, because a parallel algorithm generates independently executable *tasks* on the fly as it runs, and it can generate a large number of them, typically many more than the number of processors.

Example 13.9. A parallel algorithm with $\Theta(n/\lg n)$ parallelism can easily generate millions of parallel subcomputations or task at the same time, even when running on a multicore computer with 10 cores. For example, for $n = 10^8$, the algorithm may generate millions of independent tasks.

Definition 13.5 (Scheduler). A *scheduling algorithm* or a *scheduler* is an algorithm for mapping parallel tasks to available processors. The scheduler works by taking all parallel tasks, which are generated dynamically as the algorithm evaluates, and assigning them to processors. If only one processor is available, for example, then all tasks will run on that one processor. If two processors are available, the task will be divided between the two.

Schedulers are typically designed to minimize the execution time of a parallel computation, but minimizing space usage is also important.

2.2.2 Greedy Scheduling

Definition 13.6 (Greedy Scheduler). We say that a scheduler is *greedy* if whenever there is a processor available and a task ready to execute, then it assigns the task to the processor and starts running it immediately. Greedy schedulers have an important property that is summarized by the greedy scheduling principle.

Definition 13.7 (Greedy Scheduling Principle). The *greedy scheduling principle* postulates that if a computation is run on P processors using a greedy scheduler, then the total time (clock cycles) for running the computation is bounded by

$$T_P < \frac{W}{P} + S$$

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

Optimality of Greedy Schedulers. This simple statement is powerful.

Firstly, the time to execute the computation cannot be less than $\frac{W}{P}$ clock cycles since we have a total of W clock cycles of work to do and the best we can possibly do is divide it evenly among the processors.

Secondly, the time to execute the computation cannot be any less than S clock cycles, because S represents the longest chain of sequential dependencies. Therefore we have

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

We therefore see that a greedy scheduler does reasonably close to the best possible. In particular $\frac{W}{P} + S$ is never more than twice $\max(\frac{W}{P}, S)$.

Furthermore, greedy scheduling is particularly good for algorithms with abundant parallelism. To see this, let's rewrite the inequality of the Greedy Principle in terms of the parallelism $\bar{P} = W/S$:

$$\begin{aligned} T_P &< \frac{W}{\bar{P}} + S \\ &= \frac{W}{\bar{P}} + \frac{W}{\bar{P}} \\ &= \frac{W}{\bar{P}} \left(1 + \frac{P}{\bar{P}}\right). \end{aligned}$$

Therefore, if $\bar{P} \gg P$, i.e., the parallelism is much greater than the number of processors, then the parallel time T_P is close to W/P , the best possible. In this sense, we can view parallelism as a measure of the number of processors that can be used effectively.

Definition 13.8 (Speedup). The *speedup* S_P of a P -processor parallel execution over a sequential one is defined as

$$S_P = T_s/T_P,$$

where T_s denotes the sequential time. We use the term *perfect speedup* to refer to a speedup that is equal to P .

When assessing speedups, it is important to select the best sequential algorithm that solves the same problem (as the parallel one).

Exercise 13.1. Describe the conditions under which a parallel algorithm would obtain near perfect speedups.

Remark. Greedy Scheduling Principle does not account for the time it requires to compute the (greedy) schedule, assuming instead that such a schedule can be created instantaneously and at no cost. This is of course unrealistic and there has been much work on algorithms that attempt to match the Greedy Scheduling Principle. No real schedulers can match it exactly, because scheduling itself requires work. For example, there will surely be some delay from when a task becomes ready for execution and when it actually starts executing. In practice, therefore, the efficiency of a scheduler is quite important to achieving good efficiency. Because of this, the greedy scheduling principle should only be viewed as an asymptotic cost estimate in much the same way that the RAM model or any other computational model should be just viewed as an asymptotic estimate of real time.

Chapter 14

Recurrences

This chapter covers recurrences and presents three methods for solving recurrences: the “[Tree Method](#)” the “[Brick Method](#)”, and the “[Substitution Method](#)”.

1 The Basics

Recurrences are simply recursive functions for which the argument(s) and result are numbers. As is normal with recursive functions, recurrences have a recursive case along with one or more base cases. Although recurrences have many applications, in this book we mostly use them to represent the cost of algorithms, and in particular their work and span. They are typically derived directly from recursive algorithms by abstracting the arguments of the algorithm based on their sizes, and using the cost model described in Chapter 13. Although the recurrence is itself a function similar to the algorithm it abstracts, the goal is not to run it, but instead the goal is to determine a closed form solution to it using other methods. Often we satisfy ourselves with finding a closed form that specifies an upper or lower bound on the function, or even just an asymptotic bound.

Example 14.1 (Fibonacci). Here is a recurrence written in SPARC that you should recognize:

$$\begin{aligned} F(n) = & \text{ case } n \text{ of} \\ & 0 \Rightarrow 0 \\ & | 1 \Rightarrow 1 \\ & | _ \Rightarrow F(n-1) + F(n-2) . \end{aligned}$$

It has an exact closed form solution:

$$F(n) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}} ,$$

where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. We can write this in asymptotic notation as

$$F(n) = \Theta(\varphi^n)$$

since the first term dominates asymptotically.

Example 14.2 (Mergesort Recurrence). Assuming that the input length is a power of 2, we can write the code for parallel mergesort algorithm as follows.

```
msort(A) =
  if |A| ≤ 1 then A
  else
    let (L, R) = msort(A[0 … |A|/2]) || msort(A[|A|/2 … |A|])
    in merge(L, R) end
```

By abstracting based on the length of A , and using the cost model described in Chapter 13, we can write a recurrence for the work of mergesort as:

```
Wmsort(n) =
  if n ≤ 1 then c1
  else
    let (WL, WR) = (Wmsort(n/2), Wmsort(n/2))
    in WL + WR + Wmerge(n) + c2 end
```

where the c_i are constants. Assuming $W_{merge}(n) = c_3 n + c_4$ this can be simplified to

```
Wmsort(n) = if n ≤ 1 then c1
             else 2Wmsort(n/2) + c3n + c5
```

where $c_5 = c_2 + c_4$. We will show in this chapter that this recurrence solves to

$$W_{msort}(n) = O(n \lg n)$$

using all three of our methods.

2 Some conventions

To reduce notation we use several conventions when writing recurrences.

Syntax. We typically write recurrences as mathematical relations of the form

$$W_f(n) = \begin{cases} c_1 & \text{base case 1} \\ c_2 & \text{base case 2} \\ \dots & \dots \\ \text{recursive definition} & \text{otherwise.} \end{cases}$$

Dropping the subscript. We often drop the subscript on the cost W or S (span) when obvious from the context.

Base case. Often base cases are trivial—i.e., some constant if $n \leq 1$. In such cases, we usually leave them out.

Big-O inside a recurrence. Technically using big-O notation in a recurrence as in:

$$W(n) = 2W(n/2) + O(n)$$

is not well defined. This is because $2W(n/2) + O(n)$ indicates a set of functions, not a single function. In this book when we use $O(f(n))$ in a recurrences it is meant as shorthand for $c_1 f(n) + c_2$, for some constants c_1 and c_2 . Furthermore, when solving the recurrence the $O(f(n))$ should always be replaced by $c_1 f(n) + c_2$.

Inequality. Because we are mostly concerned with upper bounds, we can be sloppy and add (positive) constants on the right-hand side of an equation. In such cases, we typically use an inequality, as in

$$W(n) \leq 2W(n/2) + n.$$

Input size inprecision. A technical issue concerns rounding of input sizes. Going back to the [mergesort example](#), note that we assumed that the size of the input to merge sort, n , is a power of 2. If we did not make this assumption, i.e., for general n , we would partition the input into two parts, whose sizes may differ by up to one element. In such a case, we could write the work recurrence as

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

When working with recurrences, we typically ignore floors and ceiling because they change the size of the input by at most one, which does not usually affect the closed form by more than a constant factor.

Example 14.3 (Mergesort recurrence revisited). Using our conventions we can write our recurrence for the work of mergesort as:

$$W(n) \leq 2W(n/2) + O(n).$$

However, when solving it is best to write it as:

$$W(n) \leq \begin{cases} c_b & \text{if } n \leq 1 \\ 2W(n/2) + c_1 n + c_2 & \text{otherwise.} \end{cases}$$

Assuming *merge* has logarithmic span, we can similarly write a recurrence for the span of the parallel mergesort as:

$$S(n) \leq S(n/2) + O(\lg n).$$

3 The Tree Method

Definition 14.1 (Tree Method). The *tree method* is a technique for solving recurrences. Given a recurrence, the idea is to derive a closed form solution of the recurrence by first unfolding the recurrence as a tree and then deriving a bound by considering the cost at each level of the tree. To apply the technique, we start by replacing the asymptotic notations in the recurrence, if any. We then draw a tree where each recurrence instance is represented by a subtree and the root is annotated with the cost that occurs at this level, that is beside the recurring costs.

After we determine the tree, we ask several questions.

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

Based on the answers to these questions, we can write the cost as a sum and calculate it.

Example 14.4. Consider the recurrence

$$W(n) = 2W(n/2) + O(n).$$

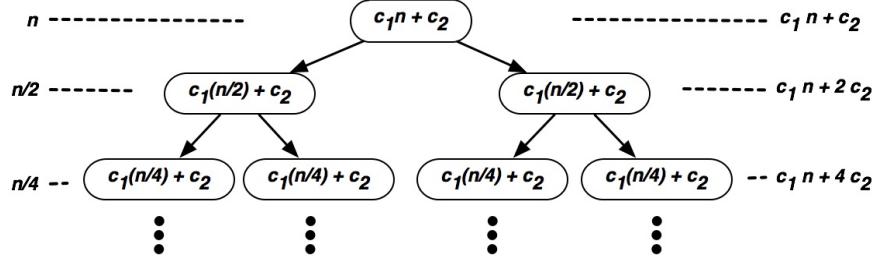
By the definition of asymptotic complexity, we can establish that

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

where c_1 and c_2 are constants.

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

The drawing below illustrates the resulting tree; each level is annotated with the problem size (left) and the cost at that level (right).



We observe that:

- level i (the root is level $i = 0$) contains 2^i nodes,
- a node at level i costs at most $c_1(n/2^i) + c_2$.

Thus, the total cost on level i is at most

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

Because we keep halving the input size, the number of levels $i \leq \lg n$. Hence, we have

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

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

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

4 The Brick Method

The *brick method* is a special case of the tree method, aimed at recurrences that grow or decay geometrically across levels of the recursion tree. A sequence of numbers has *geometric growth* if it grows by at least a constant factor (> 1) from element to element, and has *geometric decay* if it decreases by at least a constant factor. The beauty of a geometric sequence is that its sum is bounded by a constant times the last element (for geometric growth), or the first element (for geometric decay).

Exercise 14.1 (Sums of geometric series.). Consider the sum of the sequence $S = \langle 1, \alpha, \alpha^2, \dots, \alpha^n \rangle$. Show that

1. for $\alpha > 1$ (geometric growth), the sum of S is at most $\left(\frac{\alpha}{\alpha-1}\right) \cdot \alpha^n$, and
2. for $\alpha < 1$ (geometric decay), the sum of S is at most $\left(\frac{1}{1-\alpha}\right) \cdot 1$.

Hint: for the first let s be the sum, and consider $\alpha s - s$, cancelling terms as needed.

Solution. Let

$$s = \sum_{i=0}^n \alpha^i.$$

To solve the first case we use

$$\begin{aligned} \alpha s - s &= \left(\alpha \sum_{i=0}^n \alpha^i\right) - \sum_{i=0}^n \alpha^i \\ &= \left(\sum_{i=0}^n \alpha^{i+1}\right) - \sum_{i=0}^n \alpha^i \\ &= \alpha^{n+1} - 1 \\ &< \alpha^{n+1}. \end{aligned}$$

Now by dividing through by $\alpha - 1$ we get

$$s < \frac{\alpha^{n+1}}{\alpha - 1} = \left(\frac{\alpha}{\alpha - 1}\right) \cdot \alpha^n,$$

which is we wanted to show.

The second case is similar but using $s - \alpha s$.

In the tree method, if the costs grow or decay geometrically across levels (often the case), then for analyzing asymptotic costs we need only consider the cost of the root (decay), or the total cost of the leaves (growth). If there is no geometric growth or decay then it often suffices to calculate the cost of the worst level (often either the root or leaves) and multiply it by the number of levels. This leads to three cases which we refer to as root dominated, leaf dominated and balanced. Conveniently, to distinguish these three cases we need only consider the cost of each node in the tree and how it relates to the cost of its children.

Definition 14.2 (Brick Method). Consider each node v of the recursion tree, and let $N(v)$ denote its input size, $C(v)$ denote its cost, and $D(v)$ denote the set of its children. There exists constants $a \geq 1$ (base size), $\alpha > 1$ (grown/decay rate) such that:

Root Dominated For all nodes v such that $N(v) > a$,

$$C(v) \geq \alpha \sum_{u \in D(v)} C(u),$$

i.e., the cost of the parent is at least a constant factor greater than the sum of the costs of the children. In this case, the total cost is dominated by the root, and is upper bounded by $\frac{\alpha}{\alpha-1}$ times the cost of the root.

Leaves Dominated For all v such that $N(v) > a$,

$$C(v) \leq \frac{1}{\alpha} \sum_{u \in D(v)} C(u),$$

i.e., the cost of the parent is at least a constant factor less than the sum of the costs of the children. In this case, the total cost is dominated by the cost of the leaves, and is upper bounded by $\frac{\alpha}{\alpha-1}$ times the sum of the cost of the leaves. Most often all leaves have constant cost so we just have to count the number of leaves.

Balanced When neither of the two above cases is true. In this case the cost is upper bounded by the number of levels times the maximum cost of a level.

Proof. We first consider the root dominated case. For this case if the root has cost $C(r)$, level i (the root is level 0) will have total cost at most $(1/\alpha)^i C(r)$. This is because the cost of the children of every node on a level decrease by at least a factor of α to the next level. The total cost is therefore upper bounded by

$$\sum_{i=0}^{\infty} \left(\frac{1}{\alpha}\right)^i C(r).$$

This is a decaying geometric sequence and therefore is upper bounded by $\frac{\alpha}{\alpha-1} C(r)$, as claimed.

For the leaf dominated case, if all leaves are on the same level and have the same cost, we can make a similar argument as above but in the other direction—i.e. the levels increase geometrically down to the leaves. The cost is therefore dominated by the leaf level. In general, however, not all leaves are at the same level.

For the general leaf-dominated case, let L be the set of leaves. Consider the cost $C(l)$ for $l \in L$, and account a charge of $(1/\alpha)^i C(l)$ to its i -th ancestor in the tree (its parent is its first ancestor). Adding up the contributions from every leaf to the internal nodes of the tree gives the maximum possible cost for all internal nodes. This is because for this charging every internal node will have a cost that is exactly $(1/\alpha)$ the sum of the cost of the children, and this is the most each node can have by our assumption of leaf-dominated recurrences. Now summing the contributions across leaves, including the cost of the leaves themselves ($i = 0$), we have as an upper bound on the total cost across the tree:

$$\sum_{l \in L} \sum_{i=0}^{\infty} \left(\frac{1}{\alpha}\right)^i C(l).$$

This is a sum of sums of decaying geometric sequences, giving an upper bound on the total cost across all nodes of $\frac{\alpha}{\alpha-1} \sum_{l \in L} C(l)$, as claimed.

The balanced case follows directly from the fact that the total cost is the sum of the cost of the levels, and hence at most the number of levels times the level with maximum cost. \square

Remark. The term “brick” comes from thinking of each node of the tree as a brick and the width of a brick being its cost. The bricks can be thought of as being stacked up by level.

A recurrence is leaf dominated if the pile of bricks gets narrower as you go up to the root. It is root dominated if it gets wider going up to the root. It is balanced if it stays about the same width.

Example 14.5 (Root dominated). Lets consider the recurrence

$$W(n) = 2W(n/2) + n^2.$$

For a node in the recursion tree of size n we have that the cost of the node is n^2 and the sum of the cost of its children is $(n/2)^2 + (n/2)^2 = n^2/2$. In this case the cost has **decreased** by a factor of two going down the tree, and hence the recurrence is root dominated. Therefore for asymptotic analysis we need only consider the cost of the root, and we have that $W(n) = O(n^2)$.

In the leaf dominated case the cost is proportional to the number of leaves, but we have to calculate how many leaves there are. In the common case that all leaves are at the same level (i.e. all recursive calls are the same size), then it is relatively easy. In particular, one can calculate the number of recursive calls at each level, and take it to the power of the depth of the tree, i.e., (branching factor)^{depth}.

Example 14.6 (Leaf dominated). Lets consider the recurrence

$$W(n) = 2W(n/2) + \sqrt{n}.$$

For a node of size n we have that the cost of the node is \sqrt{n} and the sum of the cost of its two children is $\sqrt{n/2} + \sqrt{n/2} = \sqrt{2}\sqrt{n}$. In this case the cost has **increased** by a factor of $\sqrt{2}$ going down the tree, and hence the recurrence is leaf dominated. Each leaf corresponds to the base case, which has cost 1.

Now we need to determine how many leaves there are. Since each recursive call halves the input size, the depth of recursion is going to be $\lg n$ (the number of times one needs to halve n before getting to size 1). Now on each level the recursion is making two recursive calls, so the number of leaves will be $2^{\lg n} = n$. We therefore have that $W(n) = O(n)$.

Example 14.7 (Balanced). Lets consider the same recurrence we considered for the tree method, i.e.,

$$W(n) = 2W(n/2) + c_1n + c_2.$$

For all nodes we have that the cost of the node is $c_1n + c_2$ and the sum of the cost of the two children is $(c_1n/2 + c_2) + (c_1n/2 + c_2) = c_1n + 2c_2$. In this case the cost is about the same for the parent and children, and certainly not growing or decaying geometrically. It is therefore a balanced recurrence. The maximum cost of any level is upper bounded by $(c_1 + c_2)n$, since there are at most n total elements across any level (for the c_1n term) and at most n nodes (for the c_2n term). There are $1 + \lg n$ levels, so the total cost is upper bounded by $(c_1 + c_2)n(1 + \lg n)$. This is slightly larger than our earlier bound of $c_1n \lg n + c_2(2n - 1)$, but it makes no difference asymptotically—they are both $O(n \lg n)$.

Remark. Once you are used to using the brick method, solving recurrences can often be done very quickly. Furthermore the brick method can give a strong intuition of what part

of the program dominates the cost—either the root or the leaves (or both if balanced). This can help a programmer decide how to best optimize the performance of recursive code. If it is leaf dominated then it is important to optimize the base case, while if it is root dominated it is important to optimize the calls to other functions used in conjunction with the recursive calls. If it is balanced, then, unfortunately, both need to be optimized.

Exercise 14.2. For each of the following recurrences state whether it is leaf dominated, root dominated or balanced, and then solve the recurrence

$$\begin{aligned} W(n) &= 3W(n/2) + n \\ W(n) &= 2W(n/3) + n \\ W(n) &= 3W(n/3) + n \\ W(n) &= W(n-1) + n \\ W(n) &= \sqrt{n}W(\sqrt{n}) + n^2 \\ W(n) &= W(\sqrt{n}) + W(n/2) + n \end{aligned}$$

Solution. The recurrence $W(n) = 3W(n/2) + n$ is leaf dominated since $n \leq 3(n/2) = \frac{3}{2}n$. It has $3^{\lg n} = n^{\lg 3}$ leaves so $W(n) = O(n^{\lg 3})$.

The recurrence $W(n) = 2W(n/3) + n$ is root dominated since $n \geq 2(n/3) = \frac{2}{3}n$. Therefore $W(n) = O(n)$, i.e., the cost of the root.

The recurrence $W(n) = 3W(n/3) + n$ is balanced since $n = 3(n/3)$. The depth of recursion is $\log_3 n$, so the overall cost is n per level for $\log_3 n$ levels, which gives $W(n) = O(n \log n)$.

The recurrence $W(n) = W(n-1) + n$ is balanced since each level only decreases by 1 instead of by a constant fraction. The largest level is n (at the root) and there are n levels, which gives $W(n) = O(n \cdot n) = O(n^2)$.

The recurrence $W(n) = \sqrt{n}W(\sqrt{n}) + n^2$ is root dominated since $n^2 \geq \sqrt{n} \cdot (\sqrt{n})^2 = n^{3/2}$. In this case the decay is even faster than geometric. Certainly for any $n \geq 2$, it satisfies our root dominated condition for $\alpha = \sqrt{2}$. Therefore $W(n) = O(n^2)$.

The recurrence $W(n) = W(\sqrt{n}) + W(n/2) + n$ is root dominated since for $n > 16$, $n \geq \frac{4}{3}(\sqrt{n} + n/2)$. Note that here we are using the property that a leaf can be any problem size greater than some constant a . Therefore $W(n) = O(n)$, i.e., the cost of the root.

Advanced. In some leaf-dominated recurrences not all leaves are at the same level. An example is $W(n) = W(n/2) + W(n/3) + 1$. Let $L(n)$ be the number of leaves as a function of n . We can solve for $L(n)$ using yet another recurrence. In particular the number of leaves for an internal node is simply the sum of the number of leaves of each of its children. In the example this will give the recurrence $L(n) = L(n/2) + L(n/3)$. Hence, we need to find a function $L(n)$ that satisfies this equation. If we guess that it has the form $L(n) = n^\beta$ for some β , we can plug it into the equation and try to solve for β :

$$\begin{aligned} n^\beta &= \left(\frac{n}{2}\right)^\beta + \left(\frac{n}{3}\right)^\beta \\ &= n^\beta \left(\left(\frac{1}{2}\right)^\beta + \left(\frac{1}{3}\right)^\beta\right) \end{aligned}$$

Now dividing through by n^β gives

$$\left(\frac{1}{2}\right)^\beta + \left(\frac{1}{3}\right)^\beta = 1.$$

This gives $\beta \approx .788$ (actually a tiny bit less). Hence $L(n) < n^{.788}$, and because the original recurrence is leaf dominated: $W(n) \in O(n^{.788})$.

This idea of guessing a form of a solution and solving for it is key in our next method for solving recurrences, the substitution method.

5 Substitution Method

The tree method can be used to find the closed form solution to many recurrences but in some cases, we need a more powerful techniques that allows us to make a guess and then verify our guess via mathematical induction. The substitution method allows us to do that exactly.

Important. This technique can be tricky to use: it is easy to start on the wrong foot with a poor guess and then derive an incorrect proof, by for example, making a small mistake. To minimize errors, you can follow the following tips:

1. Spell out the constants—do not use asymptotic notation such as big- O . The problem with asymptotic notation is that it makes it super easy to overlook constant factors, which need to be carefully accounted for.
2. Be careful that the induction goes in the right direction.
3. Add additional lower-order terms, if necessary, to make the induction work.

Example 14.8. Consider the recurrence

$$W(n) = 2W(n/2) + O(n).$$

By the definition of asymptotic complexity, we can establish that

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

where c_1 and c_2 are constants.

We will prove the following theorem using strong induction on n .

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

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

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

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

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

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

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

Variants of the recurrence considered in our last example arise commonly in algorithms. Next, we establish a theorem that shows that the same bound holds for a more general class of recurrences.

Theorem 14.1 (Superlinear Recurrence). Let $\varepsilon > 0$ be a constant and consider the recurrence

$$W(n) = 2W(n/2) + k \cdot n^{1+\varepsilon}.$$

If $W(n) \leq 2W(n/2) + k \cdot n^{1+\varepsilon}$ for $n > 1$ and $W(n) \leq k$ for $n \leq 1$, then for some constant κ ,

$$W(n) \leq \kappa \cdot n^{1+\varepsilon}.$$

Proof. Let $\kappa = \frac{1}{1-1/2^\varepsilon} \cdot k$. The base case is easy: $W(1) = k \leq \kappa_1$ as $\frac{1}{1-1/2^\varepsilon} \geq 1$. For the inductive step, we substitute the inductive hypothesis into the recurrence and obtain

$$\begin{aligned} W(n) &\leq 2W(n/2) + k \cdot n^{1+\varepsilon} \\ &\leq 2\kappa \left(\frac{n}{2}\right)^{1+\varepsilon} + k \cdot n^{1+\varepsilon} \\ &= \kappa \cdot n^{1+\varepsilon} + \left(2\kappa \left(\frac{n}{2}\right)^{1+\varepsilon} + k \cdot n^{1+\varepsilon} - \kappa \cdot n^{1+\varepsilon}\right) \\ &\leq \kappa \cdot n^{1+\varepsilon}, \end{aligned}$$

where in the final step, we use the fact that for any $\delta > 1$:

$$\begin{aligned} 2\kappa \left(\frac{n}{2}\right)^\delta + k \cdot n^\delta - \kappa \cdot n^\delta &= \kappa \cdot 2^{-\varepsilon} \cdot n^\delta + k \cdot n^\delta - \kappa \cdot n^\delta \\ &= \kappa \cdot 2^{-\varepsilon} \cdot n^\delta + (1 - 2^{-\varepsilon})\kappa \cdot n^\delta - \kappa \cdot n^\delta \\ &\leq 0. \end{aligned}$$

An alternative way to prove the same theorem is to use the tree method and evaluate the sum directly. The recursion tree here has depth $\lg n$ and at level i (again, the root is at level 0), we have 2^i nodes, each costing $k \cdot (n/2^i)^{1+\varepsilon}$. Thus, the total cost is

$$\begin{aligned} \sum_{i=0}^{\lg n} k \cdot 2^i \cdot \left(\frac{n}{2^i}\right)^{1+\varepsilon} &= k \cdot n^{1+\varepsilon} \cdot \sum_{i=0}^{\lg n} 2^{-i \cdot \varepsilon} \\ &\leq k \cdot n^{1+\varepsilon} \cdot \sum_{i=0}^{\infty} 2^{-i \cdot \varepsilon}. \end{aligned}$$

But the infinite sum $\sum_{i=0}^{\infty} 2^{-i \cdot \varepsilon}$ is at most $\frac{1}{1-1/2^\varepsilon}$. Hence, we conclude $W(n) \in O(n^{1+\varepsilon})$.

□

6 Master Method

You might have learned in a previous course about the *master method* for solving recurrences. We do not like to use it, because it only works for special cases and does not help develop intuition. It requires that all recursive calls are the same size and are some constant factor smaller than n . It doesn't work for recurrences such as:

$$\begin{aligned} W(n) &= W(n-1) + 1 \\ W(n) &= W(2n/3) + W(n/3) + n^3 \\ W(n) &= \sqrt{n} W(\sqrt{n}) + 1 \end{aligned}$$

all for which the tree, brick, and substitution method work. We note, however, that the three cases of the master method correspond to limited cases of leaves dominated, balanced, and root dominated of the brick method.

Part IV

Sequences

Chapter 15

Introduction

If we were to identify the most fundamental ideas in computer science, we would probably end up converging on a list that includes abstract data types (ADTs) and the data structures used to implement them. Unlike an algorithm the data structures need to balance the cost of different functions within the ADT interface, often involving a tradeoff.

This part covers an ADT that mimics the mathematical concept of a sequence.

ADT for Sequences. Recall that an ADT is defined in terms of an interface consisting of a collection of functions (and possibly values) on a given abstract type, and without reference to the implementation. Chapter 16 defines such an interface for sequences, specifying the type and semantics for each of the functions. Many of the functions we define, such as *map*, *reduce*, *filter* and *scan*, are particularly useful in developing parallel algorithms. The chapter also covers a shorthand syntax we use in this book for these functions.

Cost Specifications for Sequences. Beyond the interface itself we need to know something about the costs of each of the functions. As discussed in Chapter 3, a data type can have many different implementations with different asymptotic costs, and the idea of a cost specification is to capture the cost of a class of implementations, without reference to the actual implementation. In a cost specification, the costs (work and span) for each function are defined asymptotically as a function of size (number of elements, in the case of sequences). Chapter 18 covers three different cost specifications for the sequence ADT. One is based on arrays, one on trees, and one on lists. None of these fully dominate each other. In all cases some functions are asymptotically more expensive in one and some in the other.

Implementations of Sequences. Cost specifications are meant to abstract away from the specific implementation and be useful for users of an ADT, but someone needs to imple-

ment a data structure that abide by the bounds. In Chapter 17 we describe how to match the bounds for the array based cost specification. We start with a small set of primitive operations with given costs and show how to implement the rest of the interface within the bounds given by the specification.

In Chapter 19, we present some examples using the sequences ADT, including several algorithms for computing prime numbers. In Chapter 20 we describe a reduced interface for sequences and a cost specification for the interface that makes updates faster. The cost specification is different from the others in that it is non-pure—costs will depend on the context.

1 Defining Sequences

From a mathematical standpoint it is possible to define sequences in several ways. One way is to use set theory. Another way to take a more formal approach based on constructive logic and define them inductively. Here we use basic set theory.

Mathematically, a sequence is an enumerated collection. As with a set, a sequence has *elements*. The *length* of the sequence is the number of elements in the sequence.

Sequences allow for repetition: an element can appear at multiple positions. The position of an element is called its *rank* or its *index*. Traditionally, the first element of the sequence is given rank 1, but, being computer scientists, we start at 0.

In mathematics, sequences can be finite or infinite but for our purposes in this book, finite sequences suffice. We therefore consider finite sequences only.

We define a sequence as a function whose domain is a contiguous set of natural numbers starting at zero. This definition, stated more precisely below, allows us to specify the semantics of various operations on sequences succinctly.

Definition 15.1 (Sequences). An α *sequence* is a mapping (function) from \mathbb{N} to α with domain $\{0, \dots, n-1\}$ for some $n \in \mathbb{N}$.

Example 15.1. Let $A = \{0, 1, 2, 3\}$ and $B = \{', a', ', b', ', c'\}$. The function

$$R = \{(0, ', a'), (1, ', b'), (3, ', a')\}$$

from A to B has domain $\{0, 1, 3\}$. The function is not a sequence, because its domain has a gap.

The function

$$Z = \{(1, ', b'), (3, ', a'), (2, ', a'), (0, ', a')\}$$

from A to B is a sequence. The first element of the sequence is $'a'$ and thus has rank 0. The second element is $'b'$ and has rank 1. The length of the sequence is 4.

Remark. Notice that in the definition sequences are parametrized by the type (i.e., set of possible values) of their elements.

Note. This mathematical definition might seem pedantic but it is useful for at least several reasons.

- It allows for a concise and precise definition of the semantics of the functions on sequences.
- Relating sequences to mappings creates a symmetry with the abstract data types such as tables or dictionaries for representing more general mappings.

Syntax 15.2 (Sequences and Indexing). As in mathematics, we use a special notation for writing sequences. The notation

$$\langle a_0, a_1, \dots, a_{n-1} \rangle$$

is shorthand for the sequence

$$\{(0, a_0), (1, a_1), \dots, ((n-1), a_{n-1})\}.$$

For any sequence a

- $a[i]$ refers to the element of a at position i ,
- $a[l \dots h]$ refers to the subsequence of a restricted to the position between l and h .

Example 15.2. Some example sequences follow.

- For the sequence $a = \langle 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 \rangle$, we have
 - $a[0] = 2$,
 - $a[2] = 5$, and
 - $a[1 \dots 4] = \langle 3, 5, 7, 11 \rangle$.
- A $\mathbb{Z} \rightarrow \mathbb{Z}$ function sequence:

$$\langle \lambda x. x^2, \lambda y. y + 2, \lambda x. x - 4 \rangle.$$

Syntax 15.3 (Ordered Pairs and Strings). We use special notation and terminology for sequences with two elements and sequences of characters.

- An *ordered pair* (x, y) is a pair of elements in which the element on the left, x , is identified as the *first* entry, and the one on the right, y , as the *second* entry.

- We refer to a sequence of characters as a *string*, and use the standard syntax for them, e.g., $'c_0c_1c_2 \dots c_{n-1}'$ is a string consisting of the n characters c_0, \dots, c_{n-1} .

Example 15.3 (Ordered Pairs and Strings). • A character sequence, or a string: $\langle 's', 'e', 'q' \rangle \equiv 'seq'$.

- An integer-and-string sequence: $\langle (10, 'ten'), (1, 'one'), (2, 'two') \rangle$.
- A string-and-string-sequence sequence: $\langle \langle 'a' \rangle, \langle 'nested', 'sequence' \rangle \rangle$.

Chapter 16

The Sequence Abstract Data Type

Sequences are one of the most prevalent ADTs (Abstract Data Types) used in this book, and more generally in computing. In this chapter, we present the interface of an ADT for sequences, describe the semantics of the functions in the ADT, and define the notation we use in this book for sequences.

1 The Abstract Data Type

Data Type 16.1 (Sequences). Define booleans as

$$\mathbb{B} = \{\text{true}, \text{false}\},$$

and orders as

$$\mathcal{O} = \{\text{less}, \text{greater}, \text{equal}\}.$$

For any element type α , the α - *sequence data type* is the type \mathbb{S}_α consisting of the set of all

α sequences, and the following values and functions on \mathbb{S}_α .

<i>length</i>	$\mathbb{S}_\alpha \rightarrow \mathbb{N}$
<i>nth</i>	$\mathbb{S}_\alpha \rightarrow \mathbb{N} \rightarrow \alpha$
<i>empty</i>	\mathbb{S}_α
<i>singleton</i>	$\alpha \rightarrow \mathbb{S}_\alpha$
<i>tabulate</i>	$(\mathbb{N} \rightarrow \alpha) \rightarrow \mathbb{N} \rightarrow \mathbb{S}_\alpha$
<i>map</i>	$(\alpha \rightarrow \beta) \rightarrow \mathbb{S}_\alpha \rightarrow \mathbb{S}_\beta$
<i>subseq</i>	$\mathbb{S}_\alpha \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{S}_\alpha$
<i>append</i>	$\mathbb{S}_\alpha \rightarrow \mathbb{S}_\alpha \rightarrow \mathbb{S}_\alpha$
<i>filter</i>	$(\alpha \rightarrow \mathbb{B}) \rightarrow \mathbb{S}_\alpha \rightarrow \mathbb{S}_\alpha$
<i>flatten</i>	$\mathbb{S}_{\mathbb{S}_\alpha} \rightarrow \mathbb{S}_\alpha$
<i>update</i>	$\mathbb{S}_\alpha \rightarrow (\mathbb{N} \times \alpha) \rightarrow \mathbb{S}_\alpha$
<i>inject</i>	$\mathbb{S}_\alpha \rightarrow \mathbb{S}_{\mathbb{N} \times \alpha} \rightarrow \mathbb{S}_\alpha$
<i>isEmpty</i>	$\mathbb{S}_\alpha \rightarrow \mathbb{B}$
<i>isSingleton</i>	$\mathbb{S}_\alpha \rightarrow \mathbb{B}$
<i>collect</i>	$(\alpha \times \alpha \rightarrow \mathcal{O}) \rightarrow \mathbb{S}_{\alpha \times \beta} \rightarrow \mathbb{S}_{\alpha \times \mathbb{S}_\beta}$
<i>iterate</i>	$(\alpha \times \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \mathbb{S}_\beta \rightarrow \alpha$
<i>reduce</i>	$(\alpha \times \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \mathbb{S}_\alpha \rightarrow \alpha$
<i>scan</i>	$(\alpha \times \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \mathbb{S}_\alpha \rightarrow (\mathbb{S}_\alpha \times \alpha)$

where the semantics of the values and functions are described in this chapter.

Syntax 16.2 (Sequence Comprehensions). Inspired by mathematical notation for sequences, we use a “sequence comprehensions” notation as defined below. In the definition,

- i is a variable ranging over natural numbers,
- x is a variable ranging over the elements of a sequence,
- e is a SPARC expression,
- e_n and e'_n are SPARC expressions whose values are natural numbers,
- e_s and e'_s are SPARC expressions whose values are a sequence,
- p is a SPARC pattern that binds one or more variables.

$ e_s $	\equiv	<i>length</i> e_s
$e_s[i]$	\equiv	<i>nth</i> e_s
$\langle \rangle$	\equiv	<i>empty</i>
$\langle e \rangle$	\equiv	<i>singleton</i> e
$\langle e : 0 \leq i < e_n \rangle$	\equiv	<i>tabulate</i> $(\lambda i. e) e_n$
$\langle e : p \in e_s \rangle$	\equiv	<i>map</i> $(\lambda p. e) e_s$
$\langle p \in e_s \mid e \rangle$	\equiv	<i>filter</i> $(\lambda p. e) e_s$
$e_s[e_n, \dots, e_{n'}]$	\equiv	<i>subseq</i> $(e_s, e_n, e'_n - e_n + 1)$
$e_s ++ e'_s$	\equiv	<i>append</i> $e_s e'_s$

2 Basic Functions

Definition 16.3 (Length and indexing). Given a sequence a , $\text{length } a$, also written $|a|$, returns the length of a (i.e., number of elements). The function nth returns the element of a sequence at a specified index, e.g. $\text{nth } a 2$, written $a[2]$, returns the element of a with rank 2. If the element demanded is out of range, the behavior is undefined and leads to an error.

Definition 16.4 (Empty and singleton). The value empty is the empty sequence, $\langle \rangle$. The function singleton takes an element and returns a sequence containing that element, e.g., $\text{singleton } 1$ evaluates to $\langle 1 \rangle$.

Definition 16.5 (Functions isEmpty and isSingleton). To identify trivial sequences such as empty sequences and singleton sequences, which contain only one element, the interface provides the functions isEmpty and isSingleton . The function isEmpty returns `true` if the sequence is empty and `false` otherwise. The function isSingleton returns `true` if the sequence consists of a one element and `false` otherwise.

3 Tabulate

Definition 16.6 (Tabulate). The function tabulate takes a function f and an natural number n and produces a sequence of length n by applying f at each position. The function f can be applied to each element in parallel. We specify tabulate as follows

$$\begin{aligned} \text{tabulate } (f : \mathbb{N} \rightarrow \alpha) (n : \mathbb{N}) : \mathbb{S}_\alpha \\ = \langle f(0), f(1), \dots, f(n-1) \rangle. \end{aligned}$$

Syntax 16.7 (Tabulate). We use the following syntax for tabulate function

$$\langle e : 0 \leq i < e_n \rangle \equiv \text{tabulate } (\lambda i. e) e_n,$$

where e and e_n are expressions, the second evaluating to an integer, and i is a variable. More generally, we can also start at any other index, as in:

$$\langle e : e_j \leq i < e_n \rangle.$$

Example 16.1 (Fibonacci Numbers). Given the function $\text{fib } i$, which returns the i^{th} Fibonacci number, the expression:

$$a = \langle \text{fib } i : 0 \leq i < 9 \rangle$$

is equivalent to

$$a = \text{tabulate } \text{fib } 9.$$

When evaluated, it returns the sequence

$$a = \langle 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 \rangle.$$

4 Map and Filter

Mapping over a sequence or filtering out elements of a sequence that does not meet a desired condition are common tasks. The sequence ADT includes the functions *map* and *filter* for these purposes.

Definition 16.8 (Map). The function *map* takes a function f and a sequence a and applies the function f to each element of a returning a sequence of equal length with the results. As with *tabulate*, in *map*, the function f can be applied to all the elements of the sequence in parallel.

We specify the behavior of *map* as follows

$$\begin{aligned} \text{map } (f : \alpha \rightarrow \beta) (a : \mathbb{S}_\alpha) : \mathbb{S}_\beta \\ = \{(i, f(x)) : (i, x) \in a\} \end{aligned}$$

or equivalently as

$$\text{map } (f : \alpha \rightarrow \beta) \langle a_1, \dots, a_{n-1} \rangle : \mathbb{S}_\alpha = \langle f(a_1), \dots, f(a_{n-1}) \rangle.$$

Syntax 16.9 (Map). We use the following syntax for the *map* function

$$\langle e : p \in e_s \rangle \equiv \text{map } (\lambda p. e) e_s,$$

where e and e_s are expressions, the second evaluating to a sequence, and p is a pattern of variables (e.g., x or (x, y)).

Definition 16.10 (Filter). The function *filter* takes a Boolean function f and a sequence a as arguments and applies f to each element of a . It then returns the sequence consisting exactly of those elements of $s \in a$ for which $f(s)$ returns true, while preserving the relative order of the elements returned.

We specify the behavior of *filter* as follows

$$\begin{aligned} \text{filter } (f : \alpha \rightarrow \mathbb{B}) (a : \mathbb{S}_\alpha) : \mathbb{S}_\alpha = \\ \{((\{(j, y) \in a \mid j < i \wedge f(y)\}|, x) : (i, x) \in a \mid f(x)\}. \end{aligned}$$

As with *map* and *tabulate*, the function f in *filter* can be applied to the elements in parallel.

Syntax 16.11 (Filter Syntax). We use the following syntax for the *filter* function

$$\langle x \in e_s \mid e \rangle \equiv \text{filter } (\lambda x. e) e_s,$$

where e and e_s are expressions. In the syntax, note the distinction between the colon ($:$) and the bar ($|$). We use the colon to draw elements from a sequence for mapping and we use the bar to select the elements that we wish to filter.

We can *map* and *filter* at the same time:

$$\begin{aligned} \langle e : x \in e_s \mid e_f \rangle &\equiv \text{map } (\lambda x. e) \\ &(\text{filter } (\lambda x. e_f) e_s). \end{aligned}$$

What appears before the colon (if any) is an expression to apply each element of the sequence to generate the result; what appears after the bar (if there is any) is an expression to apply to each element to decide whether to keep it.

Example 16.2. The expression

$$\langle x^2 : x \in a \rangle$$

is equivalent to

$$\text{map } (\lambda x. x^2) a.$$

Assuming $a = \langle 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 \rangle$ (from above), it evaluates to the sequence:

$$\langle 0, 1, 1, 4, 25, 64, 169, 441, 1156 \rangle.$$

Given the function $\text{isPrime } x$ which checks if x is prime, the expression

$$\langle x : x \in a \mid \text{isPrime } x \rangle$$

is equivalent to

$$\text{filter isPrime } a.$$

When evaluated, it returns the sequence $\langle 2, 5, 13 \rangle$.

5 Subsequences

Definition 16.12 (Subsequences). The $\text{subseq}(a, i, j)$ function extracts a contiguous subsequence of a starting at location i and with length j . If the subsequence is out of bounds of a , only the part within a is returned. We can specify subseq as follows

$$\begin{aligned} \text{subseq } (a : \mathbb{S}_\alpha) (i : \mathbb{N}) (j : \mathbb{N}) : \mathbb{S}_\alpha \\ = \{(k - i, x) : (k, x) \in a \mid i \leq k < i + j\}. \end{aligned}$$

We use the following syntax for denoting subsequences

$$a[e_i \cdots e_j] \equiv \text{subseq } (a, e_i, e_j - e_i + 1).$$

Splitting sequences. As we shall see in the rest of this book, many algorithms operate inductively on a sequence by splitting the sequence into parts, consisting for example, of the first element and the rest, a.k.a., the *head* and the *tail*, or the first half or the second half. We could define additional functions such as *splitHead*, *splitMid*, *take*, and *drop* for these purposes. Since all of these are easily expressible in terms of subsequences, we omit their discussion.

6 Append and Flatten

For constructing large sequences from smaller ones, the sequence ADT provides the functions *append* and *flatten*.

Definition 16.13 (Append). The function *append* (a, b) appends the sequence b after the sequence a . More precisely, we can specify *append* as follows

$$\begin{aligned} \text{append } (a : \mathbb{S}_\alpha) (b : \mathbb{S}_\alpha) &: \mathbb{S}_\alpha \\ &= a \cup \{(i + |a|, x) : (i, x) \in b\} \end{aligned}$$

We write $a ++ b$ as a short form for *append* a b .

Example 16.3 (Append). The *append* function

$$\langle 1, 2, 3 \rangle ++ \langle 4, 5 \rangle$$

yields

$$\langle 1, 2, 3, 4, 5 \rangle.$$

Definition 16.14 (Flatten). To append more than two sequences the *flatten* a function takes a sequence of sequences and flattens them. For the input is a sequence $a = \langle a_1, a_2, \dots, a_n \rangle$, *flatten* returns a sequence whole elements consists of those of all the a_i in order. We can specify *flatten* more precisely as follows

$$\begin{aligned} \text{flatten } (a : \mathbb{S}_{\mathbb{S}_\alpha}) &: \mathbb{S}_\alpha \\ &= \left\{ \left(i + \sum_{(k, c) \in a, k < j} |c|, x \right) : (i, x) \in b, (j, b) \in a \right\}. \end{aligned}$$

Example 16.4 (Flatten). The *flatten* function

$$\text{flatten } \langle \langle 1, 2, 3 \rangle, \langle 4 \rangle, \langle 5, 6 \rangle \rangle$$

yields

$$\langle 1, 2, 3, 4, 5, 6 \rangle.$$

7 Update and Inject

Definition 16.15 (Update). The function *update* ($a, (i, x)$), updates location i of sequence a to contain the value x . If the location is out of range for the sequence, the function returns the input sequence unchanged.

We specify *update* as follows

$$\begin{aligned} \text{update } (a : \mathbb{S}_\alpha) (i : \mathbb{N}, x : \alpha) : \mathbb{S}_\alpha \\ = \begin{cases} \{(j, y) : (j, y) \in a \mid j \neq i\} \cup \{(i, x)\} & \text{if } 0 \leq i < |a| \\ a & \text{otherwise.} \end{cases} \end{aligned}$$

Definition 16.16 (Inject). To update multiple positions at once, we can use *inject*. The function *inject* $a b$ takes a sequence b of position-value pairs and updates each position with its associated value. If a position is out of range, then the corresponding update is ignored. If multiple positions are the same, the first update in the ordering of b take effect. We define the *degree* of the update sequence b as the maximum number of updates that target any position.

Example 16.5 (Update and Inject). Given the string sequence

$$\begin{aligned} a = & \langle 'the', 'cat', 'in', 'the', 'hat' \rangle, \\ \text{update } a (1, 'rabbit') \end{aligned}$$

magically yields

$$\langle 'the', 'rabbit', 'in', 'the', 'hat' \rangle$$

since position 1 is updated with 'rabbit'. The expression

$$\text{inject } a \langle (4, 'log'), (1, 'dog'), (6, 'hog'), (4, 'bog'), (0, 'a') \rangle$$

yields

$$\langle 'a', 'dog', 'in', 'the', 'log' \rangle$$

because position 0 is updated with 'a', position 1 with 'dog', and position 4 with 'log' (the first of the two updates is applied). Because two updates target position 4 and at most 1 update targets all the other positions, the degree of the update sequence is 2.

Definition 16.17 (Nondeterministic Inject). To update multiple positions at once, we can also use nondeterministic inject *ninject*. The function *ninject* $a b$ takes a sequence b of position-value pairs and updates each position with its associated value. If a position is out of range, then the corresponding update is ignored. If multiple positions are the same, any one of the updates may take effect. The function *ninject* may thus treat duplicate updates non-deterministically. Because nondeterministic inject does not insist on determinism of updates, it may be implemented more efficiently and in lower span.

Example 16.6 (Nondeterministic Inject). Given the string sequence

$$a = \langle 'the', 'cat', 'in', 'the', 'hat' \rangle,$$

the expression

$$\text{ninject } a \langle (4, 'log'), (1, 'dog'), (6, 'hog'), (4, 'bog'), (0, 'a') \rangle$$

could yield

$\langle 'a', 'dog', 'in', 'the', 'log' \rangle$

since position 0 is updated with 'a', position 1 with 'dog', and position 4 with 'log' (the first of the two updates is applied). It could also yield

$\langle 'a', 'dog', 'in', 'the', 'log' \rangle$

The entry with position 6 is ignored since it is out of range for a .

8 Collect

Definition 16.18 (Collect). Given a sequence of *key-value* pairs, the operation *collect* “collects” together all the values for a given key. This operation is quite common in data processing, and in relational database languages such as SQL it is referred to as “Group by”. The signature of *collect* is

$\text{collect} : (\text{cmp} : \alpha \times \alpha \rightarrow \mathcal{O}) \rightarrow (a : \mathbb{S}_{\alpha \times \beta}) \rightarrow \mathbb{S}_{\alpha \times \mathbb{S}_{\beta}}$.

Here the “order set” $\mathcal{O} = \{\text{less}, \text{equal}, \text{greater}\}$.

The first argument *cmp* is a function for comparing keys of type α , and must define a total order over the keys. The second argument *a* is a sequence of key-value pairs. The *collect* function collects all values in *a* that share the same key together into a sequence, ordering the values in the same order as their appearance in the original sequence.

Example 16.7 (Collect). The following sequence consists of key-value pairs each of which represents a student and the classes that they take.

$kv = \langle ('jack', '15210'), ('jack', '15213'), ('mary', '15210'), ('mary', '15213'), ('mary', '15251'), ('peter', '15150'), ('peter', '15251'), \dots \rangle$.

We can determine the classes taken by each student by using *collect cmp*, where *cmp* is a comparison function for strings

$\text{collect cmp } kv = \langle ('jack', \langle '15210', '15213', \dots \rangle), ('mary', \langle '15210', '15213', '15251', \dots \rangle), ('peter', \langle '15150', '15251', \dots \rangle), \dots \rangle$.

Note that the output sequence is ordered based on the first instance of their key in the input sequences. Similarly, the order of the classes taken by each student are the same as in the input sequence.

9 Aggregation by Iteration

Iteration is a fundamental algorithm technique. It involves a sequence of steps, taken one after another, where each step transforms the state from the previous step. Iteration is an inherently sequential process.

Definition 16.19 (The *iterate* and *iteratePrefixes*). The function *iterate* iterates over a sequence while accumulating a “running sum”, i.e., a result that changes at each step. It starts with an initial result and a sequence, and on each step updates the result based on the next element of the sequence.

The function *iterate* has the type signature

$$\text{iterate } (f : \alpha \times \beta \rightarrow \alpha) (x : \alpha) (a : \mathbb{S}_\beta) : \alpha$$

where f is a function mapping a state and an element of a to a new state, x is the initial state, a is a sequence.

The semantics of *iterate* is defined as follows.

$$\text{iterate } f x a = \begin{cases} x & \text{if } |a| = 0 \\ \text{iterate } f (f(x, a[0])) (a[1 \dots |a| - 1]) & \text{otherwise.} \end{cases}$$

A variant of iteration, the function *iteratePrefixes* takes the same arguments as *iterate* but returns a pair, where the first component is a sequence consisting of all the intermediate result computed by iteration, up to and excluding the last element, and the second component is the final results. More precisely, *iteratePrefixes* can be specified as

$$\begin{aligned} \text{iteratePrefixes } f x a = \\ \text{let } g (b, x) y = (b++ x, f(x, y)) \\ \text{in } \text{iterate } g (\langle \rangle, x) a \text{ end} \end{aligned}$$

Example 16.8. The function *iterate* computes its final result by computing a result for each element of the sequence. Concretely, *iterate* $f x a$ computes the results x_i , $0 \leq i \leq n = |a|$, where

$$\begin{aligned} x_0 &= x \\ x_1 &= f(x_0, a[0]) \\ x_2 &= f(x_1, a[1]) \\ &\vdots \\ x_n &= f(x_{n-1}, a[n-1]). \end{aligned}$$

The expression

$$\text{iterate } f x a$$

thus evaluates to x_n .

The expression

iteratePrefixes f x a

performs the same computation and returns $(\langle x_0, \dots, x_{n-1} \rangle, x_n)$.

Example 16.9 (Iteration). For a sequence of length 5, iteration computes its final result as

iterate f x a = f(f(f(f(f(v, a[0]), a[1]), a[2]), a[3]), a[4]).

For example,

iterate +' 0 ⟨ 2, 5, 1, 6 ⟩

returns 14 since it starts with the integer state 0 and then one by one adds the integer elements 2, 5, 1 and 6 of the sequence to the state.

Similarly

iterate '-' 0 ⟨ 2, 5, 1, 6 ⟩

returns $((0 - 2) - 5) - 1 - 6 = -14$.

The function

iterate +' 0 (map zeroWhenEven a),

which uses the function *zeroWhenEven* to map even numbers to zero, sums up only the odd numbers in sequence *a*, returning 6

Exercise 16.1 (Rightmost Positive). Design an algorithm that, for each element in a sequence of integers, finds the rightmost positive number to its left. If there is no positive element to the left of an element, the algorithm returns $-\infty$ for that element.

For example, given the sequence

$\langle 1, 0, -1, 2, 3, 0, -5, 7 \rangle$

the algorithm would return

$\langle -\infty, 1, 1, 1, 2, 3, 3, 3 \rangle$.

Solution. Consider the function

```
extendPositive ((ℓ, b), x) =
  if x > 0 then
    (x, b++⟨ ℓ ⟩)
  else
    (ℓ, b++⟨ ℓ ⟩)
```

This function takes as its first argument the tuple consisting of ℓ , the last positive value seen (or $-\infty$) and a sequence *b*. The second argument *x* is a new element. The function

extends the sequence b with ℓ and returns as the most recently seen positive value x if is positive or ℓ otherwise.

Using this function, we can give an algorithm for the problem of selecting the rightmost positive number to the left of each element in a given sequence a :

```
let ( $\ell, b$ ) = iterate extendPositive  $(-\infty, \langle \rangle)$   $a$ 
in  $b$ 
```

We can solve the same problem more elegantly using *iteratePrefixes*. Consider the function

```
selectPositive ( $\ell, x$ ) =
  if  $x > 0$  then
     $x$ 
  else
     $\ell$ 
```

This function takes as argument ℓ , the last positive value seen, and x , the new element from the sequence. The function then returns x if is positive or ℓ otherwise. We can now we can give an algorithm for the problem of selecting the rightmost positive number preceding each element in a given sequence a as

```
let ( $\ell, b$ ) = iteratePrefixes selectPositive  $-\infty a$ 
in  $b$ 
```

Note (Iteration and order of operations). Iteration is a powerful technique but can be too big of a hammer, especially when used unnecessarily. For example, when summing the elements in a sequence, we don't need to perform the addition operations in a particular order because addition operations are associative and thus they can be performed in any order desired. The iteration-based algorithm for computing the sum does not take advantage of this property, computing instead the sum in a left-to-right order. As we will see next, we can take advantage of associativity to sum up the elements of a sequence in parallel.

10 Aggregation by Reduction

Reduction. The term *reduction* refers to a computation that repeatedly applies an associative binary operation to a collection of elements until the result is reduced to a single value. Recall that associative operations are defined as operations that allow commuting the order of operations.

Associativity.

Definition 16.20 (Associative Function). A function $f : \alpha \times \alpha \rightarrow \alpha$ is associative if $f(f(x, y), z) = f(x, f(y, z))$ for all x, y and z of type α .

Example 16.10. Many functions are associative.

- Addition and multiplication on natural numbers are associative, with 0 and 1 as their identities, respectively.
- Minimum and maximum are also associative with identities ∞ and $-\infty$ respectively.
- The *append* function on sequences is associative, with identity being the empty sequence.
- The union operation on sets is associative, with the empty set as the identity.

Note. Associativity implies that when applying f to some values, the order in which the applications are performed does not matter. Associativity does not mean that you can reorder the arguments to a function (that would be commutativity).

Important (Associativity of Floating Point Operations). Floating point operations are typically not associative, because performing them in different orders can lead to different results because of loss of precision.

Definition 16.21 (The *reduce* operation). In the sequence ADT, we use the function *reduce* to perform a reduction over a sequence by applying an associative binary operation to the elements of the sequence until the result is reduced to a single value. The operation function has the type signature

$\text{reduce } (f : \alpha \times \alpha \rightarrow \alpha) (id : \alpha) (a : \mathbb{S}_\alpha) : \alpha$

where f is an associative function, a is the sequence, and id is the *left identity* of f , i.e., $f(id, x) = x$ for all $x \in \alpha$.

When applied to an input sequence with a function f , *reduce* returns the “sum” with respect to f of the input sequence. In fact if f is associative this sum is equal to iteration. We can define the behavior of *reduce* inductively as follows

$$\text{reduce } f \text{ id } a = \begin{cases} id & \text{if } |a| = 0 \\ a[0] & \text{if } |a| = 1 \\ f \left(\text{reduce } f \text{ id } (a[0 \dots \lfloor \frac{|a|}{2} \rfloor - 1]), \right. \\ \left. \text{reduce } f \text{ id } (a[\lfloor \frac{|a|}{2} \rfloor \dots |a| - 1]) \right) & \text{otherwise.} \end{cases}$$

Example 16.11 (Reduce and append). The expression

$\text{reduce } \text{append } \langle \rangle \langle 'another', 'way', 'to', 'flatten' \rangle$

evaluates to

$'anotherwaytoflatten'.$

Important. The function *reduce* is more restrictive than *iterate* because it is the same function but with extra restrictions on its input (i.e. that f be associative, and id is a left identity). If the function f is associative, then we have

$\text{reduce } f \text{ id } a = \text{iterate } f \text{ id } a.$

Exercise 16.2. Give an example function f , a left identity x , and an input sequence a such that $\text{iterate } f x a$ and $\text{reduce } f x a$ return different results.

Important. Although we will use reduce only with associative functions, we define it for all well-typed functions. To deal properly with functions that are non-associative, the specification of reduce makes precise the order in which the argument function f is applied. For instance, when reducing with floating point addition or multiplication, we will need to take the order of operations into account. Because the specification defines the order in which the operations are applied, every (correct) implementation of reduce must return the same result: the result is deterministic regardless of the specifics of the algorithm used in the implementation.

Exercise 16.3. Given that reduce and iterate are equivalent for associative functions, why would we use reduce ?

Solution. Even though the input-output behavior of reduce and iterate may match, their cost specifications differ: unlike iterate , which is strictly sequential, reduce is parallel. In fact, as we will see in Chapter 18, the span of iterate is linear in the size of the input, whereas the span of reduce is logarithmic.

11 Aggregation with Scan

The scan function. When we restrict ourselves to associative functions, the input-output behavior of the function reduce can be defined in terms of the iterate . But the reverse is not true: iterate cannot always be defined in terms of reduce , because iterate can use the results of intermediate states computed on the prefixes of the sequence, whereas reduce cannot because such intermediate states are not available. We now describe a function called scan that allows using the results of intermediate computations and also does so in parallel.

Definition 16.22 (The functions scan and iScan). The term “scan” refers to a computation that reduces every prefix of a given sequence by repeatedly applying an associative binary operation. The scan function has the type signature

$$\text{scan } (f : \alpha * \alpha \rightarrow \alpha) (id : \alpha) (a : \mathbb{S}_\alpha) : (\mathbb{S}_\alpha * \alpha),$$

where f is an associative function, a is the sequence, and id is the left identity element of f .

The expression $\text{scan } f a$ evaluates to the cumulative “sum” with respect to f of all prefixes of the sequence a . For this reason, the scan function is referred to as *prefix sums*.

We specify the semantics of scan in terms of reduce as follows.

$$\begin{aligned} \text{scan } f id a = & \left(\left(\text{reduce } f id a[0 \dots (i-1)] : 0 \leq i < |a| \right), \right. \\ & \left. \text{reduce } f id a \right) \end{aligned}$$

For the definition, we assume that $a[0 \dots -1] = \langle \rangle$.

When computing the result for position i , $scan$ does not include the element of the input sequence at that position. It is sometimes useful to do so. To this end, we define $scanI$ ("I" stands for "inclusive").

We define the semantics of $scanI$ in terms of $reduce$ as follows.

$$scanI f id a = \langle \text{reduce } f id a[0 \dots i] : 0 \leq i < |a| \rangle$$

Example 16.12 (Scan). Consider the sequence $a = \langle 0, 1, 2 \rangle$. The prefixes of a are

- $\langle \rangle$
- $\langle 0 \rangle$
- $\langle 0, 1 \rangle$
- $\langle 0, 1, 2 \rangle$.

The prefixes of a sequence are all the subsequences of the sequence that starts at its beginning. Empty sequence is a prefix of any sequence. The computation $scan \text{ `+` } 0 \langle 0, 1, 2 \rangle$ can be written as

$$\begin{aligned} scan \text{ `+` } 0 \langle 0, 1, 2 \rangle &= (\langle \text{reduce } \text{`+`} 0 \langle \rangle, \\ &\quad \text{reduce } \text{`+`} 0 \langle 0 \rangle, \\ &\quad \text{reduce } \text{`+`} 0 \langle 0, 1 \rangle \\ &\quad \rangle, \\ &\quad \text{reduce } \text{`+`} 0 \langle 0, 1, 2 \rangle \\ &\quad) \\ &= (\langle 0, 0, 1 \rangle, 3). \end{aligned}$$

The computation $scanI \text{ `+` } 0 \langle 0, 1, 2 \rangle$ can be written as

$$\begin{aligned} scanI \text{ `+` } 0 \langle 0, 1, 2 \rangle &= \langle \text{reduce } \text{`+`} 0 \langle 0 \rangle, \\ &\quad \text{reduce } \text{`+`} 0 \langle 0, 1 \rangle, \\ &\quad \text{reduce } \text{`+`} 0 \langle 0, 1, 2 \rangle \rangle \\ &= \langle 0, 1, 3 \rangle. \end{aligned}$$

Note (Scan versus reduce). Since $scan$ can be specified in terms of $reduce$, one might be tempted to argue that it is redundant. In fact, it is not: as we shall see, performing $reduce$ repeatedly on every prefix is not work efficient. Remarkably $scan$ can be implemented by performing essentially the same work and span of $reduce$.

Example 16.13 (Copy scan). Scan is useful when we want pass information along the sequence. For example, suppose that we are given a sequence of type $\mathbb{S}_{\mathbb{N}}$ consisting only

of integers and asked to return a sequence of the same length where each element receives the previous positive value if any and $-\infty$ otherwise. For the example, for input $\langle 0, 7, 0, 0, 3, 0 \rangle$, the result should be $\langle -\infty, -\infty, 7, 7, 7, 3 \rangle$.

We considered this problem in [an example before](#) and presented an algorithm based on iteration. Because that algorithm uses iteration, it is sequential. The question is that is this necessary? Can we solve this problem without sacrificing parallelism.

We can solve this problem using inclusive scan *scan* if we can come up with a combining function f that does the crux of the work for us. Consider the function

$$\text{selectPositive}(x, y) = \text{if } y > 0 \text{ then } y \text{ else } x.$$

The function returns its right (second) argument if it is positive, otherwise it returns its the left (first) argument.

To be used in a scan, *selectPositive* must be associative. In particular we need to show that for all x, y and z , we have

$$\text{selectPositive}(x, \text{selectPositive}(y, z)) = \text{selectPositive}(\text{selectPositive}(x, y), z).$$

There are eight possibilities corresponding to each of x, y and z being either positive or not. For the cases where z is positive, it is easy to verify that either ordering returns z . For the cases that $z \leq 0$ and y is positive, it is likewise easy to verify that both orderings give y . Finally, for the cases that both $y, z \leq 0$, they both return x .

To use *selectPositive* as part of the scan function, we need to find its left identity. We can see that for any finite integer y

$$\text{selectPositive}(-\infty, y) = y.$$

Thus $-\infty$ is the left identity for *selectPositive*.

Remark (Reduce and scan). Experience in parallel computing shows that *reduce* and *scan* are powerful primitives that suffice to express many parallel algorithms on sequences. In some ways this is not surprising, because the functions allow using two important algorithm-design techniques: *reduce* function allows expressing divide-and-conquer algorithms and the *scan* function allows expressing iterative algorithms.

Chapter 17

Array Sequences

In Chapter 16, we specify the input-output behavior of the operations in the sequence ADT. In this chapter, we present an overview of how these operations can be implemented by using arrays.

1 A Parametric Implementation

The sequence ADT, as we described in Chapter 16, includes more than a dozen functions. Although it is possible to present an implementation by considering each function independently, it usually suffices to implement directly a smaller subset of the functions, which we can think of the *primitive functions*, and implement the rest of the functions in terms of the primitive ones.

In this section, we briefly describe how such an implementation could proceed based on the following primitive functions:

- *nth*,
- *length*,
- *subseq*,
- *tabulate*,
- *flatten*, and
- *inject* and *ninject*.

First, we present an implementation of the rest of the interface based on the primitive functions. We then describe in Section 1 how the implement the primitive functions.

Algorithm 17.1 (Function *empty*). We can implement the empty sequence *empty* directly in terms of *tabulate*:

$$\text{empty} = \text{tabulate} (\lambda i. i) 0.$$

Algorithm 17.2 (Function *singleton*). We can implement the function *singleton* directly in terms of *tabulate*:

$$\text{singleton } x = \text{tabulate} (\lambda i. x) 1.$$

Algorithm 17.3 (Function *map*). The function *map* is relatively easy to implement in terms of *tabulate*:

$$\text{map } f a = \text{tabulate} (\lambda i. f(a[i])) |a|.$$

Algorithm 17.4 (Function *append*). We can implement *append* directly in terms of *flatten*:

$$\text{append } a b = \text{flatten} \langle a, b \rangle.$$

We can also implement *append* by using *tabulate*. To this end, we first define a helper function:

$$\text{select } (a, b) i = \lambda i. \quad (17.1)$$

$$\quad \text{if } i < |a| \text{ then } a[i] \quad (17.2)$$

$$\quad \text{else } b[i - |a|]. \quad (17.3)$$

We can now state *append* as

$$\text{append } a b = \text{tabulate} (\text{select } (a, b)) (|a| + |b|).$$

Algorithm 17.5 (Function *filter*). We can implement *filter* by using a combination of *map* and *flatten*. The basic idea is to map the elements of the sequence for which the condition holds to singletons and map the other elements to empty sequences and then flatten.

We first define a function that “deflates” the elements for which the condition *f* does not hold:

$$\begin{aligned} \text{deflate } f x = \\ \quad \text{if } (f x) \text{ then } \langle x \rangle \\ \quad \text{else } \langle \rangle. \end{aligned}$$

We can now write *filter* as a relatively simple application of *flatten*, *map*, and *deflate*.

$$\begin{aligned} \text{filter } f a = \\ \quad \text{let } b = \text{map} (\text{deflate } f) a \\ \quad \text{in } \text{flatten } b \text{ end} \end{aligned}$$

Algorithm 17.6 (Function *update*). We can implement the function *update* in terms of *tabulate* as

```
update a (i, x) =
  tabulate (lambda j. if i = j then x else a[i])
  |a|
```

Algorithm 17.7 (Functions *isEmpty* and *isSingleton*). Emptiness and singleton checks are simple by using the *length* function:

```
isEmpty a =
  |a| = 0

isSingleton a =
  |a| = 1
```

Algorithm 17.8 (Functions *iterate*). We can implement iteration by simply iterating over the sequence from left to right.

```
iterate f x a =
  if |a| = 0 then
    x
  else if |a| = 1 then
    f(x, a[0])
  else
    iterate f (f(x, a[0])) a[1 ... |a| - 1]
```

Algorithm 17.9 (Functions *reduce*). We can implement *reduce* by using a divide-and-conquer strategy.

```
reduce f id a =
  if |a| = 0 then
    id
  else if |a| = 1 then
    a[0]
  else
    let
      mid = floor(|a|/2)
      (b, c) = (a[0 ... mid - 1], a[mid ... |a| - 1])
      (rb, rc) = (reduce f id b) || (reduce f id c)
    in
      f(rb, rc)
    end
```

Algorithm 17.10 (Scan Using Contraction). We will cover scan in more detail in Section 3. But for completeness, we present an implementation below. For simplicity, we assume that the length of the sequence is a power of two, though this is not difficult to eliminate.

```

(* Assumption: |a| is a power of two.  *)
scan f id a =
  case |a|
  | 0 => (< >, id)
  | 1 => (< id >, a[0])
  | n =>
    let
      a' = < f(a[2i], a[2i + 1]) : 0 ≤ i < n/2 >
      (r, t) = scan f id a'
    in
      (< pi : 0 ≤ i < n >, t), where pi = { r[i/2]           even(i)
                                         f(r[i/2], a[i - 1]) otherwise
    end

```

2 Implementing the Primitive Functions

To support the primitive operations efficiently we represent a sequence as an array segment (a.k.a., slice) contained within a possibly larger array along with a few additional pieces of information. More precisely, a sequence is represented as:

- an array of elements contains the elements in the sequence (but possibly more)
- a “left” and a “right” position indicates the boundaries of the slice in terms of the beginning and the ending of a contiguous section of the array that contains the elements in the sequence (in order),

Algorithm 17.11 (Function *nth*). Using arrays, indexing into any location requires a simple array access and can be achieved in constant work and span.

Algorithm 17.12 (Function *length*). Because we know the boundary positions of the array slice that corresponds to the sequence, we can calculate length by using simple arithmetic in constant work and span.

Algorithm 17.13 (Function *subseq*). Taking a subsequence of a sequence requires determining the boundaries for the new slice. We do not need to copy the elements of the array within the boundary. This operation therefore requires basic arithmetic and thus can be done in constant work and span.

Algorithm 17.14 (Function *tabulate*). Consider a call to tabulate of the form:

tabulate f n.

To construct the sequence of length *n*, we allocate a fresh array of *n* elements, evaluate *f* at each position *i* and write the result into position *i* of the array.

Because the function f can be evaluated at each element independently in parallel, this operation has the same span and that of the function f itself (maximized over all positions) and the total work is the sum of the work required to evaluate f at each position.

Algorithm 17.15 (Function *flatten*). Consider a call to *flatten* of the form:

flatten a,

where a is a sequences of sequences.

To compute the resulting sequence, we first map each element of a to its length; let ℓ be the resulting sequence. We then perform the scan $scan + 0 \ell$. This computation returns for each element of a its position in the result sequence of *flatten*. Finally, we allocate an array that can hold all of the elements of the sequences in a and write each element of a into its corresponding segment in parallel.

This cost of *scan* is $O(|a|)$ work and $O(\lg |a|)$ span. The cost of the final write of each element requires $O(\|a\|)$ work, where $\|a\| = \sum_{i=0}^{|a|-1} |a[i]|$ and constant span. Thus the total work is $O(|a| + \|a\|)$ and span is $O(\lg |a|)$.

Algorithm 17.16 (Function *inject*). Consider a call to *inject* of the form:

inject a b,

where a is a sequence of length n and b is a sequence of m updates.

To inject the updates into the sequence, we create a new array aa from a , where for all $0 \leq i < |a|$, $aa[i] = (a[i], |a|)$. We then “inject” all updates in b into aa in parallel. To handle each update of the form (j, v) at position k , we perform an atomic update operation that proceeds as follows.

```
atomicWrite aa b k =
  atomically do:
    (j, v) ← b[k]
    (w, i) ← aa[j]
    if k < i then
      aa[j] ← (v, k)
```

This update operation guarantees that of the possibly many conflicting operations the first (leftmost) one in the update sequence wins and is transferred to the result; the other updates either don’t take place or are overwritten. After all updates complete, we take aa and create another array consisting of only the value component of each element.

Under the assumption that all updates occur uniformly randomly, it is possible to prove that the number of updates to a position that is targeted by d updates is $O(\lg d)$ in expectation and the total work is $O(n + m)$. Note that $d \leq m$ and therefore $O(\lg m)$ span is also a decent upper bound but as we will see in many cases d , which is the degree, is constant.

Algorithm 17.17 (Function *ninject*). Consider a call to *ninject* of the form:

ninject a b,

where a is a sequence to be injected into and b is the updates.

To inject the updates into the sequence, we make a copy of the array a and then use an atomic write operation to write or “inject” each update independently in parallel. Each instance of the atomic write operation updates the relevant element of the copied array atomically. Assuming that each atomic write operation requires constant-work, this implementation requires linear work in the number of updates and constant span.

Remark (Benign Effects). We made careful use of side effects (memory updates) in implementing the primitive functions. Because these side effects are not visible to the programmer, the ADT remains to be purely functional and thus is safe for parallelism. Effects such as these that have no impact on purity are sometimes called *benign effects*.

Chapter 18

Cost of Sequences

In this chapter, we present several cost specifications for the Chapter 16. These cost specifications pertain to implementations that use several common representations for sequences based on [arrays](#), [trees](#), and [lists](#).

The cost specifications do not require describing the particular implementations, but implementations that match the given costs indeed use data structures based on arrays, trees, and lists (respectively). However, for example, there might many tree implementations that match the tree cost specification.

All of the the cost bounds we give are based on “pure” implementations, as discussed in Chapter 8. In particular all functions create new data without changing the old data. In the case of the array cost specification, this means that updates are expensive since they need to copy the array. In Chapter 20 we discuss both the “impure” costs and an interface that remains pure, but reduces the cost to the same as the “impure” case when used in a particular way.

1 Cost Specifications

Cost specifications describe the cost—in terms of work and span—of the functions in an ADT. Typically many specific implementations match a specific cost specification. For example, for the tree-sequence specification for sequences (Section 3, an implementation can use one of many balanced binary tree data structures available.

To use a cost specification, we don’t need to know the details of how these implementations work. Cost specifications can thus be viewed as an abstraction over implementation details that do not matter for the purposes of the algorithm.

Note. Cost specifications are similar to prices on restaurant menus. If we view the functions

of the ADT as the dishes in a menu, then the cost specification is the price tag for each dish. Just as the cost of the dishes in a menu does not change from day to day as the specific details of the preparation process changes (e.g., different cooks may prepare the dish, the origin of the ingredients may vary from one day to the next), cost specifications offer a layer of abstraction over implementation details that a client of the ADT need not know.

Definition 18.1 (Domination of cost specifications). There are usually multiple ways to implement an ADT and thus there can be multiple cost specifications for the same ADT. We say that one cost specification *dominates* another if for each and every function, its asymptotic costs are no higher than those of the latter.

Example 18.1. Of the three cost specifications that we consider in this chapter, none dominates another. The list-based cost specification, however, is almost dominated by the others, because it is nearly completely sequential.

Choosing cost specifications. When deciding which of the possibly many cost specification to use for a particular ADT, we usually notice that there are certain trade-offs: some functions will be cheaper in one and while others are cheaper in another. In such cases, we choose the cost specification that minimizes the cost for the algorithm that we wish to analyze. After we decide the specification to use, what remains is to select the implementation that matches the specification, which can include additional considerations.

Example 18.2. If an algorithm makes many calls to *nth* but no calls to *append*, then we would use the array-sequence specification rather than the tree-sequence specification, because in the former *nth* requires constant work whereas in the latter it requires logarithmic work. Conversely, if the algorithm mostly uses *append* and *update*, then tree-sequence specification would be better.

Note. Following on our restaurant analogy, suppose that you wish to enjoy a nice three course meal in a nearby restaurant. Looking over the menues of the restaurants in your price range, you might realize that prices for the appetizers in one are lower than others but the main dishes are more expensive. (If that were not the case, the others would be dominated and, assuming equal quality, they would likely go out of business.) Assuming your goal is to minimize the total cost of your meal, you would therefore sum up the cost for the dishes that you plan on enjoying and make your decision based on the total sum.

2 Array Sequences

Cost Specification 18.2 (Array Sequences). The table below specifies the *array-sequence* costs. For the cost of *inject*, we define the degree of an update sequence as the maximum number of updates targeting the same position. The notation $T(-)$ refer to the trace of the corresponding operation. The specification for *scan* assumes that f has constant work and span.

Operation	Work	Span
<i>length a</i>	1	1
<i>nth a i</i>	1	1
<i>singleton x</i>	1	1
<i>empty</i>	1	1
<i>isSingleton x</i>	1	1
<i>isEmpty x</i>	1	1
<i>tabulate f n</i>	$1 + \sum_{i=0}^n W(f(i))$	$1 + \max_{i=0}^n S(f(i))$
<i>map f a</i>	$1 + \sum_{x \in a} W(f(x))$	$1 + \max_{x \in a} S(f(x))$
<i>filter f a</i>	$1 + \sum_{x \in a} W(f(x))$	$\lg a + \max_{x \in a} S(f(x))$
<i>subseq a (i, j)</i>	1	1
<i>append a b</i>	$1 + a + b $	1
<i>flatten a</i>	$1 + a + \sum_{x \in a} x $	$1 + \lg a $
<i>update a (i, x)</i>	$1 + a $	1
<i>inject a b</i>	$1 + a + b $	$\lg(\text{degree}(b))$
<i>ninject a b</i>	$1 + a + b $	1
<i>collect f a</i>	$1 + W(f) \cdot a \lg a $	$1 + S(f) \cdot \lg^2 a $
<i>iterate f x a</i>	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
<i>reduce f x a</i>	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$\lg a \cdot \max_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
<i>scan f x a</i>	$ a $	$\lg a $

The functions *length*, *nth*, *empty*, *singleton*, *isEmpty*, *isSingleton*, *subseq*, all require constant work and span.

Tabulate and Map. The functions *tabulate* and *map* total work that is equal to the sum of the work of applying *f* at each position, as well as an additional unit cost to account for *tabulate* or *map* itself.

Because it is possible to apply the function *f* in parallel—there are no dependencies among the different positions, the span is the maximum of the span of applying *f* at each position, plus 1 for the function call itself.

Example 18.3 (Tabulate and map with array costs). As an example of *tabulate* and *map*

$$\begin{aligned} W(\langle i^2 : 0 \leq i < n \rangle) &= O\left(1 + \sum_{i=1}^{n-1} O(1)\right) = O(n) \\ S(\langle i^2 : 0 \leq i < n \rangle) &= O\left(1 + \max_{i=0}^{n-1} O(1)\right) = O(1) \end{aligned}$$

because the work and span for i^2 is $O(1)$.

Filter. The work for the function *filter* is equal to the sum of the work of applying f at each position, as well as an additional unit cost, for the function call itself.

Because it is possible to apply the function f in parallel—there are no dependencies among the different positions, the span is the maximum of the span of applying f at each position, plus a logarithmic term for performing *compaction*, i.e., packing the chosen elements contiguously into the result array.

Example 18.4 (Filter). As an example of *filter*, we have

$$\begin{aligned} W(\langle x : x \in a \mid x < 27 \rangle) &= O\left(1 + \sum_{i=0}^{|a|-1} O(1)\right) = O(|a|) \\ S(\langle x : x \in a \mid x < 27 \rangle) &= O\left(\lg |a| + \max_{i=0}^{|a|-1} O(1)\right) = O(\lg |a|). \end{aligned}$$

The operation *append* requires work proportional to the length of the sequences given as input, can be implemented in constant span.

The operation *flatten* generalizes *append*, requiring work proportional to the total length of the sequences flattened, and can be implemented in parallel in logarithmic span in the number of sequences flattened.

Update and Inject. The operations *update* and *inject* both require work proportional to the length of the sequences they are given as input. It might seem surprising that *update* takes work proportional to the size of the input sequence a , since updating a single element should require constant work. The reason is that the interface is purely functional so that the input sequence needs to be copied—we are not allowed to update the old copy.

The function *update* and non-deterministic *inject* can be implemented in constant span, but deterministic *inject* required resolving conflicts more carefully and requires $O(\lg(\text{degree}(b)))$ span where the degree of the update sequence b is the maximum number of updates targeting the same position in the sequence being updated.

In the last section of this chapter, we describe single-threaded array sequences that allows updating under a sequence in constant work, but under certain restrictions.

Collect. The primary cost in implementing *collect* is a sorting step that sorts the sequence based on the keys. The work and span of *collect* is therefore determined by the work and span of (comparison) sorting with the specified comparison function f .

Cost of aggregation. The cost of aggregation functions, *iterate*, *reduce*, and *scan* are more difficult to specify, because they depend their arguments and on the intermediate values computed during evaluation.

Example 18.5 (Cost of Iterated *append*). Consider appending the following sequence of strings using *iterate*:

iterate append '' $\langle 'abc', 'd', 'e', 'f' \rangle$.

If we only count the work of *append* functions performed during evaluation, we obtain a total work of 22, because the following *append* functions are performed

1. *append* '' '' abc (work 4),
2. *append* '' abc '' d (work 5),
3. *append* '' $abcd$ '' e (work 6), and
4. *append* '' $abcde$ '' f (work 7).

Consider now appending the following sequence of strings, which is a permutation of the previous, using *iterate*:

iterate append '' $\langle 'd', 'e', 'f', 'abc' \rangle$

If we only count the work of *append* operations using the array-sequence specification, we obtain a total work of 16, because the following *append* operations are performed

1. *append* '' '' d (work 2),
2. *append* '' d '' e (work 3),
3. *append* '' de '' f (work 4) and
4. *append* '' def '' abc (work 7).

Thus, we have used iteration over two sequences, both with 4 elements, and obtained different costs even though the sequences are permutations of each other. The reason for this is that the total cost depends on the intermediate values generated during computation.

Specification of *iterate*. To specify the cost of *iterate*, we consider the intermediate values generated by an evaluation of *iterate*, whose specification, originally given in Section 9 is reproduced here for convenience.

$$\text{iterate } f \ x \ a = \begin{cases} x & \text{if } |a| = 0 \\ \text{iterate } f \ (f(x, a[0])) \ (a[1 \cdots |a| - 1]) & \text{otherwise.} \end{cases}$$

Consider an evaluation of

iterate f v a

and let

$\mathcal{T}(\text{iterate } f \text{ } v \text{ } a)$

denote the set of calls to $f(\cdot, \cdot)$ performed along with the arguments, as defined by the specification above. We refer to this set of function calls as the *trace* of *iterate* and define the cost of *iterate* as the sum of these calls.

Cost Specification 18.3 (Cost for *iterate*). Consider evaluation of *iterate f v a* and let $\mathcal{T}(\text{iterate } f \text{ } v \text{ } a)$ denote the set of calls (trace) to $f(\cdot, \cdot)$ performed along with the arguments. The work and span are as follows.

$$\begin{aligned} W(\text{iterate } f \text{ } x \text{ } a) &= O\left(1 + \sum_{f(y,z) \in \mathcal{T}(\text{iterate } f \text{ } x \text{ } a)} W(f(y,z))\right) \\ S(\text{iterate } f \text{ } x \text{ } a) &= O\left(1 + \sum_{f(y,z) \in \mathcal{T}(\text{iterate } f \text{ } x \text{ } a)} S(f(y,z))\right) \end{aligned}$$

Example 18.6 (Sorting by Iteration). As an interesting example, consider the function *mergeOne a x* for merging a sequence *a* with the singleton sequence $\langle x \rangle$ by using an assumed comparison function. The function performs $O(n)$ work in $O(\lg n)$ span, where *n* is the total number of elements in the output sequence. We can use the *mergeOne* function to sort a sequence via iteration as follows

iterSort a = iterate mergeOne ⟨ ⟩ a.

For example, on input $a = \langle 2, 1, 0 \rangle$, *iterSort* first merges $\langle \rangle$ and $\langle 2 \rangle$, then merges the result $\langle 2 \rangle$ with $\langle 1 \rangle$, then merges the resulting sequence $\langle 1, 2 \rangle$ with $\langle 0 \rangle$ to obtain the final result $\langle 0, 1, 2 \rangle$.

The trace for *iterSort* with an input sequence of length *n* consists of a set of calls to *mergeOne*, where the first argument is a sequence of sizes varying from 1 to $n - 1$, while its right argument is always a singleton sequence. For example, the final *mergeOne* merges the first $(n - 1)$ elements with the last element, the second to last *mergeOne* merges the first $(n - 2)$ elements with the second to last element, and so on. Therefore, the total work for an input sequence *a* of length *n* is

$$W(\text{iterSort } a) \leq \sum_{i=1}^{n-1} c \cdot (1 + i) = O(n^2).$$

Using the trace, we can also analyze the span of *iterSort*. Since we iterate adding in each element after the previous, there is no parallelism between merges, but there is parallelism within a *mergeOne*, whose span is logarithmic. We can calculate the total span as

$$S(\text{iterSort } a) \leq \sum_{i=1}^{n-1} c \cdot \lg(1 + i) = O(n \lg n).$$

Since average parallelism, $W(n)/S(n) = O(n/\lg n)$, we see that the algorithm has a reasonable amount of parallelism. Unfortunately, it does much too much work.

Note (Algorithm *iterSort*). Using this reduction order the algorithm is effectively working from the front to the rear, using *mergeOne* to “insert” each element into a sorted prefix where it is placed at the correct location to maintain the sorted order. The algorithm thus implements the well-known insertion sort.

Cost of *reduce*. Recall that with *reduce*, we noted that the result of the computation is not affected by the order in which the associative function is applied and in fact is the same as that of performing the same computation with *iterate*. The cost of *reduce*, however, depends on the order in which the operations are performed.

Example 18.7 (Cost of *reduce* append). Consider appending the following code

reduce append ‘ ‘ (‘abc’, ‘d’, ‘e’, ‘f’).

Suppose performing append operations in left-to-right order and count their work using the array-sequence specification. The total work is 19, because the following *append* operations are performed

1. *append* ‘abc’ ‘d’ (work 5),
2. *append* ‘abcd’ ‘e’ (work 6), and
3. *append* ‘abcde’ ‘f’ (work 7).

Consider now performing the *append* operations from right to left order. We obtain a total cost of 15, because the following *append* operations are performed

1. *append* ‘e’ ‘f’ (work 3),
2. *append* ‘d’ ‘ef’ (work 4) and
3. *append* ‘abc’ ‘def’ (work 7).

Specification of *reduce*. To specify the cost of *reduce*, we consider its trace based on its specification, as given in Section 10 reproduced below for convenience.

$$\text{reduce } f \text{ id } a = \begin{cases} id & \text{if } |a| = 0 \\ a[0] & \text{if } |a| = 1 \\ f \left(\text{reduce } f \text{ id } (a[0 \dots \lfloor \frac{|a|}{2} \rfloor - 1]), \right. \\ \left. \text{reduce } f \text{ id } (a[\lfloor \frac{|a|}{2} \rfloor \dots |a| - 1]) \right) & \text{otherwise.} \end{cases}$$

Cost Specification 18.4 (Cost of *reduce*). Consider evaluation of $\text{reduce } f \ x \ a$ and let $\mathcal{T}(\text{reduce } f \ x \ a)$ denote the set of calls to $f(\cdot, \cdot)$ performed along with the arguments. The work and span are defined as

$$\begin{aligned} W(\text{reduce } f \ x \ a) &= O\left(1 + \sum_{f(y,z) \in \mathcal{T}(\text{reduce } f \ x \ a)} W(f(y,z))\right), \text{ and} \\ S(\text{reduce } f \ x \ a) &= O\left(\lg |a| \cdot \max_{f(y,z) \in \mathcal{T}(\text{reduce } f \ x \ a)} S(f(y,z))\right). \end{aligned}$$

Work and Span of *reduce*. The work bound is simply the total work performed, which we obtain by summing across all combine functions, plus one for the *reduce*. The span bound is more interesting. The $\lg |a|$ term expresses the fact that the recursion tree in the specification of *reduce* is at most $O(\lg |a|)$ deep. Since each node in the recursion tree has span at most $\max_{f(y,z)} S(f(y,z))$, any root-to-leaf path, has at most $O(\lg |a| \cdot \max_{f(a,b)} S(f(a,b)))$ span.

Cost of *scan*. As in *iterate* and *reduce* the cost specification of *scan* depends on the intermediate results. But the dependency is more complex than can be represented by our ADT specification. For *scan*, we will stop at giving a cost specification by assuming that the function that we are scanning with performs $O(1)$ work and span.

Cost Specification 18.5 (Cost for *scan*). Consider the expression $\text{scan } f \ x \ a$, where $f(\cdot, \cdot)$ always requires $O(1)$ work and span. The work and span of the expression are defined as

$$\begin{aligned} W(\text{scan } f \ x \ a) &= O(|a|), \text{ and} \\ S(\text{scan } f \ x \ a) &= O(\lg |a|). \end{aligned}$$

3 Tree Sequences

The costs for tree sequences is given in [Cost Specification below](#). The specification represents the cost for a class of implementations that use a balanced tree to represent the sequence. The cost of each operation is similar to the array-based specification, and many are exactly the same, i.e., *length*, *singleton*, *isSingleton*, *isEmpty*, *collect*, *iterate*, *reduce*, and *scan*.

There are also differences. The work and span of the operation *nth* is logarithmic, as opposed to being constant. This is because in balanced-tree based implementation, the operation must follow a path from the root to a leaf to find the desired element element. For a sequence a , such a path has length $O(\lg |a|)$. Although *nth* does more work with tree sequences, *append* does less work. Instead of requiring linear work, the work of *append* with tree sequences is proportional to the logarithm of the ratio of the size of the larger sequence to the size of the smaller one smaller one. For example if the two sequences are

the same size, then *append* takes $O(1)$ work. On the other hand if one is length n and the other 1, then the work is $O(\lg n)$. The work of *update* is also less with tree sequences than with array sequences.

The work for operations *map* and *tabulate* are the same as those for array sequences; their span incurs an extra logarithmic overhead. The work and span of *filter* are the same for both.

Cost Specification 18.6 (Tree Sequences). We specify the *tree-sequence* costs as follows. The notation $\mathcal{T}(-)$ refer to the trace of the corresponding operation. The specification for *scan* assumes that f has constant work and span.

Operation	Work	Span
<i>length a</i>	1	1
<i>singleton x</i>	1	1
<i>isSingleton x</i>	1	1
<i>isEmpty x</i>	1	1
<i>nth a i</i>	$\lg a $	$\lg a $
<i>tabulate f n</i>	$1 + \sum_{i=0}^n W(f(i))$	$1 + \lg n + \max_{i=0}^n S(f(i))$
<i>map f a</i>	$1 + \sum_{x \in a} W(f(x))$	$1 + \lg a + \max_{x \in a} S(f(x))$
<i>filter f a</i>	$1 + \sum_{x \in a} W(f(x))$	$1 + \lg a + \max_{x \in a} S(f(x))$
<i>subseq(a, i, j)</i>	$1 + \lg(a)$	$1 + \lg(a)$
<i>append a b</i>	$1 + \lg(a / b) $	$1 + \lg(a / b) $
<i>flatten a</i>	$1 + a \lg (\sum_{x \in a} x)$	$1 + \lg(a + \sum_{x \in a} x)$
<i>inject a b</i>	$1 + (a + b) \lg a $	$1 + \lg(a + b)$
<i>ninject a b</i>	$1 + (a + b) \lg a $	$1 + \lg(a + b)$
<i>collect f a</i>	$1 + W(f) \cdot a \lg a $	$1 + S(f) \cdot \lg^2 a $
<i>iterate f x a</i>	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
<i>reduce f x a</i>	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$\lg a \cdot \max_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
<i>scan f x a</i>	$ a $	$\lg a $

4 List Sequences

The Cost Specification below defines the cost for list sequences. The specification represents the cost for a class of implementations that use (linked) lists to represent the sequence. The determining cost in list-based implementations is the sequential nature of the representation: accessing the element at position i requires traversing the list from the head to i ,

which leads to $O(i)$ work and span. List-based implementations therefore expose hardly any parallelism. Their main advantage is that they require quick access to the *head* and the *tail* of the sequence, which are defined as the first element and the suffix of the sequence that starts at the second element respectively.

The work of each operation is similar to the array-based specification. Since the data structure mostly serial, the span of each operation is essentially the same as that of its work, except that the total is taken over the spans of its components. The work and span of *subseq* operation depends on the beginning position of the subsequence, because list-based representation can share their suffixes.

Cost Specification 18.7 (List Sequences). We specify the *list-sequence* costs as follows. The notation $\mathcal{T}(-)$ refer to the trace of the corresponding operation. The specification for *scan* assumes that f has constant work and span.

Operation	Work	Span
<i>length a</i>	1	1
<i>singleton x</i>	1	1
<i>isSingleton x</i>	1	1
<i>isEmpty x</i>	1	1
<i>nth a i</i>	i	i
<i>tabulate f n</i>	$1 + \sum_{i=0}^n W(f(i))$	$1 + \sum_{i=0}^n S(f(i))$
<i>map f a</i>	$1 + \sum_{x \in a} W(f(x))$	$1 + \sum_{x \in a} S(f(x))$
<i>filter f a</i>	$1 + \sum_{x \in a} W(p(x))$	$1 + \sum_{x \in a} S(p(x))$
<i>subseq a (i, j)</i>	$1 + i$	$1 + i$
<i>append a b</i>	$1 + a $	$1 + a $
<i>flatten a</i>	$1 + a + \sum_{x \in a} x $	$1 + a + \sum_{x \in a} x $
<i>update a (i, x)</i>	$1 + a $	$1 + a $
<i>inject a b</i>	$1 + a + b $	$1 + a + b $
<i>ninject a b</i>	$1 + a + b $	$1 + a + b $
<i>collect f a</i>	$1 + W(f) \cdot a \lg a $	$1 + S(f) \cdot a \lg a $
<i>iterate f x a</i>	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
<i>reduce f x a</i>	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
<i>scan f a</i>	$ a $	$ a $

Remark. Since they are serial, list-based sequences are usually ineffective for parallel algorithm design.

Chapter 19

Examples

This chapter presents example use of the sequence ADT (Chapter 16) and its cost specification (Chapter 18). The examples include several solutions to a classic problem in computer science, that of computing prime numbers. In addition, we also briefly discuss how we might generalize the comprehensions based notation introduced in Chapter 16 for building sequences to operate on multiple domains (or variables).

1 Miscellaneous Examples

Problem 19.1 (Points in 2D). We wish to create a sequence consisting of points in two dimensional space (x, y) whose coordinates are natural numbers that satisfy the conditions:

$$0 \leq x < n \text{ and } 1 \leq y < n.$$

For example, for $n = 3$, we would like to construct the sequence

$$\langle (0, 1), (0, 2), (1, 1), (1, 2), (2, 1), (2, 2) \rangle.$$

Give an algorithm that solves this problem and write the algorithm in both plain SPARC notation and in notation using comprehensions.

Algorithm 19.1 (2D Points). The following algorithm generates the desired point sequence:

```
points2D n =  
  flatten (tabulate (lambda x . tabulate (lambda y . (x, y + 1))  
                     (n - 1))  
           n).
```

Using the basic sequence comprehension notation Syntax 16.2, we can express the same code more succinctly as

$$\text{points2D } n = \text{flatten } \langle \langle (x, y) : 1 \leq y < n \rangle : 0 \leq x < n \rangle.$$

By extending the comprehension notation slightly to allow for multiple variables, we express the same algorithm as

$$\text{points2D } n = \langle (x, y) : 1 \leq y < n, 0 \leq x < n \rangle.$$

Example 19.1. For a given n , the algorithm first generates a sequence of the form

$$\langle \langle (0, 1), (0, 2), \dots, (0, n-1) \rangle \rangle \quad (19.1)$$

$$\langle (1, 1), (1, 2), \dots, (1, n-1) \rangle \quad (19.2)$$

$$\vdots \quad (19.3)$$

$$\langle (n-1, 1), (n-1, 2), \dots, (n-1, n-1) \rangle \quad (19.4)$$

$$\rangle \quad (19.5)$$

and then concatenates the inner sequences by using *flatten*.

Important. Note that in the second “multivariate comprehensions” approach used in the [above algorithm](#), *flatten* is hidden. When using such multivariate notation, we therefore have to account for the cost of *flatten*.

Exercise 19.2 (Analyzing 2D Points). Analyze the cost of the algorithm for generating 2D points [given above](#).

Problem 19.3 (Points in 3D). Building on Problem 19.1, suppose that we wish to generate points in 3D and restrict the points to those whose coordinates (x, y, z) are natural numbers that satisfy the conditions that

$$0 \leq x \leq n-1, 1 \leq y \leq n, \text{ and } 2 \leq z \leq n+1.$$

Give an algorithm for solving this problem, using both the plain SPARC notation and sequence comprehensions.

Algorithm 19.2. Using the basic sequence comprehension notation Syntax 16.2, we can write the code for computing the sequence of all such points as

$$\begin{aligned} \text{points3D } n = \\ \text{flatten } \langle \text{flatten } \langle \langle (x, y, z) : 2 \leq z \leq n+1 \rangle : 1 \leq y \leq n \rangle : 0 \leq x \leq n-1 \rangle. \end{aligned}$$

We can also write the same algorithm more succinctly as

$$\begin{aligned} \text{points3D } n = \\ \langle (x, y, z) : 0 \leq x \leq n-1, 1 \leq y \leq n, 2 \leq z \leq n+1 \rangle. \end{aligned}$$

As in Algorithm 19.1, the multivariate notation results in a succinct algorithm but reasoning about the cost of the algorithm requires translating the algorithm to the more basic notation so that hidden *flatten*’s can be accounted for.

Problem 19.4 (Cartesian Product). Present an algorithm that returns the Cartesian product of two sequences. For example, for the sequences $a = \langle 1, 2 \rangle$, and $b = \langle 3.0, 4.0, 5.0 \rangle$, the algorithm should return the Cartesian product of a and b , which is

$$a \times b = \langle (1, 3.0), (1, 4.0), (1, 5.0), (2, 3.0), (2, 4.0), (2, 5.0) \rangle.$$

Algorithm 19.3 (Cartesian Product). Using the basic sequence comprehension notation Syntax 16.2, we can write the algorithm for computing the Cartesian Product of two sequences as

$$\begin{aligned} \text{CartesianProduct} (a, b) = \\ \text{flatten} (\text{map} (\lambda x . \text{map} (\lambda y . (x, y)) b) a). \end{aligned}$$

Using the multivariate sequence comprehension notation, we can write the same algorithm as

$$\begin{aligned} \text{CartesianProduct} (a, b) = \\ \langle (x, y) : x \in a, y \in b \rangle. \end{aligned}$$

Exercise 19.5 (All contiguous subsequences). Present an algorithm that generates all contiguous subsequences of a given sequence.

Solution. The following algorithm finds all contiguous subsequences of sequence a :

$$\langle a \langle i, \dots, j \rangle : 0 \leq i < |a|, i \leq j < |a| \rangle.$$

The same algorithm can be written more simply as

$$\langle a \langle i, \dots, j \rangle : 0 \leq i \leq j < |a| \rangle.$$

This is equivalent, in simplified sequence comprehensions notation, to

$$\text{flatten} \langle \langle a[i \dots i + j] : i \leq j < |a| \rangle : 0 \leq i < |a| \rangle$$

We can further translate this to

$$\text{flatten} (\text{tabulate} (\lambda i . \text{tabulate} (\lambda j . a[i \dots i + j])) \quad (19.6)$$

$$(|a| - i - 1) \quad (19.7)$$

$$|a|). \quad (19.8)$$

Exercise 19.6 (Cost Analysis of All Subsequences). Analyze the cost of your algorithm for Exercise 19.5.

Solution. Consider the algorithm from Solution 1:

$$\langle a[i \dots j] : 0 \leq i < |a|, i \leq j < |a| \rangle,$$

which extracts all contiguous subsequences from the sequence a .

Recall that the notation is equivalent to a nested *tabulate* first over the indices i , and then inside over the indices j . The results are then *flatten*'ed. The nesting of *tabulate*'s allows all the calls to $a[i \dots j]$ (i.e., *subseq*) to run in parallel. Let $n = |a|$. There are a total of

$$\sum_{i=1}^n i = n(n+1)/2 = O(n^2)$$

contiguous subsequences and hence that many calls to *subseq*, each of which has constant work and span according to the cost specifications. The work of the nested *tabulate*'s and the *subseq*'s is therefore $O(n^2)$. The span of the inner *tabulate* is maximum over the span of the inner *subseq*'s, which is $O(1)$. The span of the outer *tabulate* is the maximum over the inner *tabulate*'s, which is again $O(1)$. The *flatten* at the end requires $O(n^2)$ work and $O(\lg n)$ span, because the total length of all subsequences is $\frac{n(n+1)}{2} = O(n^2)$, and $|a| = n$. The total work and span are therefore

$$W(e) = O(|a|^2), \text{ and}$$

$$S(e) = O(\lg |a|).$$

Exercise 19.7 (Comprehension with Conditionals). Given sequences a of natural numbers and b of letters of the alphabet, we wish to compute the sequence that pairs each even element of a with all elements of b that are vowels.

Solution. We can write this simply by adding the filtering predicates *isEven*, which holds for even numbers, and *isVowel*, which holds for vowels.

$$\langle (x, y) : x \in a, y \in b \mid \text{isEven } x, \text{isVowel } y \rangle$$

In our more basic notation, this translates to

$$\text{flatten } \langle \langle (x, y) : y \in b \mid \text{isVowel } y \rangle : x \in a \mid \text{isEven } x \rangle.$$

Remark (Eliminating *flatten*). In this section, we considered 2D and 3D geometric examples over the space of points whose coordinates are natural numbers. For these specific examples, we can eliminate the need for *flatten* by projecting the multidimensional space into a single dimensional space and using arithmetic over natural numbers to recover the coordinates of the points. This technique, however, does not generalize. For example, if we map over two strings (sequences of characters), then there is no analogous way to project efficiently the two strings into a single string that we can map over.

Remark (Comprehensions). Many programming languages support syntax based on set-comprehensions, sometimes directly for sets (e.g., SETL), or for other collections of values such as lists, sequences, or mappings (e.g. Python, Haskell and Javascript). The comprehensions syntax, however, is not uniform among the languages. Indeed even among mathematical texts on set theory, the syntax for set comprehensions varies. In our usage, we try to be self consistent, but we are not always consistent with usage found elsewhere.

2 Computing Primes

Problem 19.8 (Primes). The *primes* problem requires finding all prime numbers less than a given natural number n .

Recall that a natural number n is a prime if it has exactly two distinct divisors 1 and itself. For example, the number 1 is not prime, but 2, 3, 7, and 9967 are. For simplicity we assume in the rest of this section, that $n \geq 2$.

Observation. If a natural number n is not prime, then it has a divisor that is at most \sqrt{n} .

The observation holds, because for any $i \times j = n$, either i or j is less than or equal to \sqrt{n} .

Algorithm 19.4 (Brute Force Primality Test).

```
isPrime n =
  let
    all = ⟨ n mod i : 1 ≤ i ≤ ⌊√n⌋ ⟩
    divisors = ⟨ x : x ∈ all | x = 0 ⟩
  in
    |divisors| = 1
  end
```

Cost of Brute Force Primality Test. Let's calculate the work and span of this algorithm based on the array sequence cost specification. The algorithm constructs a sequence of length $\lfloor\sqrt{n}\rfloor$ and then filters it. Since the work for computing $i \bmod n$ and checking that a value is zero $x = 0$ is constant, based on the array-sequence costs, we can write work as

$$W_{isPrime}(n) = O \left(1 + \sum_{i=1}^{\lfloor\sqrt{n}\rfloor} O(1) \right) = O(\sqrt{n}).$$

Similarly we can write span as:

$$S_{isPrime}(n) = O \left(\lg \sqrt{n} + \max_{i=1}^{\lfloor\sqrt{n}\rfloor} O(1) \right) = O(\lg n).$$

The $\lg \sqrt{n}$ additive terms come from the cost specification for *filter*.

Since parallelism is the ratio of work to span, it is

$$O \left(\frac{\sqrt{n}}{\lg \sqrt{n}} \right).$$

This is not an abundant amount of parallelism but adequate especially, because work is small.

Algorithm 19.5 (Brute Force Solution to the Primes Problem). Now that we can test for primality of a number, we can solve the primes problem by testing the numbers up to n . We can write the code for such a brute-force algorithm as follows.

```
primesBF n =
  let
    all = ⟨ i : 1 < i < n ⟩
    primes = ⟨ x : x ∈ all | isPrime(x) ⟩
  in
    primes
  end
```

Let's analyze work and span, again using array sequences. Constructing the sequence *all* using *tabulate* requires linear work. Filtering through *all* requires work that is the sum of the work of the calls to *isPrime*; thus we have

$$\begin{aligned} W_{\text{primesBF}}(n) &= O\left(\sum_{i=2}^{n-1} 1 + W_{\text{isPrime}}(i)\right) \\ &= O\left(\sum_{i=2}^{n-1} 1 + \sqrt{i}\right) \\ &= O\left(n^{3/2}\right). \end{aligned}$$

Similarly, the span is dominated by the maximum of the span of calls to *isPrime* and a logarithmic additive term.

$$\begin{aligned} S_{\text{primesBF}}(n) &= O\left(\lg n + \max_{i=2}^n S_{\text{isPrime}}(i)\right) \\ &= O\left(\lg n + \max_{i=2}^n \lg i\right) \\ &= O(\lg n) \end{aligned}$$

The parallelism is hence

$$\frac{W_{\text{primesBF}}(n)}{S_{\text{primesBF}}(n)} = \frac{n^{3/2}}{\lg n}.$$

This is plenty of parallelism but comes at the expense of a large amount of work.

We can improve the work for the algorithm, because the algorithm does a lot of redundant work by repeatedly checking with the divisors. To test whether a number m is prime, the algorithm checks its divisors, it then checks essentially the same divisors for multiples of m , such as $2m, 3m, \dots$, which largely overlap, because if a number divides m , it also divides its multiples.

We eliminate this redundancy by more actively eliminating numbers that are *composites*, i.e., not primes. The basic idea is to create a collection of composite numbers up to n and

use this as a *sieve*. Generating such a sieve is easy: we just have to include for any number $i \leq \sqrt{n}$, its multiples of up to $\frac{n}{i}$. Having generated the sieve, what remains is to run the numbers up to n through the sieve. To do this in parallel, we can use `use inject` or in fact (non-deterministic) `ninject`, because all updates into the same position injects the same value.

Exercise 19.9. To generate all composite numbers between 2 and n , prove that it suffices to consider all $i \in \mathbb{N}$ $1 \leq i \leq \sqrt{n}$ and all of i 's multiples up to $\frac{n}{i}$.

Solution. By the [observation above](#), any composite number n has a divisor that is at most \sqrt{n} . It therefore suffices to consider multiples of numbers less or equal to \sqrt{n} .

Algorithm 19.6 (Prime Sieve). The pseudo-code below presents the prime-sieve algorithm. The idea is to do construct the sieve as a length- n sequence of the Boolean value `true`, and then update the sequence by writing `false` into all positions that correspond to composite numbers. The remaining `true` values indicate the prime numbers.

```
primeSieve n =
  let
    (* Composite numbers. *)
    cs = ⟨i * j : 2 ≤ i ≤ ⌊√n⌋, 2 ≤ j ≤ n/i⟩
    sieve = ⟨(x, false) : x ∈ cs⟩
    all = ⟨true : 0 ≤ i < n⟩
    isPrime = ninject all sieve
    primes = ⟨i : 2 ≤ i < n | isPrime[i] = true⟩
  in
    primes
  end
```

Cost of the Sieve Algorithm. For the analysis, we shall consider the phases of the algorithm and show that the work and span are functions of the total number of composites which we denote by m .

- Generating each composite takes constant work and because it is just a multiplication. The work for generating the sequence of composites is linear in the total number of composites, m . The span is $O(\lg n)$ because of the `flatten`. Constructing the sieve requires linear work in its length, which is m , and constant span.
- The work of `ninject` is also proportional to the length of `sieve`, m , and its span is constant.
- The work for computing `primes`, using `tabulate` and `filter` is proportional to n , and the span is $O(\lg n)$.

Therefore the total work is proportional to the number composites m , which is larger than n , and the total span is $O(\lg(n + m))$. To calculate m , we can add up the number of multiples each i from 2 to $\lfloor\sqrt{n}\rfloor$ have, i.e.,

$$\begin{aligned}
m &= \sum_{i=2}^{\lfloor \sqrt{n} \rfloor} \left\lceil \frac{n}{i} \right\rceil \\
&\leq (n+1) \sum_{i=2}^{\lfloor \sqrt{n} \rfloor} \frac{1}{i} \\
&\leq (n+1)H(\lfloor \sqrt{n} \rfloor) \\
&\leq (n+2)\ln n^{1/2} \\
&= \frac{n+2}{2}\ln n.
\end{aligned}$$

Here $H(n)$ is the n^{th} harmonic number, which is known to be bounded below by $\ln n$ and above by $\ln n + 1$. We therefore have

$$\begin{aligned}
W_{\text{primeSieve}}(n) &= O(n \lg n), \text{ and} \\
S_{\text{primeSieve}}(n) &= O(\lg n).
\end{aligned}$$

We have thus reduced the work from $O(n^{3/2})$ to something much more reasonable: $O(n \lg n)$.

Remark. The algorithm for computing primes described here dates back to antiquity and attributed to Eratosthenes of Cyrene, a Greek mathematician.

Chapter 20

Ephemeral and Single-Threaded Sequences

This chapter covers implementations of [sequences](#) that support constant work updates.

1 Persistent and Ephemeral Implementations

Persistent Data Structures. The implementations and the cost models that we have discussed so far are “non-destructive” in the sense that if we use a sequence, by for example, passing the sequence to an operation such as *map*, *update*, or *inject* the sequence remains the same after the operation completes. Such implementations are sometimes called *pure* or *persistent*.

Persistence is generally a desirable property. Some algorithms benefit from persistence and it is safe for parallelism. But persistence does come with a cost, because we are not allowed to update data destructively in place. For example, in [array sequences](#), the *update a* (i, v) and *inject a b* operations require $\Omega(|a|)$ work because they have to copy the sequence a . In [tree sequences](#), *update a b* and *inject a b* require $\Theta(\lg |a|)$ and $\Theta(\lg |a| + \lg |b|)$ work, but in some algorithms this is still high.

Ephemeral Data Structures. Persistence is not always necessary. For example, an algorithm may use a data structure in a “linear” fashion, where it uses or more precisely “consumes” an instance of the data structure no more than once. Linearity is relatively common, especially in sequential algorithms. For example, an algorithm may, at each step, consume one instance of a data structure and create a new instance, which is then consumed in another step. In such use cases, we may employ destructive updates in the implementation

of the data structure: because an instance is never consumed more than once, it is safe to destruct or reuse it when making a new instance. We refer to an implementation of a data structure that destroys existing instances as *ephemeral*.

Disadvantages of Ephemeral Implementations. Even though they can be efficient, ephemeral implementations have one important disadvantage: they are generally not safe for parallelism. As an example, consider the following three sequences:

$$\begin{aligned} a &= \langle 0, 1, \dots, n-1 \rangle \\ b &= \langle (0, 0), (1, 2), (2, 4), \dots, (n-1, 2n-2) \rangle \\ c &= \langle (0, 1), (1, 3), (2, 5), \dots, (n-1, 2n-1) \rangle \end{aligned}$$

Using ephemeral sequences, the result of the following piece of code, which injects the sequence b and c into another sequence a is non-deterministic:

inject a b || inject a c.

This piece of code has as many as 2^n distinct outcomes: the element at position i is either the i^{th} even number or i^{th} odd number.

Ephemeral implementations can therefore make reasoning about the correctness parallel algorithms challenging, because we have to consider an exponential number of possibilities. [An earlier chapter](#) covers this topic in more detail. This does not mean, however, that ephemeral data structures should be avoided at all cost. They are usually acceptable in sequential algorithms. Even in parallel algorithms, it is sometimes possible to use them in a structured fashion and establish that they don't harm correctness.

2 Ephemeral Sequences

Constant Work Updates. We can create an ephemeral version of [array sequences](#) by changing the *update*, *inject*, and *ninject* primitives to update the input array destructively. For an update sequence of length m , the resulting implementation has the following improved bounds:

- $O(1)$ work and span for *update*,
- $O(m)$ work and $O(\lg d)$ span for *inject*, where d is the degree of the update sequence,
- $O(m)$ work and $O(1)$ span for *ninject*.

Note that this implementation is significantly more work efficient than the persistent one, and thus can make a real difference in complexity if the algorithm performs many updates.

3 Single-Threaded Sequences

Single-threaded sequence data structure offers a specific interface that can in some cases, combine the best of ephemeral and persistence sequences. The data structure is persistent—its functions have no externally visible effects—but its implementation internally uses benign effects. These benign effects make the cost specification more subtle.

Example 20.1. Recall that the function *update a (i, v)* updates sequence *a* at location *i* with value *v* returning the new sequence, and that *inject a b* updates sequence *a* using a sequence *b* of index-value pairs (each value is written to the corresponding index. Using arrays costs, *update* requires $\Theta(|a|)$ work, and *inject* requires $\Theta(|a| + |b|)$ work.

We can implement *inject* using *update* as follows.

$$\text{inject } a \ b = \text{ iterate update } a \ (\text{reverse } b)$$

This code iterates over *a* making each of the updates specified in *b*. The problem, beyond being completely sequential, is that each update does $\Theta(|a|)$ work so the total work is $O(|a| \cdot |b|)$ instead of $O(|a| + |b|)$. The problem is that it is a waste to copy the sequence for every update.

Data Type 20.1 (Single Threaded Sequences). For any element type α , the α -*single threaded sequence* (*stseq*) data type is the type \mathbb{T}_α consisting of the set of all α *stseq*'s, and the following functions.

$$\begin{array}{ll} \text{fromSeq} & : \mathbb{S}_\alpha \rightarrow \mathbb{T}_\alpha \\ \text{toSeq} & : \mathbb{T}_\alpha \rightarrow \mathbb{S}_\alpha \\ \text{nth} & : \mathbb{T}_\alpha \rightarrow \mathbb{N} \rightarrow \alpha \\ \text{update} & : \mathbb{T}_\alpha \rightarrow (\mathbb{N} \times \alpha) \rightarrow \mathbb{T}_\alpha \\ \text{inject} & : \mathbb{T}_\alpha \rightarrow \mathbb{S}_{\mathbb{N} \times \alpha} \rightarrow \mathbb{T}_\alpha \end{array}$$

where \mathbb{S}_α are standard sequences, and *nth*, *update*, and *inject* behave as they do for standard sequences.

An *stseq* is basically a sequence but with very little functionality. Other than converting to and from sequences, the only functions are to read from a position of the sequence (*nth*), update a position of the sequence (*update*), or update multiple positions in the sequence (*inject*). To use other functions from the sequence library, one needs to convert an *stseq* back to a sequence (using *toSeq*).

To define the cost specification we need to distinguish between the latest version of an *stseq*, and earlier versions. Whenever we update a sequence, we create a new version, and the old version is still around due to the persistence. The cost specification then gives different costs for updating the latest version and old versions. Here we only define the cost for updating and accessing the latest version, because this is the only way we will be using an *stseq*.

Cost Specification 20.2 (Single Threaded Array Sequence).

	Work	Span
<i>fromSeq a</i>	$O(a)$	$O(1)$
<i>toSeq a</i>	$O(a)$	$O(1)$
<i>nth a i</i>	$O(1)$	$O(1)$
<i>update a (i, v)</i>	$O(1)$	$O(1)$
<i>inject a b</i>	$O(b)$	$O(\lg(\text{degree}(d)))$

In the cost specification the work for both *nth* and *update* is $O(1)$, which is asymptotically as good as we can get. Again, however, this is only when *a* is the latest version of a sequence (i.e. no one else has updated it). The work for *inject* is proportional to the number of updates. It can be viewed as a parallel version of *update*.

Example 20.2 (Inject with Update Revisited). If we return to our previous example:

$$\text{inject } a b = \text{ iterate update } a (\text{reverse } b)$$

Using single threaded array sequences, the work is now just $\Theta(|b|)$ since each of the $|b|$ updates take constant work, and every update is on the last version. The span, however, is also $\Theta(|b|)$ since *iterate* is fully sequential. Therefore the built in *inject* is significantly better for parallelism, but they both take the same amount of work.

Applications in this Book. In this book we can use *stseq*'s for some graph algorithms, including breadth-first search (BFS) and depth-first search (DFS), and for hash tables.

3.1 Implementation

You might be curious about how single threaded sequences can be implemented so that they act purely functional but match the given cost specification. Here we will just briefly outline one approach.

The idea is to keep with each sequence a version number, and for each position in the sequence a version list. The version number is incremented each time the sequence is updated either with *update* or *inject*. A version list is a linked list of all the updates to the corresponding position, each with the version number of when the update was made. The most recent version is kept at the head of the list, and the rest are kept in decreasing order. A mutable (impure) array is used to keep a pointer to the head of each list, and the version number is also mutable.

We now consider how to do lookups and updates on the most recent version. For a lookup we can just look into the given location and take the first value from the head of the linked list. This takes constant work. To do an update requires updating the version number, grabbing the appropriate list, adding the value and version number to the front of the list, and then writing back the new head of the list using mutation (if done right, this is

a benign effect). The update also takes constant work. Looking up or updating an old version is more expensive. A lookup requires grabbing the list from the given position, and then looking through the list for the correct version. An update requires copying the whole array.

There are two tricky aspects. The first is ensuring that the lists do not get too large. This can be implemented by copying the whole array once the number of updates equals the length of the sequence. The second is to ensure safety in a parallel setting, which can be achieved by using atomic read-modify-write operations. We will briefly cover such operations in the final part of this course.

Chapter 21

Tree Sequences

In this chapter, we present an overview of an implementation of sequences based on balanced trees. The implementation is parameterized based on a minimal implementation of trees, which basically just allows cutting a tree in two (exposing), and appending two trees (joining).

Important. This is a freshly drafted chapter. It is incomplete and likely has bugs (don't hesitate to point them out). It is currently written at a high level, without going into the details. We don't use tree sequences much in this book, so the reader can safely skip this material.

1 Primitive Tree Sequences

As with the case of arrays, we build a full tree implementation on top of a minimal interface. In later chapters we show how to implement this interface efficiently.

Data Type 21.1 (Primitive Tree Sequences). For any element type α , the α *tree sequence data type* \mathbb{S}_α is defined as:

```
type T $\alpha$  = Zero | One of  $\alpha$  | Two of  $\mathbb{S}_\alpha \times \mathbb{S}_\alpha$ 
length :  $\mathbb{S}_\alpha \rightarrow \mathbb{N}$ 
expose :  $\mathbb{S}_\alpha \rightarrow T_\alpha$ 
join :  $T_\alpha \rightarrow \mathbb{S}_\alpha$ 
```

The *expose* function takes a tree sequence and returns *Zero* if it is empty *One*(x) if it contains a single element x , and *Two*(L, R) otherwise. In the third case the results L and R are the result of some cut of the sequence into two pieces, consisting of the left side L and the right side R , respectively.

From the point of view of a tree, the *expose* function is telling us whether the tree is empty, has a single value, or is a node with two children, L and R .

The *join* function can be viewed as the inverse. If given a *Zero* it will create an empty tree sequence. If given *One*(v) it will create a tree sequence with a single element v . If given *Two*(L, R) it appends the two tree sequences L and R together.

From the point of view of a tree, the *join* function is building an empty tree, a single valued tree, or it is joining two trees to form a new tree. The third case can be thought of as creating a new node with L and R as children. However, the implementation has to be smarter than that. In particular the two trees L and R might have very different sizes and heights. In this case the *join* might need to rebalance the tree to keep the result almost balanced.

Using the primitive tree sequence datatype we can implement all the functionality of the sequence datatype. Furthermore, assuming certain cost bounds on the functionality of the primitive sequence interface, all the costs will satisfy the cost bounds discussed in Section 3.

To define costs for our primitive tree sequences we assume that every tree sequence T has a integer *rank* associated with it, which will be denoted as $r(T)$, and that the ranks satisfy the following conditions:

1. $r(T) \in O(\log |T|)$,
2. for $\text{expose}(T) = \text{Two}(L, R)$:
 $r(L) + 1 \leq r(T) \leq r(L) + 2$ and
 $r(R) + 1 \leq r(T) \leq r(R) + 2$
3. $r(\text{join}(\text{Two}(L, R))) \leq \max(r(L), r(R)) + 1$

The rank can intuitively be thought as the height of the tree. The first condition says that the rank (“height”) is logarithmic in the size. This indicates that the tree is reasonably balanced (all leaves are within order log of the size). The second condition says that the parent must have a larger rank than both its children, but not more than two greater than either child. This means that the rank must be an upper bound on the height since the rank decreases by at least one when going to each child. The third condition says that if you join two trees, the resulting tree will have a rank that is at most one more than the larger rank of the two. Note that the resulting tree might have a rank that is equal to the larger rank. This does not break the second rule since the join might involve rotations that rearrange the tree to keep its rank low.

It turns out that these conditions are true for a variety of balancing schemes including red-black trees (rank is height), AVL trees (rank is based on black height), and weight-balanced trees (rank is the log of the size using an appropriate base).

Cost Specification 21.2 (Primitive Tree Sequences). We specify the *primitive tree sequence*

costs as follows.

Operation	Work and Span
<i>length a</i>	1
<i>expose a</i>	1
<i>join(Zero) or join(One(x))</i>	1
<i>join(Two(L, R))</i>	$1 + r(L) - r(R) $

Hence the cost of joining two trees is proportional the absolute value of the difference of their ranks.

Note that there is no parallelism in these operations. Parallelism will be used in building the sequence functions from them.

In this chapter we will not describe how to satisfy these bounds, but they can be satisfied with a variety of trees including AVL trees, Red-Black trees and Weight Balanced trees. Intuitively the bounds should make some sense. Exposing just requires looking at the root of the tree and returning the two children if not empty or a singleton. Keeping track of the length just requires storing the size of the subtree in each node of the tree and looking it up.

The $\text{join}(\text{Two}(L, R))$ is the only particularly interesting case. Intuitively if input trees L and R are the same rank, then it should be cheap since we can just create a new node as their parent and not require any balancing. The amount of rebalancing required is proportional to the difference in rank, and hence the specified cost.

2 Parametric Implementation of Tree Sequences

We now describe how to implement most of the sequence ADT interface using primitive tree sequences. The implementation is given by the following algorithms.

Algorithm 21.3 (Tree Sequence Functions).

```

empty = join(Zero)

singleton x = join(One(x))

append(a, b) = join(Two(a, b))

nth S i =
  case (expose S) of
    Zero ⇒ error
    | One(x) ⇒ if (i = 0) then x else error
    | Two(L, R) ⇒
      if (i > |L|) then nth R (i - |L|)
      else nth L i

map f S =
  case (expose S) of
    Zero ⇒ empty
    | One(v) ⇒ singleton(f(v))
    | Two(L, R) ⇒ append(map f L || map f R)

tabulate f n =
  let tab (s, e) =
    if (e ≤ s) then empty
    else if (e = s + 1) then singleton(f(s))
    else append(tab(s, (s + e)/2) || tab((s + e)/2, e))
  in tab(0, n) end

filter f S =
  case (expose S) of
    Zero ⇒ empty
    | One(x) ⇒ if f(x) then S else empty
    | Two(L, R) ⇒ append(filter f L || filter f R)

drop S n =
  case (expose S) of
    Zero ⇒ empty
    | One(x) ⇒ if (n = 0) then singleton(x)
                else empty
    | Two(L, R) ⇒
      if (n > |L|) then drop(R, n - |L|)
      else append(drop(L, n), R)

update S (i, v) =
  case (expose S) of
    Zero ⇒ empty
    | One(x) ⇒ if (i = 0) then singleton(v) else S
    | Two(L, R) ⇒
      if (i > |L|) then append(L, update R (i - |L|, v))
      else append(update L (i, v), R)

subseq S (a, n) = take(drop(S, a), n)

flatten S = reduce append empty S

```

For now we leave it as an exercise to show that these match the tree-sequence cost bounds. Some of them such as *nth*, *map* and *tabulate* are relatively straightforward. The functions *nth*, *filter* and *drop* are trickier since they involve joins on two trees that are not necessarily almost the same size.

Part V

Algorithm Design And Analysis

Chapter 22

Introduction

Designing algorithms requires a toolbox of design techniques. In this chapter we will describe some of these techniques. Chapter 23 covers the two most basic techniques, which are algorithmic reductions and brute force. In the first we simply convert one problem to another for which we already have a solution. For the second, we find a solution, or the best solution by trying “all possibilities”. Chapter 24 covers Divide-and-Conquer. This is an approach you have most likely seen before, but we go into some more depth. Chapter 25 covers a technique called “contraction”. The basic idea is to contract a problem into a single smaller instance of itself, recurse on the smaller instance, and then use it to help solve the larger instance. Chapter 26 covers an example problem, the Maximum Contiguous Subsequence Sum (MCSS) problem, and shows how it can be solved by using many different techniques.

In later parts we will cover other techniques such as randomization, and dynamic programming.

Chapter 23

Basic Techniques

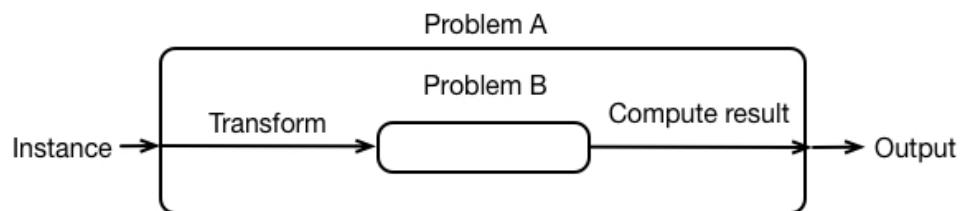
This chapter covers two basic algorithm-design techniques: reduction and brute force.

1 Algorithmic Reduction

Definition 23.1 (Algorithmic Reduction). We say that a problem A is *reducible* to another problem B , if any instance of A can be reduced to one or more instances of B . This means that it is possible to solve any instance of A by following the a three step process (also illustrated below).

- Transform the instance of problem A to one or many instances of problem B .
- Solve all instances of problem B .
- Use results to B instances to compute result to the A instance.

We sometimes refer to problem B as a *subproblem* of A , especially if the reduction involves producing multiple instances of B .



Efficiency of Reduction. The efficiency of an algorithm obtained via reduction depends on the complexity of the problem being reduced to, and the cost of transforming the instances and putting together the results to compute final result. It is therefore important to ensure that the costs of the instance transformation and the cost of computing the final result remain small in comparison to the cost of the problem being reduced to.

We generally think of a reduction as being *efficient* if the total cost of the input and output transformations are asymptotically the same as that of the problem being reduced to.

Example 23.1 (Reduction from Maximal to Sorting). We can reduce the problem of finding the maximal element in a sequence to the problem of (comparison) sorting. To this end, we sort the sequence in ascending order and then select the last element, which is one the largest elements in the input sequence.

In this reduction, we don't need to transform the input. To compute the final result, we retrieve the last element in the sequence, which usually requires logarithmic work or less. Because comparison sorting itself requires $\Theta(n \lg n)$ work for a sequence of n elements, the reduction is efficient. But, the resulting algorithm is not a work-efficient: it requires $\Theta(n \lg n)$ work, whereas we can find the maximum element of a sequence in $\Theta(n)$ work.

Example 23.2 (Reduction from Minimal to Maximal). Suppose that we wish to find the minimal element in a sequence but only know how to find the maximal element efficiently in linear work. We can reduce the minimality problem to the maximality problem by inverting the sign of all the elements in the input, finding the maximal value of the transformed input, and then inverting the sign of the result to obtain the minimum. Because these computations require linear work in the size of the input sequence, and because finding the maximal element requires linear work, this reduction is quite efficient.

The resulting algorithm, which requires linear work in total, is also asymptotically work efficient.

Remark. Reduction is a powerful technique and can allow us to solve a problem by reducing it to another problem, which might appear very different or even unrelated. For example, we might be able to reduce a problem on strings such as that of finding the shortest superstring of a set of strings to a graph problem. We have seen [an example of this earlier in genome sequencing](#).

Remark (Proving Hardness by Reduction). Reduction technique can also be used “in the other direction” to show that some problem is at least as hard as another or to establish a lower bound. In particular, if we know (or conjecture) that problem A is hard (e.g., requires exponential work), and we can reduce it to problem B (e.g., using polynomial work), then we know that B must also be hard. Such reductions are central to the study of complexity theory.

2 Brute Force

The brute-force technique involves enumerating all possible solutions, a.k.a., *candidate solutions* to a problem, and checking whether each candidate solution is valid. Depend-

ing on the problem, a solution may be returned as soon as it is found, or the search may continue until a better solution is found.

Example 23.3 (Brute-Force Sorting). We are asked to sort a set of keys a . As a first algorithm, we can apply the brute force technique by considering the candidate-solution space (sequence of keys), enumerating this space by considering each candidate, and checking that it is sorted.

More precisely speaking, this amounts to trying all permutations of a and testing that each permutation is sorted. We return as soon as we find a sorted permutation. Because there are $n!$ permutations and $n!$ is very large even for small n , this algorithm is inefficient. For example, using Sterling's approximation, we know that $100! \approx \frac{100^{100}}{e} \geq 10^{100}$. There are only about 10^{80} atoms in the universe so there is probably no feasible way we could apply the brute force method directly. We conclude that this approach to sorting is not *tractable*, because it would require a lot of energy and time for all but tiny inputs.

Remark. The total number of atoms in the (known) universe, which is less than 10^{100} is a good number to remember. If your solution requires this much work or time, it is probably intractable.

Important. Brute-force algorithms are usually naturally parallel. Enumerating all candidate solutions (e.g., all permutations, all elements in a sequence) is typically easy to do in parallel and so is testing that a candidate solution is indeed a valid solution. But brute-force algorithms are usually not work efficient and therefore not desirable—in algorithm design, our first priority is to minimize the total work and only then the span.

Example 23.4 (Maximal Element). Suppose that we are given a sequences of natural numbers and we wish to find the maximal number in the sequence. To apply the brute force technique, we first identify the solution space, which consists of the elements of the input sequence. The brute-force technique advises uses to try all candidate solutions. We thus pick each element and test that it is greater or equal to all the other elements in the sequence. When we find one such element, we know that we have found the maximal element.

This algorithm requires $\Theta(n^2)$ comparisons and can be effective for very small inputs, but it is far from an optimal algorithm, which would perform $\Theta(n)$ comparisons. .

Example 23.5 (Brute-Force Overlaps). Given two strings a and b , let's define the (maximum) *overlap* between a and b as the largest suffix of a that is also a prefix of b . For example, the overlap between "15210" and "2105" is "210", and the overlap between "15210" and "1021" is "10".

We can find the overlap between a and b by using the brute force technique. We first note that the solution space consists of the suffixes of a that are also prefixes of b . To apply brute force, we consider each possible suffix of a and check if it is a prefix of b . We then select the longest such suffix of a that is also a prefix of b .

The work of this algorithm is $|a| \cdot |b|$, i.e., the product of the lengths of the strings. Because we can try all positions in parallel, the span is determined by the span of checking that a particular prefix is also a suffix, which is $O(\lg |b|)$. Selecting the maximum requires $O(\lg |a|)$ span. Thus the total span is $O(\lg (|a| + |b|))$.

Exercise 23.1. The analysis given in the [example above](#) is not “tight” in the sense that there is a bound that is asymptotically dominated by $O(|a| \cdot |b|)$. Can you improve on the bound?

Example 23.6 (Brute-Force Shortest Paths). Consider the following problem: we are given a graph where each edge is assigned distance (e.g., a nonnegative real number), and asked to find the shortest path between two given vertices in the graph.

Using brute-force, we can solve this problem by enumerating the solution space, which is the set of all paths between the given two vertices, and selecting the path with the smallest total distance. To compute the total distance over the path, we sum up the distances of all edges on the path.

Example 23.7 (Brute-Force Shortest Distances). Consider the following variant of the shortest path problem: we are given a graph where each edge is assigned a distance (e.g., a nonnegative real number), and asked to find the shortest distance between two given vertices in the graph. This problem differs from the one above, because it asks for the shortest distance rather than the path.

Using brute-force, we might want to solve this problem by enumerating the solution set, which is the real numbers starting from zero, and selecting the smallest one. But, this is impossible, because real numbers are not countable, and cannot be enumerated.

One solution is to use the [reduction technique](#) and reduce the problem to the shortest-path version of the problem [described above](#) and then compute the distance of the returned path.

As an additional refinement, we can reformulate [the shortest-path problem](#) to return the distance of the shortest path in addition to the path itself. With this refinement, computing the final result requires no additional computation beyond returning the distance.

Important (Strengthening). We used an important technique in [the shortest-paths example](#) where we refined the problem that we are reducing to so that it returns us more information than strictly necessary. This technique is called [strengthening](#) and is commonly employed to improve efficiency by reducing redundancy.

Example 23.8 (Brute-Force Shortest Paths (Decision Version)). Consider the “decision problem” variant of the shortest path problem: we are given a graph where each edge is assigned a distance (e.g., a nonnegative real number), and we are given a [budget](#), which is a distance value. We are asked to check whether there is a path between two given vertices in the graph that is shorter than the given budget. This problem differs from the shortest path problem, because it asks us to return a “Yes” or “No” answer. Such problems are called [decision problems](#).

Using brute force, we could enumerate all candidate solutions, which are either “Yes” or “No”, and test whether each one is indeed a valid solution. In this case, this does not help, because we are back to our original problem that we started with. Again, we can “refine” our brute-force approach a bit by reducing it to the original [shortest-path problem](#), and then checking whether the distance of the resulting path is larger than the budget.

Remark. Experienced algorithms designers perform reductions between different variants of the problem as shown in the last two examples “quietly” (Example 23.7 and Example 23.8). That is, they don’t explicitly state the reduction but simply apply the brute-force technique to a slightly different set of candidate solutions. For beginners and even for experienced designers, however, recognizing such reductions can be instructive.

Remark (Utility of Brute Force). Even though brute-force algorithms are usually inefficient, they can be useful.

- Brute-force algorithms are usually easy to design and are a good starting point toward a more efficient algorithm. They can help in understanding the structure of the problem, and lead to a more efficient algorithm.
- In practice, a brute-force algorithm can help in testing the correctness of a more efficient algorithm by offering a solution that is easy to implement correctly.

Chapter 24

Divide and Conquer

This chapter describes the divide-and-conquer technique, an important algorithm-design technique, and applies it to several problems. Divide and conquer is an effective technique for solving a variety of problems, and usually leads to efficient and parallel algorithms.

1 Divide and Conquer

A divide-and-conquer algorithm has a distinctive anatomy: it has a base case to handle small instances and an inductive step with three distinct phases: “divide”, “recur”, and “combine.” The *divide* phase divides the problem instance into smaller instances; the *recur* phase solves the smaller instances; and the *combine* phase combines the results for the smaller instance to construct the result to the larger instance.

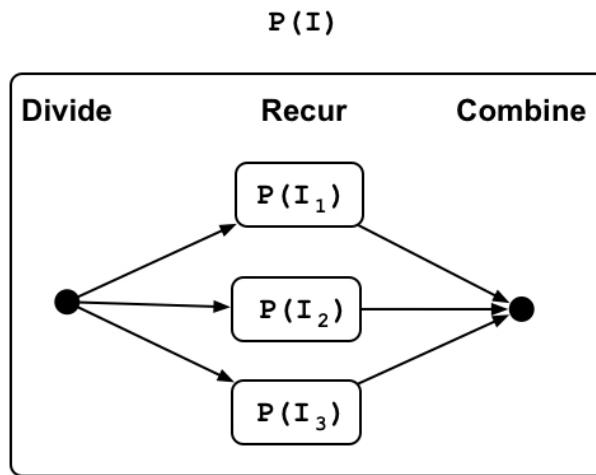
Definition 24.1 (Divide-And-Conquer Algorithm). A divide-and-conquer algorithm has the following structure.

Base Case: When the instance I of the problem P is sufficiently small, compute the solution $P(I)$ perhaps by using a different algorithm.

Inductive Step:

1. **Divide** instance I into some number of smaller instances of the same problem P .
2. **Recur** on each of the smaller instances and compute their solutions.
3. **Combine** the solutions to obtain the solution to the original instance I .

Example 24.1. The drawing below illustrates the structure of a divide-and-conquer algorithm that divides the problem instance into three independent subinstances.



Properties of Divide-and-Conquer Algorithms. Divide-and-Conquer has several important properties.

- It follows the structure of an inductive proof, and therefore usually leads to relatively simple proofs of correctness. To prove a divide-and-conquer algorithm correct, we first prove that the base case is correct. Then, we assume by strong (or structural) induction that the recursive solutions are correct, and show that, given correct solutions to smaller instances, the combined solution is correct.
- Divide-and-conquer algorithms can be work efficient. To ensure efficiency, we need to make sure that the divide and combine steps are efficient, and that they do not create too many sub-instances.
- The work and span for a divide-and-conquer algorithm can be expressed as a mathematical equation called [recurrence](#), which can be usually be solved without too much difficulty.
- Divide-and-conquer algorithms are naturally parallel, because the sub-instances can be solved in parallel. This can lead to significant amount of parallelism, because each inductive step can create more independent instances. For example, even if the algorithm divides the problem instance into two subinstances, each of those subinstances could themselves generate two more subinstances, leading to a geometric progression, which can quickly produce abundant parallelism.

Analysis of Divide-and-Conquer Algorithms. Consider an algorithm that divides a problem instance of size n into $k > 1$ independent subinstances of sizes n_1, n_2, \dots, n_k , recursively solves the instances, and combine the solutions to construct the solution to the original instance.

We can write the work of such an algorithm using the recurrence then

$$W(n) = W_{\text{divide}}(n) + \sum_{i=1}^k W(n_i) + W_{\text{combine}}(n) + 1.$$

The work recurrence simply adds up the work across all phases of the algorithm (divide, recur, and combine).

To analyze the span, note that after the instance is divided into subinstance, the subinstances can be solved in parallel (because they are independent), and the results can be combined. The span can thus be written as the recurrence:

$$S(n) = S_{\text{divide}}(n) + \max_{i=1}^k S(n_i) + S_{\text{combine}}(n) + 1.$$

Note. The work and span recurrences for a divide-and-conquer algorithm usually follow the recursive structure of the algorithm, but is a function of size of the arguments instead of the actual values.

Example 24.2 (Maximal Element). We can find the maximal element in a sequence using divide and conquer as follows. If the sequence has only one element, we return that element, otherwise, we divide the sequence into two equal halves and recursively and in parallel compute the maximal element in each half. We then return the maximal of the results from the two recursive calls. For a sequence of length n , we can write the work and span for this algorithm as recurrences as follows:

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

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

This recurrences yield

$$W(n) = \Theta(n) \text{ and}$$

$$S(n) = \Theta(\lg n).$$

Algorithm 24.2 (Reduce with Divide and Conquer). The *reduce* primitive performs a computation that involves applying an associative binary operation op to the elements of a sequence to obtain (reduce the sequence to) a final value. For example, reducing the sequence $\langle 0, 1, 2, 3, 4 \rangle$ with the $+$ operation gives us $0 + 1 + 2 + 3 + 4 = 10$. If the operation requires constant work (and thus span), then the work and span of a reduction is $\Theta(n)$ and $\Theta(\lg n)$ respectively.

We can write the code for the reduce primitive on sequences as follows.

```
reduceDC f id a =
  if isEmpty a then
    id
  else if isSingleton a then
    a[0]
  else
    let
      (l, r) = splitMid a
      (a, b) = (reduceDC f id l || reduceDC f id r)
    in
      f(a, b)
  end
```

2 Merge Sort

In this section, we consider the comparison sorting problem and the merge-sort algorithm, which offers a divide-and-conquer algorithm for it.

Definition 24.3 (The Comparison-Sorting Problem). Given a sequence a of elements from a universe U , with a total ordering given by $<$, return the same elements in a sequence r in sorted order, i.e. $r[i] \leq r[i + 1]$, $0 < i \leq |a| - 1$.

Algorithm 24.4 (Merge Sort). Given an input sequence, merge sort divides it into two sequences that are approximately half the length, sorts them recursively, and merges the sorted sequences. Mergesort can be written as follows.

```
mergeSort a =
  if |a| \leq 1 then
    a
  else
    let
      (l, r) = splitMid a
      (l', r') = (mergeSort l || mergeSort r)
    in
      merge(l', r')
    end
```

Note. In the merge sort algorithm given above the base case is when the sequence is empty or contains a single element. In practice, however, instead of using a single element or empty sequence as the base case, some implementations use a larger base case consisting of perhaps ten to twenty keys.

Correctness and Cost. To prove correctness we first note that the base case is correct. Then by induction we note that l' and r' are sorted versions of l and r . Because l and r together contain exactly the same elements as a , we conclude that $\text{merge}(l', r')$ returns a sorted version of a .

For the work and span analysis, we assume that merging can be done in $\Theta(n)$ work and $\Theta(\lg n)$ span, where n is the sum of the lengths of the two sequences. We can thus write the work and span for this sorting algorithm as

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

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

The recurrences solve to

$$W(n) = \Theta(n \lg n)$$

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

Remark (Quick Sort). Another divide-and-conquer algorithm for sorting is the quick-sort algorithm. Like merge sort, quick sort requires $\Theta(n \log n)$ work, which is optimal for the comparison sorting problem, but only “in expectation” over random decisions that it makes during its execution. While merge sort has a trivial divide step and interesting combine step, quick sort has an interesting divide step but trivial combine step. We will study quick sort in greater detail.

3 Sequence Scan

Intuition for Scan with Divide and Conquer. To develop some intuition on how to design a divide-and-conquer algorithm for the sequence scan problem, let’s start by dividing the sequence in two halves, solving each half, and then putting the results together.

For example, consider the sequence $\langle 2, 1, 3, 2, 2, 5, 4, 1 \rangle$. If we divide in the middle and scan over the two resulting sequences we obtain (b, b') and (c, c') , such that

$$(b, b') = (\langle 0, 2, 3, 6 \rangle, 8), \text{ and}$$

$$(c, c') = (\langle 0, 2, 7, 11 \rangle, 12).$$

Note now that b already gives us the first half of the solution. To compute the second half, observe that in calculating c in the second half, we started with the identity instead of the sum of the first half, b' . Therefore, if we add the sum of the first half, b' , to each element of c , we would obtain the desired result.

Algorithm 24.5 (Scan with Divide and Conquer). By refining the intuitive description above, we can obtain a divide-and-conquer algorithm for sequences scan, which is given below.

```

scanDC f id a =
  if |a| = 0 then
    (⟨ ⟩, id)
  else if |a| = 1 then
    (⟨ id ⟩, a[0])
  else
    let
      (b, c) = splitMid a
      ((l, b'), (r, c')) = (scanDC f id b || scanDC f id c)
      r' = ⟨ f(b', x) : x ∈ r ⟩
    in
      (append (l, r'), f(b', c'))
    end

```

Remark. Observe that this algorithm takes advantage of the fact that id is really the identity for f , i.e. $f(id, x) = x$.

Cost Analysis. We consider the work and span for the algorithm. Note that the combine step requires a map to add b' to each element of c , and then an append. Both these take $O(n)$ work and $O(1)$ span, where $n = |a|$. This leads to the following recurrences for the whole algorithm:

$$\begin{aligned} W(n) &= 2W(n/2) + O(n) & \in & O(n \log n) \\ S(n) &= S(n/2) + O(1) & \in & O(\log n). \end{aligned}$$

Although this is much better than $O(n^2)$ work, we can do better by using another design technique called contraction.

4 Euclidean Traveling Salesperson Problem

We consider a variant of the well-known Traveling Salesperson Problem (TSP) and design a divide-and-conquer heuristic for it. This variant, known as the Euclidean Traveling Salesperson Problem (eTSP), is **NP** hard. It requires solving the TSP problem in graphs where the vertices (e.g., cities) lie in a Euclidean space and the edge weights (e.g., distance measure between cities) is the Euclidean distance. More specifically, we're interested in the planar version of the eTSP problem, defined as follows:

Definition 24.6 (The Planar Euclidean Traveling Salesperson Problem). Given a set of points P in the 2-d plane, the *planar Euclidean traveling salesperson* (eTSP) problem is to find a tour of minimum total distance that visits all points in P exactly once, where the distance between points is the Euclidean (i.e. ℓ_2) distance.

Example 24.3. Assuming that we could go from one place to another using your personal airplane, this is the problem we would want to solve to find a minimum length route visiting your favorite places in Pittsburgh.

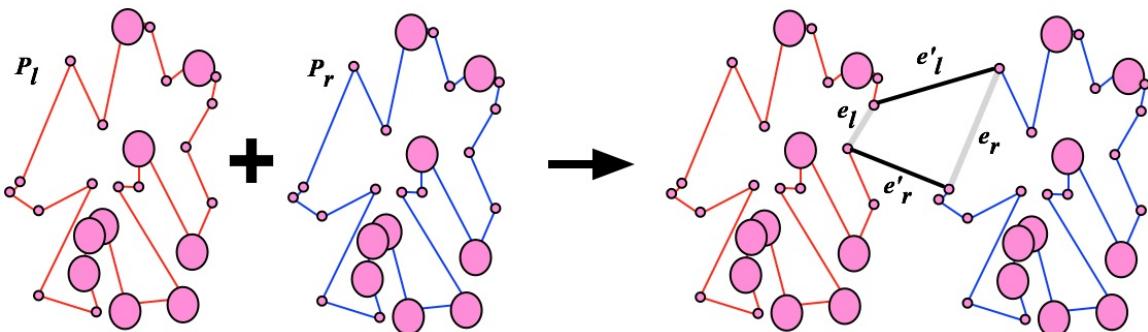
As with the TSP, eTSP is NP-hard, but it is easier to approximate. Unlike the TSP problem, which only has constant approximations, it is known how to approximate this problem to an arbitrary but fixed constant accuracy ε in polynomial time (the exponent of n has $1/\varepsilon$ dependency). That is, such an algorithm is capable of producing a solution that has length at most $(1 + \varepsilon)$ times the length of the best tour.

Note. In Section 2, we cover another approximation algorithm for a metric variant of TSP that is based on Minimum Spanning Trees (MST). That approximation algorithm gives a constant-approximation guarantee.

Intuition for a Divide and Conquer Algorithm for eTSP. We can solve an instance of the eTSP problem by splitting the points by a cut in the plane, solving the eTSP instances on the two parts, and then merging the solutions in some way to construct a solution for the original problem.

For the cut, we can pick a cut that is orthogonal to the coordinate lines. We could for example find the dimension along which the points have a larger spread, and then cut just below the median point along that dimension.

This division operation gives us two smaller instances of eTSP, which can then be solved independently in parallel, yielding two cycles. To construct the solution for the original problem, we can merge the solutions. To merge the solution in the best possible way, we can take an edge from each of the two smaller instances, remove them, and then bridge the end points across the cut with two new edges. For each such pair of edges, there are two possible ways that we can bridge them, because when we are on the one side, we can jump to any one of the endpoints of the two bridges. To construct the best solution, we can try out which one of these yields the best solution and take that one.



To choose which swap to make, we consider all pairs of edges of the recursive solutions consisting of one edge $e_l = (u_l, v_l)$ from the left and one edge $e_r = (u_r, v_r)$ from the right

and determine which pair minimizes the increase in the following cost:

$$\text{swapCost}((u_\ell, v_\ell), (u_r, v_r)) = \|u_\ell - v_r\| + \|u_r - v_\ell\| - \|u_\ell - v_\ell\| - \|u_r - v_r\|$$

where $\|u - v\|$ is the Euclidean distance between points u and v .

Algorithm 24.7 (Divide-and-Conquer eTSP). By refining the intuition described above, we arrive at a divide-and-conquer algorithm for solving eTSPs, whose pseudo-code is shown below.

```

eTSP (P) =
  if |P| < 2 then
    raise TooSmall
  else if |P| = 2 then
    ⟨(P[0], P[1]), (P[1], P[0])⟩
  else
    let
      ( $P_\ell, P_r$ ) = split P along the longest dimension
      ( $L, R$ ) = (eTSP  $P_\ell$ ) || (eTSP  $P_r$ )
      ( $c, (e, e')$ ) = minValfirst { (swapCost( $e, e'$ ), ( $e, e'$ )) :  $e \in L, e' \in R$  }
    in
      swapEdges (append ( $L, R$ ),  $e, e'$ )
    end
  
```

The function $\text{minVal}_{\text{first}}$ uses the first value of the pairs to find the minimum, and returns the (first) pair with that minimum. The function $\text{swapEdges}(E, e, e')$ finds the edges e and e' in E and swaps the endpoints. As there are two ways to swap, it picks the cheaper one.

Remark. This heuristic divide-and-conquer algorithm is known to work well in practice.

Cost Analysis. Let's analyze the cost of this algorithm in terms of work and span. We have

$$W(n) = 2W(n/2) + O(n^2)$$

$$S(n) = S(n/2) + O(\log n)$$

We have already seen the recurrence $S(n) = S(n/2) + O(\log n)$, which solves to $O(\log^2 n)$. Here we'll focus on solving the work recurrence.

To solve the recurrence, we apply a [theorem proven earlier](#), and obtain

$$W(n) = O(n^2).$$

Strengthening. In applications of divide-and-conquer technique that we have considered so far in this chapter, we divide a problem instance into instances of the same problem. For example, in sorting, we divide the original instance into smaller instances of the sorting problem. Sometimes, it is not possible to apply this approach to solve a given problem, because solving the same problem on smaller instances does not provide enough information to solve the original problem. Instead, we will need to gather more information when solving the smaller instances to solve the original problem. In this case, we can *strengthen* the original problem by requiring information in addition to the information required by the original problem. For example, we might strengthen the sorting problem to return to us not just the sorted sequence but also a histogram of all the elements that count the number of occurrences for each element.

5 Divide and Conquer with Reduce

Many divide-and-conquer algorithms have the following structure, where *emptyVal*, *base*, and *myCombine* span for algorithm specific values.

```
myDC a =
  if |a| = 0 then
    emptyVal
  else if |a| = 1 then
    base(a[0])
  else
    let (l, r) = splitMid a in
      (l', r') = (myDC l || myDC r)
    in
      myCombine (l', r')
    end
```

Algorithms that fit this pattern can be implemented in one line using the sequence *reduce* function. Turning a divide-and-conquer algorithm into a reduce-based solution is as simple as invoking *reduce* with the following parameters

reduce myCombine emptyVal (map base a).

Important. This pattern does not work in general for divide-and-conquer algorithms. In particular, it does not work for algorithms that do more than a simple split that partitions their input in two parts in the middle. For example, it cannot be used for implementing the quick-sort algorithm, because the divide step partitions the data with respect to a pivot. This step requires picking a pivot, and then filtering the data into elements less than, equal, and greater than the pivot. It also does not work for divide-and-conquer algorithms that split more than two ways, or make more than two recursive calls.

Chapter 25

Contraction

This chapter describes the contraction technique for algorithm design and applies it to several problems.

1 Contraction Technique

A contraction algorithm has a distinctive anatomy: it has a base case to handle small instances and an inductive step with three distinct phases: “contract,” “recur”, and “expand.” It involves solving recursively smaller instances of the same problem and then expanding the solution for the larger instance.

Definition 25.1 (Contraction). A contraction algorithm for problem P has the following structure.

Base Case: If the problem instance is sufficiently small, then compute and return the solution, possibly using another algorithm.

Inductive Step(s): If the problem instance is sufficiently large, then

- Apply the following two steps, as many times as needed.
 1. *Contract*: “contract”, i.e., map the instance of the problem P to a smaller instance of P .
 2. *Solve*: solve the smaller instance recursively.
- *Expand* the solutions to smaller instance to solve the original instance.

Remark. Contraction differs from divide and conquer in that it allows there to be only one *independent* smaller instance to be recursively solved. There could be multiple *dependent* smaller instances to be solved one after another (sequentially).

Properties of Contraction. Contraction algorithms have several important properties.

- Due to their inductive structure, we can establish the correctness of a contraction algorithm using principles of induction: we first prove correctness for the base case, and then prove the general (inductive) case by using strong induction, which allows us to assume that the recursive call is correct.
- The work and span of a contraction algorithm can be expressed as a mathematical recurrence that reflects the structure of the algorithm itself. Such recurrences can then usually be solved using well-understood techniques, and without significant difficulty.
- Contraction algorithms can be work efficient, if they can reduce the problem size geometrically (by a constant factor greater than 1) at each contraction step, and if the contraction and the expansion steps are efficient.
- Contraction algorithms can have a low span (high parallelism), if size of the problem instance decreases geometrically, and if contraction and expansion steps have low spans.

Example 25.1 (Maximal Element). We can find the maximal element in a sequence a using contraction as follows. If the sequence has only one element, we return that element, otherwise, we can map the sequence a into a sequence b which is half the length by comparing the elements of a at consecutive even-odd positions and writing the larger into b . We then find the largest in b and return this as the result.

For example, we map the sequence $\langle 1, 2, 4, 3, 6, 5 \rangle$ to $\langle 2, 4, 6 \rangle$. The largest element of this sequence, 6 is then the largest element in the input sequence.

For a sequence of length n , we can write the work and span for this algorithm as recurrences as follows

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

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

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

2 Reduce with Contraction

The *reduce* primitive performs a computation that involves applying an associative binary operation op to the elements of a sequence to obtain (reduce the sequence to) a final value. For example, reducing the sequence $\langle 0, 1, 2, 3, 4 \rangle$ with the $+$ operation gives us $0 + 1 + 2 + 3 + 4 = 10$. Recall that the type signature for *reduce* is as follows.

reduce ($f : \alpha * \alpha \rightarrow \alpha$) ($id : \alpha$) ($a : \mathbb{S}_\alpha$) : α ,

where f is a binary function, a is the sequence, and id is the left identity for f .

Even though we can define *reduce* broadly for both associative and non-associative functions, in this section, we assume that the function f is associative.

Generalizing the algorithm for computing the maximal element leads us to an implementation of an important parallelism primitive called *reduce*. The crux in using the contraction technique is to design an algorithm for reducing an instance of the problem to a geometrically smaller instance by performing a parallel contraction step. To see how this can be done, consider instead applying the function f to consecutive pairs of the input.

For example if we wish to compute the sum of the input sequence

$$\langle 2, 1, 3, 2, 2, 5, 4, 1 \rangle$$

by using the addition function, we can contract the sequence to

$$\langle 3, 5, 7, 5 \rangle.$$

Note that the contraction step can be performed in parallel, because each pair can be considered independently in parallel.

By using this contraction step, we have reduced the input size by a factor of two. We next solve the resulting problem by invoking the same algorithm and apply expansion to construct the final result. We note now that by solving the smaller problem, we obtain a solution to the original problem, because the sum of the sequence remains the same as that of the original. Thus, the expansion step requires no additional work.

Algorithm 25.2 (Reduce with Contraction). An algorithm for *reduce* using contraction is shown below; for simplicity, we assume that the input size is a power of two.

```
(* Assumption: |a| is a power of 2 *)
reduceContract f id a =
  if |a| = 1 then
    a[0]
  else
    let
      b = ⟨ f(a[2i], a[2i + 1]) : 0 ≤ i < [|a|/2] ⟩
    in
      reduceContract f id b
    end
```

Cost of Reduce with Contraction. Assuming that the function being reduced over performs constant work, parallel tabulate in the contraction step requires linear work, we can thus write the work of this algorithm as

$$W(n) = W(n/2) + n.$$

This recurrence solves to $O(n)$.

Assuming that the function being reduced over performs constant span, parallel tabulate in the contraction step requires constant span; we can thus write the work of this algorithm as

$$S(n) = S(n/2) + 1.$$

This recurrence solves to $O(\log n)$.

3 Scan with Contraction

We describe how to implement the *scan* sequence primitive efficiently by using contraction. Recall that the *scan* function has the type signature

$$\text{scan } (f : \alpha * \alpha \rightarrow \alpha) (id : \alpha) (a : \mathbb{S}_\alpha) : (\mathbb{S}_\alpha * \alpha)$$

where f is an associative function, a is a sequence, and id is the identity element of f . When evaluated with a function and a sequence, *scan* can be viewed as applying a reduction to every prefix of the sequence and returning the results of such reductions as a sequence.

Example 25.2. Applying *scan* ‘+’, i.e., “plus scan” on the sequence $\langle 2, 1, 3, 2, 2, 5, 4, 1 \rangle$ returns

$$(\langle 0, 2, 3, 6, 8, 10, 15, 19 \rangle, 20).$$

We will use this as a running example.

Based on its specification, a direct algorithm for *scan* is to apply a reduce to all prefixes of the input sequence. Unfortunately, this easily requires quadratic work in the size of the input sequence.

We can see that this algorithm is inefficient by noting that it performs lots of redundant computations. In fact, two consecutive prefixes overlap significantly but the algorithm does not take advantage of such overlaps at all, computing the result for each overlap independently.

By taking advantage of the fact that any two consecutive prefixes differ by just one element, it is not difficult to give a linear work algorithm (modulo the cost of the application of the argument function) by using iteration.

Such an algorithm may be expressed as follows

$$\text{scan } f \text{ id } a = h(\text{iterate } g(\langle \rangle, id) a),$$

where

$$g((b, y), x) = ((\text{append } \langle y \rangle b), f(y, x))$$

and

$$h(b, y) = ((\text{reverse } b), y)$$

where *reverse* reverses a sequence.

This algorithm is correct but it is almost entirely sequential, leaving no room for parallelism.

Scan via Contraction, the Intuition. Because *scan* has to compute some value for each prefix of the given sequence, it may appear to be inherently sequential. We might be inclined to believe that any efficient algorithms will have to keep a cumulative “sum,” computing each output value by relying on the “sum” of the all values before it.

We will now see that we can implement *scan* efficiently using contraction. To this end, we need to reduce a given problem instance to a geometrically smaller instance by applying a contraction step. As a starting point, let’s apply the same idea as we used for implementing [reduce with contraction](#).

Applying the contraction step from the *reduce* algorithm described above, we would reduce the input sequence

$$\langle 2, 1, 3, 2, 2, 5, 4, 1 \rangle$$

to the sequence

$$\langle 3, 5, 7, 5 \rangle,$$

which if recursively used as input would give us the result

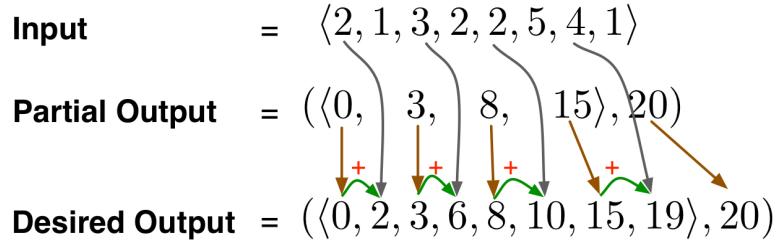
$$(\langle 0, 3, 8, 15 \rangle, 20).$$

Notice that in this sequence, the elements in even numbered positions are consistent with the desired result:

$$(\langle 0, 2, 3, 6, 8, 10, 15, 19 \rangle, 20).$$

Half of the elements are correct because the contraction step, which pairs up the elements and reduces them, does not affect, by associativity of the function being used, the result at a position that do not fall in between a pair.

To compute the missing element of the result, we will use an expansion step and compute each missing elements by applying the function element-wise to the corresponding elements in the input and the results of the recursive call to *scan*. The drawing below illustrates this expansion step.



Algorithm 25.3 (Scan Using Contraction, for Powers of 2). Based on the intuitive description above, we can write the pseudo-code for *scan* as follows. For simplicity, we assume that n is a power of two.

```
(* Assumption: |a| is a power of two.  *)
scan f id a =
  if |a| = 0 then
    (<>, id)
  else if |a| = 1 then
    (<id>, a[0])
  else
    let
      a' = < f(a[2i], a[2i + 1]) : 0 ≤ i < n/2 >
      (r, t) = scan f id a'
    in
      (< pi : 0 ≤ i < n >, t), where pi = { r[i/2]           even(i)
                                         f(r[i/2], a[i - 1])  otherwise
    end
```

Cost of Scan with Contraction. Let's assume for simplicity that the function being applied has constant work and constant span. We can write out the work and span for the algorithm as a recursive relation as

$$W(n) = W(n/2) + n, \text{ and}$$

$$S(n) = S(n/2) + 1,$$

because 1) the contraction step which tabulates the smaller instance of the problem performs linear work in constant span, and 2) the expansion step that constructs the output by tabulating based on the result of the recursive call also performs linear work in constant span.

These recursive relations should look familiar. They are the same as those that we ended up with when we analyzed the work and span of our contraction-based implementation of *reduce* and *yield*.

$$W(n) = O(n)$$

$$S(n) = O(\log n).$$

Chapter 26

Maximum Contiguous Subsequence Sum

This chapter reviews the classic problem of finding the contiguous subsequence of a sequence with the maximal value, and provides several algorithms for the problem by applying several design techniques including [brute force](#), [reduction](#), and [divide and conquer](#).

1 The Problem

Definition 26.1 (Subsequence). A *subsequence* b of a sequence a is a sequence that can be derived from a by deleting zero or more elements of a without changing the order of remaining elements.

Example 26.1. Several examples follow.

- The sequence $\langle 0, 2, 4 \rangle$ is a subsequence of $\langle 0, 1, 2, 2, 3, 4, 5 \rangle$.
- The sequence $\langle 2, 4, 3 \rangle$ is not a subsequence of $\langle 0, 1, 2, 2, 3, 4, 5 \rangle$ but $\langle 2, 3, 4 \rangle$ is.

Definition 26.2 (Contiguous Subsequence). A *contiguous subsequence* is a subsequence that appears contiguously in the original sequence. For any sequence a of n elements, the subsequence

$$b = a[i \dots j], 0 \leq i \leq j < n,$$

consisting of the elements of a at positions $i, i + 1, \dots, j$ is a contiguous subsequence of b .

Example 26.2. For $a = \langle 1, -2, 0, 3, -1, 0, 2, -3 \rangle$, here are some contiguous subsequences:

- $\langle 1 \rangle$,
- $\langle -2, 0, 3 \rangle$, and
- $\langle 3, -1, 0, 2, -3 \rangle$.

The sequence $\langle -1, 2, -3 \rangle$ is not a contiguous subsequence, even though it is a subsequence.

Definition 26.3 (The Maximum Contiguous Subsequence (MCS) Problem). Given a sequence of integers, the *Maximum Contiguous Subsequence Problem* (MCS) requires finding the contiguous subsequence of the sequence with maximum total sum, i.e.,

$$\text{MCS}(a) = \arg \max_{0 \leq i, j < |a|} \left(\sum_{k=i}^j a[k] \right).$$

We define the sum of an empty sequence to be $-\infty$.

Example 26.3. For $a = \langle 1, -2, 0, 3, -1, 0, 2, -3 \rangle$, a maximum contiguous subsequence is, $\langle 3, -1, 0, 2 \rangle$; another is $\langle 0, 3, -1, 0, 2 \rangle$.

Definition 26.4 (The Maximum Contiguous Subsequence Sum (MCSS) Problem). Given a sequence of integers, the *Maximum Contiguous Subsequence Sum Problem* (MCSS)

requires finding the total sum of the elements in the contiguous subsequence of the sequence with maximum total sum, i.e.,

$$\text{MCSS}(a) = \max \left\{ \sum_{k=i}^j a[k] : 0 \leq i, j < |a| \right\}.$$

Example 26.4. For $a = \langle 1, -2, 0, 3, -1, 0, 2, -3 \rangle$, a maximum contiguous subsequence is, $\langle 3, -1, 0, 2 \rangle$; another is $\langle 0, 3, -1, 0, 2 \rangle$. Thus $\text{MCSS}(a) = 4$.

For the empty sequence, $\text{MCSS} = -\infty$ because the sum of an empty sequence is defined as $-\infty$.

Note. Here we only consider sequences of integers and the addition operation to compute the sum, the techniques that we describe should apply to sequences of other types and other associative sum operations.

Lower Bound. To solve the MCSS problem, we need to inspect, at the very least, each and every element of the sequence. This requires linear work in the length of the sequence and thus solve the MCSS problem requires $\Omega(n)$ work.

Note (History of the Problem). The study of maximum contiguous subsequence problem goes to 1970's. The problem was first proposed in by Ulf Grenander, a Swedish statistician and a professor of applied mathematics at Brown University, in 1977. The problem has several names, such maximum subarray sum problem, or maximum segment sum problem, the former of which appears to be the name originally used by Grenander. Grenander intended it to be a simplified model for maximum likelihood estimation of patterns in digitized images, whose structure he wanted to understand.

According to Jon Bentley (Jon Bentley, Programming Pearls (1st edition), page 76.) in 1977, Grenander described the problem to Michael Shamos of Carnegie Mellon University who overnight designed a divide and conquer algorithm, which corresponds to our [first divide-and-conquer algorithm](#). When Shamos and Bentley discussed the problem and Shamos' solution, they thought that it was probably the best possible. A few days later Shamos described the problem and its history at a Carnegie Mellon seminar attended by statistician Joseph (Jay) Kadane, who designed the work efficient algorithm within a minute. Kadane's algorithm correspond to the [linear work and span algorithm](#) described below.

Roadmap. The remaining sections apply various algorithm-design techniques to the MCS and MCSS problems. To exercise our vocabulary for algorithm design, the content is organized to identify carefully the design techniques, sometimes at a level of precision that may, especially in subsequent reads, feel pedantic.

2 Brute Force

This section presents a first solution to the MCSS problem by using the [brute force technique](#).

Algorithm 26.5 (MCSS: Brutest Force). We can solve the MCSS problem by brute force. First, we identify the set of candidate solutions as the set of all integers. Then we enumerate all integers and, for each one, check that there is a contiguous subsequence whose sum is equal to that integer. We stop when we find the largest integer with a matching subsequence.

Perhaps obviously, such an algorithm would not terminate, because we don't know when to stop. Notice, however, that the solution is bounded by the sum of all positive integers in the sequence. We can thus stop the search when we reach that bound.

This algorithm terminates but has the undesirable characteristic that its bound depends on the values of the elements in the sequence rather than its length.

Reduction to MCS. [Our first algorithm](#) is rather inefficient in the worst case, because it tries a large number of candidate solutions. We can achieve a better bound by reducing MCSS problem to the [Maximum Contiguous Subsequence \(MCS\) problem](#), which

requires finding the contiguous subsequence with the largest sum.

The reduction itself is straightforward: because both problems operate on the same input, there is no need to transform the input. To compute the output, we sum the elements in the sequence returned by the MCS problem. Using *reduce*, this requires $O(n)$ work and $O(\lg n)$ span. Thus, the work and span of the reduction is $O(n)$ and $O(\lg n)$ respectively.

Algorithm 26.6 (MCS: Brute Force). We can solve the MCS problem by brute force: we enumerate all candidate solutions, which consist of all the contiguous subsequences of the sequence, and find the one with the largest sum. To generate all contiguous subsequences, we can generate all pairs of integers (i, j) , $0 \leq i \leq j < n$, compute the sum of the subsequence that corresponds to the pair, and pick the one with the largest total. We write the algorithm as follows:

```

MCSBF a =
  let
    maxSum ((i, j, s), (k, l, t)) = if s > t then (i, j, s) else (k, l, t)
    b = ⟨ (i, j, reduce + 0 a[i ··· j]) : 0 ≤ i ≤ j < n ⟩
    (i, j, s) = reduce maxSum (-1, -1, -∞) b
  in
    (i, j)
  end

```

Cost of Brute Force MCSS. Using array sequence costs, generating the n^2 subsequences requires a total of $O(n^2)$ work and $O(\lg n)$ span. Reducing over each subsequence using *reduce* adds linear work per subsequence, bringing the total work to $O(n^3)$. The final *reduce* that select the maximal subsequence require $O(n^2)$ work. The total work is thus dominated by the computation of sums for each subsequence, and therefore is $O(n^3)$.

Because we can generate all pairs in parallel and compute their sum in parallel in $\Theta(\lg n)$ span using *reduce*, the algorithm requires $\Theta(\lg n)$ span.

Algorithm 26.7 (MCSS: Brute Force). Our first algorithm uses brute force technique and a reduction to the MCS problem, which we again solve by brute force using the [brute-force MCS algorithm](#). We can write the algorithm as follows:

```

MCSSBF a =
  let
    (i, j) = MCSBF a
    sum = reduce '+' 0 a[i ··· j]
  in
    sum
  end.

```

Strengthening. The [brute force algorithm](#) has some redundancy: to find the solution, it computes the result for the MCS problem and then computes the sum of the result se-

quence, which is already computed by the MCS algorithm. We can eliminate this redundancy by strengthening the MCS problem and requiring it to return the sum in addition to the subsequence.

Exercise 26.1. Describe the changes to the algorithms Algorithm 26.6 and Algorithm 26.7 to implement the strengthening described above. How does strengthening impact the work and span costs?

Algorithm 26.8 (MCSS: Brute Force Strengthened). We can write the algorithm based on strengthening directly as follows. Because the problem description requires returning only the sum, we simplify the algorithm by not tracking the subsequences.

```
MCSSBF a =
  let
    b = ⟨ reduce + 0 a[i ··· j] : 0 ≤ i ≤ j < n ⟩
  in
    reduce max -∞ b
  end
```

Cost Analysis. Let's analyze the work and span of [the strengthened brute-force algorithm](#) by using [array sequences](#) and by appealing to our cost bounds for *reduce*, *subseq*, and *tabulate*. The cost bounds for enumerating all possible subsequences and computing their sums is as follows.

$$\begin{aligned} W(n) &= 1 + \sum_{1 \leq i \leq j \leq n} W_{\text{reduce}}(j - i) \leq 1 + n^2 \cdot W_{\text{reduce}}(n) = \Theta(n^3) \\ S(n) &= 1 + \max_{1 \leq i \leq j \leq n} S_{\text{reduce}}(j - i) \leq 1 + S_{\text{reduce}}(n) = \Theta(\lg n) \end{aligned}$$

The final step of the brute-force algorithm is to find the maximum over these $\Theta(n^2)$ combinations. Since the reduce for this step has $\Theta(n^2)$ work and $\Theta(\lg n)$ span the cost of the final step is subsumed by other costs analyzed above. Overall, we have an $\Theta(n^3)$ -work $\Theta(\lg n)$ -span algorithm.

Note. Note that the span requires the maximum over $\binom{n}{2} \leq n^2$ values, but since $\lg n^k = k \lg n$, this is simply $\Theta(\lg n)$.

Summary. When trying to apply the brute-force technique to the MCSS problem, we encountered a problem. We solved this problem by reducing MCSS problem to another problem, MCS. We then realized a redundancy in the resulting algorithm and eliminated that redundancy by strengthening MCS. This is a quite common route when designing a good algorithm: we find ourselves refining the problem and the solution until it is (close to) perfect.

3 Applying Reduction

In the previous section, we used the brute-force technique to develop an algorithm that has

logarithmic span but large (cubic) work. In this section, we apply the reduction technique to obtain a low span and work-efficient (linear work) algorithm for the MCSS problem.

3.1 Auxiliary Problems

Overlapping Subsequences and Redundancy. To understand how we might improve the amount of work, observe that the brute-force algorithm performs a lot of repeated and thus redundant work. To see why, consider the subsequences that start at some location. For each position, e.g., the middle, the algorithm considers a subsequence that starts at the position and at any other position that comes after it. Even though each subsequence differs from another by a single element (in the ending positions), the algorithm computes the total sum for each of these subsequences independently, requiring linear work per subsequence. The algorithm does not take advantage of the overlap between the many subsequences it considers.

Reducing Redundancy. We can reduce redundancy by taking advantage of the overlaps and computing all subsequences that start or end at a given position together. Our basic strategy in applying the reduction technique is to use this observation. To this end, we define two problems that are closely related to the MCSS problem, present efficient and parallel algorithm for these problems, and then reduce the MCSS to them.

Definition 26.9 (MCSSS). The *Maximum Contiguous Subsequence Sum with Start*, abbreviated **MCSSS**, problem requires finding the maximum contiguous subsequence of a sequence that starts at a given position.

Definition 26.10 (MCSSE Problem). The *Maximum Contiguous Subsequence with Ending*, i.e., the **MCSSE** problem requires finding the maximum contiguous subsequence ending at a specified end position.

Reducing MCSS to MCSSS and MCSSE. Observe that we can reduce the MCSS problem to MCSSS problem by enumerating over all starting positions, solving MCSSS for each position, and taking the maximum over the results. A similar reduction works for the MCSSE problem.

Because the inputs to all these problems are essentially the same, we don't need to transform the input. To compute the output for MCSS, we need to perform a *reduce*. The reduction itself is thus efficient.

Algorithm 26.11 (An Optimal Algorithm for MCSSS). We can solve the MCSSS problem by first computing the sum for all subsequences that start at the given position using *scan*

and then finding their maximum.

```

MCSSSOpt a i =
  let
    b = scanI +' 0 a [i … (|a| - 1)]
  in
    reduce max -∞ b
  end

```

Because the algorithm performs one scan and one reduce, it performs $\Theta(n - i)$ work in $\Theta(\lg n - i)$ span. This is asymptotically optimal because, any algorithm for the MCSSS problem must inspect at least $n - i$ elements of the input sequence.

Algorithm 26.12 (An Optimal Algorithm for MCSSS). To solve the MCSSE problem efficiently and in low span, we observe that any contiguous subsequence of a given sequence can be expressed in terms of the difference between two prefixes of the sequence: the subsequence $A[i \dots j]$ is equivalent to the difference between the subsequence $A[0 \dots j]$ and the subsequence $A[0 \dots i - 1]$.

Thus, we can compute the sum of the elements in a contiguous subsequence as

$$\text{reduce } +' 0 a[i \dots j] = (\text{reduce } +' 0 a[0 \dots j]) - (\text{reduce } +' 0 a[0 \dots i - 1])$$

where the “-” is the subtraction operation on integers.

This observation leads us to a solution to the MCSSE problem. Consider an ending position j and suppose that we have the sum for each prefix that ends at $i < j$. Since we can express any subsequence ending at position j by subtracting the corresponding prefix, we can compute the sum for the subsequence $A[i \dots j]$ by subtracting the sum for the prefix ending at j from the prefix ending at $i - 1$. Thus the maximum contiguous sequence ending at position j starts at position i which has the minimum of all prefixes up to i . We can compute the minimum prefix that comes before j by using just another scan. These observations lead to the following algorithm.

```

MCSSSEOpt a j =
  let
    (b, v) = scan +' 0 a[0 … j]
    w = reduce min ∞ b
  in
    v - w
  end

```

Using array sequences, this algorithm performs $\Theta(j)$ work and $\Theta(\lg(j))$ span. This is optimal because any algorithm for MCSSE must inspect at least j elements of the input sequence.

3.2 Reduction to MCSSS

Algorithm 26.13 (MCSS: Reduced Force). We can find a more efficient brute-force algorithm for MCSS by reducing the problem to MCSSS and using [the optimal algorithm for it](#).

The idea is to try all possible start positions, solve the MCSSS problem for each, and select their maximum. The code for the algorithm is shown below.

```
MCSSReducedForce a =
  reduce max -∞ ⟨ (MCSSSOpt a i) : 0 ≤ i < n ⟩ .
```

In the worst case, the algorithm performs $\Theta(n^2)$ work in $\Theta(\lg n)$ span, delivering a linear-factor improvement in work.

Remark. By reducing MCSS to MCSSS, we were able to eliminate a certain kind of redundancy: namely those that occur when solving for subsequences starting at a given position. In the next section, we will see, how to improve our bound further.

3.3 Reduction to MCSSE

Algorithm 26.14 (MCSS by Reduction to MCSSE). We can solve the MCSS problem by enumerating all instances of the MCSSE problem and selecting the maximum.

```
MCSSReducedForce2 a =
  reduce max -∞ ⟨ (MCSSEOpt a i) : 0 ≤ i < |a| ⟩
```

This algorithm has $O(n^2)$ work and $O(\lg n)$ span.

The two algorithms obtained by [reduction to MCSSS](#) and [reduction to MCSSE](#) both reduce some of the redundant work, but not all, because they don't reduce redundancies when solving for subsequences ending (or starting) at different positions. Next, we will see, how to eliminate these redundant computations.

Lemma 26.1 (MCSSE Extension). Suppose that we are given the maximum contiguous sequence, M_i ending at position i . We can compute the maximum contiguous sequence ending at position $i + 1$, M_{i+1} , from this by noticing that

- $M_{i+1} = M_i ++ \langle a[i] \rangle$, or
- $M_{i+1} = \langle a[i] \rangle$,

depending on the sum for each.

Exercise 26.2. Prove [the MCSSE Extension lemma](#).

The algorithm for solving MCSS by reduction to MCSSE solves many instances of MCSSE in parallel. If we give up some parallelism, it turns out that we can improve the work efficiency further based on [the MCSSE Extension lemma](#). The idea is to iterate over the sequence and solve the MCSSE problem for each ending position. To solve the MCSSE problem, we then take the maximum over all positions.

Algorithm 26.15 (MCSS with Iteration). The SPARC code for the algorithm for MCSS obtained by reduction to MCSSE is shown below. We use the function *iteratePrefixes* to iterate over the input sequence and construct a sequence whose i^{th} position contains the solution to the MCSSE problem at that position.

```

MCSSIterative a =
  let
    f (sum, x) =
      if sum + x ≥ x then
        sum + x
      else
        x
    b = iteratePrefixes f -∞ a
  in
    reduce max -∞ b
  end

```

Cost Analysis. Using array sequences, *iteratePrefixes* and *reduce* we are both linear work, because the functions f and \max both perform constant work. Because of *iteratePrefixes*, the span is also linear.

Algorithm 26.16 (MCSS: Work Optimal and Low Span). In our *scan*-based [algorithm for MCSSE](#), we used the observation that the maximal contiguous subsequence ending at a given position is identified by subtracting the prefix at the ending position from the minimum sum over all preceding prefixes. Our new algorithm, uses the same intuition but refines it further by noticing that

- we can compute the sum for all prefixes in one scan, and
- we can compute the minimum prefix sum preceding all positions in one scan.

After computing these quantities, all that remains is to take the difference and select the maximum.

```

MCSSOpt a =
let
  (b, v) = scan '+' 0 a
  c = append b ⟨ v ⟩
  (d, _) = scan min ∞ c
  e = ⟨ c[i] - d[i] : 0 < i < |a| ⟩
in
  reduce max -∞ e
end

```

Example 26.5. Consider the sequence a

$$a = \langle 1, -2, 0, 3, -1, 0, 2, -3 \rangle.$$

Compute

$$\begin{aligned} (b, v) &= \text{scan} + 0 a \\ c &= \text{append } b \langle v \rangle. \end{aligned}$$

We have $c = \langle 0, 1, -1, -1, 2, 1, 1, 3, 0 \rangle$.

The sequence c contains the prefix sums ending at each position, including the element at the position; it also contains the empty prefix.

Using the sequence c , we can find the minimum prefix up to all positions as

$$(d, _) = \text{scan} \text{ min } \infty c$$

to obtain

$$d = \langle \infty, 0, 0, -1, -1 - 1, -1, -1, -1 \rangle.$$

We can now find the maximum subsequence ending at any position i by subtracting the value for i in c from the value for all the prior prefixes calculated in d .

Compute

$$\begin{aligned} e &= \langle c[i] - d[i] : 0 < i < |a| \rangle \\ &= \langle 1, -1, 0, 3, 2, 2, 4, 1 \rangle. \end{aligned}$$

It is not difficult to verify in this small example that the values in e are indeed the maximum contiguous subsequences ending in each position of the original sequence. Finally, we take the maximum of all the values in e to compute the result

$$\text{reduce max } -\infty e = 4.$$

Cost of the Algorithm. Using array sequences, and the fact that addition and minimum take constant work, the algorithm performs $\Theta(n)$ work in $\Theta(\lg n)$ span. The algorithm is work optimal because any algorithm must inspect each and every element of the sequence to solve the MCSS problem.

4 Divide And Conquer

4.1 A First Solution

Dividing the Input. To apply divide and conquer, we first need to figure out how to divide the input. There are many possibilities, but dividing the input in two halves is usually a good starting point, because it reduces the input for both subproblems equally, reducing thus the size of the largest component, which is important in bounding the overall span. Correctness is usually independent of the particular strategy of division.

Let us divide the sequence into two halves, recursively solve the problem on both parts, and combine the solutions to solve the original problem.

Example 26.6. Let $a = \langle 1, -2, 0, 3, -1, 0, 2, -3 \rangle$. By using the approach, we divide the sequence into two sequences b and c as follows

$$b = \langle 1, -2, 0, 3 \rangle$$

and

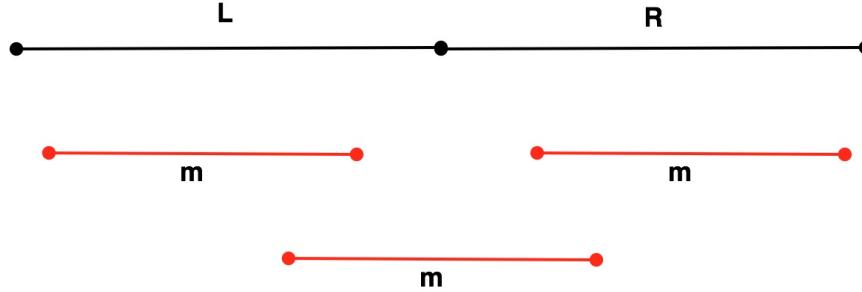
$$c = \langle -1, 0, 2, -3 \rangle$$

We can now solve each part to obtain 3 and 2 as the solutions to the subproblems. Note that there are multiple sequences that yield the maximum sum.

Using Solutions to Subproblems. To construct a solution for the original problem from those of the subproblems, let's consider where the solution subsequence might come from. There are three possibilities.

1. The maximum sum lies completely in the left subproblem.
2. The maximum sum lies completely in the right subproblem.
3. The maximum sum overlaps with both halves, spanning the cut.

The three cases are illustrated below



The first two cases are already solved by the recursive calls, but not the last. Assuming we can find the largest subsequence that spans the cut, we can write our algorithm as shown below.

Algorithm 26.17 (Simple Divide-and-Conquer for MCSS). Using a function called *bestAcross* to find the largest subsequence that spans the cut, we can write our algorithm as follows.

```

MCSSDC a =
  if |a| = 0 then
    -∞
  else if |a| = 1 then
    a[0]
  else
    let
      (b, c) = splitMid a
      (mb, mc) = (MCSSDC b || MCSSDC c)
      mbc = bestAcross (b, c)
    in
    max{mb, mc, mbc}
  end

```

Algorithm 26.18 (Maximum Subsequence Spanning the Cut). We can reduce the problem of finding the maximum subsequence spanning the cut to two problems that we have seen already: Maximum-Contiguous-Subsequence Sum with Start, MCSSS, and Maximum-Contiguous-Subsequence Sum at Ending, MCSSE. The maximum sum spanning the cut is the sum of the largest suffix on the left plus the largest prefix on the right. The prefix of the right part is easy as it directly maps to the solution of MCSSS problem at position 0. Similarly, the suffix for the left part is exactly an instance of MCSSE problem. We can thus use the algorithms that we have seen in the previous section for solving this problem, Algorithm 26.11, and, Algorithm 26.12 respectively.

The cost of both of these algorithms is $\Theta(n)$ work and $\Theta(\lg n)$ span and thus the total cost is also the same.

Example 26.7. In the example above, the largest suffix on the left is 3, which is given by the sequence $\langle 3 \rangle$ or $\langle 0, 3 \rangle$.

The largest prefix on the right is 1 given by the sequence $\langle -1, 0, 2 \rangle$. Therefore the largest sum that crosses the middle is $3 + 1 = 4$.

4.1.1 Correctness

To prove a divide-and-conquer algorithm correct, we use the technique of strong induction, which enables to assume that correctness remains correct for all smaller subproblems. We now present such a correctness proof for the algorithm *MCSSDC*.

Theorem 26.2 (Correctness of the algorithm *MCSSDC*). Let a be a sequence. The algorithm *MCSSDC* returns the maximum contiguous subsequence sum in a given sequence—and returns $-\infty$ if a is empty.

Proof. The proof will be by (strong) induction on length of the input sequence. Our induction hypothesis is that the theorem above holds for all inputs smaller than the current input.

We have two base cases: one when the sequence is empty and one when it has one element. On the empty sequence, the algorithm returns $-\infty$ and thus the theorem holds. On any singleton sequence $\langle x \rangle$, the MCSS is x , because

$$\max \left\{ \sum_{k=i}^j a[k] : 0 \leq i < 1, 0 \leq j < 1 \right\} = \sum_{k=0}^0 a[0] = a[0] = x.$$

The theorem therefore holds.

For the inductive step, let a be a sequence of length $n \geq 1$, and assume inductively that for any sequence a' of length $n' < n$, the algorithm correctly computes the maximum contiguous subsequence sum. Now consider the sequence a and let b and c denote the left and right subsequences resulted from dividing a into two parts (i.e., $(b, c) = \text{splitMida}$). Furthermore, let $a[i \dots j]$ be any contiguous subsequence of a that has the largest sum, and this value is v . Note that the proof has to account for the possibility that there may be many other subsequences with equal sum. Every contiguous subsequence must start somewhere and end after it. We consider the following 3 possibilities corresponding to how the sequence $a[i \dots j]$ lies with respect to b and c :

- If the sequence $a[i \dots j]$ starts in b and ends c . Then its sum equals its part in b (a suffix of b) and its part in c (a prefix of c). If we take the maximum of all suffixes in c and prefixes in b and add them this is equal the maximum of all contiguous sequences bridging the two, because $\max \{x + y : x \in X, y \in Y\} = \max \{x \in X\} + \max \{y \in Y\}$. By assumption this equals the sum of $a[i \dots j]$ which is v . Furthermore by induction m_b and m_c are sums of other subsequences so they cannot be any larger than v and hence $\max\{m_b, m_c, m_{bc}\} = v$.

- If $a[i \cdots j]$ lies entirely in b , then it follows from our inductive hypothesis that $m_b = v$. Furthermore m_c and m_{bc} correspond to the maximum sum of other subsequences, which cannot be larger than v . So again $\max\{m_b, m_c, m_{bc}\} = v$.
- Similarly, if $a_{i..j}$ lies entirely in c , then it follows from our inductive hypothesis that $m_c = \max\{m_b, m_c, m_{bc}\} = v$.

We conclude that in all cases, we return $\max\{m_b, m_c, m_{bc}\} = v$, as claimed. \square

4.1.2 Cost Analysis

By Algorithm 26.18, we know that the maximum subsequence crossing the cut in $\Theta(n)$ work and $\Theta(\lg n)$ span. Note also that *splitMid* requires $O(1)$ work and span for array sequences and $O(\lg n)$ work and span for tree sequences, so in either case the work and span are bounded by $O(\lg n)$. We thus have the following recurrences with array-sequence or tree-sequence specifications

$$\begin{aligned} W(n) &= 2W(n/2) + \Theta(n) \\ S(n) &= S(n/2) + \Theta(\lg n). \end{aligned}$$

Using the definition of big- Θ , we know that

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

where k_1 and k_2 are constants. By using the tree method, we can conclude that $W(n) = \Theta(n \lg n)$ and $S(n) = \lg^2 n$.

Solving the Recurrence Using Substitution Method. Let's now redo the recurrences above using the substitution method. Specifically, we'll prove the following theorem using (strong) induction on n .

Theorem 26.3. Let a constant $k > 0$ be given. If $W(n) \leq 2W(n/2) + k \cdot n$ for $n > 1$ and $W(1) \leq k$ for $n \leq 1$, then we can find constants κ_1 and κ_2 such that

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

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

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

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

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

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

4.2 Divide And Conquer with Strengthening

Our first divide-and-conquer algorithm performs $O(n \lg n)$ work, which is $O(\lg n)$ factor more than the optimal. In this section, we shall reduce the work to $O(n)$ by being more careful about avoiding redundant work.

Intuition. Our divide-and-conquer algorithm has an important redundancy: the maximum prefix and maximum suffix are computed recursively to solve the subproblems for the two halves but are computed again at the combine step of the divide-and-conquer algorithm.

Because these are computed as part of solving the subproblems, we could return them from the recursive calls. To do this, we will strengthen the problem so that it returns the maximum prefix and suffix. This problem, which we shall call **MCSSPS**, matches the original MCSS problem in its input and returns strictly more information. Solving MCSS using MCSSPS is therefore trivial. We thus focus on the MCSSPS problem.

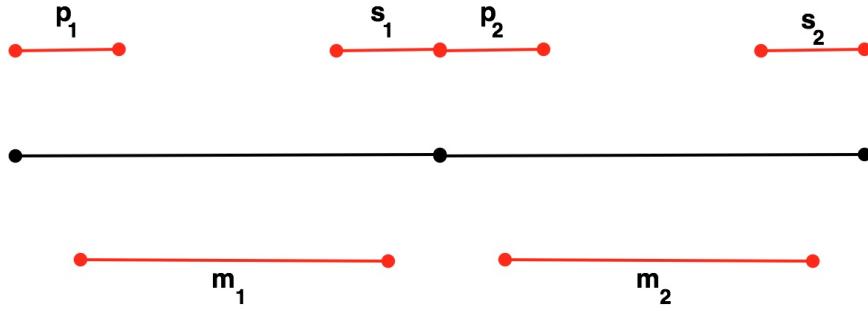
Solving MCSSPS. We can solve this problem by strengthening our divide-and-conquer algorithm from the previous section. We need to return a total of three values:

- the max subsequence sum,
- the max prefix sum, and
- the max suffix sum.

At the base cases, when the sequence is empty or consists of a single element, this is easy to do. For the recursive case, we need to consider how to produce the desired return values from those of the subproblems. Suppose that the two subproblems return (m_1, p_1, s_1) and (m_2, p_2, s_2) .

One possibility to compute as result

$$(\max(s_1 + p_2, m_1, m_2), p_1, s_2).$$



Note that we don't have to consider the case when s_1 or p_2 is the maximum, because that case is checked in the computation of m_1 and m_2 by the two subproblems.

This solution fails to account for the case when the suffix and prefix can span the whole sequence.

We can fix this problem by returning the total for each subsequence so that we can compute the maximum prefix and suffix correctly. Thus, we need to return a total of four values:

- the max subsequence sum,
- the max prefix sum,
- the max suffix sum, and
- the overall sum.

Having this information from the subproblems is enough to produce a similar answer tuple for all levels up, in constant work and span per level. Thus what we have discovered is that to solve the strengthened problem efficiently we have to strengthen the problem once again. Thus if the recursive calls return (m_1, p_1, s_1, t_1) and (m_2, p_2, s_2, t_2) , then we return

$$(\max(s_1 + p_2, m_1, m_2), \max(p_1, t_1 + p_2), \max(s_1 + t_2, s_2), t_1 + t_2).$$

Algorithm 26.19 (Linear Work Divide-and-Conquer MCSS).

```

 $MCSSDCAux\ a =$ 
   $\text{if } |a| = 0 \text{ then}$ 
     $(-\infty, -\infty, -\infty, 0)$ 
   $\text{else if } |a| = 1 \text{ then}$ 
     $(a[0], a[0], a[0], a[0])$ 
   $\text{else}$ 
     $\text{let}$ 
       $(b, c) = \text{splitMid } a$ 
       $((m_1, p_1, s_1, t_1), (m_2, p_2, s_2, t_2)) = (MCSSDCAux\ b \ || \ MCSSDCAux\ c)$ 
       $\text{in}$ 
         $(\max(s_1 + p_2, m_1, m_2),$ 
         $\max(p_1, t_1 + p_2),$ 
         $\max(s_1 + t_2, s_2),$ 
         $t_1 + t_2)$ 
       $\text{end}$ 
     $\text{end}$ 
 $MCSSDC\ a =$ 
   $\text{let}$ 
     $(m, -, -, -) = MCSSDCAux\ a$ 
   $\text{in}$ 
     $m$ 
   $\text{end}$ 

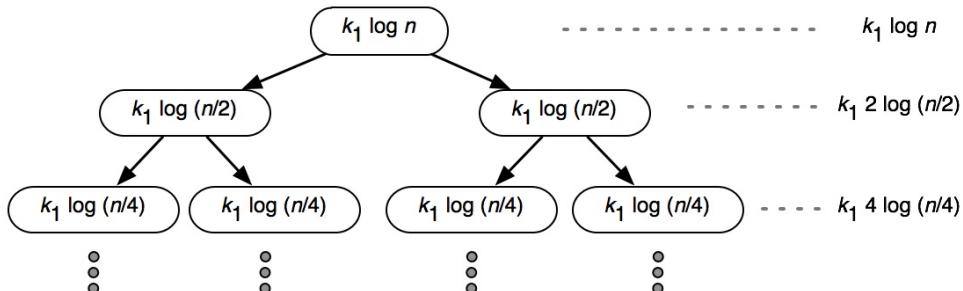
```

Cost Analysis. Since $splitMid$ requires $O(\lg n)$ work and span in both array and tree sequences, we have

$$W(n) = 2W(n/2) + O(\lg n)$$

$$S(n) = S(n/2) + O(\lg n).$$

The $O(\lg n)$ bound on *splitMid* is not tight for array sequences, where *splitMid* requires $O(1)$ work, but this loose upper bound suffices to achieve the bound on the work that we seek. Note that the span is the same as before, so we'll focus on analyzing the work. Using the tree method, we have



Therefore, the total work is upper-bounded by

$$W(n) \leq \sum_{i=0}^{\lg n} k_1 2^i \lg(n/2^i)$$

It is not so obvious to what this sum evaluates, but we can bound it as follows:

$$\begin{aligned} W(n) &\leq \sum_{i=0}^{\lg n} k_1 2^i \lg(n/2^i) \\ &= \sum_{i=0}^{\lg n} k_1 (2^i \lg n - i \cdot 2^i) \\ &= k_1 \left(\sum_{i=0}^{\lg n} 2^i \right) \lg n - k_1 \sum_{i=0}^{\lg n} i \cdot 2^i \\ &= k_1 (2n - 1) \lg n - k_1 \sum_{i=0}^{\lg n} i \cdot 2^i. \end{aligned}$$

We're left with evaluating $s = \sum_{i=0}^{\lg n} i \cdot 2^i$. Observe that if we multiply s by 2, we have

$$2s = \sum_{i=0}^{\lg n} i \cdot 2^{i+1} = \sum_{i=1}^{1+\lg n} (i-1)2^i,$$

so then

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

Substituting this back into the expression we derived earlier, we have $W(n) \leq k_1(2n - 1) \lg n - 2k_1(n \lg n - n + 1) \in O(n)$ because the $n \lg n$ terms cancel.

We can solve the recurrence by using substitution method also. We'll make a guess that $W(n) \leq \kappa_1 n - \kappa_2 \lg n - \kappa_3$. More precisely, we prove the following theorem.

Theorem 26.4. Let $k > 0$ be given. If $W(n) \leq 2W(n/2) + k \cdot \lg n$ for $n > 1$ and $W(n) \leq k$ for $n \leq 1$, then we can find constants κ_1, κ_2 , and κ_3 such that

$$W(n) \leq \kappa_1 \cdot n - \kappa_2 \cdot \lg n - \kappa_3.$$

Proof. Let $\kappa_1 = 3k, \kappa_2 = k, \kappa_3 = 2k$. We begin with the base case. Clearly, $W(1) = k \leq \kappa_1 - \kappa_3 = 3k - 2k = k$. For the inductive step, we substitute the inductive hypothesis into

the recurrence and obtain

$$\begin{aligned}
 W(n) &\leq 2W(n/2) + k \cdot \lg n \\
 &\leq 2(\kappa_1 \frac{n}{2} - \kappa_2 \lg(n/2) - \kappa_3) + k \cdot \lg n \\
 &= \kappa_1 n - 2\kappa_2(\lg n - 1) - 2\kappa_3 + k \cdot \lg n \\
 &= (\kappa_1 n - \kappa_2 \lg n - \kappa_3) + (k \lg n - \kappa_2 \lg n + 2\kappa_2 - \kappa_3) \\
 &\leq \kappa_1 n - \kappa_2 \lg n - \kappa_3,
 \end{aligned}$$

where the final step uses the fact that $(k \lg n - \kappa_2 \lg n + 2\kappa_2 - \kappa_3) = (k \lg n - k \lg n + 2k - 2k) = 0 \leq 0$ by our choice of κ 's. \square

Part VI

Probability

Chapter 27

Introduction

This part covers discrete probability theory. Chapter 28 describes probability spaces, events, probabilities and conditional probabilities. Chapter 29 describes random variables. Chapter 30 presents expectations, which can be used to summarize random variables by their average or mean value.

Chapter 28

Probability Spaces

This chapter introduces the basics of discrete probability theory.

1 Probability Spaces and Events

Probability theory is a mathematical study of uncertain situations such as a dice game. In probability theory, we model a situation with an uncertain *outcome* as an *experiment* and reason carefully about the likelihood of various outcomes in precise mathematical terms.

Example 28.1. Suppose we have two *fair* dice, meaning that each is equally likely to land on any of its six sides. If we toss the dice, what is the chance that their numbers sum to 4? To determine the probability we first notice that there are a total of $6 \times 6 = 36$ distinct outcomes. Of these, only three outcomes sum to 4 (1 and 3, 2 and 2, and 3 and 1). The probability of the event that the number sum up to 4 is therefore

$$\frac{\text{\# of outcomes that sum to 4}}{\text{\# of total possible outcomes}} = \frac{3}{36} = \frac{1}{12}$$

Sample Spaces and Events. A *sample space* Ω is an arbitrary and possibly infinite (but countable) set of possible outcomes of a probabilistic experiment. Any experiment will return exactly one outcome from the set. For the dice game, the sample space is the 36 possible outcomes of the dice, and an experiment (roll of the dice) will return one of them. An *event* is any subset of Ω , and most often representing some property common to multiple outcomes. For example, an event could correspond to outcomes in which the dice add to 4—this subset would be of size 3. We typically denote events by capital letters from the start of the alphabet, e.g. A, B, C . We often refer to the individual elements of Ω as *elementary events*. We assign a probability to each event. Our model for probability is defined as follows.

Definition 28.1 (Probability Space). A probability space consists of a *sample space* Ω representing the set of possible outcomes, and a *probability measure*, which is a function \mathbf{P} from all subsets of Ω (the *events*) to a probability (real number). These must satisfy the following axioms.

- **Nonnegativity:** $\mathbf{P}[A] \in [0, 1]$.
- **Additivity:** for any two disjoint events A and B (i.e., $A \cap B = \emptyset$),

$$\mathbf{P}[A \cup B] = \mathbf{P}[A] + \mathbf{P}[B].$$

- **Normalization:** $\mathbf{P}[\Omega] = 1$.

Note (Infinite Spaces). Probability spaces can have countably infinite outcomes. The additivity rule generalizes to infinite sums, e.g., the probability of the event consisting of the union of infinitely many number of disjoint events is the infinite sum of the probability of each event.

Note. When defining the probability space, we have not specified carefully the exact nature of events, because they may differ based on the experiment and what we are interested in. We do, however, need to take care when setting up the probabilistic model so that we can reason about the experiment correctly. For example, each outcome of the sample space must correspond to one unique actual outcome of the experiment. In other words, they must be mutually exclusive. Similarly, any actual outcome of the experiment must have a corresponding representation in the sample space.

Example 28.2 (Throwing Dice). For our example of throwing two dice, the sample space consists of all of the 36 possible pairs of values of the dice:

$$\Omega = \{(1, 1), (1, 2), \dots, (2, 1), \dots, (6, 6)\}.$$

Each pair in the sample space corresponds to an outcome of the experiment. The outcomes are mutually exclusive and cover all possible outcomes of the experiment.

For example, having the first dice show up 1 and the second 4 is an outcome and corresponds to the element $(1, 4)$ of the sample space Ω .

The event that the “the first dice is 3” corresponds to the set

$$\begin{aligned} A &= \{(d_1, d_2) \in \Omega \mid d_1 = 3\} \\ &= \{(3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (3, 6)\}. \end{aligned}$$

The event that “the dice sum to 4” corresponds to the set

$$\begin{aligned} B &= \{(d_1, d_2) \in \Omega \mid d_1 + d_2 = 4\} \\ &= \{(1, 3), (2, 2), (3, 1)\}. \end{aligned}$$

Assuming the dice are unbiased, the probability measure is defined by all elementary events having equal probability, i.e.,

$$\forall x \in \Omega, \quad \mathbf{P}[\{x\}] = \frac{1}{36}.$$

The probability of the event A (that the first dice is 3) is thus

$$\mathbf{P}[A] = \sum_{x \in A} \mathbf{P}[\{x\}] = \frac{6}{36} = \frac{1}{6}.$$

If the dice were biased so the probability of a given value is proportional to that value, then the probability measure would be $\mathbf{P}[\{(x, y)\}] = \frac{x}{21} \times \frac{y}{21}$, and the probability of the event B (that the dice add to 4) would be

$$\mathbf{P}[B] = \sum_{x \in B} \mathbf{P}[\{x\}] = \frac{1 \times 3 + 2 \times 2 + 3 \times 1}{21 \times 21} = \frac{10}{441}.$$

2 Properties of Probability Spaces

Given a probability space, we can prove several properties of probability measures by using the three axioms that they must satisfy.

For example, if for two events A and B . We have

- if $A \subseteq B$, then $\mathbf{P}[A] \leq \mathbf{P}[B]$,
- $\mathbf{P}[A \cup B] = \mathbf{P}[A] + \mathbf{P}[B] - \mathbf{P}[A \cap B]$.

2.1 The Union Bound

The union bound, also known as Boole's inequality, is a simple way to obtain an upper bound on the probability of any of a collection of events happening. Specifically for a collection of events A_0, A_1, \dots, A_{n-1} the bound is:

$$\mathbf{P}\left[\bigcup_{0 \leq i < n} A_i\right] \leq \sum_{i=0}^{n-1} \mathbf{P}[A_i]$$

This bound is true unconditionally. To see why the bound holds we note that the elementary events in the union on the left are all included in the sum on the right (since the union comes from the same set of events). In fact they might be included multiple times in the sum on the right, hence the inequality. In fact the sum on the right could add to more than one, in which case the bound is not useful. The union bound can be useful in generating high-probability bounds for algorithms. For example, when the probability of each of n events is very low, e.g. $1/n^5$ and the sum remains very low, e.g. $1/n^4$.

2.2 Conditional Probability

Conditional probability allows us to reason about dependencies between observations. For example, suppose that your friend rolled a pair of dice and told you that they sum up to 6, what is the probability that one of dice has come up 1? Conditional probability has many practical applications. For example, given that a medical test for a disease comes up positive, we might want to know the probability that the patient has the disease. Or, given that your computer has been working fine for the past 2 years, you might want to know the probability that it will continue working for one more year.

Definition 28.2 (Conditional Probability). For a given probability space, we define the *conditional probability* of an event A given B , as the probability of A occurring given that B occurs as

$$\mathbf{P}[A | B] = \frac{\mathbf{P}[A \cap B]}{\mathbf{P}[B]}.$$

The conditional probability measures the probability that the event A occurs given that B does. It is defined only when $\mathbf{P}[B] > 0$.

Conditional Probability is a Probability Measure. Conditional probability satisfies the three axioms of probability measures and is itself a probability measure. We can thus treat conditional probabilities just as ordinary probabilities. Intuitively, conditional probability can be thought as a focusing and re-normalization of the probabilities on the assumed event B .

Example 28.3. Consider throwing two fair dice and calculate the probability that the first dice comes up 1 given that the sum of the two dice is 4. Let A be the event that the first dice comes up 1 and B the event that the sum is 4. We can write A and B in terms of outcomes as

$$\begin{aligned} A &= \{(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6)\} \text{ and} \\ B &= \{(1, 3), (2, 2), (3, 1)\}. \end{aligned}$$

We thus have $A \cap B = \{(1, 3)\}$. Since each outcome is equally likely,

$$\mathbf{P}[A | B] = \frac{\mathbf{P}[A \cap B]}{\mathbf{P}[B]} = \frac{|A \cap B|}{|B|} = \frac{1}{3}.$$

2.3 Law of Total Probability

Conditional probabilities can be useful in estimating the probability of an event that may depend on a selection of choices. The total probability theorem can be handy in such circumstances.

Theorem 28.1 (Law of Total Probability). Consider a probabilistic space with sample space Ω and let A_0, \dots, A_{n-1} be a partition of Ω such that $\mathbf{P}[A_i] > 0$ for all $0 \leq i < n$. For any event B the following holds:

$$\begin{aligned}\mathbf{P}[B] &= \sum_{i=0}^{n-1} \mathbf{P}[B \cap A_i] \\ &= \sum_{i=0}^{n-1} \mathbf{P}[A_i] \mathbf{P}[B | A_i]\end{aligned}$$

Example 28.4. Your favorite social network partitions your connections into two kinds, near and far. The social network has calculated that the probability that you react to a post by one of your far connections is 0.1 but the same probability is 0.8 for a post by one of your near connections. Suppose that the social network shows you a post by a near and far connection with probability 0.6 and 0.4 respectively.

Let's calculate the probability that you react to a post that you see on the network. Let A_0 and A_1 be the event that the post is near and far respectively. We have $\mathbf{P}[A_0] = 0.6$ and $\mathbf{P}[A_1] = 0.4$. Let B the event that you react, we know that $\mathbf{P}[B | A_0] = 0.8$ and $\mathbf{P}[B | A_1] = 0.1$.

We want to calculate $\mathbf{P}[B]$, which by total probability theorem we know to be

$$\begin{aligned}\mathbf{P}[B] &= \mathbf{P}[B \cap A_0] + \mathbf{P}[B \cap A_1] \\ &= \mathbf{P}[A_0] \mathbf{P}[B | A_0] + \mathbf{P}[A_1] \mathbf{P}[B | A_1]. \\ &= 0.6 \cdot 0.8 + 0.4 \cdot 0.1 \\ &= 0.52.\end{aligned}$$

2.4 Independence

It is sometimes important to reason about the dependency relationship between events. Intuitively we say that two events are independent if the occurrence of one does not affect the probability of the other. More precisely, we define independence as follows.

Definition 28.3 (Independence). Two events A and B are *independent* if

$$\mathbf{P}[A \cap B] = \mathbf{P}[A] \cdot \mathbf{P}[B].$$

We say that multiple events A_0, \dots, A_{n-1} are *mutually independent* if and only if, for any non-empty subset $I \subseteq \{0, \dots, n-1\}$,

$$\mathbf{P}\left[\bigcap_{i \in I} A_i\right] = \prod_{i \in I} \mathbf{P}[A_i].$$

Independence and Conditional Probability. Recall that $\mathbf{P}[A | B] = \frac{\mathbf{P}[A \cap B]}{\mathbf{P}[B]}$ when $\mathbf{P}[B] > 0$. Thus if $\mathbf{P}[A | B] = \mathbf{P}[A]$ then $\mathbf{P}[A \cap B] = \mathbf{P}[A] \cdot \mathbf{P}[B]$. We can thus define independence in terms of conditional probability but this works only when $\mathbf{P}[B] > 0$.

Example 28.5. For two dice, the events $A = \{(d_1, d_2) \in \Omega \mid d_1 = 1\}$ (the first dice is 1) and $B = \{(d_1, d_2) \in \Omega \mid d_2 = 1\}$ (the second dice is 1) are independent since

$$\begin{aligned}\mathbf{P}[A] \times \mathbf{P}[B] &= \frac{1}{6} \times \frac{1}{6} = \frac{1}{36} \\ &= \mathbf{P}[A \cap B] = \mathbf{P}[\{(1, 1)\}] = \frac{1}{36}.\end{aligned}$$

However, the event $C \equiv \{X = 4\}$ (the dice add to 4) is not independent of A since

$$\begin{aligned}\mathbf{P}[A] \times \mathbf{P}[C] &= \frac{1}{6} \times \frac{3}{36} = \frac{1}{72} \\ &\neq \mathbf{P}[A \cap C] = \mathbf{P}[\{(1, 3)\}] = \frac{1}{36}.\end{aligned}$$

A and C are not independent since the fact that the first dice is 1 increases the probability they sum to 4 (from $\frac{1}{12}$ to $\frac{1}{6}$).

Exercise 28.1. For two dice, let A be the event that first roll is 1 and B be the event that the sum of the rolls is 5. Are A and B independent? Prove or disprove.

Consider now the same question but this time define B to be the event that the sum of the rolls is 7.

Chapter 29

Random Variables

This chapter introduces the random variables and their use in probability theory.

Definition 29.1 (Random Variable). A *random variable* X is a real-valued function on the outcomes of an experiment, i.e., $X : \Omega \rightarrow \mathbb{R}$, i.e., it assigns a real number to each outcome. For a given probability space there can be many random variables, each keeping track of different quantities. We typically denote random variables by capital letters from the end of the alphabet, e.g. X , Y , and Z . We say that a random variable is *discrete* if its range is finite or countable infinite. Throughout this book, we only consider discrete random variables.

Example 29.1. For throwing two dice, we can define random variable as the sum of the two dice

$$X(d_1, d_2) = d_1 + d_2 ,$$

the product of two dice

$$Y(d_1, d_2) = d_1 \times d_2 ,$$

or the value of the first dice the two dice:

$$Z(d_1, d_2) = d_1 .$$

Definition 29.2 (Indicator Random Variable). A random variable is called an *indicator random variable* if it takes on the value 1 when some condition is true and 0 otherwise.

Example 29.2. For throwing two dice, we can define indicator random variable as getting doubles

$$Y(d_1, d_2) = \begin{cases} 1 & \text{if } d_1 = d_2 \\ 0 & \text{if } d_1 \neq d_2 . \end{cases}$$

Using our shorthand, the event $\{X = 4\}$ corresponds to the event “the dice sum to 4”.

Notation. For a random variable X and a value $x \in \mathbb{R}$, we use the following shorthand for the event corresponding to X equaling x :

$$\{X = x\} \equiv \{y \in \Omega \mid X(y) = x\} ,$$

and when applying the probability measure we use the further shorthand

$$\mathbf{P}[X = x] \equiv \mathbf{P}[\{X = x\}] .$$

Example 29.3. For throwing two dice, and X being a random variable representing the sum of the two dice, $\{X = 4\}$ corresponds to the event “the dice sum to 4”, i.e. the set

$$\{y \in \Omega \mid X(y) = 4\} = \{(1, 3), (2, 2), (3, 1)\} .$$

Assuming unbiased coins, we have that

$$\mathbf{P}[X = 4] = 1/12 .$$

Remark. The term random variable might seem counter-intuitive because it is actually a function not a variable. As such, a random variable is not really random, because it is a well defined deterministic function on the sample space. However, when thought in conjunction with the random experiment that selects the events, a random variable takes on its value based on a random process.

1 Probability Mass Function

Definition 29.3 (Probability Mass Function). For a discrete random variable X , we define its *probability mass function* or *PMF*, written $\mathbf{P}_X(\cdot)$, for short as a function mapping each element x in the range of the random variable to the probability of the event $\{X = x\}$, i.e.,

$$\mathbf{P}_X(x) = \mathbf{P}[X = x] .$$

Example 29.4. The probability mass function for the indicator random variable X indicating whether the outcome of a roll of dice is comes up even is

$$\begin{aligned} \mathbf{P}_X(0) &= \mathbf{P}[\{X = 0\}] = \mathbf{P}[\{1, 3, 5\}] = 1/2, \text{ and} \\ \mathbf{P}_X(1) &= \mathbf{P}[\{X = 1\}] = \mathbf{P}[\{2, 4, 6\}] = 1/2. \end{aligned}$$

The probability mass function for the random variable X that maps each outcome in a roll of dice to the smallest Mersenne prime number no less than the outcome is

$$\begin{aligned} \mathbf{P}_X(3) &= \mathbf{P}[\{X = 3\}] = \mathbf{P}[\{1, 2, 3\}] = 1/2, \text{ and} \\ \mathbf{P}_X(7) &= \mathbf{P}[\{X = 7\}] = \mathbf{P}[\{4, 5, 6\}] = 1/2. \end{aligned}$$

Note that much like a probability measure, a probability mass function is a non-negative function. It is also additive in a similar sense: for any distinct x and x' , the events $\{X = x\}$ and $\{X = x'\}$ are disjoint. Thus for any set \bar{x} of values of X , we have

$$\mathbf{P}[X \in \bar{x}] = \sum_{x \in \bar{x}} \mathbf{P}_X(x).$$

Furthermore, since X is a function on the sample space, the events corresponding to the different values of X partition the sample space, and we have

$$\sum_x \mathbf{P}_X(x) = 1.$$

These are the important properties of probability mass functions: they are non-negative, normalizing, and are additive in a certain sense.

We can also compute the probability mass function for multiple random variables defined for the same probability space. For example, the *joint probability mass function* for two random variables X and Y , written $\mathbf{P}_{X,Y}(x, y)$ denotes the probability of the event $\{X = x\} \cap \{Y = y\}$, i.e.,

$$\mathbf{P}_{X,Y}(x, y) = \mathbf{P}[\{X = x\} \cap \{Y = y\}] = \mathbf{P}[X = x, Y = y].$$

Here $\mathbf{P}[X = x, Y = y]$ is shorthand for $\mathbf{P}[\{X = x\} \cap \{Y = y\}]$.

In our analysis or randomized algorithms, we shall repeatedly encounter a number of well-known random variables and create new ones from existing ones by composition.

2 Bernoulli, Binomial, and Geometric RVs

Bernoulli Random Variable. Suppose that we toss a coin that comes up a head with probability p and a tail with probability $1 - p$. The *Bernoulli random variable* takes the value 1 if the coin comes up heads and 0 if it comes up tails. In other words, it is an indicator random variable indicating heads. Its probability mass function is

$$\mathbf{P}_X(x) = \begin{cases} p & \text{if } x = 1 \\ 1 - p & \text{if } x = 0. \end{cases}$$

Binomial Random Variable. Consider n Bernoulli trials with probability p . We call the random variable X denoting the number of heads in the n trials as the *Binomial random variable*. Its probability mass function for any $0 \leq x \leq n$ is

$$\mathbf{P}_X(x) = \binom{n}{x} p^x (1 - p)^{n-x}.$$

Geometric Random Variable. Consider performing Bernoulli trials with probability p until the coin comes up heads and X denote the number of trials needed to observe the first head. The random variable X is called the *geometric random variable*. Its probability mass function for any $0 \leq x$ is

$$\mathbf{P}_X(x) = (1 - p)^{x-1} p.$$

3 Functions of Random Variables

It is often useful to “apply” a function to one or more random variables to generate a new random variable. Specifically if we have a function $f : \mathbb{R} \rightarrow \mathbb{R}$ and a random variable X we can compose the two giving a new random variable:

$$Y(x) = f(X(x))$$

We often write this shorthand as $Y = f(X)$. Similarly for two random variables X and Y we write $Z = X + Y$ as shorthand for

$$Z(x) = X(x) + Y(x)$$

or equivalently

$$Z = \lambda x. (X(x) + Y(x))$$

The probability mass function for the new variable can be computed by “massing” the probabilities for each value. For example, for a function of a random variable $Y = f(X)$, we can write the probability mass function as

$$\mathbf{P}_Y(y) = \mathbf{P}[Y = y] = \sum_{x \mid f(x)=y} \mathbf{P}_X(x).$$

Example 29.5. Let X be a Bernoulli random variable with parameter p . We can define a new random variable Y as a transformation of X by a function $f(\cdot)$. For example, $Y = f(X) = 9X + 3$ is random variable that transforms X , e.g., $X = 1$ would be transformed to $Y = 12$. The probability mass function for Y reflects that of X . Its probability mass function is

$$\mathbf{P}_Y(y) = \begin{cases} p & \text{if } y = 12 \\ 1 - p & \text{if } y = 3. \end{cases}$$

Example 29.6. Consider the random variable X with the probability mass function

$$\mathbf{P}_X(x) = \begin{cases} 0.25 & \text{if } x = -2 \\ 0.25 & \text{if } x = -1 \\ 0.25 & \text{if } x = 0 \\ 0.25 & \text{if } x = 1 \end{cases}$$

We can calculate the probability mass function for the random variable $Y = X^2$ as follows $\mathbf{P}_Y(y) = \sum_{x \mid x^2=y} \mathbf{P}_X(x)$. This yields

$$\mathbf{P}_Y(y) = \begin{cases} 0.25 & \text{if } y = 0 \\ 0.5 & \text{if } y = 1 \\ 0.25 & \text{if } y = 4. \end{cases}$$

4 Conditioning

In the same way that we can condition an event on another, we can also condition a random variable on an event or on another random variable. Consider a random variable X and an event A in the same probability space, we define the *conditional probability mass function* of X conditioned on A as

$$\mathbf{P}_{X \mid A} = \mathbf{P}[X = x \mid A] = \frac{\mathbf{P}[\{X = x\} \cap A]}{\mathbf{P}[A]}.$$

Since for different values of x , $\{X = x\} \cap A$'s are disjoint and since X is a function over the sample space, conditional probability mass functions are normalizing just like ordinary probability mass functions, i.e., $\sum_{x \in X} \mathbf{P}_{X \mid A}(x) = 1$. Thus just as we can treat conditional probabilities as ordinary probabilities, we can treat conditional probability mass functions also as ordinary probability mass functions.

Example 29.7. Roll a pair of dice and let X be the sum of the face values. Let A be the event that the second roll came up 6. We can find the conditional probability mass function

$$\begin{aligned} \mathbf{P}_{X \mid A}(x) &= \frac{\mathbf{P}[\{X = x\} \cap A]}{\mathbf{P}[A]} \\ &= \begin{cases} \frac{1/36}{1/6} = 1/6 & \text{if } x = 7, \dots, 12. \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Conditional Probability Mass Function. Since random variables closely correspond with events, we can condition a random variable on another. More precisely, let X and Y be two random variables defined on the same probability space. We define the *conditional probability mass function* of X with respect to Y as

$$\mathbf{P}_{X \mid Y}(x \mid y) = \mathbf{P}[X = x \mid Y = y].$$

We can rewrite this as

$$\begin{aligned} \mathbf{P}_{X \mid Y}(x \mid y) &= \mathbf{P}[X = x \mid Y = y] \\ &= \frac{\mathbf{P}[X = x, Y = y]}{\mathbf{P}[Y = y]} \\ &= \frac{\mathbf{P}_{X,Y}(x,y)}{\mathbf{P}_Y y}. \end{aligned}$$

Conditional PMFs are PMFs. Consider the function $\mathbf{P}_{X|Y}(x|y)$ for a fixed value of y . This is a non-negative function of x , the event corresponding to different values of x are disjoint, and they partition the sample space, the conditional mass functions are normalizing

$$\sum_x \mathbf{P}_{X|Y}(x|y) = 1.$$

Conditional probability mass functions thus share the same properties as probability mass functions.

By direct implication of its definition, we can use conditional probability mass functions to calculate joint probability mass functions as follows

$$\begin{aligned}\mathbf{P}_{X,Y}(x,y) &= \mathbf{P}_X(x)\mathbf{P}_{Y|X}(y|x) \\ \mathbf{P}_{X,Y}(x,y) &= \mathbf{P}_Y(y)\mathbf{P}_{X|Y}(x|y).\end{aligned}$$

As we can compute total probabilities from conditional ones as we saw earlier in this section, we can calculate marginal probability mass functions from conditional ones:

$$\mathbf{P}_X(x) = \sum_y \mathbf{P}_{X,Y}(x,y) = \sum_y \mathbf{P}_Y(y)\mathbf{P}_{X|Y}(x|y).$$

5 Independence

As with the notion of independence between events, we can also define independence between random variables and events. We say that a random variable X is *independent of an event A* , if

$$\text{for all } x : \mathbf{P}[\{X = x\} \cap A] = \mathbf{P}[X = x] \cdot \mathbf{P}[A].$$

When $\mathbf{P}[A]$ is positive, this is equivalent to

$$\mathbf{P}_{X|A}(x) = \mathbf{P}_X(x).$$

Generalizing this to a pair of random variables, we say a random variable X is *independent of a random variable Y* if

$$\text{for all } x, y : \mathbf{P}[X = x, Y = y] = \mathbf{P}[X = x] \cdot \mathbf{P}[Y = y]$$

or equivalently

$$\text{for all } x, y : \mathbf{P}_{X,Y}(x,y) = \mathbf{P}_X(x) \cdot \mathbf{P}_Y(y).$$

In our two dice example, a random variable X representing the value of the first dice and a random variable Y representing the value of the second dice are independent. However X is not independent of a random variable Z representing the sum of the values of the two dice.

Chapter 30

Expectation

This chapter introduces expectation and its use in probability theory.

1 Definitions

The *expectation* of a random variable X in a probability space (Ω, \mathbf{P}) is the sum of the random variable over the elementary events weighted by their probability, specifically:

$$\mathbf{E}_{\Omega, \mathbf{P}}[X] = \sum_{y \in \Omega} X(y) \cdot \mathbf{P}[\{y\}].$$

For convenience, we usually drop the (Ω, \mathbf{P}) subscript on \mathbf{E} since it is clear from the context.

Example 30.1. Assuming unbiased dice ($\mathbf{P}[(d_1, d_2)] = 1/36$), the expectation of the random variable X representing the sum of the two dice is:

$$\mathbf{E}[X] = \sum_{(d_1, d_2) \in \Omega} X(d_1, d_2) \times \frac{1}{36} = \sum_{(d_1, d_2) \in \Omega} \frac{d_1 + d_2}{36} = 7.$$

If we bias the coins so that for each dice the probability that it shows up with a particular value is proportional to the value, we have $\mathbf{P}[(d_1, d_2)] = (d_1/21) \times (d_2/21)$ and:

$$\mathbf{E}[X] = \sum_{(d_1, d_2) \in \Omega} \left((d_1 + d_2) \times \frac{d_1}{21} \times \frac{d_2}{21} \right) = 8 \frac{2}{3}.$$

It is usually more natural to define expectations in terms of the probability mass function of the random variable

$$\mathbf{E}[X] = \sum_x x \cdot \mathbf{P}_X(x).$$

Example 30.2. The expectation of an indicator random variable X is the probability that the associated predicate is true (i.e. that $X = 1$):

$$\begin{aligned}\mathbf{E}[X] &= 0 \cdot \mathbf{P}_X(0) + 1 \cdot \mathbf{P}_X(1). \\ &= \mathbf{P}_X(1).\end{aligned}$$

Example 30.3. Recall that the probability mass function for a Bernoulli random variable is

$$\mathbf{P}_X(x) = \begin{cases} p & \text{if } x = 1 \\ 1 - p & \text{if } x = 0. \end{cases}$$

Its expectation is thus

$$E[X] = p \cdot 1 + (1 - p) \cdot 0 = p.$$

Example 30.4. Recall that the probability mass function for geometric random variable X with parameter p is

$$\mathbf{P}_X(x) = (1 - p)^{x-1} p.$$

The expectation of X is thus

$$\begin{aligned}E[X] &= \sum_{x=1}^{\infty} x \cdot (1 - p)^{x-1} p \\ &= p \cdot \sum_{x=1}^{\infty} x \cdot (1 - p)^{x-1}\end{aligned}$$

Bounding this sum requires some basic manipulation of sums. Let $q = (1 - p)$ and rewrite the sum as $p \cdot \sum_{x=0}^{\infty} xq^{x-1}$. Note now the term xq^{x-1} is the derivative of q^x with respect to q . Since the sum $\sum_{x=0}^{\infty} q^x = 1/(1 - q)$, its derivative is $1/(1 - q)^2 = 1/p^2$. We thus have conclude that $E[X] = 1/p$.

Example 30.5. Consider performing two Bernoulli trials with probability of success $1/4$. Let X be the random variable denoting the number of heads.

The probability mass function for X is

$$\mathbf{P}_X(x) = \begin{cases} 9/16 & \text{if } x = 0 \\ 3/8 & \text{if } x = 1 \\ 1/16 & \text{if } x = 2. \end{cases}$$

Thus $\mathbf{E}[X] = 0 + 1 \cdot 3/8 + 2 \cdot 1/16 = 7/8$.

2 Composing Expectations

Recall that functions or random variables are themselves random variables (defined on the same probability space), whose probability mass functions can be computed by considering the random variables involved. We can thus also compute the expectation of a random variable defined in terms of others. For example, we can define a random variable Y as a function of another variable X as $Y = f(X)$. The expectation of such a random variable can be calculated by computing the probability mass function for Y and then applying the formula for expectations. Alternatively, we can compute the expectation of a function of a random variable X directly from the probability mass function of X as

$$E[Y] = E[f(X)] = \sum_x f(x) \mathbf{P}_X(x).$$

Similarly, we can calculate the expectation for a random variable Z defined in terms of other random variables X and Y defined on the same probability space, e.g., $Z = g(X, Y)$, as computing the probability mass function for Z or directly as

$$E[Z] = E[g(X, Y)] = \sum_{x,y} g(x, y) \mathbf{P}_{X,Y}(x, y).$$

These formulas generalize to function of any number of random variables.

3 Linearity of Expectations

An important special case of functions of random variables is the linear functions. For example, let $Y = f(X) = aX + b$, where $a, b \in \mathbb{R}$.

$$\begin{aligned} \mathbf{E}[Y] &= \mathbf{E}[f(X)] &= \mathbf{E}[aX + b] \\ &= \sum_x f(x) \mathbf{P}_X(x) &= \sum_x (ax + b) \mathbf{P}_X(x) \\ &= a \sum_x x \mathbf{P}_X(x) + b \sum_x \mathbf{P}_X(x) &= a \mathbf{E}[X] + b. \end{aligned}$$

Similar to the example, above we can establish that the linear combination of any number of random variables can be written in terms of the expectations of the random variables. For example, let $Z = aX + bY + c$, where X and Y are two random variables. We have

$$\mathbf{E}[Z] = \mathbf{E}[aX + bY + c] = a\mathbf{E}[X] + b\mathbf{E}[Y] + c.$$

The proof of this statement is relatively simple.

$$\begin{aligned}
 \mathbf{E}[Z] &= \mathbf{E}[aX + bY + c] \\
 &= \sum_{x,y} (ax + by + c)\mathbf{P}_{X,Y}(x, y) \\
 &= a \sum_{x,y} x\mathbf{P}_{X,Y}(x, y) + b \sum_{x,y} y\mathbf{P}_{X,Y}(x, y) + \sum_{x,y} c\mathbf{P}_{X,Y}(x, y) \\
 &= a \sum_x \sum_y x\mathbf{P}_{X,Y}(x, y) + b \sum_y \sum_x y\mathbf{P}_{X,Y}(x, y) + \sum_{x,y} c\mathbf{P}_{X,Y}(x, y) \\
 &= a \sum_x x \sum_y \mathbf{P}_{X,Y}(x, y) + b \sum_y y \sum_x \mathbf{P}_{X,Y}(x, y) + \sum_{x,y} c\mathbf{P}_{X,Y}(x, y) \\
 &= a \sum_x x\mathbf{P}_X(x) + b \sum_y y\mathbf{P}_Y(y) + c \\
 &= a\mathbf{E}[X] + b\mathbf{E}[Y] + c.
 \end{aligned}$$

An interesting consequence of this proof is that the random variables X and Y do not have to be defined on the same probability space. They can be defined for different experiments and their expectation can still be summed. To see why note that we can define the joint probability mass function $\mathbf{P}_{X,Y}(x, y)$ by taking the Cartesian product of the sample spaces of X and Y and spreading probabilities for each arbitrarily as long as the marginal probabilities, $\mathbf{P}_X(x)$ and $\mathbf{P}_Y(y)$ remain unchanged.

The property illustrated by the example above is known as the *linearity of expectations*. The linearity of expectations is very powerful often greatly simplifying analysis. The reasoning generalizes to the linear combination of any number of random variables.

Linearity of expectation occupies a special place in probability theory, the idea of replacing random variables with their expectations in other mathematical expressions do not generalize. Probably the most basic example of this is multiplication of random variables. We might ask is $\mathbf{E}[X] \times \mathbf{E}[Y] = \mathbf{E}[X \times Y]$? It turns out it is true when X and Y are independent, but otherwise it is generally not true. To see that it is true for independent random variables we have (we assume x and y range over the values of X and Y respectively):

$$\begin{aligned}
 \mathbf{E}[X] \times \mathbf{E}[Y] &= (\sum_x x\mathbf{P}[\{X = x\}]) \left(\sum_y y\mathbf{P}[\{Y = y\}] \right) \\
 &= \sum_x \sum_y (xy\mathbf{P}[\{X = x\}]\mathbf{P}[\{Y = y\}]) \\
 &= \sum_x \sum_y (xy\mathbf{P}[\{X = x\} \cap \{Y = y\}]) \text{ due to independence} \\
 &= \mathbf{E}[X \times Y]
 \end{aligned}$$

For example, the expected value of the product of the values on two (independent) dice is therefore $3.5 \times 3.5 = 12.25$.

Example 30.6. In [a previous example](#), we analyzed the expectation on X , the sum of the two dice, by summing across all 36 elementary events. This was particularly messy for the biased dice. Using linearity of expectations, we need only calculate the expected value of each dice, and then add them. Since the dice are the same, we can in fact just multiply by two. For example for the biased case, assuming X_1 is the value of one dice:

$$\begin{aligned}
 \mathbf{E}[X] &= 2\mathbf{E}[X_1] \\
 &= 2 \times \sum_{d \in \{1, 2, 3, 4, 5, 6\}} d \times \frac{d}{21} \\
 &= 2 \times \frac{1+4+9+16+25+36}{21} \\
 &= 8 \frac{2}{3}.
 \end{aligned}$$

4 Conditional Expectation

Definition 30.1 (Conditional Expectation). We define the conditional expectation of a random variable X for a given value y of Y as

$$\mathbf{E}[X | Y = y] = \sum_x x \mathbf{P}_{X|Y}(x | y).$$

Theorem 30.1 (Total Expectations Theorem). The expectation of a random variable can be calculated by “averaging” over its conditional expectation given another random variable:

$$\mathbf{E}[X] = \sum_y \mathbf{P}_Y(y) \mathbf{E}[X | Y = y].$$

5 Variance and Standard Deviation

Definition 30.2 (Variance). The variance of a random variable is defined as

$$\sigma^2 = \mathbf{E}[(X - \mu)^2].$$

Definition 30.3 (Standard Deviation). The variance of a random variable is defined as the square-root of its variance.

$$\sigma = \sqrt{\mathbf{E}[(X - \mu)^2]}.$$

6 Markov's Inequality

Consider a non-negative random variable X . We can ask how much can X exceed its expected value. Because the expectation is taken by averaging X over all outcomes, and it cannot take on negative values, X cannot take on a much larger value with significant probability. If it did it would contribute too much to the sum. More generally X cannot be a multiple of β larger than its expectation with probability greater than $1/\beta$. This is because this part on its own would contribute more than $\beta \mathbf{E}[X] \times \frac{1}{\beta} = \mathbf{E}[X]$ to the expectation, which is a contradiction.

Theorem 30.2 (Markov's Inequality). If X is a non-negative random variable, then

$$\mathbf{P}[X \geq \alpha] \leq \frac{\mathbf{E}[X]}{\alpha}.$$

or equivalently (by substituting $\alpha = \beta \mathbf{E}[X]$)

$$\mathbf{P}[X \geq \beta \mathbf{E}[X]] \leq \frac{1}{\beta}.$$

Proof. Let Y be a random variable that lower bounds X as follows

$$Y = \begin{cases} 0 & \text{if } X < \alpha \\ \alpha & \text{otherwise.} \end{cases}$$

By definition, $Y \leq X$ and therefore $\mathbf{E}[Y] \leq \mathbf{E}[X]$. Furthermore,

$$\begin{aligned} \mathbf{E}[Y] &= \alpha \mathbf{P}[Y = \alpha] \\ &= \alpha \mathbf{P}[X \geq \alpha] \leq \mathbf{E}[X]. \end{aligned}$$

Thus it follows that $\mathbf{P}[X \geq \alpha] \leq \mathbf{E}[X] / \alpha$. \square

7 Chebyshev's Inequality

[Markov's inequality](#) applies only to non-negative random variables. Chebyshev's inequality generalizes Markov's for all random variables. It basically states that a random variable strays away from its mean slowly as controlled by its variance.

Theorem 30.3 (Chebyshev's Inequality). If X is a random variable with expectation μ and variance σ^2 , then

$$\mathbf{P}[|X - \mu| \geq \gamma] \leq \frac{\sigma^2}{\gamma^2}.$$

or equivalently

$$\mathbf{P}[|X - \mu| \geq k\sigma] \leq \frac{1}{k^2},$$

Proof. Define the random variable $Y = (X - \mu)^2$ and note that Y is non-negative. Apply now [Markov's inequality](#) with $\alpha = \gamma^2$ to obtain

$$\mathbf{P}[(X - \mu)^2 \geq \gamma^2] \leq \frac{\mathbf{E}[(X - \mu)^2]}{\gamma^2}.$$

Note now that the event $(X - \mu)^2 \geq \gamma^2$ is the same as $|X - \mu| \geq \gamma$ and [by the definition of variance of random variable](#), we have

$$\mathbf{P}[|X - \mu| \geq \gamma] \leq \frac{\sigma^2}{\gamma^2}.$$

Substituting $\gamma = k\sigma$ yields the second form. \square

Chapter 31

Introduction

This chapter introduces randomized algorithms and presents an overview of the techniques used for analyzing them.

1 Randomized Algorithms

Definition 31.1 (Randomized Algorithm). We say that an algorithm is *randomized* if it makes random choices. Algorithms typically make their random choices by consulting a *source of randomness* such as a (pseudo-)random number generator.

Example 31.1. A classic randomized algorithm is the quick-sort algorithm, which selects a random element, called the pivot, and partitions the input into two by comparing each element to the pivot. To select a randomly chosen pivot from n elements, the algorithm needs $\lg n$ bits of random information, usually drawn from a pseudo-random number generator.

Definition 31.2 (Las Vegas and Monte Carlo Algorithms). There are two distinct uses of randomization in algorithms.

The first method, which is more common, is to use randomization to weaken the cost guarantees, such as the work and span, of the algorithm. That is, randomization is used to organize the computation in such a way that the impact is on the cost but not on the correctness. Such algorithms are called *Las Vegas algorithms*.

Another approach is to use randomization to weaken the correctness guarantees of the computation: an execution of the algorithm might or might not return a correct answer. Such algorithms are called *Monte Carlo algorithms*.

Note. In this book, we only use Las Vegas algorithms. Our algorithm thus always return the correct answer, but their costs (work and span) will depend on random choices.

Random Distance Run. Every year around the middle of April the Computer Science Department at Carnegie Mellon University holds an event called the “Random Distance Run”. It is a running event around the track, where the official die tosser rolls a die immediately before the race is started. The die indicates how many initial laps everyone has to run. When the first person is about to complete the laps, the die is rolled again to determine the additional laps to be run. Rumor has it that Carnegie Mellon scientists have successfully used their knowledge of probabilities to train for the race and to adjust their pace during the race (e.g., how fast to run at the start).

Thanks to Carnegie Mellon CSD PhD Tom Murphy for the design of the 2007 T-shirt.



1.1 Advantages of Randomization

Randomization is used quite frequently in algorithm design, because of its several advantages.

- **Simplicity:** randomization can simplify the design of algorithms, sometimes dramat-

ically.

- **Efficiency:** randomization can improve efficiency, e.g., by facilitating “symmetry breaking” without relying on communication and coordination.
- **Robustness:** randomization can improve the robustness of an algorithm, e.g., by reducing certain biases.

Example 31.2 (Primality Testing). A classic example where randomization simplifies algorithm design is primality testing. The problem of *primality testing* requires determining whether a given integer is prime. By applying the theories of primality developed by Russian mathematician Arthujov, Miller and Rabin developed a simple randomized algorithm for the problem that only requires polynomial work. For over 20 years it was not known whether the problem could be solved in polynomial work without randomization. Eventually a polynomial time algorithm was developed, but it is more complex and computationally more costly than the randomized version. Hence in practice everyone still uses the randomized version.

Definition 31.3 (Symmetry Breaking). In algorithm design, the term *symmetry breaking* refers to an algorithm’s ability to distinguish between choices that otherwise look equivalent. For example, parallel algorithms sometimes use symmetry breaking to select a portion of a larger structure, such as a subset of the vertices of a graph, by making local decisions without necessarily knowing the whole of the structure. Randomization can be used to implement symmetry breaking.

Example 31.3. Suppose that we wish to select a subsequence of a sequence under the condition that no adjacent elements are selected. For example, given the sequence

$$\langle 0, 1, 2, 3, 4, 5, 6, 7 \rangle,$$

we could select

$$\langle 0, 2, 4, 6 \rangle,$$

or

$$\langle 1, 3, 6 \rangle,$$

but not

$$\langle 0, 1, 3, 6 \rangle.$$

An algorithm can make such a selection by “flipping a coin” for each element and selecting the element if it has flipped heads and its next (following) neighbor has flipped tails.

Exercise 31.1. Prove that the algorithm described above is correct.

Exercise 31.2. What is the expected length of the selected subsequence in terms of the length of the input sequence?

Algorithmic Bias. As algorithms take on more and more sophisticated decisions, the questions of bias naturally arise. Deterministic algorithm are particularly susceptible to creating bias when they make decisions based on partial or imperfect decisions and repeat that same decisions on many problem instances. Randomness can reduce such bias, e.g., by choosing one of the alternatives based on some probability distribution.

Example 31.4. Consider the algorithm operating a self driving vehicle. In certain circumstances, the algorithm might have to choose one of two actions none of which are desirable. If the algorithm makes its decisions deterministically, then it will repeat the same decision in identical situations, leading to a bias towards one of the bad choices. Such an algorithm could thus make biased decisions, which in total could lead to ethically unacceptable outcomes.

1.2 Disadvantages of Randomization

Complexity of Analysis. Even though randomization can simplify algorithms, it usually complicates their analysis. Because we only have to analyze an algorithm once, but use it many times, we consider this cost to be acceptable.

Uncertainty. Randomization can increase uncertainty. For example, a randomized algorithm could get unlucky in the random choices that it makes and take a long time to compute the answer. In some applications, such as real-time systems, this uncertainty may be unacceptable.

Example 31.5. The randomized quicksort algorithm can perform anywhere from $\Omega(n^2)$ to $\Theta(n \lg n)$ work and a run of quicksort that requires $\Omega(n^2)$ work could take a very long time to complete. Depending on the application, it can therefore be important to improve the algorithm to avoid this worst case.

2 Analysis of Randomized Algorithms

Definition 31.4 (Expected and High-Probability Bounds). In analyzing costs for randomized algorithms there are two types of bounds that are useful: expected bounds, and high-probability bounds.

- *Expected bounds* inform us about the average cost across all random choices made by the algorithm.
- *High-probability* bounds inform us that it is very unlikely that the cost will be above some bound. For an algorithm, we say that some property is true with *high probability* if it is true with probability $p(n)$ such that

$$\lim_{n \rightarrow \infty} (p(n)) = 1,$$

where n is an algorithm specific parameter, which is usually the instance size.

As the terms suggest, expected bounds characterize average-case behavior whereas high-probability bounds characterize the common-case behavior of an algorithm.

Example 31.6. If an algorithm has $\Theta(n)$ expected work, it means that when averaged over all random choices it makes in all runs, the algorithm performs $\Theta(n)$ work. Because expected bounds are averaged over all random choices in all possible runs, there can be runs that require more or less work. For example once in every $1/n$ tries the algorithm might require $\Theta(n^2)$ work, and (or) once in every \sqrt{n} tries the algorithm might require $\Theta(n^{3/2})$ work.

Example 31.7. As an example of a high-probability bound, suppose that we have n experiments where the probability that work exceeds $O(n \lg n)$ is $1/n^k$. We can use the union bound to prove that the total probability that the work exceeds $O(n \lg n)$ is at most $n \cdot 1/n^k = 1/n^{k-1}$. This means that the work is $O(n \lg n)$ with probability at least $1 - 1/n^{k-1}$. If $k > 2$, then we have a high probability bound of $O(n \lg n)$ work.

Remark. In computer science, the function $p(n)$ in the definition of high probability is usually of the form $1 - \frac{1}{n^k}$ where n is the instance size or a similar measure and k is some constant such that $k \geq 1$.

Analyzing Expected Work. Expected bounds are quite convenient when analyzing work (or running time in traditional sequential algorithms). This is because the linearity of expectations allows adding expectations across the components of an algorithm to get the overall expected work. For example, if the algorithm performs n tasks each of which take on average 2 units of work, then the total work on average across all tasks will be $n \times 2 = 2n$ units.

Analyzing Expected Span. When analyzing span, expectations are less helpful, because bounding span requires taking the maximum of random variables, rather than their sum. And the expectation of the maximum of two random variables is not equal to the maximum of expectations of the random variables. To bound the span, we will usually need stronger guarantees than expectation, typically in the form of high probability bounds. High-probability bounds allow us to bound the expectation of the maximum of a number of random variables by showing that it is highly unlikely for any one of them to be large.

Exercise 31.3. Consider a game in which we draw some number of tasks at random such that a task has length n with probability $1/n$ and has length 1 otherwise. The expected length of a task is therefore bounded by 2. Imagine now drawing n tasks and waiting for all them to complete, assuming that each task can proceed in parallel independently of other tasks. Prove that the expected completion time is not constant.

Note. The exercise corresponds closely to computing the span of a computation, because the time waited depends on the length of the longest task.

Exercise 31.4. Repeat the same exercise with slightly different probabilities: a randomly chosen task has length n with probability $1/n^3$ and 1 otherwise. Prove now that the expected completion time is bounded by a constant.

Chapter 32

Order Statistics

This chapter presents the problem of computing the order statistics of a sequence and a randomized algorithm for this problem.

1 The Order Statistics Problem

Definition 32.1 (Order Statistics Problem). Given a sequence, an integer k where $0 \leq k < |a|$, and a comparison operation $<$ that defines a total order over the elements of the sequence, find the k^{th} **order statistics**, i.e., k^{th} smallest element (counting from zero) in the sequence.

Reduction to Sorting. We can solve this problem by reducing it to sorting: we first sort the input and then select the k^{th} element. Assuming that comparisons require constant work, the resulting algorithm requires $O(n \lg n)$ work, but we wish to do better: in particular we would like to achieve linear work and $O(\lg^2 n)$ span.

2 Randomized Algorithm for Order Statistics

This section presents a randomized algorithm for computing order statistics that uses the contraction technique: it solves a given problem instance by reducing it to a problem instance whose size is geometrically smaller in expectation.

Algorithm 32.2 (Contraction-Based Select). For the purposes of simplicity, let's assume that sequences consist of unique elements and consider the following algorithm that uses randomization to contract the problem to a smaller instance. The algorithm divides the

input into left and right sequences, ℓ and r , and figures out the side k^{th} smallest must be in, and explores that side recursively. When exploring the right side, r , the algorithm adjusts the parameter k because the elements less or equal to the pivot p are being thrown out (there are $|a| - |r|$ such elements).

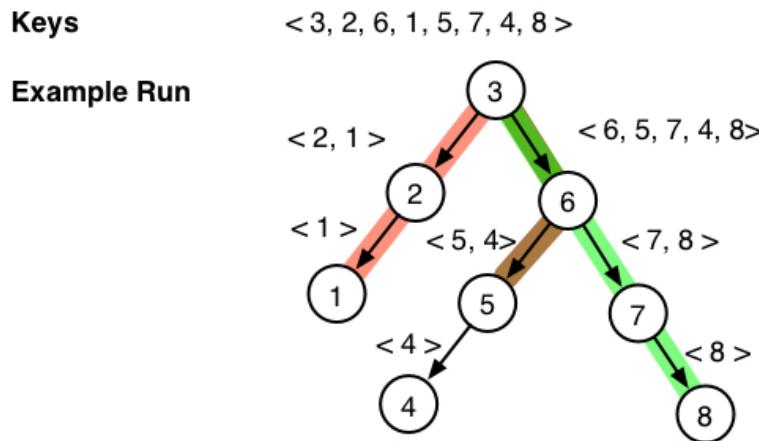
```

1  select a k =
2  let
3    p = pick a uniformly random element from a
4     $\ell = \{x \in a \mid x < p\}$ 
5     $r = \{x \in a \mid x > p\}$ 
6  in
7    if ( $k < |\ell|$ ) then select  $\ell$  k
8    else if ( $k < |a| - |r|$ ) then p
9    else select  $r$  ( $k - (|a| - |r|)$ )
10 end

```

Example 32.1. Example runs of `select` illustrated by a *pivot tree*. For illustrative purposes, we show all possible recursive calls being explored down to singleton sequences. In reality, the algorithm explores only one path.

- The path highlighted with red is the path of recursive calls taken by `select` when searching for the first-order statistics, $k = 0$.
- The path highlighted with brown is the path of recursive calls taken by `select` when searching for the fifth-order statistics, $k = 4$.
- The path highlighted with green is the path of recursive calls taken by `select` when searching for the eighth-order statistics, $k = 7$.



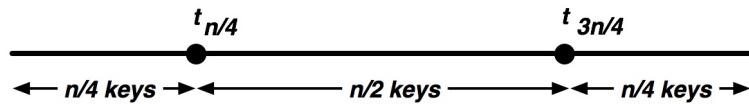
3 Analysis

We analyze the work and span of [the randomized algorithm for order statistics](#) and show that the `select` algorithm on input of size n performs $O(n)$ work in expectation and has $O(\lg^2 n)$ span with high probability.

4 Intuitive Analysis

Before we cover the more precise analysis, let's develop some intuition by considering the probability that we split the input sequence more or less evenly.

Observation. Recall that the rank of an element in a sequence is the position of the element in the corresponding sorted sequence. Consider the rank of the pivot selected at a call to `select`. If the selected pivot has rank greater than $n/4$ and less than $3n/4$, then the size of the input passed to the next recursive call is at most $3n/4$. Because all elements are equally likely to be selected as a pivot the probability that the selected pivot has rank greater than $n/4$ and less than $3n/4$ is $\frac{3n/4 - n/4}{n} = 1/2$. The figure below illustrates this.



Need for Luck. The observation above implies that at each recursive call, the instance size decreases by a constant fraction of $3/4$ with probability $1/2$. Thus if we are lucky, we successfully decrease the input size by a constant fraction.

But what if we are unlucky? Consider two successive recursive calls, the probability that the instance size decreases by $3/4$ after two calls is the probability that it decreases in either, which is at least $1 - \frac{1}{2} \cdot \frac{1}{2} = \frac{3}{4}$. More generally, after $c > 1$ such successive calls, the probability that the size decreases by a factor of $\frac{3}{4}$ is $1 - \frac{1}{2^c}$.

Note that $\lim_{c \rightarrow \infty} 1 - \frac{1}{2^c} = 1$. For $c = 10$, this probability is 0.999 and thus we are highly likely to get lucky after a small number of calls. Thus intuitively speaking, we expect `select` to complete in $c \lg n$ depth (recursive calls) for some constant c . Because the algorithm performs linear work and logarithmic span to filter the input sequence and then recurs, we can write the total work as a geometrically decreasing sum totaling up to $O(n)$ and span is $O(\lg^2 n)$.

4.1 Complete Analysis

We now present an analysis that makes precise described in [the intuitive analysis](#).

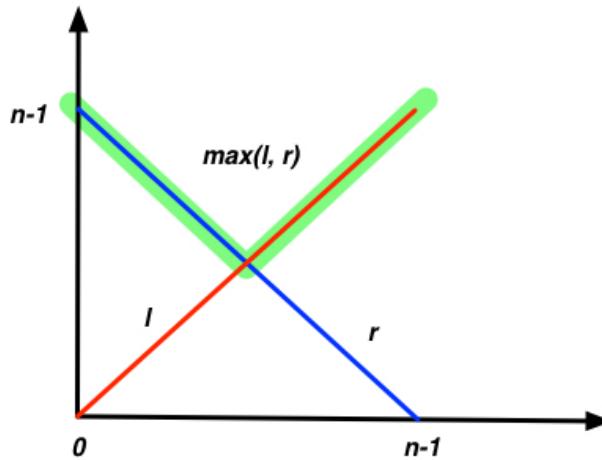
The Recurrences. Let $n = |a|$ and consider the partition of a into ℓ and r . Define

$$X(n) = \frac{\max\{|\ell|, |r|\}}{n}$$

as the fractional size of the larger side. Because *filter* requires linear work and logarithmic span, we can write the work and span of the algorithm as

$$\begin{aligned} W(n) &\leq W(X(n) \cdot n) + O(n) \\ S(n) &\leq S(X(n) \cdot n) + O(\lg n). \end{aligned}$$

Bounding the Expected Fraction. For the analysis, we will bound $\mathbf{E}[X(n)]$, i.e., the expected fraction of the instance size solved by the recursive call. Because all elements are equally likely to be chosen, we can calculate the size of ℓ and size of r as a function of the *rank* of the pivot, i.e., its position in the sort of the input sequence. If the pivot has rank i , then ℓ has length i and r has length $n - i - 1$. The drawing illustrates the sizes of ℓ and r and their maximum as a function of the rank of the pivot.



Since the algorithm chooses the pivot uniformly randomly, i.e., with probability $1/n$, we can write the expectation for $X(n)$ as

$$\mathbf{E}[X(n)] = \frac{1}{n} \sum_{i=0}^{n-1} \frac{\max\{i, n-i-1\}}{n} \leq \frac{1}{n} \sum_{j=n/2}^{n-1} \frac{2}{n} \cdot j \leq \frac{3}{4}$$

(Recall that $\sum_{i=x}^y i = \frac{1}{2}(x+y)(y-x+1)$.)

Important. Note that expectation bound holds for all input sizes n .

The calculation of $\mathbf{E}[X(n)]$ tells us that in expectation, $X(n)$ is a smaller than 1. Thus when bounding the work we should have a nice geometrically decreasing sum that adds up to $O(n)$. But it is not quite so simple, because the constant fraction is only in expectation. For example, we could get unlucky for a few contraction steps and leading to little or no reduction in the size of the input sequence.

We next show that even if we are unlucky on some steps, the expected size will indeed go down geometrically. Together with the linearity of expectations this will allow us to bound the work.

Theorem 32.1 (Expected Size of Input). Starting with size n , the expected size of a in algorithm `select` after d recursive calls is at most $\left(\frac{3}{4}\right)^d n$.

Proof. The proof is by induction on the depth of the recursion d . In the base case, $d = 0$ and the lemma holds trivially. For the inductive case assume that the lemma holds for some $d \geq 0$. Consider now the $(d+1)^{th}$ recursive call. Let Y_d be the random variable denoting the instance size and let Z_d denote the pivot chosen, at the depth d^{th} . For any value of y and z , let $f(y, z)$ be the fraction of the input reduced by the choice of the pivot at position z for an input of size y . We can write the expectation for the input size at $(d+1)^{st}$ call as

$$\begin{aligned} \mathbf{E}[Y_{d+1}] &= \sum_{y,z} y f(y, z) \mathbf{P}_{Y_d, Z_d}(y, z) \\ &= \sum_y \sum_z y f(y, z) \mathbf{P}_{Y_d}(y) \mathbf{P}_{Z_d | Y_d}(z | y) \\ &= \sum_y y \mathbf{P}_{Y_d}(y) \sum_z f(y, z) \mathbf{P}_{Z_d | Y_d}(z | y) \\ &\leq \sum_y y \mathbf{P}_{Y_d}(y) \mathbf{E}[X(y)] \\ &\leq \frac{3}{4} \sum_y y \mathbf{P}_{Y_d}(y) \\ &\leq \frac{3}{4} \mathbf{E}[Y_d]. \end{aligned}$$

Note that we have used the bound

$$\mathbf{E}[X(y)] = \sum_z f(y, z) \mathbf{P}_{Z_d | Y_d}(z | y) \leq \frac{3}{4},$$

which we established above.

We thus conclude that $\mathbf{E}[Y_{d+1}] \leq \frac{3}{4} \mathbf{E}[Y_d]$, which this trivially solves to the bound given in the theorem, since at $d = 0$ the input size is n . \square

Remark. Note that the proof of this theorem would have been relatively easy if the successive choices made by the algorithm were independent but they are not, because the size to the algorithm at each recursive call depends on prior choices of pivots.

4.1.1 Work Analysis

We now have all the ingredients to complete the analysis.

The work at each level of the recursive calls is linear in the size of the input and thus can be written as $W_{\text{select}}(n) \leq k_1 n + k_2$, where n is the input size. Because at least one element, the pivot, is taken out of the input for the recursive call at each level, there are at most n levels of recursion. Thus, by using the theorem below, we can bound the expected work as

$$\begin{aligned}\mathbf{E}[W_{\text{select}}(n)] &\leq \sum_{i=0}^n (k_1 \mathbf{E}[Y_i] + k_2) \\ \mathbf{E}[W_{\text{select}}(n)] &\leq \sum_{i=0}^n (k_1 n \left(\frac{3}{4}\right)^i + k_2) \\ &\leq k_1 n \left(\sum_{i=0}^n \left(\frac{3}{4}\right)^i\right) + k_2 n \\ &\leq 4k_1 n + k_2 n \\ &\in O(n).\end{aligned}$$

Note. As we shall see in subsequent chapters, many contraction algorithms have the same property that the problem instances go down by an expected constant factor at each contraction step.

4.1.2 Span Analysis

Because the span at each level is $O(\lg n)$ and because the depth is at most n , we can bound the span of the algorithm by $O(n \lg n)$ in the worst case. But we expect the average span to be a lot better because chances of picking a poor pivot over and over again, which would be required for the linear span is unlikely. To bound the span in the expected case, we shall use [bound on the expected input size](#) established above, and bound the number of levels in `select` with high probability.

A High-Probability Bound for Span. Consider depth $d = 10 \lg n$. At this depth, the expected instance size upper bounded by

$$n \left(\frac{3}{4}\right)^{10 \lg n}.$$

With a little math this is equal to $n \times n^{-10 \lg(4/3)} \approx n^{-3.15}$.

By [Markov's inequality](#), if the expected size is at most $n^{-3.15}$ then the probability of that the size is at least 1 is bounded by

$$\mathbf{P}[Y_{10 \lg n} \geq 1] \leq \frac{E[Y_{10 \lg n}]}{1} = \frac{1}{n^{3.15}} \leq \frac{1}{n^3}.$$

In applying Markov's inequality, we choose 1, because we know that the algorithm terminates for that input size. We have therefore shown that the number of steps is $O(\lg n)$ with

high probability, i.e., with probability $1 - \frac{1}{n^3}$. Each step has span $O(\lg n)$ so the overall span is $O(\lg^2 n)$ with high probability.

Note that by increasing the constant factor from 10 to 20, we could decrease the probability to $n^{-7.15}$, which is extremely unlikely: for $n = 10^6$ this is 10^{-42} .

An Expected Bound for Span. Using the high probability bound, we can bound the expected span by using the [Total Expectations Theorem](#).

For brevity let the random variable Y be defined as $Y = Y_{10 \lg n}$,

$$\begin{aligned}\mathbf{E}[S] &= \sum_y \mathbf{P}_Y(y) \mathbf{E}[S \mid Y = y]. \\ &= \sum_{y \leq 1} \mathbf{P}_Y(y) \mathbf{E}[S \mid Y = y] + \sum_{y > 1} \mathbf{P}_Y(y) \mathbf{E}[S \mid Y = y] \\ &\leq 1 \cdot O(\lg^2 n) + \frac{1}{n^3} O(n) \\ &= O(\lg^2 n).\end{aligned}$$

The expectation calculation above has two terms.

- The first term considers the case of $y < 1$. In this case, we know that the span is $\Theta(\lg^2 n)$, because the span of each recursive call is $\Theta(\lg n)$, as dominated by the span of the *filter* operation. The probability that $y < 1$ is at most 1.
- The second part considers the case of $y \geq 1$. We know that this case happens with probability at most $\frac{1}{n^3}$ and thus can afford to use a loose bound on the expected span, e.g., total expected work, which is $O(n)$. As a result, this part contributes only a constant to the total.

Exercise 32.1. Prove that the pivot tree has $O(\lg n)$ height, and is therefore balanced, with high probability.

Alternative Span Analysis. Let the random variable $X(n) = \max\{|\ell|, |r|\}$, which is the size of larger subsequence. The span of *select* is upper bounded by the sizes of these larger subsequences. Because we use *filter* to compute the subsequences we have the following recurrence for span for input size n

$$S(n) = S(X(n)) + O(\lg n).$$

For the analysis, we shall condition the span on the random variable denoting the size of the maximum half and apply Theorem 30.1.

$$\mathbf{E}[S(n)] = \sum_{m=n/2}^n \mathbf{P}[X(n) = m] \cdot \mathbf{E}[S(n) \mid (X(n) = m)].$$

The rest is algebra

$$\begin{aligned}
 \mathbf{E}[S(n)] &= \sum_{x=n/2}^n \mathbf{P}[X(n) = x] \cdot \mathbf{E}[S(n) \mid (X(n) = x)] \\
 &\leq \mathbf{P}\left[X(n) \leq \frac{3n}{4}\right] \cdot \mathbf{E}\left[S\left(\frac{3n}{4}\right)\right] + \mathbf{P}\left[X(n) > \frac{3n}{4}\right] \cdot \mathbf{E}[S(n)] + c \cdot \lg n \\
 &\leq \frac{1}{2}\mathbf{E}\left[S\left(\frac{3n}{4}\right)\right] + \frac{1}{2}\mathbf{E}[S(n)] + c \cdot \lg n \\
 \implies \mathbf{E}[S(n)] &\leq \mathbf{E}\left[S\left(\frac{3n}{4}\right)\right] + 2c \lg n.
 \end{aligned}$$

This is a recursion in $\mathbf{E}[S(\cdot)]$ and solves easily to $\mathbf{E}[S(n)] = O(\lg^2 n)$.

Chapter 33

The Quick Sort Algorithm

This chapter presents an analysis of the randomized Quick sort algorithm. [The first section](#) presents the quicksort algorithm and its variants. [The second section](#) analyzes the randomized variant.

1 Quicksort

Algorithm 33.1 (Generic Quicksort). The code below show the quicksort algorithm, where the pivot-choosing step is intentionally left under-specified.

```
1  quicksort a =
2      if |a| = 0 then a
3      else
4          let
5              p = pick a pivot from a
6              a1 = {x ∈ a | x < p}
7              a2 = {x ∈ a | x = p}
8              a3 = {x ∈ a | x > p}
9              (s1, s3) = (quicksort a1) || (quicksort a3)
10             in
11                 s1++a2++s3
12             end
```

Quicksort exposes plenty of parallelism:

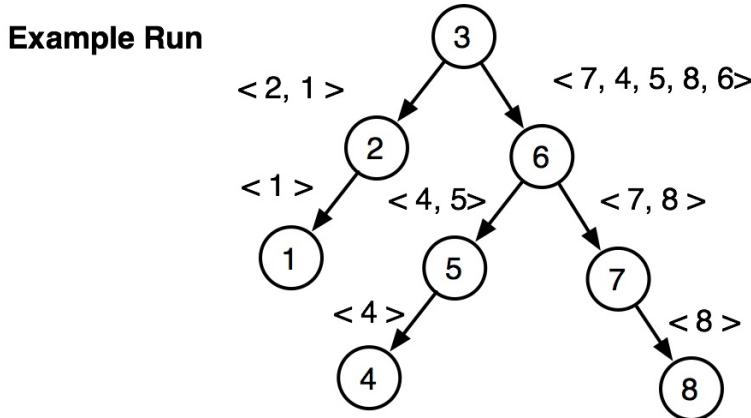
- the two recursive recursive calls are parallel, and

- the filters for selecting elements greater, equal, and less than the pivot are also internally parallel.

Pivot Tree. Each call to *quicksort* either makes no recursive calls (the base case) or two recursive calls. We can therefore represent an execution of *quicksort* as a binary search tree, where the nodes—tagged with the pivots chosen—represent the recursive calls and the edges represent the caller-callee relationships. We refer to such a tree as a *pivot tree* and use pivot trees to reason about the properties of *quicksort* such as its work and span.

Example 33.1 (Pivot Tree). An example run of *quicksort* along with its pivot tree. In this run, the first call chooses 3 as the pivot.

Keys $\langle 7, 4, 2, 3, 5, 8, 1, 6 \rangle$



Pivot Tree, Work, and Span. Given a pivot tree for a run of *quicksort*, we can bound the work and the span of that run by inspecting the pivot tree. Note that for an sequence of length n , the work at each call to *quicksort* is $\Theta(n)$ excluding the calls to the recursive calls, because the *filter* calls each take $\Theta(n)$ work. Similarly, the span is $\Theta(\lg n)$, because *filter* requires $\Theta(\lg n)$ span. Thus we can bound work by adding up the instance size of each node in the pivot tree; the instance size is exactly the number of nodes in that subtree. Likewise for span, we can compute the sum of the logarithm of the instance sizes along each path and take their maximum. Or more simply we can bound the span by multiplying the length of the longest path by $\Theta(\lg n)$.

Impact of Pivot Selection on Work and Span. Because the chosen pivots determine exactly how balanced the pivot tree is, they can have a significant impact on the work and span of the *quicksort* algorithm. Let's consider several alternatives and their impact on work and span.

- **Always pick the first element as pivot:** In this case, inputs that are sorted or nearly sorted can lead to high work and span. For example, if the input sequence is sorted in increasing order, then the smallest element in the input is chosen as the pivot. This leads to an unbalanced, lopsided pivot tree of depth n . The total work is $\Theta(n^2)$, because $n-i$ keys will remain at level i and the total work is $\sum_{i=0}^{n-1} (n-i-1)$. Similarly, if the sequence is sorted in decreasing order, we will end up with a pivot tree that is lopsided in the other direction. In practice, it is not uncommon for inputs to *quicksort*, or any sort, to be sorted or nearly sorted.
- **Pick the median of three elements as the pivot:** Another strategy is to take the first, middle, and the last elements and pick the median of them. For sorted inputs, this strategy leads to an even partition of the input, and the depth of the tree is $\Theta(\lg n)$. In the worst case, however, this strategy is no better than the first strategy. Nevertheless, this is the strategy used broadly in practice.
- **Pick a random element as the pivot:** In this strategy, the algorithm selects one of the elements, uniformly randomly from its input as the pivot. It is not immediately clear what the work and span of this strategy is, but intuitively, when we choose a random pivot, the size of each side is not far from $n/2$ in expectation. This doesn't give us a proof but it gives us some reason to expect that this strategy could result in a tree of depth $\Theta(\lg n)$ in expectation or with high probability. Indeed, in [the analysis section](#), we prove that selecting a random pivot gives us expected $\Theta(n \lg n)$ work and a recursion tree of depth $\Theta(\lg n)$ leading to $\Theta(\lg^2 n)$ span since each recursive call has $\Theta(\lg n)$ span.

Exercise 33.1. Describe the worst-case input for the version of *quicksort* that uses the second strategy above.

2 Analysis of Quicksort

This section presents an analysis of the randomized quicksort algorithm which selects its pivots randomly.

Assumptions. We assume that the algorithm is [as shown previously](#) and that the pivot is chosen as a uniformly random element of the input sequence.

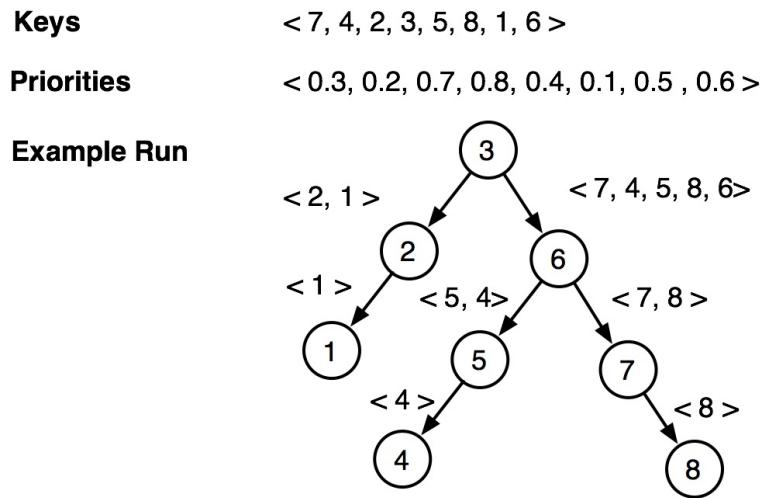
To streamline the analysis, we make the following two assumptions.

- We “simulate” randomness with priorities: before the start of the algorithm, we assign each key a priority uniformly at random from the real interval $[0, 1]$ such that each key has a unique priority. The algorithm then picks in Line 5 the key with the highest priority.
- We assume a version of *quicksort* that compares the pivot p to each key in the input sequence once (instead of 3 times).

Notice that once the priorities are decided at the beginning, the algorithm is completely deterministic.

Exercise 33.2. Rewrite the quicksort algorithm to use the comparison once when comparing the pivot with each key at a recursive call.

Example 33.2 (Randomness and Priorities). An execution of *quicksort* with priorities and its pivot tree, which is a binary-search-tree, illustrated.



Exercise 33.3. Convince yourself that the two presentations of randomized *quicksort* are fully equivalent (modulo the technical details about how we might store the priority values).

Roadmap. The rest of this section is organized as follows. First, we present [an intuitive analysis](#). Then, we present [a mathematical analysis](#). And finally, we outline [an alternative analysis](#), which uses recurrence relations.

2.1 Intuition

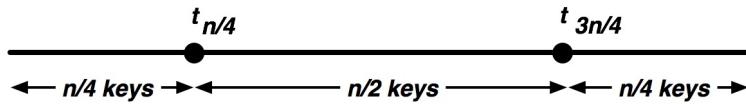
We can reason about the work and span of quicksort in a way similar to the randomized *select* algorithm that we considered in [the previous chapter](#).

Observations. Recall that rank of an element in a sequence is the position of the element in the sorted sequence. Now consider the rank of the pivot selected at a step. If the rank of the pivot is greater $n/4$ and less than $3n/4$ then we say that we are *lucky*, because the instance size at the next recursive call is at most $3n/4$. Because all elements are equally

likely to be selected as a pivot, the probability of being lucky is

$$\frac{3n/4 - n/4}{n} = 1/2.$$

The figure below illustrates this.



Multiple Tries for Better Luck. By the above observation, we know that, if we are lucky then the instance size decreases by a constant fraction (3/4 or less).

But what if we are unlucky? Consider two successive recursive calls, the probability that the input size decreases by 3/4 after two calls is the probability that it decreases at either call, which is at least $1 - \frac{1}{2} \cdot \frac{1}{2} = \frac{3}{4}$. More generally, after $c > 1$ such successive calls, the probability that the input size decreases by a factor of $\frac{3}{4}$ is at least $1 - \frac{1}{2^c}$, which quickly approaches 1 as c increases. For example if $c = 10$ then this probability is 0.999.

In other words, chances of getting unlucky at any given step might be reasonably high (1/2) but chances of getting unlucky over and over again is low, and we only need to get lucky once.

This means that with quite high probability, a constant number c of recursive calls is almost guaranteed to decrease the input size by a constant fraction. Thus we expect *quicksort* to complete after $k \lg n$ levels for some constant k .

Therefore, intuitively, the total work is $\Theta(n \lg n)$ and the span is $\Theta(\lg^2 n)$ in expectation. But, this is not a proof.

2.2 The Analysis

Observations. Before we get to the analysis, let's observe some properties of *quicksort*. For these observations, it might be helpful to consider the example shown above.

- In *quicksort*, a comparison always involves a pivot and another key.
- Because the pivot is not part of the instance solved recursively, a key can become a pivot at most once.
- Each key eventually becomes a pivot, or is a singleton base case.

Based on the first two observations, we conclude that each pair of keys is compared at most once.

2.2.1 Expected Work for Quicksort

Work and Number of Comparisons. We are now ready to analyze the expected work of randomized *quicksort*. Instead of bounding the work directly, we will use a surrogate metric—the number of comparisons—because it is easier to reason about it mathematically. Observe that the total work of *quicksort* is bounded by the number of comparisons because the work required to partition the input at each recursive calls is asymptotically bounded by the number of comparisons between the keys and the pivot. To bound the work, asymptotically, it therefore suffices to bound the total number of comparisons.

Random Variables. We define the random variable $Y(n)$ as the number of comparisons *quicksort* makes on input of size n . For the analysis, we will find an upper bound on $\mathbf{E}[Y(n)]$. In particular we will show a bound of $O(n \lg n)$ irrespective of the the order keys in the input sequence.

In addition to $Y(n)$, we define another random variable X_{ij} that indicates whether keys with rank i and j are compared. More precisely, consider the final sort of the keys $t = \text{sort}(a)$ and for any element element t_i . Consider two positions $i, j \in \{0, \dots, n-1\}$ in the sequence t and define following random variable

$$X_{ij} = \begin{cases} 1 & \text{if } t_i \text{ and } t_j \text{ are compared by } \text{quicksort} \\ 0 & \text{otherwise.} \end{cases}$$

Total Number of Comparisons. By the [observations stated above](#), we know that in any run of *quicksort*, each pair of keys is compared at most once. Thus $Y(n)$ is equal to the sum of all X_{ij} 's, i.e.,

$$Y(n) \leq \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} X_{ij}$$

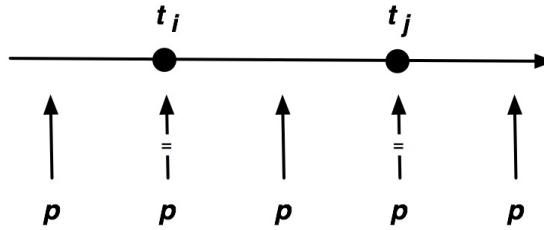
In the sum above, we only consider the case of $i < j$, because we only want to count each comparison once.

By linearity of expectation, we have

$$\mathbf{E}[Y(n)] \leq \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \mathbf{E}[X_{ij}]$$

Since each X_{ij} is an indicator random variable, $\mathbf{E}[X_{ij}] = \mathbf{P}[X_{ij} = 1]$.

Calculating $\mathbf{P}[X_{ij}]$. To compute the probability that t_i and t_j are compared (i.e., $\mathbf{P}[X_{ij} = 1]$), let's take a closer look at the *quicksort* algorithm and consider the first pivot p selected by the algorithm. The drawing below illustrates the possible relationships between the pivot p and the elements t_i and t_j .



Notice first that the pivot p is the element with highest priority. For X_{ij} , where $i < j$, we distinguish between three possible scenarios as illustrated in the drawing above:

- $p = t_i$ or $p = t_j$; in this case t_i and t_j are compared and $X_{ij} = 1$.
- $t_i < p < t_j$; in this case t_i is in a_1 and t_j is in a_3 and t_i and t_j will never be compared and $X_{ij} = 0$.
- $p < t_i$ or $p > t_j$; in this case t_i and t_j are either both in a_1 or both in a_3 , respectively. Whether t_i and t_j are compared will be determined in some later recursive call to *quicksort*.

Note now that the argument above applies to any recursive call of *quicksort*.

Lemma 33.1 (Comparisons and Priorities). For $i < j$, let t_i and t_j be the keys with rank i and j , and p_i or p_j be their priorities. The keys t_i and t_j are compared if and only if either p_i or p_j has the highest priority among the priorities of keys with ranks $i \dots j$.

Proof. For the proof, let p_i be the priority of the key t_i with rank i .

Assume first that t_i (t_j) has the highest priority. In this case, all the elements in the subsequence $t_i \dots t_j$ will move together in the pivot tree until t_i (t_j) is selected as pivot. When it is selected as pivot, t_i and t_j will be compared. This proves the first half of the claim.

For the second half, assume that t_i and t_j are compared. For the purposes of contradiction, assume that there is a key t_k , $i < k < j$ with a higher priority between them. In any collection of keys that include t_i and t_j , t_k will become a pivot before either of them. Since $t_i \leq t_k \leq t_j$ it will separate t_i and t_j into different buckets, so they are never compared. This is a contradiction; thus we conclude there is no such t_k .

□

Bounding $E[Y(n)]$. Therefore, for t_i and t_j to be compared, p_i or p_j has to be bigger than all the priorities in between. Since there are $j - i + 1$ possible keys in between (including both i and j) and each has equal probability of being the highest, the probability that either

i or j is the highest is $2/(j - i + 1)$. Therefore,

$$\begin{aligned}\mathbf{E}[X_{ij}] &= \mathbf{P}[X_{ij} = 1] \\ &= \mathbf{P}[p_i \text{ or } p_j \text{ is the maximum among } \{p_i, \dots, p_j\}] \\ &= \frac{2}{j - i + 1}.\end{aligned}$$

We can write the expected number of comparisons made in randomized *quicksort* is

$$\begin{aligned}\mathbf{E}[Y(n)] &\leq \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \mathbf{E}[X_{ij}] \\ &= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \frac{2}{j - i + 1} \\ &= \sum_{i=0}^{n-1} \sum_{k=2}^{n-i} \frac{2}{k} \\ &\leq 2 \sum_{i=0}^{n-1} H_n \\ &= 2nH_n \in O(n \lg n).\end{aligned}$$

Note that in the derivation of the asymptotic bound, we used the fact that $H_n = \ln n + O(1)$. Recall that $H_n = \sum_{k=1}^n \frac{1}{k}$ is the “harmonic number” for n .

Note. Indirectly by Lemma 33.1, we have also shown that the average work for the “basic” deterministic *quicksort*, which always pick the first element as pivot, is also $\Theta(n \lg n)$. Just shuffle the input sequence randomly and then apply the basic *quicksort* algorithm. Since shuffling the input randomly results in the same input as picking random priorities and then reordering the data so that the priorities are in decreasing order, the basic *quicksort* on that shuffled input does the same operations as randomized *quicksort* on the input in the original order. Thus, if we averaged over all permutations of the input the work for the basic *quicksort* is $O(n \lg n)$ on average.

Remark (Ranks and Comparisons). The bound

$$\mathbf{E}[X_{ij}] = \frac{2}{j - i + 1}$$

indicates that the closer two keys are in the sorted order (t) the more likely it is that they are compared. It is instructive to interpret this in the context of pivot tree.

For example, the keys t_i is compared to t_{i+1} with probability 1. Indeed, one of t_i and t_{i+1} must be an ancestor of the other in the pivot tree, because there is no other element that could be the root of a subtree that has t_i in its left subtree and t_{i+1} in its right subtree. Regardless, t_i and t_{i+1} will be compared.

If we consider t_i and t_{i+2} there could be such an element, namely t_{i+1} , which could have t_i in its left subtree and t_{i+2} in its right subtree. That is, with probability $1/3$, t_{i+1} has the

highest probability of the three and t_i is not compared to t_{i+2} , and with probability $2/3$ one of t_i and t_{i+2} has the highest probability and, the two are compared.

In general, the probability of two elements being compared is inversely proportional to the number of elements between them when sorted. The further apart the less likely they will be compared. Analogously, the further apart the less likely one will be the ancestor of the other in the related pivot tree.

2.2.2 Expected Analysis of Quicksort: Span

We now analyze the span of *quicksort*. All we really need to calculate is the depth of the pivot tree, since each level of the tree has span $O(\lg n)$ —needed for the filter. We argue that the depth of the pivot tree is $O(\lg n)$ by relating it to the number of contraction steps of the [randomized order statistics algorithm](#) *select* that we considered earlier. For the discussion, we define the i^{th} node of the pivot tree as the node corresponding to the i^{th} smallest key. This is also the i^{th} node in an in-order traversal.

Lemma 33.2 (Quicksort and Order Statistics). The path from the root to the i^{th} node of the pivot tree is the same as the steps of *select* on $k = i$. That is to say that the distribution of pivots selected along the path and the sizes of each problem is identical.

Proof. Note that that *select* is the same as *quicksort* except that it only goes down one of the two recursive branches—the branch that contains the k^{th} key.

Recall that for *select*, we showed that the length of the path is more than $10 \lg n$ with probability at most $1/n^{3.15}$. This means that the length of any path being longer than $10 \lg n$ is tiny. \square

This does not suffice to conclude, however, that there are no paths longer than $10 \lg n$, because there are many paths in the pivot tree, and because we only need one to be long to impact the span. Luckily, we don't have too many paths to begin with. We can take advantage of this property by using the [union bound](#), which says that the probability of the union of a collection of events is at most the sum of the probabilities of the events.

To apply the union bound, consider the event that the depth of a node along a path is larger than $10 \lg n$, which is $1/n^{3.5}$. The total probability that any of the n leaves have depth larger than $10 \lg n$ is

$$\mathbf{P} [\text{depth of } \textit{quicksort} \text{ pivot tree} > 10 \lg n] \leq \frac{n}{n^{3.15}} = \frac{1}{n^{2.15}}.$$

We thus have our high probability bound on the depth of the pivot tree.

The overall span of randomized *quicksort* is therefore $O(\lg^2 n)$ with high probability. As in *select*, we can establish an expected bound by using [Theorem 30.1](#).

Exercise 33.4. Complete the span analysis of proof by showing how to apply the Total Expectation Theorem.

2.3 Alternative Analysis of Quicksort

Another way to analyze the work of *quicksort* is to write a recurrence for the expected work (number of comparisons) directly. This is the approach taken by Tony Hoare in his original paper. For simplicity we assume there are no equal keys (equal keys just reduce the cost). The recurrence for the number of comparisons $Y(n)$ done by *quicksort* is then:

$$Y(n) = Y(X(n)) + Y(n - X(n) - 1) + n - 1$$

where the random variable $X(n)$ is the size of the set a_1 (we use $X(n)$ instead of X_n to avoid double subscripts). We can now write an equation for the expectation of $Y(n)$.

$$\begin{aligned} \mathbf{E}[Y(n)] &= \mathbf{E}[Y(X(n)) + Y(n - X(n) - 1) + n - 1] \\ &= \mathbf{E}[Y(X(n))] + \mathbf{E}[Y(n - X(n) - 1)] + n - 1 \\ &= \frac{1}{n} \sum_{i=0}^{n-1} (\mathbf{E}[Y(i)] + \mathbf{E}[Y(n - i - 1)]) + n - 1 \end{aligned}$$

where the last equality arises since all positions of the pivot are equally likely, so we can just take the average over them. This can be by guessing the answer and using substitution. It gives the same result as our previous method. We leave this as exercise.

Span Analysis. We can use a similar strategy to analyze span. Recall that in randomized *quicksort*, at each recursive call, we partition the input sequence a of length n into three subsequences a_1 , a_2 , and a_3 , such that elements in the subsequences are less than, equal, and greater than the pivot, respectively. Let the random variable $X(n) = \max\{|a_1|, |a_3|\}$, which is the size of larger subsequence.

The span of *quicksort* is determined by the sizes of these larger subsequences. For ease of analysis, we will assume that $|a_2| = 0$, as more equal elements will only decrease the span. As this partitioning uses *filter* we have the following recurrence for span for input size n

$$S(n) = S(X(n)) + O(\lg n).$$

For the analysis, we shall condition the span on the random variable denoting the size of the maximum half and apply Theorem 30.1.

$$\mathbf{E}[S(n)] = \sum_{m=n/2}^n \mathbf{P}[X(n) = m] \cdot \mathbf{E}[S(n) \mid (X(n) = m)].$$

The rest is algebra

$$\begin{aligned}
 \mathbf{E}[S(n)] &= \sum_{x=n/2}^n \mathbf{P}[X(n) = x] \cdot \mathbf{E}[S(n) \mid (X(n) = x)] \\
 &\leq \mathbf{P}\left[X(n) \leq \frac{3n}{4}\right] \cdot \mathbf{E}\left[S\left(\frac{3n}{4}\right)\right] + \mathbf{P}\left[X(n) > \frac{3n}{4}\right] \cdot \mathbf{E}[S(n)] + c \cdot \lg n \\
 &\leq \frac{1}{2}\mathbf{E}\left[S\left(\frac{3n}{4}\right)\right] + \frac{1}{2}\mathbf{E}[S(n)] + c \cdot \lg n \\
 \implies \mathbf{E}[S(n)] &\leq \mathbf{E}\left[S\left(\frac{3n}{4}\right)\right] + 2c \lg n.
 \end{aligned}$$

This is a recursion in $\mathbf{E}[S(\cdot)]$ and solves easily to $\mathbf{E}[S(n)] = O(\lg^2 n)$.

3 Concluding Remarks

History of quick sort. Quick sort is one of the earliest and most famous algorithms. It was invented and analyzed by Tony Hoare around 1960. This was before the big-O notation was used to analyze algorithms. Hoare invented the algorithm while an exchange student at Moscow State University while studying probability under Kolmogorov—one of founders of the field of probability theory.

Our presentation of the [quick sort algorithm](#) differs from Hoare's original, which partitions the input sequentially by using two fingers that move from each end and by swapping two keys whenever a key was found on the left greater than the pivot and on the right less than the pivot. The analysis covered in this chapter is different from Hoare's original analysis.

It is interesting that while Quick sort is a quintessential example of a recursive algorithm, at the time of its invention, no programming language supported recursion and Hoare went into some depth explaining how recursion can be simulated with a stack.

Remark. In Chapter 36, we will see that the analysis of quick sort presented here is effectively identical to the analysis of a certain type of balanced tree called Treaps. It is also the same as the analysis of “unbalanced” binary search trees under random insertion.

Part VII

Binary Search Trees

Chapter 34

Introduction

This chapter presents the motivation behind Binary Search Trees (*BSTs*) and an Abstract Data Type (ADT) for them. It also briefly discusses the well-known implementation techniques for BSTs.

1 Motivation

Searching Dynamic Collections. Searching efficiently is one of the most important goals in the design of data structures. Of the many search data structures that have been designed and are used in practice, search trees, more specifically balanced binary search trees (BSTs), occupy a coveted place because of their broad applicability to many different problems. For example, in this book, we rely on binary search trees to implement `set` and `table` (dictionary) abstract data types. Sets and tables are instrumental in many algorithms, including for example graph algorithms, many of which we cover later in the book.

A Total Order. BSTs require that the keys being stored come from a total order so we can store the keys in “sorted order”. This makes BSTs particularly useful when our searches are based on the order, such as finding all keys between two values (a range search), or given a key, finding the next larger key. They are also useful if the only access we have to keys is the ability to compare them. In these cases other search structures such as hash tables (discussed in a later chapter) are of little utility. Even when we do not need ordering, BSTs have some other benefits over hash tables. An important one is that it is easy to make them “persistent” so that an update leaves the old tree unmodified while creating a new tree.

Dynamic versus Static. If we are interested in searching a static or unchanging collec-

tion of elements, then BSTs are not necessary and we can use sequences instead. More specifically, we can represent a collection as a sorted sequence and use binary search to implement searches. Using array sequences such a binary search requires logarithmic work. If we wish to support dynamic collections, which allow, for example, inserting and deleting keys, then a sequence based implementation would require linear work for these updates since all, or many, keys might need to be moved. BSTs support various updates, including insertions and deletions, in logarithmic work.

Sequential versus Parallel Aggregate Operations. In the traditional treatment of algorithms, which focuses on sequential algorithms, binary search trees revolve around three operations: insertion, deletion, and search. While these operations are important, they are not sufficient for parallelism, since they perform a single “update” at a time. We therefore consider aggregate operations, such as union, intersection, set-difference, filter, map, and reduce. All of these can be implemented in parallel. Then instead of inserting one element at a time, for example, we can use a union to insert a whole set of values.

Roadmap. We first [define binary search trees](#). We then present [an ADT for binary search trees](#), and describe a [parametric implementation](#) of the ADT. The parametric implementation uses only one non-trivial operation, *joinMid*, which joins together two trees with a key in the middle. As a result, we are able to reduce the problem of implementing the BST ADT to the problem of implementing just the function *joinMid*. We then present a specific instance of the parametric implementation using [Treaps](#).

2 Preliminaries

We start with some basic definitions and terminology involving rooted and binary search trees. Recall first that a [rooted tree](#) is a tree with a distinguished root node.

Definition 34.1 (Full Binary Tree). A *full binary tree* is an ordered [rooted tree](#) where each node is either a *leaf*, which has no children, or an *internal node*, which has exactly two children: a *left child* and a *right child*. This can be defined as the recursive type:

$$\begin{aligned} \text{type } \text{tree} &= \text{Leaf of } \alpha \\ &\mid \text{Node of } \text{tree} \times \beta \times \text{tree} \end{aligned}$$

where α is the type of value stored at the leaves, and β is the type stored at the internal nodes.

For a given node in a binary tree, we define the *left subtree* of the node as the subtree rooted at the left child, and the *right subtree* of the node as the subtree rooted at the right child.

In this book we only consider storing values on the internal nodes and assume leaves have no values associated with them. We will therefore assume from now on that all values are

stored at nodes. It is useful to define different traversal orders on binary trees. A traversal starts at the root and inductively traverses each subtree. The order that is most important to us is the *in-order* traversal. It can be defined as follows:

Definition 34.2. The *in-order* traversal of a full binary tree is given by the order of elements in the sequence returned by:

```
inOrder T =
  case T of
    Leaf ⇒ ⟨ ⟩
  | Node(L, k, R) ⇒ inOrder(L) ++ ⟨ k ⟩ ++ inOrder(R)
```

i.e., it recursively puts the values from the left subtree first, then the key k at the node, and then the values from the right subtree. We assume leaves have no values.

Another common order is the *pre-order*, which visits keys in the order given by:

```
preOrder T =
  case T of
    Leaf ⇒ ⟨ ⟩
  | Node(L, k, R) ⇒ ⟨ k ⟩ ++ preOrder(L) ++ preOrder(R)
```

i.e., first the key, then the left subtree, and finally the right subtree.

When the values (keys) we store at each node have a total ordering defined by a comparison $<$, we will use the following notation: For complete binary trees T, T_1 and T_2 , and a value k , we use:

$$\begin{aligned} T < k &\equiv \text{for all } k' \in T, k' < k \\ k < T &\equiv \text{for all } k' \in T, k < k' \\ T_1 < T_2 &\equiv \text{for all } k_1 \in T_1, k_2 \in T_2, k_1 < k_2 \end{aligned}$$

We are now ready to define binary search trees.

Definition 34.3 (Binary Search Tree (BST)). Consider a set S taken from a total order defined by the comparison $<$. A *binary search tree* (BST) over S is a full binary tree T (with no leaf values) that satisfies the following conditions.

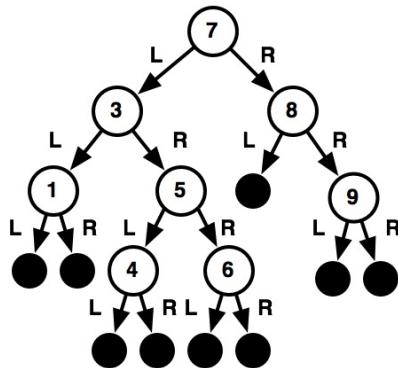
1. There is a one-to-one mapping $k(v)$ from internal tree nodes of T to elements in S , and
2. $\text{inOrder}(T)$ is sorted by $<$.

In the definition, the second condition is referred to as the *BST property*. It can be equivalently stated as: for all internal nodes $(L, k, R) \in T, L < k < R$ (using our notation above).

We often refer to the elements of S in a BST as keys, and use $\text{dom}(T)$ to indicate the domain (keys) in a BST T .

We define the *size* of a BST S as the number of keys in the tree, and also write it as $|S|$. We define the *depth* of either a leaf or node in a BST, as the length of the path from the root to that leaf or node. The root always has depth 0. We define the *height* of a BST, denoted as $h(T)$, as the maximum depth of any leaf. An empty tree has height 0, and a tree with a single node has height 1.

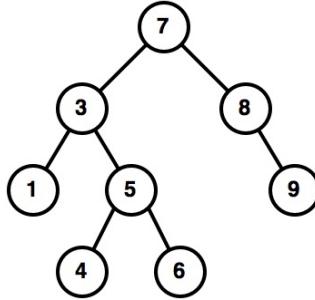
Example 34.1. An example binary search tree over the set of natural numbers $\{4, 1, 7, 9, 83, 6, 5\}$, and defined by the standard total ordering over the natural numbers, is given by the following tree T :



On the left the L and R indicate the left (first) and right (second) child, respectively. All internal nodes (white) have a key associated with them while the leaves (black) are empty. The keys satisfy the BST property—the in-order ordering, given by $\text{inOrder}(T)$ is $\langle 1, 3, 4, 5, 6, 7, 8, 9 \rangle$, is sorted. Or, equivalently, for every node, the keys in the left subtree are less, than the key at the root, and the ones in the right subtree are greater.

The size of the tree is 8, because there are 8 keys in the tree. The height of the tree is 4, e.g., the path from root 7 to a child of node 4.

In the illustration of the tree, the edges are oriented away from the root, to indicate the direction we can search from starting at the root. When illustrating binary search trees, we usually replace the directed arcs with undirected edges, leaving the orientation to be implicit. We also drop the leaves since they contain no information, and implicitly assume the left branch (L) is on the left and the right branch (R) is on the right. Given these conventions, we get the following simpler illustration of a BST, which we will be using in the rest of the book.



Here we are assuming that we just store keys at the nodes of the trees, and that the keys represent an ordered set. In practice we often store a value associated with each key at each node, and perhaps other information. These do not change the fundamental ideas we discuss here, so we will ignore them at first and then come back to them.

3 Searching a BST

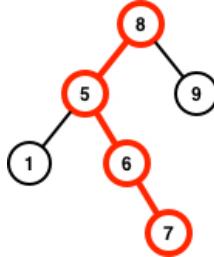
As mentioned, the primary goal of a BST is to do fast searches for a particular key and whether there is a copy in the tree. Fortunately, it is relatively easy to search for a key in a BST, and even to find the next larger or smaller key in the tree. To find a particular key we can start at the root r and if k equals the key at the root, call it k' , then we have found our key, otherwise if $k < k'$, then we know that k cannot appear in the right subtree, so we only need to search the left subtree, and if $k > k'$, then we only have to search the right subtree. Continuing the search, we will either find the key or reach a leaf and conclude that the key is not in the tree. Based on this idea the following algorithm will return whether a key k is in a BST T .

Algorithm 34.4 (Searching a BST).

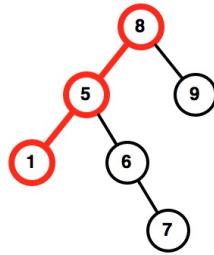
```

find  $T$   $k$  =
  case  $T$  of
    Leaf  $\Rightarrow$  false
    | Node( $L, k', R$ )  $\Rightarrow$ 
      if ( $k = k'$ ) then true
      else if ( $k < k'$ ) then find  $L$   $k$ 
      else find  $R$   $k$ 
  
```

Example 34.2. A successful search (*find*) for 7. The search path is highlighted.



An unsuccessful search (find) for 4.



It is important to note that the search only visits a path from the root to some node (either a leaf or internal node). Therefore the search will visit at most as many nodes as the height of the tree, $h(T)$. Assuming the comparison $<$ takes constant work, this means the work for $find$ is $O(h(T))$.

4 Balancing BSTs

Given that the work to find a key (and also for other operations) is proportional to the height of a BST, our goal should therefore be to keep the height low. In the worst case the height could be equal to the size. This would be true if every node had one child that is a leaf. Such a tree is clearly very unbalanced. Our goal therefore implies we should keep the tree “balanced” such that all leaves of the tree are at approximately at the same depth. We formalize this notion as follows.

Definition 34.5 (Perfectly Balanced BSTs). A binary tree is *perfectly balanced* if it has the minimum possible height. For a binary search tree with n keys, a perfectly balanced tree has height exactly $\lceil \lg(n + 1) \rceil$.

Ideally we would like to use only perfectly balanced trees. If we never make changes to the tree, we could balance it once and for all. If, however, we want to update the tree by, for example, inserting new keys, then maintaining such perfect balance is costly. In fact, it turns out to be impossible to maintain a perfectly balanced tree while allowing insertions

in $O(\lg n)$ work. BST data structures therefore aim to keep approximate balance instead of a perfect one. There are various schemes that keep trees nearly balanced for any sized tree. These schemes maintain invariants at each node that ensure this near balance.

Definition 34.6 (Nearly Balanced BSTs). We refer to a balancing scheme as maintaining *near balance*, or simply *balance*, if all trees with n elements that satisfy the scheme's invariants have height $O(\lg n)$. In some cases this is satisfied in expectation or with high probability.

Balanced BST Data Structures. There are many balancing schemes for BSTs. Most either try to maintain height balance (the children of a node are about the same height) or weight balance (the children of a node are about the same size). Here we list a few such balancing schemes:

1. *AVL trees* are the earliest nearly balanced BST data structure (1962). AVL trees maintain the invariant that the two children of each node differ in height by at most one, which implies near balance.
2. *Red-Black trees* maintain the invariant that all leaves have a depth that is within a factor of 2 of each other. The depth invariant is ensured by a scheme of coloring the nodes red and black.
3. *Weight balanced (BB[α]) trees* maintain the invariant that the left and right subtrees of a node of size n each have size at least αn for $0 < \alpha \leq 1 - \frac{1}{\sqrt{2}}$. The BB stands for bounded balance, and adjusting α gives a tradeoff between search and update costs.
4. *Treaps* associate a random priority with every key and maintain the invariant that the keys are stored in heap order with respect to their priorities (the term “Treap” is short for “tree heap”). Treaps guarantee near balance with high-probability.
5. *Splay trees* are an amortized data structure that does not guarantee near balance, but instead guarantees that for any sequence of m insert, find and delete operations each does $O(\lg n)$ amortized work.

There are several other balancing schemes for BST data structures (e.g. scapegoat trees and AA trees), as well as many that allow larger degrees, including 2–3 trees, brother trees, and B trees.

Remark. Many of the existing BST data structures were developed for sequential computing. Some of these data structures such as Treaps, which we describe here, generalize naturally to parallel computing. But some others, such as data structures that rely on amortization techniques can be challenging to support in the parallel setting.

5 An Interface for Sets

As discussed, BSTs are useful for representing sets of keys that have a total ordering, and require dynamic changes. Let us consider what functions could be useful for such “dy-

namic" sets. Certainly we will require some basic operations such as creating an empty set, creating a singleton set, or returning the size of the set. Also, we would like to find an element in a set, insert an element into a set, and delete an element from a set. You might have studied how to do this with BSTs with particular balancing schemes (e.g. AVL trees, or red-black trees) in previous courses.

However, at a higher level we would like to supply bulk operations on sets, such as taking the union of two sets, filtering a set so that only elements that satisfy a predicate remain, or summing the elements of a set with respect to some associative operation. For tables (i.e., when we associate a value with each key), we might also want to map some function over the values to generate a new table, or filter based on the values. Such bulk operations are very useful in programming with sets and tables. They are also important for taking advantage of parallelism since individual inserts and deletes are inherently sequential, but bulk operations can often be parallelized.

To implement these bulk operations it is useful to build them on top of some primitives. Building them on top of insertion, deletion will not be effective since they are inherently sequential. Instead, as we will see, it is useful to build them on top of three other operations `split`, `joinM`, and `joinPair`. These are described below. As described in the next chapter, `split` and `joinPair` can be implemented in terms of `joinM` so all we will really need is `joinM` (actually a slight variant, called `joinMid`).

Here we present an abstract data type that supplies a collection of useful functions over ordered sets. We will extend this list with further functions in the following chapters. An important aspect of this interface is that it is designed to support parallelism.

Data Type 34.7 (BST). For a universe of totally ordered keys \mathbb{K} , the BST ADT consists of a type \mathbb{T} representing a power set of keys and the functions whose types are specified as:

<i>empty</i>	$: \mathbb{T}$
<i>singleton</i>	$: \mathbb{K} \rightarrow \mathbb{T}$
<i>size</i>	$: \mathbb{T} \rightarrow \mathbb{N}$
<i>find</i>	$: \mathbb{T} \rightarrow \mathbb{K} \rightarrow \mathbb{B}$
<i>delete</i>	$: (\mathbb{T} \times \mathbb{K}) \rightarrow \mathbb{T}$
<i>insert</i>	$: (\mathbb{T} \times \mathbb{K}) \rightarrow \mathbb{T}$
<i>union</i>	$: (\mathbb{T} \times \mathbb{T}) \rightarrow \mathbb{T}$
<i>intersection</i>	$: (\mathbb{T} \times \mathbb{T}) \rightarrow \mathbb{T}$
<i>difference</i>	$: (\mathbb{T} \times \mathbb{T}) \rightarrow \mathbb{T}$
<i>split</i>	$: (\mathbb{T} \times \mathbb{K}) \rightarrow (\mathbb{T} \times \mathbb{B} \times \mathbb{T})$
<i>joinPair</i>	$: (\mathbb{T} \times \mathbb{T}) \rightarrow \mathbb{T}$
<i>joinM</i>	$: (\mathbb{T} \times \mathbb{K} \times \mathbb{T}) \rightarrow \mathbb{T}$
<i>filter</i>	$: (\mathbb{K} \rightarrow \text{bool}) \rightarrow \mathbb{T} \rightarrow \mathbb{T}$
<i>reduce</i>	$: (\mathbb{K} \times \mathbb{K} \rightarrow \mathbb{K}) \rightarrow \mathbb{K} \rightarrow \mathbb{T} \rightarrow \mathbb{K}$

and functionality is defined below.

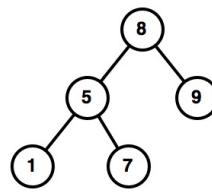
The ADT supports two constructors: *empty* and *singleton*. As their names imply, the func-

tion *empty* creates an empty BST and the function *singleton* creates a BST with a single key.

The function *find* searches for a given key and returns a boolean indicating success.

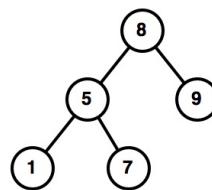
The functions *insert* and *delete* insert and delete a given key into or from the BST.

Example 34.3. Consider the following tree.

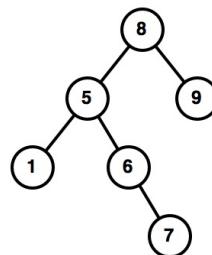


- Searching for 5 in the tree above returns `true`.
- Searching for 6 in the tree above returns `false`.

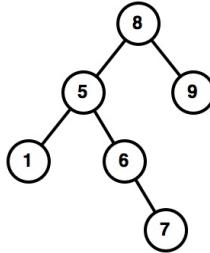
Example 34.4 (Insertion). Consider the following tree.



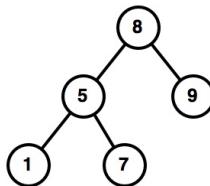
Inserting the key 6 into this tree returns the following tree.



Example 34.5 (Deletion). Consider the following tree.



Deleting the key 6 from this tree returns the following tree.

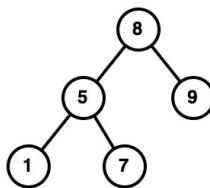


Union, Intersection, and Difference. The function *union* takes two BSTs and returns a BST that contains the union of the keys in the two BSTs. The function *intersection* takes two BSTs and returns a BST that contains the keys that appear in both (i.e., the intersection of the sets). The function *difference* takes two BSTs T_1 and T_2 and returns a BST that contains the keys in T_1 that are not in T_2 .

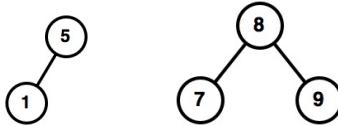
Note that *union* can be thought of as a “parallel” insert since it can add multiple keys at once. Similarly *difference* can be thought of as a “parallel” delete, since it will remove multiple keys at once.

Split. The function *split* takes a tree T and a key k and splits T into two trees: one consisting of all the keys of T less than k , and another consisting of all the keys of T greater than k . It also returns a Boolean value indicating whether k appears in T . The exact structure of the trees returned by *split* can differ from one implementation to another: the specification only requires that the resulting trees to be valid BSTs and that they contain the keys less than k and greater than k , leaving their structure otherwise unspecified.

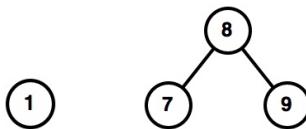
Example 34.6. Consider the following input tree.



- Splitting the input tree at 6 yields two following trees, consisting of the keys less than 6 and those greater than 6, and returns `false` to indicate that 6 is not in the input tree.

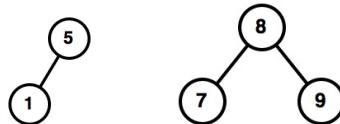


- Splitting the input tree at 5 yields the following two trees, consisting of the keys less than 5 and those greater than 5, and also returns `true` to indicate that 5 is found in the input tree.

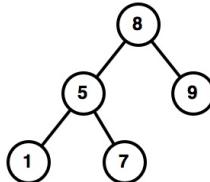


JoinPair. The function *joinPair* takes two trees T_1 and T_2 such that all the keys in T_1 are less than the keys in T_2 . The function returns a tree that contains all the keys in T_1 and T_2 . The exact structure of the tree returned by *joinPair* can differ from one implementation to another: the specification only requires that the resulting tree is a valid BST and that it contains all the keys in the trees being joined.

Example 34.7. Joining the two trees below using the function *joinPair*



yields the following three.



JoinM. The function *joinM* takes a tree T_1 , a key k , and another tree T_2 such that $T_1 < k < T_2$. It returns a tree containing all of the three. It is similar to *joinPair* except it includes the middle value (hence the *M* instead of *Pair*). As with *joinPair* the exact structure of the tree returned can differ from one implementation to another.

Chapter 35

Parametric BSTs

In this chapter we build all the functionality of trees based on a very simple interface in which the only interesting function is *joinMid*. This one function captures all that is needed to rebalance trees to maintain the balance criteria.

1 The Parametric Data Type

We will describe parallel algorithms that support the full functionality of a BST based on the following simple interface, which has only one interesting function, *joinMid*. The interface abstracts away from the particular balancing scheme, and captures everything about balancing in *joinMid*. For most balancing schemes it is powerful enough to efficiently implement all the functionality we want for BSTs. For example it will allow us to design algorithms for *find*, *insert* and *delete* that take $O(\log n)$ work on a tree of size n . It will also allow us to design simple parallel algorithms for various functions.

In the [next chapter](#) we will discuss how to implement this interface with a particular balancing scheme called Treaps. The interface also works with other balancing schemes such as red-black trees, AVL trees, and weight-balanced trees.

Data Type 35.1 (Parametric BST).

```
type  $\mathbb{K}$            (* The key type. Must support  $<$  *)
type  $\mathbb{T}$            (* The abstracted tree type *)
type  $\mathbb{E} = \text{Leaf}$  (* An exposed tree *)
           | Node of  $(\mathbb{T} \times \mathbb{K} \times \mathbb{T})$ 
size :  $\mathbb{T} \rightarrow \mathbb{N}$ 
expose :  $\mathbb{T} \rightarrow \mathbb{E}$ 
joinMid :  $\mathbb{E} \rightarrow \mathbb{T}$   (* Join and rebalance *)
```

The type \mathbb{K} is the key type (from a total order), and the type \mathbb{T} is the type of the tree itself. The *expose* function exposes the root of the tree returning whether the tree is a *Leaf* (i.e., empty) or an internal *Node*. If it is an internal node, the returned type includes a triple of the left subtree, the key at the root, and the right subtree. The actual implementation of the balanced tree is likely to include other information in each node, but this is hidden by the interface.

The function *joinMid* is the inverse of *expose*. It takes either a *Leaf* or a *Node*(L, k, R) consisting of a left tree L , a key k , and a right tree R , where $L < k < R$. If the argument is a leaf, it just creates a leaf (i.e., empty tree). Otherwise, conceptually, *JoinMid* creates a new node with a left branch L , a key k , and a right branch R . However, it might have to rebalance the tree to maintain the invariants of the balancing scheme. This might require some rotations, and the particular rotations required will depend on the balancing scheme. It might also have to update other information in the root node.

The function *size*(T) returns the number of keys in T . Its functionality should be clear, but to implement it efficiently, i.e., in $O(1)$ work, we need to store in each node of the tree the size of its subtree. This is an example of information in the node that is not returned by *expose*, and must be updated by *JoinMid*.

The interface we give does not associate values with each key, but it would be easy to add values. In particular we could define:

$$\begin{aligned} \text{type } \alpha \mathbb{E} = & \text{ Leaf} \\ & \mid \text{ Node of } (\mathbb{T} \times \mathbb{K} \times \alpha \times \mathbb{T}) \end{aligned}$$

where α is the type of the value. This would allow us to implement tables (dictionaries), and all the implementations given in the rest of this chapter still work. We stick with simple sets (no values), because it is simpler.

2 Algorithms based on joinMid

We now discuss algorithms for all the functions in the BST interface given in Section 5 based on just using *joinMid*.

Algorithm 35.2 (Empty, Singleton and JoinM). We start with the following very simple definitions.

$$\text{empty} = \text{joinMid}(\text{Leaf})$$

$$\text{singleton } k = \text{joinMid}(\text{Node}(\text{empty}, k, \text{empty}))$$

$$\text{joinM } (L, k, R) = \text{joinMid}(\text{Node}(L, k, R))$$

Note that $joinM$ is just a version of $JoinMid$ that always takes three arguments (it cannot create a leaf).

Algorithm 35.3 (Split).

```

split( $T, k$ ) =
  case expose( $T$ ) of
    Leaf  $\Rightarrow$  (empty, false, empty)
    | Node( $L, k', R$ )  $\Rightarrow$ 
      case compare( $k, k'$ ) of
        Equal  $\Rightarrow$  ( $L, true, R$ )
        | Less  $\Rightarrow$ 
          let ( $L_l, x, L_r$ ) = split( $L, k$ )
          in ( $L_l, x, joinM(L_r, k', R)$ ) end
        | Greater  $\Rightarrow$ 
          let ( $R_l, x, R_r$ ) = split( $R, k$ )
          in ( $joinM(L, k', R_l), x, R_r$ ) end

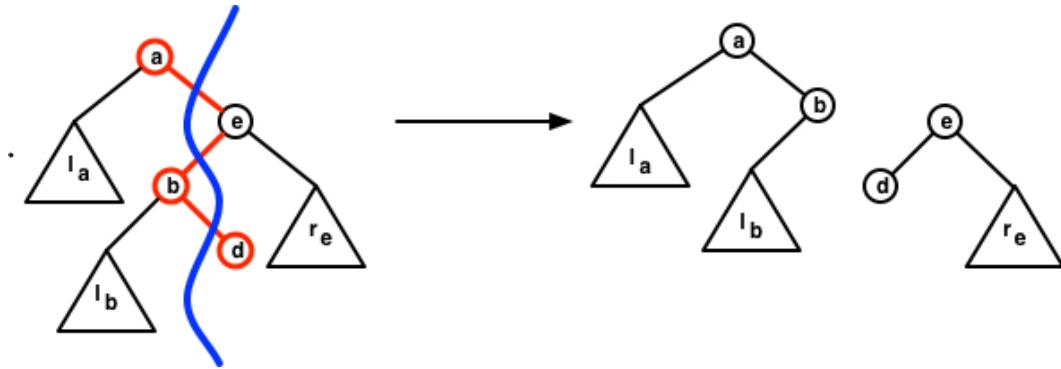
```

The $split(T, k)$ algorithm works by first exposing the root of T . If it is a leaf, we are done and can just return empty for the trees less and greater than k , and false in the middle since the key is not in T . If T is a node, then we need to compare k to the key k' at that node, which can have three outcomes:

1. If $k = k'$ we have found the key k so we return true in the middle and L and R as the lesser and greater trees.
2. If $k < k'$ we know k can only appear in the left subtree L , and that all keys in the right subtree R are greater than k . We can therefore recurse into L using $split(L, k)$. This returns (L_l, x, L_r) . The x tells us whether k appears in L so we can return it as the middle value. Similarly L_l contains all the keys less than k , so we can return it on the left. This leaves us with L_r, k' and R , which are all greater than k . We can join these with $joinM(L_r, k', R)$, giving us the tree of greater keys for the right result.
3. If $k > k'$, it is symmetric to the above.

As we will see, the $split$ function is very useful for other algorithms.

Example 35.1. The $split$ algorithm on a BST and key c , which is not in the BST. The $split$ traverses the path $\langle a, e, b, d \rangle$ turning right at a and b (since c is larger) and turning left at e and d (since c is smaller). The pieces are put back together into the two resulting trees on the way back up the recursion.



Algorithm 35.4 (joinPair).

```

minKey (T, k) =
  case expose T of
    Leaf  $\Rightarrow$  k
  | Node(L, k', _)  $\Rightarrow$  minKey(L, k')

joinPair(T1, T2) =
  case expose(T2) of
    Leaf  $\Rightarrow$  T1
  | Node(L, k, R)  $\Rightarrow$ 
    let km = minKey(L, k)
    (L', R') = split(T2, km)
    in joinM(T1, km, T2')
```

This algorithm works by first finding the minimum key $k_m \in T_2$, then using *split* to remove it from T_2 , returning T_2' , and finally using a *joinM*(T_1, k_m, T_2') to put the parts together.

Algorithm 35.5 (Insert and Delete). An *Insert*(T, k) can be implemented as:

```

insert(T, k) =
  let (L, _, R) = split T k
  in joinM(L, k, R) end
```

It splits the tree T at the key to be inserted, and then joins back together with the key in the middle. Note that whether the key appeared in the original tree is irrelevant and ignored.

A *Delete*(T, k) can be similarly implemented as:

```

delete(T, k) =
  let (L, _, R) = split T k
  in joinPair(L, R) end
```

Again, it splits the tree T at the key to be inserted, and then joins the tree back together, but now without the key in the middle. Note, again, that whether the key appeared in the original tree is irrelevant and ignored.

3 Parallel Functions

So far all our algorithms have been sequential, and, as we will show, all take $O(\log n)$ work. We now look at some functions and corresponding algorithms that take advantage of parallelism.

Algorithm 35.6 (Union). The *union* algorithm uses divide and conquer.

```

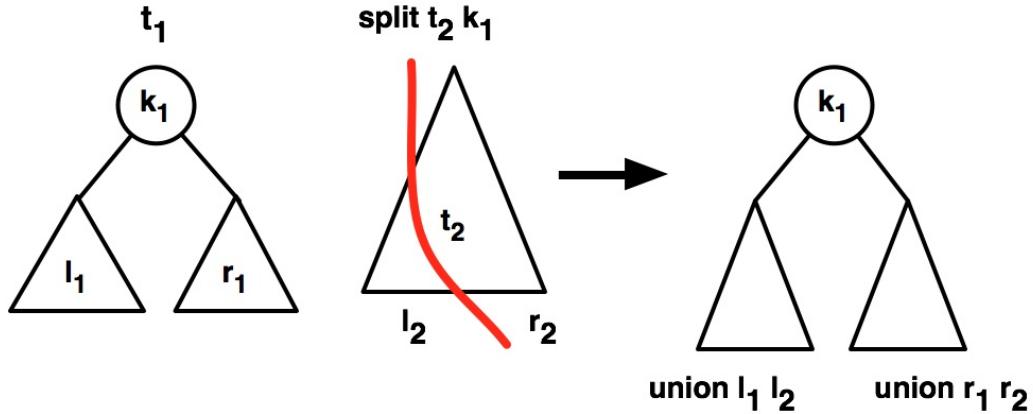
union( $T_1, T_2$ ) =
  case (expose  $T_1, expose T_2$ ) of
    ( $Leaf, \dots$ )  $\Rightarrow T_2$ 
    ( $\dots, Leaf$ )  $\Rightarrow T_1$ 
  | ( $Node(L_1, k_1, R_1), \dots$ )  $\Rightarrow$ 
    let ( $L_2, \dots, R_2$ ) = split( $T_2, k_1$ )
    ( $L, R$ ) = ( $union(L_1, L_2) \parallel union(R_1, R_2)$ )
    in joinM( $L, k_1, R$ ) end

```

The idea is to split both trees at some key k , recursively union the two parts with keys less than k , and the two parts with keys greater than k and then join them. Note that the key k might exist in both trees but will only be placed in the result once, because the *split* operation will not include k in any of the two trees returned.

Note that we chose the key at the root of the first tree, T_1 , to split the second, T_2 . We could equally well have done it the other way, choosing the root of T_2 to split T_1 .

Example 35.2. The union of tree t_1 and t_2 illustrated.



Algorithm 35.7 (Intersect). The *intersection* algorithm is similar to *union*.

```

intersect( $T_1, T_2$ ) =
  case (expose  $T_1$ , expose  $T_2$ ) of
    (Leaf,  $\_$ )  $\Rightarrow$  empty
    ( $\_, \text{Leaf}$ )  $\Rightarrow$  empty
    | (Node( $L_1, k_1, R_1$ ),  $\_$ )  $\Rightarrow$ 
      let ( $L_2, a, R_2$ ) = split( $T_2, k_1$ )
          ( $L, R$ ) = (intersect( $L_1, L_2$ ) || intersect( $R_1, R_2$ ))
      in if  $a$  then joinM( $L, k_1, R$ )
         else joinPair( $L, R$ )
      end
  end

```

As with *union*, the implementation splits both trees by using the key k_1 at the root of the first tree, and computes intersections recursively. It then computes the result by joining the results from the recursive calls and including the key k_1 if it is found in both trees. Note that since the trees are BSTs, checking for the intersections of the left and right subtrees recursively suffices to find all shared keys because the *split* function places all keys less than and greater than the given key to two separate trees.

Algorithm 35.8 (Difference). And there is little difference with the *difference* algorithm.

```

difference( $T_1, T_2$ ) =
  case (expose  $T_1$ , expose  $T_2$ ) of
    ( $Leaf, \dots$ )  $\Rightarrow$  empty
    ( $\dots, Leaf$ )  $\Rightarrow$   $T_1$ 
  | ( $Node(L_1, k_1, R_1), \dots$ )  $\Rightarrow$ 
    let ( $L_2, a, R_2$ ) = split( $T_2, k_1$ )
    ( $L, R$ ) = (difference( $L_1, L_2$ ) || difference( $R_1, R_2$ ))
    in if  $a$  then joinPair( $L, R$ )
    else joinM( $L, k_1, R$ )
  end

```

Exercise 35.1. Prove correct the functions *intersection*, *difference*, and *union*.

Algorithm 35.9 (Filter).

```

filter  $f T$  =
  case expose  $T$  of
     $Leaf \Rightarrow$  empty
  |  $Node(L, k, R) \Rightarrow$ 
    let ( $L', R'$ ) = (filter  $f L$ ) || (filter  $f R$ )
    in if  $f(k)$  then joinM( $L', k, R'$ )
    else joinPair( $L', R'$ )
  end

```

Algorithm 35.10 (Reduce).

```

reduce  $f I T$  =
  case expose  $T$  of
     $Leaf \Rightarrow I$ 
  |  $Node(L, k, R) \Rightarrow$ 
    let ( $L', R'$ ) = (reduce  $f I L$ ) || (reduce  $f I R$ )
    in  $f(L', f(k, R'))$  end

```

4 Cost Specification

There are many ways to implement an efficient data structure that matches our BST ADT. Many of these implementations more or less match the same cost specification, with the main difference being whether the bounds are worst-case, expected case (probabilistic), or amortized. These implementations all use balancing techniques to ensure that the depth of the BST remains $O(\lg n)$, where n is the number of keys in the tree. For the purposes specifying the costs, we don't distinguish between worst-case, amortized, and probabilistic bounds, because we can always rely on the existence of an implementation that matches the desired cost specification. When using specific data structures that match the specified bounds in an amortized or randomized sense, we will try to be careful when specifying the bounds.

Cost Specification 35.11 (BSTs). The *BST* cost specification is defined as follows. The variables n and m are defined as $n = \max(|t_1|, |t_2|)$ and $m = \min(|t_1|, |t_2|)$ when applicable. Here, $|t|$ denotes the size of the tree t , defined as the number of keys in t .

	Work	Span
<i>empty</i>	$O(1)$	$O(1)$
<i>singleton</i> k	$O(1)$	$O(1)$
<i>split</i> t k	$O(\lg t)$	$O(\lg t)$
<i>join</i> t_1 t_2	$O(\lg(t_1 + t_2))$	$O(\lg(t_1 + t_2))$
<i>find</i> t k	$O(\lg t)$	$O(\lg t)$
<i>insert</i> t k	$O(\lg t)$	$O(\lg t)$
<i>delete</i> t k	$O(\lg t)$	$O(\lg t)$
<i>intersect</i> t_1 t_2	$O(m \cdot \lg \frac{n}{m})$	$O(\lg n)$
<i>difference</i> t_1 t_2	$O(m \cdot \lg \frac{n}{m})$	$O(\lg n)$
<i>union</i> t_1 t_2	$O(m \cdot \lg \frac{n}{m})$	$O(\lg n)$

The [Cost Specification for BSTs](#) can be realized by several balanced BST data structures such as Treaps (in expectation), red-black trees (in the worst case), and splay trees (amortized).

In the rest of this section, we justify the cost bounds by assuming the existence of logarithmic time *split* and *join* functions, and by using our parametric implementation described above.

Note that the cost of *empty* and *singleton* are constant. The work and span costs of *find*, *insert*, and *delete* are determined by the *split* and *join* functions and are thus logarithmic in the size of the tree. The cost bounds for *union*, *intersection*, and *difference* are similar; they are also more difficult to see.

4.1 Cost of Union, Intersection, and Difference

We analyze the cost for *union* as implemented by the [parametric data structure](#). Essentially the same analysis applies to the functions *intersection* and *difference*, whose structures are the same as *union*.

For the analysis, we consider an execution of *union* on two trees t_1 and t_2 and define $m = ||t_1||$ and $n = ||t_2||$ as the number of keys in the trees.

We first present an analysis under some relatively strong assumptions. We then show that these assumptions can be eliminated.

Balancing Assumptions. Consider now a call to *union* with parameters t_1 and t_2 . To simplify the analysis, we make several assumptions.

1. **Perfect balance:** t_1 is perfectly balanced (i.e., the left and right subtrees of the root have size at most $\|t_1\|/2$).
2. **Perfect splits:** each time we split t_2 with a key from t_1 , the resulting trees have exactly half the size of t_2 .
3. **Split the larger tree:** $\|t_1\| < \|t_2\|$.

4.1.1 Analysis with Balancing Assumptions

The Recurrence. Under the balancing assumptions, we can write the following recurrence for the work of *union*:

$$W_{\text{union}}(m, n) = 2W_{\text{union}}(m/2, n/2) + W_{\text{split}}(n) + W_{\text{join}}\left(\frac{m+n}{2}, \frac{m+n}{2}\right) + O(1)$$

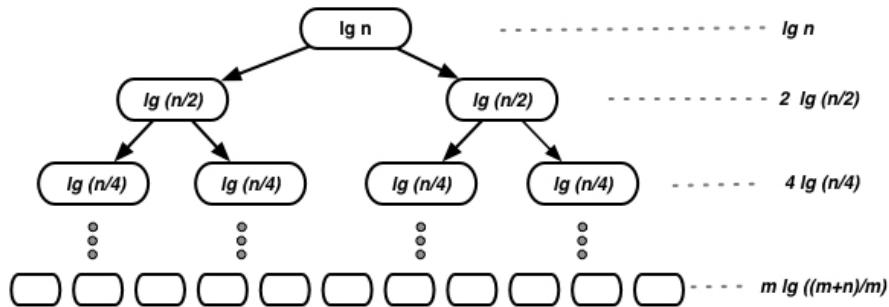
By assumption, we know that

- $W_{\text{split}}(n) = O(\lg n)$, and
- $W_{\text{join}}\left(\frac{m+n}{2}, \frac{m+n}{2}\right) = O(\lg n)$, because $m \leq n$.

We can therefore write the recurrence as

$$W_{\text{union}}(m, n) = \begin{cases} 2W_{\text{union}}(m/2, n/2) + \lg n & \text{if } m > 0 \\ 1 & \text{otherwise} \end{cases}$$

Recurrence Tree. We can draw the recurrence tree showing the work of *union* as shown below. In the illustration, we omit the leaves of the recurrence, which correspond to the case when the first argument t_1 is empty (*Leaf*). The leaves of the recurrence tree thus correspond to calls to *union* where t_1 consists of a single key.



Solving the Recurrence: Brick Method. To solve the recurrence, we start by analyzing the structure of the recursion tree shown above.

Let's find the number of leaves in the tree. Notice that the tree bottoms out when $m = 1$ and before that, m is always split in half (by assumption t_1 is perfectly balanced). The tree t_2 does not affect the shape of the recursion tree or the stopping condition. Thus, there are exactly m leaves in the tree. Notice also that the leaves are at depth $\lg m$.

Let's now determine the size of the argument t_2 at the leaves. We have m keys in t_1 to start with, and by assumption they split t_2 evenly at each recursive call. Thus, the leaves have all the same size of $\frac{n}{m}$. The cost of the leaves is thus $O(\lg \frac{n}{m})$. Since there are m leaves, the whole bottom level costs

$$O(m \lg \frac{n}{m}).$$

We now show that the cost of the leaves dominates the total cost. First observe that the total work at any internal level i in the tree is $2^i \lg n / 2^i$. Thus the ratio of the work at adjacent

levels is

$$\frac{2^{i-1} \lg n / 2^{i-1}}{2^i \lg n / 2^i} = \frac{1}{2} \frac{\lg n - i + 1}{\lg n - i},$$

where $i < \lg m \leq \lg n$. Observe that for all $i < \lg n - 1$, this ratio is less than 1. This means that for all levels except for the last, the total work at each level decreases by a constant fraction less than 1.

Observe now that at the final level in the tree, we have $i = \lg m - 1 \leq \lg n - 1$. For this level, we can bound the fraction from above by taking $i = \lg n - 1$:

$$\frac{1}{2} \frac{\lg n - i + 1}{\lg n - i} \leq \frac{1}{2} \frac{\lg n - \lg n + 1 + 1}{\lg n - \lg n + 1} = 1.$$

Thus the total work is asymptotically dominated by the total work of the leaves, which is $O(m \lg \frac{n}{m})$.

Solving the Recurrence: Direct Derivation. We can establish the same fact more precisely. We start by writing the total cost by summing over all levels, omitting for simplicity the constant factors, and assuming that $n = 2^a$ and $m = 2^b$,

$$W(n, m) = \sum_{i=0}^b 2^i \lg \frac{n}{2^i}.$$

We can rewrite this sum as

$$\sum_{i=0}^b 2^i \lg \frac{n}{2^i} = \lg n \sum_{i=0}^b 2^i - \sum_{i=0}^b i 2^i = a \sum_{i=0}^b 2^i - \sum_{i=0}^b i 2^i.$$

Focus now on the second term. Note that

$$\sum_{i=0}^b i 2^i = \sum_{i=0}^b \sum_{j=i}^b 2^j = \sum_{i=0}^b \left(\sum_{j=0}^b 2^j - \sum_{k=0}^{i-1} 2^k \right).$$

Substituting the closed form for each inner summation and simplifying leads to

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

Now go back and plug this into the original work bound and simplify

$$\begin{aligned}
 W(n, m) &= \sum_{i=0}^b 2^i \lg \frac{n}{2^i} \\
 &= a \sum_{i=0}^b 2^i - \sum_{i=0}^b i 2^i \\
 &= a(2^{b+1} - 1) - (b 2^{b+1} + 1) \\
 &= a 2^{b+1} - a - b 2^{b+1} - 1 = 2m(a - b) - a - 1 \\
 &= 2m(\lg n - \lg m) - a - 1 = 2m \lg \frac{n}{m} - a - 1 \\
 &= O(m \lg \frac{n}{m}).
 \end{aligned}$$

While the direct method may seem complicated, it is more robust than the brick method, because it can be applied to analyze essentially any algorithm, whereas the Brick method requires establishing a geometric relationship between the cost terms at the levels of the tree.

Span Analysis. For the span bound note that the tree has exactly $O(\lg m)$ levels and the cost for each level is $O(\lg n)$. The total span is therefore $O(\lg m \lg n)$. It turns out, though we will not describe here, it is possible to change the algorithm slightly to reduce the span to $O(\lg n)$. We will therefore specify the span of *union* as $O(\lg n)$.

4.1.2 Removing the Balancing Assumptions

The [assumptions](#) that we made for the analysis may seem unrealistic. In this section, we describe how to remove these assumptions.

First Assumption. Let's consider the first assumption, which require t_1 to be perfectly balanced (4.1). This assumption is perhaps the easiest to remove. The balance of the tree t_1 does not impact the work bound, because the work cost is leaf-dominated. But, we do need the tree to have logarithmic depth for the span bound. This means that the tree only needs to approximately balanced.

Removing the Second Assumption. Let's now remove the second assumption (4.1). This assumption requires the keys in t_1 to split subtrees of t_2 in half every time. This is obviously not realistic but any unevenness in the splitting only helps reduce the work—i.e., the perfect split is the worst case. The fundamental reason for this is that logarithm is a concave function.

To see this consider the cost at some level i . There are $k = 2^i$ nodes in the recursion tree and let us say the sizes of second tree at these nodes are n_1, \dots, n_k , where $\sum_j n_j = n$. Then, the

total cost for the level is

$$c \cdot \sum_{j=1}^k \lg(n_j).$$

Because logarithm is a concave function, we know that

$$\begin{aligned} c \cdot \sum_{j=1}^k \lg(n_j) &\leq c \cdot \sum_{j=1}^k \lg(n/k) \\ &= c \cdot 2^i \cdot \lg(n/2^i). \end{aligned}$$

This means that our assumption results in the highest cost and therefore represents the worst case.

Note (Jensen's Inequality). The inequality that we used in the argument above is an instance of Jensen's inequality, named after Johan Jensen, a Danish mathematician.

Removing the Third Assumption. Our final assumption requires t_1 to be smaller than t_2 (4.1).

This is relatively easy to enforce: if t_1 is larger, then we can reverse the order of arguments. If they are the same size, we need to be a bit more precise in our handling of the base case in our summation but that is all.

Chapter 36

Treaps

The parametric data structure presented in Chapter 35 established an interesting observation: to implement the BST ADT, we only need to provide an implementation of *joinMid*.

In this chapter, we implement the 35.1 interface based on a data structure called *Treaps* (tree heaps). We show that *joinMid*, and *split* based on it, take $O(\log n)$ work (and span).

Treaps achieve their efficiency by maintaining BSTs that are probabilistically balanced. Of the many balanced BST data structures, Treaps are likely the simplest, but, since they are randomized, they only guarantee approximate balance, though with high probability.

1 Treap Properties

The idea behind Treaps is to associate each key with a randomly selected priority. Then and in addition to maintaining the BST property on the keys, treaps maintain a “heap ordering” on these priorities. The heap ordering is the same as you might have seen when studying binary heaps, leading to the name “Tree Heap” or shortened to “Treap.”

Definition 36.1 (Treap). A Treap is a binary search tree over a set K along with a *priority* for each key given by

$$p : \mathbb{K} \rightarrow \mathbb{Z}$$

that in addition to satisfying the BST property on the keys K , satisfies the *heap property* on the priorities $p(k), k \in K$. In particular for every internal node u of the tree that has a parent node v :

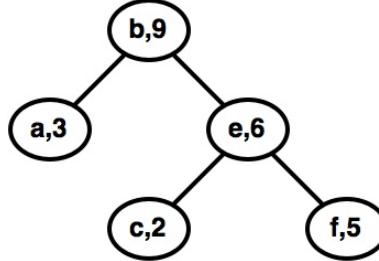
$$p(k(v)) \geq p(k(u))$$

where $k(v)$ denotes the key of a node. This simply states that the priority of the parent is greater than the priorities of its children.

Example 36.1. The following key-priority pairs $(k, p(k))$,

$$(a, 3), (b, 9), (c, 2), (e, 6), (f, 5) ,$$

where the keys are ordered alphabetically, form the following Treap:



since 9 is larger than 3 and 6, and 6 is larger than 2 and 5.

Assigning Priorities. So how do we assign priorities? As we briefly suggested in the informal discussion above, it turns out that if the priorities are selected randomly then the tree is guaranteed to be near balanced, i.e. $O(\lg |S|)$ height, with high probability. We will show this shortly. Since we are using a function $p(\cdot)$ to generate the priorities, we assume it is a random function (i.e., it always returns the same integer for a given key, but which value it returns is random).

We will assume the priorities are unique—i.e., every distinct key maps to a unique priority. This is not necessary for the algorithms and bounds given in this chapter, but it simplifies the description and analysis.

The Treap Type. In our discussion we will use the following recursive type for the definition of a BST type based on treaps.

$$\begin{aligned} \text{type } \mathbb{T} &= T\text{Leaf} \\ &\mid T\text{Node } \text{of } (\mathbb{T} \times \mathbb{K} \times \mathbb{Z} \times \mathbb{T}) \end{aligned}$$

where the $T\text{Node}$ type consists of a 4 tuple (L, k, p, R) , with L as the left child, k as the key, p as the priority, and R as the right child. We use $T\text{Leaf}$ and $T\text{Node}$ to distinguish them from Leaf and Node in the parametric tree interface.

Exercise 36.1. Prove that if the priorities are unique, then there is exactly one tree structure that satisfies the Treap properties.

2 Height Analysis of Treaps

We can analyze the height of Treaps by relating their structure to the recursion tree of

quicksort, which [we have already studied](#).

Algorithm 36.2 (Treap Generating Quicksort). The following variant of quicksort generates a treap. This algorithm is almost identical to our previous quicksort except that it uses *Node* instead of *append*, and because it is generating a treap consisting of unique keys, the algorithm retains only one key equaling the pivot.

```

1  qsTree a =
2    if |a| = 0 then TLeaf
3    else let
4      k = the key  $k \in a$  for which  $p(k)$  is the largest
5      L =  $\langle x \in a \mid x < k \rangle$ 
6      R =  $\langle x \in a \mid x > k \rangle$ 
7       $(L', R') = (qsTree L) \parallel (qsTree R)$ 
8      in
9      TNode (L', k, p(k), R')
10     end

```

The tree generated by $qsTree(a)$ is the Treap for the sequence a . This can be seen by induction. It is true for the base case. Now assume by induction it is true for the trees returned by the two recursive calls. The tree returned by the main call is then also a Treap since the pivot x has the highest priority, and therefore is correctly placed at the root, the subtrees and in heap order by induction, and because the keys in l are less than the pivot, and the keys in r are greater than the pivot, the tree has the BST property.

Based on this isomorphism, we can bound the height of a Treap by the recursion depth of quicksort. Recall that when studying the [order statistics problem](#) we proved that if we pick the priorities at random, the recursion depth is $O(\lg n)$ with high probability. Based on this fact, and by using [union bound](#), we then proved that the depth of the quicksort recursion (pivot) tree is [logarithmic— \$O\(\lg n\)\$ —with high probability](#). We therefore conclude that that the height of a Treap is $O(\lg n)$ with high probability.

3 The Treap Data Structure

We are now ready to implement the [35.1](#) ADT with Treaps. In particular we need an algorithm for the *joinMid* function. It must maintain the Treap invariants. We hold off on implementing a *size* function until the next chapter when we discuss augmenting trees. Our Treap implementation of the parametric BST ADT is defined as follows.

Data Structure 36.3 (Treaps). An implementation of the [35.1](#) ADT.

```

type  $\mathbb{K}$ 
type  $\mathbb{T} = TLeaf \mid TNode \text{ of } (\mathbb{T} \times \mathbb{K} \times \mathbb{Z} \times \mathbb{T})$ 
type  $\mathbb{E} = Leaf \mid Node \text{ of } (\mathbb{T} \times \mathbb{K} \times \mathbb{T})$ 

priority  $T =$ 
  case  $T$  of
     $TLeaf \Rightarrow -\infty$ 
    |  $TNode(L, k, p, R) \Rightarrow p$ 

join( $T_1, (k, p), T_2$ ) :  $\mathbb{T} \times (\mathbb{K} \times \mathbb{Z}) \times \mathbb{T} \rightarrow \mathbb{T} =$ 
  if ( $p > \text{priority}(T_1)$ )  $\wedge (p > \text{priority}(T_2))$  then
     $TNode(T_1, k, p, T_2)$ 
  else if ( $\text{priority}(T_1) > \text{priority}(T_2)$ ) then
    case  $T_1$  of  $TNode(L_1, k_1, p_1, R_1)$ 
       $\Rightarrow TNode(L_1, k_1, p_1, \text{join}(R_1, (k, p), T_2))$ 
  else
    case  $T_2$  of  $TNode(L_2, k_2, p_2, R_2)$ 
       $\Rightarrow TNode(\text{join}(T_1, (k, p), L_2), k_2, p_2, R_2)$ 

expose  $T : \mathbb{T} \rightarrow \mathbb{E} =$ 
  case  $T$  of
     $TLeaf \Rightarrow Leaf$ 
    |  $TNode(L, k, R) \Rightarrow Node(L, k, R)$ 

joinMid  $T : \mathbb{E} \rightarrow \mathbb{T} =$ 
  case  $T$  of
     $Leaf \Rightarrow TLeaf$ 
    |  $Node(L, k, R) \Rightarrow \text{join}(L, (k, p(k)), R)$ 

```

Join Algorithm. We now consider the algorithm for $\text{join}(T_1, (k, p), T_2)$. Here p is the priority of the key k . Due to the requirements of joinMid we are ensured that $T_1 < k < T_2$, and we assume that T_1 and T_2 each satisfy the treap invariants (i.e., BSTs and heap ordered priorities). To maintain the treap invariants on the results, we not only need to maintain the ordering for a BST, but also need to maintain the heap property. In the following discussion, we refer to the priority of a tree as the priority of its root if it is a node, or $-\infty$ if it is a leaf. The helper function $\text{priority}(T)$ returns this priority using $O(1)$ work.

To maintain the heap property, the algorithm first checks if it is “lucky” and priority p is already greater than the priority of T_1 and T_2 . In this case it can just make a node directly and is done. If not, it then needs to check which of T_1 and T_2 has a higher priority since the root of that one needs to become the overall root. In the first case $\text{priority}(T_1) > \text{priority}(T_2)$. In this case since the priority is not negative infinity, we know T_1 is a node (not a leaf) and let (L_1, k_1, p_1, R_1) be its contents (we need not match on $TLeaf$. The key k_1 needs to be at the root since it has the highest priority, and L_1 needs to be its left child since all other keys

are greater than k_1 . This leaves us with R_1 , k , and T_2 . These can just be joined recursively with $\text{join}(R_1, k, p, T_2)$. We know that $R_1 < k < T_2$, so the arguments are valid. There is a symmetric case when $\text{priority}(T_1) < \text{priority}(T_2)$.

Cost of Join. Calculating the cost of join is straightforward. In particular on each step it either finishes, goes down one level in T_1 , or goes down one level in T_2 . In each case the work before the recursive call is constant. Therefore the overall work for join is bounded by $h(T_1) + h(T_2)$. As stated the height of a treap T is bounded by $O(\log |T|)$ with high probability. This means the cost of join is $O(\log |T_1| + \log |T_2|) = O(\log(|T_1| + |T_2|))$ with high probability.

Cost of Split. We now analyze the cost of [the *split* algorithm](#) based on our implementation of *joinMid*.

We note that the *split* algorithm traverses the tree visiting each level once. Now at each level it does constant work plus the work of the *joinM*. Notice, however, that the key k' used in the $\text{joinM}(L_r, k', R)$ (or $\text{joinM}(L, k', R_l)$) has a higher priority than either of the two trees L_r and R . This is because in the input tree, it was above both subtrees, and by the treap invariant must have had a higher priority. Therefore the *join* as described above will take constant work—it just needs to check that the priority of the key is greater than the priority of both trees, and can then on success will make the treap node immediately. Therefore the overall cost of each recursive call in *split* is constant, and the overall cost of $\text{split}(T)$ is $O(h(|T|))$, which is $O(\log |T|)$ with high probability.

Chapter 37

Augmenting Binary Search Trees

In our discussions of BSTs thus far, we only stored the left and right children, the key, and some balance information (the priority for treaps) within each node. In many cases, we wish to augment tree nodes with more information. In this chapter, we describe how we might augment BST nodes with such additional information, e.g., additional values, subtree sizes, and other aggregate values of the subtree, such as the sum of the keys.

1 Augmenting with Values

Perhaps the simplest form of augmentation involves storing a value along with each key in the BST. This allows us to represent a mapping from each key to an associated value. Such mappings are sometimes called dictionaries, tables, or key-value stores.

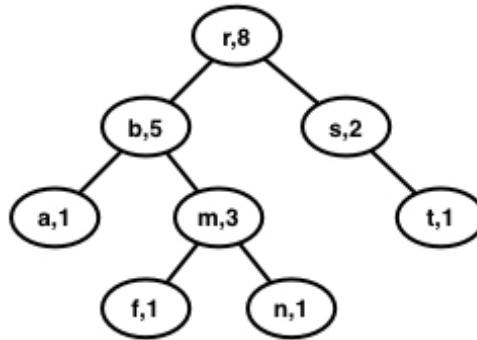
Implementing BSTs augmented with values is straightforward. Firstly we store the value in each node along with its key. We often refer to these as a key-value pair. Many of our functions will then use the key-value pair as an argument. In some cases, we might need to change the types of functions. For example, *find* and *split* will now likely return an optional value instead of a boolean.

A complete interface for tables is given in Chapter 39.

2 Augmenting with Size

As a more complex augmentation, we might want to associate with each node in the tree a size field that tells us how many keys there are in the subtree rooted at that node is.

Example 37.1. An example BST, where keys are ordered lexicographically and the nodes are augmented with the sizes of subtrees.



Size in $O(1)$ Work. To implement a size-augmented tree, we can keep a *size* field at each node and compute the size of the nodes as they are created. To support this field in our tree data structure, such as treaps, we can make sure each tree node has a size field. For example, for treaps a node could be defined as

```

type T = TLeaf
| TNode of (T * K * Z * Z * T)
  
```

where the *TNode* type consists of a 5 tuple (L, k, p, n, R) , with L as the left child, k as the key, p as the priority, n as the size, and R as the right child. We could then read the size in $O(1)$ work, as in:

```

size T =
  case T of
    TLeaf => 0
    | TNode(_, _, _, n, _) => n
  
```

Whenever we create a new node we can calculate its size by summing the sizes of the two subtrees and adding one more for the node itself. For treaps we could define

```

makeNode (L, k, p, R) =
  TNode(L, k, p, size(L) + size(R) + 1, R)
  
```

Then in the [join algorithm for Treaps](#), we could replace the three occurrences $TNode(\cdot, \cdot, \cdot, \cdot)$ with $makeNode$. That is the only change that needs to be made.

2.1 Example: Rank and Select in BSTs

Suppose that we wish to extend the [BST ADT](#) with the following additional functions.

- Function $rank T k$ returns the rank of the key k in the tree, i.e., the number of keys in t that are less than or equal to k .
- Function $select T i$ returns the key with the rank i in t .

Such functions arise in many applications.

Algorithm 37.1 (Rank and Select). If we have a way to count the number of nodes in a subtree, then we can easily implement the $rank$ and $select$ functions. The algorithms below give implementations by using a size operation for computing the size of a tree, written $|T|$ for tree T .

```

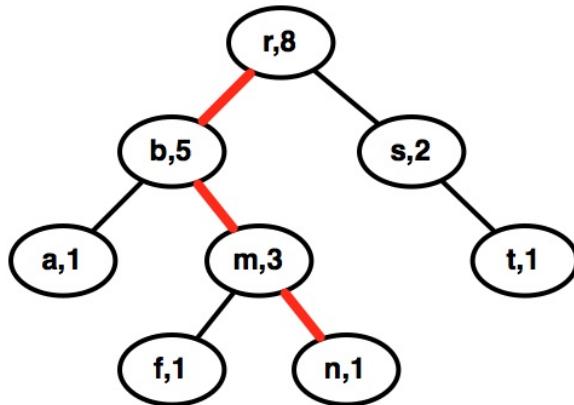
1  rank T k =
2  case expose T of
3    Leaf  $\Rightarrow$  0
4    | Node (L, k', R)  $\Rightarrow$ 
5      case compare (k, k') of
6        Less  $\Rightarrow$  rank L k
7        | Equal  $\Rightarrow$  |L| + 1
8        | Greater  $\Rightarrow$  |L| + 1 + (rank R k)

1  select T i =
2  case expose T of
3    Leaf  $\Rightarrow$  raise exception OutOfRange
4    | Node (L, k, R)  $\Rightarrow$ 
5      case compare (i, |L| + 1) of
6        Less  $\Rightarrow$  select L i
7        | Equal  $\Rightarrow$  k
8        | Greater  $\Rightarrow$  select R (i - |L| - 1)

```

Cost of rank and select. With balanced trees such as Treaps, the $rank$ and $select$ functions require logarithmic span but linear work, because computing the size of a subtree takes linear time in the size of the subtree. If, however, we augment the tree so that at each node, we store the size of the subtree rooted at that node, then work becomes logarithmic, because we can find the size of a subtree in constant work.

Example 37.2. An example BST, where keys are ordered lexicographically and the nodes are augmented with the sizes of subtrees. The path explored by $rank(T, n)$ and $select(T, 4)$ is highlighted.

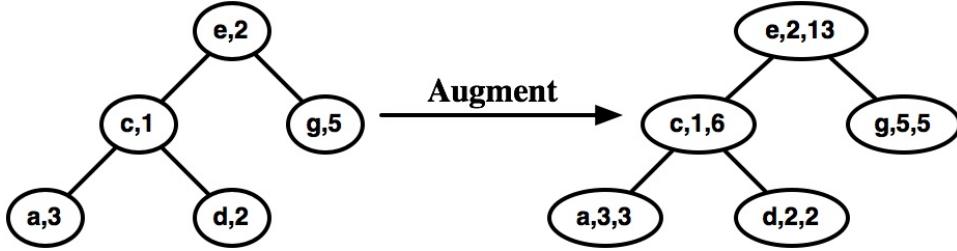


Exercise 37.1. Consider the function $splitRank(t, i)$, which splits the tree t into two and returns the trees t_1 and t_2 such that t_1 contains all keys with rank less than i and t_2 contains all keys with rank is greater or equal to i . Such a function can be used for example to write divide-and-conquer algorithms on imperfectly balanced trees. Describe how to implement the algorithm $splitRank$ by writing its pseudo-code and analyze its work and span.

3 Augmenting with Reduced Values

To compute rank-based properties of keys in a BST, we augmented the BST so that each node stores the size of its subtree. More generally, we might want to associate with each node a *reduced value* that is computed by reducing over the subtree rooted at the node by a user specified associative function f . In general, there is no restriction on how the reduced values may be computed, they can be based on keys or additional values that the tree is augmented with. To compute reduced values, we simply store with every node u of a binary search tree, the reduced value of its subtree (i.e. the sum of all the reduced values that are descendants of u , possibly also the value at u itself).

Example 37.3. The following drawing shows a tree with key-value pairs on the left, and the augmented tree on the right, where each node additionally maintains the sum of its subtree.



The sum at the root (13) is the sum of all values in the tree ($3 + 1 + 2 + 2 + 5$). It is also the sum of the reduced values of its two children (6 and 5) and its own value 2.

The value of each reduced value in a tree can be calculated as the sum of its two children plus the value stored at the node. This means that we can maintain these reduced values by simply taking the “sum” of these three values whenever a node is created. We can thus change a data structure to support reduced values by changing the way a node is created. In such a data structure, if the function that we use for reduction performs constant work, then the work and the span bound for the data structure remains unaffected.

The changes to the Treap structure (or any other balanced tree structure) would be very similar to what we did to augment with sizes. In particular we would change the type of a *TNode* to include the reduced value. We would then add a function for extracting it. This would return the identity *I* for *f* if the tree is empty. Finally we would modify our *makeNode* function to apply *f*. Note that for reduced values to make sense we need to store both a value at each node, and the reduced value of all such values in the subtree. Assuming values have type *val*, we could change the tree nodes to:

```

type T = TLeaf
| TNode of (T × K × Z × Z × val × val × T)

```

where the *TNode* type now consists of a 7 tuple (L, k, p, n, v, r, R) , with *L* as the left child, *k* as the key, *p* as the priority, *n* as the size, *v* as the value, *r* as the reduced value, and *R* as the right child. The changes to the code are then simply:

```

reducedVal T =
  case T of
    TLeaf => I
    | TNode(_, _, _, _, r, _) => r

makeNode (L, k, v, p, R) =
  let r = f(reducedVal(L), f(v, reducedVal(R)))
  s = size(L) + 1 + size(R)
  in TNode(L, k, p, s, v, r, R) end

```

Note. This idea can be used with any binary search tree, not just Treaps. We only need to replace the function for creating a node so that as it creates the node, it also computes a reduced value for the node by summing the reduced values of the children and the value of the node itself.

Remark. In an imperative implementation of binary search trees, when a child node is (destructively) updated, the reduced values for the nodes on the path from the modified node to the root must be recomputed.

Part VIII

Sets and Tables

Chapter 38

Sets

Sets are both a mathematical structure and an abstract datatype supported by many programming languages. This chapter presents an ADT for finite sets of elements taken from a fixed domain and several cost models, including one based on [balanced binary search trees](#).

1 Motivation

“A *set* is a gathering together into a whole of definite, distinct objects of our perception or of our thought—which are called *elements* of the set.”

Georg Cantor, from “Contributions to the founding of the theory of transfinite numbers.”

Set theory, founded by Georg Cantor in the second half of the nineteenth century, is one of the most important advances in mathematics. From it came the notions of countably vs. uncountably infinite sets, and ultimately the theory of computational undecidability, i.e. that computational mechanisms such as the λ -calculus or Turing Machine cannot compute all functions. Set theory has also formed the foundations on which other branches of mathematics can be formalized. Early set theory, sometimes referred to as naïve set theory, allowed anything to be in a set. This led to several paradoxes such Russell’s famous paradox:

let $R = \{x \mid x \notin x\}$, then $R \in R \iff R \notin R$.

Such paradoxes were resolved by the development of axiomatic set theory. Typically in such a theory, the universe of possible elements of a set needs to be built up from primitive notions, such as the integers or reals, by using various composition rules, such as Cartesian products.

Our goals in this book are much more modest than trying to understand set theory and its many interesting implications. Here we simply recognize that in algorithm design sets are a very useful data type in their own right, but also in building up more complicated data types, such as graphs. Furthermore particular classes of sets, such as mappings (or tables), are themselves very useful, and hence deserve their own interface. We discuss tables [in the next chapter](#).

Other chapters cover data structures on binary search trees (Chapter 34) and hashing (Chapter 63) that can be used for implementing the sets and tables interfaces. Applications of sets and tables include [graphs](#). We note that [sequences](#) are a particular type of table—one where the domain of the tables are the integers from 0 to $n - 1$.

2 Sets ADT

Data Type 38.1 (Sets). For a universe of elements \mathbb{U} that support equality (e.g. the integers or strings), the SET abstract data type is a type \mathbb{S} representing the power set of \mathbb{U} (i.e., all subsets of \mathbb{U}) limited to sets of finite size, along with the functions below.

<i>size</i>	$: \mathbb{S} \rightarrow \mathbb{N}$
<i>toSeq A</i>	$: \mathbb{S} \rightarrow \mathbb{Seq}$
<i>empty</i>	$: \mathbb{S}$
<i>singleton</i>	$: \mathbb{U} \rightarrow \mathbb{S}$
<i>fromSeq</i>	$: \mathbb{Seq} \rightarrow \mathbb{S}$
<i>filter</i>	$: ((\mathbb{U} \rightarrow \mathbb{B}) \rightarrow \mathbb{S}) \rightarrow \mathbb{S}$
<i>intersection</i>	$: \mathbb{S} \rightarrow \mathbb{S} \rightarrow \mathbb{S}$
<i>difference</i>	$: \mathbb{S} \rightarrow \mathbb{S} \rightarrow \mathbb{S}$
<i>union</i>	$: \mathbb{S} \rightarrow \mathbb{S} \rightarrow \mathbb{S}$
<i>find</i>	$: \mathbb{S} \rightarrow \mathbb{U} \rightarrow \mathbb{B}$
<i>delete</i>	$: \mathbb{S} \rightarrow \mathbb{U} \rightarrow \mathbb{S}$
<i>insert</i>	$: \mathbb{S} \rightarrow \mathbb{U} \rightarrow \mathbb{S}$

Where \mathbb{N} is the natural numbers (non-negative integers) and $\mathbb{B} = \{\text{true}, \text{false}\}$.

Syntax 38.2 (Sets). In SPARC we use the standard set notation $\{e_0, e_1, \dots, e_n\}$ to indicate a set. The notation \emptyset or $\{\}$ refers to an empty set. We also use the conventional mathematical syntax for set functions such as $|S|$ (size), \cup (union), \cap (intersection), and \setminus (difference). In addition, we use set comprehensions for *filter* and for constructing sets from other sets.

The objects that are contained in a set are called [members](#) or the [elements](#) of the set. Recall that a [set](#) is a collection of distinct objects. This requires that the universe \mathbb{U} they come from support equality. It might seem that all universes support equality, but consider functions. When are two functions equal? It is not even decidable whether two functions are equal. From a practical matter, there is no way to implement sets without an equality function

over potential elements. In fact efficient implementations additionally require either a hash function over the elements of \mathbb{U} and/or a total ordering.

The Set ADT consists of basic functions on sets. The function *size* takes a set and returns the number of elements in the set. The function *toSeq* converts a set to a sequence by ordering the elements of the set in an unspecified way. Since elements of a set do not necessarily have a total ordering, the resulting order is arbitrary. This means that *toSeq* is could return different orderings in different implementations. We, however, assume it always returns the same ordering for the same set when using the same implementation—i.e., it is a function.. We specify *toSeq* as follows

$$\text{toSeq } (\{x_0, x_1, \dots, x_n\} : \mathbb{S}) : \text{seq} = \langle x_0, x_1, \dots, x_n \rangle$$

where the x_i are an arbitrary ordering.

Several functions enable constructing sets. The function *empty* returns an empty set:

$$\text{empty} : \mathbb{S} = \emptyset$$

The function *singleton* constructs a singleton set from a given element.

$$\text{singleton}(x : \mathbb{U}) : \mathbb{S} = \{x\}$$

The function *fromSeq* takes a sequence and returns a set consisting of the distinct elements of the sequence, eliminating duplicate elements. We can specify *fromSeq* as returning the range of the sequence A (recall that a sequence is a partial function mapping from natural numbers to elements of the sequence).

$$\text{fromSeq } (a : \text{seq}) : \mathbb{S} = \text{range } a$$

Several functions operate on sets to produce new sets. The function *filter* selects the elements of a sequence that satisfy a given Boolean function, i.e.,

$$\text{filter } (f : \mathbb{U} \rightarrow \mathbb{B}) (a : \mathbb{S}) : \mathbb{S} = \{x \in a \mid f(x)\}.$$

The functions *intersection*, *difference*, and *union* perform the corresponding set operation on their arguments:

$$\text{intersection } (a : \mathbb{S}) (b : \mathbb{S}) : \mathbb{S} = a \cap b$$

$$\text{difference } (a : \mathbb{S}) (b : \mathbb{S}) : \mathbb{S} = a \setminus b$$

$$\text{union } (a : \mathbb{S}) (b : \mathbb{S}) : \mathbb{S} = a \cup b$$

We refer to the functions *intersection*, *difference*, and *union* as *bulk updates*, because they allow updating with a large set of elements “in bulk.”

The functions *find*, *insert*, and *delete* are singular versions of the bulk functions *intersection*, *union*, and *difference* respectively. The *find* function checks whether an element is in a set—it is the basic membership test for sets.

$$\text{find } (a : \mathbb{S}) (x : \mathbb{U}) : \mathbb{B} = \begin{cases} \text{true} & \text{if } x \in A \\ \text{false} & \text{otherwise} \end{cases}$$

We can also specify the *find* function in terms of set intersection:

$$\text{find } (a : \mathbb{S}) (x : \mathbb{U}) : \mathbb{B} = |a \cap \{x\}| = 1.$$

The functions *delete* and *insert* delete an existing element from a set, and insert a new element into a set, respectively:

$$\begin{aligned} \text{delete } (a : \mathbb{S}) (x : \mathbb{U}) : \mathbb{S} &= a \setminus \{x\} \\ \text{insert } (a : \mathbb{S}) (x : \mathbb{U}) : \mathbb{S} &= a \cup \{x\} \end{aligned}$$

Iteration and reduction over sets can be easily defined by converting them to sequences, as in

$$\begin{aligned} \text{iterate } f x a &= \text{Sequence.iterate } f (\text{toSeq } a) \\ \text{reduce } f x a &= \text{Sequence.reduce } f (\text{toSeq } a) \end{aligned}$$

Notice that in the Set ADT although the universe \mathbb{U} is potentially infinite (e.g. the integers), \mathbb{S} only consists of finite sized subsets. Unfortunately this restriction means that the interface is not as powerful as general set theory, but it makes computation on sets feasible. A consequence of this requirement is that the interface does not include a function that takes the complement of a set—such a function would generate an infinite sized set from a finite sized set (assuming the size of U is infinite).

Exercise 38.1. Convince yourself that there is no way to create an infinite sized set using the interface and with finite work.

Example 38.1. Some functions on sets:

$$\begin{aligned} |\{a, b, c\}| &= 3 \\ \{x \in \{4, 11, 2, 6\} \mid x < 7\} &= \{4, 2, 6\} \\ \text{find } \{6, 2, 9, 11, 8\} 4 &= \text{false} \\ \{2, 7, 8, 11\} \cup \{7, 9, 11, 14, 17\} &= \{2, 7, 8, 9, 11, 14, 17\} \\ \text{toSeq } \{2, 7, 8, 11\} &= \langle 8, 11, 2, 7 \rangle \\ \text{fromSeq } \langle 2, 7, 2, 8, 11, 2 \rangle &= \{8, 2, 11, 7\} \end{aligned}$$

Remark. You may notice that the interface does not contain a *map* function. If we interpret *map*, as in sequences, to take in a collection, apply some function to each element and return a collection of the same size, then it does not make sense for sets. Consider a function that always returns 0. Mapping this over a set would return all zeros, which would then be collapsed into a singleton set, containing exactly 0. Therefore, such a *map* would allow reducing the set of arbitrary size to a singleton.

Remark. Most programming languages either support sets directly (e.g., Python and Ruby) or have libraries that support them (e.g., in the C++ STL library and Java collections framework). They sometimes have more than one implementation of sets. For example, Java has sets based on hash tables and balanced trees. Unsurprisingly, the set interface in different libraries and languages differ in subtle ways. So, when using one of these interfaces you should always read the documentation carefully.

3 Cost of Sets

Sets can be implemented in several ways. If the elements of a set are drawn from natural numbers, it is sometimes possible and effective to represent the set as an array-based sequence. If the elements accept a hash function, then hash-tables can be used to store the elements in a sequence. This approach is commonly used in practice and is quite efficient in terms of work and space. Finally, if the elements don't accept a hash function but accept a comparison operator that can totally order all elements, then sets can be represented by using binary search trees. All of these approaches assume an equality function on the elements (with natural numbers, the equality coincides with the equality on natural numbers).

Cost Specification 38.3 (Arrays for Enumerable Sets). Let the universe U be defined as the set $\{0, 1, \dots, u - 1\}$ for some $u \in \mathbb{N}$. We can represent *enumerable sets* of the form $S \subseteq U$ by using a boolean sequence of length $|U|$ that indicates for each $i \in U$ whether $i \in S$ or not. Using array sequences, operations on enumerable sets can be implemented according to the following cost specification.

Operation	Work	Span
<i>size a</i>	u	1
<i>singleton x</i>	u	1
<i>toSeq a</i>	u	1
<i>filter f a</i>	$u + \sum_{x \in a} W(f(x))$	$1 + \max_{x \in a} S(f(x))$
<i>intersection a₁ a₂</i>	u	1
<i>union a₁ a₂</i>	u	1
<i>difference a₁ a₂</i>	u	1
<i>find a e</i>	1	1
<i>insert a x</i>	u	1
<i>delete a x</i>	u	1

Example 38.2. Consider a graph whose vertices are labeled by natural numbers up to 8. We can represent a graph whose vertices are $\{0, 2, 4, 6\}$ with the sequence: $\langle 1, 0, 1, 0, 1, 0, 1, 0 \rangle$.

With some simple optimization these bounds can be improved. For example we could include the size or the set along with the sequence boolean sequence, which would reduce the cost of *size* to $O(1)$. Also, since words of memory can contain multiple bits, we could also pack multiple bits into each word. If words have b bits, this would convert each $O(u)$ bound to $O(u/b)$.

Tree Representation for Sets. If the elements in the universe \mathbb{U} accept a comparison function that defines a total order over \mathbb{U} , then we can use a balanced binary search tree to represent sets. This representation allows us to implement the Sets ADT reasonably efficiently.

For the specification [specified below](#), we assume that the comparison function requires constant work and span.

Cost Specification 38.4 (Tree Sets). The cost specification for tree-based implementation of sets follow.

Operation	Work	Span
<i>size a</i>	1	1
<i>singleton x</i>	1	1
<i>toSeq a</i>	$ a $	$\lg a $
<i>filter f a</i>	$\sum_{x \in a} W(f(x))$	$\lg a + \max_{x \in a} S(f(x))$
<i>intersection a b</i>	$m \cdot \lg(1 + \frac{n}{m})$	$\lg(n)$
<i>union a b</i>	"	"
<i>difference a b</i>	"	"
<i>find a e</i>	$\lg a $	$\lg a $
<i>insert a x</i>	"	"
<i>delete a x</i>	"	"

where $n = \max(|a|, |b|)$ and $m = \min(|a|, |b|)$, and assuming comparison of elements takes constant work.

Let's consider these cost specifications in some more detail. The cost for *filter* is effectively the same as for sequences, and therefore should not be surprising. It assumes the function f is applied to the elements of the set in parallel. The cost for the functions *find*, *insert*, and *delete* are what one might expect from a balanced binary tree implementation. Basically the tree will have $O(\lg n)$ depth and each function will require searching the tree from the root to some node. We cover such an implementation in Chapter 35.

The work bounds for the bulk functions (*intersection*, *difference*, and *union*) may seem confusing, especially because of the expression inside the logarithm. To shed some light on the cost, it is helpful to consider two cases, the first is when one of the sets is a single element and the other when both sets are equal length. In the first case the bulk functions are doing the same thing as the single element functions *find*, *insert*, and *delete*. Indeed if we implement the single element functions on a set A using the bulk ones, then we would like it to be the case that we get the same asymptotic performance. This is indeed the case since we have that $m = 1$ and $n = |A|$, giving:

$$W(n) \in O\left(\lg\left(1 + \frac{n}{1}\right)\right) = O(\lg n).$$

Now let's consider the second case when both sets have equal length, say n . In this case we have $m = n$ giving

$$W(n) \in O\left(n \cdot \lg\left(1 + \frac{n}{n}\right)\right) = O(n).$$

We can implement *find*, *delete*, and *insert* in terms of the functions *intersection*, *difference*, and *union* (respectively) by making a singleton set out of the element that we are interested in. Such an implementation would be asymptotically efficient, giving us the work and span as the direct implementations.

Conversely, we can also implement the bulk functions in terms of the singleton ones by iteration. Because it uses iteration the resulting algorithms are sequential and also work inefficient. For example, if we implement *union* by inserting n elements into a second set of n elements, the cost would be $O(n \lg n)$. We would obtain a similar bound when implementing *difference* with *delete*, and *intersection* with *find* and *insert*. For this reason, we prefer to use the bulk functions *intersection*, *difference*, and *union* instead of *find*, *delete*, and *insert* when possible.

Example 38.3. We can convert a sequence to a set by inserting the elements one by one as follows

```
fromseq a = Seq.iterate Set.insert ∅ a.
```

This implementation is work efficient but sequential. We can write a work efficient and parallel function as follows

```
fromSeq a = Seq.reduce Set.union ∅ ⟨{x} : x ∈ a⟩.
```

Chapter 39

Tables

In this chapter we describe an interface and cost model for tables (also called maps or dictionaries). In addition to traditional functions such as insertion, deletion and search, we consider bulk functions that are better suited for parallelism.

1 Interface

The ability to map keys to associated values is crucial as a component of many algorithms and applications. Mathematically this corresponds to a map or function. In programming languages, data types to support such functionality are variously called maps, dictionaries, tables, key-value stores, and associative arrays. In this book we use the term table. Traditionally the focus has been on supporting insertion, deletion and search. In this book we are interested in parallelism so, as with sets, it is useful to have bulk functions, such as intersection and map.

Data Type 39.1 (Tables). For a universe of keys \mathbb{K} supporting equality, and a universe of values \mathbb{V} , the TABLE abstract data type is a type \mathbb{T} representing the power set of $\mathbb{K} \times \mathbb{V}$

restricted so that each key appears at most once. The data type supports the following:

<i>size</i>	$: \mathbb{T} \rightarrow \mathbb{N}$
<i>empty</i>	$: \mathbb{T}$
<i>singleton</i>	$: \mathbb{K} \times \mathbb{V} \rightarrow \mathbb{T}$
<i>domain</i>	$: \mathbb{T} \rightarrow \mathbb{S}$
<i>tabulate</i>	$: (\mathbb{K} \rightarrow \mathbb{V}) \rightarrow \mathbb{S} \rightarrow \mathbb{T}$
<i>map</i>	$: (\mathbb{V} \rightarrow \mathbb{V}) \rightarrow \mathbb{T} \rightarrow \mathbb{T}$
<i>filter</i>	$: (\mathbb{K} \times \mathbb{V} \rightarrow \mathbb{B}) \rightarrow \mathbb{T} \rightarrow \mathbb{T}$
<i>intersection</i>	$: (\mathbb{V} \times \mathbb{V} \rightarrow \mathbb{V}) \rightarrow \mathbb{T} \rightarrow \mathbb{T} \rightarrow \mathbb{T}$
<i>union</i>	$: (\mathbb{V} \times \mathbb{V} \rightarrow \mathbb{V}) \rightarrow \mathbb{T} \rightarrow \mathbb{T} \rightarrow \mathbb{T}$
<i>difference</i>	$: \mathbb{T} \rightarrow \mathbb{T} \rightarrow \mathbb{T}$
<i>find</i>	$: \mathbb{T} \rightarrow \mathbb{K} \rightarrow (\mathbb{V} \cup \perp)$
<i>delete</i>	$: \mathbb{T} \rightarrow \mathbb{K} \rightarrow \mathbb{T}$
<i>insert</i>	$: (\mathbb{V} \times \mathbb{V} \rightarrow \mathbb{V}) \rightarrow \mathbb{T} \rightarrow (\mathbb{K} \times \mathbb{V}) \rightarrow \mathbb{T}$
<i>restrict</i>	$: \mathbb{T} \rightarrow \mathbb{S} \rightarrow \mathbb{T}$
<i>subtract</i>	$: \mathbb{T} \rightarrow \mathbb{S} \rightarrow \mathbb{T}$

Throughout the set \mathbb{S} denotes the powerset of the keys \mathbb{K} , \mathbb{N} are the natural numbers (non-negative integers), and $\mathbb{B} = \{\text{true}, \text{false}\}$.

Syntax 39.2 (Table Notation). In the book we will write

$$\{k_1 \mapsto v_1, k_2 \mapsto v_2, \dots\}$$

for a table that maps k_i to v_i .

The function *size* returns the size of the table, defined as the number of key-value pairs, i.e.,

$$\text{size } (a : \mathbb{T}) : \mathbb{N} = |a|.$$

The function *empty* generates an empty table, i.e.,

$$\text{empty} : \mathbb{T} = \emptyset.$$

The function *singleton* generates a table consisting of a single key-value pair, i.e.,

$$\text{singleton } (k : \mathbb{K}, v : \mathbb{V}) : \mathbb{T} = \{k \mapsto v\}.$$

The function *domain*(a) returns the set of all keys in the table a .

Larger tables can be created by using the *tabulate* function, which takes a function and a set of key and creates a table by applying the function to each element of the set, i.e.,

$$\text{tabulate } (f : \mathbb{K} \rightarrow \mathbb{V}) (a : \mathbb{S}) : \mathbb{T} = \{k \mapsto f(k) : k \in a\}.$$

The function *map* creates a table from another by mapping each key-value pair in a table to another by applying the specified function to the value while keeping the keys the same:

$$\text{map } (f : \mathbb{V} \rightarrow \mathbb{V}) (a : \mathbb{T}) : \mathbb{T} = \{k \mapsto f(v) : (k \mapsto v) \in a\}.$$

The function *filter* selects the key-value pairs in a table that satisfy a given function:

$$\text{filter } (f : \mathbb{K} \times \mathbb{V} \rightarrow \mathbb{B}) (a : \mathbb{T}) : \mathbb{T} = \{(k \mapsto v) \in a \mid f(k, v)\}.$$

The function *intersection* takes the intersection of two tables to generate another table. To handle the case for when the key is already present in the table, the function takes a *combining* function f of type $\mathbb{V} \times \mathbb{V} \rightarrow \mathbb{V}$ as an argument. The combining function f is applied to the two values with the same key to generate a new value. We specify intersection as

$$\begin{aligned} \text{intersection } (f : \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{V}) (a : \mathbb{T}) (b : \mathbb{T}) : \mathbb{T} \\ = \{k \mapsto f(\text{find } a k, \text{find } b k) : k \in (\text{domain}(a) \cap \text{domain}(b))\}. \end{aligned}$$

The function *difference* subtracts one table from another by throwing away all entries in the first table whose key appears in the second.

$$\begin{aligned} \text{difference } (a : \mathbb{T}) (b : \mathbb{T}) : \mathbb{T} \\ = \{(k \mapsto v) \in a \mid k \notin \text{domain}(b)\}. \end{aligned}$$

The function *union* unions the key value pairs in two tables into a single table. As with *intersection*, the function takes a combining function to determine the ultimate value of a key that appears in both tables. We specify *union* in terms of the *intersection* and *difference* functions.

$$\begin{aligned} \text{union } (f : \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{V}) (a : \mathbb{T}) (b : \mathbb{T}) : \mathbb{T} \\ = (\text{intersection } f a b) \cup (\text{difference } a b) \cup (\text{difference } b a) \end{aligned}$$

The function *find* returns the value associated with the key k . As it may not find the key in the table, its result may be bottom (\perp).

$$\text{find } (a : \mathbb{T}) (k : \mathbb{K}) : \mathbb{B} = \begin{cases} v & \text{if } (k \mapsto v) \in a \\ \perp & \text{otherwise} \end{cases}$$

Given a table, the function *delete* deletes a key-value pair for a specified key from the table:

$$\text{delete } (a : \mathbb{T}) (k : \mathbb{K}) = \{(k' \mapsto v') \in a \mid k \neq k'\}.$$

The function *insert* inserts a key-value pair into a given table. It can be thought as a singleton version of *union* and specified as such:

$$\begin{aligned} \text{insert } (f : \mathbb{V} * \mathbb{V} \rightarrow \mathbb{V}) (a : \mathbb{T}) (k : \mathbb{K}, v : \mathbb{V}) : \mathbb{T} \\ = \text{union } f a (\text{singleton } (k, v)). \end{aligned}$$

The function *restrict* restricts the domain of the table to a given set:

$$\text{restrict } (a : \mathbb{T}) (b : \text{set}) : \mathbb{T} = \{k \mapsto v \in a \mid k \in b\}.$$

It is similar to *intersection*, and in fact we have the equivalence:

$$\text{intersection first } a b = \text{restrict } a (\text{domain } b)$$

where $\text{first}(x, y) = x$. It can also be viewed as a bulk version of *find* since it finds all the keys of b that appear in a .

The function *subtract* deletes from a table the entries that belong a specified set:

$$\begin{aligned} \text{subtract } (a : \mathbb{T}) (b : \text{set}) : \mathbb{T} \\ = \{(k \mapsto v) \in a \mid k \notin b\}. \end{aligned}$$

It is similar to *difference*, and in fact we can define

$$\text{difference } a b = \text{subtract } a (\text{domain } b)$$

It can also be viewed as a bulk version of *delete* since it deletes all the keys of b from a .

In addition to these functions, we can also provide a *collect* function that takes a sequence a of key-value pairs and produces a table that maps every key in a to all the values associated with it in a , gathering all the values with the same key together in a sequence. Such a function can be implemented in several ways. For example, we can use the *collect* function that operate on sequences (Chapter 16) as discussed previously and then use *tabulate* to make a table out of this sequence. We can also implement it more directly using *reduce* as follows.

Algorithm 39.3 (*collect* on Tables).

$$\begin{aligned} \text{collect } a &= \text{Sequence.reduce} \\ &\quad (\text{Table.union Sequence.append}) \\ &\quad \{\} \\ &\quad \langle \{k \mapsto \langle v \rangle\} : (k, v) \in a \rangle \end{aligned}$$

Syntax 39.4 (Tables (continued)). We will also use the following shorthands:

$$\begin{aligned} a[k] &\equiv \text{find } A k \\ \{k \mapsto f(x) : (k \mapsto x) \in a\} &\equiv \text{map } f a \\ \{k \mapsto f(x) : k \in a\} &\equiv \text{tabulate } f a \\ \{(k \mapsto v) \in a \mid p(k, v)\} &\equiv \text{filter } p a \\ a \setminus m &\equiv \text{subtract } a m \\ a \cup b &\equiv \text{union second } a b \end{aligned}$$

where $\text{second}(a, b) = b$.

Example 39.1. Define tables a and b and set S as follows.

$$\begin{aligned} a &= \{ 'a' \mapsto 4, 'b' \mapsto 11, 'c' \mapsto 2 \} \\ b &= \{ 'b' \mapsto 3, 'd' \mapsto 5 \} \\ c &= \{3, 5, 7\}. \end{aligned}$$

The examples below show some functions, also using our syntax.

<i>find</i> :	$a['b']$	$= 11$
<i>filter</i> :	$\{k \mapsto x \in a \mid x < 7\}$	$= \{'a' \mapsto 4, 'c' \mapsto 2\}$
<i>map</i> :	$\{k \mapsto 3 \times v : k \mapsto v \in b\}$	$= \{'b' \mapsto 9, 'd' \mapsto 15\}$
<i>tabulate</i> :	$\{k \mapsto k^2 : k \in c\}$	$= \{3 \mapsto 9, 5 \mapsto 25, 9 \mapsto 81\}$
<i>union</i> :	$a \cup b$	$= \{'a' \mapsto 4, 'b' \mapsto 3, 'c' \mapsto 2, 'd' \mapsto 5\}$
<i>union</i> :	$union + (a, b)$	$= \{'a' \mapsto 4, 'b' \mapsto 14, 'c' \mapsto 2, 'd' \mapsto 5\}$
<i>subtract</i> :	$a \setminus \{'b', 'd', 'e'\}$	$= \{'a' \mapsto 4, 'c' \mapsto 2\}$

2 Cost Specification for Tables

The costs of the table functions are very similar to those for sets. As with sets there is a symmetry between the three functions *restrict*, *union*, and *subtract*, and the three functions *find*, *insert*, and *delete*, respectively, where the prior three are effectively “parallel” versions of the earlier three.

Cost Specification 39.5 (Tables).

Operation	Work	Span
<i>size a</i>	1	1
<i>singleton</i> (k, v)	1	1
<i>domain a</i>	$ a $	$\lg a $
<i>filter p a</i>	$\sum_{(k \mapsto v) \in a} W(p(k, v))$	$\lg a + \max_{(k \mapsto v) \in a} S(f(k, v))$
<i>map f a</i>	$\sum_{(k \mapsto v) \in a} W(f(v))$	$\lg a + \max_{(k \mapsto v) \in a} S(f(v))$
<i>find a k</i>	$\lg a $	$\lg a $
<i>delete a k</i>	“	“
<i>insert f a</i> (k, v)	“	“
<i>intersection f a b</i>	$m \lg(1 + \frac{n}{m})$	$\lg(n + m)$
<i>difference a b</i>	“	“
<i>union f a b</i>	“	“
<i>restrict a c</i>	“	“
<i>subtract a c</i>	“	“

where $n = \max(|a|, |b|)$ and $m = \min(|a|, |b|)$ or $n = \max(|a|, |c|)$ and $m = \min(|a|, |c|)$ as applicable. For *insert*, *union* and *intersection*, we assume that $W(f(\cdot, \cdot)) = S(f(\cdot, \cdot)) = O(1)$.

Remark. Abstract data types that support mappings of some form are referred to by various names including mappings, maps, tables, dictionaries, and associative arrays. They are perhaps the most studied of any data type. Most programming languages have some form of mappings either built in (e.g. dictionaries in Python, Perl, and Ruby), or have libraries to support them (e.g. map in the C++ STL library and the Java collections framework).

Remark. Tables are similar to sets: they extend sets so that each key now carries a value. Their cost specification and implementations are also similar.

Chapter 40

Ordering and Augmentation

Ordered sets and tables assume the keys belong to a total ordering and support operations based on this ordering. This chapter also describes augmented tables.

The set and table interfaces described so far do not include any operations that use the ordering of the elements or keys. This allows the interfaces to be defined on types that don't have a natural ordering, which makes the interfaces well-suited for an implementation based on hash tables. In many applications, however, it is useful to order the keys and use various ordering operations. For example in a database one might want to find all the customers who spent between 50 and 100 dollars, all emails in the week of August 22, or the last stock transaction before noon on October 11th.

For these purposes we can extend the operations on sets and tables with some additional operations that take advantage of ordering. ADT 40.1 defines the operations supported by ordered sets, which simply extend the operations on sets. The operations on *ordered tables* are completely analogous: they extend the operations on tables in a similar way. Note that *split* and *join* are essentially the same as the operations we defined for binary search trees (Chapter 34).

1 Ordered Sets Interface

Data Type 40.1 (Ordered Sets). For a totally ordered universe of elements $(\mathbb{U}, <)$ (e.g. the integers or strings), the *Ordered Set* abstract data type is a type \mathbb{S} representing the powerset of \mathbb{U} (i.e., all subsets of \mathbb{U}). It supports the following functions.

All functions from the set ADT (ADT 38.1) plus the following.

<i>first</i>	$: \mathbb{S} \rightarrow (\mathbb{U} \cup \{\perp\})$
<i>first</i> (A)	$= \min [A]$
<i>last</i>	$: \mathbb{S} \rightarrow (\mathbb{U} \cup \{\perp\})$
<i>last</i> (A)	$= \max [A]$
<i>previous</i>	$: (\mathbb{S} \times \mathbb{U}) \rightarrow (\mathbb{U} \cup \{\perp\})$
<i>previous</i> (A, k)	$= \max \{k' \in [A] \mid k' < k\}$
<i>next</i>	$: (\mathbb{S} \times \mathbb{U}) \rightarrow (\mathbb{U} \cup \{\perp\})$
<i>next</i> (A, k)	$= \min \{k' \in [A] \mid k' > k\}$
<i>split</i>	$: (\mathbb{S} \times \mathbb{U}) \rightarrow \mathbb{S} \times \mathbb{B} \times \mathbb{S}$
<i>split</i> (A, k)	$= \left(\{k' \in [A] \mid k' < k\}, k \overset{?}{\in} S, \{k' \in [A] \mid k' > k\} \right)$
<i>join</i>	$: (\mathbb{S} \times \mathbb{S}) \rightarrow \mathbb{S}$
<i>join</i> (A_1, A_2)	$= [A_1] \cup [A_2]$ assuming $(\max [A_1]) < (\min [A_2])$
<i>getRange</i>	$: \mathbb{S} \rightarrow \mathbb{U} \times \mathbb{U} \rightarrow \mathbb{S}$
<i>getRange</i> ($A, (k_1, k_2)$)	$= \{k \in [A] \mid k_1 \leq k \leq k_2\}$
<i>rank</i>	$: (\mathbb{S} \times \mathbb{U}) \rightarrow \mathbb{N}$
<i>rank</i> (A, k)	$= \{k' \in [A] \mid k' < k\} $
<i>select</i>	$: (\mathbb{S} \times \mathbb{N}) \rightarrow (\mathbb{U} \cup \{\perp\})$
<i>select</i> (A, i)	$= k \in [A]$ such that $\text{rank}(A, k) = i$ or \perp if there is no such k
<i>splitRank</i>	$: (\mathbb{S} \times \mathbb{N}) \rightarrow \mathbb{S} \times \mathbb{S}$
<i>splitRank</i> (A, i)	$= (\{k \in [A] \mid k < \text{select}(A, i)\},$ $\{k \in [A] \mid k \geq \text{select}(A, i)\})$

where \mathbb{N} is the natural numbers (non-negative integers) and $\mathbb{B} = \{\text{true}, \text{false}\}$. For A of type \mathbb{S} , $[|A|]$ denotes the mathematical set of keys from A . We assume max or min of the empty set returns the special element \perp .

Example 40.1. Consider the following sequence ordered lexicographically:

$$A = \{\text{'artie'}, \text{'burt'}, \text{'finn'}, \text{'mike'}, \text{'rachel'}, \text{'sam'}, \text{'tina'}\}$$

- $\text{first } A \rightarrow \text{'artie'}$.
- $\text{next } (A, \text{'quinn'}) \rightarrow \text{'rachel'}$.
- $\text{next } (A, \text{'mike'}) \rightarrow \text{'rachel'}$.
- $\text{getRange } A (\text{'burt'}, \text{'mike'}) \rightarrow \{\text{'burt'}, \text{'finn'}, \text{'mike'}\}$.
- $\text{rank } (A, \text{'rachel'}) \rightarrow 4$.
- $\text{rank } (A, \text{'quinn'}) \rightarrow 4$.
- $\text{select } (A, 5) \rightarrow \text{'sam'}$.
- $\text{splitRank } (A, 3) \rightarrow (\{\text{'artie'}, \text{'burt'}, \text{'finn'}\},$
 $\{\text{'mike'}, \text{'rachel'}, \text{'sam'}, \text{'tina'}\})$

2 Cost specification: Ordered Sets

We can implement ordered sets and tables using binary search trees. Implementing *first* is straightforward since it only requires traversing the tree down the left branches until a left branch is empty. Similarly *last* need only to traverse right branches.

Exercise 40.1. Describe how to implement *previous* and *next* using the other ordered set functions.

To implement *split* and *join*, we can use the same operations as supplied by binary search trees. The *getRange* operation can easily be implemented with two calls to *split*. To implement efficiently *rank*, *select* and *splitRank*, we can augment the underlying binary search tree implementation with sizes as described in (Chapter 37).

Cost Specification 40.2 (Tree-based ordered sets and tables). The cost for the ordered set and ordered table functions is the same as for tree-based sets (Cost Specification 38.4) and tables (Cost Specification 39.5) for the operations supported by sets and tables. The work and span for all the operations in ADT 40.1 is $O(\lg n)$, where n is the size of the input set or table, or in the case of *join* it is the sum of the sizes of the two inputs.

3 Interface: Augmented Ordered Tables

An interesting extension to ordered tables (and perhaps tables more generally) is to augment the table with a reducer function. We shall see some applications of such augmented tables but let's first describe the interface and its cost specification.

Definition 40.3 (Reducer-Augmented Ordered Table ADT). For a specified reducer, i.e., an associative function on values $f : \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{V}$ and identity element I_f , a reducer-augmented ordered table supports all operations on ordered tables and the following operation

$$\begin{aligned} \text{reduceVal} &: \mathbb{T} \rightarrow \mathbb{V} \\ \text{reduceVal } A &= \text{Table.reduce } f \ I_f \ A \end{aligned}$$

The *reduceVal A* function just returns the sum of all values in A using the associative operator f that is part of the data type. It might seem redundant to support this function since it can be implemented by the existing *reduce* function. But, by augmenting the table with a reducer, we can do reductions much more efficiently. In particular, by using augmented binary search trees, we can implement *reduceVal* in $O(1)$ work and span.

Example 40.2 (Analyzing Profits at **TRAMLAW**). Let's say that based on your expertise in algorithms you are hired as a consultant by the giant retailer **TRAMLAW**. Tramlaw sells over 10 billion items per year across its 8000+ stores. As with all major retailers, it keeps track of every sale it makes and analyzes these sales regularly to make business and marketing decisions. The sale record consists of a timestamp when the sale was made, the amount of the sale and various other information associated with the sale.

Tramlaw wants to be able to quickly determine the total amount of sales within any range of time, e.g. between 5pm and 10pm last Friday, or during the whole month of September, or during the halftime break of the last Steeler's football game. It uses this sort of information, for example, to decide on staffing levels or marketing strategy. It needs to maintain the database so that it can be updated quickly, including not just adding new sales to the end, but merging in information from all its stores, and possibly modifying old data (e.g. if some item is returned or a record is found to be in error).

How would you do this? Well after thinking a few minutes you remember ordered tables with reduced values from 210. You realize that if you keep the sales information keyed based on timestamps, and maintain the sum of sales amounts as the reduced values then all the operations required are cheap. In particular the function f is simply addition. Now the following will restrict the sum in any range:

reduceVal (getRange (T, t_1, t_2))

This will take $O(\lg n)$ work, which is much cheaper than $O(n)$. Now let's say Tramlaw wanted to do a somewhat more complicated query where they want to know the total sales between 5 and 7 pm on every day over the past year. You could do this by applying the query above once for each day. These can all be done in parallel and summed in parallel. The total work will be $365 \times O(\lg n)$, which is still much cheaper than looking at all data over the past year.

Example 40.3 (A Jig with QADSAN). Now in your next consulting job **QADSAN** hires you to more efficiently support queries on their stock exchange data. For each stock they keep track of the time and amount of every trade. This data is updated as new trades are made. As with Tramlaw, tables might also need to be unioned since they might come from different sources : e.g. the Tokyo stock exchange and the New York stock exchange. Qasdan wants to efficiently support queries that return the maximum price of a trade during any time range (t_1, t_2) .

You tell them that they can use an ordered table with reduced values using max as the combining function f . The query will be exactly the same as with your consulting jig with Tramlaw, *getRange* followed by *reduceVal*, and it will similarly run in $O(\lg n)$ work.

Chapter 41

Example: Indexing and Searching

As an application of the sets and tables ADTs, we consider the problem of indexing a set of documents to provide fast and efficient search functions on documents.

Suppose that you are given a collection of documents where each document has a unique identifier assumed to be a string and a contents, which is again a string and you want to support a range of functions including

- **word search:** find the documents that contain a given word,
- **logical-and search:** find the documents that contain a given word and another,
- **logical-or search:** find the documents that contain a given word or another,
- **logical-and-not search:** find the documents that contain a given word but not another.

This interface roughly corresponds to that offered by search engines on the web, e.g., those from Google and Microsoft. When searching the web, we can think of the url of a page on the web as its identifier and its contents, as the text (source) of the page. When your search term is two words such as "parallel algorithms", the term is treated as a logical-and search. This is the common search from but search-engines allow you to specify other kinds of queries described above (typically in a separate interface).

Example 41.1. As a simple document collection, consider the following posts made by your friends yesterday.

```


$$T = \langle ('jack', 'chess is fun'),
          ('mary', 'I had fun in dance club today'),
          ('nick', 'food at the cafeteria sucks'),
          ('josefa', 'rock climbing was a blast'),
          ('peter', 'I had fun at the party, food was great') \rangle$$


```

where the identifiers are the names, and the contents is the tweet.

On this set of documents, searching for 'fun' would return

{'jack', 'mary', 'peter'}.

Searching for 'club' would return {'mary'}.

We can solve the search problem by employing a brute-force algorithm that traverses the document collection to answer a search query. Such an algorithm would require at least linear work in the number of the documents, which is unacceptable for large collections, especially when interactive searches are desirable. Since in this problem, we are only interested in querying a static or unchanging collection of documents, we can *stage* the algorithm: first, we organize the documents into an *index* and then we use the index to answer search queries. Since, we build the index only once, we can afford to perform significant work to do so. Based on this observation, we can adopt the following ADT for indexing and searching our document collection.

Data Type 41.1 (Document Index).

```

type word      = string
type id        = string
type contents  = string
type docs      = string
type index     = string
makeIndex      : (id × contents sequence) → index
find           : index → word → docs
queryAnd       : (docs × docs) → docs
queryOr        : (docs × docs) → docs
queryAndNot   : (docs × docs) → docs
size           : docs → N
toSeq          : docs → id sequence

```

Example 41.2. For our collection of tweets in the previous example, we can use *makeIndex* to make an index of these tweets and define a function to find a word in this index as follows.

$fw : word \rightarrow docs = find (makeIndex T)$

We build the index and then partially apply *find* on the index. This way, we have staged the computation so that the index is built only once; the subsequent searches will use the index.

For example, the code,

```
toSeq (queryAnd ((fw 'fun'), queryOr ((fw 'food'), (fw 'chess'))))
```

returns all the documents (tweets) that contain 'fun' and either 'food' or 'chess', which are {'jack', 'peter'}. The code,

```
size (queryAndNot ((fw 'fun'), (fw 'chess')))
```

returns the number of documents that contain 'fun' and not 'chess', which is 2.

We can implement this interface using sets and tables. The *makeIndex* function can be implemented as follows.

Algorithm 41.2 (Make Index).

```
makeIndex docs =
let
  tagWords(i, d) = ⟨ (w, i) : w ∈ tokens(d) ⟩
  pairs = flatten ⟨ tagWords(i, d) | (i, d) ∈ docs ⟩
  words = Table.collect pairs
in
  ⟨ w ↦ Set.fromSeq d | (w ↦ d) ∈ words ⟩
end
```

The function *tokens*(*d*) : *string* → *string sequence* takes a string and splits it up into a sequence of words.

The *tagWords* function takes a document as a pair consisting of the document identifier and contents, breaks the document into tokens (words) and tags each token with the identifier returning a sequence of these pairs. Using this function, we construct a sequence of word-identifier pairs. We then use *Table.collect* to construct a table that maps each word to a sequence of identifiers. Finally, for each table entry, we convert the sequence to a set so that we can perform set functions on them.

Example 41.3. Here is an example of how *makeIndex* works. We start by tagging the words with their document identifier, using *tagWords*.

```
tagWords ('jack', 'chess is fun')
```

returns

```
⟨ ('chess', 'jack'), ('is', 'jack'), ('fun', 'jack') ⟩
```

To build the index, we apply *tagWords* to all documents, and flatten the result to a single sequence. Using *Table.collect*, we then collect the entries by word creating a sequence of

matching documents. In our example, the resulting table has the form:

```
words = {('a' ↦ ('melissa')),  
         ('at' ↦ ('nick','peter')),  
         ...  
         ('fun' ↦ ('jack','mary','peter')),  
         ...}
```

Finally, for each word the sequences of document identifiers is converted to a set.

The rest of the interface can be implemented as follows:

Algorithm 41.3 (Index Functions).

```
find T v = Table.find T v  
queryAnd A B = Set.intersection A B  
queryOr A B = Set.union A B  
queryAndNot A B = Set.difference A B  
size A = Set.size A  
toSeq A = Set.toSeq A
```

Costs. Assuming that all tokens have a length upper bounded by a constant, the cost of *makeIndex* is dominated by the collect, which is basically a sort. The work is therefore $O(n \log n)$ and the span is $O(\log^2 n)$.

For an index I with n words, the cost of *find* $I w$ is $O(\log n)$ work and span since it just involves a find in the table. The cost of *queryAnd*, *queryOr* and *queryAndNot* are the same as for intersection, union and difference on tables—i.e., $O(m \log(1 + \frac{n}{m}))$ work and $O(\log n + \log m)$ span for inputs of size n and m , $m < n$.

Part IX

Graphs

Chapter 42

Graphs and their Representation

This chapter describes graphs and the techniques for representing them.

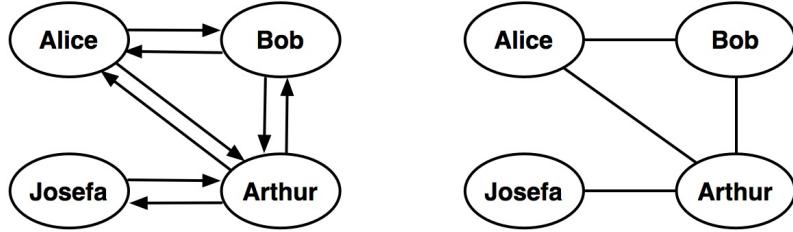
1 Graphs and Relations

Graphs (sometimes referred to as networks) are one of the most important abstractions in computer science. They are typically used to represent relationships between things from the most abstract to the most concrete, e.g., mathematical objects, people, events, and molecules. Here, what we mean by a “relationship” is essentially anything that we can represent abstractly by the mathematical notion of a relation. Recall that `relation` is defined as a subset of the Cartesian product of two sets.

Example 42.1 (Friends). We can represent the friendship relation between a set of people P as a subset of the Cartesian product of the people, i.e., $G \subset P \times P$. If $P = \{Alice, Arthur, Bob, Josepha\}$ then a relation might be:

$$\{(Alice, Bob), (Alice, Arthur), (Bob, Alice), (Bob, Arthur), (Arthur, Josefa), (Arthur, Bob), (Arthur, Alice), (Josefa, Arthur)\}$$

This relation can then be represented as a directed graph where each arc denotes a member of the relation or as an undirected graph where directed edge denotes a pair of the members of the relation of the form (a, b) and (b, a) .



Background on Graphs. Chapter 6 presents a brief review of the graph terminology that we use in this book. The rest of this chapter and other chapters on graphs assumes familiarity with the basic graph terminology reviewed therein.

2 Applications of Graphs

Graphs are used in computer science to model many different kinds of data and phenomena. This section briefly mentions some of the many applications of graphs.

Social Network Graphs. In social network graphs, vertices are people and edges represent relationships among the people, such as who knows whom, who communicates with whom, who influences whom or others in social organizations. An example is the twitter graph of who follows whom. These can be used to determine how information flows, how topics become hot, how communities develop, etc.

Transportation Networks. In road networks, vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many widely used map applications to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.

Utility Graphs. In utility graphs, vertices are junctions and edges are conduits between junctions. Examples include the power grid carrying electricity throughout the world, the internet with junctions being routers, and the water supply network with the conduits being physical pipes. Analyzing properties of these graphs is very important in understanding the reliability of such utilities under failure or attack, or in minimizing the costs to build infrastructure that matches required demands.

Document-Link Graphs. In document-link graphs, vertices are a set of documents, and edges are links between documents. The best example is the link graph of the web, where

each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites.

Graphs in Compilers. Graphs are used extensively in compilers. Vertices can represent variables, instructions, or blocks of code, and the edges represent relationships among them. Often one of the first steps of a compiler is to turn the written syntax for the program into a graph, which is then manipulated. Such graphs can be used for type inference, for so called data flow analysis, register allocation and many other purposes.

Robot Motion Planning. Vertices represent the states a robot can be in and the edges the possible transitions between the states. This requires approximating continuous motion as a sequence of discrete steps. Such graph plans are used, for example, in planning paths for autonomous vehicles.

Neural Networks and Deep Learning. Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 10^{11} neurons and close to 10^{15} synapses. Neural networks are also used for learning a variety of relationships from large data sets.

Protein-Protein Interactions Graphs. Vertices represent proteins and edges represent interactions between them; such interactions usually correspond to biological functions of proteins. These graphs can be used, for example, to study molecular pathways—chains of molecular interactions in a cellular process. Humans have over 120K proteins with millions of interactions among them.

Finite-Element Meshes. Vertices are cells in space, and edges represent neighboring cells. In engineering many simulations of physical systems, such as the flow of air over a car or airplane wing, the spread of earthquakes through the ground, or the structural vibrations of a building, involve partitioning space into discrete elements (cells), and modeling the interaction of neighboring elements as a graph.

Graphs in Quantum Field Theory. Vertices represent states of a quantum system and the edges the transitions between them. The graphs can be used to analyze path integrals and summing these up generates a quantum amplitude.

Semantic Networks. Vertices represent words or concepts and edges represent the relationships among the words or concepts. These have been used in various models of how

humans organize their knowledge, and how machines might simulate such an organization.

Graphs in Epidemiology. Vertices represent individuals and directed edges the transfer of an infectious disease from one individual to another. Analyzing such graphs has become an important component in understanding and controlling the spread of diseases.

Constraint Graphs. Vertices are items and edges represent constraints among them. For example the GSM network for cell phones consists of a collection of overlapping cells. Any pair of cells that overlap must operate at different frequencies. These constraints can be modeled as a graph where the cells are vertices and edges are placed between cells that overlap.

Dependence Graphs. Vertices are tasks or jobs that need to be done, and edges are constraints specifying what tasks needs to be done before other tasks. The edges represent dependences or precedences among items. Such graphs are often used in large projects in laying out what components rely on other components and used to minimize the total time or cost to completion while abiding by the dependences. These graphs should be acyclic.

3 Graphs Representations

We can represent graphs in many different ways. To choose an efficient and fast representation for a graph, it is important to know the kinds of operations that are needed by the algorithms that we expect to use on that graph. Common operations a graph $G = (V, E)$ include the following.

- (1) Map a function over the vertices $v \in V$.
- (2) Map a function over the edges $(u, v) \in E$.
- (3) Map a function over the (in or out) neighbors of a vertex $v \in V$.
- (4) Return the degree of a vertex $v \in V$.
- (5) Determine if the edge (u, v) is in E .
- (6) Insert or delete an isolated vertex.
- (7) Insert or delete an edge.

Different representations do better on some operations and worse on others. Cost can also depend on the *density* of the graph, i.e. the relationship of the number of vertices and number of edges.

To enable high-level, mathematical reasoning about algorithms, we represent graphs by using the abstract data types such as [sequences](#), [sets](#), and [tables](#). This approach enables specifying the algorithms at a high level and then selecting the lowest cost implementation for each algorithm.

Assumptions.

- In the rest of the chapter, we focus on directed graphs. To represent undirected graphs, we can simply keep each edge in both directions. In some cases, it suffices to keep an edge in just one direction.
- For the following discussion, consider a graph $G = (V, E)$ with n vertices and m edges.
- Throughout we assume that we only delete isolated vertices. If a vertex is incident on edges, then this means that we first have to delete the edges before deleting the vertex.

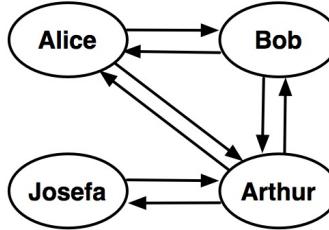
3.1 Edge Sets

Perhaps the simplest representation of a graph is based on its definition. Assuming we have a universe of possible vertices \mathcal{V} (e.g., the integers, or character strings), we can represent directed graphs in that universe as:

$$G = (\mathcal{V} \text{ set}, (\mathcal{V} \times \mathcal{V}) \text{ set}).$$

The $(\mathcal{V} \text{ set})$ is the set of vertices and the $((\mathcal{V} \times \mathcal{V}) \text{ set})$ is the set of directed edges. The sets could be represented with lists, arrays, trees, or hash tables.

Example 42.2. Using the edge-set representation, the directed graph



can be represented as:

$$\begin{aligned}
 \mathcal{V} &= \text{string} \\
 V &= \{\text{Alice, Arthur, Bob, Josefa}\} : \mathcal{V} \text{ set} \\
 E &= \{(\text{Alice, Bob}), (\text{Alice, Arthur}), (\text{Josefa, Arthur}), (\text{Bob, Arthur}), \\
 &\quad (\text{Arthur, Josefa}), (\text{Arthur, Bob}), (\text{Arthur, Alice}), (\text{Bob, Alice})\} \\
 &: (\mathcal{V} \times \mathcal{V}) \text{ set}
 \end{aligned}$$

Consider the [tree-based cost specification](#) for sets. Using edge sets for a graph with m edges, we can determine if a directed edge (u, v) is in the graph with $O(\lg m) = O(\lg n)$ work using a *find*, and *insert* or *delete* an edge (u, v) in the same work.

Although edge sets are efficient for finding, inserting, or deleting an edge, they are not efficient if we want to identify the neighbors of a vertex v . For example, finding the set of out edges of v requires filtering the edges whose first element matches v :

$$\{(x, y) \in E \mid v = x\}.$$

For m edges this requires $\Theta(m)$ work and $\Theta(\lg n)$ span, which is not work efficient.

Exercise 42.1. Prove that for a graph with n vertices and m edges, $O(\lg m) = O(\lg n)$.

Solution. For any graph, we have $m \leq n^2$ and therefore $O(\lg m) = O(\lg n)$.

Cost Specification 42.1 (Edge Sets for Graphs). For a graph represented as $G = (\mathcal{V} \text{ set}, (\mathcal{V} \times \mathcal{V}) \text{ set})$ and assuming a tree-based cost model for sets, we have the following costs for common graph operations.

	Work	Span
Map a function over all vertices $v \in V$	$\Theta(n)$	$\Theta(\lg n)$
Map a function over all edges $(u, v) \in E$	$\Theta(m)$	$\Theta(\lg n)$
Map a function over neighbors of a vertex	$\Theta(m)$	$\Theta(\lg n)$
Find the degree of a vertex	$\Theta(m)$	$\Theta(\lg n)$
Is edge $(u, v) \in E$	$\Theta(\lg n)$	$\Theta(\lg n)$
Insert or delete a vertex	$\Theta(\lg n)$	$\Theta(\lg n)$
Insert or delete an edge	$\Theta(\lg n)$	$\Theta(\lg n)$

This assumes the function being mapped has constant work and span. For vertex deletion, we assume that the vertex is isolated (has no incident edges).

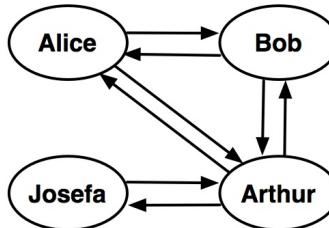
Exercise 42.2. What is the cost of deleting a vertex with out-degree d ?

3.2 Adjacency Tables

Definition 42.2 (Adjacency Table Representation). The *adjacency-table* representation of a graph consists of a table mapping every vertex to the set of its out-neighbors and can be defined as

$$G = (\mathcal{V} \times (\mathcal{V} \text{ set})) \text{ table}.$$

Example 42.3. Using the adjacency-table representation, the directed graph



can be represented as:

$$\{ \quad \quad \quad (42.1)$$

$$\text{Alice} \mapsto \{\text{Arthur, Bob}\}, \quad (42.2)$$

$$\text{Bob} \mapsto \{\text{Alice, Arthur}\}, \quad (42.3)$$

$$\text{Arthur} \mapsto \{\text{Alice, Josefa}\}, \quad (42.4)$$

$$\text{Josefa} \mapsto \{\text{Arthur}\} \quad (42.5)$$

$$\} \quad (42.6)$$

The adjacency-table representation supports efficient access to the out neighbors of a vertex by using a table lookup. Assuming the tree-based cost model for tables, this requires $\Theta(\lg n)$ work and span.

We can check if a directed edge (u, v) is in the graph by first obtaining the adjacency set for u , and then using a *find* operation to determine if v is in the set of neighbors. Using a tree-based cost model, this requires $\Theta(\lg n)$ work and span.

Inserting an edge, or deleting an edge requires $\Theta(\lg n)$ work and span. The cost of finding, inserting or deleting an edge is therefore the same as with edge sets.

Note that after we find the out-neighbor set of a vertex, we can apply a constant work function over the neighbors in $\Theta(d_G(v))$ work and $\Theta(\lg d_G(v))$ span.

Cost Specification 42.3 (Adjacency Tables). For a graph represented as $G = (\mathcal{V} \times (\mathcal{V} \text{ set})) \text{ table}$ and assuming a tree-based cost model for sets and tables, we have that:

Operation	Work	Span
Map a function over all vertices $v \in V$	$\Theta(n)$	$\Theta(\lg n)$
Map a function over all edges $(u, v) \in E$	$\Theta(m)$	$\Theta(\lg n)$
Map a function over neighbors of a vertex	$\Theta(\lg n + d_g(v))$	$\Theta(\lg n)$
Find the degree of a vertex	$\Theta(\lg n)$	$\Theta(\lg n)$
Is edge $(u, v) \in E$	$\Theta(\lg n)$	$\Theta(\lg n)$
Insert or delete a vertex	$\Theta(\lg n)$	$\Theta(\lg n)$
Insert or delete an edge	$\Theta(\lg n)$	$\Theta(\lg n)$

This assumes the function being mapped uses constant work and span.

Note. The adjacency-table representation is more efficient than the edge-set only for operations that involve operating locally on individual vertices and their out-edges.

3.3 Adjacency Sequences

Definition 42.4 (Adjacency Sequences for Enumerable Graphs). For enumerable graphs $G = (V, E)$, where $V = \{0 \dots (n - 1)\}$, we can use sequences to improve the efficiency

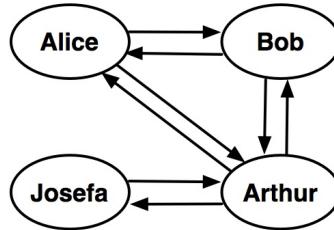
of the adjacency table representation. Sequences can be used for both the outer table and inner set. The type of a graph in this representation is thus

$$G = (\text{int seq}) \text{ seq.}$$

Here the length of the outer sequence is n , and the length of each inner sequences equals the degree of the corresponding vertex.

This representation allows for fast random access, requiring only $\Theta(1)$ work to access the i^{th} vertex.

Example 42.4. We can relabel the directed graph



by assigning the labels 0, 1, 2, 3 to Alice, Arthur, Bob, Josefa respectively. We can represent the resulting enumerable graph with the following adjacency sequence:

$$\langle 1, 2 \rangle, \quad (42.8)$$

$$\langle 0, 2, 3 \rangle, \quad (42.9)$$

$$\langle 0,1 \rangle, \quad (42.10)$$

$$\langle 1 \rangle \quad (42.11)$$

$$\rangle. \quad (42.12)$$

Exercise 42.3. What is the cost of finding the out-neighbors of a vertex?

Solution. It is $\Theta(1)$ work and span.

Exercise 42.4. Why does the cost of mapping over all edges require $\Omega(n)$ work?

Solution. Because any algorithm that maps over all the edges must touch each and every vertex.

Cost Specification 42.5 (Adjacency Sequence). Consider a graph with vertices $V = \{0, \dots, n - 1\}$, represented as $G = (\text{int seq}) \text{ seq}$. Assuming an (persistent) array-sequence cost model, the

cost of the key graph operations are as follows.

Operation	Work	Span
Map a function over all vertices $v \in V$	$\Theta(n)$	$\Theta(1)$
Map a function over all edges $(u, v) \in E$	$\Theta(n + m)$	$\Theta(1)$
Map a function over neighbors of a vertex	$\Theta(d_g(v))$	$\Theta(1)$
Find the degree of a vertex	$\Theta(1)$	$\Theta(1)$
Is edge $(u, v) \in E$	$\Theta(d_g(u))$	$\Theta(\lg d_g(u))$
Insert or delete a vertex	$\Theta(n)$	$\Theta(1)$
Insert or delete an edge	$\Theta(n)$	$\Theta(1)$

For mapping over vertices and edges, we assume that the function being mapped has constant work and span. For vertex deletions, we assume that all edges incident on the vertex has been removed and (the vertex is isolated).

Using ephemeral array sequences can improve work efficiency of deletion operations. A vertex can be marked deleted in $O(1)$ work. An edge can likewise be found and marked deleted in $\Theta(\lg d_g(n))$ work. Insertions can also be done more efficiently by an “array doubling technique”, which yields amortized constant work per update. Recall, however, that ephemeral operations can be tricky to use in parallel algorithms.

Exercise 42.5. Give a constant-span algorithm for deleting an edge from a graph.

Hint (Finding an Element via Injection). Give a linear-work, constant-span algorithm for finding the position of an element in a sequence in constant span using sequence `inject`.

Hint (Deleting an Element). Give a linear-work, constant-span algorithm for deleting an element at a specified position in a sequence.

Adjacency List Representation. In the adjacency sequence representation, we can represent the inner sequences (the out-neighbor sequence of each vertex) by using arrays or lists. If we use lists, then the resulting representation is the same as the classic *adjacency list* representation of graphs. This is a traditional representation used in sequential algorithms. It is not well suited for parallel algorithms since traversing the adjacency list of a vertex will take span proportional to its degree.

Mixed Adjacency Sequences and Tables. It is possible to mix adjacency tables and adjacency sequences by having either the inner sets or the outer table be a sequence, but not both. Using sequences for the inner sets has the advantage that it defines an ordering over the edges of a vertex. This can be helpful in some algorithms.

3.4 Adjacency Matrices

For enumerable graphs that are dense (i.e., m is not much smaller than n^2), representing the graph as a boolean matrix can make sense. For a graph with n vertices such a matrix has n

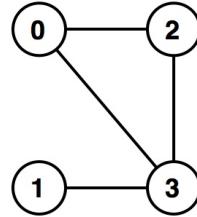
rows and n columns and contains a `true` (or 1) in location (i, j) (i -th row and j -th column) if and only if $(i, j) \in E$. Otherwise it contains a `false` (or 0). For undirected graphs the matrix is symmetric and contains `false` (or 0) along the diagonal since undirected graphs have no self edges. For directed graphs the `trues` (1s) can be in arbitrary positions.

A matrix can be represented as a sequence of sequences of booleans (or zeros and ones), for which the type of the representation is:

$$G = (\text{bool seq}) \text{ seq}$$

For a graph with n vertices the outer sequence and all the inner sequences have equal length n .

Example 42.5. The graph:



has the adjacency matrix:

$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

which can be represented as the nested sequence:

$$\langle \langle 0, 0, 1, 1 \rangle, \tag{42.13}$$

$$\langle 0, 0, 0, 1 \rangle, \tag{42.14}$$

$$\langle 1, 0, 0, 1 \rangle, \tag{42.15}$$

$$\langle 1, 1, 1, 0 \rangle \rangle. \tag{42.16}$$

Cost Specification 42.6 (Adjacency Matrix). For a graph represented as an adjacency matrix with $V = \{0, \dots, n - 1\}$ and $G = (\text{bool seq}) \text{ seq}$, and assuming the an array-sequence cost model, we have that:

Operation	Work	Span
Map a function over all vertices $v \in V$	$\Theta(n)$	$\Theta(1)$
Map a function over all edges $(u, v) \in E$	$\Theta(n^2)$	$\Theta(1)$
Map a function over neighbors of a vertex	$\Theta(n)$	$\Theta(1)$
Find the degree of a vertex	$\Theta(n)$	$\Theta(\lg n)$
Is edge $(u, v) \in E$	$\Theta(1)$	$\Theta(1)$
Insert or delete a vertex	$\Theta(n^2)$	$\Theta(1)$
Insert or delete an edge	$\Theta(n)$	$\Theta(1)$

These costs assume the function being mapped uses constant work and span.

As with other representations, using ephemeral sequences can improve the efficiency of update operations (insert/delete a vertex or an edge) to amortized constant time.

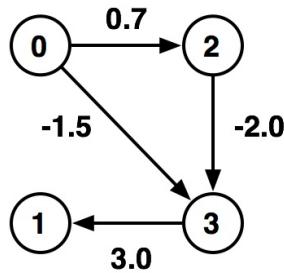
Exercise 42.6. Give a constant-span algorithm for computing the complement of a graph.

Solution. Map over the vertices a function that for each out-edge complements the boolean.

3.5 Representing Weighted Graphs

Many applications of graphs require associating values with the edges of a graph resulting in [weighted or labeled graphs](#). This section presents several techniques for representing such graphs.

Example 42.6. An example directed weighted graph.



Label Table. This chapter covered three different representations of graphs: edge sets, adjacency tables, and adjacency sequences. We can extend each of these representations to support edge-weights by representing the function from edges to weights as a separate table. This *weight table* maps each edge to its value and allows finding weight of an edge $e = (u, v)$ by using a table lookup.

Example 42.7. For the weighted graph shown [above](#) the weight table is:

$$W = \{(0, 2) \mapsto 0.7, (0, 3) \mapsto -1.5, (2, 3) \mapsto -2.0, (3, 1) \mapsto 3.0\}.$$

Weight tables work uniformly with all graph representations, and they are elegant, because they separate edge weights from the structural information. However, keeping a separate weight table creates redundancy, wasting space and possibly requiring additional work to lookup the weights. We can eliminate the redundancy by storing the weights directly with the edge.

For example, when using the edge-set representation for graphs, we can keep the weight along with the edge in the edge sets. Similarly, when using the adjacency-table representation, we can replace each set of neighbors with a set consisting of the neighbor and the weight of the edge from the vertex to the neighbor. Finally, we can extend an adjacency sequences by creating a sequence of neighbor-weight pairs for each out edge of a vertex.

Example 42.8. For the weighted graph shown [above](#) the adjacency table representation is

$$G = \{0 \mapsto \{2 \mapsto 0.7, 3 \mapsto -1.5\}, 2 \mapsto \{3 \mapsto -2.0\}, 3 \mapsto \{1 \mapsto 3.0\}\},$$

and the adjacency sequence representation is

$$G = \langle \langle (2, 0.7), (3, -1.5) \rangle, \langle \rangle, \langle (3, -2.0) \rangle, \langle (1, 3.0) \rangle \rangle.$$

Chapter 43

Graph Search

The terms *graph search* and *graph traversal* refer to a class of algorithms that systematically explore the vertices and edges of a graph. Graph-search can be used to solve many interesting problems on (directed or undirected) graphs and is indeed at the heart of many graph algorithms. In this chapter, we introduce the concept of a graph search, and develop a generic graph-search algorithm. We then consider further specializations of this generic algorithm, including the [priority-first search \(PFS\)](#) algorithms, [breadth-first search \(BFS\)](#), and [depth-first search \(DFS\)](#).

Assumption (Directed and Undirected Graphs). Because undirected graphs can be represented as directed graphs where each edge is replaced with two edges in opposite directions, we consider in this chapter directed graphs only.

1 Generic Graph Search

Definition 43.1 (Source). A graph search usually starts at a specific *source* vertex $s \in V$ or a set of vertices *sources*. Graph search then searches out from the source(s) and iteratively *visits* the unvisited neighbors of vertices that have already been visited.

Definition 43.2 (Visited Vertices). Graph search algorithms keep track of the *visited vertices*, which have already been visited, and avoid re-visiting them. We typically denote the set of visited vertices with the variable X .

Definition 43.3 (Frontier and Discovered Vertices). For a graph $G = (V, E)$ and a visited set $X \subset V$, the *frontier set* or simply the *frontier* is the set of un-visited out-neighbors of X , i.e., the set $N_G^+(X) \setminus X$. We often denote the frontier set with the variable F . We refer to each vertex in the frontier as a *discovered vertex*.

Reminder. Recall that $N_G^+(v)$ are the out-neighbors of the vertex v in the graph G , and $N_G^+(U) = \bigcup_{v \in U} N_G^+(v)$ (i.e., the union of all out-neighbors of U).

Algorithm 43.4 (Graph Search (Single Source)). The generic graph-search algorithm that starts at a single source vertex s is given below.

```

1  graphSearch( $G, s$ ) =
2    let
3      explore  $X F$  =
4        if ( $|F| = 0$ ) then  $X$ 
5        else
6          let
7            choose  $U \subseteq F$  such that  $|U| \geq 1$ 
8            visit  $U$ 
9             $X = X \cup U$ 
10            $F = N_G^+(X) \setminus X$ 
11           in explore  $X F$  end
12     in
13     explore {} { $s$ }
14   end

```

The algorithm starts by initializing the visited set of vertices X with the empty set and the frontier with the source s . It then proceeds in a number of *rounds*, each corresponding to a recursive call of *explore*. At each round, the algorithm selects a subset U of vertices in the frontier (possibly all), visits them, and updates the visited and the frontier sets. If multiple vertices are selected in U , they can usually be visited in parallel. The algorithm terminates when the frontier is empty.

Note. The generic graph search algorithm as presented only keeps track of the visited set X . Many real applications keep track of additional information.

Exercise 43.1. Does the algorithm visit all the vertices in the graph?

Solution. The algorithm does not necessarily visit all the vertices in the graph. In particular vertices for which there is no path from the source will never be visited.

Graph Search is Generic. Since the function `graphSearch` is not specific about the set of vertices to be visited next, it can be used to describe many different graph-search algorithms. In the rest of the book, we consider three methods for selecting the subset, each leading to a specific graph-search algorithm.

- Selecting all of the vertices in the frontier leads to [breadth-first search \(BFS\)](#).
- Selecting the single most recent vertex added to the frontier leads to [depth-first search \(DFS\)](#).
- Selecting the highest-priority vertex (or vertices) in the frontier, by some definition of priority, leads to [priority-first search \(PFS\)](#).

As we will see, [breadth-first search](#) is parallel because it can visit many vertices at once (the whole frontier). In contrast, depth-first-search is sequential, because it only visits one vertex at a time.

2 Reachability

It is sometimes useful to find all vertices that can be reached in a graph from a source. Graph search can solve this problem for any choice of U on each round.

Definition 43.5 (Reachability). We say that a vertex v is *reachable* from u in the graph G (directed or undirected) if there is a path from u to v in G .

Problem 43.2 (The Graph Reachability Problem). For a graph $G = (V, E)$ and a vertex $v \in V$, return all vertices $U \subseteq V$ that are reachable from v .

Theorem 43.1 (Graph Search Solves Reachability). The function `graphSearch` ($G = (V, E)$) s returns exactly the set of vertices that are reachable in G from $s \in V$, and does so in at most $|V|$ rounds, and for any selection of U on each round.

Proof. The algorithm finishes in at most $|V|$ rounds since it visits at least one vertex on each round.

It returns only reachable vertices by induction on the round number. In particular if all vertices X_i on round i are reachable from s , then all vertices added on round $i + 1$ are reachable. This is because by the definition of frontier there is a path through X_i and extending one to all $X_{i+1} \setminus X_i$.

We prove that it returns all reachable vertices by contradiction. Let's say that a vertex v is reachable from s and is not in the set R returned by `graphSearch` $G s$. Consider any simple path from s to v . This must exist since v is reachable from s . Let u be the vertex immediately before v on the path. It cannot be in R because if it were then v would have been in the frontier on the final termination round and the algorithm would not have finished. Similarly the item before u on the path could not be in R , and this repeats all the way back to the source s . This is a contradiction since s must be in R .

□

Example 43.1 (Web Crawling). An example of graph search are web crawlers that try to find all pages available on the web, or at least those reachable from some source(s). They start with the URL of some source page, probably one with lots of links. The crawler visits the source page, and adds all the URLs on the page to the frontier. It then picks some number of URLs from the frontier and visits them, possibly in parallel, adding the unvisited URLs within each to the frontier. When repeated this process is a graph search and will therefore reveal all web pages reachable from the source URL.

In practice the crawl might start with many sources instead of just one. It might also be sloppy on immediately adding visited pages to the visited set, allowing for more asynchronous parallelism at the cost of possibly visiting pages more than once.

Connected Components. When a graph is undirected then the reachability problem from s is the same as finding the [connected component](#) that contains s . If we go through all the vertices, then we can identify all the connected components in the graph. This method, however, is sequential. In later chapters we will see how to find connected components in polylogarithmic span using graph contraction.

Exercise 43.3 (Multi-Source Search). Present a multi-source version of the [single-source graph search algorithm](#).

3 Graph-Search Tree

Consider the time at which a vertex v is visited. We can “blame” the visit of v to an in-neighbor u of v that has already been visited. Because each vertex is visited at most once, we can define a “search tree” by including in the tree the visited vertices and their blamed vertices.

Definition 43.6 (Graph-Search Tree). Let $G = (V, E)$ be a graph. A *graph-search tree* for an execution of the graph search algorithm of G is a rooted tree over the visited vertices $X \subseteq V$ and the edges $E' \subseteq E$ such that every vertex $v \in X \setminus \{s\}$ has a parent u that is in X when v is visited and the $(u, v) \in E$. The source s is the root of the tree (and has no parent).

Note. The definition allows the parent of a vertex v to be any in-neighbor of v , which is in X at the time that v is visited. In many specific graph-search algorithms, there is usually a specific parent (and a specific edge) that can be “blamed” for the discovery of v .

4 Priority-First Search (PFS)

Many graph-search algorithms can be viewed as visiting vertices in some priority order. The idea is to assign a priority to all vertices in the frontier, allowing the priority of a vertex to change whenever an in-neighbor of that vertex is visited. When picking the set of vertices U to visit, we have several options:

- the highest priority vertex, breaking ties arbitrarily,
- all highest priority vertices, or
- all vertices close to being the highest priority, perhaps the top k .

This specialization of generic graph search is called *Priority-First Search* or *PFS* for short.

In this book we only consider instances of PFS that follow the first two cases above. The third case, where we select the k highest priority vertices, is sometimes called *beam search*.

Note. PFS is a relatively common form of graph search. We will use it in [Breadth-First Search](#), [Dijkstra's algorithm](#) and [Prim's algorithm](#).

Priority-first search is a greedy technique, because it greedily selects among the choices available (the vertices in the frontier) based on some “cost function” (the priorities) and never backs up. Algorithms based on PFS are hence often referred to as greedy algorithms.

Sometimes, especially in the artificial intelligence literature, PFS is called *best-first search*. In this context the search explores a set of vertices, often called nodes, representing states or possible solutions of a problem. The importance of each node can be evaluated based on some application-specific heuristic. Picking the nodes carefully can significantly reduce the number of nodes needed to reach a satisfactory node (state or solution).

Chapter 44

Breadth-First Search

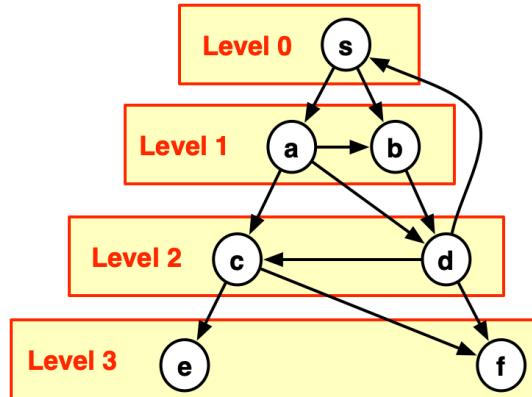
The *breadth-first search* or *BFS* algorithm is a special case of the [generic graph-search algorithm](#). The BFS algorithm visits vertices in the graph in the order of their distances from the source. In addition to solving the reachability problem, BFS can be used to find the shortest (unweighted) path from a source vertex to all other vertices, determining if a graph is bipartite, bounding the diameter of an undirected graph, partitioning graphs, and as a subroutine for finding the maximum flow in a flow network (using Ford-Fulkerson's algorithm). As with the other graph searches, BFS can be applied to both directed and undirected graphs. BFS is inherently parallel, at least if the diameter of the graph is modest.

1 BFS and Distances

Definition 44.1 (Distance of a Vertex). To understand how BFS operates, consider a graph $G = (V, E)$ and a source vertex $s \in V$. For a given source vertex s , define the *distance* of a vertex $v \in V$ from s as the shortest distance from s to v , that is the number of edges on the shortest path connecting s to v in G , denoted as $\delta_G(s, v)$.

Definition 44.2 (Breadth First Search). Breadth First Search (BFS) is a graph search that explores a given graph “outward” in all directions in increasing order of distances. More precisely, for all distances $i < j$ in the graph, a vertex at distance i is visited before a vertex at distance j .

Example 44.1. An example graph and the distances of its vertices to the source vertex 0 are illustrated below. We can imagine the vertices at each distance to form a *layer* in the graph. BFS visits the vertices on layers 0, 1, 2, and 3 in that order. For example, because f is on layer 3, we have that $\delta(s, f) = 3$. In fact there are three shortest paths of equal length: $\langle s, a, c, f \rangle$, $\langle s, a, d, f \rangle$ and $\langle s, b, d, f \rangle$.

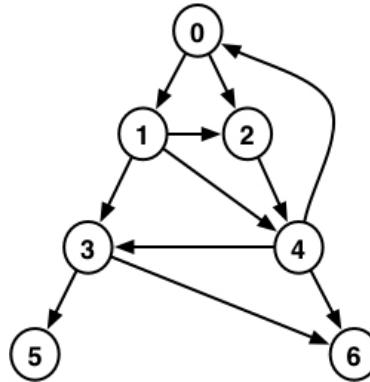


Note. BFS does not specify the order in which the vertices at a given distance should be visited: they can be visited in arbitrary order one by one, or they can all be visited at the same time in parallel. We first present a [sequential algorithm](#) for BFS that visits the vertices at a distance one by one. We then present a [parallel algorithm](#) that visits the vertices at each distance all at once.

Reminder (Representing Enumerable Graphs). Consider an enumerable graph $G = (V, E)$ where $V = \{0, 1, \dots, n - 1\}$. As discussed [earlier in adjacency sequence representations](#) we can represent enumerable graphs as an adjacency sequence—that is, a sequence of sequences, where the i^{th} inner sequence contains the out-neighbors of vertex i . This representation allows for finding the out-neighbors of a vertex in constant-work (and span).

Example 44.2. The enumerable graph below can be represented as

$$\langle \langle 1, 2 \rangle, \langle 2, 3, 4 \rangle, \langle 4 \rangle, \langle 5, 6 \rangle, \langle 3, 4, 6 \rangle, \langle \rangle, \langle \rangle \rangle.$$



2 Sequential BFS

We present a sequential BFS algorithm that visits the vertices at a distance one by one and

present an implementation that achieves linear work and span in the size of the graph (total number of vertices and edges).

Algorithm 44.3 (Sequential BFS: Reachability). The sequential breadth first search (BFS) algorithm is a specialization of [graph search](#) where a frontier vertex with the closest distance is visited in each round. Because in general there can be many vertices at the same distance, any one of the vertices can be selected, breaking ties arbitrarily. The code for the algorithm is shown below. As in [generic graph search](#), the algorithm revolves around a frontier F and a visited set X . To ensure that vertices are visited in distance order, the frontier keeps for each vertex and its distance. The algorithm returns the set of vertices X reachable from s in G , and the maximum over all vertices $v \in X$ of $\delta_G(s, v)$.

```

1  BFSReach ( $G = (V, E)$ )  $s =$ 
2  let explore  $X F i =$ 
3    if ( $|F| = 0$ ) then  $(X, i)$ 
4    else let
5       $(u, j) = \arg \min_{(v, k) \in F} (k)$ 
6       $X = X \cup \{u\}$ 
7       $F = F \setminus \{(u, j)\}$ 
8       $F = F \cup \{(v, j + 1) : v \in N_G^+(u) \mid v \notin X \wedge (v, \_) \notin F\}$ 
9      in explore  $X F j$  end
10 in explore  $\{\} \{(s, 0)\} 0$  end

```

Exercise 44.1. The computation of the new F is not quite defined in the same way as in the [generic graph search](#). Prove that the technique used here is consistent with that of the generic algorithm.

Exercise 44.2. Prove that the algorithm is correct, i.e., visits all reachable vertices from the source in the order of their distances to the source.

2.1 Cost of Sequential BFS

Sequential BFS accepts a simple and efficient implementation. To streamline the presentation and analysis, we consider here a version of the algorithm for enumerable graphs.

To start with, note that [the algorithm](#) uses the visited set X and the frontier F in a linear fashion. That is, the algorithm only uses the most version of X and F . This allows using ephemeral data structures to represent the visited set and the frontier.

Representation of the Visited Set. We can use a boolean sequence of size $|V|$ to represent the visited set. The value in the i^{th} position indicates whether the vertex i is visited or not. We initialize the sequence with all `false`'s, indicating that none of the vertices are visited. When we visit a vertex, we *update* the corresponding element to `true`. Because the visited set is used linearly, we can represent it with an ephemeral array, where *update* and *sub* (lookup) operations require constant work and span.

Representation of the Frontier. The frontier needs support several operations, including

1. checking that a vertex is not in the frontier,
2. removing the vertex with the smallest distance, and
3. adding the neighbors of a vertex into the frontier.

To check that a vertex is in the frontier, we can maintain an array of indicators, indicating whether that vertex is in the frontier or not. This is very similar to the representation of the visited set X [described above](#). Because a vertex cannot be in the visited set and in the frontier at the same time, we can merge these two representations and keep in X a value indicating, whether a vertex is visited, or in the frontier, remains to be unexplored. Intuitively, we can think of these three different values as three distinct colors, typically called, white (unexplored), in the frontier (gray), and visited (black). This “color abstraction” is due to Dijkstra.

To represent the frontier, we can use a standard priority queue data structure, which support insert and remove operations in logarithmic work and span. Because BFS visits the vertices only in increasing order of their distances, we only insert into the frontier vertices whose distances are no smaller than those in the frontier. In other words, the priorities increase monotonically as the algorithm proceeds.

Based on this observation, we can use simpler priority queue data structure, i.e., just a simple ephemeral queue. Such a data structure requires constant work and span to *push* to the tail of the queue and to *pop* from the head of the queue. In each round, we simply *pop* the vertex u at the head of the queue and visit u by marking the visited sequence. After the visit, we take all of u ’s out-neighbors and check for each if they are visited. We *push* each unvisited out-neighbor into the tail of the queue. This implementation maintains the invariant that if a vertex has distance smaller than another, then it is closer to the head of the queue.

Note. Dijkstra was well-known for his meticulous attention to use of color. During his tenure at University of Texas at Austin, he would consistently question visiting faculty about their choice of color in their presentations. For example, he would ask “why is that red and not blue?”

Analysis. For per-round costs, consider a round, i.e., recursive invocation of the function *explore*. The work includes checking whether the frontier (queue) is empty or not, which is constant work. If not empty, then we *pop* a vertex in constant work, and then mark it visited, also in constant work. We then find all the out-neighbors of the vertex, which requires constant work by using the [array-sequence based representation](#). We then check for each neighbor whether it is visited and if not, *push* it to the tail of the queue, which requires constant work per out-neighbor.

Thus the only non-constant work in a round involves the handling of out-neighbors. To bound this cost, observe that each vertex is pushed onto the queue at most once. Thus the

total number of *push* operations is bounded by the number of vertices and their work cost is $O(|V|)$. To bound the cost checking that each neighbor is visited, note that each such check corresponds exactly to one edge in the graph, and thus their number is bounded by m , and the their total work cost is $O(|E|)$.

Because creating the initial sequence keeping track of visited vertices requires $\Theta(|V|)$ work, the total work of sequential BFS is $O(|V| + |E|)$.

Exercise 44.3. Prove that the queue based implementation of sequential BFS is correct.

3 Parallel BFS

The [sequential BFS algorithm](#) visits one vertex in each round. But a moments reflection should convince us that the vertices in the same round are already in the frontier and could all be visited at the same time. In this section, we present a parallel BFS algorithm that does exactly that.

Algorithm 44.4 (Parallel BFS: Reachability). The parallel BFS algorithm is an instance of Algorithm 43.4 where **all** frontier vertices are visited in each round. The code for the algorithm is shown below. The algorithm searches the graph outward, starting at distance 0 (just the source s), and visiting the vertices at a distance on each round. It maintains the distance i , which it is currently visiting. The algorithms returns the set of vertices X reachable from s in G , and the maximum over all vertices $v \in X$ of $\delta_G(s, v)$.

```

1  BFSReach ( $G = (V, E)$ ,  $s$ ) =
2  let explore  $X$   $F$   $i$  =
3      if ( $|F| = 0$ ) then  $(X, i - 1)$ 
4      else let
5           $X = X \cup F$ 
6           $F = N_G^+(F) \setminus X$ 
7          in explore  $X$   $F$   $(i + 1)$  end
8  in explore  $\{\}$   $\{s\}$  0 end

```

Parallel BFS and Distances. The [Parallel BFS algorithm](#) maintains an important invariant on the distance of the vertices and their visit order. To see this invariant, let X_i denote the vertices visited before distance (round) i . At round i , the algorithm visits vertices at a distance i , and because the algorithm visits distances in increasing order, the vertices in X_i are exactly those with distance less than i from the source. The following lemma establishes this invariant.

Lemma 44.1 (Parallel BFS and Distances). In $\text{BFSReach } (G = (V, E)) s$, at the beginning of every invocation of *explore* (line 3), we have

$$\begin{aligned} X &= X_i &= \{v \in V \mid \delta_G(s, v) < i\}, \text{ and} \\ F &= F_i &= \{v \in V \mid \delta_G(s, v) = i\} \end{aligned}$$

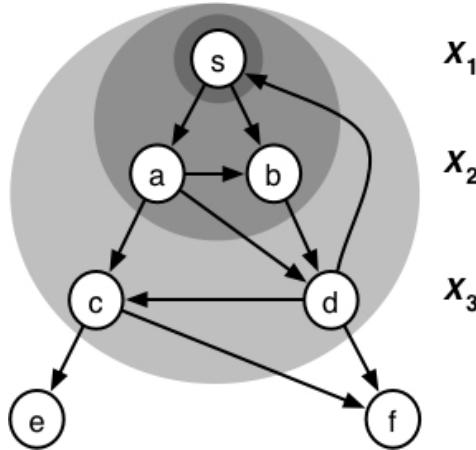
Proof. We prove the lemma by induction on the distance i .

In the base case, $i = 0$, and we have $X_0 = \{\}$ and $F_0 = \{s\}$. This is true, because no vertex has distance less than 0 from s and only s has distance 0 from s .

For the inductive step, we assume the properties are correct for i and want to show they are correct for $i + 1$. For X_{i+1} , the algorithm takes the union of all vertices at distance less than i (X_i) and all vertices at distance exactly i (F_i). So X_{i+1} is exactly the vertices at a distance less than $i + 1$.

The algorithm computes F_{i+1} , as all neighbors of F_i minus the set X_{i+1} . By induction hypothesis, all vertices F_i have distance i from s and therefore any neighbor v of F has $\delta_G(s, v) \leq i + 1$. Furthermore, any vertex at distance $i + 1$ is reachable from a vertex at distance i and therefore, the out-neighbors of F_i contain all vertices at distance $i + 1$. Removing X_{i+1} , we are therefore left with exactly those vertices at distance $i + 1$. \square

Example 44.3. The figure below illustrates the BFS visit order by using overlapping circles from smaller to larger. Initially, X_0 is empty and F_0 is the single source vertex s , as it is the only vertex that is a distance 0 from s . X_1 is all the vertices that have distance less than 1 from s (just s), and F_1 contains those vertices that are on the middle ring, a distance exactly 1 from s (a and b). The outer ring contains vertices in F_2 , which are a distance 2 from s (c and d). Notice that vertices in a frontier can share the same neighbors (e.g., a and b share d). $N_G(F)$ is defined as the *union* of neighbors of the vertices in F to avoid duplicate vertices.



Exercise 44.4. In general, from which frontiers could the vertices in $N_G(F_i)$ come when the graph is undirected? What if the graph is directed?

3.1 Cost of Parallel BFS

We analyze the cost the [BFS variant for reachability](#) from source. For the analysis, we represent the main data structures by using tree-based sets and tables. This approach applies

to graphs whose vertices accept a comparison (total-order) operation. For a graph with m edges and n vertices, we show that the algorithm requires $O(m \lg n)$ work $O(d \lg^2 n)$ span where d the largest distance d of any reachable vertex from the source.

Bounding Cost using Aggregation. When analyzing the cost of BFS, a natural method is to sum the work and span over the rounds of the algorithm, each of which visits vertices at a specific distance. In contrast with recurrence based analysis, this approach makes the cost somewhat more concrete but can be made complicated by the fact that the cost per round depends on the structure of the graph. We bound the cost by observing that BFS visits each vertex at most once, and since the algorithm only looks at a vertex's out-edges when visiting it, the algorithm also only uses every edge at most once.

Let's first analyze the cost per round. In each round, the only non-trivial work consists of the union $X \cup F$, the calculation of neighbors $N = N_G^+(F)$, and the set difference to calculate the new frontier $F = N \setminus X$. The cost of these operations depends on the number of out-edges of the vertices in the frontier. Let's use $\|F\|$ to denote the number of out-edges for a frontier plus the size of the frontier, i.e., $\|F\| = \sum_{v \in F} (1 + d_G^+(v))$. We show that the costs for each round and the total are:

	Work	Span
$X \cup F$	$O(F \lg n)$	$O(\lg n)$
$N_G^+(F)$	$O(\ F\ \lg n)$	$O(\lg^2 n)$
$N \setminus X$	$O(\ F\ \lg n)$	$O(\lg n)$
Total over d rounds	$O(m \lg n)$	$O(d \lg^2 n)$

The first and last lines fall directly out of the tree-based cost specification for the set ADT. Note that $|N| \leq \|F\|$. The second line is a bit more involved. The union of out-neighbors is implemented as

$$\begin{aligned} N_G^+(F) = & \text{Table.reduce } \text{Set.Union} \\ & \{ \} \\ & (\text{Table.restrict } G F) \end{aligned}$$

Let $G_F = \text{Table.restrict } G F$. The work to find G_F is $O(|F| \lg n)$. For the cost of the union, note that the set union results in a set whose size is no more than the sum of the sizes of the sets unioned. Furthermore recall that the cost of set union on sets of size k and l ($k \leq l$) is bounded by $O(k \log(1 + l/k)) \subset O(k + l)$. The $O(k + l)$ follows from the fact that for $x \geq 0$, $\log x < x$. The total work per level of reduce is therefore no more than $\|F\|$. Since there are $O(\lg n)$ such levels, the work is bounded by

$$\begin{aligned} W(\text{reduce union } \{ \} G_F) \\ = O\left(\lg |G_F| \sum_{(v \mapsto N(v)) \in G_F} (1 + |N(v)|)\right) \\ = O(\lg n \cdot \|F\|). \end{aligned}$$

The span is bounded by

$$S(\text{reduce union } \{\} G_F) = O(\lg^2 n),$$

because each union has span $O(\lg n)$ and the reduction tree is bounded by $\lg n$ depth.

Focusing on a single round, we can see that the cost per vertex and edge visited in that round is $O(\lg n)$. Furthermore we know that every reachable vertex only appears in the frontier exactly once. Therefore, all the out-edges of a reachable vertex are also processed only once. Thus the cost per edge W_e and per vertex W_v over the algorithm is the same as the cost per round. We thus conclude that $W_v = W_e = O(\lg n)$. Since the total work is $W = W_v n + W_e m$ (recall that $n = |V|$ and $m = |E|$), we thus conclude that

$$\begin{aligned} W_{BFS}(n, m, d) &= O(n \lg n + m \lg n) \\ &= O(m \lg n), \text{ and} \\ S_{BFS}(n, m, d) &= O(d \lg^2 n). \end{aligned}$$

We drop the $n \lg n$ term in the work since for BFS we cannot reach any more vertices than there are edges.

Notice that span depends on d . In the worst case $d \in \Omega(n)$ and BFS is sequential. Many real-world graphs, however, have low diameter; for such graphs BFS has good parallelism.

4 Shortest Paths and Shortest-Path Trees

Thus far we have used BFS for reachability. But as established in Lemma 44.1, the [Parallel BFS algorithm](#) effectively calculates the distance to each of the reachable vertices. It is therefore relatively straightforward to extend the algorithm to keep track of the distances, as illustrated by the algorithm below.

Algorithm 44.5 (Unweighted Shortest Paths). The following variant of the [Parallel BFS algorithm](#) takes a graph and a source and returns a table mapping every reachable vertex v to $\delta_G(s, v)$. The algorithm uses the table X both to keep track of the visited vertices and for each of these vertices to keep its distance from s . When visiting the vertices in the frontier the algorithm adds them to X with distance i .

```

1  BFSDistance( $G = (V, E), s$ ) =
2  let explore  $X$   $F$   $i$  =
3      if ( $|F| = 0$ ) then  $(X, i - 1)$ 
4      else let
5           $X = X \cup \{v \mapsto i : v \in F\}$ 
6           $F = N_G^+(F) \setminus \text{domain}(X)$ 
7          in explore  $X$   $F$   $(i + 1)$  end
8      in explore  $\{\} \{s\}$  0 end

```

Shortest Paths. In addition to computing distances, we can also use BFS to compute a shortest path from the source to each reachable vertex. The basic idea is to use the [graph-search tree](#) for BFS. Recall that the search tree is over the visited vertices, and the parent of each vertex v is an in-neighbor of v that was already visited when v is visited.

Theorem 44.2 (BFS Tree Gives Shortest Paths). Given a graph-search tree for BFS, the path from any vertex v to the source s in the tree when reversed is a shortest path from s to v in G .

Proof. The proof is by induction on the distance of the vertex. For the distance-0 vertex, we only have the source and the lemma holds trivially. Suppose that the lemma holds for all distances up to but excluding distance i . Consider a vertex v visited at distance i . We know that all in-neighbors already visited when v is visited are at distance $i - 1$, because otherwise, by Lemma 44.1, the algorithm would visit v earlier. Note that if the in-neighbors are at a larger distance, then, by Lemma 44.1, they would not have been visited. By induction, the property holds for all in-neighbors of v . Because a shortest path to v goes through one of the in-neighbors, the property also holds for v .

□

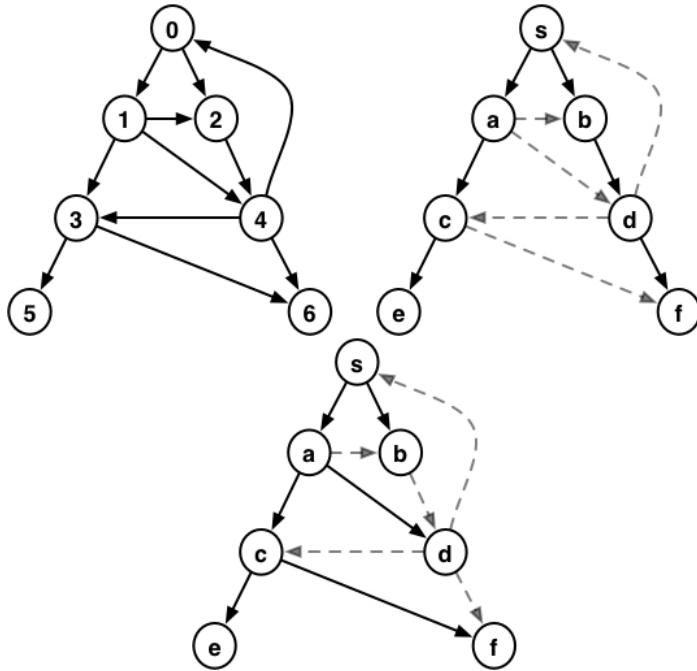
Shortest-path Tree. Because of this property we refer to a graph-search tree generated by BFS as a (unweighted) shortest-path tree. Given such a shortest-path tree, we can then compute the shortest path to a particular vertex v by following the path in the tree from s to v .

We can generate a shortest-path tree in two ways. One way is to change BFS slightly to keep track of parents. We do this later in Algorithm 44.6. Another is to post process the result of Algorithm 44.5. In particular each vertex can pick as its parent any one of its in-neighbors that has a distance that is closer than its own distance. If X is a mapping of vertices to their distances from s in G (as returned by $BFSDistance$), then the shortest path tree can be computed as:

$$T = \{v \mapsto u : (u, v) \in E \mid X[u] < X[v]\}.$$

In T each visited vertex, other than s , will map to its parent in the shortest path tree. For a given v there can be multiple edges that satisfy $X(u) < X(v)$, and the construction of the table, and corresponding tree, will pick one of them.

Example 44.4. A directed graph (left) and two possible BFS trees with distances from s (middle and right). Non-tree edges, which are edges of the graph that are not on a shortest paths are indicated by dashed lines.



Single-Source Shortest Paths. The problem of finding the shortest path from a source to all other vertices (unreachable vertices are assumed to have infinite distance), is called the single-source shortest path problem. Here we are measuring the length of a path as the number of edges along the path. In later chapters we consider shortest paths where each edge has a weight (length), and the shortest paths are the ones for which the sums of the weights are minimized. Breadth-first search does not work for the weighted case.

BFS Tree with Sequences. Assume performing parallel BFS on an enumerable graph, which can be [represented using sequences](#) also see [the reminder above](#).

Notice that in BFS the visited set is used linearly and thus we can represent it with an ephemeral data structure. Because the vertices are enumerable, we can use a sequence, specifically an Chapter 20ephemeral or a single-threaded array sequence. To mark the visited vertices we use *inject*, which requires constant work per vertex. For marking vertices, we can use either a Boolean flag to indicate its status, or the label of the parent vertex (if any). The latter representation can help up construct a BFS tree.

Algorithm 44.6 (BFS Tree with Sequences). We present a variant of BFS that computes the shortest-path tree from the source by using sequence-based data structures. The algorithm returns a shortest-paths tree as a sequence mapping each vertex (position) to its parent in the shortest-paths tree. The source points to itself and unvisited vertices contain `None`.

The algorithm is similar to the [generic graph search algorithm](#) with one important difference: the visited sequence X contains the parents for both the visited and the frontier

vertices, instead of just for the visited vertices. This approach streamlines the computation of the shortest path tree. The frontier F is represented using a sequence of vertices.

Each round starts by visiting the vertices in the frontier. Next, it computes the sequence N of the unvisited neighbors of vertices in the frontier, each paired with $\text{Some}(v)$ for the vertex $v \in F$ it came from. It does this by calculating for each vertex $v \in F$ its unvisited neighbors (the function $f(v)$) and then flattening the results.

The algorithm uses *inject* to write each of the frontier vertices into X . Any given vertex in N can appear multiple times with different source vertices, but *inject* will select one parent for each, breaking ties arbitrarily. The iteration completes by having each vertex in N check if it was selected and keeping those that were selected as the next frontier. This removes any duplicates from N so every new frontier vertex will appear exactly once.

```

1  BFSTree ( $G, s$ ) =
2  let
3    explore( $X, F$ ) =
4    if ( $|F| = 0$ ) then  $X$ 
5    else let
6      (* Visit  $F$  *)
7       $f(v) = \langle (u, \text{Some } v) : u \in G[v] \mid X[u] = \text{None} \rangle$ 
8       $N = \text{Seq}.\text{flatten} \langle f(v) : v \in F \rangle$ 
9       $X = \text{Seq}.\text{inject } X \ N$ 
10      $F = \langle u : (u, v) \in N \mid X[u] = v \rangle$ 
11     in explore( $X, F$ ) end
12      $X = \langle \text{None} : v \in \langle 0, \dots, |G| - 1 \rangle \rangle$ 
13      $X = \text{Seq}.\text{update } X (s, \text{Some } s)$ 
14   in explore( $X, \{s\}$ ) end

```

4.1 Cost with Sequences

We analyze the work and span of [the variant of BFS that returns a shortest-paths tree](#).

Because the algorithm uses visited-set linearly, i.e., once the visited vertices X is updated, it is never used again. We can use ephemeral sequences or single-threaded sequences to represent X . In such a representation all *update* and *inject* operations take constant work per position updated. Initialization requires $\Theta(n)$ work and constant span.

Cost Analysis. The cost of the algorithm is dominated by *flatten*, *inject*, and by the construction of the next frontier F in Lines 8, 9, and 10. The following table gives costs for each round, and then the total across rounds including also the $\Theta(n)$ initialization cost.

Line	Work	Span
8 (<i>f(v) and flatten</i>)	$O(\ F_i\)$	$O(\lg n)$
9 (<i>inject</i>)	$O(\ F_i\)$	$O(1)$
10 (<i>the new F</i>)	$O(\ F_i\)$	$O(\lg n)$
Total over d rounds	$O(n + m)$	$O(d \lg n)$

As before, d is the length of the longest reachable path from the root.

Chapter 45

Depth-First Search

The Depth-First Search (DFS) algorithm is a special case of the generic graph-search algorithm, where each round visits the frontier vertex that is most recently discovered. DFS has many applications. For example, it can be used to solve the reachability problem, to find cycles in a graph, to topologically sort a directed acyclic graph, to find strongly connected components of a directed graph, and to test whether a graph is biconnected or not. Unlike BFS, DFS is inherently sequential, because it only visits one vertex at a time.

1 DFS Reachability

Recall that in graph search, in each round, we can choose any (non-empty) subset of the vertices in the frontier and visit them. We say a vertex is *discovered* when it is added to the frontier. The DFS algorithm is a specific graph search that in each round picks the most recently discovered vertex in the frontier and visits it.

Because DFS only visits one vertex in each round, the unvisited (out) neighbors of that vertex will be discovered in the round, and will be the most recent. To break ties among the out-neighbors, we assume the out-edges of each vertex are ordered. We allow any order of the out-neighbors.

Algorithm 45.1 (DFS with a Stack). We can implement DFS by representing the frontier with a stack. In each round we pop the top (first) vertex off the stack and visit it. We then push the out-neighbors discovered in the round onto the stack in their out-neighbor order. In the pseudo-code below we use a sequence for the stack, the n th F 0 removes the first element in the frontier and the sequence append (@) pushes the out-neighbors onto the frontier. As with other graph searches X is the set of visited vertices, and the algorithm returns all vertices reachable from s in G .

```

1  DFSStack ( $G, s$ ) =
2  let
3  explore  $X F$  =
4  if ( $|F| = 0$ ) then  $X$ 
5  else
6  let
7   $u = \text{nth } F 0$ 
8  visit  $u$ 
9   $X = X \cup \{u\}$ 
10  $F = \langle v : v \in N_G^+(u) \mid v \notin X \rangle @ F[1 \dots |F| - 1]$ 
11 in explore  $X F$  end
12 in explore  $\{\} \langle s \rangle$  end

```

Algorithm 45.2 (DFS, Recursively). We can implement DFS more simply via recursion. As usual, X is the set of visited vertices, which the algorithm returns. In the recursive DFS , immediately after a vertex v is visited (i.e., added to the set X), the algorithm iterates over all its neighbors trying to visit them. Therefore the next vertex the algorithm will visit is the first unvisited out-neighbor of v , if any.

```

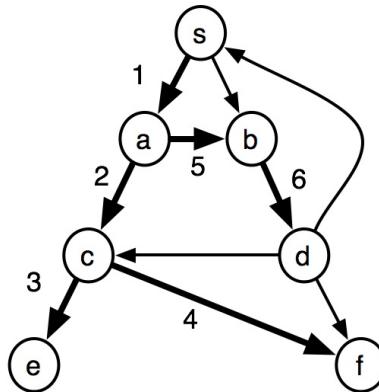
DFSReach ( $G, s$ ) =
let DFS ( $X, v$ ) =
?
  if  $v \in X$  then  $X$ 
  else
    let  $X' = X \cup \{v\}$  in
      iterate DFS  $X' N_G^+(v)$ 
    end
  in DFS ( $\{\}, s$ ) end

```

Note. The notation $v \in X$ stands for checking that v is in set X . It returns `true` if v is in X and `false` otherwise.

Note. Unlike the generic search algorithm and BFS, the algorithm does not maintain the frontier explicitly. Instead, the frontier is implicitly represented in the recursion—i.e., when we return from DFS , its caller will continue to iterate over vertices in the frontier.

Example 45.1. Below is an example of DFS on a graph where edges are ordered counter-clockwise, starting from the left. Each row of the table corresponds to the arguments to one call to $DFS(X, v)$ in the order they are called. In the last four rows the vertices have already been visited, so the call returns immediately without revisiting the vertices since they appear in X .



v	X	visit
s	$\{\}$	✓
a	$\{s\}$	✓
c	$\{s, a\}$	✓
e	$\{s, a, c\}$	✓
f	$\{s, a, c, e\}$	✓
b	$\{s, a, c, e, f\}$	✓
d	$\{s, a, c, e, f, b\}$	✓
c	$\{s, a, c, e, f, b, d\}$	7
f	$\{s, a, c, e, f, b, d\}$	7
s	$\{s, a, c, e, f, b, d\}$	7
b	$\{s, a, c, e, f, b, d\}$	7

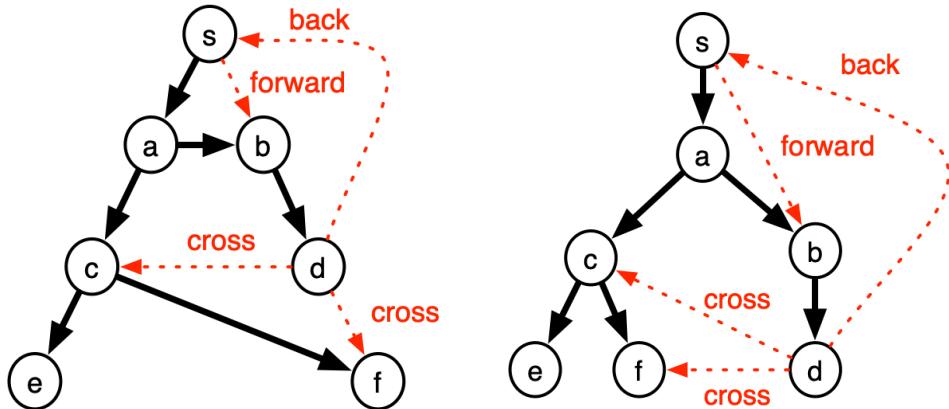
2 DFS Trees

Definition 45.3 (Tree and Non-Tree Edges in DFS). Consider performing DFS on a graph $G = (V, E)$ with some source s . We call an edge $(u, v) \in E$ a *tree edge* if the vertex v is discovered during the visit to u . The tree edges T define the *DFS tree* rooted at s .

We classify the non-tree edges further into back edges, forward edges, and cross edges.

- A non-tree edge (u, v) is a *back edge* if v is an ancestor of u in the DFS tree.
- A non-tree edge (u, v) is a *forward edge* if v is a descendant of u in the DFS tree.
- A non-tree edge (u, v) is a *cross edge* if v is neither an ancestor nor a descendant of u in the DFS tree.

Example 45.2. Tree edges (black), and non-tree edges (red, dashed) illustrated with the original graph and drawn as a tree.



Exercise 45.1. Show that undirected graphs cannot have any cross edges.

Proof. We prove by contradiction. Let's say $\{u, v\}$ is a cross edge, and without loss of generality let's say u is visited first. Then as we visit the neighbors of u we will visit v , making the edge a tree edge—a contradiction. \square

DFS Visit, Revisit, and Finish. When applying DFS, it turns out it is helpful to distinguish the following points in the algorithm for each vertex v :

- *visit*: when v is (first) visited
- *revisit*: when v is re-visited, or encountered but not visited again, and
- *finish*: when all vertices reachable from v have been visited.

Essentially all applications of DFS do their interesting work during their initial visit, the revisits, and the finish. For this reason it is useful to define [a generic version of DFS](#), which makes it possible to perform a desired computation at a visit, revisit, and finish.

Note. In the recursive formulation of DFS finish of v corresponds to when the *iterate* completes and the algorithm returns from $DFS(., v)$.

Algorithm 45.4 (Generic DFS). The generic DFS algorithm takes a graph, a source, and an application-specific state or structure Σ and threads Σ through the computation along with the visited set X . The three application-specific functions *visit*, *finish*, and *revisit* manipulate the state. The algorithm returns the final state with the visited vertices.

```

1  DFSGeneric G (( $\Sigma$ ,  $X$ ),  $v$ ) =
2    if ( $v \in X$ ) then (revisit  $\Sigma$   $v$ ,  $X$ )
3    else let
4       $\Sigma' = \text{visit } \Sigma v$ 
5       $X' = X \cup \{v\}$ 
6      ( $\Sigma'', X''$ ) = iterate (DFSGeneric G) ( $\Sigma', X'$ ) ( $N_G^+(v)$ )
7      in (finish  $\Sigma'' v$ ,  $X''$ ) end

```

Note. The generic algorithm starts takes in as an argument a visited set X (instead of starting with the empty set). This allows performing DFS multiple times over the same graph possibly with different sources without revisiting previously visited vertices.

Algorithm 45.5 (Generic DFSAll). The *DFSAll* algorithm runs DFS over all vertices of a graph, visiting each once.

```

DFSAll ( $G = (V, E)$ )  $\Sigma$  =
  iterate (DFSGeneric G) ( $\Sigma, \{\}$ )  $V$ 

```

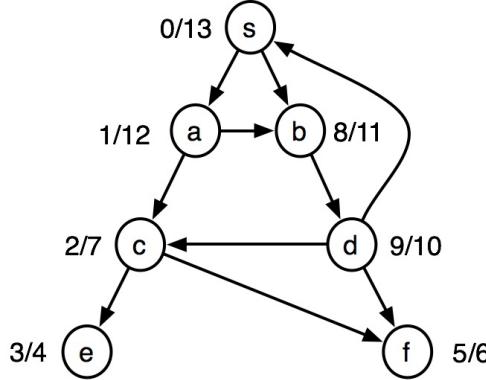
Exercise 45.2. Why would we need the function *DFSAll*?

Solution. DFS from a source visits only the vertices reachable from that source. If there are vertices that are not reachable from the source, they will not be visited. We can use *DFSAll* to make sure that all vertices in the graph are visited.

3 DFS Numbers

Definition 45.6 (DFS Numbers). In DFS, we can assign two timestamps to each vertex: the time at which a vertex runs *visit* (when first visited) and the time it runs *finish* (when done visiting its out-neighbors). These are respectively called the *visit time* and the *finish time*. We refer to the timestamps as *DFS numbers*.

Example 45.3. A graph and its DFS numbers are shown below; t_1/t_2 denotes the timestamps showing when the vertex is visited and finished respectively. Note that vertex a gets a finish time of 12 since it does not finish until all vertices reachable from its two out neighbors, c and b , have been fully explored. Vertices d , e and f have no un-visited out neighbors, and hence their finish time is one more than their visit time.



Generating DFS Numbers. We can generate the visit and finish times by using the [generic DFS algorithm](#) with the following definitions:

$$\begin{aligned}
 \Sigma_0 &= (0, \emptyset, \emptyset) \\
 \text{visit } (i, \mathcal{T}_V, \mathcal{T}_F) v &= (i+1, \mathcal{T}_V \cup \{v \mapsto i\}, \mathcal{T}_F) \\
 \text{finish } (i, \mathcal{T}_V, \mathcal{T}_F) v &= (i+1, \mathcal{T}_V, \mathcal{T}_F \cup \{v \mapsto i\}) \\
 \text{revisit } \Sigma v &= \Sigma.
 \end{aligned}$$

The state Σ is a triple representing the *time* i , a table \mathcal{T}_V mapping vertices to their visit times, and a table \mathcal{T}_F mapping vertices to their finish time. The time starts at 0 and the tables start empty. Each *visit* tags v in \mathcal{T}_V with the current time, and each *finish* tags v in \mathcal{T}_F with the current time. They both increment the time. The *revisit* function does nothing. When called as

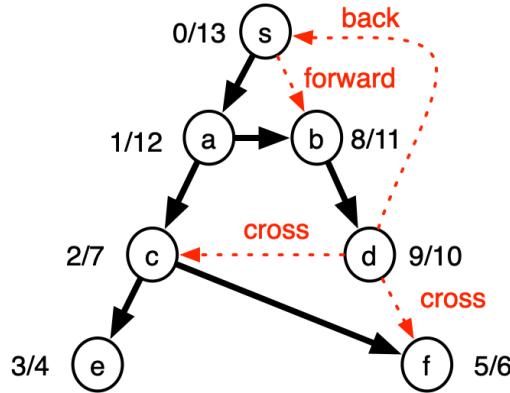
DFSAll $G \Sigma_0$

the resulting tables will include visit and finish times for all vertices in G .

Lemma 45.1 (DFS Numbers). The visit and finish times can be used to determine which edges are cross edges, forward edges, and back edges. In particular for all non-tree edges $(u, v) \in E \setminus T$ we have

$$(u, v) = \begin{cases} \text{cross} & \text{if } \mathcal{T}_V(u) > \mathcal{T}_V(v) \text{ and } \mathcal{T}_F(u) > \mathcal{T}_F(v) \\ \text{forward} & \text{if } \mathcal{T}_V(u) < \mathcal{T}_V(v) \text{ and } \mathcal{T}_F(u) > \mathcal{T}_F(v) \\ \text{back} & \text{if } \mathcal{T}_V(u) > \mathcal{T}_V(v) \text{ and } \mathcal{T}_F(u) < \mathcal{T}_F(v) \end{cases}$$

Example 45.4. An example DFS from source s , its tree and non-tree edges, and its DFS numbers are illustrated below.



Exercise 45.3. Prove the [DFS Numbers Lemma](#).

Exercise 45.4. Consider a DFS on a graph and define for each vertex v , its *exploration interval* as the time interval $[\mathcal{T}_V(v), \mathcal{T}_F(v)]$. Restate the [DFS Numbers Lemma](#) in terms of exploration intervals and their overlaps.

4 Cost of DFS

We analyze the cost of DFS, specifically the *DFSAll* function, which applies DFS to all vertices in the graph. Because *DFSAll* is generic, the cost depends on the state Σ and the function *visit*, *revisit*, and *finish*. We therefore present a bound in terms of the number of calls to these functions.

Lemma 45.2 (Bound on DFS calls). For a graph $G = (V, E)$ with m edges, and n vertices, and for any state Σ , the call *DFSAll* $G \Sigma$ makes $n + m$ calls to *DFS*, n calls to *visit* and *finish*, and m calls to *revisit*.

Proof. Because each vertex is added to X , when it is first visited, every vertex will only be visited (and finished) once. The *revisit* function gets called every time *DFS* is called but the vertex is not visited, i.e., for a total of $m + n - n = m$ times. Because each vertex is visited exactly once, every out-edge is also traversed once, invoking a call to *DFS*. There are also n calls to *DFS* directly from *DFSAll* $G \Sigma$. Total number of calls to *DFS* is therefore $n + m$. \square

Cost of Graph Operations. Each call to *DFS* performs one find operation to check $v \in X$. Every time the algorithm visits a vertex, it performs one insertion of v into X ($X \cup \{v\}$). In total, the algorithm therefore performs at most n insertion and $m + n$ find operations. To iterate over the out-neighbors of a vertex, the algorithm also have to lookup the neighbors of each vertex once.

For a tree-based implementation of sets and an adjacency table representation of graphs all operations take $O(\lg n)$ work.

For enumerable graphs with $V = \{0, \dots, n - 1\}$ we can implement DFS more efficiently using an ephemeral array sequences for X , and adjacency sequences for the graphs giving $O(1)$ work per operation.

Algorithm 45.7 (DFS with Array Sequences). The following version of DFS uses adjacency sequences for representing the graph and array sequences for keeping track of the visited vertices.

```

 $DFSSeq : (int\ seq) \ seq \rightarrow \alpha \times \ int \rightarrow \ bool\ seq$ 
1   $DFSseq\ G(\Sigma, s) =$ 
2   $\text{let } DFS((\Sigma, X), v) =$ 
3   $\quad \text{if } X[v] \text{ then } (revisit\ \Sigma\ v, X)$ 
4   $\quad \text{else let}$ 
5   $\quad \quad \Sigma' = visit\ \Sigma\ v$ 
6   $\quad \quad X' = Seq.update\ X(v, \text{true})$ 
7   $\quad \quad (\Sigma'', X'') = iterate\ DFS(\Sigma', X')(G[v])$ 
8   $\quad \text{in } (finish\ \Sigma''\ v, X'') \text{ end}$ 
10  $\quad \text{in } DFS((\Sigma, X_0), s) \text{ end}$ 

```

Cost Specification 45.8 (DFS). Consider the DFS algorithm on a graph with m out edges, and n vertices.

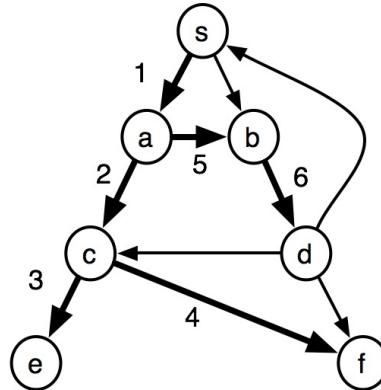
For adjacency table representation of graphs, and the tree-based cost specification for sets and tables, DFS runs in $O((m + n) \lg n)$ work and span (assuming $visit$, $finish$, and $revisit$, all take $O(\log n)$ work).

For enumerable graphs using adjacency sequences (array based), and ephemeral sequences for X , DFS runs in $O(m + n)$ work and span (assuming $visit$, $finish$, and $revisit$, all take constant work).

4.1 Parallel DFS

Difficulty of Parallel DFS. At first sight, we might think that DFS can be parallelized by searching the out edges in parallel. This will work if the searches on each out edge never “meet up”, which is the case for a tree. However, in general when portions of the graph reachable through the outgoing edges are shared, visiting them in parallel creates complications. This is because it is important that each vertex is only visited once, and in DFS it is also important that the earlier out-edge visits any shared vertices, not the later one. This makes it very difficult to parallelize.

Example 45.5. Consider the example graph drawn below.



If we search the out-edges of s in parallel, we would visit the vertices a, c and e in parallel with b, d and f . This is not the DFS order because in the DFS order b and d will be visited after a . In fact, it is BFS ordering. Furthermore the two parallel searches would have to synchronize to avoid visiting vertices, such as b , twice.

Remark (DFS is P-Complete). Depth-first search is known to be P-complete, a class of computations that can be done in polynomial work but are widely believed not to admit a polylogarithmic span algorithm. A detailed discussion of this topic is beyond the scope of this book, but it provides evidence that DFS is unlikely to be highly parallel.

5 Cycle Detection

Definition 45.9 (Cycle-Detection Problem). The *cycle-detection problem* requires determining whether there is a cycle in a graph.

We can use DFS to detect cycles in a directed graph. To see this, consider the different kinds of non-tree edges in a DFS. Forward edges don't create cycles because they go from ancestors to descendants, and thus create an alternative path between vertices but in the same direction as the tree edges. Cross edges don't create cycles because they create a unidirectional path from one set of vertices to another. Back edges do create cycles by making an ancestor reachable from a descendant (which is reachable from a descendant).

Theorem 45.3 (Back Edges Imply Cycles). A directed graph $G = (V, E)$ has a cycle if and only if a *DFSAll* on G has a back edge.

Proof. To prove the theorem we consider both directions of the implication.

If a graph has a back edge (u, v) , then there is a path from v to u in the DFS tree, followed by the edge (u, v) , forming thus a cycle.

Consider a graph with a cycle and perform DFS on the graph. Consider the first vertex v at which the DFS enters the cycle and let u be the vertex before v in the cycle, i.e., the edge (u, v) is on the cycle. The DFS will next visit all the vertices in the cycle, because each vertex is reachable from v , and revisit v through the edge (u, v) . Because v is an ancestor of u in the DFS tree, (u, v) is a back edge. \square

By the [theorem above](#) we can check for cycles in a directed graph by generating the DFS numbers and checking that there is no back edge. Alternatively, we can check for cycles directly with DFS by maintaining a set of ancestors and checking that there is no edge that leads to an ancestor.

Algorithm 45.10 (Directed Cycle Detection). Define the state Σ is a pair containing a boolean *flag* indicating whether a cycle has been found and a set *ancestors* containing all ancestors of the current vertex. Define the initial state Σ_0 and the functions *visit*, *finish*, and *revisit* as follows.

$$\begin{aligned}\Sigma_0 &= (\text{false}, \emptyset) \\ \text{visit } (\text{flag}, \text{ancestors}) \, v &= (\text{flag}, \text{ancestors} \cup \{v\}) \\ \text{finish } (\text{flag}, \text{ancestors}) \, v &= (\text{flag}, \text{ancestors} \setminus \{v\}) \\ \text{revisit } (\text{flag}, \text{ancestors}) \, v &= (\text{flag} \vee (v \in \text{ancestors}), \text{ancestors})\end{aligned}$$

The function *cycleDetect* defined as

$$\text{cycleDetect } G = \text{first } (\text{DFSAll } G \, \Sigma_0)$$

returns `true` if and only if there are any directed cycles in G .

To maintain *ancestors*, we add a vertex to the set when we first visit it and remove it when we finish it. On a revisit to a vertex v we simply check if v is in *ancestors* and if it is, then there is a cycle.

Cycle Detection in Undirected Graphs. We can try to apply [the cycle-detection algorithm](#) to undirected graphs by representing an undirected edge as two directed edges, one in each direction. Unfortunately this does not work, because it will find that every edge $\{u, v\}$ forms a cycle of length two, from u to v and back to u .

Exercise 45.5. Design a cycle finding algorithm for undirected graphs.

6 Topological Sort

Definition 45.11 (Directed Acyclic Graph (DAG)). A *directed acyclic graph*, or a *DAG*, is a directed graph with no cycles.

DAGs and Partial Orders. A partial order \leq_p over a set of elements is a binary relation that satisfies

- transitivity ($a \leq_p b$ and $b \leq_p c$ implies $a \leq_p c$), and
- antisymmetry (for distinct a and b , $a \leq_p b$ and $b \leq_p a$ cannot both be true).

A partial order allows elements to be unordered—i.e., neither $a \leq_p b$ nor $b \leq_p a$ —hence, the term “partial”.

In a graph, if we interpret $a \leq_p b$ as b is reachable from a then transitivity is true for all graphs, and antisymmetry is true if there are no cycles in the graph. If two vertices cannot reach each other, they are unordered. Reachability in a DAG therefore defines a partial order over the vertices. In the rest of this section, we associate with DAGs a partial order relation that corresponds to reachability.

Given a DAG, it is sometimes useful to put the vertices in a total order that is consistent with the partial order. That is to say that if $a \leq_p b$ in the partial ordering then a is before b in the total ordering. This is always possible, and is called a topological sort. The term “topological” comes from the fact that if we order the vertices in a line from left to right, all edges in the graph would go from left to right.

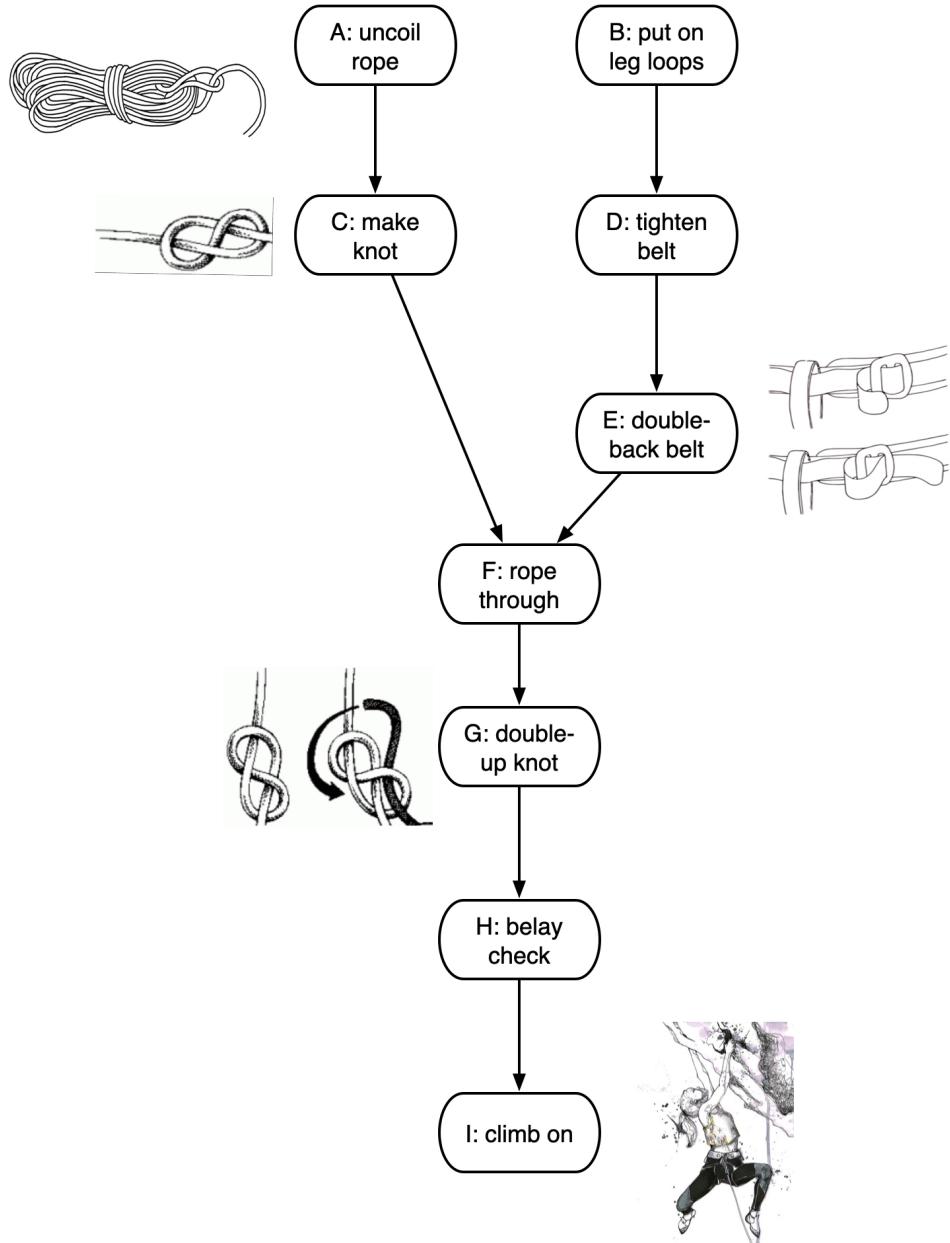
Example 45.6 (Dependency Graphs). Topological sort is often useful for *dependency graphs*, which are directed graphs where vertices represent tasks to be performed and edges represent the dependencies between them. Accomplishing nearly anything that is reasonably complex requires performing a number of tasks some of which depend on each other. For example, on a cold winter morning, we would put our socks first and then our shoes. Likewise, we would put on our pullover before our coat. If we organize these tasks as a graph, a topological sort of the graph tells us how we can dress without breaking the dependencies.

Note that dependency graphs should not have cycles, because otherwise it would be impossible to satisfy the dependencies. Dependency graphs are therefore DAGs.

Definition 45.12 (Topological Sort of a DAG). The topological sort (or topological ordering) of a DAG (V, E) is a total ordering, $v_1 < v_2 \dots < v_n$ of the vertices in V such that for any directed edge $(v_i, v_j) \in E$, $i < j$ holds. Equivalently if v_i can reach v_j then $i \leq j$.

Example 45.7 (Climbers Sort). As an example, consider what a rock climber must do before starting a climb to protect herself in case of a fall. For simplicity, we only consider the tasks of wearing a harness and tying into the rope. The example is illustrative of many situations which require a set of actions or tasks with dependencies among them.

The graph below presents a subset of the tasks that a rock climber must follow before they start climbing, along with the dependencies between the tasks. The graph is a directed graph with the vertices being the tasks, and the directed edges being the dependences between tasks. Performing each task and observing the dependencies in this graph is crucial for the safety of the climber—any mistake puts the climber as well as her belayer and other climbers into serious danger. While instructions are clear, errors in following them abound.



There are many possible topological orderings for the DAG. For example, following the tasks in alphabetical order yield a topological sort. For a climber's perspective, this is not a good order, because it has too many switches between the harness and the rope. To minimize errors, climbers prefer to put on the harness (tasks B, D, and E, in that order), prepare the rope (tasks A and B), rope through, and finally complete the knot, get her gear checked, and climb on (tasks F, G, and I, in that order).

The following property of DFS numbers allows us to use them to solve topological sorting using DFS.

Lemma 45.4 (DAG Finish Order). For *DFSAll* on a DAG G , if a vertex u is reachable from v then u will finish before v .

Proof. We consider two cases.

1. u is visited before v . In this case u must finish before v is visited otherwise there would be a path from u to v and hence a cycle.
2. v is visited before u . In this case since u is reachable from v it must be visited while searching from v and therefore finish before v finishes.

□

This lemma implies that if we order the vertices by finishing time (latest first), then all vertices reachable from a vertex v will appear after v in the ordering. This is exactly the property we require from a topological sort. We could therefore simply generate the DFS numbers and sort by decreasing finish time. We can also calculate the ordering directly as follows.

Algorithm 45.13 (Topological Sort). Define the state Σ as a sequence of vertices representing a topological sort of the vertices finished thus far. Define the initial state Σ_0 and the functions *visit*, *finish*, and *revisit* as follows.

$$\begin{aligned}\Sigma_0 &= \langle \rangle \\ \text{visit } \Sigma v &= \Sigma \\ \text{finish } \Sigma v &= \langle v \rangle @ \Sigma \\ \text{revisit } \Sigma v &= \Sigma\end{aligned}$$

The function *decreasingFinish* defined as

$$\text{decreasingFinish } G = \text{first} (\text{DFSAll } G \Sigma_0)$$

returns a topological order for all the vertices of the DAG G .

This specialization of DFS adds a vertex to the front of the sequence Σ each time a vertex finishes. This effectively ensures that vertices are inserted in decreasing order of their finish time (the last vertex to finish will be the first). The *visit* and *revisit* functions do nothing.

Exercise 45.6. Prove that for enumerable graphs, the work and span of the [topological sort](#) algorithm can be bounded by $O(|V| + |E|)$.

Hint. You will need to represent sequences in a way that ensures that all of *visit*, *revisit* and *finish* requires constant work.

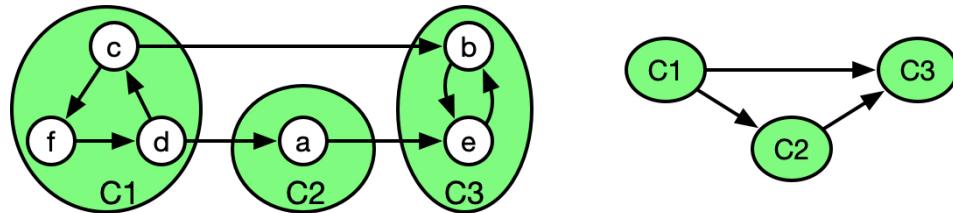
7 Strongly Connected Components (SCC)

Definition 45.14 (Strongly Connected Graph). A directed graph $G = (V, E)$ is *strongly connected* if all vertices can reach each other.

Definition 45.15 (Strongly Connected Components). For a directed graph $G = (V, E)$, a subgraph H of G is a *strongly connected component* if H is strongly connected and is *maximal*, i.e., adding more vertices and edges from G into H , breaks strong connectivity.

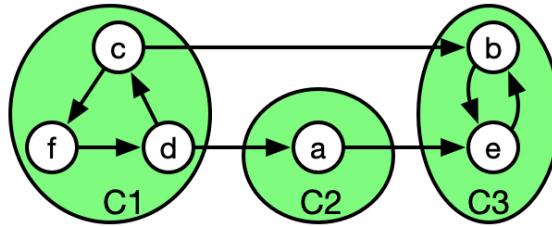
Definition 45.16 (Component DAG). Contracting each strongly connected component in a graph into a vertex and eliminating duplicate edges between components yields a *component DAG*, where each component is represented by a single vertex.

Example 45.8. The graph illustrated below on the left has three strongly connected components. The component DAG on the right shows the graph that remains after we contract each component to a single vertex.



Definition 45.17 (SCC Problem). The *Strongly Connected Components (SCC) problem* requires finding the strongly connected components of a graph and returning them in topological order.

Example 45.9. For the graph



the strongly connected components in topological order are:

$$\langle \{c, f, d\}, \{a\}, \{e, b\} \rangle$$

Intuition for an Algorithm. To develop some intuition towards an algorithm for the SCC problem, suppose that we find one representative vertex for each component and we topologically sort the representatives according to the components DAG. We know that if we run a DFS from a representative vertex u , we will find all the vertices in the connected component of u , but also possibly more, because vertices in another component can be reachable, leading to an undesirable “overflow.” Notice now that if we reach a vertex v that is not within u ’s component, then v ’s component “comes after” u ’s in the topological sort of the DAG of components, because all edges between components flow from “left to right” in the topological order.

We can prevent this “overflow” by flipping the direction of each edge in the graph—this is called transposing the graph. This does not change the reachability relation within a connected component but the edges between components now flow from “right to left” with respect to the topological sort of the component DAG. This means that if we search out of a representative of a component, we will reach the vertices in the component and the (non-component) vertices that are in earlier components.

Recall that we assumed that our representatives are topologically sorted. Imagine now starting with the first representative and performing a DFS, we know that there will be no overflow. Let’s mark these vertices visited and move on to the second representative and perform a DFS. We may know reach vertices from the first component, but this is easy to detect, because we have already marked them visited, and there will be no overflow. In this way, we can find all the connected components with a forward sweep through our representatives, while doing DFS in the transposed graph.

Our algorithm will effectively follow this intuition but with a twist: we cannot topologically sort an arbitrary graph, specifically graphs that have cycles. We will therefore consider relaxation and sort the vertices of the graph in decreasing finish time by using the function *decreasingFinish* from Algorithm 45.13. This suffices because it will correctly order vertices in different components.

Lemma 45.5 (First Visited). For a directed graph $G = (V, E)$ if $u \in V$ is the first vertex visited by a DFS in its component, then for all v reachable from u , $\mathcal{T}_F(u) > \mathcal{T}_F(v)$. Recall that $\mathcal{T}_F(x)$ is the finish time of x .

Proof. Since u is the first visited in its component, and it can reach all other vertices in its component, it will visit and finish all of them before finishing. Therefore if v is in the same component, v will finish first. If v is in a different component there are two cases:

1. v is visited before u . Since u is not reachable from v , v must have finished before u is even visited.
2. u visited before v . In this case u is an ancestor of v and therefore must finish after v finishes.

Note these two cases are effectively the same as in Lemma 45.4. \square

Algorithm 45.18 (Strongly Connected Components). The algorithm *SCC* starts by sorting the vertices of the graph in decreasing of their finish time by using the function *decreasingFinish* from Algorithm 45.13. The function *DFSReach* returns the set of all vertices reachable from v that have not already been visited as indicated by X , as well as an updated X including the vertices that it visits. The *iterate* goes over the vertices in F in order. The *if* ($|A| = 0$) skips over empty sets, which arises when the vertex v has already been visited.

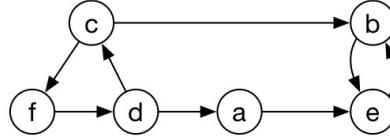
```

SCC ( $G = (V, E)$ ) =
  let
     $F = \text{decreasingFinish } G$ 
     $G^T = \text{transpose } G$ 

    SCCOne ( $(X, \text{comps}), v$ ) =
      let  $(X', \text{comp}) = \text{DFSReach } G^T (X, v)$  in
        if  $|\text{comp}| = 0$  then
           $(X', \text{comps})$ 
        else
           $(X', \langle \text{comp} \rangle @ \text{comps})$ 
        end
      in
      iterate SCCOne ( $\{\}, \langle \rangle$ )  $F$ 
    end
  end

```

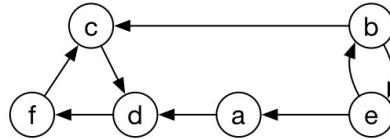
Example 45.10. Consider the *SCC* algorithm on the following graph G :



If *decreasingFinish* processes the vertices in alphabetical order, then we have that:

$$F = \langle c, f, d, a, e, b \rangle .$$

In particular when a is visited, $\langle a, e, b \rangle$ will be added and when c is visited $\langle c, f, d \rangle$ will be added. The transpose graph G^T is:



The *iterate* of DFS on G^T for each vertex in F , gives for each step:

$DFSReach G^T (\{\}, c)$	$\rightarrow \{c, d, f\}$
$DFSReach G^T (\{c, d, f\}, f)$	$\rightarrow \{\}$
$DFSReach G^T (\{c, d, f\}, d)$	$\rightarrow \{\}$
$DFSReach G^T (\{c, d, f\}, a)$	$\rightarrow \{a\}$
$DFSReach G^T (\{a, c, d, f\}, e)$	$\rightarrow \{b, e\}$
$DFSReach G^T (\{a, b, c, d, e, f\}, b)$	$\rightarrow \{\}$

All the empty sets are dropped, so the final output is:

$$\langle \{c, d, f\}, \{a\}, \{b, e\} \rangle .$$

Theorem 45.6 (SCC Correctness). Algorithm 45.18 solves the SCC problem.

Proof. Summary: Running *DFSReach* on the first vertex x of a SC component X will visit exactly the vertices in X since the other components that it can reach in G^T (can reach it in G) have already been visited (first from each is earlier in F), and no other components can be reached from x in G^T . This leaves just X , which is reachable and unvisited.

Full proof: Because of Lemma 45.5, the first vertex from each SC component in F appears before all other vertices it can reach. Thus when the *iterate* hits the first vertex x in a component X , the following conditions are true:

1. No vertex in X has been visited since we are working on the transpose of the graph, and all reachable vertices from x come after x in F .
2. All vertices in X are reachable from x in the transpose graph since transposing does not affect reachability within a component.
3. All other components that can reach X have previously been visited since their first element appears before x in F , and all other vertices of the component are reachable from the first.
4. All other components that cannot reach X in G cannot be reached from X in the transpose G^T —edges are reversed.

Therefore when running *DFSReach* on x we will visit exactly the vertices in X —they have not been visited (1) and are reachable (2), and all other components have either been visited (3), or are unreachable (4). Running *DFSReach* on any vertex that is not first in its component will return the empty set since its component has already been visited.

Finally the components are returned in topological order since in F the first vertex in each component are in topological order, and hence will be visited in that order. \square

Exercise 45.7. Give the algorithm for *DFSReach* used in the [SCC algorithm](#) in terms of the [generic DFS algorithm](#).

Exercise 45.8. What is the work and span of the [SCC algorithm](#)? Work out the representation that you use for the graphs as well as other data structures such as the visited set.

8 Discussions

Compared to [BFS](#) DFS has the disadvantage of being sequential. If we don't insist on visiting vertices in DFS order, then it is possible to parallelize DFS; such parallel unordered DFS algorithms are beyond the scope of this book. Given that BFS exposes plenty of parallelism, one might ask why do we need DFS at all. There are several reasons why.

- BFS and DFS can have very different space requirements. For most practical graphs, the frontier in BFS will be significantly larger than the frontier in DFS, because most graphs have small diameters. In some cases, keeping in memory the frontier of BFS could be a challenge.
- In some applications, the graph is so large that it cannot possibly be represented directly but must be “discovered” as we traverse it. In such graphs, computing the frontier of BFS could be infeasible. Many graphs representing large search spaces have this property, e.g., a game search tree or the configuration space in (robot) motion planning. Because DFS “drills down” to the “goal”, it performs much better.

- In same applications DFS exhibits better “data locality” compared to BFS. For example, many garbage collection algorithms benefit from DFS rather than BFS, because DFS traversal order corresponds more closely with data layout.

Chapter 46

Introduction

Given a graph where edges are labeled with weights (or distances) and a source vertex, what is the shortest path between the source and some other vertex? Problems requiring us to answer such queries are broadly known as *shortest-paths problems*.

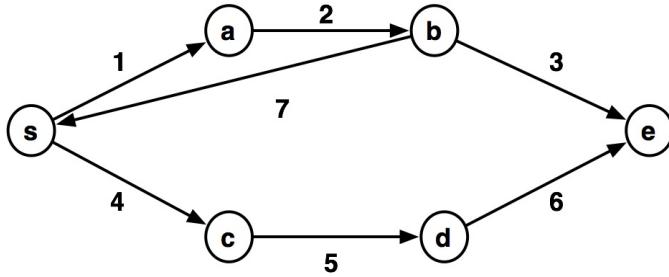
In this chapter, we define several flavors of shortest-path problems and in the next three chapters describe three algorithms for solving them: [Dijkstra's](#), [Bellman-Ford's](#), and [Johnson's](#) algorithms. Dijkstra's algorithm is more efficient but it is mostly sequential and it works only for graphs where edge weights are non-negative. Bellman-Ford's algorithm is a good parallel algorithm and works for all graphs but performs significantly more work. Dijkstra's and Bellman-Ford's algorithms both find shortest paths from a single source, Johnson's algorithm extends this to find shortest paths between all pairs of vertices. It is also highly parallel.

1 Path Weights

Consider a weighted graph $G = (V, E, w)$, where V is the set of vertices, E is the set of edges, and $w : E \rightarrow \mathbb{R}$ is a function mapping each edge to a real number, or a weight. The graph can either be directed or undirected. For convenience we define $w(u, v) = \infty$ if $(u, v) \notin E$.

Definition 46.1 (Path Weight). Given a weighted graph, we define the *weight of the path* in the graph as the sum of the weights of the edges along that path.

Example 46.1. In the following graph the weight of the path $\langle s, a, b, e \rangle$ is 6. The weight of the path $\langle s, a, b, s \rangle$ is 10.

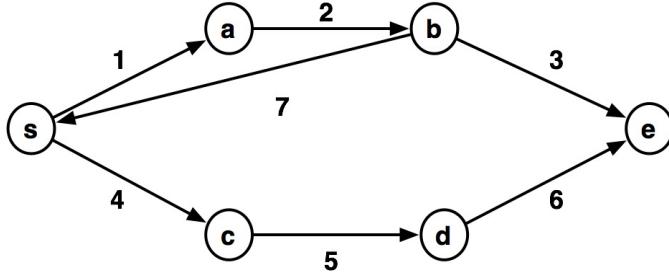


Definition 46.2 (Shortest Paths and Distance). For a weighted graph $G(V, E, w)$ a *shortest weighted path*, or *shortest path* from vertex u to vertex v is a path from u to v with minimal weight. In other words, a shortest path is the path with the smallest weight among all paths from u to v . Note that there could be multiple paths with equal weight; if so they are all shortest paths from u to v .

We define the *distance* from u to v , written $\delta_G(u, v)$, as the weight of a shortest path from u to v . If there is no path from u to v , then the distance is infinity, i.e., $\delta_G(u, v) = \infty$.

Note. Historically, the term “shortest path” is used as a brief form for “shortest weighted path” even though the term “shortest path” is inconsistent with the use of the term “weight.” Some authors use the term “length” instead of weight; this approach has the disadvantage that length are usually thought to be non-negative.

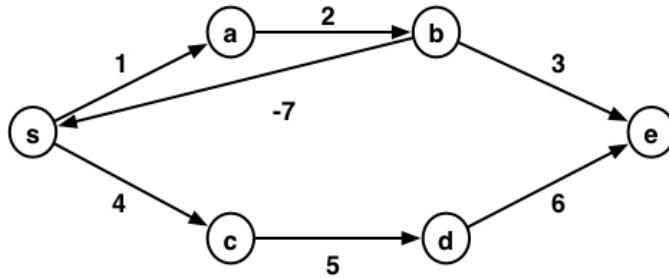
Example 46.2. In the following graph, the shortest path from s to e is $\langle s, a, b, e \rangle$ with weight 6.



Negative Edge Weights. Negative edge weights have significant influence on shortest paths, because they could cause the weight of paths to become non-monotonic: we can decrease the weight of a path by adding a negative-weight edge to the path. Furthermore, negative-weight edges allow for cycles with negative total weight. Such cycles in turn can lead to shortest paths with total weight $-\infty$.

For these reasons, we will need to be careful about negative edge weights when computing shortest paths. As we will discover, even if there are no negative weight cycles, negative edge weights make finding shortest paths more difficult.

Example 46.3 (Negative Weights and Cycles). In the graph below, the cycle $\langle s, a, b, s \rangle$ has a weight of -4 and the cycle $\langle s, a, b, s, a, b, s \rangle$ has a weight of -8 . This means that the shortest path from s to e has distance $-\infty$.



2 Shortest Path Problems

Shortest path problems come in several flavors, such as single-pair, single-source, and all-pairs problems.

Problem 46.1 (Single-Pair Shortest Paths). Given a weighted graph G , a source vertex s , and a destination vertex t , the *single-pair* shortest path problem is to find the shortest weighted path from s to t in G .

Problem 46.2 (Single-Source Shortest Paths (SSSP)). Given a weighted graph $G = (V, E, w)$ and a source vertex s , the *single-source shortest path (SSSP) problem* is to find a shortest weighted path from s to every other vertex in V .

Problem 46.3 (All-Pairs Shortest Paths). Given a weighted graph, the *all-pairs shortest path problem* is to find the shortest paths between all pairs of vertices in the graph.

Problem 46.4 (SSSP⁺). Consider a variant of the SSSP problem, where all the weights on the edges are non-negative (i.e. $w : E \rightarrow \mathbb{R}^+$). We refer to this as the *SSSP⁺ problem*.

Note. Shortest-path problems typically require finding only one of the possibly many shortest paths between two vertices considered. In some cases, we only care about the distance $\delta_G(u, v)$ but not the path itself.

3 The Sub-Paths Property

Definition 46.3 (Sub-Path). A *sub-path* of a path is itself a path that is contained within the path.

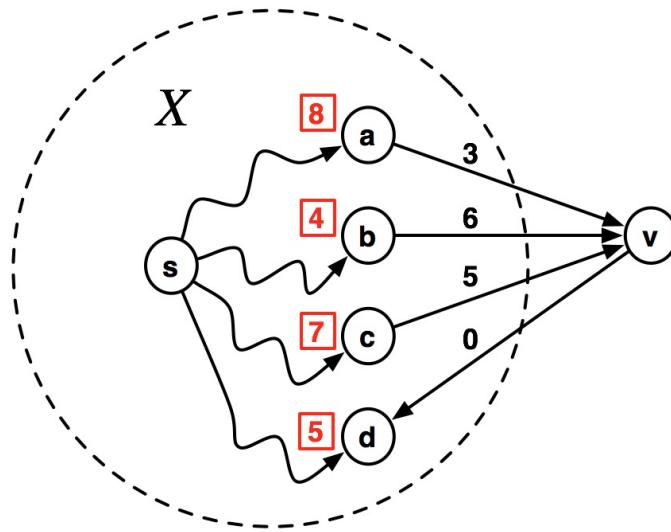
Example 46.4. If $p = \langle a, b, c, e \rangle$ is a path in a graph, then the following are some valid subpaths of p :

- $\langle a, b, c, e \rangle$,
- $\langle a, b, c \rangle$,
- $\langle b, c \rangle$, and
- $\langle b \rangle$.

Definition 46.4 (Sub-Paths Property). The *sub-paths property* states a basic fact about shortest paths: any sub-path of a shortest path is itself a shortest path. The sub-paths property makes it possible to construct shortest paths from smaller shortest paths. It is used in all efficient shortest paths algorithms we know about.

Example 46.5 (Subpaths property). If a shortest path from Pittsburgh to San Francisco goes through Chicago, then that shortest path includes the shortest path from Pittsburgh to Chicago, and from Chicago to San Francisco.

Applying the Sub-Paths Property. To see how the sub-paths property can help find shortest paths, consider the graph G shown below.



Suppose that an oracle has told us the shortest paths from s to all vertices except for the vertex v , shown in red squares. We want to find the shortest path to v .

By inspecting the graph, we know that the shortest path to v goes through either one of a , b , or c . Furthermore, by the sub-paths property, we know that the shortest path to v consists of the shortest path to one of a , b , or c , and the edge to v . Thus, all we have to do is to find the vertex u among the in-neighbors of v that minimizes the distance to v , i.e., $\delta_G(s, u)$ plus the additional edge weight to get to v . The weight of the shortest path to v is therefore

$$\min (\delta_G(s, a) + 3, \delta_G(s, b) + 6, \delta_G(s, c) + 5).$$

The shortest path therefore goes through the vertex b with distance 10, which minimizes the weight. We will use this observation in both Dijkstra's and Bellman-Ford's algorithm.

Exercise 46.5. Prove that the sub-paths property is correct.

Solution. We prove the property by using proof by contradiction. Consider a shortest path p and suppose that it has a sub-path q that is not itself a shortest path between its end-points. Replace now the sub-path q with a shortest path between the same end points. This results in a path p' that is shorter than p . But p is a shortest path to start with, so a contradiction.

Chapter 47

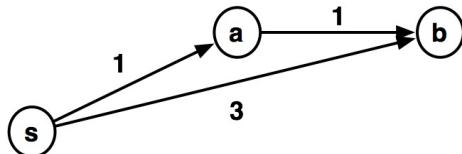
Dijkstra's Algorithm

Dijkstra's algorithm solves the single source shortest path problem with non-negative edge weights, i.e., the SSSP⁺ problem. It uses priority-first search. It is a sequential algorithm.

1 Dijkstra's Property

In Chapter 44, we saw how BFS can be used to solve the single-source shortest path problem on graphs without edge weights, or, equivalently, where all edges have weight 1. BFS, however, does not work on general weighted graphs, because it ignores edge weights (there can be a shortest path with more edges).

Example 47.1. Consider the following directed graph with 3 vertices.



In this graph, a BFS visits b and a on the same round, marking the shortest path to both as directly from s . However the path to b via a is shorter. Since BFS never visits b again, it will not find the actual shortest path.

Brute Force. Let's start by noting that since no edges have negative weights, there cannot be a negative-weight cycle. One can therefore never make a path shorter by visiting a vertex twice—i.e., a path that cycles back to a vertex cannot have less weight than the path

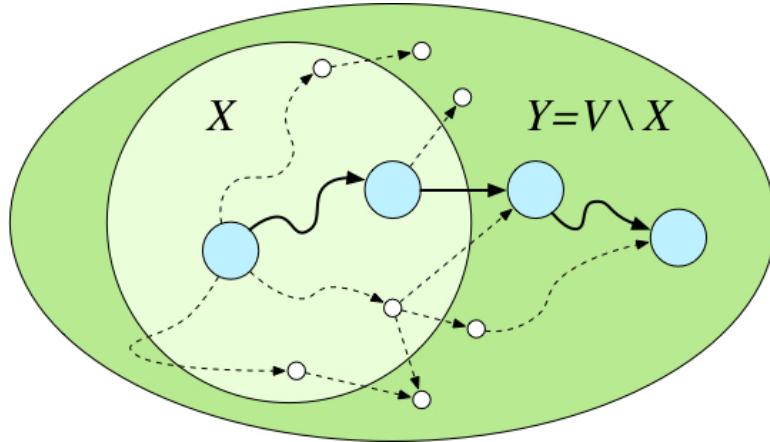
that ends at the first visit to the vertex. When searching for a shortest path, we thus have to consider only the simple paths, i.e., paths with no repeated vertices.

Based on this observation we can use a brute-force algorithm for the SSSP⁺ problem, that, for each vertex, considers all simple paths between the source and a destination and selects the shortest such path. Unfortunately there can be a large number of paths between any pair of vertices (exponential in the number of vertices), so any algorithm that tries to look at all paths is not likely to scale beyond very small instances.

Applying the Sub-Paths Property. Let's try to reduce the work. The brute-force algorithm does redundant work since it does not take advantage of the sub-paths property, which allows us to build shortest paths from smaller shortest paths.

Suppose that we have an oracle that tells us the shortest paths from s to some subset of the vertices $X \subset V$ with $s \in X$. Also let's define Y to be the vertices not in X , i.e.,

$$Y = V \setminus X.$$



Consider now this question: can we efficiently determine the shortest path to any one of the vertices in Y ? If we could do this, then we would have an algorithm to add new vertices to X repeatedly, until we are done.

To see if this can be done, let's define the *frontier* as the set of vertices that are neighbors of X but not in X , i.e. $N^+(X) \setminus X$ (as in graph search). Observe that any path that leaves X must go through a frontier vertex on the way out. Therefore for every $v \in Y$ the shortest path from s must start in X , since $s \in X$, and then leave X via a vertex in the frontier.

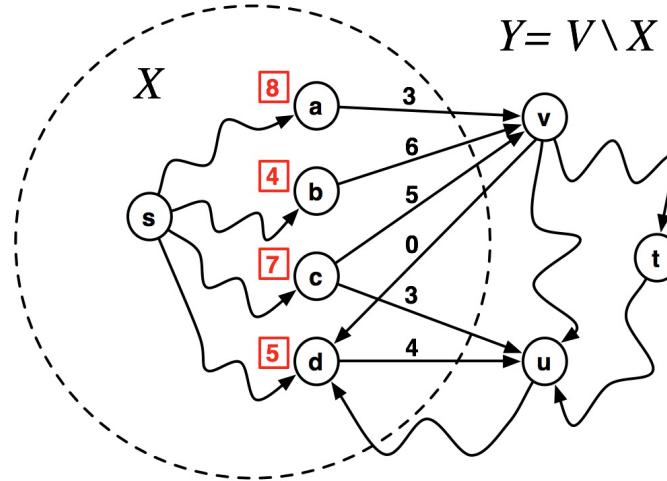
Now consider for each vertex v in the frontier F , the shortest path length that consists of a path through X and then an edge to v , i.e., $p(v) = \min_{x \in X} (\delta_G(s, x) + w(x, v))$. Among these consider the vertex v with overall shortest path length, $\min_{v \in F} p(v)$. Observe that no other vertex in Y can be closer to the source than v because

- all paths to Y must go through the frontier when exiting X , and
- edge weights are non-negative so a sub-path cannot be longer than the full path.

Therefore for non-negative edge weights the path that minimizes $\delta_G(s, x) + w(x, v)$, over all $x \in X$ and $v \in Y$, is a minimum length path that goes from s to Y , i.e., $\min_{y \in Y} \delta_G(s, y)$. There can be ties. We thus have answered the question that we set out to answer: we can indeed determine the shortest path to one more vertex.

Lemma 47.1 establishes this intuition more formally.

Example 47.2. In the following graph suppose that we have found the shortest paths from the source s to all the vertices in X (marked by numbers next to the vertices). The overall shortest path to any vertex on the frontier via X and one more edge, is the path to vertex u via vertex d , with length $5 + 4 = 9$. If edge weights are non-negative there cannot be any shorter way to get to u , whatever the path length from v to u is, therefore we know that $\delta(s, u) = 9$. Note that if edges can be negative, then it could be shorter to go through v since the path length from v to u could be negative.



Lemma 47.1 (Dijkstra's Property). Consider a (directed) weighted graph $G = (V, E, w)$, and a source vertex $s \in V$. Consider any partitioning of the vertices V into X and $Y = V \setminus X$ with $s \in X$, and let

$$p(v) = \min_{x \in X} (\delta_G(s, x) + w(x, v))$$

then $\min_{y \in Y} p(y) = \min_{y \in Y} \delta_G(s, y)$.

Note (The Lemma Explained in plain English). The overall shortest-path weight from s via a vertex in X directly to a neighbor in Y (in the frontier) is as short as any path from s to any vertex in Y .

Proof. Consider a vertex $v_m \in Y$ such that $\delta_G(s, v_m) = \min_{v \in Y} \delta_G(s, v)$, and a shortest path from s to v_m in G . The path must go through an edge from a vertex $v_x \in X$ to a vertex v_t in Y . Since there are no negative weight edges, and the path to v_t is a sub-path of the path to v_m , $\delta_G(s, v_t)$ cannot be any greater than $\delta_G(s, v_m)$ so it must be equal. We therefore have

$$\min_{y \in Y} p(y) \leq \delta_G(s, v_t) = \delta_G(s, v_m) = \min_{y \in Y} \delta_G(s, y),$$

but the leftmost term cannot be less than the rightmost, so they must be equal. \square

The reader might have noticed that the terminology that we used in explaining Dijkstra's algorithm closely relates to that of graph search. More specifically, recall that priority-first search is a graph search, where each round visits the frontier vertex with the highest priority. If, as usual, we denote the visited set by X , we can define the priority for a vertex v in the frontier, $p(v)$, as the weight of the shortest-path consisting of a path to $x \in X$ and an additional edge from x to v , as in Lemma 47.1. We can thus define Dijkstra's algorithm in terms of graph search.

Algorithm 47.1 (Dijkstra's Algorithm). For a weighted graph $G = (V, E, w)$ and a source s , Dijkstra's algorithm is priority-first search on G that

- starts at s with $d(s) = 0$,
- uses priority $p(v) = \min_{x \in X} (d(x) + w(x, v))$ (to be minimized), and
- sets $d(v) = p(v)$ when v is visited.

When finished, returns $d(v)$.

Note. Dijkstra's algorithm visits vertices in non-decreasing shortest-path weight since on each round it visits unvisited vertices that have the minimum shortest-path weight from s .

Theorem 47.2 (Correctness of Dijkstra's Algorithm). Dijkstra's algorithm returns $d(v) = \delta_G(s, v)$ for v reachable from s .

Proof. We show that for each step in the algorithm, for all $x \in X$ (the visited set), $d(x) = \delta_G(s, x)$. This is true at the start since $X = \{s\}$ and $d(s) = 0$. On each step the search adds vertices v that minimizes $P(v) = \min_{x \in X} (d(x) + w(x, v))$. By our assumption we have that $d(x) = \delta_G(s, x)$ for $x \in X$. By Lemma 47.1, $p(v) = \delta_G(s, v)$, giving $d(v) = \delta_G(s, v)$ for the newly added vertices, maintaining the invariant. As with all priority-first searches, it will eventually visit all reachable v . \square

2 Dijkstra's Algorithm with Priority Queues

As described so far, Dijkstra's algorithm does not specify how to calculate or maintain the priorities. One way is to calculate all priorities of the frontier on each round of the

search. This is effectively how Dijkstra originally described the algorithm. However, it is more efficient to maintain the priorities with a priority queue. We describe and analyze an implementation of Dijkstra's algorithm using priority queues.

Algorithm 47.2 (Dijkstra's Algorithm using Priority Queues). An implementation of Dijkstra's algorithm that uses a priority queue to maintain $p(v)$ is shown below. The priority queue PQ supports *deleteMin* and *insert* operations.

```

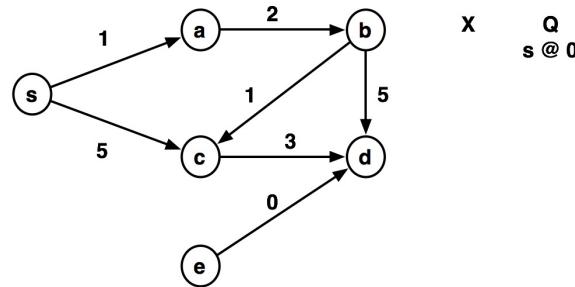
1  dijkstraPQ  $G s =$ 
2  let
3    dijkstra  $X Q =$           (*  $X$  maps vertices to distances. *)
4    case  $PQ.\underline{deleteMin}$   $Q$  of
5      (None,  $_$ )  $\Rightarrow X$           (* Queue empty, finished. *)
6      | (Some  $(d, v), Q'$ )  $\Rightarrow$ 
7        if  $(v, \_) \in X$  then dijkstra  $X Q'$       (* Already visited, skip. *)
8        else
9          let
10             $X' = X \cup \{(v, d)\}$           (* Set final distance of  $v$  to  $d$ . *)
11             $\underline{relax} (Q, (u, w)) = PQ.\underline{insert} (d + w, u) Q$ 
12             $Q'' = \underline{iterate relax} Q' (N_G^+(v))$   (* Add neighbors to  $Q$ . *)
13            in dijkstra  $X' Q''$  end
14     $Q_0 = PQ.\underline{insert} (0, s) PQ.\underline{empty}$       (* Initial  $Q$  with source. *)
15  in dijkstra  $\{\} Q_0$  end

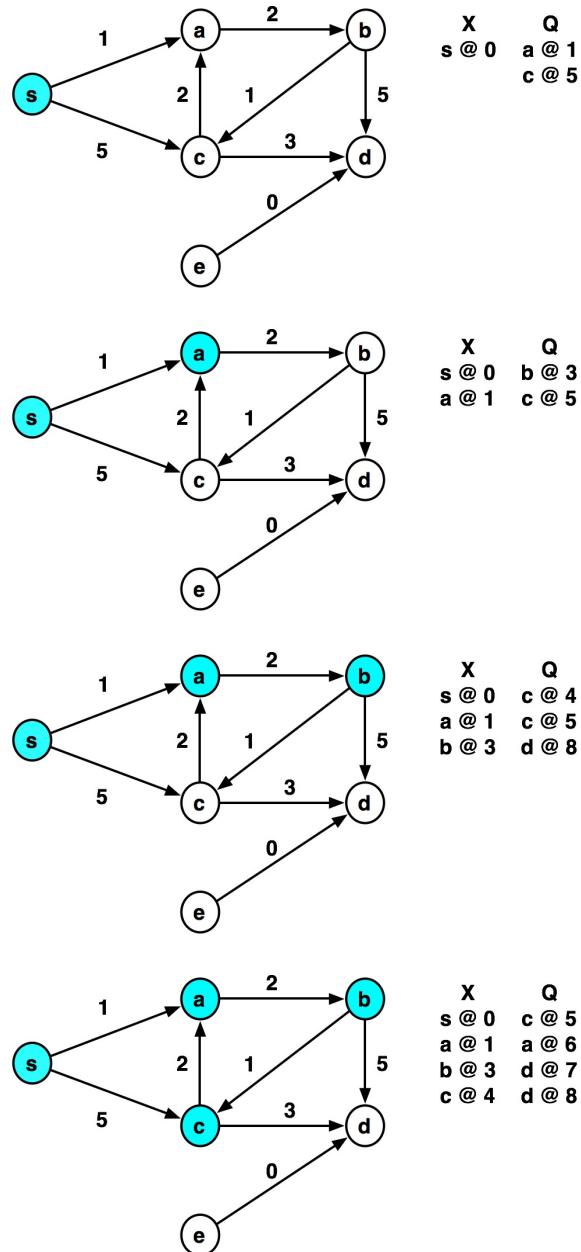
```

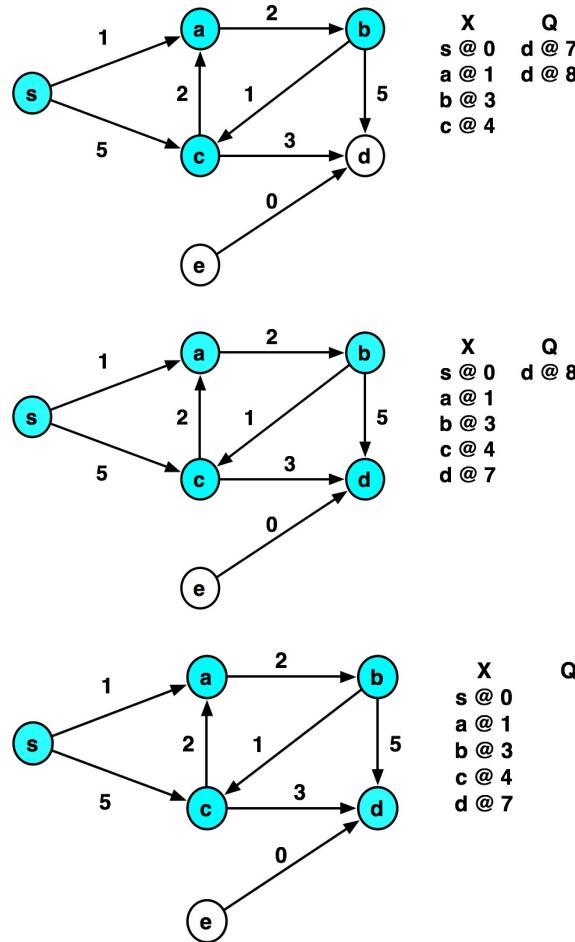
We assume $N_G^+(v)$ returns a pair $(u, w(v, u))$ for each neighbor u .

Remark. This algorithm only visits one vertex at a time even if there are multiple vertices with equal distance. It would also be correct to visit all vertices with equal minimum distance in parallel. In fact, BFS can be thought of as the special case of Dijkstra in which the whole frontier has equal distance.

Example 47.3. An example run of Dijkstra's algorithm. Note that after visiting s , a , and b , the queue Q contains two distances for c corresponding to the two paths from s to c discovered thus far. The algorithm takes the shortest distance and adds it to X . A similar situation arises when c is visited, but this time for d . Note that when c is visited, an additional distance for a is added to the priority queue even though it is already visited. Redundant entries for both are removed next before visiting d . The vertex e is never visited as it is unreachable from s . Finally, notice that the distances in X never decrease.







The algorithm maintains the visited set X as a table mapping each visited vertex u to $d(u) = \delta_G(s, u)$. It also maintains a priority queue Q that keeps the frontier prioritized based on the shortest distance from s directly from vertices in X . On each round, the algorithm selects the vertex x with least distance d in the priority queue and, if it has not already been visited, visits it by adding $(x \mapsto d)$ to the table of visited vertices, and then adds all its neighbors v to Q with the priorities $d(x) + w(x, v)$ (i.e. the distance to v through x).

Note that a neighbor might already be in Q since it could have been added by another of its in-neighbors. Q can therefore contain duplicate entries for a vertex with different priorities, but what is important is that the minimum distance will always be pulled out first. Line 7 checks to see whether a vertex pulled from the priority queue has already been visited and discards it if it has. This algorithm is just a concrete implementation of the previously described Dijkstra's algorithm.

Remark. There are a couple other variants on Dijkstra's algorithm using priority queues.

One variant checks whether u is already in X inside the *relax* function, and if so does not

inserts it into the priority queue. This does not affect the asymptotic work bounds, but might give some improvement in practice.

Another variant decreases the priority of the neighbors instead of adding duplicates to the priority queue. This requires a more powerful priority queue that supports a *decreaseKey* function.

3 Cost Analysis of Dijkstra's Algorithm

Data Structures. To analyze the work and span of priority-queue based implementation of Dijkstra's algorithm shown above, let's first consider the priority queue ADT's that we use. For the priority queue, we assume $PQ.insert$ and $PQ.deleteMin$ have $O(\lg n)$ work and span. We can represent the mapping from visited vertices to their distances as a table, and input graph as an adjacency table.

Analysis. To analyze the work, we calculate the work for each significant operation and sum them up to find the total work. Each vertex is only visited once and hence each out edge is only visited once. The table below summarizes the costs of the operations, along with the number of calls made to each. We use, as usual, $n = |V|$ and $m = |E|$.

Operation	Line	Number of calls	PQ	Tree Table
$PQ.deleteMin$	4	m	$O(\lg m)$	—
$PQ.insert$	11	m	$O(\lg m)$	—
$find$	7	m	—	$O(\lg n)$
$insert$	10	n	—	$O(\lg n)$
$N_G^+(v)$	12	n	—	$O(\lg n)$
<i>Total</i>	—	$O(m \lg n)$	$O(m \lg n)$	$O(m \lg n)$

The work is therefore

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

Since the algorithm is sequential, the span is the same as the work.

Decrease Key Operation. We can improve the work of the algorithm by using a priority queues that supports a *decreaseKey* operation. This leads to $O(m + n \log n)$ work across all priority queue operations.

Exercise 47.1. Give a version of [Dijkstra's algorithm](#) that uses the *decreaseKey* operation.

Remark. For enumerated graphs the cost of the tree tables could be improved by using adjacency sequences for the graph, and ephemeral or single-threaded sequences for the

distance table, which are used linearly. This does not improve the overall asymptotic work of [Dijkstra's algorithm](#) because the cost of the priority queue operations at $O(m \log n)$ dominate.

In the *decreaseKey* version of the algorithm, the cost of priority queue operations are $O(m + n \log n)$. In this case, using sequences for graphs still does not help, because of the $O(n \log n)$ term dominates. But a more efficient structure for the mapping of vertices to distances could help.

Chapter 48

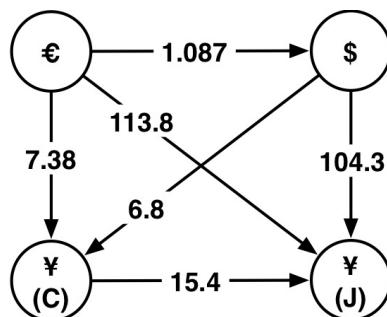
Bellman-Ford's Algorithm

Bellman-Ford's algorithm solves the single source shortest path problem with arbitrary edge weights, and with significant parallelism.

1 Graphs with Negative Edge Weights

Negative edge weights might appear to be unnatural. For example, in a “map” graph, which represents intersections and the roads between them, quantities of interest such as the travel time, or the length of the road between two vertices are non-negative. Negative weights, do however, arise when we consider more complex properties and when we reduce other problems to shortest paths.

Example 48.1 (Reducing Currency Exchange to Shortest Paths). Consider the following *currency exchange* problem: given a set of currencies and a set of exchange rates between them, find the sequence of exchanges to get from currency u to currency v that gives the maximum yield. Here is an example:



The best sequence of exchanges from euros to Japanese yen, is through US dollars and Chinese renminbi.

We now reduce this to the shortest paths problem. Given a unit in currency u , a path of exchanges from u to v with rates r_0, \dots, r_l will yield $r_0 \times \dots \times r_l$ units in currency v . This is a product, but we can use the logarithm function to it to a sum. The idea is for each exchange rate $r(u, v)$ to include an edge (u, v) with weight $w(u, v) = -\lg(r(u, v))$. The weights can be negative. Now we have that for a path P with weights w_0, \dots, w_l

$$\begin{aligned} w(P) &= w_0 + \dots + w_l \\ &= -(\lg(r_0) + \dots + \lg(r_l)) \\ &= -\lg(r_0 \times \dots \times r_l) \end{aligned}$$

Hence by finding the shortest path we will find the path that has the greatest yield, and furthermore the yield will be $2^{-w(P)}$.

Exercise 48.1. In the currency exchange example, what does a negative weight cycle imply?

Impact of Negative Weights on Shortest Paths. Consider a graph with negative edge weights. Are shortest paths on this graph always well defined?

To answer this question, consider two cases.

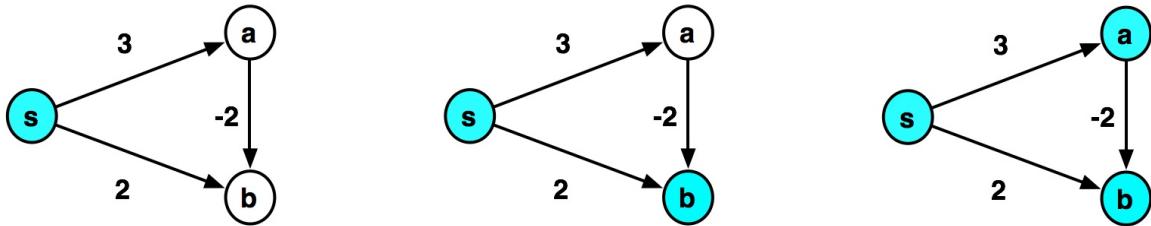
- First, assume that the graph does not have any cycles with total negative weight, i.e., for any cycle the sum of the weights of the edges on that cycle is not less than zero. In this case, we can conclude that there is a shortest path between any two vertices that is simple, i.e., contains no cycles.
- Second, assume that the graph has a cycle with total negative weight. In this case, any shortest simple path that passes through a vertex in this cycle can be made shorter by extending the path with the cycle. Furthermore, this process can be repeated, ultimately allowing us to prove that the shortest path is $-\infty$.

Based on this analysis, we expect a shortest path algorithm to alert us when it encounters a negative-weight cycle. In other words, if there is a relevant negative-weight cycle, then the algorithm should terminate and indicate the presence of such a cycle, otherwise, the algorithm should compute the shortest paths as desired.

Exercise 48.2. Prove that if a graph has no negative-weight cycles, then there is a shortest path between any two vertices that is simple.

Dijkstra with Negative Weights. Recall that Dijkstra's assumes non-negative (edge) weights. This assumption allows the algorithm to consider simple paths (with no cycles) only and to construct longer shortest paths from shorter ones iteratively. The absence of negative weights is crucial to this approach and the [Dijkstra's property](#), which underpins the approach does not hold in the presence of negative weights.

Example 48.2. To see where Dijkstra's property fails with negative edge weights consider the following example.



Dijkstra's algorithm would visit b then a and leave b with a distance of 2 instead of the correct distance 1. The problem is that when Dijkstra's algorithm visits b , it fails to consider the possibility of there being a shorter path from a to b (which is impossible with non-negative edge weights).

2 Bellman-Ford's Algorithm

Intuition behind Bellman-Ford. To develop some intuition for finding shortest paths in graphs with negative edge weights, let's recall the sub-path property of shortest paths. This property states that any sub-path of a shortest path is a shortest path (between its end vertices). The sub-paths property holds regardless of the edge weights.

Dijkstra's algorithm exploits this property by building longer paths from shorter ones, i.e., by building shortest paths in non-decreasing order. With negative edge weights this does not work anymore, because paths can get shorter as we add edges.

There is another way to exploit the same property: building paths that contain more and more edges. To see how, suppose that we have found the shortest paths from a source to all vertices with k or fewer edges or "hops". We can compute the shortest path with $(k+1)$ or fewer hops from k or fewer hops extending all paths by one edge if doing so leads to a shorter path and leaving them unchanged otherwise. If the graph does not have any negative-weight cycles, then all shortest paths are simple and thus contain $|V| - 1$ or fewer edges. Thus we only have to repeat this process as many times as the number of vertices. This is the basic idea behind [Bellman-Ford's algorithm](#), the details of which are worked out in the rest of this chapter.

Definition 48.1 (k -hop Distance). For a graph G , the *k -hop distance*, written $\delta_G^k(u, v)$, is the shortest path from u to v considering only paths with at most $k \geq 0$ edges. If no such path with k or fewer edges exists, then $\delta_G^k(u, v) = \infty$

Computing k -hop distances. Given $G = (V, E)$, suppose now that we have calculated the k -hop distances, $\delta_G^k(s, v)$, to all vertices $v \in V$. To find the $(k + 1)$ -hop distance to v , we consider the incoming edges of v and pick the shortest path to that vertex that arrives at an in-neighbor u using k edges, i.e., $\delta_G^k(s, u)$, and then takes the edge (u, v) , i.e.,

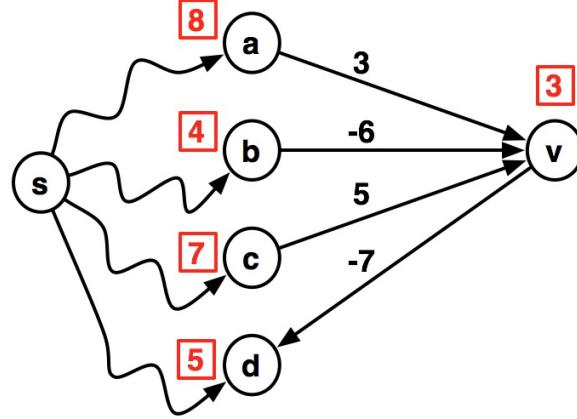
$$\delta^{k+1}(v) = \min(\delta^k(v), \min_{x \in N^-(v)} (\delta^k(x) + w(x, v))).$$

Recall that $N^-(v)$ indicates the in-neighbors of vertex v .

Example 48.3. In the following graph G , suppose that we have found the shortest paths from the source s to vertices using k or fewer edges. Each vertex u is labeled with its k -distance to s , written $\delta_G^k(s, u)$. The weight of the shortest path to v using $k + 1$ or fewer edges is

$$\min(\delta_G^k(s, v), \min \delta_G^k(s, a) + 3, \delta_G^k(s, b) - 6, \delta_G^k(s, c) + 5).$$

The shortest path with at most $k + 1$ edges has weight -2 and goes through vertex b .



Computing Shortest Paths Iteratively. Now that we [know how to construct shortest paths with more hops](#), we can iterate the approach to obtain an algorithm for computing shortest paths in $G = (V, E)$ from a source $s \in V$ as follows.

- Start by determining $\delta_G^0(s, v)$ for all $v \in V$. Since no vertex other than the source is reachable with a path of length 0, we have:
 - $\delta_G^0(s, s) = 0$, and
 - $\delta_G^0(s, v) = \infty$ for all $v \neq s$.
- Next, iteratively for each round $k > 0$, compute for each vertex in parallel $\delta_G^{k+1}(s, v)$ using $\delta_G^k(s, \cdot)$'s as [described above](#).

The only question left is when to stop the iteration. Note that because the $(k + 1)$ -hop distance for a vertex is calculated in terms of the k -hop distances, if the $(k + 1)$ -hop distances of all vertices remain unchanged relative to the k -hop distances, then the $k + 2$ -hop distances will also be the same. Therefore we can stop when an iteration produces no change in the distances of vertices.

But, are we guaranteed to stop? Not, if we have negative-weight cycles reachable from the source, because such a cycle will decrease distances of some vertices on every iteration. But, we can detect negative cycles by checking that the distances have not converged after $|V|$ iterations, because a simple (acyclic) path in a graph can include at most $|V| - 1$ edges and, in the absence of negative-weight cycles, there always exist a simple shortest path.

Algorithm 48.2 (Bellman Ford Algorithm). The pseudo-code below shows the Bellman-Ford algorithm for computing shortest paths in weighted graphs, where edge weights can be negative. The algorithm terminates and returns either the weight of the shortest paths for all vertices reachable from the source s or it indicates that the graph has a negative-weight cycle.

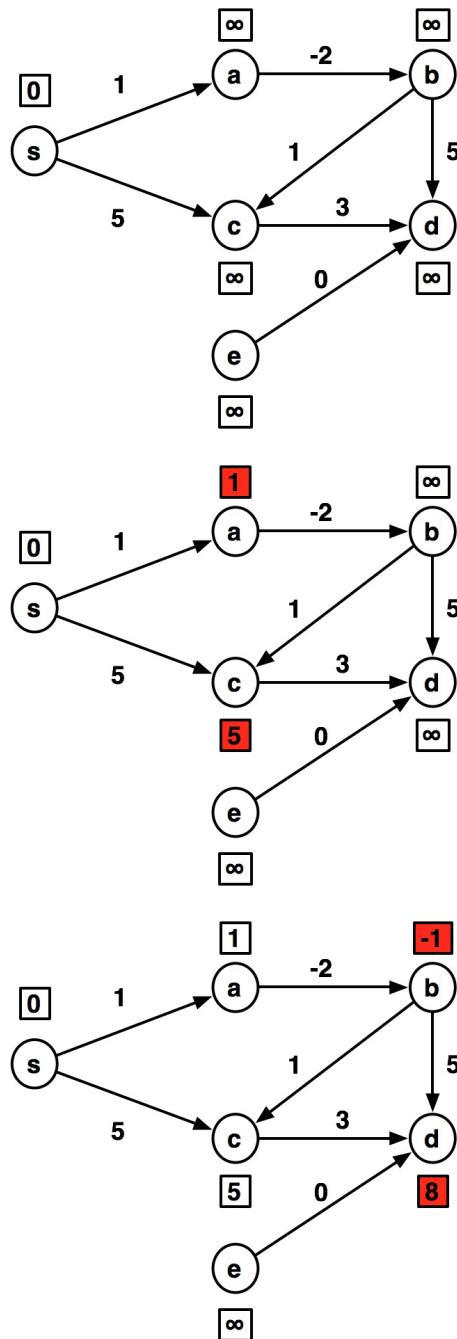
The algorithm runs until convergence or until $|V|$ iterations have been performed. If after $|V|$ iterations, and the distances does not converge, then the algorithm concludes that there is a negative-weight cycle that is reachable from the source vertex and returns `None` (Line 9).

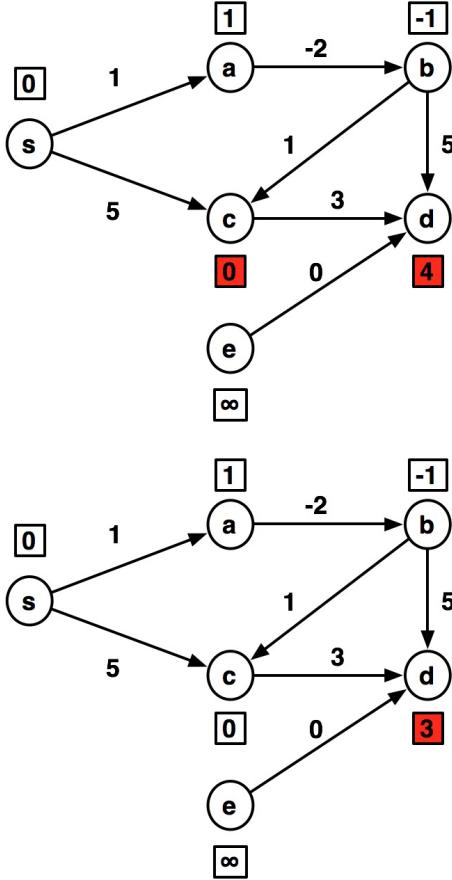
```

1  BellmanFord (G = (V, E)) s =
2  let
3  BF D k =
4  let
5       $D_{\text{in}}(v) = \min_{u \in N_G^-(v)} (D[u] + w(u, v))$  (* Min in distance. *)
6       $D' = \{v \mapsto \min(D[v], D_{\text{in}}(v)) : v \in V\}$  (* New distances. *)
7  in
8      if ( $k = |V|$ ) then None (* Negative cycle, quit. *)
9      else if ( $\text{all } \{D[v] = D'[v] : v \in V\}$ ) then
10         Some  $D$  (* No change so return distances. *)
11      else BF  $D'$  ( $k + 1$ ) (* Repeat. *)
12  end
13   $D_0 = \{v \mapsto \infty : v \in V \setminus \{s\}\} \cup \{s \mapsto 0\}$  (* Initial distances. *)
14  in BF  $D_0$  0 end

```

Example 48.4. Several steps of the Bellman Ford algorithm are shown below. The numbers with squares indicate the current distances and highlight those that has changed on each step.





Theorem 48.1 (Correctness of Bellman-Ford). Given a directed weighted graph $G = (V, E, w)$, $w : E \rightarrow R$, and a source s , the *BellmanFord* algorithm returns either $\delta_G(s, v)$ for all vertices reachable from s , or indicates that there is a negative weight-cycle in G that is reachable from s .

Proof. By induction on the number of edges k in a path. The base case is correct since $D_s = 0$. For all $v \in V \setminus s$, on each step a shortest (s, v) path with up to $k + 1$ edges must consist of a shortest (s, u) path of up to k edges followed by a single edge (u, v) . Therefore if we take the minimum of these we get the overall shortest path with up to $k + 1$ edges. For the source the self edge will maintain $D_s = 0$. The algorithm can only proceed to n rounds if there is a reachable negative-weight cycle. Otherwise a shortest path to every v is simple and can consist of at most n vertices and hence $n - 1$ edges. \square

3 Cost Analysis

Data Structures. To analyze the cost Bellman-Ford, we first determine the representations

for each structure in the algorithm. For the distance structure D , we use a table mapping vertices to their distances. For graphs, we consider two different representations, one with tables and another with sequences. In each case, we represent the graph as a mapping from each vertex v to another inner mapping from each in-neighbor u to $w(u, v)$. In turn, we represent the mapping with a table or with a sequence.

Cost with Tables. Consider the cost of one round of the [Bellman-Ford's algorithm](#), i.e., a call to BF excluding recursive calls. The only nontrivial computations are on Lines 5, 6 and 9.

Lines 5 and 6 *tabulate* over the vertices. As the [cost specification for tables with BSTs](#) show, the cost of *tabulate* over the vertices is the sum of the work for each vertex, and the maximum of the spans plus $O(\lg n)$.

Now consider Line 5.

- First, the algorithm finds the neighbors in the graph, using a *find* $G v$. This requires $O(\lg |V|)$ work and span.
- Second, the algorithm performs a *map* over the in-neighbors. Each position of the *map* requires a *find* in the distance table for $D[u]$, a *find* in the weight table for $w(u, v)$, and an *add*. The *find* operations take $O(\lg |V|)$ work and span.
- Third, the algorithm reduces over the in-neighbors to determine the shortest path through an in-neighbor. This requires $O(1 + |N_G(v)|)$ work and $O(\lg |N_G(v)|)$ span.

Line 6 performs a *tabulate* by taking for each vertex the minimum of $D[v]$ and $D_{\text{in}}(v)$, which requires constant work for each vertex. Using $n = |V|$ and $m = |E|$, we can write the work for Lines 5 and 6 as

$$\begin{aligned} W_{BF}(n, m) &= O\left(\sum_{v \in V} \left(\lg n + |N_G^-(v)| + \sum_{u \in N_G^-(v)} (1 + \lg n)\right)\right) \\ &= O((n + m) \lg n). \end{aligned}$$

The first term is for looking up the current distance, the second term is for reduce, and the third term is the cost for mapping over the neighbors.

Similarly, we can write the span by replacing the sums over vertices with maximums:

$$\begin{aligned} S_{BF}(n, m) &= O\left(\max_{v \in V} \left(\lg n + \lg |N_G^-(v)| + \max_{u \in N_G^-(v)} (1 + \lg n)\right)\right) \\ &= O(\lg n). \end{aligned}$$

Line 9 performs a *tabulate* and a *reduce* and thus requires $O(n \lg n)$ work and $O(\lg n)$ span.

Because the number of rounds (calls to BF) is bounded by n , and because the rounds are sequential, we multiply the work and span for each round by the number of calls to

compute the total work and span, which, assuming $m \geq n$, are

$$\begin{aligned} W_{BF}(n, m) &= O(nm \lg n) \\ S_{BF}(n, m) &= O(n \lg n). \end{aligned}$$

Cost with Sequences. If we assume that the graphs is enumerable, then the vertices are identified by the integers $\{0, 1, \dots, |V| - 1\}$ and we can use sequences to represent the graph. We can therefore use sequences instead of tables to represent [the three mappings](#) used by the algorithm. In particular, we can use an [adjacency sequence](#) for the graphs and the in-neighbors, and an array sequence for the distances. This improves the work for looking up in-neighbors or distances from $O(\log n)$ to $O(1)$. Retracing the steps of [the previous analysis](#) and using the improved costs we obtain:

$$\begin{aligned} W_{BF}(n, m) &= O\left(\sum_{v \in V} \left(1 + |N_G^-(v)| + \sum_{u \in N_G^-(v)} 1\right)\right) \\ &= O(n + m) \\ S_{BF}(n, m) &= O\left(\max_{v \in V} \left(1 + \lg |N_G^-(v)| + \max_{u \in N_G^-(v)} 1\right)\right) \\ &= O(\lg n). \end{aligned}$$

Hence the overall complexity for *BellmanFord* with sequences, assuming $m \geq n$, is

$$\begin{aligned} W(n, m) &= O(nm) \\ S(n, m) &= O(n \lg n) \end{aligned}$$

Using array sequences thus reduces the work by a $O(\lg n)$ factor.

Chapter 49

Johnson's Algorithm

Johnson's algorithm solves the all-pairs shortest paths (APSP) problem. It allows for negative weights and is significantly more efficient than simply running Bellman-Ford's algorithm from each source. It uses a clever trick to tweak the weights and eliminate negative ones without altering shortest paths.

All Pairs from Single Source. One way to solve the APSP problem is by running the Bellman-Ford algorithm from each vertex. For a graph with n vertices and m edges, this gives an algorithm with total work

$$W(n, m) = O(mn) \times n = O(mn^2).$$

Johnson's algorithm improves on this by using Dijkstra's algorithm to find the shortest paths from each source vertex and by using the more expensive Bellman-Ford's algorithm to tweak the weights. Johnson's algorithm proceeds in two phases.

1. The first phase runs Bellman-Ford's algorithm and uses the result to update the weights on the edges and eliminate all negative weights.
2. The second phase runs Dijkstra's algorithm from each vertex in parallel.

For a graph with n vertices and m edges, Johnson's algorithm has the following costs:

	Work	Span
1 \times Bellman Ford	$O(mn)$	$O(n \log n)$
$n \times$ Dijkstra	$n \times O(m \log n)$	$O(m \log n)$
Total	$O(mn \log n)$	$O(m \log n)$.

The work improves over the naive $O(mn^2)$ bound by a factor of $n/\log n$, and the span is no more than a single Dijkstra. The parallelism is therefore $\Theta(n)$, which is significant.

Potentials. To update the weights on the graph we will use Bellman-Ford to assign a “potential” $p(v)$ to each vertex v . We then add and subtract the potentials on the endpoints of each edge to get new weights. This adjustment change the weights of paths in the graph, but it will not change the shortest path (i.e. its sequence of edges) between any two vertices. The following lemma establishes this property, more formally.

Lemma 49.1 (Path Potentials). Consider a weighted graph $G = (V, E, w)$, and any assignment $p(v) : V \rightarrow \mathbb{R}$. For

$$w'(u, v) = w(u, v) + p(u) - p(v) ,$$

and $G' = (V, E, w')$, we have that:

$$\delta_G(u, v) = \delta_{G'}(u, v) - p(u) + p(v) .$$

Proof. Summary: Along any path all potentials except the first and last cancel since each is subtracted from the incoming edge and added to the outgoing edge. Therefore the total weight of a path is the original weight $+p(u) - p(v)$. Since this only depends on u and v , which path is the shortest from u to v does not change, and the change in weight $(p(u) - p(v))$ can be subtracted out.

Full proof: Consider any path of edges $P = \langle (h_0, t_0), \dots, (h_l, t_l) \rangle$ from u to v . Here h and t means head and tail of each edge, and hence $h_0 = u$, $t_l = v$ and $t_i = h_{i+1}$ for $0 \leq i < l$. In the original graph the path length is

$$w(P) = \sum_{i=0}^l w(h_i, t_i) .$$

In the modified graph the path length is

$$\begin{aligned} w'(P) &= \sum_{i=0}^l (w(h_i, t_i) + p(h_i) - p(t_i)) \\ &= w(P) + \sum_{i=0}^l p(h_i) - \sum_{i=0}^l p(t_i) \end{aligned}$$

Now if we look at the two sums on the right, and given that $h_{i+1} = t_i$, all but the first term of the first sum and last term of the second sum pairwise cancel out. This leaves just $p(h_0) - p(t_l) = p(u) - p(v)$. Therefore

$$w'(P) = w(P) + p(u) - p(v) .$$

Since $p(v)$ and $p(u)$ are the same for any path from u to v , this does not change which path(s) from u to v is shortest, just the weight of that path. Therefore

$$\delta_{G'}(u, v) = \delta_G(u, v) + p(u) - p(v) ,$$

giving

$$\delta_G(u, v) = \delta_{G'}(u, v) - p(u) + p(v) .$$

□

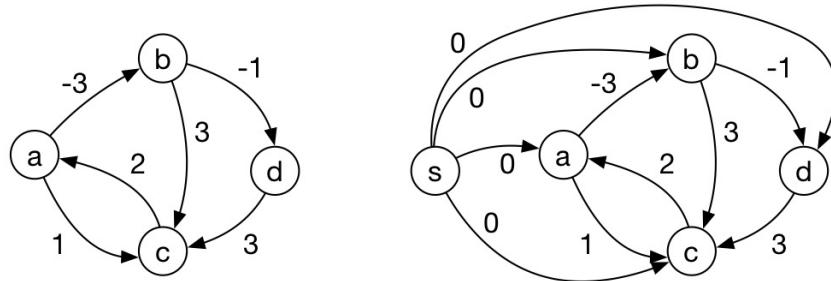
Algorithm 49.1 (Johnson's Algorithm). The pseudo-code for Johnson's algorithm is shown below. It starts by adding a dummy source vertex s to the graph and zero weight edges between the source and all other vertices. It then runs [Bellman-Ford's algorithm](#) on this graph and adjusts the weights by using the shortest paths computed. As established by Lemma 49.2 this adjustment eliminates all negative weights from the graph. Finally, it runs [Dijkstra's algorithm](#) for each vertex u and returns the distance from u to all other (reachable) vertices.

```

JohnsonAPSP ( $G = (V, E, w)$ ) =
  let
     $G^+ =$  Add a dummy vertex  $s$  to  $G$ ,
    and a zero weight edge from  $s$  to all  $v \in V$ .
     $D = BellmanFord(G^+, s)$ 
     $w'(u, v) = w(u, v) + D[u] - D[v]$  (*  $w'(u, v) \geq 0$  *)
     $G' = (V, E, w')$ 

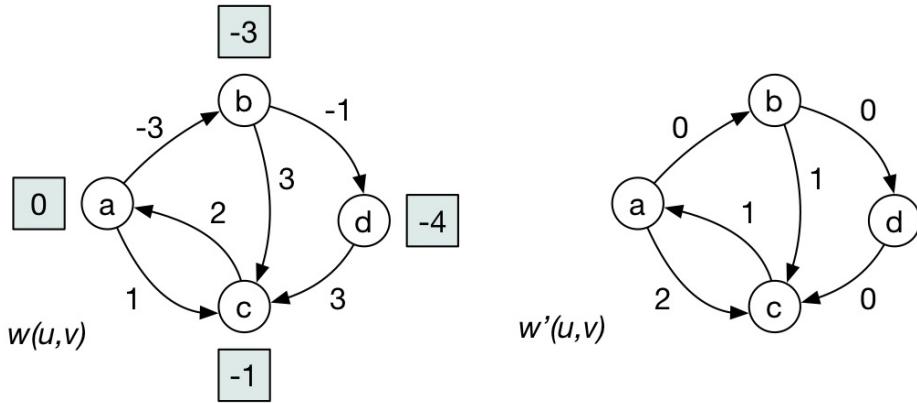
     $Dijkstra' u =$ 
      let  $\Delta_u = Dijkstra G' u$ 
      in  $\{(u, v) \mapsto (d - D[u] + D[v]) : (v \mapsto d) \in \Delta_u\}$  end
    in  $\bigcup_{u \in V} (Dijkstra' u)$  end
  
```

Example 49.1. As an example of Johnson's algorithm consider the graph on the left.



On the right we show the graph after adding the sources and the edges from it to all other vertices.

We now run Bellman-Ford's algorithm on this graph. It gives the distances shown in squares below on the left.



On the right we show the graph after the weights have been updated so that $w'(u, v) = w(u, v) + D[u] - D[v]$. This graph has no negative weight cycles.

To calculate the distance from (a, d) , for example, we can use $\delta_G(a, d) = \delta_{G'}(a, d) - D[a] + D[d]$, which gives the correct distance of $0 + 0 + (-4) = -4$.

All the $\delta_{G'}(u, v)$ are calculated using Dijkstra's.

Lemma 49.2 (Non-Negative Weights). For $p(v) = \delta_{G^+}(s, v)$, all edge weights $w'(u, v) = w(u, v) + p(u) - p(v)$ are non-negative.

Proof. Summary: For an original edge (u, v) with weight $-a$, the distance to v has to be at least a less than that to u . This difference $p(u) - p(v) \geq a$ will cancel out the negative edge.

Full proof: The sub-paths property tells us that:

$$\delta_{G'}(s, v) \leq \delta_{G'}(s, u) + w(u, v).$$

Since the shortest path cannot be longer than the shortest path through u , we have that:

$$\begin{aligned} 0 &\leq \delta_{G'}(s, u) + w(u, v) - \delta_{G'}(s, v) \\ &= w(u, v) + \delta_{G'}(s, u) - \delta_{G'}(s, v) \\ &= w(u, v) + p(u) - p(v) \end{aligned}$$

□

Note. Note that the *Dijkstra* function in [Johnson's algorithm](#) readjusts the path weights to zero out the impact of the potentials. This readjustment guarantees that the final distances are the correct distances.

Remark. Although we set the weights from the “dummy” source to each vertex to zero, any finite weight for each edge will do. In fact all that matters is that the distances from the source to all vertices in G' are non-infinite. Therefore if there is a vertex in the original graph G that can reach all other vertices then we can use it as the source and there is no need to add a new source.

Part X

Graph Contraction and Applications

Chapter 50

Introduction

Overview. In earlier chapters, we have mostly covered techniques for solving problems on graphs that were developed in the context of sequential algorithms. Some of the algorithms we considered were parallel while others were not. For example, we saw that [BFS](#) has some parallelism since each level can be explored in parallel but there was no parallelism in [DFS](#). There was no parallelism in [Dijkstra's algorithm](#), but there was plenty of parallelism in the [Bellman-Ford algorithm](#) and [Johnson's algorithm](#).

In this part of the book, we cover the “graph contraction” technique. This technique was specifically designed to be used in parallel algorithms and allows obtaining polylogarithmic span for certain graph problems. This chapter presents an overview of graph contraction. The following chapters present two specializations [Edge Contraction](#) and [Star Contraction](#) of graph contraction, and apply the technique to [graph connectivity](#).

Video: Motivation. <https://www.youtube.com/embed/bNO6LzXzcgQ>

1 Preliminaries

Video: Graph Theory for Graph Contraction. <https://www.youtube.com/embed/sP2T2IGBT1U>

Note. The material here and the followup chapters on graph contraction relies on the graph terminology introduced in the background [chapter on graph theory](#).

Definition 50.1 (Graph Partition). Given a graph G , a *graph partition* of G is a collection of graphs

$$H_0 = (V_0, E_0), \dots, H_{k-1} = (V_{k-1}, E_{k-1}),$$

such that $\{V_0, \dots, V_{k-1}\}$ is a set partition of V and H_0, \dots, H_{k-1} are [vertex-induced subgraphs](#) of G with respect to V_0, \dots, V_{k-1} .

We refer to each subgraph H_i as a *block* or *part* of G .

Definition 50.2 (Internal and Cut Edges). Given a partition $H_0 = (V_0, E_0), \dots, H_{k-1} = (V_{k-1}, E_{k-1})$ of a graph $G = (V, E)$, we define two kinds of edges: internal edges and cut edges.

- We call an edge $\{v_1, v_2\}$ an *internal edge*, if $v_1 \in V_i$ and $v_2 \in V_i$. Note that $\{v_1, v_2\} \in E_i$.
- We call an edge $\{v_1, v_2\}$ a *cut edge*, if $v_1 \in V_i$ and $v_2 \in V_j$ and $i \neq j$.

Exercise 50.1. One way to partition a graph is to make each connected component a block. What are the internal and cut edges in such a partition?

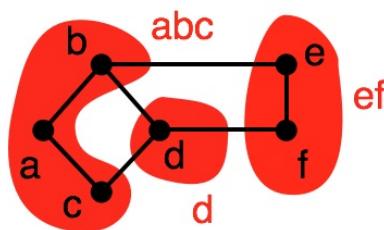
Solution. There are no cut edges between the partitions. All edges of the graph are internal edges.

Definition 50.3 (Partition Map). We sometimes describe a graph partition with a tuple consisting of

1. a set of labels for the blocks, and
2. a *partition map* that maps each vertex to the label of its block.

The labels can be chosen arbitrarily but it is usually conceptually and computationally easier to use a vertex inside a block as a representative for that block.

Example 50.1. The partition $\{\{a, b, c\}, \{d\}, \{e, f\}\}$ of the vertices $\{a, b, c, d, e, f\}$, defines three blocks as the corresponding [vertex-induced subgraphs](#).



The edges $\{a, b\}$, $\{a, c\}$, and $\{e, f\}$ are internal edges, and the edges $\{c, d\}$, $\{b, d\}$, $\{b, e\}$ and $\{d, f\}$ are cut edges.

By labeling the blocks 'abc', 'd' and 'ef', we can specify the graph partition with following partition map:

$$(\{abc, d, ef\}, \quad (50.1)$$

$$\{a \mapsto abc, b \mapsto abc, c \mapsto abc, d \mapsto d, e \mapsto ef, f \mapsto ef\}). \quad (50.2)$$

Instead of assigning a fresh label to each block, we can choose a representative vertex. For example, by picking a , d , and e as representatives, we can represent the partition above using the following partition map

$$(\{a, d, e\}, \quad (50.3)$$

$$\{a \mapsto a, b \mapsto a, c \mapsto a, d \mapsto d, e \mapsto e, f \mapsto e\}). \quad (50.4)$$

2 Graph Contraction

Video: Graph Contraction Technique. <https://www.youtube.com/embed/jaNhjt8F9ww>

Graph contraction is a [contraction technique](#) for computing properties of graphs in parallel. As a contraction technique, it is used to solve a problem instance by reducing it to a smaller instance of the same problem.

Graph contraction plays important role in parallel algorithm design, because divide-and-conquer can be difficult to apply to graph problems efficiently. Divide-and-conquer techniques usually require partitioning graphs into smaller graphs in a balanced fashion such that the number of cut edges is minimized. Because graphs can be highly irregular, they can be difficult to partition. In fact, graph partitioning problems are typically NP-hard.

Quotient Graph. The key idea behind graph contraction is to contract the input graph to a smaller *quotient graph*, solve the problem on the quotient graph, and then use that solution to construct the solution for the input graph. We can specify this technique as an inductive algorithm-design technique as follows.

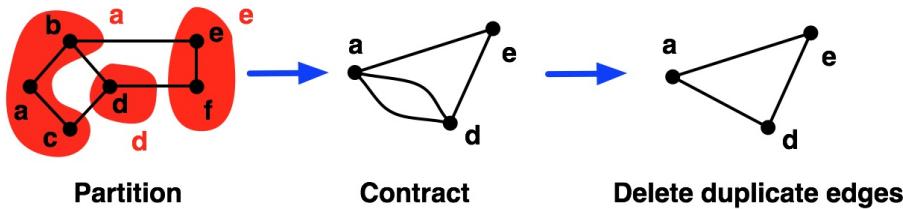
Definition 50.4 (Graph-Contraction Technique). Graph contraction technique has a base case and an inductive case. Each application of the inductive step is called a *round* of graph contraction. In a graph contraction, rounds are repeated until the graph is small, e.g., the graph has no remaining edges.

Base case: If the graph is small (e.g., it has no edges), then compute the desired result.

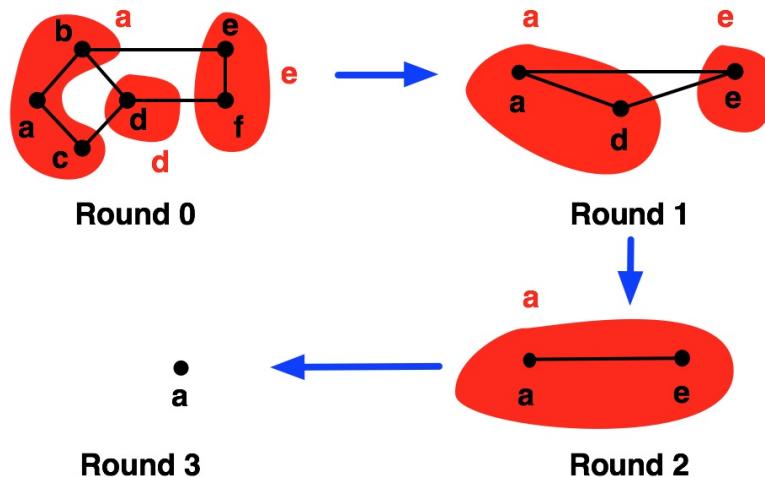
Inductive case:

- **Contraction step:** contract the graph into a smaller quotient graph.
 - Partition the graph into blocks.
 - Contract each block to a single super-vertex.
 - Drop internal edges.
 - Reroute cut edges to corresponding super-vertices.
- **Recursive step:** Recursively solve the problem for the quotient graph.
- **Expansion step:** By using the result for the quotient graph, compute the result for the input graph.

Example 50.2. One round of graph contraction:



Contracting a graph down to a single vertex in three rounds:



Construction of the Quotient Graph. To construct a quotient graph, we represent each block in the partition with a vertex, which we call a *super-vertex*. We then “map” the edges of the graph to the quotient graph. Consider each edge (u, v) in the graph.

- If the edge is an internal edge, then we skip the edge.
- If the edge is a cut edge, then we create a new edge between the super-vertices representing the blocks containing u and v .

Because there can be many cut edges between two blocks, this approach may create multiple edges between two super-vertices. We may remove duplicate edges or leave them in the graph, in which case we would be working with multigraphs. Either approach has its benefits and may, depending on the application, be preferable over the other.

Important. Graph contraction is guided by a graph partition, which leads to blocks whose vertices are disjoint. During the construction of the quotient graph, each vertex in the graph is therefore mapped to a unique vertex in the quotient graph.

Applying Graph Contraction. The ultimate goal of graph contraction technique is to reduce the size of the graph by a constant fraction (possibly in expectation) at each round of contraction. Depending on the graphs of interest many different graph-partition techniques can be used to achieve this goal. As described, the graph-contraction technique is generic in the kind of graph partition used. In the following chapters on [Edge Contraction](#) and [Star Contraction](#) we consider two techniques, edge partitioning and star partitioning, and the resulting graph-contraction algorithms.

Chapter 51

Edge Contraction

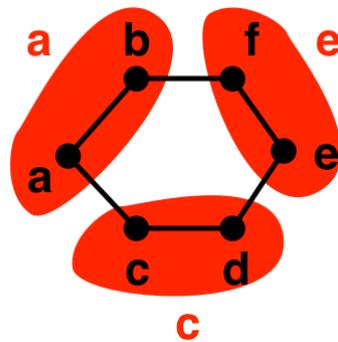
This section describes the edge partition and edge contraction. Edge contraction is an instance of a graph-contraction where blocks being contracted correspond to edges.

Video: Edge Contraction. <https://www.youtube.com/embed/Bcbq9-dqWPo>

1 Edge Partition

Definition 51.1 (Edge Partition). An *edge partition* is a [graph partition](#) where each block is either a single vertex or two vertices connected by an edge.

Example 51.1. An example edge partition in which every block consists of two vertices and an edge between them.



Exercise 51.1. Give an example graph whose edge partitions always contain a block that consists of a single vertex.

Solution. Any graph which has an isolated vertex, i.e., a vertex with no incident edges would work.

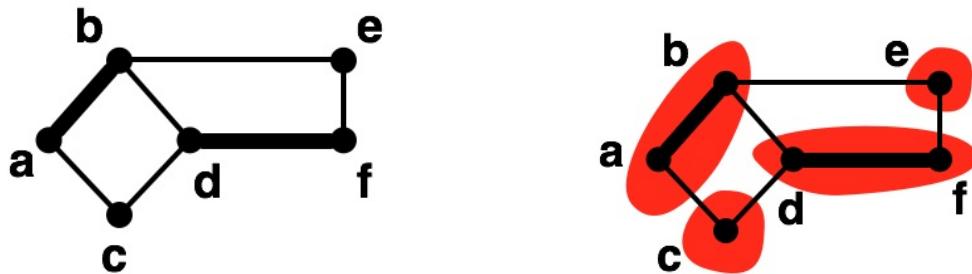
Edge Partitions and Vertex Matching. Finding an edge partition of a graph is closely related to the problem of finding an independent edge set or a vertex matching. A vertex matching in a graph is a subset of the edges that do not share an endpoint, i.e., no two edges are incident on the same vertex. We can construct an edge partition from a vertex matching by constructing a block for each edge in the matching and placing all the remaining vertices into their own singleton blocks.

Definition 51.2 (Vertex Matching). A *vertex matching* for an undirected graph $G = (V, E)$ is a subset of edges $M \subseteq E$ such that no two edges in M are incident on the same vertex. In other words, each vertex in M have degree at most 1.

The problem of finding the largest vertex matching for a graph is called the *maximum vertex matching* problem.

Algorithms for Maximum Vertex Matching. Maximum Vertex Matching is a well-studied problem and many algorithms have been proposed, including one that can solve the problem in $O(\sqrt{|V||E|})$ work.

Example 51.2 (Vertex Matching). A vertex matching for a graph (highlighted edges) and the corresponding blocks.



The vertex matching defines four blocks (circled), two of them defined by the edges in the matching, $\{a, b\}$ and $\{d, f\}$, and two of them are the unmatched vertices c and e .

Note. For edge contraction, we do not need a maximum matching but one that it is sufficiently large.

Algorithm 51.3 (Greedy Vertex Matching). We can use a greedy algorithm to construct a vertex matching by iterating over the edges while maintaining an initially empty matching M . The greedy algorithm considers each edge and proceeds as follows:

- if no edge in M is already incident on its endpoints, then the algorithm adds the edge to M ,
- otherwise, the algorithm tosses away the edge.

Exercise 51.2. Does the greedy vertex matching algorithm always returns a maximum vertex matching?

Solution. No.

Exercise 51.3. Prove that the greedy algorithm finds a solution within a factor two of optimal.

Exercise 51.4. Is the greedy algorithm parallel?

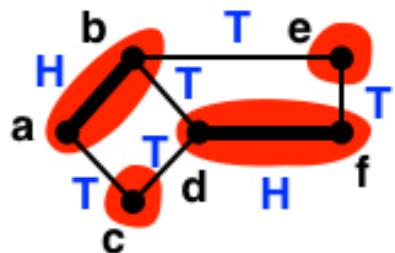
Solution. The greedy algorithm is sequential, because each decision depends on previous decisions.

Randomized Symmetry Breaking. To find a [vertex matching](#) in parallel, we want to make local and parallel decisions at each vertex independent of other vertices. One possibility is for each vertex to select one of its neighbors arbitrarily but in some deterministic fashion. Such a selection can be made in parallel but there is one problem: multiple vertices might select the same vertex to match with.

We therefore need a way to *break the symmetry* that arises when two vertices try to match with the same vertex. To this end, we can use randomization. There are several different ways to use randomization but they are all essentially the same and yield the same bounds with a constant factor.

Algorithm 51.4 (Parallel Vertex Matching). To compute a vertex matching the [parallel vertex matching algorithm](#) flips a coin for each edge in parallel. The algorithm then selects an edge (u, v) and matches u and v , if the coin for the edge comes up heads and all the edges incident on u and v flip tails.

Example 51.3 (Parallel Vertex Matching). An example run of the parallel vertex matching algorithm.



Exercise 51.5. Prove that the algorithm produces a vertex matching, i.e., it guarantees that a vertex is matched with at most one other vertex.

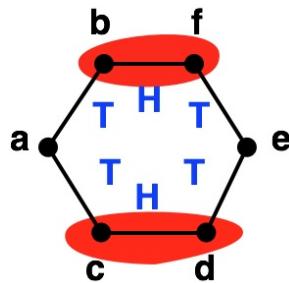
1.1 Analysis of Parallel Edge Partition

We analyze the effectiveness of the parallel [edge partition algorithm](#) in selecting a matching that consists of as many edge blocks (equivalently as few singleton blocks) as possible. We first consider cycle graphs and then general graphs.

1.1.1 Cycle Graphs

Probability of Selecting an Edge in a Cycle. We want to determine the probability that an edge is selected in a cycle, where each vertex has exactly two neighbors. Because the coins are flipped independently at random, and each vertex has degree two, the probability that an edge picks heads and its two adjacent edges pick tails is $\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}$.

Example 51.4 (Edge Partition of a Cycle). A graph consisting of a single cycle.



Each edge flips a coin that comes up either heads (H) or tails (T). We select an edge if it turns up heads and all other edges incident on its endpoints are tails. In the example the edges $\{c, d\}$ and $\{b, f\}$ are selected.

Expected Number of Edges Selected. To analyze the number of edges (blocks) selected in expectation, let R_e be an indicator random variable denoting whether e is selected or not, that is $R_e = 1$ if e is selected and 0 otherwise. Recall that the expectation of indicator random variables is the same as the probability it has value 1 (true). Therefore we have $E[R_e] = 1/8$. Thus summing over all edges, we conclude that expected number of edges selected is $\frac{m}{8}$ (note, $m = n$ in a cycle graph). Thus we conclude that in expectation, a constant fraction ($\frac{1}{8}$) of the edges are selected to be their own blocks.

Exercise 51.6. Modify the algorithm to improve the expected number of edges selected.

Improving the Expectation. There are several ways to improve the number of selected edges. One way is for each vertex to pick one of its neighbors and to select an edge (u, v) if

it was picked by both u and v . In the case of a circle, this increases the expected number of selected edges to $\frac{m}{4}$.

Another way is let each edge pick a random number in some range and then select an edge if it is the local maximum, i.e., it picked the highest number among all the edges incident on its end points. This increases the expected number of selected edges to $\frac{m}{3}$.

1.1.2 Star Graphs

Limitation of Edge Partition. Although our edge partition algorithm works quite well on cycle graphs, it does not work well for arbitrary graphs. The problem is in an edge partition, only one edge incident on a vertex can be its own block. Therefore if there is a vertex with high degree, then only one of its edges can be selected. Star graphs are a canonical example of such graphs, although there are many others.

Definition 51.5 (Star Graph). A *star graph* $G = (V, E)$ is an undirected graph with

- a *center* vertex $v \in V$, and
- a set of edges E that attach v directly to the rest of the vertices, called *satellites*, i.e., $E = \{ \{v, u\} : u \in V \setminus \{v\} \}$.

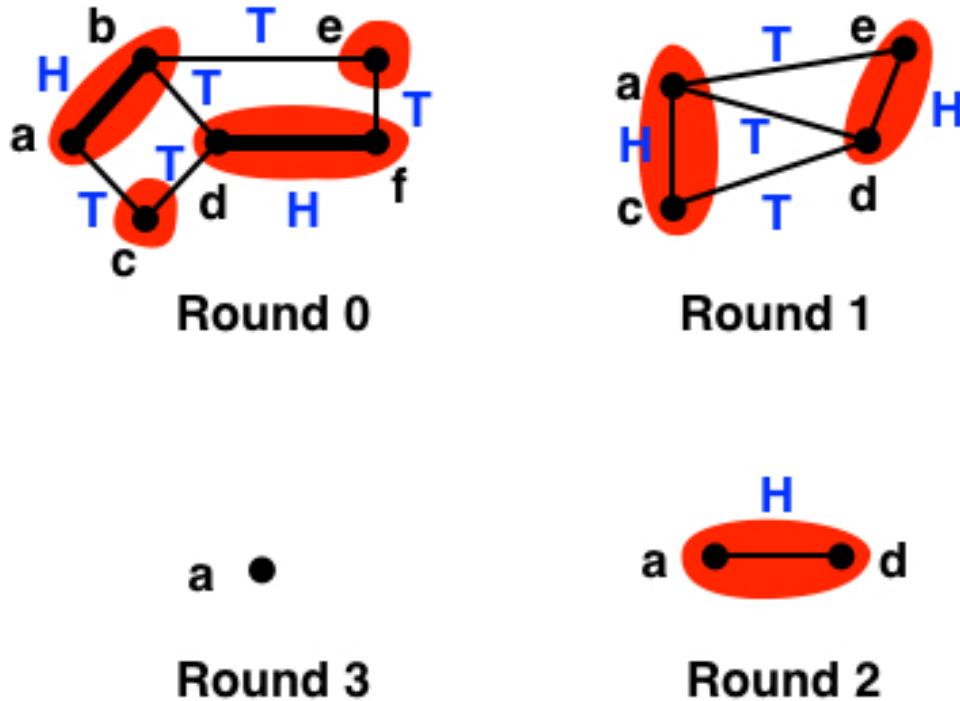
Example 51.5. The following are star graphs:

- a single vertex, and
- a single edge.

2 Edge Contraction

Algorithm 51.6 (Parallel Edge Contraction). Parallel edge contraction algorithm is a specialization of the graph contraction technique that uses the parallel [vertex matching algorithm](#) to partition the graph for contraction.

Example 51.6 (Edge contraction). An example parallel edge contraction illustrated.



Analysis of Edge Contraction. The [analysis of edge partition](#) established that using edge partition, we are able to select in expectation $\frac{1}{8}$ of the edges as their own blocks if the graph is a cycle. Therefore, after one round of contraction, the number of vertices and edges in a cycle decrease by an expected constant fraction.

In [randomized algorithms chapter](#), we showed that if each round of an algorithm reduces the size by a constant fraction in expectation, and if the random choices in the rounds are independent, then the algorithm will finish in $O(\lg n)$ rounds with high probability. Recall that all we needed to do is multiply the expected fraction that remain across rounds and then use Markov's inequality to show that after some $k \lg n$ rounds the probability that the problem size is at least 1 is very small. For a cycle graph, this technique leads to an algorithm for graph contraction with linear work and $O(\lg^2 n)$ span.

Analysis for Star Graphs. Edge contraction works quite poorly on other graphs such as star graphs, and can result in a partition with many singleton blocks. This is because in an edge partition, only one of the edges incident on a vertex can be its own block (Section 1.1.2), leading to a poor contraction ratio. Edge contraction therefore is not effective for general graphs.

Chapter 52

Star Contraction

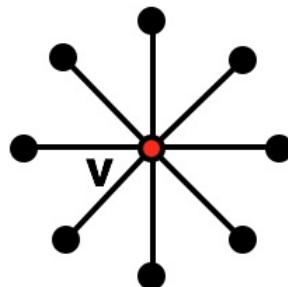
This chapter covers star partition and star contraction, an efficient and parallel graph-contraction technique for general graphs.

1 Star Partition

In an [edge partition](#), if an edge incident on a vertex v is selected as a block, then none of the other edges incident on v can be their own block. This limits the effectiveness of the edge partition technique, because it is unable to contract graphs with high-degree vertices significantly. In this section, we describe an alternative technique, star partition, that does not have this limitation.

Definition 52.1 (Star Partition). A *star partition* of a graph G is a partition of G where each block is vertex-induced subgraph with respect to a [star graph](#).

Example 52.1. Consider star graph with center v and eight satellites.

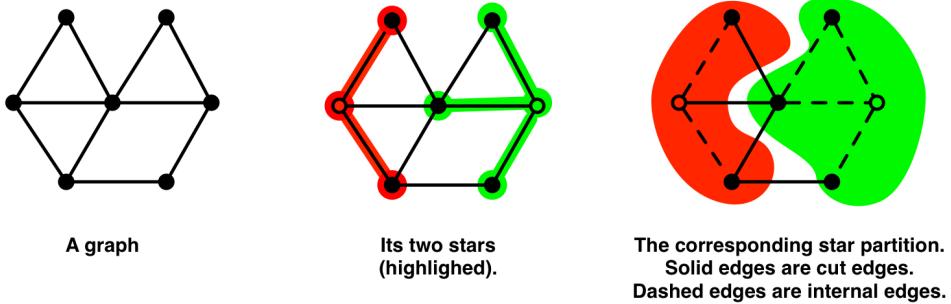


- A partition consisting of the whole graph is a star partition, where the only block is

the graph itself, induced by the star graph.

- A partition where each block is an isolated vertex is a star partition, because each block is a vertex-induced subgraph of a single vertex, which is a star.

Example 52.2. Consider the graph shown below on the left. To partition this graph, we first find two disjoint stars, which are highlighted. Each star induces a block consisting of its vertices and the corresponding edges of the graph. These two blocks form a star partition of the graph. Note that in a star partition, a block might not be a star.



Constructing a Star Partition (Sequential). We can construct a star partition sequentially by iteratively adding stars until the vertices are exhausted as follows.

- Select an arbitrary vertex v from the graph and make v the center of a star.
- Attach as satellites all the neighbors of v in the graph.
- Remove v and its satellites from the graph.

Computing a Star Partition (Parallel). We can construct a star partition in parallel by making local independent decisions for each vertex, and using randomization to break symmetry. One approach proceeds as follows.

- Flip a coin for each vertex.
- If a vertex flips heads, then it becomes the center of a star.
- If a vertex flips tails, then there are two cases.
 - The vertex has a neighbor that flips heads. In this case, the vertex selects the neighbor (breaking ties arbitrarily) and becomes a satellite.
 - The vertex doesn't have a neighbor that flips heads. In this case, the vertex becomes a center.

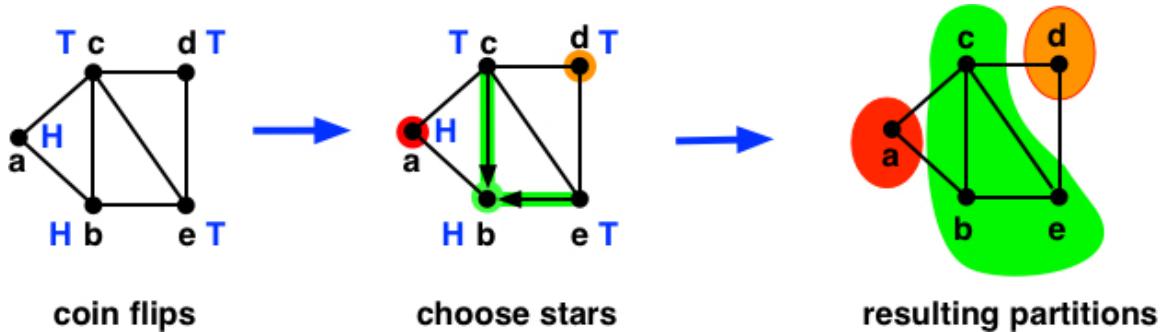
Note that if a vertex doesn't have a neighbor (it is "isolated"), then it will always become a center.

Definition 52.2 (Isolated Vertices). We say that a vertex is *isolated* in a graph if it doesn't have a neighbor.

Note. The [parallel approach](#) to star partition is not optimal, because it might not always create the smallest number of stars. This is acceptable for us, because we only need to reduce the size of the graph by some constant factor.

Example 52.3 (Randomized Star Partition). The example below illustrates how we may partition a graph using the parallel star partition algorithm described above. Vertices a and b , which flip heads, become centers. Vertices c and e , which flipped tails, attempt to become satellites by finding a center among their neighbors, breaking ties arbitrarily. If a vertex does not have a neighbor that is a center (flipped heads), then it becomes a singleton star (e.g., vertex d).

The resulting star partition has three stars: the star with center a (with no satellites), the star with center b (with two satellites), and the singleton star d . The star partition thus yields three blocks, which are defined by the subgraphs induced by each star.



Algorithm 52.3 (Parallel Star Partition). To specify the star-partition algorithm, we need a source of randomness. We assume that each vertex has access to a random coin flip

heads $v : V \times \mathbb{Z} \rightarrow \mathbb{B}$,

which returns `true` if the vertex v flips heads and `false` otherwise for this partition.

The function `starPartition`, whose pseudo-code is given below, takes as argument a graph and a round number, and returns a graph partition specified by a set of centers and a partition map from all vertices to centers.

The algorithm starts by flipping a coin for each vertex and selecting the edges that point from tails to heads—this gives the set of edges TH . In this set of edges, there can be multiple edges from the same non-center. Since we want to choose one center for each satellite, we remove duplicates in Line 6, by creating a set of singleton tables and merging

them, which selects one center per satellite. This completes the selection of satellites and their centers.

Next, the algorithm determines the set of centers as all the non-satellite vertices. To complete the process, the algorithm maps each center to itself (Line 10). These operations effectively promote unmatched non-centers to centers, forming singleton stars, and matches all centers with themselves. Finally, the algorithm constructs the [partition map](#) by uniting the mapping for the satellites and the centers.

```

1  starPartition  $G = (V, E) =$ 
2  let
3      (* Find the arcs from satellites to centers. *)
4       $TH = \{(u, v) \in E \mid \neg(\text{heads } u) \wedge (\text{heads } v)\}$ 
5      (* Partition map: satellites map to centers *)
6       $P_s = \bigcup_{(u, v) \in TH} \{u \mapsto v\}$ 
7      (* Centers are non-satellite vertices *)
8       $V_c = V \setminus \text{domain}(P_s)$ 
9      (* Map centers to themselves *)
10      $P_c = \{u \mapsto u : u \in V_c\}$ 
11     in
12      $(V_c, P_s \cup P_c)$ 
13     end

```

Note. Most machines don't have true sources of randomness, the function *heads* is therefore usually implemented with a pseudorandom number generator or with a good hash function.

In the algorithm, Line 6 creates a set of singleton tables and merges them. This can be implemented using sets and tables as follows.

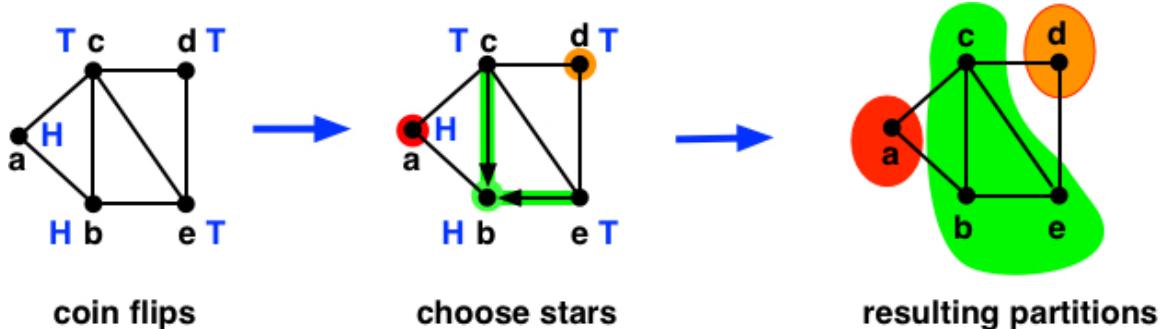
```

Set.reduce  (Table.union (lambda (x, y) . x))
             $\emptyset$ 
             $\{\{u \mapsto v\} : (u, v) \in TH\}$ 

```

Note that we supply to the *union* operation a function that selects the first of the two possibilities; this is an arbitrary choice and we could have favored the second.

Example 52.4. Consider the graph below and the random coin flips.



The star-partition algorithm proceeds on this example as follows. First, it computes

$$TH = \{(c, a), (c, b), (e, b)\},$$

as the edges from satellites to centers. Now, it converts each edge into a singleton table, and merges all the tables into one table, which is going to become a part of the partition map:

$$P_s = \{c \mapsto b, e \mapsto b\}.$$

Note that the edge (c, a) has been removed since when uniting the tables, we select only one element for each key in the domain. Now for all remaining vertices $V_c = V \setminus \text{domain}(P) = \{a, b, d\}$ we map them to themselves, giving:

$$P_c = \{a \mapsto a, b \mapsto b, d \mapsto d\}.$$

The vertices in P_c are the centers. Finally we merge P and P_c to obtain the partition map

$$P_s \cup P_c = \{a \mapsto a, b \mapsto b, c \mapsto b, d \mapsto d, e \mapsto b\}.$$

Implementation. Suppose that we are given an enumerable graph with n vertices and m edges. We can represent the graph using [an edge set representation](#) and represent the sets with sequences. This means that we have a sequence of vertices and a sequence of edges.

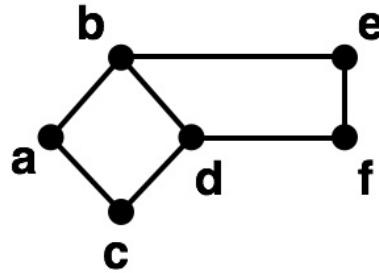
This representation enables a relatively clean implementation of the [star-partition algorithm](#), as shown by the pseudo-code below. The implementation follows the pseudo-code for the algorithm but is able to compute the satellites and centers compactly by using a sequence *inject* operation. The implementation first constructs a vertex sequence V' where each vertex maps to itself. It then constructs a sequence TH of “updates” from vertices that flip heads into tails, and inject TH into the sequence of vertices V' . The resulting sequence P maps each vertex that flipped tails to a center, if the vertex has a neighbor that flipped heads. The sequence P ensures that a vertex that has flipped heads remains unaffected by the injection, e.g., if vertex i has flipped heads, then $P[i] = i$. We can thus compute set of centers by filtering over P and use the sequence P to represent the partition map for

satellites and centers jointly.

```
starPartition (G = (V, E)) =
  let
    V' = ⟨ j : 0 ≤ j < |V| ⟩
    TH = ⟨ (u, v) ∈ E | ¬(heads u) ∧ (heads v) ⟩
    P = Seq.inject V' TH
    VC = ⟨ j ∈ P | P[j] = j ⟩
  in (VC, P) end
```

Reminder (Edge-Set Representation). The edge set representation of a graph consists of a set of vertices and a set of edges, where each undirected edge is represented with two arcs, one in each direction.

Example 52.5. The edge-set representation of an undirected graph is shown below.



$$\begin{aligned} V &= \{a, b, c, d, e, f\} \\ E &= \{(a, b), (b, a), (b, d), (b, e), (e, b), (d, b), (d, f), (a, c), \\ &\quad (c, a), (c, d), (d, c), (d, f), (f, d), (e, f), (f, e)\} \end{aligned}$$

1.1 Analysis of Star Partition

Theorem 52.1 (Cost of Star Partition). Based on the array-based cost specification for sequences, the cost of *starPartition* is

$$O(n + m)$$

work and

$$O(\lg n)$$

span for a graph with n vertices and m edges.

Exercise 52.1. Prove the theorem.

Number of Satellites. Let us also bound the number of satellites found by *starPartition*. Note first that there is a one-to-one mapping between the satellites and the set P_s computed by the algorithm. The following lemma establishes that on a graph with n non-isolated vertices, the number of satellites is at least $n/4$ in expectation. As we will see this means that we can use star partition to perform graph contraction with logarithmic span.

Lemma 52.2 (Number of Satellites). For a graph G with n_\bullet non-isolated vertices, the expected number of satellites in a call to *starPartition* (G, i) with any i is at least $n_\bullet/4$.

Proof. For any vertex v , let H_v be the event that a vertex v comes up heads, T_v that it comes up tails, and R_v that $v \in \text{domain}(P)$ (i.e, it is a satellite). Consider any non-isolated vertex $v \in V(G)$. By definition, we know that a non-isolated vertex v has at least one neighbor u . So, we know that $T_v \wedge H_u$ implies R_v , since if v is a tail and u is a head v must either join u 's star or some other star. Therefore, $\mathbf{P}[R_v] \geq \mathbf{P}[T_v] \mathbf{P}[H_u] = 1/4$. By the linearity of expectation, the expected number of satellites is

$$\begin{aligned} \mathbf{E} \left[\sum_{v:v \text{ non-isolated}} \mathbb{I}\{R_v\} \right] &= \sum_{v:v \text{ non-isolated}} \mathbf{E}[\mathbb{I}\{R_v\}] \\ &\geq n_\bullet/4. \end{aligned}$$

The final inequality follows because we have n_\bullet non-isolated vertices and because the expectation of an indicator random variable is equal to the probability that it takes the value 1. \square

2 Star Contraction

Definition 52.4 (Star Contraction). *Star contraction* is an instance of graph contraction that uses star partitions to contract the graph.

Algorithm 52.5 (Star Contraction). The pseudo-code below gives a higher-order star-contraction algorithm. The algorithm takes as arguments the graph G and two functions:

- *base* function specifies the computation in the base case, and
- *expand* function computes the result for the larger graph from the quotient graph.

In the base case, the graph contains no edges and the function *base* is called on vertex set.

In the recursive case, the graph is partitioned by a call to *star partition* (Line 6), which returns the set of (centers) super-vertices V' and P the *partition map* mapping every $v \in V$ to a $v' \in V'$. The set V' defines the super-vertices of the quotient graph. Line 7 computes the edges of the quotient graph by routing the end-points of each edge in E to the corresponding super-vertices in V' as specified by partition-map P . Note that the filter

$P[u] \neq P[v]$. removes self edges. The algorithm then recurs on the quotient graph (V', E') . The algorithm then computes the result for the whole graph by calling the function *expand* on the result of the recursive call R .

```

1  starContract base expand (G = (V, E)) =
2  if |E| = 0 then
3      base (V)
4  else
5      let
6          (V', P) = starPartition (V, E)
7          E' = {(P[u], P[v]) : (u, v) ∈ E | P[u] ≠ P[v]}
8          R = starContract base expand (V', E')
9      in
10     expand (V, E, V', P, R)
11 end

```

Theorem 52.3 (Work and Span of Star Contraction). For a graph $G = (V, E)$, we can contract the graph into a number of isolated vertices in $O((|V| + |E|) \lg |V|)$ work and $O(\lg^2 |V|)$ span.

Proof structure and assumptions. For the proof, we will consider work and span separately and assume that

- function *base* has constant span and linear work in the number of vertices passed as argument, and
- function *expand* has linear work and logarithmic span in the number of vertices and edges at the corresponding step of the contraction.

Span of Star Contraction. Let n_\bullet be the number of non-isolated vertices. In star contraction, once a vertex becomes isolated, it remains isolated until the final round, since contraction only removes edges. Let n'_\bullet denote the number of non-isolated vertices after one round of star contraction. We can write the following recurrence for the span of star contraction.

$$S(n_\bullet) = \begin{cases} S(n'_\bullet) + O(\lg n) & \text{if } n_\bullet > 0 \\ 1 & \text{otherwise.} \end{cases}$$

Observe that $n'_\bullet = n_\bullet - X$, where X is the number of satellites (as defined earlier in the lemma about *starPartition*), which are removed at a step of contraction. Because $\mathbf{E}[X] = n_\bullet/4$, $\mathbf{E}[n'_\bullet] = 3n_\bullet/4$. This is a familiar recurrence, which we know solves to $O(\lg^2 n_\bullet)$, and thus $O(\lg^2 n)$, in expectation.

Work of Star Contraction. For work, we would like to show that the overall work is linear, because we might hope that the graph size is reduced by a constant fraction on each round. Unfortunately, this is not the case. Although we have shown that star contraction can remove a constant fraction of the non-isolated vertices in one round, we have not bounded the number of edges removed.

Because removing a satellite also removes the edge that attaches it to its star's center, each round removes at least as many edges as vertices. But this does not help us bound the number of edges removed by a linear function of m , because there can be as many as n^2 edges in the graph.

To bound the work, we will consider non-isolated and isolated vertices separately. Let n'_\bullet denote the number of non-isolated vertices after one round of star contraction. For the non-isolated vertices, we have the following work recurrence:

$$W(n_\bullet, m) \leq \begin{cases} W(n'_\bullet, m) + O(n_\bullet + m) & \text{if } n_\bullet > 1 \\ 1 & \text{otherwise.} \end{cases}$$

This recursion solves to

$$\mathbf{E}[W(n_\bullet, m)] = O(n_\bullet + m \lg n_\bullet) = O(n + m \lg n).$$

To bound the work on isolated vertices, we note that there at most n of them at each round and thus, the additional work is $O(n \lg n)$.

We thus conclude that the total work is

$$O((n + m) \lg n).$$

Note. Consider as an example a star contraction where n and m have the following values in each round.

round	vertices	edges
1	n	m
2	$n/2$	$m - n/2$
3	$n/4$	$m - 3n/4$
4	$n/8$	$m - 7n/8$

It is clear that the number of edges does not drop below $m - n$, so if there are $m > 2n$ edges to start with, the overall work will be $O(m \lg n)$.

Chapter 53

Graph Connectivity

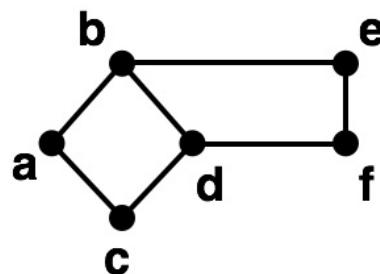
This chapter presents a parallel graph connectivity algorithm that uses graph contraction (more specifically star contraction).

1 Preliminaries

Definition 53.1 (The Graph Connectivity Problem). Given an undirected graph $G = (V, E)$, the *graph-connectivity problem* requires finding all of the connected components of G by specifying the set of vertices in each component.

Assumption (Graph Representation). Throughout this chapter, we use an edge-set representation for graphs, where every edge is represented as a pair of vertices, in both orders. This is effectively equivalent to a directed graph representation of undirected graphs with two arcs per edge. As usual we use n and m to denote the number of vertices and edges respectively.

Example 53.1. The edge-set representation of an undirected graph is shown below.



$$\begin{aligned}
 V &= \{a, b, c, d, e, f\} \\
 E &= \{(a, b), (b, a), (b, d), (b, e), (e, b), (d, b), (d, f), (a, c), \\
 &\quad (c, a), (c, d), (d, c), (d, f), (f, d), (e, f), (f, e)\}
 \end{aligned}$$

2 Algorithms for Connectivity

Sequential Algorithms for Connectivity. The graph connectivity problem can be solved by using graph search as follows.

- Start at any vertex and find, using DFS or BFS, all vertices reachable from that vertex and mark them visited. This creates the first connected component.
- Select another vertex, and if it has not already been visited, then search from that vertex to create the second component. Repeat until all the vertices are considered.

This approach leads to perfectly sensible sequential algorithms for graph connectivity, but the resulting algorithms have large span and are therefore poor parallel algorithms.

DFS is a purely sequential algorithm and has span $\Omega(m)$. BFS can yield some parallelism but still the span of BFS is lower bounded by the diameter of a component, which can be as large as $n - 1$, e.g., a “chain” of n vertices has diameter $n - 1$. Even when the diameter of the graph is small, the span can be high, because we have to iterate over the components sequentially. The span is lower bounded by the number of components, which can be large.

Algorithm 53.2 (Component Count). The pseudo-code below shows a graph-contraction based algorithm for determining the number of connected components in a graph. The call to *starPartition* (Algorithm 52.3) on Line 6 returns the set of (centers) super-vertices V' and a table P mapping every $v \in V$ to a $v' \in V'$.

The set V' defines the super-vertices of the quotient graph. Line 7 completes the computation of the quotient graph.

- It computes the edges of the quotient graph by routing the end points of each edge to the corresponding super-vertices in V' , which is specified by the table P ;
- It removes all self edges via the filter $P[u] \neq P[v]$.

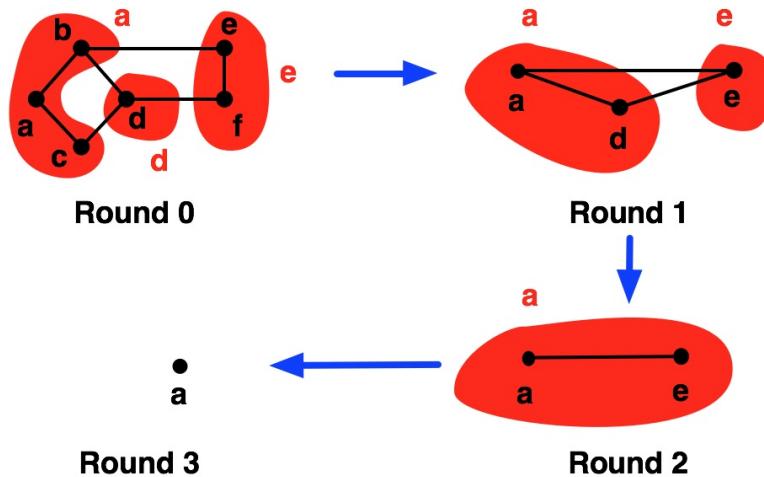
The algorithm then recursively computes the number of connected components in the quotient graph. Recursion bottoms out when the graph contains no edges, where the number of components is equal to the number of vertices.

```

1   countComponents ( $G = (V, E)$ ) =
2     if  $|E| = 0$  then
3        $|V|$ 
4     else
5       let
6          $(V', P) = \text{starPartition} (V, E)$ 
7          $E' = \{(P[u], P[v]) : (u, v) \in E \mid P[u] \neq P[v]\}$ 
8          $R = \text{countComponents} (V', E')$ 
9       in
10       $R$ 
11    end

```

Example 53.2. Consider an execution of *countComponents* that contracts the graph as follows.



The values of V' , P , and E' after each round of the contraction is shown below.

$$\begin{array}{ll}
 \text{Round 1} & \begin{array}{l} V' = \{a, d, e\} \\ P' = \{a \mapsto a, b \mapsto a, c \mapsto a, d \mapsto d, e \mapsto e, f \mapsto e\} \\ E' = \{(a, e), (e, a), (a, d), (d, a), (d, e), (e, d)\} \end{array} \\
 \text{Round 2} & \begin{array}{l} V' = \{a, e\} \\ P' = \{a \mapsto a, d \mapsto a, e \mapsto e\} \\ E' = \{(a, e), (e, a)\} \end{array} \\
 \text{Round 3} & \begin{array}{l} V' = \{a\} \\ P' = \{a \mapsto a, e \mapsto a\} \\ E' = \{\} \end{array}
 \end{array}$$

Exercise 53.1. Express *countComponents* in terms of higher order function *starContract* (Algorithm 52.5) by specifying the functions *base* and *expand*.

Computing Components. We can modify [algorithm for counting components](#) to compute the components themselves. The idea is to construct recursively a mapping from vertices to their components. The [algorithm below](#) implements this idea.

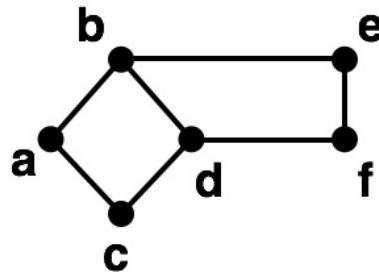
Algorithm 53.3 (Graph Connectivity). The algorithm below computes the connected components of the input graph G and returns a tuple consisting of 1) a representative for each component, and 2) a mapping from the vertices in the graph to the representative of their component.

```

1  connectedComponents ( $G = (V, E)$ ) =
2    if  $|E| = 0$  then
3       $(V, \{v \mapsto v : v \in V\})$ 
4    else
5      let
6         $(V', P) = \text{starPartition} (V, E)$ 
7         $E' = \{(P[u], P[v]) : (u, v) \in E \mid P[u] \neq P[v]\}$ 
8         $(V'', C) = \text{connectedComponents} (V', E')$ 
9      in
10      $(V'', \{u \mapsto C[v] : (u \mapsto v) \in P\})$ 
11   end

```

Example 53.3. Applying `connectedComponents` to the following graph

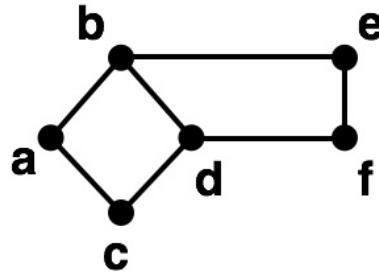


might return:

$(\{a\}, \{a \mapsto a, b \mapsto a, c \mapsto a, d \mapsto a, e \mapsto a, f \mapsto a\})$

This is because there is a single component and all vertices will map to that component label. In this case a was picked as the representative, but any of the initial vertices is a valid representative, in which case all vertices would map to it.

Example 53.4. Consider the following graph.



Suppose that `starPartition` returns:

$$\begin{aligned} V' &= \{a, d, e\} \\ P' &= \{a \mapsto a, b \mapsto a, c \mapsto a, d \mapsto d, e \mapsto e, f \mapsto e\}. \end{aligned}$$

This pairing corresponds to the case where a, d and e are chosen as centers.

Because the graph is connected, the recursive call to `connectedComponents` (V', E') will map all vertices in V' to the same vertex. Let's say this vertex is a giving:

$$\begin{aligned} V'' &= \{a\} \\ P'' &= \{a \mapsto a, d \mapsto a, e \mapsto a\}. \end{aligned}$$

Now $\{u \mapsto P'[v] : (u \mapsto v) \in P\}$ will for each vertex-super-vertex pair in P , look up what that super-vertex got mapped to in the recursive call. For example, vertex f maps to vertex e in P so we look up e in P' , which gives us a so we know that f is in the component a . Overall the result is:

$$\{a \mapsto a, b \mapsto a, c \mapsto a, d \mapsto a, e \mapsto a, f \mapsto a\}.$$

Note. The only difference between the [algorithm for counting components](#) and the [algorithm for computing the components](#) is the base case, and “expansion step” (Definition ??) on Line 10 of Algorithm 53.3.

In the base case instead of returning the size of V returns all vertices in V along with a mapping from each one to itself. This is a valid answer since if there are no edges each vertex is its own component. In the inductive case, before returning from the recursion, Line 10 updates the mapping C from vertices to their representatives by looking up the component that the super-vertex of a vertex belongs to, which is given by P . This involves looking up $C[v]$ for every $(u \mapsto v) \in P$. If we view a mapping as a function, then the expansion step is equivalent to function composition, i.e., $C \circ P$.

Exercise 53.2. Express `countComponents` in terms of higher order function `starContract` (Algorithm 52.5) by specifying the functions `base` and `expand`.

Exercise 53.3. What is the work and span of the [algorithm for counting components](#) ? Explain your choice of the representation for the graph. What happens if you choose a different representation?

Exercise 53.4. What is the work and span of the [algorithm for computing the components](#) ? Explain your choice of the representation for the graph. What happens if you choose a different representation?

Part XI

Minimum Spanning Trees

Chapter 54

Introduction

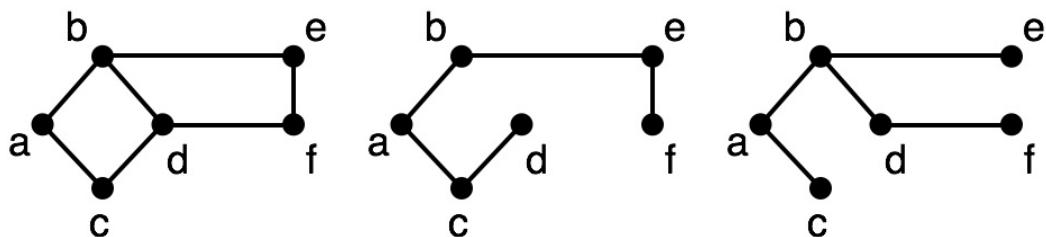
This chapter defines the Minimum Spanning Tree (MST) problem and introduces a key lemma called the “light-edge lemma” that nearly all algorithms for solving the problem utilizes.

1 Spanning Trees

Recall that we say that an undirected graph is a forest if it has no cycles and a tree if it is also connected. Given a connected, undirected graph, we might want to identify a subset of the edges that form a tree, while including all the vertices. We call such a tree a spanning tree.

Definition 54.1 (Spanning Tree). For a connected undirected graph $G = (V, E)$, a spanning tree is a tree $T = (V, E')$ with $E' \subseteq E$.

Example 54.1. A graph (top), and two spanning trees for it.



Exercise 54.1. How many edges does a spanning tree have?

Solution. A graph can have many spanning trees, but all have $|V|$ vertices and $|V| - 1$ edges.

Lemma 54.1 (Spanning Trees Edge Replacement). Let $G = (V, E)$ be a connected graph and let T be a spanning tree of G . Consider some edge $e = \{u, v\} \in E$ that is not in T . Let e' be any edge on the path from u to v in T and let the tree T' be $T \setminus \{e'\} \cup \{e\}$, that is a tree obtained by swapping e for e' . The tree T' is a spanning tree of G .

Proof. Consider any path in T that uses $e' = \{u', v'\}$. We can re-route this path to use $e = \{u, v\}$ instead and thus the path is a valid path in T' . Thus, T' is connected, and is acyclic. Furthermore T' has exactly the same number of nodes and edges as T and thus is a spanning tree. \square

Sequential Algorithms for Spanning Trees. We can find a spanning tree of a graph by using graph search.

- A DFS-tree is a spanning tree, because it includes a path from the source to all the vertices in the graph.
- Similarly, we can construct a spanning tree based on BFS by including in the spanning tree each edge that leads to the discovery of an unvisited vertex.

DFS and BFS are work-efficient algorithms for computing spanning trees but as their span can be large. DFS in particular is sequential. The span of BFS can be as large as the diameter of the graph, which can be large.

Parallel Algorithms for Spanning Trees. We can compute a spanning tree for a graph by using [graph contraction](#) and, specifically [star contraction](#). The idea is to use star contraction and add all the edges that are selected to define the stars to the spanning tree. Because graph contraction has poly-logarithmic span in expectation, this approach yields a good parallel algorithm, though work can be suboptimal.

Exercise 54.2. Give an algorithm for computing the spanning tree of a graph using star contraction. Prove that the algorithm is correct.

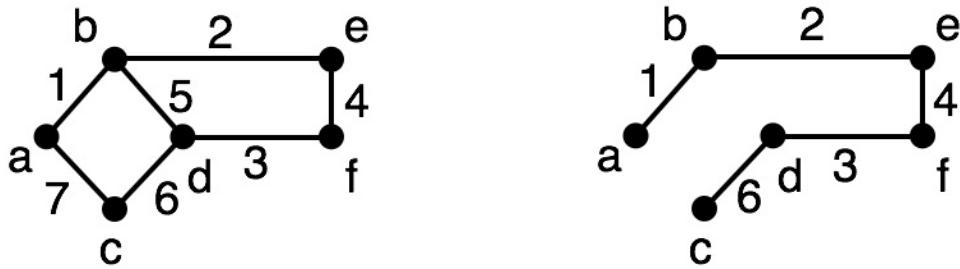
2 Minimum Spanning Trees

A graph can have many spanning trees. In some cases, such as in weighted graphs, we may be interested in finding the spanning tree with the smallest total weight (i.e., sum of the weights of its edges).

Definition 54.2 (Minimum Spanning Trees). Given a connected, undirected weighted graph $G = (V, E, w)$, the minimum (weight) spanning tree (MST) problem requires finding a spanning tree of minimum weight, where the weight of a tree T is defined as

$$w(T) = \sum_{e \in E(T)} w(e).$$

Example 54.2. A graph (top) and its MST (bottom).



Example 54.3 (Network Design). Minimum spanning trees have many interesting applications in network design, i.e., in the design of a network that includes vertices and connections between them. In such network design problems, it can be important to minimize some cost function, defined in terms of the connections in the network. As an example, suppose that you are wiring a building so that all the rooms are connected via bidirectional communication wires. Suppose that you can connect any two rooms at the cost of the wire connecting the rooms, which depends on the specifics of the building and the rooms but is always a positive real number. We can represent this problem as a minimization problem over a graph, where vertices represent rooms and weighted edges represent possible connections and their cost, attached to the graph as weights. To minimize the cost of the wiring, we can find a minimum spanning tree of the graph. One of the algorithms that we cover in this chapter (Boruvka's algorithm) was discovered when developing the electric network for the historical country of Moravia, which is today part of Czech Republic.

Distinct Edge-Weights Assumption. Throughout the discussion of minimum spanning trees, we assume that all edges of graphs have distinct weights. This assumption causes no loss-of-generality, because we can always break ties between edges by ordering them arbitrarily as long as the ordering is deterministic. One way to achieve this is to order edges based on their end-points or assign a unique label to each and break ties by comparing the labels. Such an ordering can be done “statically” ahead of time before running our favorite MST algorithm or “dynamically” as the algorithm runs. Another way is to tweak the edge weights to ensure uniqueness of the weights without altering the ordering of edges with distinct weights, though arguably this approach is rather clumsy from a practical point of view.

Lemma 54.2 (MST Edge Replacement). Let $G = (V, E)$ be a weighted graph and let T be an MST for G . Let $e = \{u, v\} \in E$ be an edge that is not in T . Then e is heavier than any edge e' on the path between u and v in T .

Proof. By [Edge-Replacement Lemma](#) for spanning trees we know that replacing e' with e yields a spanning tree T' . If e lighter than e' , T' is lighter than T and thus T cannot be an MST; a contradiction. \square

Exercise 54.3. Show that for any undirected connected graph with unique edge weights, there exists one unique minimum spanning tree.

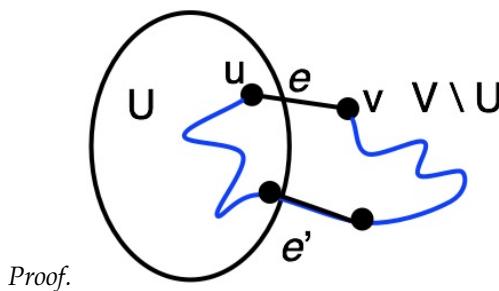
3 Light-Edge Property

There are several algorithms for computing minimum spanning trees. These algorithms are all based on the same underlying property about cuts in a graph, which we will refer to as the *light-edge property*. Intuitively, the light-edge property states that if we partition the graph into two blocks, the minimum edge between the two blocks is in the MST. The light-edge property gives a way to identify algorithmically the edges of the MST.

Definition 54.3 (Graph Cut). For a graph $G = (V, E)$, a cut is defined in terms of a non-empty proper subset $U \subsetneq V$. The set U partitions the graph into blocks induced by the vertex set U and the vertex set $V \setminus U$, which together are called the *cut* and written as the cut $(U, V \setminus U)$. We refer to the edges between the two parts as the *cut edges* written $E(U, V \setminus U)$. We sometimes say that a cut edge *crosses* the cut.

Example 54.4. If the subset U in the definition of graph-cuts is a single vertex v of the graph. The cut edges consist of all edges incident on v .

Lemma 54.3 (Light-Edge Property). Let $G = (V, E, w)$ be a connected undirected, weighted graph with distinct edge weights. For any cut of G , the minimum weight edge that crosses the cut is contained in the minimum spanning tree of G .

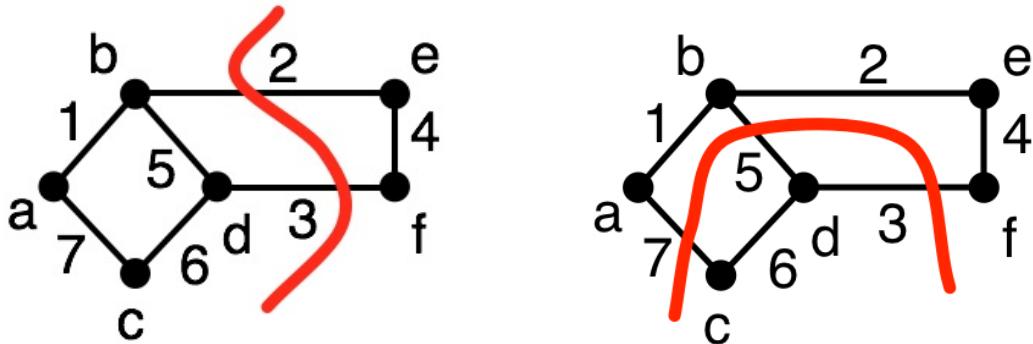


The proof is by contradiction. Assume the lightest edge $e = \{u, v\}$ is not in an MST. Since the MST spans the graph, there is a simple path P connecting u and v in the MST (i.e., consisting of only edges in the MST). Path P crosses the cut between U and $V \setminus U$ at least once since u and v are on opposite sides of the cut.

Let e' be an edge on the path P that crosses the cut. Because e is the lightest edge crossing the cut, e' is heavier than e . But by [Edge-Replacement Lemma](#) for spanning trees, we

know that we can insert e into the MST and delete e' and obtain a lighter spanning tree. This is a contradiction, and thus the lightest edge crossing the cut is in the MST. \square

Example 54.5 (Cuts and Light Edges). The figures below illustrates two cuts. For each cut, we can find the light edge that crosses that cut, which are the edges with weight 2 (top) and 3 (bottom) respectively.



Light-Edge Property and Algorithms. An important implication of the light-edge property as proved in Lemma 54.3 is that any lightest edge that crosses a cut can be immediately added to the MST. In fact, the algorithms that we consider in this section all take advantage of this implication. For example, Kruskal's algorithm constructs the MST by greedily adding the overall minimum edge. Prim's algorithm grows an MST incrementally by considering a cut between the current MST and the rest of graph. Boruvka's algorithm constructs a tree in parallel by considering the cut defined by each and every vertex.

Exercise 54.4. Consider any undirected, connected graph G with unique edge weights. Show that for any cycle in the graph, the heaviest edge on the cycle is not in the MST of G .

4 Approximating Metric TSP via MST

TSP and MST. There is an interesting connection between minimum spanning trees and the symmetric Traveling Salesperson Problem (TSP), an NP-hard problem: certain instances of TSP can be approximated successfully by using MST's. In this section, we present such an approximation algorithm.

Lower Bounding TSP with MST. Recall that in TSP problem, we are given a set of n cities (vertices) and are interested in finding a tour that visits all the vertices exactly once and returns to the origin. In the symmetric case of the problem, the edges are undirected (or equivalently the distance is the same in each direction). For the TSP problem, we usually consider complete graphs, where there is an edge between any two vertices. Even if a

graph is not complete, we can typically complete it by inserting edges with large weights that make sure that the edge never appears in a solution. Here we also assume the edge weights are non-negative.

Since the solution to the TSP problem visits every vertex once (returning to the origin), it spans the graph. But the solution is not a tree but a cycle in which each vertex is visited once. Dropping any edge from the solution therefore would yield a spanning tree. Therefore, a solution to the TSP problem cannot have less total weight than that of a minimum spanning tree. We can thus conclude that for undirected graphs with non-negative edge weights, a minimum spanning tree can be used to obtain a lower bound for the (symmetric) TSP problem.

Approximating TSP with MST. As we shall see in the rest of this section, minimum spanning trees can also be used to find an approximate solutions to the TSP problem, effectively finding an upper bound. This, however, requires one more condition on the TSP problem. In particular in addition to requiring that weights are non-negative we require that all distances satisfy the triangle inequality—i.e., for any three vertices a, b , and c , $w(a, c) \leq w(a, b) + w(b, c)$.

Definition 54.4 (Metric Traveling Salesperson (TSP) Problem). Given a complete undirected graph $G = (V, E)$ with edge weights $W : E \rightarrow \mathbb{R}$ such that

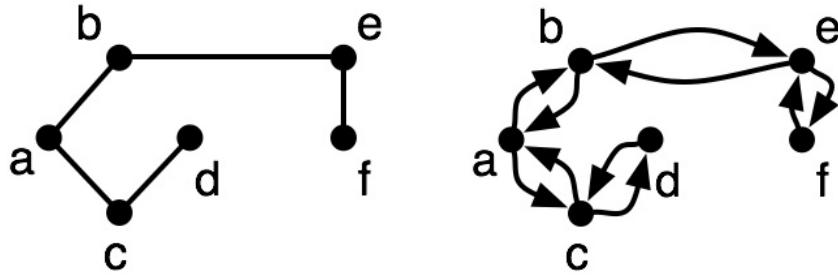
- for all $e \in E$, $W(e) \geq 0$, and
- for all $u, v, w \in E$, $W(u, v) + W(v, w) \geq W(u, w)$,

find the minimum-weight cycle that visits all the vertices.

From a TSP to an Euler Tour. We would like a way to use the MST to generate a path to take as an approximate solution to the metric TSP problem. To do this we first consider a path based on the MST that can visit a vertex multiple times, and then take shortcuts to ensure we only visit each vertex once.

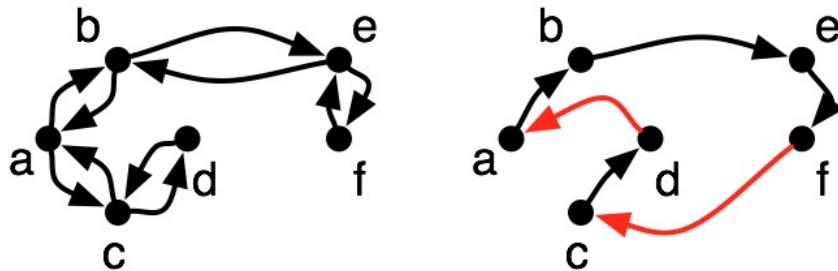
Given a minimum spanning tree T we can start at any vertex s and take a path based on the depth-first search on the tree from s . In particular whenever we visit a new vertex v from vertex u we traverse the edge from u to v and when we are done visiting everything reachable from v we then back up on this same edge, traversing it from v to u . This way every edge in our path is traversed exactly twice, and we end the path at our initial vertex. If we view each undirected edge as two directed edges, then this path is a so-called *Euler tour* of the tree—i.e. a cycle in a graph that visits every edge exactly once. Since T spans the graph, the Euler tour will visit every vertex at least once, but possibly multiple times.

Example 54.6 (Euler Tour). The figure on the right shows an Euler tour of the tree on the left. Starting at a , the tour visits $a, b, e, f, e, b, a, c, d, c, a$.



Shortcuts. Recall that in the TSP problem, the underlying graph is complete and thus there is an edge between every pair of vertices. Because it is possible to take an edge from any vertex to any other, we can take shortcuts to avoid visiting vertices multiple times. More precisely what we can do is the following: when we are about to go back to a vertex that the tour has already visited, instead find the next vertex in the tour that has not been visited and go directly to it. We call this a shortcut edge.

Example 54.7 (Shortcuts). The figure on the right shows a solution to TSP with shortcuts, drawn in red. Starting at a, we can visit a, b, e, f, c, d, a.



Final Bounds. We are now ready to give an upper bound on the TSP problem in terms of MST. Note that by the triangle inequality the shortcut edges are no longer than the paths that they replace. Thus by taking shortcuts, the total distance is not increased. Since the Euler tour traverses each edge in the minimum spanning tree twice (once in each direction), the total weight of the path is exactly twice the weight of the MST. With shortcuts, we obtain a solution to the TSP problem that is at most the weight of the Euler tour, and hence at most twice the weight of the MST. Because the weight of the MST is also a lower bound on the TSP, the solution we have found is within a factor of 2 of optimal. This means our approach is an approximation algorithm for TSP that approximates the solution within a factor of 2. This can be summarized as:

$$W(\text{MST}(G)) \leq W(\text{TSP}(G)) \leq 2W(\text{MST}(G)) .$$

Remark. It is possible to reduce the approximation factor to 1.5 using a well known algorithm developed by Nicos Christofides at CMU in 1976. The algorithm is also based on the MST problem, but is followed by finding a vertex matching on the vertices in the MST with odd-degree, adding these to the tree, finding an Euler tour of the combined graph, and again shortcutting. Christofides algorithm was one of the first approximation algorithms and it took over 40 years to improve on the result, and only very slightly.

Chapter 55

Sequential MST Algorithms

This chapter reviews two sequential algorithms, [Prim's](#) and [Kruskal's](#), for computing Minimum Spanning Trees.

1 Prim's Algorithm

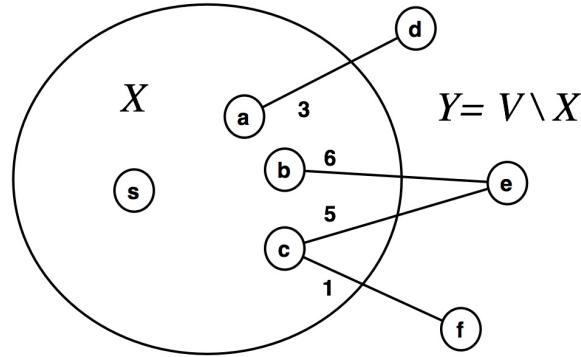
Prim's algorithm performs a priority-first search to construct the minimum spanning tree. To see the basic idea behind the algorithm, imagine that we have already visited a set X of vertices. By Lemma 54.3, we know that the minimum-weight edge e with one of its endpoint in X and the other in $V \setminus X$ is in the MST, because it is a minimum edge crossing the cut defined by X . We can therefore add e to the MST and include the other endpoint in X .

Algorithm 55.1 (Prim's Algorithm). For a weighted undirected graph $G = (V, E, w)$ and a source s , Prim's algorithm is priority-first search on G starting at an arbitrary $s \in V$ with $T = \emptyset$. The algorithm visits the next vertex in the frontier with the least priority, where the priority of a vertex is defined as

$$p(v) = \min_{x \in X} w(x, v).$$

After visiting a vertex the algorithm extends the tree with the chosen edge $T = T \cup \{(u, v)\}$ when visiting v ($w(u, v) = p(v)$). When the algorithm terminates, T is the set of edges in the MST.

Example 55.1. A step of Prim's algorithm. Since the edge (c, f) has minimum weight across the cut (X, Y) , the algorithm will "visit" f adding (c, f) to T and f to X .



Exercise 55.1. Prove the correctness of Prim's algorithm.

Cost of Prim's Algorithm. In the worst case, Prim's algorithm visits every edge exactly once. Because visiting an edge requires the removal of the edge from the priority queue representing the frontier, the cost is $O(\lg n)$ per edge. This is the dominating cost, and using binary heaps for the priority queue yields work and span of $O(m \lg n)$. This can be reduced to $O(m + n \lg n)$ by using Fibonacci heaps.

Prim's and Dijkstra's. Prim's algorithm is quite similar to Dijkstra's algorithm for shortest paths. Both are priority-first search algorithms. The only differences are

- Prim's algorithm starts at an arbitrary vertex instead of at a source,
- Prim's algorithm uses the priority

$$p(v) = \min_{x \in X} w(x, v)$$

instead of the priority used by Dijkstra's:

$$\min_{x \in X} (d(x) + w(x, v))$$

- Prim's algorithm maintains a tree T instead of a table of distances $d(v)$.

Because of the similarity to implement Prim's algorithm, we can basically use the same priority-queue implementation as in Dijkstra's algorithm. Such an implementation requires $O(m \log n)$ work and span.

Remark. Prim's algorithm was invented in 1930 by Czech mathematician Vojtech Jarnik and later independently in 1957 by computer scientist Robert Prim. Edsger Dijkstra's rediscovered it in 1959 in the same paper he described his famous shortest path algorithm.

2 Kruskal's Algorithm

As described in Kruskal's original paper, the algorithm is:

"Perform the following step as many times as possible: Among the edges of G not yet chosen, choose the shortest edge which does not form any loops with those edges already chosen." [Kruskal, 1956]

In more modern terminology we would replace "shortest" with "lightest" and "loops" with "cycles".

Correctness of Kruskal's Algorithm. Kruskal's algorithm is correct since it maintains the invariant on each step that the edges chosen so far are in the MST of G . This is true at the start. Now on each step, any edge that forms a cycle with the already chosen edges cannot be in the MST. This is because adding it would violate the tree property of an MST and we know, by the invariant, that all the other edges on the cycle are in the MST. Now considering the edges that do not form a cycle, the minimum weight edge must be a "light edge" since it is the least weight edge that connects the connected subgraph at either endpoint to the rest of the graph. Finally we have to argue that all the MST edges have been added. Well we considered all edges, and only tossed the ones that we could prove were not in the MST (i.e. formed cycles with MST edges).

We could finish our discussion of Kruskal's algorithm here, but a few words on how to implement the idea efficiently are warranted. In particular checking if an edge forms a cycle might be expensive if we are not careful. Indeed it was not until many years after Kruskal's original paper that an efficient approach to the algorithm was developed. Note that to check if an edge (u, v) forms a cycle, all one needs to do is test if u and v are in the same connected component as defined by the edges already chosen. One way to do this is by contracting an edge (u, v) whenever it is added—i.e., collapse the edge and the vertices u and v into a single super-vertex. However, if we implement this directly, we would need to update all the other edges incident on u and v . This can be expensive since an edge might need to be updated many times.

Union-Find. To get around these problem it is possible to update the edges lazily. What we mean by lazily is that edges incident on a contracted vertex are not updated immediately, but rather later when the edge is processed. At that point the edge needs to determine what supervertices (components) its endpoints are in. This idea can be implemented with a ***union-find data structure***. The ADT for a union-find data structure consists of the following operations on a union-find structure U :

- *insert* $U v$: insert the vertex v into U ,
- *union* $U (u, v)$: join the two elements u and v into a single super-vertex,

- $find(U, v)$: return the super-vertex in which v belongs, possibly itself,
- $equals(u, v)$: return true if u and v are the same super-vertex. Now we can simply process the edges in increasing order.

Algorithm 55.2 (Union-Find Kruskal).

```

1  kruskal ( $G = (V, E, w)$ ) =
2    let
3      addEdge ( $((U, T), e = (u, v))$ ) =
4        let  $u' = find(U, u)$ 
5           $v' = find(U, v)$ 
6          in if ( $equals(u', v')$ ) then  $(U, T)$ 
7            else ( $union(U, u', v'), T \cup e$ )
8          end
9         $U = iterate insert \emptyset V$ 
10        $E' = sort(E, w)$ 
11     in
12       iterate addEdge ( $U, \emptyset$ )  $E'$ 
13   end

```

Exercise 55.2. Prove that Kruskal's algorithm correctly find the MST of a undirected graph with unique edge weights.

Cost of Kruskal's. To analyze the work and span of the algorithm we first note that there is no parallelism, so the span equals the work. To analyze the work we can partition it into the work required for sorting the edges and then the work required to iterate over the edges using union and find. The sort requires $O(m \log n)$ work. The union and find operations can be implemented in $O(\log n)$ work each requiring another $O(m \log n)$ work since they are called $O(m)$ times. The overall work is therefore $O(m \log n)$. It turns out that the union and find operations can actually be implemented with less than $O(\log n)$ amortized work, but this does not reduce the overall work since we still have to sort.

Chapter 56

Parallel MST Algorithms

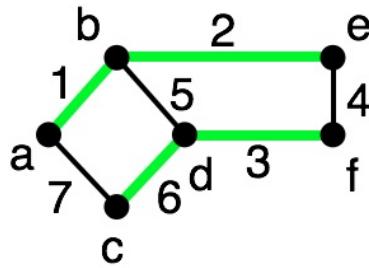
This chapter presents a parallel algorithm, due to Boruvka, for computing minimum spanning trees in parallel. As with all parallel algorithms, the algorithm is trivially a sequential algorithm also, and in fact sequential version of the algorithm are known to perform well in practice.

1 Boruvka's Algorithm

As discussed in previous sections, Kruskal and Prim's algorithm are sequential algorithms. In this section, we present an MST algorithm that runs efficiently in parallel using graph contraction. This parallel algorithm is based on an approach by Boruvka. As Kruskal's and Prim's, Boruvka's algorithm constructs the MST by inserting light edges but unlike them, it inserts many light edges at once.

Vertex Bridges. To see how we can select multiple light edges, recall that by Lemma 54.3 all light edges that cross a cut must be in the MST. Consider now a cut that is defined by a vertex v and the rest of the vertices in the graph. The edges that cross this cut are exactly the edges incident on v . Therefore, by the light edge rule, for v , the minimum weight edge between it and its neighbors is in the MST. Since this argument applies to all vertices at the same time, the minimum weight edges incident on any vertex is in the MST. We call such edges *vertex-bridges* or more simply as *bridges*.

Example 56.1. The vertex bridges of the graph are highlighted.



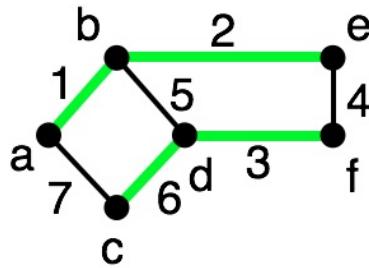
1.1 Algorithm Idea

Let's start by finding the bridge for each and every vertex in the graph. We know that the bridges are all in the MST and thus we can insert them into the MST in parallel. At this point, we might be done—we might have already selected all the MST edges and we can stop. But in most cases, we will not have a spanning tree.

To see how we can proceed, note that the bridges define a partition of the graph, because each and every vertex is the end-point of some bridge and thus is in a block. Consider now the edges that remain internal to a block but are not bridges. Such an edge cannot be in the MST, because inserting it into the MST would create a cycle. The edges that cross the blocks, however, could be in the MST. We therefore want to eliminate the internal edges from consideration, but keep the cross edges. We can do so by performing a graph contraction based on the partition defined by the bridges.

Boruvka's algorithm iterates this approach until the graph is reduced to a single vertex.

Example 56.2 (One Round of Boruvka's Algorithm). Consider the graph below and the highlighted vertex-bridges.

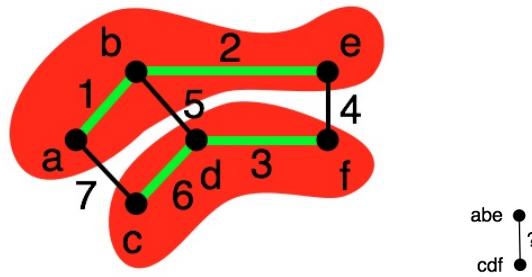


- The vertices a and b both pick edge $\{a, b\}$;
- vertex c picks $\{c, d\}$, d ;
- the vertices d and f both pick $\{d, f\}$, and

- e picks $\{e, b\}$.

The edge (e, f) , which is in the MST, is not selected (neither e nor f pick it).

To proceed, we can take the partitions defined by bridges and contract them by using graph contraction. The figure below illustrates such a contraction. After the contraction completes, we obtain multiple edges between the the resulting partitions.



Redundant Edges. When performing graph contraction, we have to be careful about redundant edges. In our discussion of graph contraction of unweighted graphs, we mentioned that we may treat redundant edges differently based on the application. In unweighted graphs, the task is usually simple because we can keep any one of the redundant edges, and it usually does not matter which one. When the edges have weights, however, we have to decide to keep all the edges or select some of the edges to keep. For the purposes of MST, in particular, we can keep all the edges or keep just the edge with the minimum weight, because the others, cannot be in the MST. In the example above, we would keep the edge with weight 4.

Summary. What we just covered is exactly Boruvka's idea. He did not discuss implementing the contraction in parallel. At the time, there were not any computers let alone parallel ones. In summary, Boruvka's algorithm can be described as follows.

Algorithm 56.1 (Boruvka). While there are edges remaining:

1. select the minimum weight edge out of each vertex and contract each part defined by these edges into a vertex;
2. remove self edges, and when there are redundant edges keep the minimum weight edge; and
3. add all selected edges to the MST.

1.2 Boruvka's Algorithm with Tree Contraction

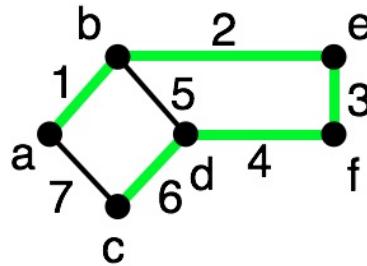
We can implement Boruvka's algorithm as described in Section 1.1 by using tree contraction. In this section we describe how to do this and analyze its cost.

Tree Contraction. To contract the partition defined by the vertex-bridges, we cannot use edge or star contraction, because the blocks may not correspond to an edge or a star. The blocks in general are trees, because each vertex selects exactly one bridge. To contract the block that is induced by the bridges in the block, it suffices to contract along the tree formed by the tree edges. We can do this by removing all non-bridge edges from a block and contracting the block by applying star contraction to it. Because each round of star contraction applied on a tree yields another tree, the number of edges goes down with the number of vertices.

Therefore the total work to contract all the partitions is bounded by $O(n)$ if using array sequences. The span remains $O(\log^2 n)$.

After contracting each tree, we have to update the edges, by re-routing the cut edges between blocks to their new endpoints. This can lead to multiple edges between two vertices, effectively giving us a multi-graph. This can be an effective solution, and allows the updating the edges in $O(m)$ work.

Example 56.3. An example where finding the bridges for all vertices yields a tree that is not a star graph. Note that the selected bridges form a minimum spanning tree.



Cost of Boruvka by Using Tree Contraction. Let's first bound the number of rounds of contraction. Observe that contracting a bridge removes exactly one vertex (contraction of an edge can be viewed as folding one endpoint into the other). Therefore, if k bridges are selected then k vertices are removed.

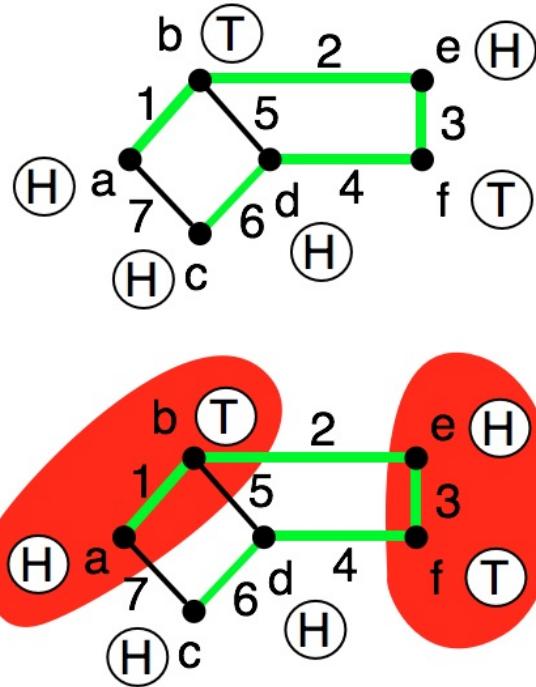
Because each vertex picks a vertex bridge independently in parallel, it is possible that $k = n$. In this case, we would be able to fold all the vertices in one round. In the general case, however, one edge can be chosen by two vertices as vertex bridges. Therefore at least $n/2$ vertex bridges are picked and thus $n/2$ vertices will be removed. Consequently, Boruvka's algorithm will take at most $\lg n$ rounds of selecting bridges and contracting by using 1.2.

Because updating all cut edges requires $O(m)$ work and because there are $\log n$ rounds, Boruvka's algorithm takes $O(m \log n)$ work and $O(\log^3 n)$ span.

1.3 Boruvka's Algorithm with Star Contraction

Star Partition on Bridges. We now describe how to improve the span of Boruvka by a logarithmic factor by interleaving steps of star contraction with steps of finding the vertex-bridges. The idea is to apply star contraction to the subgraph of the graph induced by the bridges. The key observation is that because each non-isolated vertex has a bridge, the subgraph is large enough to give us a constant contraction ratio. Specifically, we will prove that the technique reduces the number of vertices by a constant factor (in expectation), leading to logarithmic number of total rounds. Consequently, we will reduce the overall span for finding the MST from $O(\log^3 n)$ to $O(\log^2 n)$ and maintain the same work.

Example 56.4. An example of Boruvka with star contraction.



Algorithm 56.2 (Star Partition along Bridges). Given a function `vertexBridges` (G) that finds the vertex-bridges out of each vertex in G , the function `bridgeStarPartition` performs star contraction along the vertex bridges. To apply star contraction, the algorithm modifies standard `starContract` function so that after flipping coins, we only contract edges which are vertex-bridges. In the code w denotes the weight of the edge (u, v) .

```

1  bridgeStarPartition ( $G = (V, E)$ ,  $i$ ) =
2    let
3       $Eb = \text{vertexBridges } (G)$ 
4       $P = \{u \mapsto (v, w) \in Eb \mid (\text{flips } (u) = T) \wedge (\text{flips } (v) = H)\}$ 
5       $V' = V \setminus \text{domain}(P)$ 
6    in
7       $(V', P)$ 
8    end

```

Contraction Ratio. Before we go into details about how we might keep track of the MST and other information, let us try to understand what effects this change has on the number of vertices contracted away. If we have n non-isolated vertices, the following lemma shows that the algorithm *bridgeStarPartition* still selects $n/4$ satellites in expectation on each step, and this contracting the graph along these edges will lead to a $1/4$ factor reduction in the number of vertices. The lemma thus implies that this MST algorithm will take only $O(\log n)$ rounds, just like the original star contraction algorithm.

Lemma 56.1 (Number of Bridged Satellites). For a graph G with n non-isolated vertices, let X_n be the random variable indicating the number of vertices removed by *bridgeStarPartition* (G, r). Then, $\mathbf{E}[X_n] \geq n/4$.

Proof. The proof is pretty much identical to our proof for *starContract* except here we're not working with the whole edge set, only a restricted one Eb . Let $v \in V(G)$ be a non-isolated vertex. Like before, let H_v be the event that v comes up heads, T_v that it comes up tails, and R_v that $v \in \text{domain } (P)$ (i.e. it is removed). Since v is a non-isolated vertex, v has neighbors—and one of them has the minimum weight, so there exists a vertex u such that $(v, u) \in Eb$. Then, we have that $T_v \wedge H_u$ implies R_v since if v is a tail and u is a head, then v must join u . Therefore, $\mathbf{P}[R_v] \geq \mathbf{P}[T_v] \mathbf{P}[H_u] = 1/4$. By the linearity of expectation, we have that the number of removed vertices is

$$\mathbf{E} \left[\sum_{v: v \text{ non-isolated}} \mathbb{I}\{R_v\} \right] = \sum_{v: v \text{ non-isolated}} \mathbf{E}[\mathbb{I}\{R_v\}] \geq n/4$$

since we have n vertices that are non-isolated. \square

Exercise 56.1. Compare the proof [the bridged-satellites lemma](#) to [the original star-partition lemma](#). What remains the same, what has changed?

Tracking Edges. There is a little bit of trickiness in constructing the result MST. As the graph contracts, the endpoints of each edge changes. Therefore, if we want to return the edges of the minimum spanning tree, then we might need to keep track of changes in how edges are re-routed between vertices. To avoid this, we associate a unique label with every edge and return the tree as a set of labels (i.e. the labels of the edges in the spanning tree). We also associate the weight directly with the edge. The type of each edge is therefore

(vertex \times vertex \times weight \times label), where the two vertex endpoints can change as the graph contracts but the weight and label stays fixed. This leads to the slightly-updated version of `bridgeStarPartition` that appears in the algorithm given below.

Algorithm 56.3 (Boruvka's based on Star Contraction). The function `vertexBridge(G)` finds the minimum edge out of each vertex v and maps v to the pair consisting of the neighbor along the edge and the edge label. To this end, the function makes a singleton table for each edge and then merge all the tables with a function to resolve collisions, which favors lighter edge.

The function `bridgeStarPartition` performs star contraction on the subgraph induced by the bridges. It starts by selecting the bridges and then in Line 12 it picks from bridges the edges that go from a tail to a head. It then generates a mapping from tails to heads along minimum edges, creating stars. Line 13 removes all vertices that are in this mapping to star centers.

The function `bridgeStarPartition` is ready to be used in the MST code, similar to the `graphContract` code studied last time, except we return the set of labels for the MST edges instead of the remaining vertices. The code is given below. The MST algorithm is called by running `MST(G, \emptyset , 0)`. As an aside, we know that T is a spanning forest on the contracted nodes.

```

1  vertexBridges E =
2  let
3      ET =  $\{(u, v, w, l) \mapsto \{u \mapsto (v, w, l)\} : (u, v, w, l) \in E\}$ 
4      select  $((v_1, w_1, l_1), (v_2, w_2, l_2)) =$ 
5          if  $(w_1 \leq w_2)$  then  $(v_1, w_1, l_1)$  else  $(v_2, w_2, l_2)$ 
6      in
7          reduce (union select) {} ET
8      end
9  bridgeStarPartition (G = (V, E)) =
10 let
11     Eb = vertexBridges G
12     P =  $\{(u \mapsto (v, w, \ell)) \in Eb \mid (\text{flip}(u) = T) \wedge (\text{flip}(v) = H)\}$ 
13     V' = V \ domain(P)
14     in
15     (V', P)
16     end
17 MST (V, E) T =
18 if  $(|E| = 0)$  then T
19 else
20     let
21         (V', PT) = bridgeStarPartition (V, E)
22         P =  $\{u \mapsto v : u \mapsto (v, w, \ell) \in PT\} \cup \{v \mapsto v : v \in V'\}$ 
23         T' =  $\{\ell : u \mapsto (v, w, \ell) \in PT\}$ 
24         E' =  $\{(P[u], P[v], w, l) : (u, v, w, l) \in E \mid P[u] \neq P[v]\}$ 
25     in
26     MST (V', E') (T  $\cup$  T')
27 end

```

Remark. Even though Boruvka's algorithm is not the only parallel algorithm, it was the earliest, invented in 1926, as a method for constructing an efficient electricity network in Moravia in the Czech Republic. It was re-invented many times over.

Part XII

Dynamic Programming

Chapter 57

Introduction

Dynamic programming is an inductive algorithm-design technique. Similar to divide-and-conquer, it solves larger instances of a problem in terms of smaller ones. The main difference is that many of the smaller instances are shared among recursive calls, making it worthwhile to save the partial solutions so they can be reused. Dynamic programming is usually inherently parallel, but taking advantage of the parallelism is a bit more tricky because the sharing of solutions need to be properly coordinated.

Origins of “Dynamic Programming”. Unlike many of the other algorithmic techniques, the name “dynamic programming” sheds little light on how the technique works. We could blame the bad name on the person who invented the idea, Richard Bellman. However, as the following quote from Bellman indicates, it has more to do with the US government during the McCarthy years, 1950-1957, an era when many people were jailed for expressing their constitutional rights of free speech.

“An interesting question is, ‘Where did the name, dynamic programming, come from?’ The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I’m not using the term lightly; I’m using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, ‘programming.’ I wanted to get across the idea that this was dynamic,

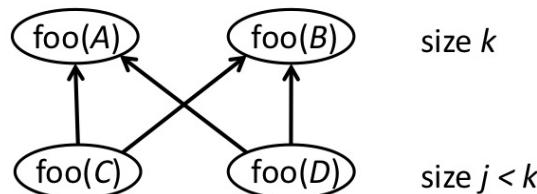
this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. This, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities".

Richard Bellman ("Eye of the Hurricane: An autobiography", World Scientific, 1984)

The Bellman-Ford shortest path algorithm covered in Chapter 48 is named after Richard Bellman and Lester Ford. That algorithm is a dynamic-programming algorithm, when looked at the right way.

It is all about sharing. When using divide-and-conquer, we reduce the problem instance that we want to solve into multiple smaller instances and assume that the smaller instances are solved recursively and independently. When calculating total work, we therefore add up the work from each recursive call. In some cases, the smaller instances are simply not independent, because solving two instances may both involve solving a smaller shared instance. For example, two instances of size k may both need the solution to the same instance of size $j < k$. The idea behind dynamic programming is to take advantage of such "sharing" and instead of solving the smaller instance twice, to solve it once and share the result, effectively re-using the result as needed. In general such sharing can make significant, as much as exponential, improvement in work.

Example 57.1. Two smaller instance of the problem (function call) foo being shared by two larger instances.



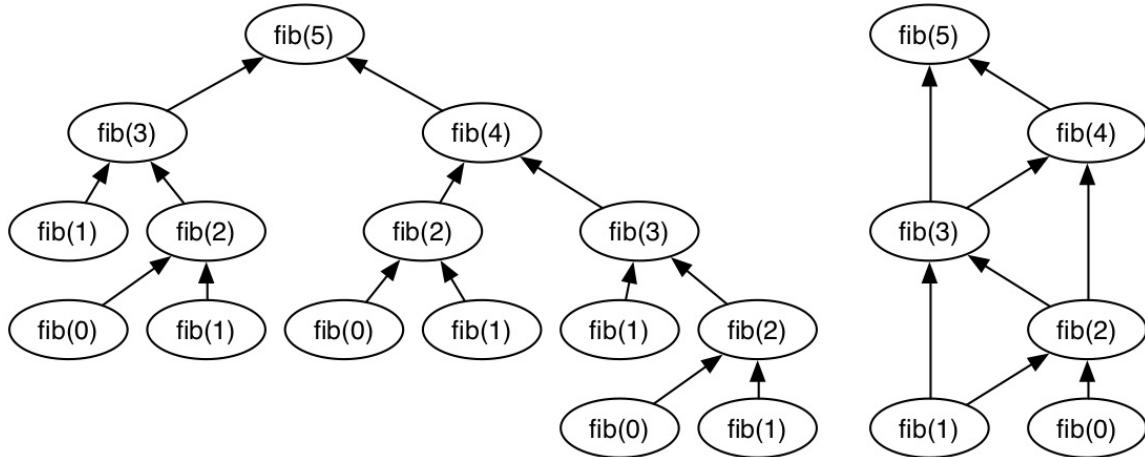
Although sharing the results in this example makes at most a factor of two difference in work, in general sharing the results can make an exponential difference in the work performed.

Example 57.2. Consider the following algorithm for calculating the Fibonacci numbers.

```

 $\text{fib}(n) =$ 
  if ( $n \leq 1$ ) then 1
  else  $\text{fib}(n - 1) + \text{fib}(n - 2)$ 
  
```

This recursive algorithm takes exponential work in n as indicated by the recursion tree below on the left for $\text{fib}(5)$. If the results from the instances are shared, however, then the algorithm only requires linear work, as illustrated below on the right.



Here many of the calls to fib are reused by two other calls. Note that the root of the tree or DAG is the problem we are trying to solve, and the leaves of the tree or DAG are the base cases.

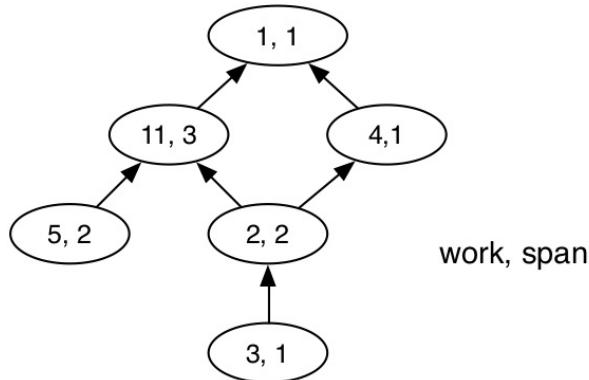
Representing Sharing with DAGs. With divide-and-conquer the composition of a problem instance in terms of smaller instances is typically described as a tree, and in particular the so called recursion tree. With dynamic programming, to account for sharing, the composition can instead be viewed as a Directed Acyclic Graph (DAG). Each vertex in the DAG corresponds to a problem instance and each edge goes from an instance of size j to one of size $k > j$ —i.e. each directed edge (arc) is directed from a smaller instances to a larger instance that uses it. The edges therefore represent dependencies between the source and destination (i.e. the source has to be calculated before the destination can be). The leaves of this DAG (i.e. vertices with no in-edges) are the base cases of our induction (instances that can be solved directly), and the root of the DAG (the vertex with no out-edges) is the instance we are trying to solve. More generally we might actually have multiple roots if we want to solve multiple instances.

Work and Span. Abstractly dynamic programming can therefore be best viewed as evaluating a DAG by propagating values from the leaves (in degree zero) to the root (out degree zero) and performing some calculation at each vertex based on the values of its in-neighbors. Based on this view, calculating the work and span of a dynamic program is relatively straightforward. We can associate with each vertex a work and span required for that vertex. We then have

- The *work* of a dynamic program viewed as a DAG is the sum of the work of the vertices of that DAG, and
 - the *span* of a dynamic program viewed as a DAG is the heaviest vertex-weighted path in the DAG—i.e., the weight of each path is the sum of the spans of the vertices along it.

Whether a dynamic programming algorithm has much parallelism (work over span) will depend on the particular DAG. As usual the parallelism is defined as the work divided by the span. If this is large, and grows asymptotically, then the algorithm has significant parallelism. Most dynamic programs have significant parallelism but some do not.

Example 57.3. Consider the following DAG:



where we have written the work and span on each vertex. This DAG does $5 + 11 + 3 + 2 + 4 + 1 = 26$ units of work and has a span of $1 + 2 + 3 + 1 = 7$.

The challenging part of developing a dynamic programming algorithm for a problem is in determining what DAG to use. The best way to do this is to think inductively: how can we solve an instance of a problem by composing the solutions to smaller instances? After we formulate an inductive solution, we then think about whether the solutions can be shared and how much savings can be achieved by sharing. As with all algorithmic techniques, being able to come up with solutions takes practice.

Definition 57.1 (Optimization and Decision Problem). Most problems that can be tackled with dynamic programming are optimization or decision problems. An *optimization problem* requires finding a solution that optimizes some criteria (e.g., finding a shortest path, or finding the longest contiguous subsequence sum).

Sometimes we want to enumerate (list) all optimal solutions, or count the number of such solutions. A *decision problem* is one in which we are trying to find if a solution to a problem exists. Again we might want to count or enumerate the valid solutions.

When solving an optimization or an enumeration problem, we usually solve many smaller instances of the same problem and therefore may improve total work by sharing common solutions via dynamic programming.

Coding Dynamic Programs. Although dynamic programming can be viewed abstractly as a DAG, in practice we need to implement (code) the dynamic program. There are two common ways to do this, which are referred to as the top-down and bottom-up approaches. The *top-down* approach starts at the root(s) of the DAG and uses recursion, as in divide-and-conquer, but remembers solutions to subproblems so that when the algorithm needs to solve the same instance many times, only the first call does the work and the remaining calls just look up the solution. Storing solutions for reuse is called *memoization*. The *bottom-up* approach starts at the leaves of the DAG and typically processes the DAG in some form of level order traversal—for example, by processing all problems of size 1 and then 2 and then 3, and so on.

Each approach has its advantages and disadvantages. Using the top-down approach (recursion with memoization) can be quite elegant and can be more efficient in certain situations by evaluating only those instances actually needed. The bottom up approach (level order traversal of the DAG) can be easier to parallelize and can be more space efficient, but always requires evaluating all instances. There is also a third technique for solving dynamic programs that works for certain problems, which is to find the shortest path in the DAG where the weights on edges are defined in some problem specific way.

Summary. In summary the approach to coming up with a dynamic programming solution to a problem is as follows.

1. Is it a decision or optimization problem?
2. Define a solution recursively (inductively) by composing the solution to smaller problems.
3. Identify any sharing in the recursive calls, i.e. calls that use the same arguments.
4. Model the sharing as a DAG, and calculate the work and span of the computation based on the DAG.
5. Decide on an implementation strategy: either bottom up top down, or possibly shortest paths.

It is important to remember to first formulate the problem abstractly in terms of the inductive structure, then think about it in terms of how substructure is shared in a DAG, and only then worry about coding strategies.

Remark (Problems with Efficient Dynamic Programming Solutions). There are many problems with efficient dynamic programming solutions. Here we list just some of them.

1. Fibonacci numbers
2. Using only addition compute (n choose k) in $O(nk)$ work
3. Edit distance between two strings
4. Edit distance between multiple strings
5. Longest common subsequence
6. Maximum weight common subsequence
7. Can two strings S_1 and S_2 be interleaved into S_3
8. Longest palindrome
9. longest increasing subsequence
10. Sequence alignment for genome or protein sequences
11. Subset sum
12. Knapsack problem (with and without repetitions)
13. Weighted interval scheduling
14. Line breaking in paragraphs
15. Break text into words when all the spaces have been removed
16. Chain matrix product
17. Maximum value for parenthesizing $x_1/x_2/x_3\dots/x_n$ for positive rational numbers
18. Cutting a string at given locations to minimize cost (costs n to make cut)
19. All shortest paths
20. Find maximum independent set in trees
21. Smallest vertex cover on a tree
22. Optimal BST
23. Probability of generating exactly k heads with n biased coin tosses
24. Triangulate a convex polygon while minimizing the length of the added edges
25. Cutting squares of given sizes out of a grid
26. Change making
27. Box stacking
28. Segmented least squares problem
29. Counting Boolean parenthesization – true, false, or, and, xor, count how many parenthesization return true
30. Balanced partition – given a set of integers up to k , determine most balanced two way partition
31. Largest common subtree

Chapter 58

Two Problems

In this chapter we cover two problems that are well suited for dynamic programming solutions: subsets sums, and minimum edit distance.

1 Subset Sums

The first problem we cover in this chapter is a decision problem, the subset sum problem. It takes as input a multiset of numbers, i.e. a set that allows duplicate elements, and sees if any subset sums to a target value. More formally:

Definition 58.1 (Subset Sum (SS) Problem). The *subset sum* (SS) problem is, given a multiset of positive integers S and a positive integer value k , determine if there is any $X \subseteq S$ such that $\sum_{x \in X} x = k$.

Example 58.1. $SS(\{1, 4, 2, 9\}, 8)$ returns *false* since there is no subset of 1, 4, 2, and 9 that adds up to 8. However, $SS(\{1, 4, 2, 9\}, 12)$ returns *true* since $1 + 2 + 9 = 12$.

Definition 58.2 (Subset Sum (SS) Problem). The *subset sum* (SS) problem is, given a multiset of positive integers S and a positive integer value k , determine if there is any $X \subseteq S$ such that $\sum_{x \in X} x = k$.

Example 58.2. $SS(\{1, 4, 2, 9\}, 8)$ returns *false* since there is no subset of 1, 4, 2, and 9 that adds up to 8. However, $SS(\{1, 4, 2, 9\}, 12)$ returns *true* because $1 + 2 + 9 = 12$.

Hardness of the SS Problem. In the general case when k is unconstrained, then SS problem is a classic NP-hard problem. However, our goal here is more modest. We are going to consider the case where we include the value of k in the work bounds as a variable.

We show that as long as k is polynomial in $|S|$ then the work is also polynomial in $|S|$. Solutions of this form are often called *pseudo-polynomial* work (or time) solutions.

The SS problem can be solved using brute force by simply considering all possible subsets. This takes exponential work since there are an $2^{|S|}$ subsets. For a more efficient solution, one should consider an inductive solution to the problem. As greedy algorithms tend to be efficient, you should first consider some form of greedy method that greedily takes elements from S . Unfortunately the greedy method does not work. The problem is that in general there is no way to know for a particular element whether to include it or not. Greedily adding it could be a mistake, and recall that in greedy algorithms once you make a choice you cannot go back and undo your choice.

Since we do not know whether to add an element or not, we could try both cases, i.e. finding a sum with and without that element. This leads to a divide-and-conquer approach for solving $SS(S, k)$ in which we pick one element a out of the set S (any will do), and then make two recursive calls, one with a included in X (the elements of S that sum to k) and one without a . For the call in which we include a we need to subtract the value a from k and in the other case we leave k as is. Here is an algorithm based on this idea. It assumes the input is given as a list (the order of the elements of S in the list does not matter):

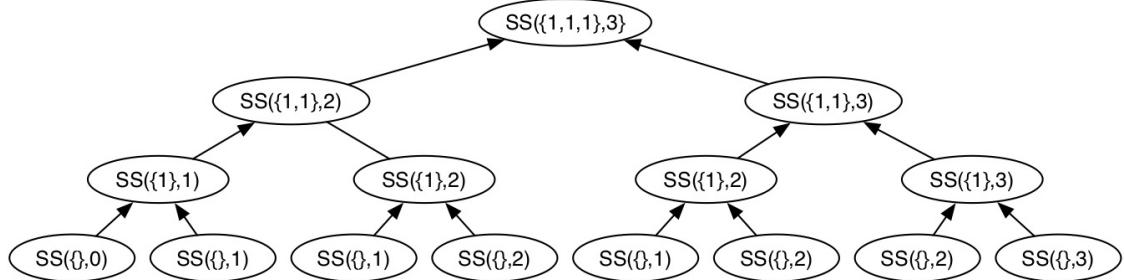
Algorithm 58.3 (Recursive Subset Sum).

```
SS( $S, k$ ) =
  case ( $S, k$ ) of
    ( $\_, 0$ )  $\Rightarrow$  true
    | ( $Nil, \_$ )  $\Rightarrow$  false
    | ( $Cons(a, R), \_$ )  $\Rightarrow$ 
      if ( $a > k$ ) then  $SS(R, k)$ 
      else ( $SS(R, k - a)$  or  $SS(R, k)$ )
```

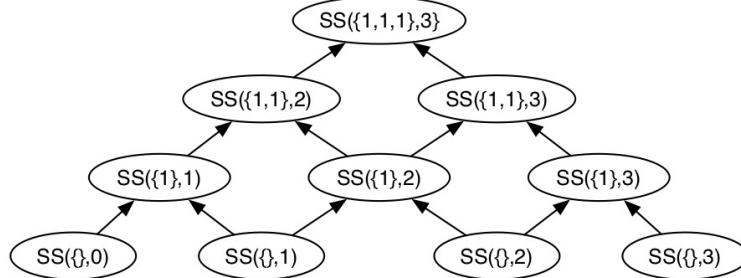
The first two cases are the base cases. In particular if $k = 0$ then the result is true since the empty set sums to zero, and the empty set is a subset of any set. If $k \neq 0$ and S is empty, then the result is false since there is no way to get k from an empty set. If S is not empty but its first element a is greater than k , then we clearly can not add a to X , and we need only make one recursive call. The last line is the main inductive case where we either include a or not. In both cases we remove a from S in the recursive call to SS , and therefore use R . In the left case we are including a in the set so we have to subtract its value from k . In the right case we are not, so k remains the same. The algorithm is correct by induction—the base cases are correct, and inductively we assume the subproblems are correct and then note that those are the only two possibilities.

The [recursive algorithm](#) for the SS problem leads to a binary recursion tree that might be $n = |S|$ deep. In this tree, there are 2^n leaves and each path from root to the leaf represent a subset. This implies that the work of the algorithm could be $O(2^n)$, which is large. We can improve work by observing that there is a large amount of sharing of subproblems.

Example 58.3. Consider $SS(\{1, 1, 1\}, 3)$. This clearly should return true because $1 + 1 + 1 = 3$. The recursion tree is as follows.



There are many calls to SS in this tree with the same arguments. In the bottom row, for example there are three calls each to $SS(\emptyset, 1)$ and $SS(\emptyset, 2)$. If we coalesce the common calls we get the following DAG:



Improving Work by Sharing. The question is how do we calculate how much sharing there is, or more specifically how many distinct subproblems are there in. For an initial instance $ss(S, k)$ there are only $|S|$ distinct lists that are ever used (each suffix of S). Furthermore, the value of the second argument in the recursive calls only decreases and never goes below 0, so it can take on at most $k + 1$ values. Therefore the total number of possible instances of SS (vertices in the DAG) is $|S|(k + 1) = O(k|S|)$.

To calculate the overall work we need to sum the work over all the vertices of the DAG. However, each vertex only needs to do some constant number of operations (a comparison, a subtract, a logical or, and a few branches). Therefore each node does constant work and we have that the overall work is:

$$W(SS(S, k)) = O(k|S|)$$

To calculate the span we need to know the heaviest path in the DAG. Again the span of each vertex is constant, so we only need to count the number of nodes in a path. The length of the longest path is at most $|S|$ since on each level we remove one element from the set. Therefore we have:

$$S(SS(S, k)) = O(|S|)$$

and together this tells us that the parallelism is $O(W/S) = O(k)$.

At this point we have not fully specified the algorithm since we have not explained how to take advantage of the sharing—certainly the recursive code we wrote would not. We will get back to this after one more example. Again we want to emphasize that the first two orders of business are to figure out the inductive structure and figure out what instances can be shared.

To make it easier to determine an upper bound on the number of subproblems in a DP DAG it can be convenient to replace any sequences (or lists) in the argument to the recursive function with an integer indicating our current position in the input sequence(s). For the subset sum problem this leads to the following variant of our previous algorithm:

Algorithm 58.4 (Recursive Subset Sum (Indexed)).

```

 $SS(S, k) =$ 
let  $SS'(i, j) =$ 
  case  $(i, j)$  of
     $(-, 0) \Rightarrow \text{true}$ 
     $(0, -) \Rightarrow \text{false}$ 
     $| - \Rightarrow$ 
      if  $(S[i - 1] > j)$  then  $SS'(i - 1, j)$ 
      else  $(SS'(i - 1, j - S[i - 1]) \text{ or } SS'(i - 1, j))$ 
  in  $SS'(|S|, k)$  end

```

In the algorithm the $i - 1$ represents the element we are currently considering. We start with $i = |S|$ and when $i = 0$ we are done (the algorithm reaches the base case). As we will see later this has a second important advantage—it makes it easier for a program to recognize when arguments are equal so they can be reused.

Remark. Why do we say the SS algorithm we described is pseudo-polynomial? The size of the subset sum problem is defined to be the number of bits needed to represent the input. Therefore, the input size of k is $\log k$. But the work is $O(2^{\log k} |S|)$, which is exponential in the input size. That is, the complexity of the algorithm is measured with respect to the length of the input (in terms of bits) and not on the numeric value of the input. If the value of k , however, is constrained to be a polynomial in $|S|$ (i.e., $k \leq |S|^c$ for some constant c) then the work is $O(k|S|) = O(|S|^{c+1})$ on input of size $c \log |S| + |S|$, and the algorithm is polynomial in the length of the input.

2 Minimum Edit Distance

The second problem we consider is a optimization problem, the minimum edit distance problem.

Definition 58.5 (Minimum Edit Distance (MED) Problem). The *minimum edit distance problem* or *MED* problem for short is, given a character set Σ and two sequences of characters $S = \Sigma^*$ and $T = \Sigma^*$, determine the minimum number of insertions and deletions of single characters required to transform S to T .

Example 58.4. Consider the sequence

$$S = \langle A, B, C, A, D, A \rangle$$

we could transform it to

$$T = \langle A, B, A, D, C \rangle$$

with 3 edits (delete the C in the middle, delete the last A, and insert a C at the end). This is the best that can be done so we have that $MED(S, T) = 3$.

Applications of MED. Finding the minimum edit distance is an important problem that has many applications. For example in version control systems such as `git` or `svn` when you update a file and commit it, the system does not store the new version but instead only stores the “differences” from the previous version. (Alternatively it might store the new version, but use the differences to encode the old version.) Storing the differences can be quite space efficient since often the user is only making small changes and it would be wasteful to store the whole file. Variants of the minimum edit distance problem are used to find this difference. Edit distance can also be used to reduce communication costs by only communicating the differences from a previous version. It turns out that edit-distance is also closely related to approximate matching of genome sequences. In many of these applications it is useful to know in addition to the minimum number of edits, the actual edits. It is easy to extend the approach in this section for this purpose, but we leave it as an exercise.

Remark. The algorithm used in the Unix “diff” utility was invented and implemented by Eugene Myers, who also was one of the key people involved in the decoding of the human genome at Celera.

Greedy Algorithm. To solve the MED problem we might consider trying a greedy method that scans the sequences finding the first difference, fixing it and then moving on. Unfortunately no simple greedy method is known to work. The difficulty is that there are two ways to fix the error—we can either delete the offending character, or insert a new one. If we greedily pick the wrong edit, we might not end up with an optimal solution. Note that this is similar to the subset sum problem where we did not know whether to include an element or not.

Example 58.5. Consider the sequences

$$S = \langle A, B, C, A, D, A \rangle$$

and

$$T = \langle A, B, A, D, C \rangle.$$

We can match the initial characters $A - A$ and $B - B$ but when we come to $C - A$ in S and T , we have two choices for editing C , delete C or insert A . However, we do not know which leads to an optimal solution because we don’t know the rest of the sequences. In the example, if we insert an A , then a suboptimal number of edits will be required.

As with the subset sum problem, since we cannot decide which choice to make (in this case deleting or inserting), why not try both. This again leads to a recursive solution. In the solution we can start at either end of the string, and go along matching characters, and whenever two characters do not match, we try both a deletion and an insertion, recurse on the rest of the string, and pick the best of the two choices. This idea leads to the following algorithm (S and T are given as lists, and we start from the front).

Algorithm 58.6 (Recursive MED).

```

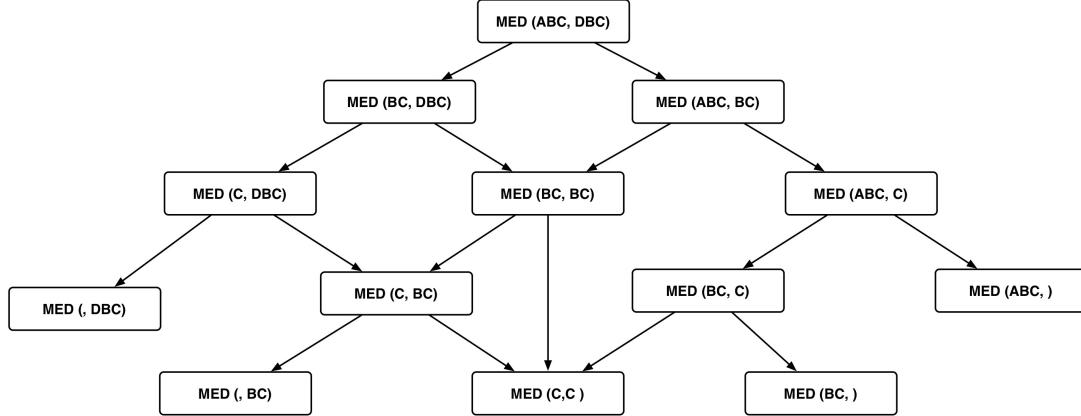
 $MED(S, T) =$ 
   $\text{case } (S, T) \text{ of}$ 
     $(\_, \text{Nil}) \Rightarrow |S|$ 
     $| (\text{Nil}, \_) \Rightarrow |T|$ 
     $| (\text{Cons}(s, S'), \text{Cons}(t, T')) \Rightarrow$ 
       $\text{if } (s = t) \text{ then } MED(S', T')$ 
       $\text{else } 1 + \min(MED(S, T'), MED(S', T))$ 

```

In the first base case where T is empty we need to delete all of S to generate an empty string requiring $|S|$ deletions. In the second base case where S is empty we need to insert all of T , requiring $|T|$ insertions. If neither is empty we compare the first character of each string, s and t . If these characters are equal we can just skip them and make a recursive call on the rest of the sequences. If they are different then we need to consider the two cases. The first case ($MED(S, T')$) corresponds to inserting the value t . We pay one edit for the insertion and then need to match up S (which all remains) with the tail of T (we have already matched up the head t with the character we inserted). The second case ($MED(S', T)$) corresponds to deleting the value s . We pay one edit for the deletion and then need to match up the tail of S (the head has been deleted) with all of T .

The recursive algorithm for MED performs exponential work. In particular the recursion tree is a full binary tree (each internal node has two children) and has a depth that is linear in the size of S and T . Observe, however, that there are many calls to MED with the same arguments. We thus view the computation as a DAG in which each vertex corresponds to a call to MED with distinct arguments. An edge is placed from u to v if the call v uses u .

Example 58.6. An example MED instance with sharing.



The call to $MED(\langle B, C \rangle, \langle D, B, C \rangle)$, for example, makes recursive calls to $MED(\langle C \rangle, \langle D, B, C \rangle)$ (corresponding to the deletion of B from the first string) and $MED(\langle B, C \rangle, \langle B, C \rangle)$ (corresponding to the insertion of D into the second string). One of the calls is shared with the call to $MED(\langle A, B, C \rangle, \langle B, C \rangle)$

Work and Span. To determine the work we need to know how many vertices there are in the DAG. We can place an upper bound on the number of vertices by bounding the number of distinct arguments. There can be at most $|S| + 1$ possible values of the first argument since in recursive calls we only use suffixes of the original S and there are only $|S| + 1$ such suffixes (including the empty and complete suffixes). Similarly there can be at most $|T| + 1$ possible values for the second argument. Therefore the total number of possible distinct arguments to MED on original strings S and T is $(|T| + 1)(|S| + 1) = O(|S||T|)$. Furthermore the depth of the DAG (heaviest path) is $O(|S| + |T|)$ since each recursive call either removes an element from S or T so after $|S| + |T|$ calls there cannot be any element left. Finally we note that assuming we have constant work operations for removing the head of a sequence (e.g. using a list) then each vertex of the DAG takes constant work and span.

All together this gives us

$$W(MED(S, T)) = O(|S||T|)$$

and

$$S(MED(S, T)) = O(|S| + |T|).$$

As in subset sum we can again replace the lists used in MED with integer indices pointing to where in the sequence we are currently at. This gives the following variant of the MED algorithm:

Algorithm 58.7 (Recursive MED (Indexed)).

```

 $MED(S, T) =$ 
let  $MED'(i, j) =$ 
  case  $(i, j)$  of
     $(i, 0) \Rightarrow i$ 
     $| (0, j) \Rightarrow j$ 
     $| (i, j) \Rightarrow$  if  $(S[i - 1] = T[i - 1])$  then  $MED'(i - 1, j - 1)$ 
     $| \text{else } 1 + \min(MED'(i, j - 1), MED'(i - 1, j))$ 
  in  $MED'(|S|, |T|)$  end

```

This variant starts at the end of the sequence instead of the start, but is otherwise equivalent to our previous version. This form makes it more clear that there are only $|S| \times |T|$ distinct arguments, and will make it easier to implement efficiently, as we discuss next.

Chapter 59

Optimal Binary Search Trees

We consider the problem of finding the optimally balanced tree that minimizes the expected cost of searches for a given probability distribution on the search queries.

Background. As we saw in [an earlier chapter](#) Binary Search Trees can be used to store a dynamically changing set of keys and perform search queries on them efficiently. The cost of searching for a key is linear in the depth of the key in the tree, or equivalently in the length of the path from the root to the key. In a balanced BST with n keys the average depth of each key is approximately $\log n$.

Optimizing for a Query Distribution. Balanced search trees minimize the worst-case cost of an access by making sure that all keys are as close to the root as possible. This “pessimistic” perspective is not always necessary, because we sometimes have more information about the query pattern for the keys, and specifically the frequency of queries for each key. In such an application, it would be better to place frequently queried keys closer to the root even if this causes other, less frequently queried keys, being further away from the root. More precisely, suppose that for a query we have a probability density function that maps each key to its probability of being queried. We may then want to come up with a binary search tree that minimizes the expected cost of a query under that probability distribution.

Example 59.1. Suppose that we have a dictionary for the English language that we would like to use to answer queries from students learning English as a foreign language. We could use a binary search tree to store the entries in the dictionary. In such an application, certain words will be more frequently queried than others, e.g., queries for the word “lamp” will appear more than the word “epistemology”. We can take advantage of this by placing such words closer to the root even if this causes less frequently accessed words to be further away from the root.

Definition 59.1 (Optimal Binary Search Tree (OBST) Problem). The *optimal binary search tree* (OBST) problem is given an ordered set of keys S and a probability function $p : S \rightarrow [0 : 1]$:

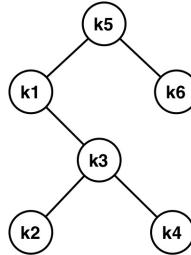
$$\min_{T \in \text{Trees}(S)} \left(\sum_{s \in S} d(s, T) \cdot p(s) \right)$$

where $\text{Trees}(S)$ is the set of all BSTs on S , and $d(s, T)$ is the depth of the key s in the tree T (the root has depth 1).

Example 59.2. For example we might have the following keys and associated probabilities

key	k_1	k_2	k_3	k_4	k_5	k_6
$p(\text{key})$	1/8	1/32	1/16	1/32	1/4	1/2

Then the tree below has cost $31/16$, which is optimal. Creating a tree with these two solutions as the left and right children of S_i , respectively, leads to the optimal solution given S_i as a root.



Exercise 59.1. Find another tree with equal cost.

Brute Force. The brute force solution would be to generate every possible binary search tree, compute their cost, and pick the one with the lowest costs. But the number of such trees is $O(4^n)$ which is prohibitive.

Exercise 59.2. Write a recurrence for the total number of distinct binary search trees with n keys.

Optimal Substructure Property. Consider an optimal binary search tree for a sequence of unique keys S and probability law P and let r be the root of the tree. Observe now that each subtree of the root is an optimal binary search tree, because otherwise, we could replace them with a binary search tree that improves the expected cost of queries for the subtree, which in turn would improve the grant total. This common property of optimization problems is sometimes called the *optimal substructure property*. This property is sometimes a clue that either a greedy or dynamic programming algorithm might apply.

Exercise 59.3. Can we solve the optimal binary search tree problem by using the greedy technique?

Solution. A greedy approach might be to pick the key k with highest probability and make it the root of the binary search tree. We may then construct the two subtrees recursively on the two sets less and greater than k . This does not necessarily give us the optimal binary search tree, however, because for example the key with the highest property might be largest key and cause the path for all other keys to be increased by one. As an exercise (to the exercise), construct such an example.

A Recursive Solution. Let S be all the keys placed in sorted order. Observe that any subtree of a BST on S contains the keys of a contiguous subsequence of S . We can therefore define subproblems in terms of a contiguous subsequence of S . We will use $S_{i,j}$ to indicate the subsequence starting at the key with rank i and going to key with rank j (inclusive of both). We will then use the pair (i, j) to be the surrogate for $S_{i,j}$.

For subproblem $S_{i,j}$, suppose that we pick key S_r ($i \leq r \leq j$) as the root. We can now solve the OSBT problem on the prefix $S_{i,r-1}$ and suffix $S_{r+1,j}$. Let T be the tree on the keys $S_{i,j}$ with root S_r , and T_L, T_R be its left and right subtrees. We can write the expected cost as follows.

$$\begin{aligned}
 \text{Cost}(T) &= \sum_{s \in T} d(s, T) \cdot p(s) \\
 &= p(S_r) + \sum_{s \in T_L} (d(s, T_L) + 1) \cdot p(s) + \sum_{s \in T_R} (d(s, T_R) + 1) \cdot p(s) \\
 &= \sum_{s \in T} p(s) + \sum_{s \in T_L} d(s, T_L) \cdot p(s) + \sum_{s \in T_R} d(s, T_R) \cdot p(s) \\
 &= \sum_{s \in T} p(s) + \text{Cost}(T_L) + \text{Cost}(T_R)
 \end{aligned}$$

That is, the cost of a subtree T is the probability of accessing the root (i.e., the total probability of accessing the keys in the subtree) plus the cost of searching its left subtree and the cost of searching its right subtree. When we add the base case this leads to a recursive algorithm.

Exercise 59.4. When computing the cost for the tree, one thought would have been to compute the cost for each subtree of the root and add these two costs and the cost of the root ($p(S_r)$) to get the cost of this solution. Would this simpler approach have worked?

Solution. This does not work, because it does not take into account the increase in the cost of the keys in the subtrees by being one edge below the root.

Algorithm 59.2 (Recursive Optimal Binary Search Tree).

```

OBST S =
  if |S| = 0 then 0
  else ∑s ∈ S p(s) + mini ∈ {1...|S|} (OBST(S1,i-1) + OBST(Si+1,|S|))

```

Exercise 59.5. How would you return the optimal tree in addition to the cost of the tree?

Sharing. Without sharing, the recursive solution requires exponential work. To reduce the cost, we can take advantage of sharing among the calls to $OBST$ and represent the computation with a DAG. To bound the number of vertices in the DAG, we count the number of possible arguments to $OBST$. Because each argument is a contiguous subsequence from the original sequence S , we can count the total number of contiguous subsequences as

$$\sum_{i=0}^{n-1} (i+1) = n(n+1)/2.$$

The idea is that for each element with rank i , there are $i+1$ distinct starting positions $0 \dots i$ and summing over all gives us the bound. The number vertices in the DAG is therefore $O(n^2)$. Furthermore the longest path of vertices in the DAG is bounded by $O(n)$, because each call to $OBST$ removes one key, causing the to stop after n calls.

Total Work and Span. The cost of each vertex in the DAG (each recursive in our code not including the subcalls) is $O(|S|) = O(n)$, because we need to consider each position and sum up the probabilities of all the keys. The span is $O(\log n)$, because we can compute the minimum by using a reduction.

Now multiplying the number of vertices by the work of each gives us the upper bound of $O(n^3)$ on the work. For the span, we multiply the span each vertex with length of the path to obtain the upper bound $O(n \log n)$.

Algorithm 59.3 (Recursive Optimal Binary Search Tree (indexed)). We present a streamlined algorithm that uses integer indexes to identify subproblems. In particular we specify a subsequence of the original sorted sequence of keys S by its offset from the start (i) and its length l . We then get the following recursive routine.

```

 $OBST\ S =$ 
  let
    (* Determine  $OBST\ S[i, \dots, i+l-1]$  *)
     $OBST'\ (i, l) =$ 
      if  $l = 0$  then 0
      else  $\sum_{k=0}^{l-1} p(S[i+k]) + \min_{k=0}^{l-1} (OBST'(i, k) + OBST'(i+k+1, l-k-1))$ 
    in  $OBST\ (0, |S|)$  end
  
```

Similar Problems. This example of the optimal BST is one of several applications of dynamic programming which effectively based on trying all binary trees and determining an optimal tree given some cost criteria. Another such problem is the matrix chain product problem. In this problem one is given a chain of matrices to be multiplied ($A_1 \times A_2 \times \dots \times A_n$) and wants to determine the cheapest order to execute the multiplies. For example given

the sequence of matrices $A \times B \times C$ it can either be ordered as $(A \times B) \times C$ or as $A \times (B \times C)$. If the matrices have sizes 2×10 , 10×2 , and 2×10 , respectively, it is much cheaper to calculate $(A \times B) \times C$ than $A \times (B \times C)$. Since \times is a binary operation any way to evaluate our product corresponds to a tree, and hence our goal is to pick the optimal tree. The matrix chain product problem can therefore be solved in a very similar structure as the OBST algorithm and with the same cost bounds.

Chapter 60

Implementing Dynamic Programming

In previous chapters, we have considered several problems and showed that these problems admit a recursive solution, which via sharing, can be computed in polynomial work. Without sharing, the recursive solutions would require exponential work. In this chapter, we present two techniques for implementing such sharing. The two techniques called, [bottom-up](#), and [top-down](#), are similar but different enough to have each their own advantages.

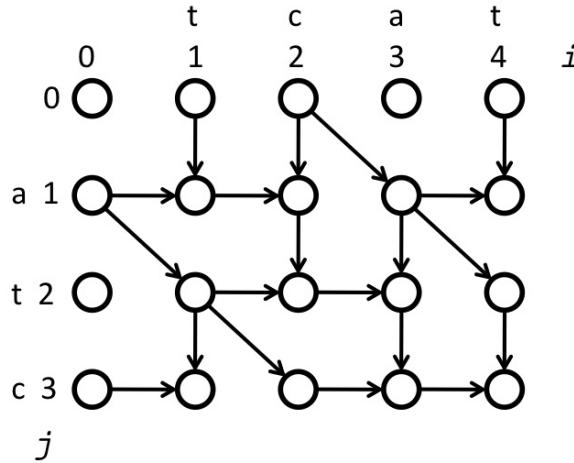
1 Bottom-Up Method

Constructing the DAG Bottom-Up. Given the DAG of a recursive solution, observe that the leaves of the DAG may be computed directly because they don't depend on any other results. As we compute each vertex, imagine "pebbling" the vertex so as to indicate that it is now available. Now, once the leaves are pebbled, any other vertex of the DAG, all of whose in-neighbors are pebbled may also be computed and pebbled. This process of pebbling a vertex whose in-neighbors are pebbled may be continued until all vertices of the DAG, including the root, are pebbled. The pebbling of the root, in turn, completes the computation and yields the result.

Depending on the topology of the DAG, there is usually plenty of freedom about which vertex or vertices, we can pebble. Often times, there are not just one but many vertices, all of which may be pebbled at once, i.e., in parallel. Depending on our goals, we may prefer one pebbling strategy over another, to optimize for various metrics such as work, space, and parallelism.

Example 60.1 (Minimum Edit Distance). Consider Minimum Edit Distance problem for

the two strings $S = \text{tcat}$ and $T = \text{atc}$. We can draw the DAG as follows.



In the DAG, there are three types of edges

1. down edges
2. horizontal edges, which go from left to right, and
3. diagonal edges.

The numbers represent the i and the j for that position in the string. We draw the DAG with the root at the bottom right, so that the vertices are structured the same way we might fill an array indexed by i and j .

As an example, consider $\text{MED}(4, 3)$. The characters $S[4]$ and $T[3]$ are not equal so the recursive calls are to $\text{MED}(3, 3)$ and $\text{MED}(4, 2)$. This corresponds to the vertex to the left and the one above. Now if we consider $\text{MED}(4, 2)$ the characters $S[4]$ and $T[2]$ are equal so the recursive call is to $\text{MED}(3, 1)$. This corresponds to the vertex diagonally above and to the left. In fact whenever the characters $S[i]$ and $T[j]$ are not equal we have edges from directly above and directly to the left, and whenever they are equal we have an edge from the diagonal to the left and above.

Based on the direction of the edges, we can pebble the vertices of the DAG in several different ways. For example, we can pebble the DAG row-wise, by pebbling the first row from left to right, and then the second row, and so on. Symmetrically, we can pebble the DAG column-wise, starting with the first column and pebbling from top to bottom, and then proceeding to the second and so on. The row-wise and column-wise pebbling orders both yield a sequential algorithm. Finally, we can pebble the dag diagonally from top left towards bottom right. In this order, each position within a diagonal may be pebbled in parallel, leading thus to a parallel algorithm.

Algorithm 60.1 (Bottom up MED). The pseudo-code for the bottom-up algorithm for MED is given below. This algorithm pebbles the DAG diagonally and stores the result of each vertex in a table M . Because the table is indexed by two integers, it can be represented by an array, which allows constant work random access. Each call to *diagonals* processes one diagonal and updates the table M . The size of the diagonals grows and then shrinks. We note that the index calculations are tricky.

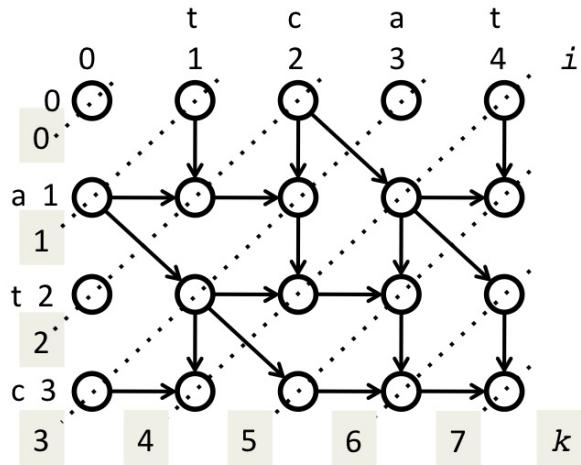
```

med S T =
let
  medOne M (i, j) =
    case (i, j) of
      (i, 0) => i
      (0, j) => j
      (i, j) =>
        if (S[i - 1] = T[j - 1]) then M[i - 1, j - 1]
        else 1 + min(M[i, j - 1], M[i - 1, j])

diagonals M k =
  if (k > |S| + |T|) then
    M
  else
    let
      s = max(0, k - |T|)
      e = min(k, |S|)
      MM = M ∪ {(i, k - i) ↦ medOne M (i, k - i) : i ∈ {s, ..., e}}
    in
      diagonals MM(k + 1)
    end
  in
  diagonals {} 0
end

```

Example 60.2. The drawing below illustrates the diagonals as pebbled by [the bottom-up algorithm](#) given above.



2 Top-Down Method: Memoization

Definition 60.2 (Memoization and Memo Table). The top-down approach is based on running the recursive code pretty much as is, but with some additional structure to store results as they are computed. In particular, we keep a mapping from the arguments of the recursive function to its solutions, and each time we return from a recursive call, we add the argument-result pair to the mapping. This way when we come across the same argument a second time we can just look up the solution instead of having to recompute it. This process is called *memoization*, and the table used to map the arguments to solutions is called a *memo table*.

Implementing Memoization. Being able to store and look up previous results efficiently is key to effective memoization. This in turn requires being able to check quickly if an argument is equal to a previous argument. If the arguments are complex data structures then such an equality test might be expensive on its own. Beyond checking for equality, looking up a previous result also requires storing the mapping in a table in which insertions and lookups are fast. Obvious choices are either balanced search trees or hash tables. The first requires a total order over the possible arguments, and an efficient way to check if one argument is “less” than another. The second requires that the arguments can be hashed, and to be efficient we need that it is unlikely that two different arguments hash to the same value.

Checking equality of arbitrary arguments, comparing them, or hashing them can be complicated. Fortunately in many cases, arguments can be arranged so that they are integers or small sequences of integers (tuples, triples, etc). Such integer-valued arguments simplify the implementation of such operations: we can cheaply compare or hash integer arguments.

Example 60.3. For the examples that we have consider so far including [the Subset Sum Problem](#), [the Minimum Edit Distance Problem](#), and the [Optimal Binary Search Problem](#), we have set up *surrogate* integer values to represent the input values.

Algorithm 60.3 (The Memo Function). To implement the memoization we define the following function:

```
memo f M a =
  case find M a of
    SOME(v) => (M, v)
  | NONE => let (M', v) = f M a
             in (update(M', a, v), v) end
```

In this function f is the function that is being memoized, M is the memo table, and a is the argument to f . This function simply looks up the value a in the memo table. If it exists, then it returns the corresponding result. Otherwise it evaluates the function on the argument, and as well as returning the result it stores it in the memo.

As an example, we can now write med using memoization as follows.

Algorithm 60.4 (Memoized MED). The pseudo-code below uses memoization to achieve sharing of solutions to subproblems.

```
med S T =
  let
    medOne M (i, j) =
      case (i, j) of
        (_, 0) => (M, i)
      | (0, _) => (M, j)
      | _ => if (S[i - 1] = T[i - 1]) then
                memo medOne M (i - 1, j - 1)
              else
                let
                  (M2, v1) = memo medOne M (i, j - 1)
                  (M3, v2) = memo medOne M2 (i - 1, j)
                in
                  (M3, 1 + min(v1, v2))
                end
      ( _, r ) = medOne {} (|S|, |T|)
  in
  r
end
```

Note. Note that the memo table M is threaded through the algorithm. In particular every call to MED takes a memo table as an argument, and returns a memo table as a result (possibly updated). Because of this passing, the code is purely functional.

Limitation of Top-Down Method. The top-down approach as described is inherently sequential. By threading the memo table through the computation, we force a total ordering on all calls to *med*. It is possible to solve these problems by combining several techniques

- using hidden state to implement the *memo* function such that the memo table is used implicitly
- using concurrent hash tables to store the results so that parallel calls might be in flight at the same time, and
- using synchronization variables to make sure that no function is computed more than once.

These techniques are all advanced techniques and are beyond the scope of this book.

Part XIII

Priority Queues

Chapter 61

Priority Queues

A priority queue maintains a set of elements from a total ordering, allowing at least insertion of a new element and deleting and returning the minimum element. We used priority queues in [priority-first graph search](#), in [Dijkstra's algorithm](#), and in [Prim's algorithm](#) for minimum spanning trees.

In this chapter we focus on meldable priority queues which support a *meld* function that can meld (or merge) two priority queues into one.

Data Type 61.1 (Meldable Priority Queue). Given a totally ordered set \mathbb{S} , a Meldable Priority Queue (MPQ) is a type \mathbb{T} representing subsets of \mathbb{S} , along with the following values and functions:

<i>empty</i>	$:$	\mathbb{T}
<i>singleton</i>	$:$	$\mathbb{S} \rightarrow \mathbb{T}$
<i>findMin</i>	$:$	$\mathbb{T} \rightarrow (\mathbb{S} \cup \{\perp\})$
<i>insert</i>	$:$	$\mathbb{T} \times \mathbb{S} \rightarrow \mathbb{T}$
<i>deleteMin</i>	$:$	$\mathbb{T} \rightarrow \mathbb{T} \times (\mathbb{S} \cup \{\perp\})$
<i>meld</i>	$:$	$\mathbb{T} \times \mathbb{T} \rightarrow \mathbb{T}$
<i>fromSeq</i>	$:$	$\mathbb{S} \text{ seq} \rightarrow \mathbb{T}$

The function *singleton*(e) creates a priority queue with just the element e . The function *findMin*(Q) returns the minimum element, or \perp (or `None`) if the queue is empty. The function *deleteMin*(Q) removes the minimum value and returns the new queue along with the value. If the queue is empty it returns \perp (or `None`). The *meld*(Q_1, Q_2) creates a priority queue with the union of the elements in Q_1 and Q_2 .

Algorithm 61.2 (Heapsort). A priority queue can also be used to implement a version of selection sort, often referred to as *heapsort*. The sort can be implemented by inserting all keys into a priority queue, and then removing them one by one, as follows.

```

sort S =
  let q0 = Sequence.iter PQ.insert PQ.empty S
  hsort q =
    case PQ.deleteMin q of
      (None, None) => []
      | (q', Some (v)) => Seq.append (v) (hsort q')
  in hsort q0 end

```

The heapsort algorithm is completely sequential, but given $O(\log n)$ work implementations of *insert* and *deleteMin*, and using lists or tree sequences (so the append is cheap) does optimal $O(n \log n)$ work.

Priority queues have many applications beyond heapsort and priority first search in graphs, including:

- Huffman codes,
- clustering algorithms,
- event simulation, and
- kinetic algorithms for motion simulation.

Another function that is sometimes useful is *decreaseKey* that decreases the value of a key.

1 Implementing Priority Queues

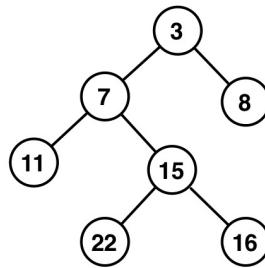
Priority queues can be implemented by using a variety of data structures.

Linked lists or Arrays. Perhaps the simplest implementation would be to use a sorted or unsorted linked list or an array. In such implementations, one of *deleteMin* and *insert* is fast and the other is slow, perhaps unacceptably so as it can take as much as $\Omega(n)$ work and span, where n is the size of the priority queue.

Balanced Trees. Another implementation is to use balanced binary search trees (e.g., treaps, or red-black trees). With balanced binary trees a *deleteMin* has to find the leftmost key and delete it. This can easily be implemented with $O(\log n)$ work and span. An *insert* is just a tree insert, and again takes $O(\lg n)$ work and span. However doing a *meld* on two heaps requires running a union on two trees, which can require $O(n)$ work if both heaps have size n .

Heaps. Many implementations of priority queues are based on the idea of a heap. As *min-heap* is a rooted tree such that the key stored at every node is less than or equal to the keys of all its descendants. Similarly a max-heap is one in which the key at a node is greater or equal to all its descendants. Compared to binary search trees, which need to maintain a total order over the search keys, heaps need only maintain a partial ordering over the keys. There are several kinds of heaps including complete binary heaps (briefly described below), leftist heaps (described in the next section), binomial heaps, pairing heaps, and Fibonacci heaps.

Example 61.1. An example min-heap illustrated.



Binary Heaps. A (complete) binary heap is a particular implementation of a heap that maintains two invariants:

- Shape property: A complete binary tree (all the levels of the tree are completely filled except the bottom level, which is filled from the left).
- Heap property.

Because of the shape property, a binary heap can be maintained in a sequence with the root in position 0 and the following simple functions for determining the left child, right child and parent of the node at location i :

$$\begin{aligned} \text{left } i &= 2 \times i + 1 \\ \text{right } i &= 2 \times i + 2 \\ \text{parent } i &= \lceil i/2 \rceil - 1 \end{aligned}$$

If the resulting index is out of range, then there is no left child, right child, or parent, respectively.

Insertion can be implemented by adding the new key to the end of the sequence, and then traversing from that leaf to the root swapping with the parent if less than the parent. Deletion can be implemented by removing the root and replacing with the last key in the sequence, and then moving down the tree if either child of the node is less than the key at the node. Binary heaps have the same asymptotic bounds as balanced binary search trees, but are likely faster in practice if the maximum size of the priority queue is known ahead of time. If the maximum size is not known, then some form of dynamically sized array is needed.

Cost Summary. The table below summarizes the costs of different implementations of priority queues, including leftist heaps covered later in this chapter, and their costs on the four key functions. Note that, a big win for leftist heaps is in the super fast *meld* operation—logarithmic as opposed to roughly linear in other data structures.

	<i>insert</i>	<i>deleteMin</i>	<i>meld</i>	<i>fromSeq</i>
Unsorted List	$O(1)$	$O(n)$	$O(m + n)$	$O(n)$
Sorted List	$O(n)$	$O(1)$	$O(m + n)$	$O(n \log n)$
Balanced Trees	$O(\log n)$	$O(\log n)$	$O(m \log(1 + \frac{n}{m}))$	$O(n \log n)$
Binary Heaps	$O(\log n)$	$O(\log n)$	$O(m + n)$	$O(n)$
Leftist Heap	$O(\log n)$	$O(\log n)$	$O(\log m + \log n)$	$O(n)$

2 Meldable Priority Queues

This section presents an implementation of a meldable priority queue that has the same work and span costs as binary search trees or binary heaps for insertion and deleting the minimum, but also has an efficient *meld*. In particular the *meld* function takes $O(\log n + \log m)$ work and span, where n and m are the sizes of the two priority queues to be merged.

The structure we consider is called a “leftist heap”, which is a binary tree that maintains the heap property, but unlike binary heaps, it does not maintain the complete binary tree property.

There are two important properties of a min-heap:

1. The minimum is always at the root.
2. The heap only maintains a partial order on the keys (unlike a BST that maintains the keys in a total order).

The first property allows us to access the minimum quickly, and it is the second that gives us more flexibility than available in a BST.

Let us consider how to implement the three operations *deleteMin*, *insert*, and *fromSeq* on a heap. Like *join* for binary search trees, the *meld* operation, makes the other operations easy to implement.

To implement *deleteMin* we can simply remove the root and *meld* the two subtrees rooted at the children of the root.

To implement *insert*(Q, v), we can just create a singleton node with the value v and then *meld* it with the heap for Q .

With *meld*, implementing *fromSeq* in parallel is easy using *reduce*:

```
fromSeq S =
  Seq.reduce Q.meld Q.empty (Seq.map Q.singleton S)
```

This is parallel, and assuming *meld* takes logarithmic work in the input sizes, this requires only $O(n)$ work and $O(\log^2 n)$ span.

Implementing Meld. The only operation we need to care about, therefore, is the *meld* operation. Suppose that we are given two heaps to meld. By inspecting the roots of the heaps, we can determine that the smaller one will be the root of the new melded heap. Thus, all we have to do now is construct the left and the right subtrees of the root. At this point, we have three trees to consider—the left-subtree and the right-subtree of the chosen root, and the other tree. Let us keep the left subtree in its place—as the left-subtree of the new root—and construct the right subtree by melding the two remaining trees. We can then construct the right subtree by a recursive application of the algorithm, until we encounter trivial trees such as an empty tree. In summary, to meld two heaps, we choose the heap with the smaller root and meld the other heap with its right subtree.

This idea leads to the following algorithm.

Data Structure 61.3 (Naïve Meldable Binary Heap).

```
datatype PQ = Leaf
  | Node of (key × PQ × PQ)

meld(A, B) =
  case(A, B)
  | (Leaf, Leaf) => A
  | (Leaf, _) => B
  | (Node(ka, La, Ra), Node(kb, Lb, Rb)) =>
    if ka < kb
    then Node(ka, La, meld(Ra, B))
    else Node(kb, Lb, meld(A, Rb))

empty = Leaf

singleton(k) = Node(k, Leaf, Leaf)

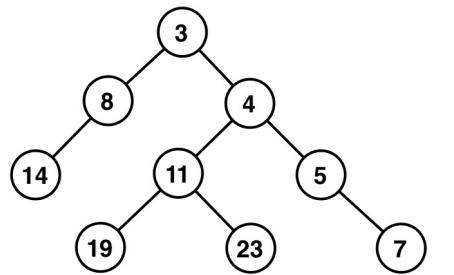
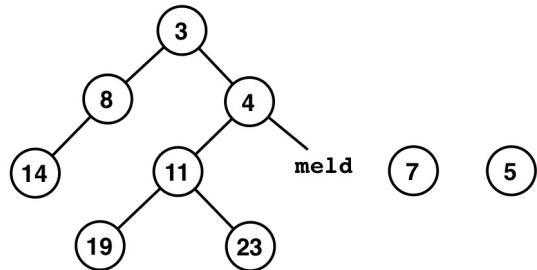
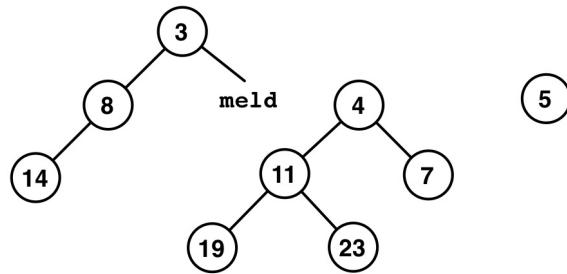
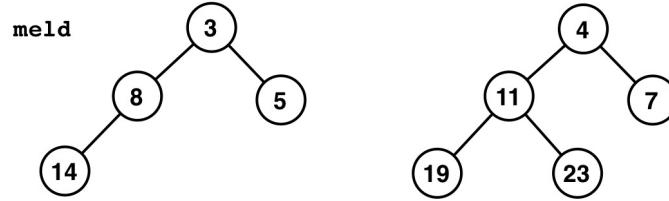
insert(k, Q) = meld(singleton k, Q)

deleteMin(Q) =
  case Q of
  | Leaf => (Q, None)
  | Node(k, L, R) => (meld(L, R), Some k)

fromSeq S =
  Seq.reduce meld empty (Seq.map singleton S)
```

Cost of Naïve Meld. The *meld* algorithm traverses the right spine of each tree (recall that the right spine of a binary tree is the path from the root to the rightmost node). The problem is that the tree could be very imbalanced, and in general, we can not put any useful bound on the length of these spines—in the worst case all nodes could be on the right spine. In this case the *meld* function could take $\Theta(|A| + |B|)$ work.

Example 61.2. An example *meld* operations on two heaps illustrated.



2.1 Leftist Heaps

Fixing the Imbalance Problem. It turns out there is a relatively easy fix to [the imbalance problem](#). The idea is to keep the trees so that the trees are always deeper on the left than

the right. To implement this idea, we associate a “rank” with each node in the binary tree and ensure during a meld operation that the tree is deeper on the left.

Definition 61.4 (Rank). The *rank* of a node x is

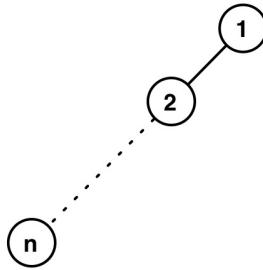
$$\text{rank}(x) = \# \text{ of nodes on the right spine of the subtree rooted at } x,$$

and more formally:

$$\begin{aligned} \text{rank}(\text{Leaf}) &= 0 \\ \text{rank}(\text{node}(_, _, R)) &= 1 + \text{rank}(R) \end{aligned}$$

Definition 61.5 (Leftist Property and Leftist Heaps). A leftist heap is a heap where the *leftist property* holds: for any node x in the heap, $\text{rank}(L(x)) \geq \text{rank}(R(x))$, where $L(x)$ and $R(x)$ are the left and the right child of x respectively.

Example 61.3. An example leftist heap.



Note. At an intuitive level, the leftist property implies that most of entries (mass) will pile up to the left, making the right spine of such a heap relatively short. Leftist heaps can therefore be extremely unbalanced. This is fine, however, because the data structure will always work on the shorter paths in the tree. We will make this idea precise in the following lemma which will prove later; we will see how we can take advantage of this fact to support fast meld operations.

Data Structure 61.6 (Leftist Heap).

```

datatype  $PQ$  = Leaf
           | Node of (int × key ×  $PQ$  ×  $PQ$ )
rank  $A$  = case  $A$  of
           Leaf = 0
           | Node( $r, \_, \_, \_$ ) =  $r$ 
makeLeftistNode ( $v, L, R$ ) =
  if (rank  $L < rank R$ )
  then Node(1 + rank  $L, v, R, L$ )
  else Node(1 + rank  $R, v, L, R$ )
meld ( $A, B$ ) =
  case ( $A, B$ ) of
    ( $\_, Leaf$ ) =>  $A$ 
    | ( $Leaf, \_$ ) =>  $B$ 
    | (Node( $\_, k_a, L_a, R_a$ ), Node( $\_, k_b, L_b, R_b$ )) =>
      if  $k_a < k_b$  then
        makeLeftistNode( $k_a, L_a, meld(R_a, B)$ )
      else
        makeLeftistNode( $k_b, L_b, meld(A, R_b)$ )

```

Leftist heap data structure extends the [naive data structure](#) in small but important ways to ensure efficiency.

Leftist heaps maintain a rank field for every node and maintain the leftist property by “piling” trees with larger ranks to the left via the *makeLeftistNode* function. The *meld* algorithm takes advantage of this by recurring only into the right subtree of a node, which is guaranteed to have smaller rank. Other operations can be implemented in terms of the *meld* operation as with the [naive data structure](#).

Note. The only real difference between [the naive data structure](#) and [leftist heaps](#) is that the latter uses *makeLeftistNode* to create a node and ensure that the resulting heap satisfies the leftist property (assuming the two input heaps L and R did). It makes sure that the rank of the left child is at least as large as the rank of the right child by switching the two children if necessary. To implement *makeLeftistNode* efficiently, the data structure also maintains the rank value on each node.

To analyze the cost of leftist heaps, we start by establishing [a key lemma](#) that states that leftist heaps have a short right spine, about $\log n$ in length. We then prove [the main theorem](#) that shows that the *meld* operation requires logarithmic work in the size of two heaps being melded.

Lemma 61.1 (Leftist Rank). In a leftist heap with n entries, the rank of the root node is at most $\log_2(n + 1)$.

Proof. To prove the lemma, we first prove a claim that relates the number of nodes in a leftist heap to the rank of the heap.

Claim: If a heap has rank r , it contains at least $2^r - 1$ entries.

To prove this claim, let $n(r)$ denote the number of nodes in the smallest leftist heap with rank r . It is not hard to convince ourselves that $n(r)$ is a monotone function; that is, if $r' \geq r$, then $n(r') \geq n(r)$. With that, we will establish a recurrence for $n(r)$. By definition, a rank-0 heap has 0 nodes. We can establish a recurrence for $n(r)$ as follows. Consider the heap with root note x that has rank r . It must be the case that the right child of x has rank $r - 1$, by the definition of rank. Moreover, by the leftist property, the rank of the left child of x must be at least the rank of the right child of x , which in turn means that $\text{rank}(L(x)) \geq \text{rank}(R(x)) = r - 1$. As the size of the tree rooted x is $1 + |L(x)| + |R(x)|$, the smallest size this tree can be is

$$\begin{aligned} n(r) &= 1 + n(\text{rank}(L(x))) + n(\text{rank}(R(x))) \\ &\geq 1 + n(r-1) + n(r-1) = 1 + 2 \cdot n(r-1). \end{aligned}$$

Solving the recurrence (a full binary tree of depth r), we get $n(r) \geq 2^r - 1$, which proves the claim. \square

To prove our lemma, i.e., the rank of the leftist heap with n nodes is at most $\log(n + 1)$, we simply apply the claim. Consider a leftist heap with n nodes and suppose it has rank r . By the claim it must be the case that $n \geq n(r)$, because $n(r)$ is the fewest possible number of nodes in a heap with rank r . But then, by the claim above, we know that $n(r) \geq 2^r - 1$, so

$$n \geq n(r) \geq 2^r - 1 \implies 2^r \leq n + 1 \implies r \leq \log_2(n + 1).$$

This concludes the proof that the rank of a leftist heap is $r \leq \log_2(n + 1)$. \square

Theorem 61.2 (Leftist Heap Work). If A and B are leftists heaps then the $\text{meld}(A, B)$ algorithm runs in $O(\log(|A|) + \log(|B|))$ work and returns a leftist heap containing all elements from A and from B .

Proof. The code for meld only traverses the right spines of A and B , advancing by one node in one of the heaps. Therefore, the process takes at most $\text{rank}(A) + \text{rank}(B)$ steps, and each step does constant work. Since both trees are leftist, by [the main lemma](#), the work is bounded by $O(\log(|A|) + \log(|B|))$. To prove that the result is leftist we note that the only way to create a node in the code is with makeLeftistNode . This function guarantees that the rank of the left branch is at least as great as the rank of the right branch. \square

Part XIV

Hashing

Chapter 62

Foundations

Hash functions and hash tables are widely used techniques in computer science. Even though their developments have historically been intertwined, hash functions today are used for many purposes beyond implementing hash tables. In this chapter, we describe hash functions as computational structures on their own right, and then discuss their use in [hash tables](#).

1 Introduction

In computer science, the term “hashing” refers to general idea of “mixing up” information content of an object to produce a “hash value” that has certain desirable properties such as compactness and some degree of randomness, so that different objects are unlikely to hash to the same value. Hash functions have found numerous applications in computer science.

Example 62.1 (A Culinary Analogy). To help develop some intuition about hashing, we can use a culinary analogy. Let’s imagine sitting at a big table on which rests several dozens of plates full of delicious pieces of fruits, apples, pears, oranges, pineapples, etc. Floating our gaze over the table and we are amazed by the plentifullness but also develop a sense of familiarity: we know each and every fruit on the table, even if we have never seen so many different kinds on the same table. We start entertaining the thoughts of savoring some but given that there are so many, we possibly can’t do that without wasting much. So we close our eyes and start dreaming about the more practical scenario of the same table but this time each kind of fruit is replaced by a small slice of it. Excited, we start savoring the fruits. After each bite, we are able to identify the fruit that we just tasted, excepting the golden delicious and the red delicious apples, which taste quite similar, and which we could have identified had they not been peeled.

Note. The point of [the example](#) is that we don’t need to eat a whole watermelon to identify it.

Exercise 62.1. Play the same game as in the culinary analogy but this time with more complex dishes imagine, for example, some yakitori, luohan zhai, pad sew ew, sundubu jjigae, fesenjan, pelmeni, mezze, lasagna, coq au vin, hamburger, etc.

Solution. It is not possible to “slice” these dishes and still be able to identify them, because of their fairly non-uniform structure. Instead we can “sample” by taking small pieces of the dish in many different places and then mixing them to construct a representative sample that represents the taste of the dish. This is similar to how hash functions work.

Applications of Hashing. Some applications of hashing include:

1. In our discussion of Treaps, we describe how hashing can be used to generate the “random” priorities. Our analysis assumed the priorities were truly random with respect to the keys, but it can be shown that a limited form of randomness that arises out of relatively simple hash functions is sufficient.
2. In cryptography so-called one-way hash functions can be used to hide information. These functions are easy to compute in the forward direction but hard to determine given the hash value, what object created it—and hence the name “one way”. One application is in digital signatures where a secure hash function is used to describe a large document with a small number of bits. These signatures can be used to *authenticate* the source of the document, ensure the *integrity* of the document as any change to the document invalidates the signature, and prevent *repudiation* where the sender denies signing the document.
3. Another application of one-way hash functions is for password protection. Instead of storing passwords in plain text, only the hash of the password is stored. To verify whether a password entered is correct, the hash of the password is compared to the stored value.
4. String commitment protocols use hash functions to hide what string a sender has committed so that the receiver gets no information. Later, the sender sends a key that allows the receiver to reveal the string. In this way, the sender cannot change the string once it is committed, and the receiver can verify that the revealed string is the committed string. Such protocols might be used to flip a coin across the Internet: The sender flips a coin and commits the result. In the meantime the receiver calls heads or tails, and the sender then sends the key so the receiver can reveal the coin flip.
5. Hashing can be used to approximately match documents, or even parts of documents. *Fuzzy matching* hashes overlapping parts of a document and if enough of the hashes match, then it concludes that two documents are approximately the same. Big search engines look for similar documents so that on search result pages they don’t show the many slight variations of the same document (e.g., in different formats). It is also used in spam detection, as spammers make slight changes to email to bypass spam filters or to push up a document’s content rank on a search results page. When looking for malware, fuzzy hashing can quickly check if code is similar to known malware. Geneticists use it to compare sequences of genes fragments with a known

sequence of a related organism as a way to assemble the fragments into a reasonably accurate genome.

6. Hashing is used to implement hash tables. In hash tables one is given a set of keys $K \subset \alpha$ and needs to map them to a range of integers so they can be stored at those locations in an array. The goal is to spread the keys out across the integers as well as possible to minimize the number of keys that collide in the array. You should not confuse the terms hash function and hash table. They are two separate ideas, and the latter uses the former.

Secure Hash Algorithm (SHA). Due to important role that hash functions play in security, their development has received much attention from the government. NSA (National Security Agency) in particular has developed the *SHA* family of hash function; SHA stands for “Secure Hash Algorithm.” The current generation *SHA-2* includes a set of cryptographic hash functions that are 224, 256, 384, and 512 bits, which are named respectively as SHA-224, SHA-256, SHA-384, and SHA-512.

SHA functions have the characteristic that a small change to the contents leads to a large change in the hash code, a.k.a., the *avalanche effect*.

Example 62.2. The text below illustrates the output of the deployment process of the Diderot project on Google Cloud. Deployment involves creating a virtual machine on the cloud, installing all the needed software, and then copying the Diderot source code from the local computer (a laptop in this case) onto the virtual machine on the cloud. The integrity of this copy operation is checked by using the “SHA256” hash function. The hash code, a.k.a., the digest, is computed before the transmission, transmitted along with the copied contents, and compared against a freshly computed digest for the transmitted contents. This check ensures that the transmission operation did not corrupt the contents being copied onto the cloud.

```
$ gcloud app deploy app_test.yaml --project diderot-cmu
Beginning deployment of service [default]
Building and pushing image for service [default]
Started cloud build [1e27052a-be74-4a38-be05-807042ca1146].
----- REMOTE BUILD OUTPUT -----
starting build "1e27052a-be74-4a38-be05-807042ca1146"
Successfully built 2abdfcbff888
Successfully tagged [...] appengine/default.20180414t103220:latest
PUSH
Pushing us.gcr.io/diderot-cmu/appengine/default.20180414t103220:latest
Repository: [us.gcr.io/diderot-cmu/appengine/default.20180414t103220]
21df82f90a72: Preparing
21df82f90a72: Waiting
67b0784928b9: Pushed
724aba9dc62d: Layer already exists
77c1da6d3730: Pushed
```

```

latest: digest: sha256:39df2d79576d3c204b8772150052610e65308706f169b72b0351c164a69c2
size: 3889

DONE
-----
Updating service [default]...done.
Stopping version [diderot-cmu/default/20180403t134333].
Updating service [default]...done.
Updating service [default]...done.
Deployed service [default] to [https://diderot-cmu.appspot.com]

```

2 Hash Functions

Definition 62.1 (Hash Function). A *hash function* h is a function from a domain \mathcal{U} , typically called the *universe* to a range $\mathbb{N}_{< m} = \{0, 1, \dots, m - 1\}$, where m is a positive natural number, i.e.,

$$h : \mathcal{U} \rightarrow \mathbb{N}_{< m}.$$

The size of the range m is usually significantly smaller than the size of the universe.

An element of the universe is called a *key*. An element of the range is called a *hash value*, *hash code*, or sometimes *digest*.

Definition 62.2 (Collisions). For distinct $x, y \in \mathcal{U}$, if $h(x) = h(y)$, then we say that $h(x)$ and $h(y)$ *collide*. We also say sometimes that x and y collide when the hash function is clear from the context.

Exercise 62.2. Consider any hash function h from a domain \mathcal{U} to a range $\mathbb{N}_{< m} = \{0, 1, \dots, m - 1\}$, where m is a positive natural number. Prove that if $|\mathcal{U}| > (n - 1)m$, then for any n , there exists a set of n keys $K \subseteq \mathcal{U}$ such that all keys in K hash to the same hash code.

Hint. Consider applying the pigeon-hole principle.

Properties of Good Hash Functions. A good hash function, $h : \mathcal{U} \rightarrow \mathbb{N}_{< m}$ should have at least the following qualities (informally):

- **Cost:** it should not be too difficult to compute, e.g., it should ideally require linear work in the size of the key.
- **Compactness:** it should require a small amount of memory to store and to compute.
- **Coverage:** its image should match its range, i.e., for any $0 \leq i < m$, there exists $x \in \mathcal{U}$ such that $h(x) = i$. In other words, the hash function should be surjective.

- **Collision Avoidance:** It should be unlikely that an arbitrary two keys map to the same hash value.
- **Mixing:** Given a small set of keys

$$\{x_1, \dots, x_{k-1}\} \subset \mathcal{U}$$

and their hash codes

$$h(x_1) \cdots h(x_{k-1}),$$

it should be difficult to predict the hash code $h(x_k)$ of any other key $x_k \notin \{x_0, x_1, \dots, x_{k-1}\}$. In a formal form, this is referred to as k -wise independence.

Example 62.3. Let \mathcal{U} be the set of all natural numbers and consider the following hash function

$$h(x) = x \bmod 4.$$

This is not a good hash function because, it only considers the least significant two bits of the key, and thus does not mix well. Just a few different applications of the hash could reveal the behavior of h , making it easy to predict the hash of any key in the universe.

More generally any hash function of the form

$$h(x) = x \bmod a^b,$$

where $a, b \in \mathbb{N}$, is not a good hash function for a similar reason: the function treats the input key as a number base a and takes the least significant b digits.

Example 62.4. Let \mathcal{U} be set of all natural numbers and let p be a prime number. Consider the hash function

$$h(x) = x \bmod p.$$

This is not a good hash function because it is relatively easy to predict by for example trying out some arguments of the form $x, x + 1, x + 2, x + 2^2, \dots$. Thus by observing the behavior of function on logarithmically many values, we can make a good guess for any value $y \in \mathcal{U}$. More generally, $h(x) = h(x + cp)$ for any c . In other words, the function does not mix well.

Example 62.5 (Random Hash Function). Consider a universe \mathcal{U} and a range $\mathbb{N}_{< m}$. We can construct a hash function for the universe by picking, for each key, a uniformly random natural number less than m . Such a random hash function has several important qualities.

- The function thus mixes its input keys well. Because the function is random, it is difficult to predict the value of any key from a small number of observations.
- The function evenly spreads collision over its range: for any $x, y \in \mathcal{U}$, if $x \neq y$ then the probability that $h(x)$ and $h(y)$ collide is $1/m$.

The problem with this hash function is that it is not compact: for each key in the universe, we have to remember the hash value that it maps to, which can require an extremely large amounts of space (memory).

Remark. Even though uniformly random hash functions are not compact, they are commonly assumed in the design of algorithms, because they offer a clean theoretical model. This is sometimes referred to as the *simple uniform hashing assumption*.

Definition 62.3 (Simple Uniform Hashing). The *simple uniform hashing* postulates that for any universe and any range there is a hash function that ensures that each key has equal probability of being mapped to any valid hash code independent of what the other elements are mapped to.

Collision Avoidance. One key challenge is designing hash functions is avoiding collisions. We can show, however, that collisions are impossible to avoid completely even for hash functions that have a relatively large range.

To see this, let's recall a fun fact: the *birthday paradox*. The “paradox” states that we only need 23 people in a room to have a 50% chance that at least two people have the same birthday. If we have 60 people, then we have a 99% chance that two people have the same birthday.

We can generalize the birthday paradox to show that when hashing to a range size m , we expect a collision to occur with only $\sqrt{\frac{1}{2}\pi m}$ keys.

A related question is how many keys do we need until every hash-code in the range is taken (mapped to). One can show that if the hash function has the range $\mathbb{N}_{< m}$ for some m , and for $\Theta(m \log m)$ distinct keys, then with constant probability (or even high-probability) every hash-code will be used. This property is related to the *coupon-collector's problem*.

Note. There is nothing paradoxical about the “birthday paradox”, which is simply a consequence of counting.

Exercise 62.3. Given a universe \mathcal{U} and a range $\mathbb{N}_{< m}$. Let h be a random hash function that is constructed by selecting for each key in the universe a random hash-code in the range. Prove that the hash function h satisfies

- for all $x \in \mathcal{U}$, and for all $i, 0 \leq i < m$,

$$\mathbf{P}[h(x) = i] = 1/m.$$

- for all $x, y \in \mathcal{U}$ such that $x \neq y$

$$\mathbf{P}[h(x) = h(y)] = 1/m.$$

Exercise 62.4. Given a universe \mathcal{U} and a range $\mathbb{N}_{< m}$, consider the set of all functions \mathcal{H} and let $h \in \mathcal{H}$ be a function that is uniformly randomly chosen from \mathcal{H} . Prove the following two statements

- For all $x \in \mathcal{U}$, and for all $i, 0 \leq i < m$,

$$\mathbf{P}_{h \in \mathcal{H}} [h(x) = i] = 1/m.$$

- For all $x, y \in \mathcal{U}$ such that $x \neq y$

$$\mathbf{P}_{h \in \mathcal{H}} [h(x) = h(y)] = 1/m.$$

As we will see, we refer to classes (sets) of hash functions for which the second property hold as “universal.”

Exercise 62.5. Given the universe \mathcal{U} and the range $\mathbb{N}_{\leq m}$, construct a set of hash functions \mathcal{H} such that for all $x \in \mathcal{U}$, and for all $i, 0 \leq i < m$,

$$\mathbf{P}_{h \in \mathcal{H}} [h(x) = i] = 1/m$$

but the following does not hold: for all $x, y \in \mathcal{U}$ such that $x \neq y$

$$\mathbf{P}_{h \in \mathcal{H}} [h(x) = h(y)] = 1/m.$$

Solution. Let \mathcal{H} be the set of all distinct constant functions, each of which maps all the elements in the universe to a single hash code in the range. The first property holds because for a uniformly randomly hash function, each hash code is equally likely to be selected. The second property does not hold, because each hash function is a constant function and thus the relevant probability is 1.0.

3 Universal Hashing

As illustrated by [the random hashing example](#), we can construct a hash function by selecting for each key a uniformly random hash code. Such a hash function minimizes the number of collision, the probability that any two keys collide is $1/m$ for the range $\mathbb{N}_{\leq m}$, but it does not accept a compact representation. Compactness fails, because we have to remember the mapping of keys to hash values explicitly. A natural question is whether it is possible to construct a hash function that is compact and cheap to compute but has the same guarantee over collisions. In this section, we shall see that this is indeed possible.

Definition 62.4 (Universal Class of Hash Functions). Let \mathcal{H} be a class (set) of hash function from a universe \mathcal{U} to the range $\mathbb{N}_{\leq m}$ for some m . We say that \mathcal{H} is *universal* if the probability that two distinct keys of the universe collide under a uniformly randomly chosen hash function is a most $1/m$, i.e.,

$$\forall x, y \in \mathcal{U}, \text{ such that } x \neq y : \mathbf{P}_{h \in \mathcal{H}} [h(x) = h(y)] \leq 1/m.$$

There are many different techniques for obtaining classes of universal hash functions. We will state here, without proof, a couple classes of universal hash functions that are commonly used.

Theorem 62.1 (Multiplicative Hashing with Offset)). Consider a finite universe $\mathcal{U} \subset \mathbb{N}$ and any range $\mathbb{N}_{< m}$. Let p be a prime number that is greater or equal to any key in the universe. For integers a and b , $0 < a < p$ and $0 \leq b < p$, let

$$h_{a,b}(x) = (ax + b \bmod p) \bmod m.$$

The class of hash functions \mathcal{H} defined as

$$\mathcal{H} = \{h_{a,b}(x) \mid 0 < a < p, 0 \leq b < p\}$$

is universal.

The [multiplicative-hashing theorem](#) makes it relatively easy to construct a class of compact and efficient hash functions that are universal. One somewhat concerning assumption could be that we need a prime number larger than the keys in the universe. The next theorem eliminates this assumption by allowing us to work with essentially any prime number equal to the range of the hash value instead of the potentially much larger universe.

Theorem 62.2 (Dot-Product Hashing). Let m be a prime number and r be a positive integer. Consider the universe $\mathcal{U} = \mathbb{N}_{< m^r}$ and the range $\mathbb{N}_{< m}$. For any natural number a , $0 \leq a < m^r$, let

$$h_a(x) = \left(\sum_{i=0}^{r-1} a_i \cdot x_i \right) \bmod m,$$

where a_i and x_i denote the i^{th} digit of a and x in base m .

The class of hash functions \mathcal{H} defined as

$$\mathcal{H} = \{h_a(x), \mid 0 \leq a < m^r\}$$

is universal.

Intuition. The idea behind the theorem is to read the keys of the universe as numbers in base m , which is a prime number. We select r to be big enough such that all keys are natural numbers less than m^r . Given some $0 \leq a < m^r$, we then define the hash function $h(x)$ to the sum of the products of digits of a and the key x modulo p .

Remark. The theorem fixes the hash codes to be numbers congruent to a prime m , i.e., the integers between 0 and $m - 1$. For the theorem to be effective, we would therefore need to select m to be close to the number of distinct hash codes that we are interested in.

Example 62.6 (Universal Hashing for Strings). Dot-product hashing yields a natural hash function for strings. Let r be the maximum length of the strings and interpret each character of the string as a natural number. Select a prime p to bound the value of each character and to be large enough to reduce the probability of collision to be small.

For any length- r string a , define

$$h_a(x) = \left(\sum_{i=0}^{r-1} a_i \cdot x_i \right) \bmod p,$$

where a_i and x_i denote the i^{th} character of a_i and x_i .

The class of hash functions \mathcal{H} defined as

$$\mathcal{H} = \{h_a(x) \mid a \text{ is a string of length } r\}$$

is universal.

Bounding the Number of Collisions. The quantity of interest in understanding the effectiveness of hashing is the number of collisions that a key may be involved in. To understand this quantity, let's define $C_{x,y}$ to be an indicator random variable such that

$$C_{x,y} = \begin{cases} 1 & \text{if } h(x) = h(y) \\ 0 & \text{otherwise.} \end{cases}$$

Because $C_{x,y}$ is an indicator random variable its expectation is the same as the probability that it is 1. Assuming universal hashing (or more strongly simple uniform hashing), we know that for any $x \neq y$

$$\mathbf{E}[C_{x,y}] = \mathbf{P}[C_{x,y} = 1] = \frac{1}{m}.$$

Suppose now that we have n keys that we wish to hash and we wish to bound the number of collisions that any key is involved in. Define the random variable C_x to be total number of keys other than x that collide with x .

$$\begin{aligned} C_x &= \sum_{y, y \neq x} C_{x,y} \\ \mathbf{E}[C_x] &= \sum_{y, y \neq x} \mathbf{E}[C_{x,y}] \\ \mathbf{E}[C_x] &\leq \frac{n-1}{m} \quad [\text{by the union bound}] \\ &\leq \frac{n}{m}. \end{aligned}$$

We can similarly bound the total number of collisions across all keys. To this end let C be the random variable denoting the total number of collisions. Because there are exactly $\binom{n}{2}$ distinct pairs of keys that could collide, we can bound the expectation of C as

$$\begin{aligned} C &= \sum_{x, y, x \neq y} C_{x,y} \\ \mathbf{E}[C] &= \sum_{x, y, x \neq y} \mathbf{E}[C_{x,y}] \\ \mathbf{E}[C] &\leq \binom{n}{2} \cdot \frac{1}{m} \quad [\text{by the union bound}] \\ \mathbf{E}[C] &\leq \frac{n^2}{2m}. \end{aligned}$$

Let's summarize these bounds.

- If the range of the class of the universal hash functions m is large compared to the number keys n , then we expect a relatively small number of collisions for any key.
- Summed over all keys, the expected number of collisions is a bit larger, but still proportional to the square of the number of keys hashed.

In many cases, it will be sufficient to bound the expected number of collisions per key by a constant, and thus it is sufficient to consider $m = O(n)$, e.g., $m = 2n$.

In other cases, it is desirable to reduce the number of collisions further, so that for example, we have only a few collisions across all keys. This can be achieved by choosing the range of our functions to be larger, e.g., for $m = n^2$, the expected number of collisions is $1/2$.

Exercise 62.6. Consider a universal class of hash function \mathcal{H} for some universe U and $T \subseteq U$ be any subset of U . Prove that for any key $x \in U$, the number of keys in T whose hash value collides with x under a uniformly randomly chosen hash function $h \in \mathcal{H}$ is constant.

Chapter 63

Hash Tables

This chapter presents a key data structure in computer science, hash tables, and several different ways of implementing them by using [hash functions](#).

Definition 63.1 (Hash Tables). A *hash table* is an abstract data type that supports the following operations on key-value pairs, where keys are drawn from a universe (e.g., integers, strings, records) and accept an equality test (function).

- The *createTable* function takes as argument an equality function on keys, a hash function generator that returns a hash function given a natural number that specifies the size of its range, and an initial size and creates an empty hash table of that given size.
- The *insert* function takes as argument a hash table and a key-value pair and inserts the pair into the table.
- The *lookup* function takes a hash table and a key and returns the value for the key stored in the hash table if any, or indicates that the key is not found.
- The *loadAndSize* function takes a hash table and returns the number of key-value pairs stored in the table and the size of the table.
- The *resize* function takes a hash table and a new size, usually double or half the current size, and returns a new hash table that contains the same key-value pairs as in the original paper, nothing less and nothing more.

Hash tables enable us to maintain a dynamically changing mapping from keys to values. In this sense, they are a special case of table data type that we have seen in the past. They differ from tables in several ways.

- Hash tables don't require the keys to be totally ordered and don't demand a comparison function on keys. Instead they require the keys to be hashable.

- They support a narrower set of operations that revolve around insertions and deletions.

Design of Hash Tables: Nested and Flat. The main challenge in designing hash tables is resolving collisions, where two keys hash to the same hash code. There are several well-studied collision resolution strategies.

- **Nested tables:** use an outer table to map each hash code to an inner table that contains the key-value pairs that map to that hash code. The inner table can be represented in several different ways, including as a lists, or as another hash table. If the inner table is a list, the technique is called “separate chaining.”
- **Flat Tables or Open Addressing:** Use a single, flat table mapping keys to entries.

Between the two possibilities, the nested tables are more flexible and more amenable to analysis. The analysis of hash tables typically depends on how “crowded” the table is, which is quantified by the “load factor.”

Definition 63.2 (Load Factor). For a hash table of size m with n key-value pairs stored in the table, the *load factor*, written as α , is defined as

$$\alpha = \frac{n}{m}.$$

1 Nested Tables

1.1 A Parametric Design

The basic structure of a nested table is naturally recursive: keep an outer table that maps each key to an inner table, which can be structures as desired. Given a key, we use an outer hash function to determine the inner table that the key maps to. We then use the inner table to resolve the collisions. Because the outer hash function maps keys to a prefix of the natural numbers, the domain of the outer table is natural numbers less than the current size m . We can thus use an array to represent the outer table and locate the inner table efficiently with constant work.

Example 63.1. Consider the following table mapping keys to values.

```
{'aa' ↦ 'a', 'bb' ↦ 'b', 'cc' ↦ 'b', 'dd' ↦ 'd', 'ee' ↦ 'e',
 'ff' ↦ 'f', 'gg' ↦ 'g', 'hh' ↦ 'h', 'ii' ↦ 'i', 'jj' ↦ 'j'}.
```

Let

$$h(x) = \left(\sum pos(x[i]) \right) \bmod m$$

be a hash function that maps each string to a hash code by summing up the positions of its characters in the alphabet (counting from zero) modulo the table size $m = 5$.

We can use the following nested hash table for our key-value pairs.

$$\begin{aligned} \{0 \mapsto \{ & 'aa' \mapsto 'a', 'ff' \mapsto 'f' \}, \\ 1 \mapsto \{ & 'dd' \mapsto 'd', 'ii' \mapsto 'i' \}, \\ 2 \mapsto \{ & 'bb' \mapsto 'b', 'gg' \mapsto 'g' \}, \\ 3 \mapsto \{ & 'ee' \mapsto 'e', 'jj' \mapsto 'j' \}, \\ 4 \mapsto \{ & 'cc' \mapsto 'c', 'hh' \mapsto 'h' \}, \\ \} \end{aligned}.$$

Bounding the Size Inner Tables. The key quantity of interest in understanding the efficiency of nested tables is the size of an inner table. Since any inner table stores the key-value pairs that collide with each other, we can bound the their size in terms of conflicts.

Conflicts can be very high in general but not so if we use universal hash functions. Recall that we bounded the expected number of conflicts for any key x in terms of the

$$\mathbf{E}[C_x] \leq \frac{n}{m} = \alpha.$$

This means that the size of an inner table in $O(1 + \alpha)$ in expectation.

Thus, if we ensure that the load factor of the table remains a constant by making sure for example that $n \leq cm$, for some constant c , then we know that the size of each inner table is constant in expectation.

Keeping the Load Factor Small. Because the size of the table m is fixed and n changes, the load factor can increase as a result of insertions. To keep the load factor from growing, we can resize the table, by for example doubling it every time the load factor exceeds the desired bound. The cost of the resize operation can be amortized because doubling ensures that the new keys pay for the old ones.

Exercise 63.1. Describe how you can implement the hash table interface specified above by using nested tables. For the inner tables use the Table ADT that you have learned about earlier but leave out the implementation and thus the costs unspecified.

Exercise 63.2. Does it make sense to reduce the size of the hash table? If so, then under what conditions and how?

1.2 Separate Chaining

Definition 63.3 (Separate Chaining). The [parametric implementation](#) uses an array to represent the outer table but does not specify how to implement the inner table.

Perhaps the simplest way to implement the inner table is to use a list representation that stores at each node one or more key-value pairs. Such an implementation is called *separate chaining* or simply as *chaining*.

In separate chaining, insertion proceeds by first locating the inner table, a list, and then inserting the key-value pair at the head of the list; this requires constant work. Lookups could proceed by first looking up the list using the hash code of the key being searched, and then searching for the key from the head of the list using the key equality function; this requires work linear in the length of the list. Deletions could proceed by first looking up the key and then deleting it, again requiring work linear in the length of the list.

Example 63.2. Recall the example, where we are given the following table mapping keys to values.

```
{'aa' ↦ 'a', 'bb' ↦ 'b', 'cc' ↦ 'b', 'dd' ↦ 'd', 'ee' ↦ 'e',
 'ff' ↦ 'f', 'gg' ↦ 'g', 'hh' ↦ 'h', 'ii' ↦ 'i', 'jj' ↦ 'j'}.
```

Let

$$h(x) = \left(\sum pos(x[i]) \right) \bmod m$$

be a hash function that maps each string to a hash code by summing up the positions of its characters in the alphabet (counting from zero) modulo the table size $m = 5$.

Using chaining, we represent this table as

```
{0 ↦ [('aa', 'a'), ('ff', 'f')],
 1 ↦ [('dd', 'd'), ('ii', 'i')],
 2 ↦ [('bb', 'b'), ('gg', 'g')],
 3 ↦ [('ee', 'e'), ('jj', 'j')],
 4 ↦ [('cc', 'c'), ('hh', 'h')]
}.
```

Cost Analysis of Separate Chaining. As described, *insert*, *delete*, and *lookup* operations all spend $O(1 + \alpha)$ work traversing the chain. Because the hash function takes constant work, total expected work for these operations is $O(1 + \alpha)$. Thus assuming that α is a constant, the total expected work for these operations is $O(1)$.

Exercise 63.3. Describe how to implement the *resize* operation and bound its cost.

1.3 Perfect Hashing

Nested tables with [separate chaining](#) gives us expected constant time bounds on the key hash table operations. Consider now the special case where we know exactly the set of keys that we wish to store in the table. In other words, we only wish to perform *lookup* operations on a static set of keys.

In this special case, we can achieve worst case work for *lookup* operations by using a nested hash table, where the inner table itself is a hash table with chaining. To ensure constant-work in the worst case, we will make sure that all chains (lists) in the inner table has length at most one, i.e., they contain a single key-value pair or they are empty. In other words, for the inner table, we guarantee the absence of collisions.

To this end, we are going to use a result from universal hashing. Recall that for a hash table with range-size m , the expected number of collisions among n key is

$$\mathbf{E}[C] \leq \frac{n^2}{2m}$$

and the probability that there is a collision is at most

$$\frac{n^2}{2m}.$$

Thus, if we choose $m = n^2$, then we have

$$\mathbf{E}[C] \leq \frac{1}{2}$$

and the probability that there is a collision is at most

$$\frac{1}{2}.$$

This is a lot of space of course and can be unaffordable, but we can imagine applying this approach to each inner table, because we expect them to be small.

Algorithm 63.4 (Perfect Hashing). We are given n key-value pairs that we wish to store in a hash table. We can construct a perfect hash table for the set of key-value pairs as follows.

- Choose a uniformly random hash function h from a universal hash family with a range of n , the total number of key-value pairs that we wish to store.
- Use h for the outer table and determine the key-value pairs for each inner table T_i , $0 \leq i < n$. Let n_i be the number of key-value pairs in the inner table.
- For each inner table T_i with n_i key-value pairs, select a hash function whose range is n_i^2 from a universal hash family. Check that the hash function guarantees absence of collisions for the keys in T_i . If there are collisions, choose another hash function. Step when a hash function h_i that guarantees the absence of collisions is found.

- Represent each inner table T_i by using a hash table with chaining and the hash function h_i that guarantees absence of collisions.

Example 63.3. Consider the following table mapping keys to values with $n = 10$ key-value pairs.

$$\{('aa' \mapsto 'a'), ('bb' \mapsto 'b'), ('cc' \mapsto 'b'), ('dd' \mapsto 'd'), ('ee' \mapsto 'e'), ('ff' \mapsto 'f'), ('gg' \mapsto 'g'), ('hh' \mapsto 'h'), ('ii' \mapsto 'i'), ('jj' \mapsto 'j')\}.$$

Let

$$h(x) = \left(\sum pos(x[i]) \right) \bmod m$$

be a hash function that maps each string to a hash code by summing up the positions of its characters in the alphabet (counting from zero) modulo the table size m .

In perfect hashing we select $m = n$, thus $m = 10$. First, we build the outer table, determining for each hash-code the key-value pairs that map to that hash code. This gives us the following hash table.

$$\begin{aligned} & \{0 \mapsto \{('aa' \mapsto 'a'), ('ff' \mapsto 'f')\}, \\ & \quad 1 \mapsto \{\}, \\ & \quad 2 \mapsto \{('bb' \mapsto 'b'), ('gg' \mapsto 'g')\}, \\ & \quad 3 \mapsto \{\}, \\ & \quad 4 \mapsto \{('cc' \mapsto 'c'), ('hh' \mapsto 'h')\}, \\ & \quad 5 \mapsto \{\}, \\ & \quad 6 \mapsto \{('dd' \mapsto 'd'), ('ii' \mapsto 'i')\}, \\ & \quad 7 \mapsto \{\}, \\ & \quad 8 \mapsto \{('ee' \mapsto 'e'), ('jj' \mapsto 'j')\}, \\ & \quad 9 \mapsto \{\} \end{aligned} \}.$$

Next, we select for each inner table a new hash function uniformly at random from our class of universal functions. In our case, we can select hash functions of the form Let

$$h_i(x) = \left(\sum pos(a \cdot x[i]) \right) \bmod m_i,$$

where m_i is the size of the i^{th} inner hash table and $0 \leq a < m_i$. Recall that m_i 's are square of the number of key-value pairs in that table. In our case, we have $m_0 = m_2 = m_4 = m_6 = m_8 = 4$ and $m_1 = m_3 = m_5 = m_7 = m_9 = 0$.

For simplicity, we shall choose the following hash function for all inner tables.

$$h_i(x) = \left(\sum pos(x[i]) \right) \bmod 4$$

This gives us the perfect hashing for each inner table.

$$\begin{aligned} 0 &\mapsto \{0 \mapsto ['aa' \mapsto 'a'], 2 \mapsto ['ff' \mapsto 'f']\}, \\ 1 &\mapsto \{\}, \\ 2 &\mapsto \{0 \mapsto ['gg' \mapsto 'g'], 2 \mapsto ['bb' \mapsto 'b']\}, \\ 3 &\mapsto \{\}, \\ 4 &\mapsto \{0 \mapsto ['cc' \mapsto 'c'], 2 \mapsto ['hh' \mapsto 'h']\}, \\ 5 &\mapsto \{\}, \\ 6 &\mapsto \{0 \mapsto ['ii' \mapsto 'i'], 2 \mapsto ['dd' \mapsto 'd']\}, \\ 7 &\mapsto \{\}, \\ 8 &\mapsto \{0 \mapsto ['ee' \mapsto 'e'], 2 \mapsto ['jj' \mapsto 'j']\}, \\ 9 &\mapsto \{\}, \\ \} &. \end{aligned}$$

Analysis of Perfect Hashing. By construction, perfect hashing guarantees the absence of collisions in the inner table, it therefore supports $O(1)$ lookup time.

Perhaps the most interesting quantity that we are interested in is the size of the hash table, including of course the outer and the inner tables. We prove that this is linear, i.e., $O(n)$, in expectation when storing n key-value pairs. To establish this bound we need to sum up the sizes of all inner tables, each of which is quadratic in the number of key-value pairs that it stores. To this end, imagine the complete directed graph with n vertices, where each vertex represents a key-value pair stored and each distinct pair of vertices is connected by two edges, one in each direction, and each vertex has one self-loop. Observe now that the total space of the inner tables corresponds exactly to the number edges between vertices that are within the same inner table. Next observe that the two endpoints of an edge are within the same inner table if the outer hash code of the corresponding keys collide. In other words, the number of such edges is two times the total number of collisions, plus n to account for the self loops. Because we use universal hashing, we know that the total expected number of collisions in the outer hash table is $\frac{n^2}{2m}$, where m is the size of the range of the hash function. Since we know that $m = n$, the bound on the expected space usage is

$$2 \frac{n^2}{2n} + n = 2n.$$

Exercise 63.4. What is the probability that a perfect hash table uses more than $2n$, say $16n$ space?

Exercise 63.5. Analyze the work required to construct a hash table for n key-value pairs.

2 Flat Tables or Open Addressing

When using flat tables, we store all key-value pairs in a single table that maps keys to key-value pairs. We minimize the impact of collisions by keeping the load factor of the table low. Because the table is flat, however, keys that map to the same hash-code can interact in interesting ways, e.g., when two keys collide and map to the same hash code, only one could be mapped by the hash code. We therefore have to be careful about dealing with collisions.

The basic idea behind flat hash tables is to perform a sequence of “probes” until a suitable position in the hash table is found. More precisely, consider a hash table of size m . To insert a key-value pair into the table, we repeatedly *probe* the table in different position until we find an available position and claim that position. We refer to the sequence of probes as a *probe sequence*, and for correctness require it to try out all positions in the table. As we shall see, probe sequences can be generated in several different ways.

Definition 63.5 (Probe Sequence). For a hash table of with m entries, a *probe sequence* is a permutation of $\mathbb{N}_{\leq m} = \{0, 1, \dots, m - 1\}$.

2.1 A Parametric Implementation of Flat Tables

Data Structure 63.6 (Parametric Flat Hash Tables). We present an implementation of open addressing by assuming that for the current hash function of size m , we have m hash function

$$h_0(x), h_1(x), \dots, h_{m-1}(x)$$

that generate the probe sequence for any key x .

To specify the implementation, we assume that we are given the types for keys and values, *key* and *value* respectively. We also assume the existence of a function *eqKey* for checking that two keys are equal.

We define the type of a hash table as

$$\begin{aligned} \text{type } entry &= \text{Empty} \\ &\quad | \\ &\quad \text{Dead} \\ &\quad | \\ &\quad \text{Live of } \text{key} \times \text{value} \\ \text{type } hashTable &= \text{entry array} \end{aligned}$$

The first variant *Empty* of *entry* indicates an empty entry, the second *Dead* indicates that the entry has been deleted, and the third indicates that the entry is live and has the given key and value.

Keeping track of deleted entries enables the implementation to find a key when its probe sequence interleaves with the probe sequence of another key, which may later be deleted.

```

1  lookup ( $T, k$ ) =
2  let
3  lookup'  $i$  =
4  case  $T[h_i(k)]$  of
5   $Empty \Rightarrow None$ 
6   $| Dead \Rightarrow lookup'(i + 1)$ 
7   $| Live(k', v') \Rightarrow$ 
8   $if keyEqual(k, k') then Some v$ 
9   $else lookup'(i + 1)$ 
10 in lookup' 0 end

```

The *insert* function is very similar to *lookup* but it updates the table with the given key-value pair. For simplicity, we assume that key is not in the table, which can be checked by using a *lookup* first.

```

1  insert ( $T, k, v$ ) =
2  let
3  insert'  $i$  =
4  case  $T[h_i(k)]$  of
5   $Empty \Rightarrow update(T, h_i(k), Live(k, v))$ 
6   $| Dead \Rightarrow update(T, h_i(k), Live(k, v))$ 
7   $| Live(k', v') \Rightarrow$ 
8   $if keyEqual(k, k') then ()$ 
9   $else insert'(i + 1)$ 
10 in insert' 0 end

```

The delete function is similar. For simplicity, we assume that the key is indeed in the table; this can be checked by performing a *lookup* first.

```

1  delete ( $T, k$ ) =
2  let
3  delete'  $i$  =
4  case  $T[h_i(k)]$  of
5   $Empty \Rightarrow ()$ 
6   $| Dead \Rightarrow delete(i + 1)$ 
7   $| Live(k', v') \Rightarrow$ 
8   $if keyEqual(k, k') then update(T, k, Dead)$ 
9   $else delete'(i + 1)$ 
10 in delete' 0 end

```

Example 63.4. Let T be the following table

0	1	2	3	4	5	6	7
B		D	E	A		F	

if key E has the probe sequence

$$\langle 7, 4, 2, \dots \rangle,$$

$lookup(T, E)$ would first visit position 7, which is full, and then position 4 where it finds E.

Example 63.5. Let T be the following table, where * indicates a deleted entry.

0	1	2	3	4	5	6	7
	B		D	*	A		F

if key D has the probe sequence

$$\langle 7, 4, 3, \dots \rangle,$$

$lookup(T, E)$ would first visit position 7, which is full, and then position 4, which is deleted, and then position 3, where it finds D.

Example 63.6. Suppose the hash table has the following keys:

0	1	2	3	4	5	6	7
	B			E	A		F

Now if for a key D we had the probe sequence $\langle 1, 5, 3, \dots \rangle$, then we would find position 1 and 5 full (with B and E) and place D in position 3 giving:

0	1	2	3	4	5	6	7
	B		D	E	A		F

Cost Analysis of Flat Tables. The cost analysis of flat tables becomes tricky because of the impact of deleted keys and the interaction between keys that collide. Here we present an informal analysis for a table of size m with n stored key-value pairs, where $n \leq m$, and thus the load factor $\alpha \leq 1$. We make several assumptions.

- We assume that the probe sequence executed by an operation is a uniformly randomly chosen permutation of $0, \dots, m - 1$.
- We assume simple uniform hashing, which postulates that each key is given a uniformly randomly chosen hash-code independently of all the others.
- We assume that there are no deletions, and thus the table entries are either empty or occupied but not marked deleted or dead.

Under these assumptions, let's first bound the number of probes needed until we find an empty cell in the hash table. This is a Bernoulli trial with a success probability of $1 - \alpha$. Therefore, the expected number of trials is $\frac{1}{1 - \alpha}$.

This means that an insertion and an unsuccessful lookup will require $\frac{1}{1-\alpha}$ work in expectation.

For a successful lookup, consider some key x that is the i^{th} key to be inserted into the table. To insert the key, we first find an empty cell, which requires $\frac{1}{1-i/m} = \frac{m}{m-i}$, because the load factor for the table is i/m . Now, observe that the probe sequence for a key is always deterministic. Thus a successful search will repeat the same probe sequence as the insertion and find the key. Thus, we the successful search for the i^{th} key requires $\frac{m}{m-i}$ work in expectation.

We can write the average expected cost over all keys as

$$\frac{1}{n} \sum_{i=0}^n \frac{m}{m-i}.$$

This is bounded by

$$\frac{1}{\alpha} \left(\ln \frac{1}{1-\alpha} \right)$$

because

$$\begin{aligned} \frac{1}{n} \sum_{i=0}^n \frac{m}{m-i} &= \frac{m}{n} \left(\sum_{i=0}^m \frac{1}{i} - \sum_{i=0}^{m-n} \frac{1}{i} \right) \\ &= \frac{m}{n} (H_m - H_{m-n}) \\ &\leq \frac{m}{n} \left(\ln \frac{m}{m-n} \right) \\ &\leq \frac{1}{\alpha} \left(\ln \frac{1}{1-\alpha} \right). \end{aligned}$$

(The bound on $H_m - H_{m-n}$ can be obtained by using integration.)

Exercise 63.6. Show that the parametric implementation of the flat hash table above can be implemented by using just a single higher-order function, which in turn can be used to implement *lookup*, *insert*, and *delete*.

Exercise 63.7. Complete the implementation of the parametric flat hash table by describing the algorithms and writing the pseudo-code for the remaining operations, e.g., *resize*.

2.2 Linear Probing

Definition 63.7 (Linear Probing). *Linear probing* is a flat table implementation, where the probe sequence is defined by m hash function of the form

$$h_i(k) = (h(k) + i) \mod m.$$

Each position in the table determines a single probe sequence, so there are only m possible probe sequences.

Primary Clustering. The problem with linear probing is that keys tend to cluster. It suffers from *primary clustering*: Any key that hashes to any position in a cluster (not just collisions), must probe beyond the cluster and adds to the cluster size. Worse yet, primary clustering not only makes the probe sequence longer, it also makes it more likely that it will lengthen further.

What is the impact of clustering for an unsuccessful search? Let's consider two extreme examples when the table is half full, $\alpha = 1/2$ (or equivalently, $m = 2n$). Clustering is minimized when every other location in the table is empty. In this case, the average number of probes needed to insert a new key k is $3/2$: One probe to check cell $h(k)$, and with probability $1/2$ that cell is full and it needs to look at the next location which, by construction, must be empty. In the worst case, all the keys are clustered, let's say at the end of the table. If k hashes to any of the first n locations, only one probe is needed. But hashing to the n^{th} location would require probing all n full locations before finally wrapping around to find an empty location. Similarly, hashing to the second full cell, requires probing $(n - 1)$ full cells plus the first empty cell, and so forth. Thus, under uniform hashing the average number of probes needed to insert a key would be

$$1 + [n + (n - 1) + (n - 2) + \dots + 1]/m = 1 + n(n + 1)/2m \approx n/4$$

Even though the average cluster length is 2, the cost for an unsuccessful search is $n/4$. In general, each cluster j of length n_j contributes $n_j(n_j + 1)/2$ towards the total number of probes for all keys. Its contribution to the average is proportional the *square* of the length of the cluster, making long cluster costly.

Remark. Although it can perform poorly in the worst case, linear probing is known to be quite competitive, when the load factors are in the range 30-70% as clusters tend to stay small. In addition, a few extra probes is mitigated when sequential access is much faster than random access, as in the case of caching. Because of primary clustering, though, it is sensitive to quality of the hash function or the particular mix of keys that result in many collisions or clumping. Therefore, it may not be a good choice for general purpose hash tables.

2.3 Quadratic Probing

Definition 63.8 (Quadratic Probing). Quadratic probing is a flat-table implementation, where the probe sequence cause probes to move away from clusters, by making increasing larger jumps. The probe sequence is defined by functions

$$h_i(k) = (h(k) + i^2) \mod m.$$

Definition 63.9 (Secondary Clustering). Although, quadratic probing avoids primary clustering, it still has *secondary clustering*: when two keys hash to the same location, they have the same probe sequence. Since there are only m locations in the table, there are only m possible probe sequences.

One problem with quadratic probing is that probe sequences do not probe all locations in the table. But since there are $(p+1)/2$ quadratic residues when p is prime, we can guarantee that an empty cell can be found unless the table is crowded.

Lemma 63.1. If m is prime and the table is at least half empty, then quadratic probing will always find an empty location. Furthermore, no locations are checked twice.

Proof. Consider two probe locations $h(k) + i^2$ and $h(k) + j^2$, $0 \leq i, j < \lceil m/2 \rceil$. Suppose the locations are the same but $i \neq j$. Then

$$\begin{aligned} h(k) + i^2 &\equiv (h(k) + j^2) \pmod{m} \\ i^2 &\equiv j^2 \pmod{m} \\ i^2 - j^2 &\equiv 0 \pmod{m} \\ (i - j)(i + j) &\equiv 0 \pmod{m} \end{aligned}$$

Therefore, since m is prime either $i - j$ or $i + j$ are divisible by m . But since both $i - j$ and $i + j$ are less than m , they cannot be divisible by m . This is a contradiction.

Thus the first $\lceil m/2 \rceil$ probes are distinct and guaranteed to find an empty location. \square

Linear versus Quadratic Probing. Compared to linear probing, computing the next probe in quadratic probing is only slightly more expensive, because it can be computed without using multiplication:

$$\begin{aligned} h_i - h_{i-1} &\equiv (i^2 - (i-1)^2) \pmod{m} \\ h_i &\equiv (h_{i-1} + 2i - 1) \pmod{m}. \end{aligned}$$

Unfortunately, requiring that the table remains less than half full makes quadratic probing space inefficient.

2.4 Double Hashing

Definition 63.10. Double hashing is a flat-table implementation that uses two hash functions $h(\cdot)$ and $hh(\cdot)$, one to find the initial location to place the key and a second to determine the size of the jumps in the probe sequence. The probe sequence is defined by hash functions of the form

$$h_i(k) = (h(k) + i \cdot hh(k)) \pmod{m}.$$

Keys that hash to the same location, are likely to hash to a different jump size, and so will have different probe sequences. Thus, double hashing avoids secondary clustering by providing as many as m^2 probe sequences.

How do we ensure every location is checked? Since each successive probe is offset by $hh(k)$, every cell is probed if $hh(k)$ is relatively prime to m . Two possible ways to ensure

$hh(k)$ is relatively prime to m are, either make $m = 2^k$ and design $hh(k)$ so it is always odd, or make m prime and ensure $hh(k) < m$. Of course, $hh(k)$ cannot equal zero.

The main advantage with double hashing is that it allows for smaller tables (higher load factors) than linear or quadratic probing, but at the expense of higher costs to compute the next probe. The higher cost of computing the next probe may be preferable to longer probe sequences, especially when testing two keys equal is expensive.

3 Concluding Remarks

Hash functions are a very important technique in computer science that is used in a very broad array of applications. Although their development was intertwined with that of hash tables in the initial years of computer science, recent developments in hash functions are primarily driven by security, privacy, and error detection and correction.

Hash tables are classic data structures that are broadly employed in many real-world systems. They are a classic example of a space-time tradeoff: increase the space so table operations are faster; decrease the space but table operations are slower.

Of the different methods for implementing hash table, [nested tables](#) and [separate chaining](#) are perhaps the simplest and are less sensitive to the quality of the hash function or load factors. They are therefore usually the choice when it is unknown how many and how frequently keys may be inserted or deleted from the hash table.

[Flat tables and open addressing](#) can be more space efficient than nested tables, though the space efficiency of nested tables can also be improved by using blocking techniques. [Linear probing](#) has the advantage that it has small constants and works well with modern architectures due to better locality (the memory locations accessed are typically on the same cache line). But it suffers from [primary clustering](#), which means its performance is sensitive to collisions and to high load factors.

[Quadratic probing](#), on the other hand, avoids primary clustering, but still suffers from [secondary clustering](#), and requires rehashing as soon as the load factor reaches 50%.

[Double hashing](#) reduces clustering and thus makes high load factors feasible, but finding suitable pairs of hash functions is somewhat more difficult and increases the cost of a probe.

Part XV

Concurrency

Chapter 64

Threads, Concurrency, and Parallelism

This chapter presents an important abstraction in computer science, threads, and how they can be used to write concurrent and parallel programs.

1 Threads

Definition 64.1 (Thread). A *thread*, short for *thread of execution*, is a computation that executes a given piece of code. A program that uses multiple threads is called *multithreaded*.

We consider two operations on threads: `spawn` and `sync`.

- The operation `spawn` takes an expression, creates a thread to execute that expression, and returns the thread. Once spawned the thread starts executing concurrently with other threads in the program.
- The operation `sync` takes a thread and waits until that thread completes its execution.

Example 64.1. The piece of code below spawns two threads and assigns them the task of computing the n^{th} and $2n^{th}$ Fibonacci number. Once spawned the two threads execute concurrently.

```
let t = spawn (lambda () .fib n)
  u = spawn (lambda () .fib 2n)
  (( ), ( )) = (sync t, sync u)
in  () end
```

The function *fib* may be implemented as

```
fib x =
  if x ≤ 1 then x
  else fib (x - 1) + fib (x - 2)
```

Example 64.2. In the [example above](#), the threads compute the desired Fibonacci number but has no way of communicating the result back. The piece of code below modifies the example slightly to report the results back via references. The `sync` operations ensure that the results are computed by waiting for the threads to complete.

```
let (r, s) = (ref 0, ref 0)
  t = spawn (lambda () . r ← fib n)
  u = spawn (lambda () . s ← fib 2n)
  (( ), ( )) = (sync t, sync u)
  in (!r, !s)
  end
```

Definition 64.2 (Thread Scheduler). Multithreaded programs rely on a *thread scheduler* or *scheduler* for short, to execute the spawned threads to completion. At a given time, a scheduler can execute any subset of the spawned threads that are ready to execute, i.e., they are not waiting other threads.

Example 64.3 (One-Processor Schedule). Consider executing the multithreaded program for [computing two Finonacci numbers](#) using one processor. The thread scheduler will start by executing the “main thread” that spawns the two threads. After the two threads are spawned, the scheduler can choose to execute any one of them for any duration of time. If it divides its time evenly between the two threads, then the first one will finish, leaving only the second thread to work on. The scheduler will therefore work on the second thread, and when it completes, it will return to the main thread and return the computed results.

Example 64.4 (Two-Processor Schedule). Consider executing the multithreaded program for [computing two Finonacci numbers](#) using two processors.

The scheduler could execute the two threads in parallel until the first finishes. This would speedup the completion of the program, somewhat, but not dramatically, because one processor will be idle most of the time—this program does not have enough parallelism.

2 Concurrency and Parallelism

Definition 64.3 (Concurrency). An algorithmic problem is a *concurrency problem* if its specification involves multiple things happening at the same item.

Example 64.5 (Scheduling as a Concurrency Problem). The problem of scheduling things such as tasks to be completed in a manufacturing facility while respecting the dependencies between them, threads, requests in a web server, or jobs in a server farm, all fall into the category of “scheduling problems.” These problems are all concurrency problems.

- The problem of designing a [thread scheduler](#) is a concurrent problem because it involves potentially multiple threads that execute at the same time.
- Web servers must promptly serve many users varying from just a few to thousands and even sometimes millions at a time. Scheduling requests arriving at a web server is a concurrency problem, because requests could arrive and be processed at the same time.

Example 64.6 (Interactive Systems). Many problems involving interactive systems such as operating systems, games, and browser, are all naturally concurrent.

- To specify the behavior of an operating system, we need to talk about multiple events happening at the same time, such as multiple programs executing at once, an event such as a network event or a user input happening at the same time, etc.
- To specify a multiplayer game, we need to talk about multiple users, their actions, and how the system (game) reacts to them. Even a simple game such as single-user game involves simultaneous events such as the actions of the user and the computations performed by the graphics card. Nearly all reasonably sophisticated games are therefore naturally concurrent.
- A modern browser must handle events from the user as well as through the network, e.g., a request completing as the user also creates a new tab in their browser.

Solving Concurrency Problems. Multithreading is usually the technique of choice for solving concurrency problems such as those in the [example above](#). Using a [thread scheduler](#), such multithreaded implementations can be made to work on sequential hardware, such as a computer with any number of processors.

Example 64.7. Many OS's today are designed to exploit multicore chipsets that can provide anywhere from 2-8 cores even in small devices such as laptops and mobile phones. Such devices also typically include a multicore Graphics-Processing-Units (GPUs) that typically include anywhere from a handful to hundreds and even thousands of cores or processors.

Definition 64.4 (Parallelism). An algorithm is parallel if it performs multiple tasks at the same time. Both concurrent and non-concurrent problems typically accept parallel algorithms (solutions).

Example 64.8. Parallel implementations are commonly used in solving computationally challenging problems.

- Games with rich graphical interfaces perform many graphics computations (triangulation, rasterization, shading, etc) in parallel on the GPU.
- In automotive industry, “self-driving” systems use parallel processors and GPUs to perform various image processing and machine learning tasks.

- Scientific simulations for predicting the outcome of physical phenomena (e.g., weather and climate, inter-planetary forces in space, and fluid dynamics) require loads of computations. Parallel computers make such computations practically feasible and usable by speeding them thousands of times. For example, the European Center for Medium-Range Weather Forecasts use supercomputers with more than 10,000 processors.

Remark (Concurrency versus Parallelism). Concurrency and parallelism are largely orthogonal concepts:

- concurrency is a property of a “problem”,
- parallelism is a property of an implementation or a “solution.”

Concurrency and parallelism can and usually co-exist. Concurrent problems usually accept parallel solutions. As we saw in this class, many non-concurrent problem, which can be specified as mathematical functions mapping an input to an output, also accept parallel solutions.

Example 64.9 (Parallel Fibonacci). The function *fib* below computes the x^{th} Fibonacci number in parallel.

```

fib x dest =
  if x ≤ 1 then
    dest ← x
  else
    let
      (da, db) = (ref 0, ref 0)
      a = spawn (lambda () .fib (x - 1) da)
      b = spawn (lambda () .fib (x - 2) db)
      ((), ()) = (sync a, sync b)
    in
      dest ← !da + !db
    end

```

Example 64.10 (Parallel Fibonacci in SPARC). Using SPARC, we can write the [parallel Fibonacci function](#) as

```

fib x =
  if x ≤ 1 then
    x
  else
    let (ra, rb) = (fib (x - 1)) || (fib (x - 2)) in
      ra + rb
    end.

```

Note. The SPARC code above can be viewed as “syntactic sugar” for the code in [parallel Fibonacci example](#). Essentially any realistic implementation of SPARC will translate the code into a multithreaded version that uses `spawn` and `sync` functions, which will then be compiled into a parallel executable.

3 Mutable State and Race Conditions

Parallel versus Sequential Semantics. We can think of any parallel SPARC program as a sequential program by replacing parallel tuples with sequential ones.

The semantics of these two programs are strongly related: if the original parallel program is pure (purely functional), and does not use side effects, then corresponding sequential program is “observationally equivalent” to the parallel one.

This means that the two programs yield the same output on the same inputs.

Definition 64.5 (Sequential Elision). For any SPARC program, there is a corresponding sequential program called its *sequential elision*, that is obtained by replacing `par` (or `||`) with simple sequential pairs.

Example 64.11 (Parallel Fibonacci: Sequential Elision). Using SPARC, we can write the parallel Fibonacci function as

```
fib x =
  if x ≤ 1 then
    x
  else
    let (ra,rb) = (fib (x - 1)) || (fib (x - 2)) in
      ra + rb
    end.
```

The sequential elision of the function `fib` is:

```
fib x =
  if x ≤ 1 then
    x
  else
    let (ra,rb) = (fib (x - 1),fib (x - 2)) in
      ra + rb
    end.
```

Convince yourself that these two programs are observationally equivalent.

Mutable State. The observational equivalence property between SPARC programs and their sequential elision breaks in the presence of side effects, or mutable state. In fact, when a multithreaded or parallel program uses *mutable state*, i.e., references and destructive updates, reasoning about its correctness and efficiency can become difficult.

Example 64.12 (Concurrent Writes). Consider executing the following program

```
let x = ref 0
  (((),()) = (x ← 1) || (x ← 2)
  in print x end
```

The expressions $x \leftarrow 1$ and $x \leftarrow 2$ both update the same variable x . Because the two expressions are parallel, they can take effect in any order (even on a single processor) and the outcome of the program is non-deterministic. This program can therefore output 1 or 2.

Example 64.13 (Concurrent Additions). Consider the following program

```
let x = ref 0
  (( ), ( )) = (x ← x + 1) || (x ← x + 1)
  in  print x end.
```

The two instances of the expression $x \leftarrow x + 1$ are parallel and they update the same shared variable x . It might seem that because both expressions increment the same variable, the output of the program will always output 2. But this is not quite correct, because the expression $x \leftarrow x + 1$ is not atomic: the two instances can both start by reading x as 0 and the update it to 1. The program can therefore output 1 or 2.

Definition 64.6 (Data Race). We say that a *data race* occurs if multiple threads access the same piece of data and at least one of the threads update or write to the data. Data races usually lead to non-deterministic outcome, which is in many cases an error condition. In some cases, a data race does not impact adversely the correctness of a program. Such data races are called *benign*.

Example 64.14. The two examples above— [parallel writes](#) and [parallel additions](#)— both include a data race.

Note (Determinacy Race). Data races are sometimes called *determinacy races* because they could lead to non-deterministic outcome.

Why Use Mutable State. Given the complexities of reasoning about concurrent programs that use mutable state, we might reasonably ask:

Why use mutable state at all and why not program purely functionally?

There are two important reasons.

- On modern computers, it is impossible to avoid mutable state completely. Even if a program is purely functional, it still has to allocate memory and write into it. Therefore, at some level of abstraction, we must operate on mutable state, even when it is “hidden” behind the abstraction of purely functional programming.
- Mutable state enables implementing certain operations including purely functional ones more efficiently. For example, updating a single position in an array requires copying an array if we are not allowed to mutate it.

Races are Considered Harmful. It is now broadly accepted that data races are harmful and should be avoided to the extent possible. The reason for this is that when a program contains races, understanding its behavior becomes extremely difficult, because we have to reason about an exponentially growing number of possible interactions. For example, if we have 100 threads each of which execute 10 instructions that can cause races, then the total number of interactions we must consider are 10^{100} more than the number of atoms in the universe.

Remark. When using strongly typed languages, we can easily avoid races by being judicious about use of mutable data. For example, if we program in Parallel ML and use only the pure functional subset of the language, then we are guaranteed to avoid races. This is the reason for why we use a strongly typed language in this class.

Remark. The history of computer science is full of examples demonstrating the power of abstraction. Can you imagine for example, that before “structured programming” languages were accepted, programs were written with “GOTO” statements? Can you imagine that it was only 20 years ago that pros and cons of garbage collection were passionately debated among both researchers and practitioners? Today, hardly anyone would write code with “GOTO” statements and hardly anyone would question the benefits of garbage collection.

Teach them [young students], as soon as possible, a decent programming language that exercises their power of abstraction.

– Edsger Dijkstra

Chapter 65

A Practical and Efficient Implementation of Sequences

This chapter presents an implementation of the [Sequence ADT](#) in terms of mutable arrays, meeting the [cost specification](#) given for array sequences.

1 Motivation

Sequences as Arrays. In this section, we will explore how to efficiently implement the [Sequence ADT](#) in terms of mutable arrays, meeting the [cost specification](#) given for array sequences. In contrast to the parametric implementation given already in Chapter 17, our goal here is dive deeper by more carefully considering practical issues such as [data races](#) and performance optimizations.

Asymptotic Performance is only the Beginning. To develop parallel codes that are practically efficient, we need to pay attention to a variety of factors that extend beyond asymptotic analysis. In this chapter we will be focusing primarily on two details: granularity control (i.e. amortizing the cost of parallelism) and reducing the number of writes to memory. These factors are not too difficult to reason about, and can have a huge impact on performance.

Array Primitives. We will use the following to create, modify, and access arrays:

- `alloc n` allocates a fresh array of length n

- $a[i] \leftarrow x$ writes x at index i of array a .
- $a[i]$ reads from array a at index i .

Parallel Loops. It is useful to have a “parallel for loop” primitive, written

`parfor i from ℓ to h do e_i end`

which in parallel executes e_i for each i in the range $\ell \leq i < h$. The intent is for each e_i to perform a side-effect, perhaps by modifying index i of some array. Note that the index range is inclusive on the bottom, and exclusive on the top, such that $h - \ell$ is the number of iterations of the loop.

The cost semantics of `parfor` are very simple. The work is just the total work of all of the iterations. For span, it is the max of the spans of each iteration.

$$W(\text{parfor } i \text{ from } \ell \text{ to } h \text{ do } e_i \text{ end}) = \sum_{\ell \leq i < h} W(e_i)$$

$$S(\text{parfor } i \text{ from } \ell \text{ to } h \text{ do } e_i \text{ end}) = \max_{\ell \leq i < h} S(e_i)$$

Remark. The `parfor` should be interpreted as spawning $n = h - \ell$ `threads`, one for each iteration. In practice, it is common to implement `parfor` in terms of the usual “par” primitive (`-||-`) by recursively dividing the range (ℓ, h) in half and executing the two halves in parallel. This technically will spawn more than n threads (up to $2n - 1$), but still asymptotically meets the cost specification for `parfor` given above, except for an additional $O(\log n)$ additive factor in the span. In some computational models, however, it is advantageous to take `parfor` as a true primitive and assume it does not incur additional overhead in the span.

Sequential Loops. We will also use a standard sequential for-loop, written

`for i from ℓ to h do e_i end`

which is analogous to `parfor` except that it executes the iterations e_i sequentially, beginning with e_ℓ and ending with e_{h-1} . The work of a sequential for-loop is the same as the parallel for-loop, but the span differs: because the iterations are executed sequentially, we take the sum of the spans of the iterations. Note that there could still be parallelism within a single iteration of a sequential loop.

2 Tabulate and Granularity Control

Algorithm 65.1 (Tabulate). To implement tabulate, we can begin by allocating an array, and then in parallel initialize each one of its elements, before finally returning the array.

```
tabulate f n =
  let a = alloc n in
    parfor i from 0 to n do
      a[i]  $\leftarrow$  f i
    end;
    a
  end
```

Exercise 65.1. Argue that `tabulate` has no `data races`.

The Cost of Parallelism. The `tabulate` implementation is “fully parallel”: it spawns n `threads`, one for each call to *f*. If the cost of each $f(i)$ is significantly more than the cost of spawning (and then later synchronizing with) a single thread, then this tabulate implementation will perform well in practice, because the cost of the thread will be amortized. But we have to be careful, because threads are not cheap: each thread requires allocating memory to keep track of the state of the thread (its current code pointer, all of its “local variables”, etc.). How well does `tabulate` perform when *f* is very cheap? If we ran `tabulate (lambda i. 2i + 1) n`, which only performs a few arithmetic instructions per call to *f*, it would not perform very well in practice. This is because the cost of each thread will outweigh the amount of work the thread performs.

Definition 65.2 (Granularity). The *granularity* of a thread is the amount of work that the thread performs before it completes.

In order for it to be “profitable” to spawn a thread, we have to guarantee that the granularity of the thread is large enough to offset the cost of the thread itself.

Definition 65.3 (Granularity Control). Any technique which amortizes the cost of spawning threads (e.g. by forcing all threads to have at least some minimum satisfactory granularity) is called a *granularity control* technique.

Granularity Control by Blocking. In `tabulate`, imagine breaking up the array into a many small *blocks*, each of which is a contiguous segment of the array. We can control granularity by only spawning one thread per block, and processing within each block sequentially. To asymptotically guarantee low span, we should keep the sizes of the blocks small—say, a constant—but still large enough to amortize the cost of a thread.

Algorithm 65.4 (Blocked Tabulate). We begin by allocating an array of length *n*, and then we break up the array into $\lceil n/B \rceil$ blocks, each of size *B*, and finally process the blocks in parallel. The *i*th block starts at index $i \cdot B$, and ends at $i \cdot B + B$, except for potentially the

last block which might be smaller (when n is not divisible by B) and need to be cut off at index n . The block size B can be chosen to be any constant at least 1.

```
blockedTabulate f n =
let
  a = alloc n
  B = ... // block size, must be  $\geq 1$ 
in
  parfor i from 0 to  $\lceil \frac{n}{B} \rceil$  do
    for j from  $iB$  to  $\min(iB + B, n)$  do
      a[j]  $\leftarrow f$  j
    end
  end;
  a
end
```

3 Scan

We have seen before in Algorithm 17.10 that *scan* can be implemented in terms of *tabulate*, by contracting elements pairwise, recursively scanning, and finally expanding the output. Is this implementation efficient in practice? To answer that question, it is helpful to compare it against the fastest possible sequential algorithm we can devise.

Algorithm 65.5 (Fast Sequential Scan). A fast sequential algorithm for *scan* is simply just to iterate over the input array, accumulating a prefix sum (with respect to the parameter f) along the way, and outputting the current prefix sum at each index.

```
sequentialScan f b s =
let
  p = alloc |s|
  loop (i, x) =
    if  $i \geq |s|$  then x else ( p[i]  $\leftarrow x$ ; loop ( $i + 1$ ,  $f(x, s[i])$ ) )
  t = loop (0, b)
in
  (p, t)
end
```

Exercise 65.2. Show that Algorithm 17.10 performs approximately three times many array updates as Algorithm 65.5. Assume that f performs no array updates.

Solution. We can easily see that Algorithm 65.5 performs exactly n array updates for an input of size n .

The number of array updates performed by Algorithm 17.10 can be written as a recurrence

$u(n)$:

$$\begin{aligned} u(0) &= 0 \\ u(1) &= 1 \\ u(n) &= u(n/2) + n/2 + n \\ &= u(n/2) + 3n/2 \end{aligned}$$

In the third case, we pay $n/2$ updates to tabulate the contracted sequence of size $n/2$, then we recurse on this sequence, and finally we pay n writes to expand the output.

Clearly $u(n)$ is at least $3n/2$, and as n approaches infinity it approaches $3n$:

$$\begin{aligned} u(n) &\leq 3n/2 + 3n/4 + 3n/8 + \dots \\ &= 3n(1/2 + 1/4 + 1/8 + \dots) \\ &= 3n \end{aligned}$$

Estimating Performance by Counting Memory Updates. Scan is a great example of an algorithm where, by measuring the number of memory updates it performs, we can get a sense for how well it will perform in practice. This is particularly true when the cost of the associative function f given as argument is cheap, such as in `scan + 0 s`. The reason is that, on modern hardware, the cost of a single arithmetic instruction is significantly less than the cost of a single memory update.

Scan with Fewer Memory Updates. To reduce the number of writes performed by scan, we can once again use the `blocking` technique. We saw previously that blocking could be used to control granularity, but here we will see that it can also be used to reduce the number of memory updates. The idea simple: in our implementation of scan, rather than recurse on a sequence of size $n/2$, we can instead recurse on a much smaller sequence, say of size $n/1000$. We will accomplish this with blocking: we can split the array into n/B blocks of size B , compute the “sum” (w.r.t. the associative function f) of each block in parallel, recursively determine the prefix-sums of the block-sums, and finally expand into the output by processing the blocks again in parallel. The code Algorithm 17.10 is just a special case of this idea when $B = 2$.

Algorithm 65.6 (Blocked Scan). In the following, B is the block size which can be any constant at least 2. The function $O(k)$ gives the starting index of the k^{th} block; we cap this at n so that $O(k + 1)$ can be used as the ending index of the the k^{th} block. The function `sum` is used to compute the sum of of the blocks when we construct s' . The sequence r gives the prefix-sums of the block-sums; these are passed to the function `loop` to rescan each block

and write to the output p .

```

blockedScan f id s =
  case |s|
  | 0 => (< >, id)
  | 1 => (< id >, s[0])
  | n =>
    let
      B = ... // block size, must be  $\geq 2$ 
      O(k) = min(kB, n) // block offsets
      sum (i, j, x) =
        if i  $\geq j$  then x else sum (i + 1, j, f(x, s[i]))
      s' = < sum (O(k), O(k + 1), id) : 0  $\leq k < \lceil n/B \rceil \rangle$ 
      (r, t) = blockedScan f id s'
      p = alloc n
      loop (i, j, x) =
        if i  $\geq j$  then () else (p[i]  $\leftarrow x$ ; loop (i + 1, j, f(x, s[i])))
    in
      parfor k from 0 to  $\lceil n/B \rceil$  do
        loop (O(k), O(k + 1), r[k])
      end;
      (p, t)
    end
  
```

Exercise 65.3. Derive a closed-form expression for the number of updates performed by Algorithm 65.6 in terms of the block size B . Show that even for a small choice of B , we can make Algorithm 65.6 perform close to an optimal number of memory updates.

Solution. In this case we have that the number of updates $u(n)$ performed for an input of size n is given by:

$$\begin{aligned}
 u(0) &= 0 \\
 u(1) &= 1 \\
 u(n) &= u(n/B) + n + n/B \\
 &= u(n/B) + n \frac{B+1}{B}
 \end{aligned}$$

We can solve this recurrence similarly to how we solved Example 65.2:

$$u(n) \leq \frac{B+1}{B-1} n$$

For $B = 100$, this gives us $u(n) \leq 1.02n$, which is only 2% off of optimal (the best we could hope for is exactly n writes, because the output is size n). At $B = 1000$, this comes down to only 0.2%.

4 Filter

Algorithm 65.7 (Simple Parallel Filter). To implement filter, a simple approach is as follows. We scan to compute, for each element, the number of elements before it that satisfy the predicate. This is given by sequence C . The scan also computes m , the total number of elements that satisfy the predicate. To construct the result r , we allocate an array of length m and then write each satisfying element $s[i]$ at $r[C[i]]$.

```
filter p s =
  let
     $(C, m) = \text{scan} + 0 \langle (\text{if } p(x) \text{ then } 1 \text{ else } 0) : x \in S \rangle$ 
     $r = \text{alloc } m$ 
  in
     $\text{parfor } i \text{ from } 0 \text{ to } |s| \text{ do}$ 
       $\text{if } p(s[i]) \text{ then } r[C[i]] \leftarrow s[i] \text{ else } ()$ 
    end;
     $r$ 
  end
```

Exercise 65.4. Describe a simple modification to Algorithm 65.7 that makes exactly n calls to the predicate function.

Solution. Rather than checking $p(s[i])$ to see if $s[i]$ should be written to the output, we can instead compare $C[i]$ against $C[i + 1]$. When $C[i] \neq C[i + 1]$, we know that $s[i]$ must satisfy the predicate. For $i = n - 1$, we don't have a $C[i + 1]$ to compare against, but in this case we can instead compare against m .

Exercise 65.5. Argue that Algorithm 65.7 performs $2n + m$ array updates for an input of size n and output of size m . You may assume the scan performs an optimal number of updates.

Solution. To construct the input the scan, we perform exactly n updates. The scan itself performs n updates. Then to construct the output we perform m updates.

Exercise 65.6. Describe a sequential algorithm for filter which performs exactly m array updates, where m is the size of the output.

Solution. First, pass over the input to count how many elements satisfy the predicate (no array updates required). Next, allocate the output array. Finally, do a second pass over the input to write all satisfying elements into the output. This requires exactly m array updates.

If we make the changes suggested by Example 65.4, then Algorithm 65.7 will perform decently well in practice when the predicate function is expensive. However, when the predicate is cheap, Algorithm 65.7 will not perform well in comparison to a fast sequential filter. This is for two reasons:

1. the parallel-for loop does not have proper granularity control, and
2. the number of memory updates is non-optimal.

Algorithm 65.8 (Blocked Filter). Assuming a cheap predicate function, we can make Algorithm 65.7 more efficient by again applying the blocking technique. This will control granularity and will help perform many fewer array updates overall.

For a block size B , which must be at least 1, the algorithm is straightforward:

1. Count, for each block, how many elements within the block satisfy the predicate. Do this in parallel across the blocks.
2. Scan to compute the prefix-sums of block-counts computed in the previous step. Call this sequence C . Note that the scan also returns the size of the output.
3. In parallel for each i , filter the i^{th} block sequentially, writing the results into the output starting at $C[i]$.

Exercise 65.7. Argue that Algorithm 65.8 performs $2n/B + m$ array updates and has proper granularity control. You may assume the scan performs an optimal number of updates and has proper granularity control.

Solution. To store the block counts, we require n/B updates. The scan then performs n/B updates. Finally, there are m updates to the output. For granularity control, as long as we choose B to be large enough, the cost of processing one block will amortize the cost of a single thread.

Chapter 66

Critical Sections and Mutual Exclusion

In this chapter, we present the concepts of critical sections and how we may use synchronization instructions to ensure mutual exclusion.

Definition 66.1 (Critical Sections and Mutual Exclusion). In a concurrent program, a *critical section* is a part that cannot be executed by more than one thread at the same time. In other words, a critical section must be executed in *mutual exclusion*. Critical sections typically contain code that alters data shared among parallel computations, and could lead to data races if executed concurrently.

Example 66.1 (RadCoin). A new startup RadCoin has a very simple and highly scalable approach to financial transactions that has piqued the interest and admiration of the Wall Street. Their technology rests on two functions *debit* and *credit* that respectively take a specified amount *delta* from an account by updating the balance *bal*. The following code snippet shows their magic debit and credit functions (N.B. all IP rights reserved) and uses them to debit and credit 10 from and to the same account.

```
let
  debit bal delta =
    bal ← bal - delta
  credit bal delta =
    bal ← bal + delta
in
  (debit mybal 10) || (credit mybal 10)
end
```

We expect the balance to remain unchanged after this operation but this is not guaranteed unless the functions *debit* and *credit* execute in mutual exclusion. Suppose that we start with 50 dollars in the account. Now, consider the following two scenarios.

- The functions *credit* and *debit* both read the balance of 50 and *debit* takes out the 10 and updates the balance, and then *credit* updates the balance to 60. Effectively, the outcome is that the update from the function *debit* has been lost.
- The functions *credit* updates the balance first, and then *debit* sets the balance to 40. In this scenario, the update from the function *credit* has been lost.

Note. Consider an operation of the form

$$bal \leftarrow bal \oplus delta,$$

where \oplus can be any arithmetic operation such as plus or minus. Even though this operation is written as one operation, a real computer cannot guarantee its atomicity, because it is typically executed as three instructions

$$\begin{aligned} oldbal &\leftarrow !bal \\ newbal &\leftarrow oldbal \oplus delta \\ bal &\leftarrow newbal. \end{aligned}$$

Thus if multiple instances of the operation is on flight in parallel, their effects may take effect in memory in arbitrary total order.

Data Races and Critical Sections. If a critical section is executed concurrently, then a data race could occur. When a data race occurs, the outcome of the program usually depends on the relative timing of events in the execution, and varies from one execution to another. Most race conditions therefore break correctness properties of programs but some are benign and don't harm correctness.

Heisenbug. It can be extremely difficult to find a data race, because parallel programs usually exhibit ample non-determinacy in their execution, due to scheduling. A data race may lead to an observably incorrect behavior only a tiny fraction of the time, making it extremely difficult to observe and reproduce it. This becomes especially a problem, because when executed in the “debugging mode”, programs usually “slow down” and sometimes the effect is significant enough to cause the bug to disappear. Such a bug is called an *Heisenbug*.

Example 66.2. Many examples of data races exist in the history of computing. A particularly destructive one was the “Northeast Blackout” of 2003, which caused 45 million people in the US and 10 million people in Canada to lose electricity for an extended period of time. The data race was in an “Energy Management System” developed by General Electric (GE). Here is a quote describing the data race by a GE manager.

“There was a couple of processes that were in contention for a common data structure, and through a software coding error in one of the application processes, they were both able to get write access to a data structure at the same time. And that corruption lead to the alarm event application getting into an infinite loop and spinning.”

The data race took software engineers weeks of poring through approximately one million lines of C and C++ code. To do this, the engineers slowed down the system and injected deliberate delays in the code while feeding alarm inputs to the program. About eight weeks after the blackout, the bug was unmasked as a particularly subtle data race, triggered on August 14th by a perfect storm of events and alarm conditions on the equipment being monitored. The bug had a window of opportunity measured in milliseconds.

Definition 66.2 (Mutual Exclusion Problem). The problem of designing algorithms or protocols for ensuring mutual exclusion is called the *mutual exclusion problem* or the *critical section problem*.

Definition 66.3 (Synchronization Instructions). There are many techniques for solving mutual exclusion problems. Nearly all of these techniques involve synchronizing between the threads by using *synchronization instructions*, which can be broadly divided into three categories.

1. *Spin locks* allow a thread to “busy wait” until the critical section is “clear” of other threads.
2. *Blocking locks* allow a thread to “block” and wait for another thread to exit the critical section. When the critical section is clear, then the blocked thread receives a signal, allowing it to proceed. The term *mutex*, short for “mutual exclusion” is sometimes used to refer to a blocking lock.
3. *Atomic read-modify-write instructions*, can read and modify the contents of a memory location atomically, allowing a thread to operate safely on shared data.

Remark. One example of a mutual exclusion algorithm that does not use synchronization operations is Dekker’s algorithm for mutual exclusion. The algorithm works only for two threads.

Atomic Read-Modify-Write Instructions. In contrast to conventional instructions, these instructions differ in the sense that they can be used to perform atomically a sequence of three instructions, i.e.,

- reading the contents of reference,
- computing some value based on it, and
- writing it back into the reference atomically.

As it turns out, these instructions suffice to implement more complex concurrent operations on shared data.

Definition 66.4 (Nonblocking Synchronization). Atomic read-modify-write operations typically require special hardware support and are implemented more or less directly in hardware. Because they don’t block the executing thread, atomic read-modify-write instructions are called *nonblocking*. Nonblocking operations can be used to implement more

complex concurrent *nonblocking data structures*, such as concurrent stacks and queues, that can guarantee system-wide progress.

Definition 66.5 (Compare and Swap). *Compare-and-swap* is an atomic read-modify-write instruction that performs a memory read followed by a memory write atomically on a machine word. The type signature for the `cas` instruction is

`cas : word ref → word * word → word.`

Given a reference r and expected value exp and a target value tgt , the instruction

`cas r (exp, tgt)`

performs the following atomically:

1. let old be the contents of r ,
2. compare old with exp ,
3. if they are equal, then write into r the value tgt ,
4. otherwise, leave r unchanged, and
5. return old

Definition 66.6 (Fetch and Add). *Fetch-and-add* is an atomic read-modify-write instruction that atomically updates the contents of a memory location and returns the contents (before the update). The type signature for the `faa` instruction is

`faa : word ref → word → word.`

Given a reference r and an “increment” $delta$, the instruction

`faa r delta`

performs the following update atomically

`let v = !r` (66.1)

`r ← !r ⊕ delta` (66.2)

`in v end.` (66.3)

Here \oplus is the addition operation on machine words.

Exercise 66.1. Implement a spin-lock using `fetch-and-add`.

Exercise 66.2. Implement `fetch-and-add` using `compare-and-swap`. How does the efficiency of your implementation compare to that of the fetch-and-add instruction.

Solution.

```

fetchAndAdd r delta =
  let
    loop r =
      let old = !r
      new = old + delta
      in
        if cas r (old, new) = old then
          old
        else
          loop r
      end
    in
    loop r
  end

```

Exercise 66.3. Implement a spin-lock using [compare-and-swap](#) .

Exercise 66.4. Describe how to fix the concurrency in RadCoin's [code](#) by using [compare-and-swap](#) . Is your solution as scalable as that of RadCoin's original implementation?

Exercise 66.5. Using the [compare-and-swap](#) instruction for synchronization, design a concurrent stack data structure that can be shared by multiple threads. Such a concurrent stack data structure can be used to implement a thread scheduler for executing the threads in a parallel language such as SPARC or MPL.

Appendix A

Computations as Graphs

Any algorithmic analysis must assume a *cost model* that defines the resource costs required by a computation. Broadly accepted cost models include machine-based cost models and more abstract machine-independent models that can typically be “instantiated” to work with many different machine models. In this chapter, we review several machine-based cost models, including the RAM model and the PRAM model, and discuss the limitations of the PRAM model. We then define a more abstract model based on the idea of representing computations as graphs. This model has several key advantages over machine-based models.

1 Machine-Based Cost Models

Definition A.1 (Machine-Based Cost Model). A *machine-based (cost) model* takes a machine model and defines the cost of each instruction that can be executed by the machine—often unit cost per instruction. When using a machine-based model for analyzing an algorithm, we express our algorithm directly to operate on the model and analyze the cost of the machine instructions used by the algorithm.

Remark. Machine-based models are suitable for deriving asymptotic bounds (i.e., using big-O, big-Theta and big-Omega) but not for predicting exact runtimes. The reason for this is that on a real machine not all instructions take the same time, and furthermore not all machines have the same instructions.

1.1 RAM Model

The classic machine-based model for analyzing sequential algorithms is the *Random Access Machine* or *RAM*. In this model, a machine consists of a single processor that can

access unbounded memory; the memory is indexed by the natural numbers. The processor interprets sequences of machine instructions (code) that are stored in the memory. Instructions include basic arithmetic and logical operations (e.g. $+$, $-$, $*$, and, or, not), reads from and writes to arbitrary memory locations, and conditional and unconditional jumps to other locations in the code. Each instruction takes unit time to execute, including those that access memory. The execution-time, or simply *time* of a computation is measured in terms of the number of instructions executed by the machine. Because the model is sequential (there is only one processor) time and work are the same.

Critique of the RAM Model. Most research and development of sequential algorithms has used the RAM model for analyzing time and space costs. One reason for the RAM model's success is that it is relatively easy to reason about the costs because algorithmic pseudo code can usually be translated to the model. Similarly, code in low-level sequential languages such as C can also be translated (compiled) to the RAM Model relatively easily. When using more abstract pseudo-code or higher level languages, the translation from the algorithm to machine instructions becomes more complex and we find ourselves making strong, possibly unrealistic assumptions about costs, even sometimes without being aware of the assumptions.

More broadly, the RAM model becomes difficult to justify in modern languages. For example, in object-oriented languages certain operations may require substantially more time than others. Features of modern programming languages such as automatic memory management can also be difficult to account for in analysis. Functional features such as higher-order functions are even more difficult to reason about in the RAM model because their behavior depends on other functions that are used as arguments. Such functional features, which are the mainstay of "advanced" languages such as the ML family and Haskell, are now being adopted by more mainstream languages such as Python, Scala, and even for more primitive (closer to the machine) languages such as C++. As you might recall, C++17 has added support for Lambda function. All in all, it requires significant expertise to understand how an algorithm implemented in modern languages today may be translated to the RAM model.

1.2 PRAM: Parallel Random Access Machine

The RAM model is sequential but can be extended to use multiple processors which share the same memory. The extended model is called the Parallel Random Access Machine.

PRAM Model. A *Parallel Random Access Machine*, or *PRAM*, consist of p sequential random access machines (RAMs) sharing the same memory. The number of processors, p , is a parameter of the machine, and each processor has a unique index in $\{0, \dots, p-1\}$, called the *processor id*. Processors in the PRAM operate under the control of a common clock and execute one instruction at each time step. The PRAM model is most usually used as a *synchronous* model, where all processors execute the same algorithm and operate on

the same data structures. Because they have distinct ids, however, different processors can do different computations.

Example A.1. We can specify a PRAM algorithm for adding one to each element of an integer array with p elements as shown below. In the algorithm, each processor updates certain elements of the array as determined by its processor id, id .

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

If the array is larger than p , then the algorithm would have to divide the array up and into parts, each of which is updated by one processor.

SIMD Model. Because in the PRAM all processors execute the same algorithm, this typically leads to computations where each processor executes the same instruction but possibly on different data. PRAM algorithms therefore typically fit into *single instruction multiple data*, or *SIMD*, programming model. Example A.1 shows an example SIMD program.

Critique of PRAM. There are at least several limitations to PRAM.

- PRAM algorithm are designed typically designed to operate on any number of processors, under the assumption that the number of processor remain fixed throughout the execution of the algorithm. This might work in certain settings such as supercomputing centers but otherwise it is an unrealistic assumption. In a typical system, the operating system or other users may claim any number of the processors at any time.
- Since PRAM is a synchronous model, and it requires the algorithm designer to map or schedule computation to a fixed number of processors, the PRAM model can be awkward to work with. For example, adding a value to each element of an array is easy if the array length is equal to the number of processors, but messier if not, which is typically the case. For computations with nested parallelism, such as divide-and-conquer algorithms, this scheduling chore becomes much more complicated.
- Unlike the RAM model, which is broadly implemented by many real computers, I have not seen a broadly accepted practical computer that implements the PRAM model.

For these reason, we will not discuss the PRAM model in further detail.

2 Computation Graphs

Computation graphs is an alternative approach to analyzing the cost of algorithms that do not make strong assumptions about the underlying machine. They therefore have the advantage of being more broadly applicable and more flexible. Nevertheless they are quite low level and may not always be suitable for high-level reasoning. But they will enable us to define an even higher, algorithm-level cost models based on just two cost metrics: work and span.

Definition A.2 (Computation Graph). A *computation graph* $G = (V, E)$ is a graph with the following properties.

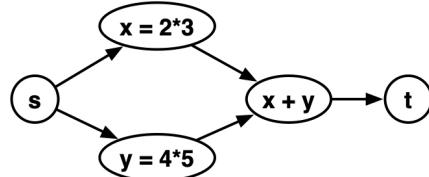
- Each vertex represent an operation to be performed.
- Each edge $(u, v) \in E$ represents the dependency (precedence) between the operations u and v .
- There is a unique *source vertex* $s \in V$ with no incoming edges such that each vertex is reachable from s
- There is a unique *sink vertex* $t \in V$ with no outgoing edges.

The exact definition of an “operation” depends on our specific assumptions and the underlying model that we assume, e.g., the RAM model, the PRAM model, or some other model that we may create, e.g., one that might reflect modern multicore computers that can perform various operations in unit time.

Example A.2. Consider the computation of the expression

$$2 * 3 + 4 * 5.$$

Assuming that the two terms can be computed in parallel, we may represent this computation by the following graph.



In the graph, x and y are variables holding the values of the terms $2 * 3$ and $4 * 5$ respectively. We may more precisely write the code for the program as follows.

```

let (x, y) = (2 * 3, 4 * 5) in
  x + y
  
```

Example A.3. Consider the expression

```
let x = 2 * 3
    y = 4 * 5
in x + y
```

where x and y are computed sequentially. We may represent this computation by the following graph.



Assumption. Throughout, we assume that an operation is a computation that can be performed in a single time step. This assumption is not crucial and can be relaxed but at the cost of increased complexity in technical details and the proofs.

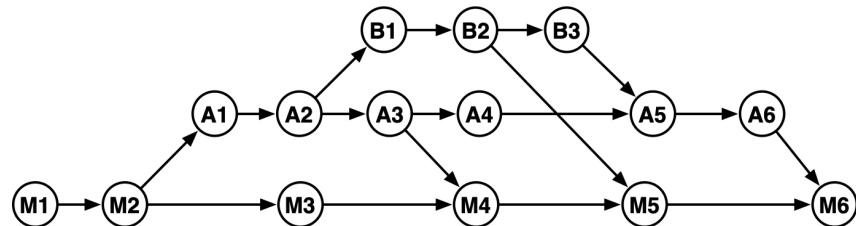
Exercise A.1. Draw the computation graph for the following computation, which maps a function over a sequence.

```
let
  fun map f a = tabulate (lambda i. (f a[i])) (0, |a|-1)
  a = (0, 1, 2, 3, 4, 5, 6, 7)
in
  map f a
end
```

Exercise A.2 (Summing a Tree). Draw the computation graph for the code given in this function for summing the keys in a tree .

Note (Dags and Partial Orders). A computation graph represents the partial order of the dependencies between the operations in the computation. Perhaps the simplest computation is a sequential one, which can be represented as a chain of (totally ordered) vertices.

Example A.4 (A Computation Dag). A hypothetical computation dag is shown below.



Work and Span. To analyze the resource needs of a computation such as total energy consumption, time, and space, we will define certain resource metrics. In this course, we will primarily be concerned about time and use two metrics—work and span—to analyze execution time of computations. We define the work and span of a computation in terms of the work span of its vertices, which is positive a natural number.

Definition A.3 (Work and Span). Let $G = (V, E)$ be a directed acyclic graph with source s and sink t . The work of the graph G , written $\mathbb{W}(G)$ is the total number of vertices in the graph

$$\mathbb{W}(G) = |V|$$

The span of the graph G , written $\mathbb{S}(G)$, is length of the longest source-to-sink path in G .

$$\mathbb{S}(G) = \max_{p \in s \xrightarrow{E} t} |p|$$

Example A.5. The [example dag shown above](#) has work 15 vertices. The work of the graph is therefore 15. The longest path in the dag is the path $(M1, M2, A1, A2, A3, A4, A5, A6, M6)$ and thus the span is 9.

Appendix B

Scheduling: Lower and Upper Bounds

1 Scheduling

The execution of a computation dag G involves executing the vertices in the dag in some partial order that is consistent with the partial order specified by the dag. That is, if there is a path from u to v in G , then u must be executed before v . We call a partial order as “execution schedule” or simply as a “schedule.” More precisely, we define a scheduler as a mapping from time steps to vertices of the dag and the “processes” that execute them.

Definition B.1 (Execution Schedule). Given a computation dag $G = (V, E)$, an *execution schedule* or a *schedule* for G is a function from processes and (time) *steps* to operations of the dag such that

- each vertex in V is executed exactly once, and
- for each edge (u, v) in the graph, the vertex u is executed before v .

We define the *length* of a schedule as the number of steps in the schedule.

Example B.1. The schedule below shows a 3-process schedule for [this dag](#).

Time Step	Process 1	Process 2	Process 3
1	M1		
2	M2		
3	M3	A1	
4		A2	
5	B1	A3	
6	B2	A4	
7	B2		M4
8		A5	M5
9	A6		
10		M6	

Exercise B.1. Prove that the definition above ensures that if there is a path from a vertex u to v in the graph, then v cannot start executing before u completes.

Solution. The second condition ensures that a schedule observes the dependencies in the dag.

Definition B.2 (Ready Vertex). For any step in the execution, we call a vertex *ready* if all the ancestors of the vertex in the dag are executed prior to that step.

Lemma B.1 (Red-Blue Dags). Consider a computation dag G with source s and sink t and consider any execution schedule for G . At any time step during the execution, color the vertices that have been executed before that step as blue and the others as red.

- The blue vertices induce a blue sub-dag of G that is connected and that has the source s .
- The red vertices induce a red sub-dag of G that is connected and that has sink t .
- All the vertices of G are in the blue or the red sub-dag. In other words, the blue and red vertices partition the dag into two sub-dags.

Proof. Proceed by induction on the schedule. □

Question. How much time do we need to execute a computation graph?

The answer to this question turns out to be nontrivial. Here, we define the corresponding problem more precisely and give several lower and upper bounds.

Definition B.3 (Offline Scheduling Problem). The *offline scheduling problem* requires finding the shortest schedule for a computation graph.

The decision problem corresponding the Offline Scheduling Problem turns out to be NP-complete. Fortunately, approximating the optimal schedule within a reasonable constant factor is “not hard”. We present two approximation algorithms, level-by-level scheduler and greedy scheduler, that can compute a 2-approximation.

Theorem B.2 (Lower Bounds). Let $G = (V, E)$ be a computation dag. Let W be the work of G , i.e., $W = \mathbb{W}(G)$, and let S be the span of G , i.e., $S = \mathbb{S}(G)$.

The following lower bounds hold for any P -process schedule of G .

- The schedule has length at least $\frac{W}{P}$.
- The schedule has length at least S .

Proof. The first lower bound follows by the simple observation that at each step, a schedule can execute at most P vertices. Because all vertices must be executed, a schedule has length at least $\frac{W}{P}$.

The second lower bound follows by the observation that the schedule cannot execute a vertex before its ancestors. Thus a schedule is at least as long as any path in the dag, and thus the span S . \square

2 Upper Bounds for Fixed Number of Processes

Definition B.4 (Level-by-Level Schedule). A *level-by-level schedule* is a schedule that executes the vertices in a given dag in level order, where the *level* of a vertex is the longest distance from the root of the dag to the vertex. More specifically, we start executing the vertices in level 0, and then level 1, and so on.

Exercise B.2. What is the difference between a level-by-level schedule/traversal of a computation graph and its breadth-first-search traversal?

Theorem B.3 (Brent). For a dag $G = (V, E)$ with work W and span S , the length of a level-by-level P -process schedule is $W/P + S$.

Proof. let W_i denote the total work at level i in this graph. A level-by-level schedule can execute the work at level i in $\lceil \frac{W_i}{P} \rceil$ steps. Thus the length of a P -process schedule is

$$\begin{aligned} T_P &\leq \sum_{i=1}^S \lceil \frac{W_i}{P} \rceil \\ &\leq \sum_{i=1}^S \lfloor \frac{W_i}{P} \rfloor + 1 \\ &\leq \lfloor \frac{W}{P} \rfloor + S \end{aligned}$$

\square

Note. This theorem is due to Australian mathematician and computer scientist Richard Peirce Brent. In his original paper, Brent considers parallel evaluation of arithmetic expressions, but his results were quickly generalized to arbitrary computations

Definition B.5 (Greedy Scheduling). Brent's theorem has later been generalized to all greedy schedules. A *greedy schedule* is a schedule that never leaves a process idle if there are ready vertices. In other words, greedy schedules keep processes busy by greedily assigning ready vertices.

Theorem B.4 (Greedy Schedule). Consider a dag $G = (V, E)$ with work W and span S . Any greedy P -process schedule has length at most $\frac{W}{P} + S \cdot \frac{P-1}{P}$.

Proof. Consider any greedy schedule with length T .

For each step $1 \leq i \leq T$, and for each process, collect a token. The token goes to the *work bin* if the process executes a vertex in that step, otherwise the process is idle and the token goes to an *idle bin*.

Because each token in the work bin corresponds to an executed vertex, there are exactly W tokens in that bin.

We will now bound the tokens in the idle bin by $S \cdot (P - 1)$. Observe that at any step in the execution schedule, there is a ready vertex to be executed (because otherwise the execution is complete). This means that at each step, at most $P - 1$ processes can contribute to the idle bin. Furthermore at each step where there is at least one idle process, we know that the number of ready vertices is less than the number of available processes. Note now that at that step, all the ready vertices have no incoming edges in the red sub-dag consisting of the vertices that are not yet executed, and all the vertices that have no incoming edges in the red sub-dag are ready. Thus executing all the ready vertices at the step reduces the length of all the paths that originate at these vertices and end at the sink vertex by one. This means that the span of the red sub-dag is reduced by one because all paths with length equal to span must originate in a ready vertex.

Because the red-subdag is initially equal to the dag G , its span is S . Thus there are at most S steps at which a process is idle. As a result the total number of tokens in the idle bin is at most $S \cdot (P - 1)$.

Because we collect P tokens in each step, the bound follows. \square

Exercise B.3. Show that the bounds for Brent's level-by-level scheduler and for any greedy scheduler is within a factor 2 of optimal.

Solution. The bound is $T_P \leq \frac{W}{P} + S$. Let's write this as

$$T_{OPT} \leq T_P \leq \frac{W}{P} + S.$$

Now we know $T_{OPT} \geq \frac{W}{P}$ and $T_{OPT} \geq S$. Thus, we have

$$T_{OPT} \leq T_P \leq \frac{W}{P} + S \leq T_{OPT} + T_{OPT} = 2T_{OPT}.$$

Thus, we have

$$T_{OPT} \leq T_P \leq 2T_{OPT}.$$

3 Upper Bounds for Dynamic Environments

The PRAM model has [several limitations](#), one of which is the typical assumption that the number of processes remains the same throughout the execution of the algorithm. This is because PRAM algorithms specify the schedule and therefore make it difficult to accommodate the setting where the number of processes may vary dynamically over time. In this section, we present a simple and elegant result due to Arora, Blumofe, and Plaxton that establishes a powerful result that shows that the greedy schedule delivers a nice upper bound even when the number of processes vary over time in a completely unpredictable manner.

Definition B.6 (Average Number of Processes). Consider a schedule with length T , where at each step, each process decides whether it wants to participate in that step or not. For such a schedule, we define the *average number of processes* or simply the *process average* as the time-averaged number of processes. More precisely, let p_i denote the number of processes participating at time step i , the process average, written \bar{P} , is

$$\bar{P} = \frac{1}{T} \sum_{i=0}^{T-1} p_i.$$

Theorem B.5 (Dynamic Greedy Scheduling). Consider a dag $G = (V, E)$ with work W and span S . Consider any greedy schedule that at each step maps available ready nodes to available processes (processes participating at that step). Such a schedule has length at most $\frac{W}{\bar{P}} + S \cdot \frac{P-1}{\bar{P}}$.

Proof. The proof is almost exactly as in the [proof for the static setting](#) where the number of processes does not change over time. Following the same arguments, we know that the total number of tokens in each bin is

$$W + (P - 1)S.$$

Now let T be the length of the schedule we know that

$$\sum_{i=0}^{T-1} p_i = W + (P - 1)S.$$

Now by [definition](#), the left hand side is nothing but $T\bar{P}$, thus we have:

$$T\bar{P} = W + (P - 1)S.$$

It follows that

$$T = \frac{W}{\bar{P}} + \frac{P-1}{\bar{P}}S.$$

□

4 Concluding Remarks

Work and Span as An Abstraction. The theorems on scheduling are powerful. They basically say that we need not care about scheduling, because it is not that hard to figure out what a good schedule will be. We can thus forget about scheduling the operations completely and express our algorithms to minimize the work and the span of the computation. Once we have an algorithm with minimal work and span, by application of the scheduling theorems, we know how the algorithms will behave.

Algorithmic Cost Models. But how do we reason about the work and span of our algorithms? The discussion in this section would suggest that we would need to create the computation graph and determine its work and span. This could work but there is a better way: define a higher-level cost model that allows us to reason about the work and span of the algorithm at the same level of abstraction as the algorithm. We can think of such cost models as “algorithmic cost models”, because they are at the same abstraction level of the algorithm (rather than the lower-level graph or machine level).

Question. Could the inelegance of PRAM and its lack of mathematical depth and beauty have predicted its demise?

Remark (From Offline to Online Scheduling). While the offline-scheduling bounds presented in this chapter are powerful, they should be taken as “existence proofs”. They prove that there exists a schedule that approximates the optimal schedule within a factor of two. But they do not show us how to compute the schedule efficiently. Specifically the question is how do we construct a greedy or a level-by-level schedule on a realistic parallel computer?

The answer to this question turns out to be highly nontrivial and continues to be a research problem even today. But for our purposes in this course, this problem is solved. We will see an “online algorithm” called work stealing that computes a greedy schedule in the near future.

Appendix C

Work Stealing Algorithm

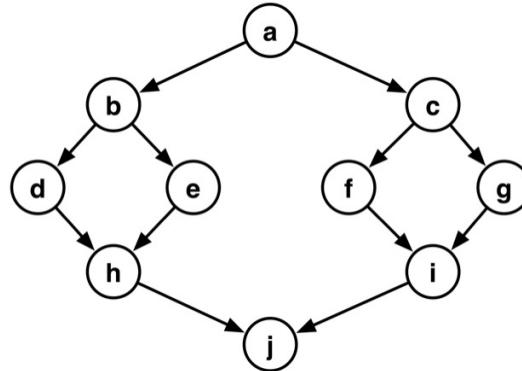
In Chapter B, we define an execution schedule or schedule for short, and defined the [offline scheduling problem](#) as the problem of finding the shortest schedule for a [computation graph](#). We then present several lower bounds and an upper bound on the length of the environment by considering both dedicated (static) and dynamic (multiprogrammed) environments. For the upper bounds, we showed that the [greedy scheduling](#) gives us a P_A -process schedule of length $\frac{W}{P_A} + S \cdot \frac{P_A - 1}{P_A}$, for a computation dag with work W and span S , where P_A is the average number of processors throughout the execution, with $P_A = P$ in the dedicated setting.

Definition C.1 (Online Scheduling Problem). The [online scheduling problem](#) requires finding the shortest schedule for a computation graph while taking account the cost of scheduling itself.

To see why computing the online scheduling is challenging, consider a straightforward algorithm for greedy scheduling.

Algorithm C.2 (Shared-Queue Scheduling). The shared-queue algorithm keeps a single queue of ready vertices for all processors. When a processor needs a vertex to execute, it takes the vertex at the head of the queue. When a processor executes a vertex that then makes ready new, or enables, vertices, it pushes them to tail of the queue. The algorithm starts the execution by assigning the root of the computation dag to one processor.

Exercise C.1. Show several different 2-processor executions of the dag shown below. These executions have a “depth first” flavor. Can you formulate precisely what this property is?



Contention. The problem with the shared-queue algorithm is contention: because the queue is shared among all processors and only one processor can access the queue at any single time step, the queue becomes a point of contention. In a modern multicore machine today, this algorithm does not scale beyond a handful or so of processors, depending on exactly how the concurrent accesses are resolved.

To reduce contention, we take a more distributed approach and employ multiple queues for ready vertices. Perhaps the most common approach for doing so is to simply give each processor its own queue and “balance load” between queues as needed in some fashion. The work stealing algorithm takes this approach.

Assumption. To streamline the presentation and the analysis, we make several assumptions.

- The root vertex has a single child. This assumption causes no loss of generality, because we can augment any graph with a root vertex that has the original root as its only child.
- Each vertex has out-degree at most two.

1 Doubly Ended Queues

The work-stealing algorithm relies on abstract data type called doubly-ended queue, which behaves like a queue but allows “popping” on both ends, instead of just one.

Definition C.3 (Deque). A *doubly ended queue* or a *deque* (read “deck”) for short is an abstract data type (ADT) that support the following three operations.

- *pushBottom* pushes a vertex at the bottom end of the deque.
- *popBottom* returns the vertex at the bottom end of the deque if any or returns **None** otherwise.

- *popTop* returns the vertex at the top end of the deque, if any, or returns **None** if the deque is empty.

Definition C.4 (Concurrent Deque). A *concurrent deque* is an abstract data type (ADT) that supports the same operations as a deque but allows *popTop* operations to be concurrent. That is multiple *popTop* operations may be called at the same time on the same deque and a concurrent deque responds to each of these operations correctly as if these operations were called sequentially, in some (nondeterministic) order. In other words, *popTop* operations execute atomically in some sequentially consistent order.

2 Work-Stealing Algorithm

Algorithm C.5 (Work Stealing). In work stealing, each worker maintains a *(concurrent) deque* of vertices. Execution starts with the root vertex in the deque of worker zero. It ends when the final vertex is executed. Each worker tries to work on its local deque as much as possible and keeps as worker-local data an *assigned vertex* for execution.

The algorithm then proceeds in a number of *rounds*. In each round, a worker executes its *assigned vertex* if any, pushes the newly enabled vertices to its deque, and obtains a new assigned vertex from its deque by popping the vertex at the bottom of its deque. If the round starts with no assigned vertex then the worker becomes a *thief* and performs a steal attempt by picking a *victim* worker at random and attempting to steals a vertex from it. To steal a vertex, the thief pops a vertex off the top of the victim's deque. A steal attempt starts and completes in the same round. Such a *steal attempt* can fail if

- the victim's deque is empty, or
- contention between workers occurs and the vertex targeted by the thief is either 1) executed, or 2) is stolen by another thief.

```

(* Assign root to worker zero *)
if (wid = 0)
    assignedVertex ← rootVertex
else
    assignedVertex ← None

while not done do
    (* Execute assigned vertex *)
    if assignedVertex <> None
        (nChildren, child1, child2) ← execute assignedVertex
        if (nChildren = 1)
            pushBottom deques[wid] child1
        else if (nChildren = 2)
            pushBottom deques[wid] child1
            pushBottom deques[wid] child2
        assignedVertex ← popBottom deques[wid]
    else
        (* Make steal attempt *)
        victim ← select a random worker
        assignedVertex ← popTop deques[wid]

```

Note (Implementation). When a vertex enables a single vertex, the algorithm pushes it to the deque at this step and pops it off at the next for execution. This is of course unnecessary and should be avoided in a real implementation.

When a vertex enables two vertices they are both pushed onto the bottom of the deque in an order that is unspecified. A realistic implementation of the algorithm typically operates on “threads” (sequentially dependent sequence of vertices) rather than vertices and therefore we would ideally like a processor to execute the next vertex in the thread. To ensure this, we would therefore push the vertex that comes next in the thread last.

Definition C.6 (Work Sequence). Consider the execution of the work stealing algorithm and let q be any worker. We define the *work sequence* of q as the sequence of vertices defined by the assigned vertex of q followed by the vertices in its deque ordered from bottom to top. If a vertex is in the work sequence of a worker, then we say that it *belongs* to that worker.

Example C.1 (Work Sequence). Consider the deque for a worker shown below along with the assigned vertex. The work sequence for this worker is $\langle v_0, v_1, v_2, v_3 \rangle$.

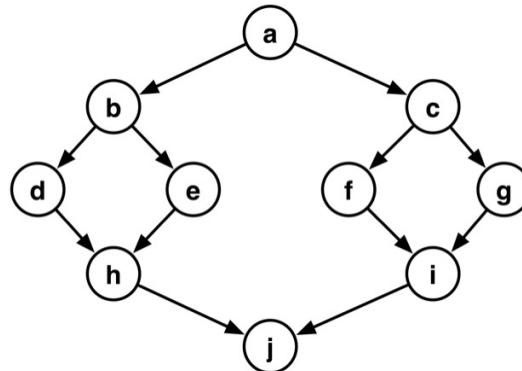
Deque



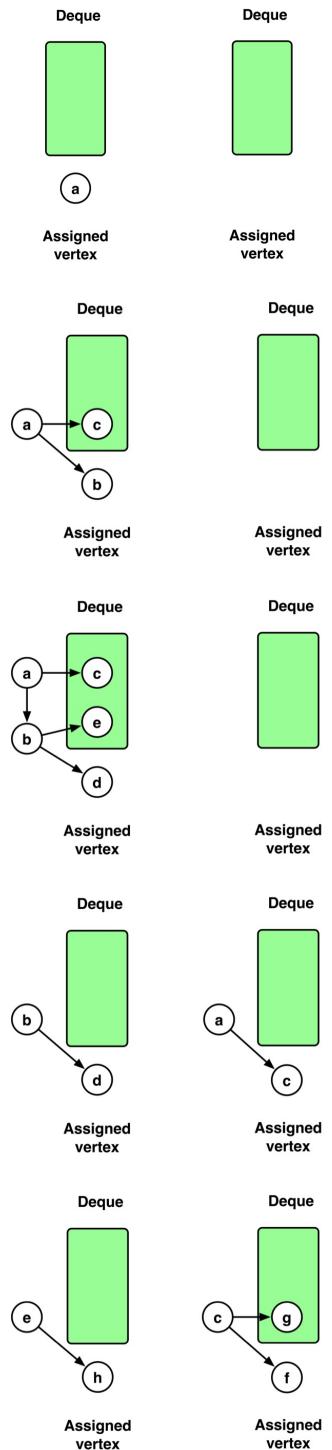
Now, if the worker completes executing v_0 , which enables no new children, then the work sequence consist of the vertices in the deque, i.e., $\langle v_1, v_2, v_3 \rangle$. If the worker, later removes v_1 from its deque and starts working on it, i.e., v_1 becomes the assigned vertex, then the work sequence remains the same, i.e., $\langle v_1, v_2, v_3 \rangle$.

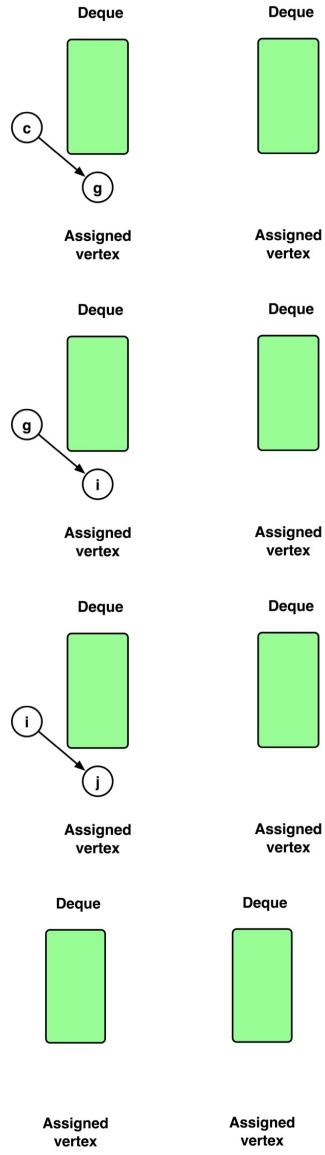
Definition C.7 (Enabling Tree). At any step in an execution, we say that a vertex is *ready* if all the ancestors of the vertex in the dag are executed prior to that step. If execution of a vertex u makes ready another vertex v , then we say that u *enables* v , and call the edge (u, v) an *enabling edge*. We call u the *designated parent* of v . Every vertex except for the root has a designated parent. Therefore the subgraph of the dag consisting of the enabling edges form a rooted tree, called the *enabling tree*. Note each execution can have a different enabling tree.

Example C.2 (A Work Stealing Run). Consider the following computation dag.

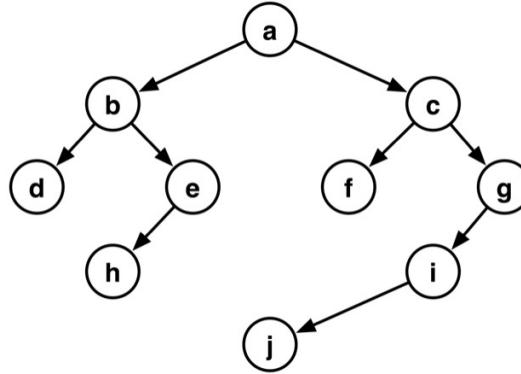


The following drawings illustrate a 2-worker execution of this dag using work stealing. Time flows left to right and top to bottom.





Example C.3 (Enabling Tree). The enabling tree for the execution above is shown below.

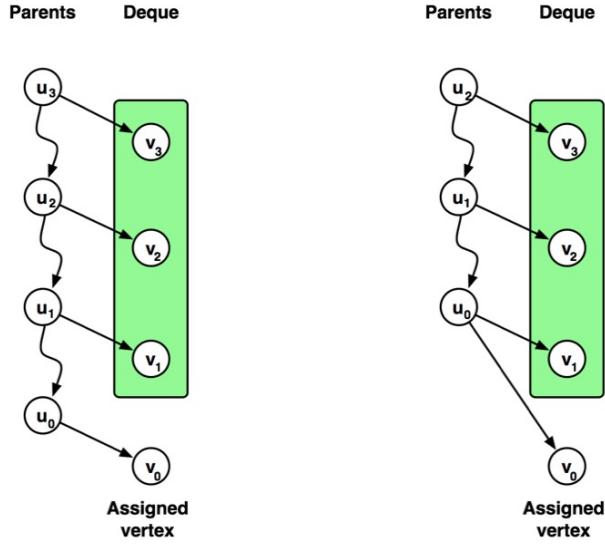


3 Structural Lemma

The work-stealing algorithm follows a specific regime for executing the vertices in its deque. Each worker uses its own deque as a stack, which yields an execution, where the order of vertices executed is locally consistent with sequential execution. The structural lemma makes precise this observation.

Lemma C.1 (Structural Lemma). Consider any time in an execution of the work-stealing algorithm after the execution of the root vertex. Let v_0, v_1, \dots, v_k denote the work sequence of any worker. Let u_0, u_1, \dots, u_k be the sequence consisting of the designated parents of the vertices in the work sequence in the same order. Then u_i is an ancestor of u_{i-1} in the enabling tree. Moreover, we may have $u_0 = u_1$ but for any $2 \leq i \leq k$, $u_{i-1} \neq u_i$, i.e., the ancestor relationship is proper.

Example C.4 (Structural Lemma). The figures below illustrate the two cases of the structural lemma.



The proof of the structural lemma is by induction on the number of rounds of the execution.

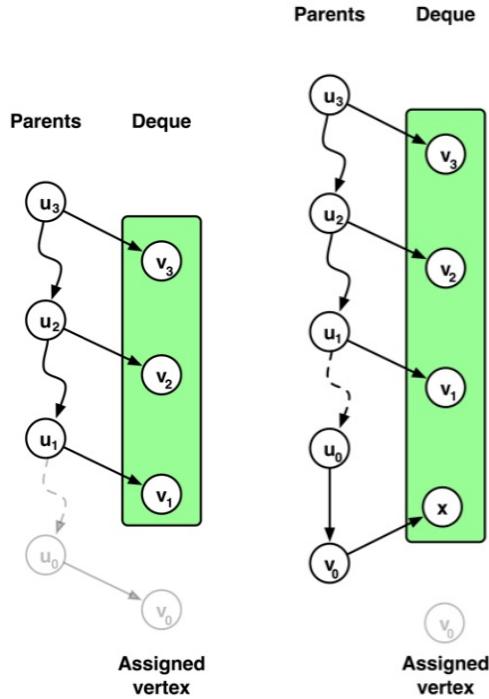
Consider the first round. At the initialization and before the beginning of the first round, all deques are empty, root vertex is assigned to worker zero but has not been executed. The root vertex is then executed. By assumption, the root vertex has a single child v , which becomes enabled and pushed onto the deque and popped again becoming the assigned vertex at the beginning of the second round. At any point in time after the execution of the root, worker zero's work sequence consist of v . The designated parent of v is the root vertex and the lemma holds trivially.

For the inductive case, assume that the lemma holds up to beginning of some later round. We will show that it holds at any point during the round and also after the completion of the round.

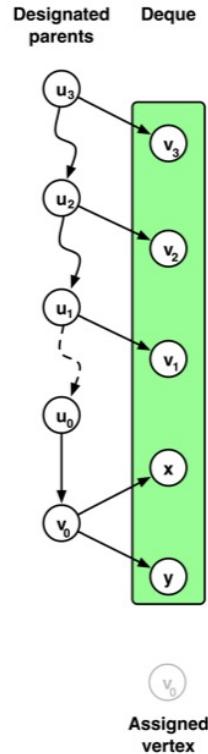
Consider any worker and its deque. We have two cases to consider.

Case 1. In this case, we have an assigned vertex, v_0 , which is executed. By the definition of work sequences, we know that v_1, \dots, v_k are the vertices in the deque. Let u_1, \dots, u_k be their designated parents. By induction, we know that u_i is an ancestor of u_{i-1} in the enabling tree and the ancestor relationship is proper except for $i = 1$, where it is possible that $u_0 = u_1$. Immediately after the execution of the assigned node, the work sequence of the worker consists of all the vertices in the deque and the lemma follows.

After the execution of the assigned vertex v_0 , we have three possibilities. First possibility is that the execution of v_0 enables no children. Because the deque remains the same, the lemma holds trivially. The second possibility, is that the execution of v_0 enables one child x , which is pushed onto the bottom of the deque. In this case, v_0 becomes the parent of x . The lemma holds. The figures below illustrate these two cases.



The third possibility is that the execution of v_0 enables two children x, y , which are pushed to the bottom of the deque in an arbitrary order. In this case, v_0 becomes the parent of x . We need to show that $v_0 \neq u_1$. This holds because $v_0 \neq u_0$ and $v_0 \neq u_1$ (note that it is possible for u_0 and u_1 to be the same). Thus the lemma holds.



After the execution of the assigned vertex completes and the children are pushed, the worker pops the vertex at the bottom of the deque. There are two cases to consider.

- If the deque is empty, then the worker finds no vertex in its deque and there is no assigned vertex at the end of the round, thus the work sequence is empty and the lemma holds trivially.
- If the deque is not empty, then the vertex at the bottom of the deque becomes the new assigned vertex. The lemma holds trivially because making the bottom vertex the assigned vertex has no impact on the work sequence of the worker and thus the correctness of the lemma.

Case 2. In this case, a successful steal takes place and removes the top vertex in the deque. In this case, the victim worker loses its top vertex, which becomes the assigned vertex of the thief. The work sequence of the victim loses its rightmost element. It is easy to see that the lemma continues to hold for the victim. When the stolen vertex is assigned, the work sequence of the thief consist of just the stolen vertex and the lemma holds for the thief.

Appendix D

Analysis of Work Stealing

The strategy behind analysis is to assign a potential to each vertex and show that the potential decreases geometrically as the execution proceeds. To streamline the analysis, we make several assumptions about the concurrent deques.

Assumption (Deque Operations). We assume that each instruction and each deque operations executes in a single step. As a result, each iteration of the loop, a round, completes in constant steps.

We assume that all deque operations take place atomically. We can think of them as starting by taking a lock for the deque, performing the desired operation, and releasing the lock. This assumption is somewhat unrealistic, because it is not known whether *popTop* can be implemented in constant time. But a relaxed version of *popTop*, which allows *popTop* to return *None* if another concurrent operation removes the top vertex in the deque, accepts a constant-time implementation. This relaxed version suffices for our purposes.

1 Balls and Bins Game

One crucial fact behind the analysis is a probabilistic lemma, called the Balls and Bins lemma. This lemma proves something relatively intuitive: if you throw as many ball as there are bins, chances are good that you will have a ball in at least a constant fraction of the bins, because chances of all balls landing in a small number of bins is low.

Lemma D.1 (Balls and Bins). Suppose that P balls are thrown uniformly and randomly into P bins, where bin $1 \leq i \leq P$ has weight $W_i \geq 0$, and $W = \sum_{i=1}^P W_i$. For each bin, define the random variable

$$X_i = \begin{cases} W_i & \text{if a ball lands in bin } i \\ 0 & \text{otherwise} \end{cases}$$

If $X = \sum_{i=1}^P X_i$, then for any $\beta, 0 < \beta < 1$, we have $P[X \geq \beta W] > 1 - \frac{1}{(1-\beta)e}$.

Proof. The proof of the lemma is a relatively straightforward application of Markov's inequality. Consider the random variable $W_i - X_i$. This random variable takes on the value 0 if a ball lands in bin i and W_i otherwise. Thus, we have

$$\begin{aligned} E[W_i - X_i] &= W_i \cdot (1 - 1/P)^P, \text{ and} \\ E[W - X] &= W \cdot (1 - 1/P)^P. \end{aligned}$$

We know that $\lim_{P \rightarrow \infty} (1 - 1/P)^P = 1/e$ and furthermore that the derivative of this function is non-negative (the function is non-decreasing). Thus we know that $(1 - 1/P)^P \leq 1/e$.

It follows that $E[W - X] \leq W/e$.

Since the random variable $W - X$ is a non-negative random variable, we can apply Markov's inequality:

$$P[W - X > (1 - \beta)W] \leq \frac{E[W - X]}{(1 - \beta)W}.$$

It follows that

$$P[X < \beta W] < \frac{1}{(1 - \beta)e},$$

and thus

$$P[X \geq \beta W] > 1 - \frac{1}{(1 - \beta)e}.$$

□

Example D.1 (An application of the lemma). Let's calculate the probability for $\beta = 1/2$. By the lemma, we know that if P balls are thrown into P bins, then the probability that the total weight of the bins that have a ball in them is at least half the total weight is $P[X \geq \frac{W}{2}] 1 - \frac{1}{0.5e}$. Since $e > 2.71$, this quantity can be calculated as approximately 0.26. We thus conclude that we using the Ball and Bins lemma, we can "collect" at least half of the weight with probability at least 0.25

2 Bound in terms of Work and Steal Attempts

Lemma D.2 (Work and Steal Bound). Consider any multithreaded computation with work W . The P -worker execution time is $O(W/P + A/P)$ steps where A is the number of steal attempts.

Proof. Consider the execution in terms of rounds

- If a worker executes a vertex in the round, then it places a token into the work bucket.
- If the worker makes a steal attempt in the round, then it places a token into the steal-attempt bucket.

There are exactly A tokens in the steal-attempt bucket and exactly W tokens in the work bucket. Thus the total number of tokens is at most $W + A$. Since in each round a worker either executes a vertex or attempts a steal, and since each round is a constant number of steps, the P -worker execution time is $T_P = O(\frac{W+A}{P})$. \square

Note. The proof assumes that each instructions including deque operations takes a (fixed) constant number of steps, because it assumes that each round contributes to the work or to the steal bucket. If this assumption is not valid, then we might need to change the notion of rounds so that they are large enough for steals to complete.

3 Bounding the Number of Steal Attempts

Our analysis will use a potential-function based method. We shall divide the computation into phases each of which decreases the potential by a constant factor. We start with a few definitions.

Definition D.1 (Weights). If $d(u)$ is the depth of a vertex u in the enabling tree, then we define the weight of u , written $w(u)$ as follows $w(u) = S - d(u)$. The root has weight S . Intuitively, the weight is equal to the distance of a vertex from the completion.

As a corollary to the structural lemma, we observe that the weights of the vertices decrease from top to bottom.

Corollary D.3. Consider the work sequence of a worker $v_0, v_1, v_2 \dots v_k$. We have $w(v_0) \leq w(v_1) < w(v_2) \dots w(k-1) < w(v_k)$.

Definition D.2 (Ready Vertices at a Round). For a round i , we use R_i to denote the set of *ready vertices* in the beginning of that round. A vertex is R_i is either assigned to a worker or is in a deque. Let $R_i(q)$ denote the set of *ready vertices belonging to a worker* q at the beginning of round i ; these are exactly the vertices in the work sequence of that worker.

Definition D.3 (Potential Function). For each vertex $v \in R_i$, we define its *potential* as

- $\phi_i(v) = 3^{2w(v)-1}$, if v is assigned, or
- $\phi_i(v) = 3^{2w(v)}$, otherwise.

Note that potential of a vertex is always a natural number.

- The *potential of worker* q is $\Phi_i(q) = \sum_{v \in R_i(q)} \phi_i(v)$.
- We write H_i for the set of workers whose dequeues are empty at the beginning of round i . We write $\Phi_i(H_i)$ for *the total potential of the workers* H_i , $\Phi_i(H_i) = \sum_{q \in H_i} \Phi_i(q)$.
- We write D_i for the set of other workers. We write $\Phi_i(D_i)$ for *the total potential of the workers* D_i , $\Phi_i(D_i) = \sum_{q \in D_i} \Phi_i(q)$.
- Define the *total potential at round* i , written Φ_i as $\Phi_i = \sum_{v \in R_i} \phi_i(v)$. We can write the total potential in round i as follows $\Phi_i = \Phi_i(H_i) + \Phi_i(D_i)$.

Definition D.4 (Beginning and termination potential). At the beginning of the computation, the only ready vertex is the root, which has a weight of S , because it is also the root of the enabling tree. Therefore the potential in the beginning of the computation is 3^{2S-1} . At the end of the computation, there are no ready vertices and thus the potential is zero.

Lemma D.4 (Top-Heavy Deques). Consider any round i and any worker $q \in D_i$. The topmost vertex in q 's deque contributes at least $3/4$ of the potential of worker q .

Proof. This lemma follows directly from the structural lemma. The case in which the topmost vertex v contributes the least of the worker is when the assigned vertex and v have the same parent. In this case, both vertices have the same depth and thus we have

$$\Phi_i(q) = \phi_i(v) + \phi_i(x) = 3^{2w(v)} + 3^{2w(x)-1} = 3^{2w(v)} + 3^{2w(v)-1} = (4/3)\phi_i(v).$$

□

Lemma D.5 (Vertex Assignment). Consider any round i and let v be a vertex that is ready but not assigned at the beginning of that round. Suppose that the scheduler assigns v to a worker in that round. In this case, the potential decreases by $2/3 \cdot \phi_i(v)$.

Proof. This is a simple consequence of the definition of the potential function:

$$\phi_i(v) - \phi_{i+1}(v) = 3^{2w(v)} - 3^{2w(v)-1} = 2/3\phi_i(v).$$

□

Lemma D.6 (Decreasing Total Potential). Total potential does not increase from one round to the next, i.e., $\Phi_{i+1} \leq \Phi_i$.

Proof. Let's now consider how the potential changes during each round. There are two cases to consider based on scheduler actions.

Case 1 Suppose that the scheduler assign a vertex v to a worker. By the Vertex Assignment Lemma, we know that the potential decreases by $2/3 \cdot \phi_i(v)$. Since $\phi_i(v)$ is positive, the potential decreases.

Note that this calculation holds regardless of where in the deque the vertex v is. Specifically, it could have been the bottom or the top vertex.

Case 2 Suppose that the scheduler executes a vertex v and pushes its children onto the deque. There are two sub-cases to consider.

Case 2.1 Suppose that the scheduler pushes onto the deque the only child x of a vertex v . Assuming that the child stays in the deque until the beginning of the next round, the potential decreases by

$$\phi_i(v) - \phi_{i+1}(x) = 3^{2w(v)-1} - 3^{2w(v)-2} = 3^{2w(v)-1}(1 - 1/3) = 2/3 \cdot \phi_i(v).$$

Case 2.2 Suppose that the scheduler pushes onto the deque two children x, y of a vertex v . Assuming that the children remain in the deque until the beginning of the next round, then potential decreases by

$$\begin{aligned} \phi_i(v) - \phi_{i+1}(x) - \phi_{i+1}(y) &= 3^{2w(v)} - 3^{2w(v)-2} - 3^{2w(v)-2} \\ &= 3^{2w(v)-1}(1 - 1/3 - 1/3) \\ &= 1/3 \cdot \phi_i(v). \end{aligned}$$

Since $\phi_i(v)$ is positive, the potential decreases in both cases. Note that, it is safe to assume that the children remain in the deque until the next round, because assignment of a vertex decreases the potential further.

In each round each worker performs none, one, or both of these actions. Thus the potential never increases. \square

We have thus ustablished that the potential decreases but this by itself does not suffice. We also need to show that it decreases by some significant amount. This is our next step in the analysis. We show that after P steal attempts the total potential decreases with constant probability.

Lemma D.7 (P Steal Attempts). Consider any round i and any later round j such that at least P steal attempts occur at rounds from i (inclusive) to j (exclusive). Then, we have

$$Pr[\Phi_i - \Phi_j \geq \frac{1}{4} \Phi_i(D_i)] > \frac{1}{4}.$$

Proof. First we use the Top-Heavy Deques Lemma to establish that if a steal attempt targets a worker with a nonempty deque as its victim, then the potential decreases by at least of a half of the potential of the victim.

Consider any worker q in D_i and let v denote the vertex at the top of its deque at round i . By Top-Heavy Deques Lemma, we have $\phi_i(v) \geq \frac{3}{4}\Phi_i(q)$.

Consider any steal attempt that occurs at round $k \geq i$.

- Suppose that this steal attempt is successful with popTop returning a vertex. The two subcases both follow by the Vertex Assignment Lemma.
 - If the returned vertex is v , then after round k , vertex v has been assigned and possibly executed. Thus, the potential has decreased by at least $\frac{2}{3}\phi_i(u)$.
 - If the returned vertex is not v , then v is already assigned and possibly executed. Thus, the potential has decreased by at least $\frac{2}{3}\phi_i(v)$.
- Suppose that the steal attempt is not successful, and popTop returns None . In this case, we know that q 's deque was empty during popTop , or some other popTop or 'popBottom' operation returned v . In all cases, vertex v has been assigned or possibly executed by the end of round k and thus, potential decreases by $\frac{2}{3}\phi_i(v)$.

Thus, we conclude that if a thief targets a worker $q \in D_i$ as victim at round $k \geq i$, then the potential decreases by at least

$$\frac{2}{3}\phi_i(u) \geq \frac{2}{3}\frac{3}{4}\Phi_i(q) = \frac{1}{2}\Phi_i(q).$$

Second, we use Ball and Bins Lemma to establish the total decrease in potential.

Consider now all P workers and P steal attempts that occur at or after round i . For each worker q in D_i , assign a weight of $\frac{1}{2}\Phi_i(q)$ and for each worker in H_i , assign a weight of 0. The total weight is thus $\frac{1}{2}\Phi_i(D_i)$. Using the Balls-and-Bins Lemma with $\beta = 1/2$, we conclude that the potential decreases by at least $\beta W = \frac{1}{4}\Phi_i(D_i)$ with probability greater than $1 - \frac{1}{(1-\beta)e} > \frac{1}{4}$.

□

Important. For this lemma to hold, it is crucial that a steal attempt does not fail unless the deque is empty or the vertex being targeted at the time is popped from the deque is some other way. This is why, we required the popTop operation called by a worker to fail only if the top vertex is removed from the deque by another worker.

Theorem D.8 (Run-Time Bound). Consider any multithreaded computation with work W and span S and execute it with non-blocking work stealing with P workers in a dedicated environment. The execution time is

- $O(W/P + S)$ in expectation, and
- $O(W/P + S + \lg(1/\epsilon))$ with probability at least $1 - \epsilon$ for any $\epsilon > 0$.

Proof. The Steal Attempt-Bound Lemma bounds the execution time in terms of steal attempts. We shall prove bounds on the number of steal attempts.

We break execution into *phases* of $\Theta(P)$ steal attempts. We show that with constant probability, a phase causes the potential to drop by a constant factor, and since the potential starts at $\Phi_0 = 3^{2S-1}$ and ends at zero, and is always an natural number, we can bound the number of phases.

The first phase begins at round 1 and ends at the first round k , where at least P throws occur during the interval $[1, k]$. The second phase begins at round $k + 1$ and so on.

Consider a phase $[i, j)$, where the next phase begins at round j . We show that $\Pr[\Phi_j \leq \frac{3}{4}\Phi_i] < \frac{1}{4}$.

Recall that the potential can be partitioned as $\Phi_i = \Phi_i(H_i) + \Phi_i(D_i)$.

- Since the phase contains at least P steal attempts, by P Steal Attempts Lemma, we know that $P[\Phi_i - \Phi_j \geq \frac{1}{4}\Phi_i(D_i)] > \frac{1}{4}$.
- We need to show that the potential also drops by a constant fraction of $\Phi_i(H_i)$. Consider some worker q in H_i . If q does not have an assigned vertex, then $\Phi_i(q) = 0$. If q has an assigned vertex v , then $\Phi_i(q) = \phi_i(v)$. In this case, worker q executes v at round i and the potential decreases by at least $\frac{1}{3}\phi_i(u)$ by an argument included in the proof of [problem:work-stealing::potential-decrease](#), Decreasing Potential Lemma.

Summing over all workers in H_i , we have $\Phi_i - \Phi_j \geq \frac{1}{3}\Phi_i(H_i)$.

Thus we conclude that $P[\Phi_i - \Phi_j \geq \frac{1}{4}\Phi_i] > \frac{1}{4}$. In other words, we established that in any phase, potential drops by a quarter with some probability $\frac{1}{4}$.

Define a phase to be *successful* if it causes the potential to decrease by at least a quarter fraction. We just established that phase is successful with probability at least 0.25. Since the start potential $\Phi_0 = 3^{2S-1}$ and ends at zero and is always an integer, the number of successful phases is at most $(2S - 1) \log_{4/3} 3 < 8S$.

The expected number of phases to obtain a single successful phase is distributed geometrically with expectation 4. Therefore, the total expected number of phases is $32S$, i.e., $O(S)$. Since each phase contains $O(P)$ steal attempts, the expected number of steal attempts is $O(SP)$.

We now establish the high-probability bound.

Suppose that the execution takes $n = 32S + m$ phases. Each phase succeeds with probability

at least $p = \frac{1}{4}$, so the expected number of successes is at least $np = 8S + m/4$. Let X the number of successes. By Chernoff bound

$$P[X < np - a] < e^{-a^2/2np},$$

with $a = m/4$. Thus if we choose $m = 32S + 16 \ln 1/\epsilon$, then we have

$$P[X < 8S] < e^{-(m/4)^2/(16S+m/2)} \leq e^{-(m/4)^2/(m/2+m/2)} = e^{-m/16} \leq e^{-16 \ln 1/\epsilon/16} = \epsilon.$$

Thus the probability that the execution takes $64S + 18 \ln 1/\epsilon$ phases or more is less than ϵ .

We conclude that the number of steal attempts is $O(S + \lg 1/\epsilon P)$ with probability at least $1 - \epsilon$.

□

Remark. The material presented here is adapted from the paper:

N. S. Arora, R. D. Blumofe, and C. Greg Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems*, 34(2), 2001.

In this chapter, we consider dedicated environments, and simplify and streamline the proof for this case. The original paper considers multiprogrammed environments.