Algorithm Design: Parallel and Sequential

Umut A. Acar || Guy E. Blelloch

September 2017

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

Part I

Introduction and Preliminaries

Chapter 1

Introduction

The topic of this book might best be described as **problem solving with *parallel* computers**. The idea is you have some problem to solve (e.g. finding the shortest path from your room to your first class), and you want to use your computer to solve it for you. Your primary concern is probably that your answer is correct (e.g. you would likely be unhappy to find yourself at the wrong class). However, you also care that you can get the answer reasonably quickly (e.g. it would not be useful if your computer had to think about it until next semester).

This book is therefore about different aspects of problem solving with computers. It is about defining precisely the problem you want to solve. It is about learning the different techniques that can be used to solve such a problem, and about designing algorithms using these techniques. It is about designing abstract data types that can be used in these algorithms, and data structures that implement these types. And, it is about analyzing the cost of those algorithms and comparing them based on their cost.

Unlike traditional books on algorithms and data structures, which are concerned with sequential algorithms (ones that are correct and efficient on sequential computers), in this book we are concerned with parallel algorithms (ones that are correct and efficient on parallel computers). However, in the approach we take in this book , sequential and parallel algorithms are not that different. Indeed the book covers most of what is covered in a traditional sequential algorithms course. In the rest of this chapter we discuss why it is important to study parallelism, why it is important to separate interfaces from implementations, and outline some algorithm-design techniques.

1.1 Parallelism

The term "parallelism" or "parallel computing" refers the ability to run multiple computations (tasks) at the same time.

Parallel systems. 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. 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. At the larger scale many such chips 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.

There are several reason for why such parallel systems and thus parallelism has become so important. First, parallelism is simply more powerful than sequential computing, where only one computation can be run at a time, because it enables solving more complex problems in shorter time. For example,an Internet search is not as effective if it cannot be completed at "interactive speeds", completing in several 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. As it turns out, performing a computation twice as fast sequentially requires eight times as much energy. Precisely speaking, energy consumption is a cubic function of clock frequency (speed). With parallelism we don't need more energy to speed up a computation, at least in principle. For example, to perform a computation in half the time, 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 while consuming the same amount of energy. In reality, there are some overheads and we will need more energy, for example, to divide the computation and combine the results. Such overheads are usually small, e.g., constant fraction over sequential computation, but can be larger. These two factors—time and energy—have become increasingly important in the last decade, catapulting parallelism to the forefront of computing.

Example 1.1. 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 that 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 kitchen utensils.

Parallel software. The important advantage of using a parallel instead of a sequential algorithm is the ability to perform sophisticated computations quickly enough to make them practical or relevant, without consuming large amounts of energy.

Example 1.2. Example timings (reported in seconds) for some algorithms on 1 and on 32 cores.

	Sequential	Parallel	
		1-core	32-core
Sorting 10 million strings	2.9	2.9	.095
Remove duplicates for 10 million strings	.66	1.0	.038
Minimum spanning tree for 10 million edges	1.6	2.5	.14
Breadth first search for 10 million edges	.82	1.2	.046

One way to quantify this advantage is to measure the performance improvements of parallelism. Example 1.2 illustrates the sort of performance improvements that can achieved today. These times are 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).

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 between sequential and parallel computation is that in the latter certain computations will be performed at the same time but this is possible only if the computations are actually ***independent***, i.e., do not depend on each other. Thus when designing a parallel algorithm, we have to identify the underlying dependencies in the computation to be performed and avoid creating unnecessary dependencies.

Example 1.3. Going back to or cooking example, suppose that we want to make a frittata in our kitchen with 3 cooks. Making a frittata is quite a bit more involved than just boiling eggs. We have to be careful about the dependencies between various tasks. For example, vegetables cannot be sauteed before they are washed and chopped the eggs cannot be added to the meal before being broken or before the vegetables are sauteed, etc.

An important challenge is therefore to design algorithms that minimize the dependencies so that more things can run in parallel. This design challenge is a primary focus of this book.

Another important challenge concerns the coding and usage of a parallel algorithm in the real world. The many forms of parallelism, ranging from small to large scale, and from general to special purpose, has led to many different programming languages and system for coding parallel algorithms. These different programming languages systems often target a particular kind of hardware, and even a particular kind of problem domain. For example, 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 for all possible applications. As it turns out, one can easily spend weeks or even months optimizing a parallel sorting algorithm on specific parallel hardware, such as a GPU.

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.

1.2 Work and Span

In this book we analyze the cost of algorithms in terms of two measures: ***work*** and ***span***. 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, which will be described in more detail in Chapter 4.

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 split among multiple processors and thus reduce the time.

The interesting question is to what extent can the work be shared. Ideally we would like the work to be evenly shared. If we had W work and P processors to work on it in parallel, then even sharing would imply each processor does $\frac{W}{P}$ work, and hence the total time is $\frac{W}{P}$. An algorithm that achieves such ideal sharing is said to have ***perfect speedup***. Perfect speedup, however, is not always possible. If our algorithm is fully sequential (each operation depends on prior operations, leaving no room for parallelism), for example, we can only take advantage of one processor, and the time would not be improved at all by adding more. There is no sharing—-at least in parallel. 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.

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

As we shall see in Section 4.2.4, even though work and span, are abstract machine-independent models of cost, 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.

Example 1.4. 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

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

In Chapter 8 we will see that the span for `mergeSort` is

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

Thus, when sorting a million keys, work is $10^6 \lg(10^6) > 10^7$, and span is $\lg^2(10^6) < 500$. This means that 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 should note that in practice, the numbers might be more conservative due to natural overheads of parallel execution.

In this book 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 and the span as the sum of the span of sequential subcomputations or maximum of the work of the parallel subcomputations. More concretely, given two subcomputations, we can calculate the work and the span of their sequential and parallel composition as follows.

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

In the table, W_1 and S_1 are the work and span of the first subcomputation and W_2 and S_2 of the second. The 1 that is added to each rule is the cost of composing the subcomputations.

The intuition behind these rules 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 1.5. 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? The greedy scheduling principle tells us that we need no more that $150/3 + 5 = 55$ minutes. That is almost a factor 3 speedup over the 150 that we would need with just one cook.

How do we actually realize the greedy schedule? In this case, this is simple, all we have to do is divide the eggs equally between our cooks.

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 1.6. [Work Efficiency] We say that a parallel algorithm is ***asymptotically work efficient*** or, simply ***work efficient***, if the work is asymptotically the same as the time for an optimal sequential algorithm that solves the same problem.

For example, the parallel mergeSort described in Example 1.4 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.

1.3 Specification, Problem, Implementation

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

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

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.

Algorithm Specification 1.7. [Comparison Sorting] Given a sequence A of n elements taken from a totally ordered set with comparison operator $\leq$, return a sequence B containing the same elements but such that $B[i] \leq B[j]$ for $0 \leq i < j < n$.

The specification describes ***what*** the algorithm should do but it does not describe ***how*** it achieves what is asked. This is intentional 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***. For example, 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 1.8. [Comparison Sort: Work-Efficient and 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 $O(n \log n)$ work but no bounds on the span. Here is another specification that requires even smaller span but allows for more work.

Cost Specification 1.9. [Comparison Sort: Fast 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.

As we discussed, we usually care more about work and thus would prefer the first specification; there might, however, be cases where the second specification is preferable.

Data Structure Specification. We specify a data structure by describing what is expected of the data structure via an ***Abstract Data Type (ADT) specification***. For example, we can specify a priority queue ADT as follows.

Abstract Data Type 1.10. [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.

As with algorithms, we usually give cost specifications to data structures. The following cost specification describes a class of basic priority queue data structures.

Cost Specification 1.11. [Priority Queue: Basic] The work and span of a priority queue operations are as follows.

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

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. An ***algorithms problem*** requires designing an algorithm that satisfies the given algorithm specification and cost specification if any. 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. The difference between an algorithms 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.

When we consider problems, it is usually clear from the context whether we are talking about algorithms or data structures. In such cases, we use the simpler terms ***specification*** and ***problem*** to refer to the algorithm/ADT specification and the corresponding problem respectively.

Implementation. We can solve an algorithms or a data-structures problem by presenting an ***implementation***. 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. We note that while the distinction between problems and algorithms is common in the literature, the distinction between abstract data types and data structures is less so.

We describe an algorithm 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.

Algorithm 1.12. [Insertion Sort]

```
insSort $f$ $s$ =
  if $|s|$ = 0 then $\langle\,\rangle$
  else insert $f$ $s$[0]  (insSort $f$ (s[1,...,n-1]))
```

In the algorithms, f is the comparison function and s is the input sequence. The algorithm uses a function (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 algorithms problem, since we are not specifying how it is implemented, but just specifying its functionality. We might also be given a cost specification for insert, e.g., for a sequence of length n the cost of insert should be $O(n)$ work and $O(\log n)$ span. Given this cost we can determine the overall asymptotic cost of sort using our composition rules described in the last section. 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.

Similarly, we can specify 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 1.13. [On the importance of specification] Several reasons underline the importance of distinguishing between specification and implementation. First, we want to be able to use a specification without knowing the details of an implementation that matches that specification. In many cases the specification of a problem is quite simple, but an efficient algorithm or data structure that solves it, i.e., the implementation, is complicated. Specifications allow us abstract from implementation details. Second, we want to be able to change or improve implementations over time. As long as each implementation matches the same specification, and the user relied only on the specification, then he or she can continue using the new implementation without worrying about their code breaking. Third, when we compare the performance of different algorithms or data structures it is important that we are not comparing apples with oranges. We have to make sure the algorithms we compare are solving the same problem, because subtle differences in the problem specification can make a significant difference in how efficiently that problem can be solved.

1.4 Problems

1-1 Essence of parallelism
When designing a parallel algorithm, we have to be careful in identifying the computations that can be performed in parallel. What is the key property of such computations?

Chapter 2

Mathematical Preliminaries

We present an overview of basic mathematical definitions used throughout the book. We assume familiarity with college-level mathematics. This chapter is far from complete but further details can be found in standard texts.

2.1 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. 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 $\mathbb{N} = \{0, 1, 2, \ldots\}$. 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.

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., $A \cap B = \emptyset$.

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 $b \in B$. In set notation this is

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

Example 2.1. The Cartesian product of $A = \{0, 1, 2, 3\}$ and $B = \{a, b\}$ is the set of all pairings:

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

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 P 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 2.2. 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 a not partition of A, because the element 6 is contained in any of the blocks.

2.2 Relations and Functions

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.

A ***mapping 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 $(a, b) \in R$. A mapping is also called a ***function***.

Example 2.3. 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.

The domain of Y is $\{0, 1, 3\}$ and the range is $\{a, b\}$. It is, however, not a sequence since there is a gap in the domain.

2.3 Graph Theory

2.3.1 Basic Definitions

Definition 2.4. 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***.

In a digraph, each arc is an ordered pair $e = (u, v)$. A digraph can have ***self loops*** (u, u). Example 2.6 shows a digraph with 4 vertices and 4 arcs.

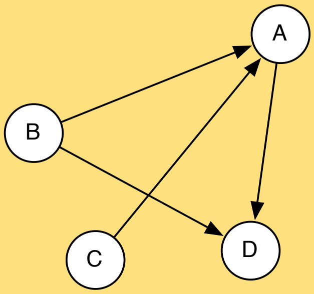
Definition 2.5. [Undirected graph] An ***undirected graph*** is a pair $G = (V, E)$ where

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

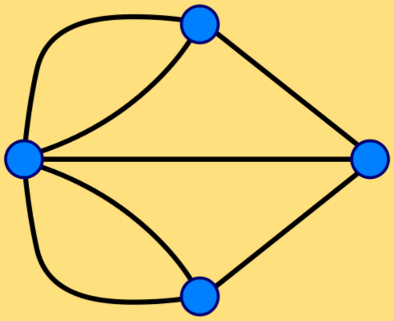
In an undirected graph, each edge is an unordered pair $e = \{u, v\}$ (or equivalently $\{v, u\}$). By this definition an undirected graph cannot have self loops since $\{v, v\} = \{v\} \notin \binom{V}{2}$.

While directed graphs represent possibly asymmetric relationships (my web page points to yours, but yours does not necessarily point back), undirected graphs represent symmetric relationships. Directed graphs are in some sense more general than undirected graphs since we can easily represent an undirected graph by a directed graph by replacing an edge with two arcs, one in each direction. Indeed, this is usually the way we represent undirected graphs in data structures.

Example 2.6. An example directed graph with 4 vertices:



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



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 and more comprehensively in Chapter 16, we consider weighted graphs, where the weights on edges can represent a distance, a capacity or the strength of the connection.

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.

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

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.

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.

Paths. A ***path*** in a graph is a sequence of adjacent vertices. More formally for a graph $G = (V, E)$, $\mathit{Paths}(G) = \{P \in V^+ \mid 1 \leq i < |P|, (P_i, P_{i+1}) \in E\}$ is the set of all paths in G, 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 2.7. 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.

Reachability and connectivity. A vertex v is ***reachable*** from a vertex u in G if there is a path starting at v and ending at u in G. We use $R_G(v)$ to indicate the set of all vertices reachable from v in G. An undirected graph is ***connected*** if all vertices are reachable from all other vertices. A directed graph is ***strongly connected*** if all vertices are reachable from all other vertices.

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

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.

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

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.

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

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.

Sparse and dense graphs. By convention we will use the following definitions:

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

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 typically on algorithms that work well for sparse graphs.

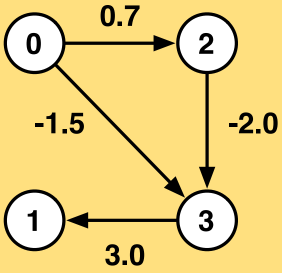
2.3.2 Weighted Graphs

Many applications of graphs require associating weights or other values with the edges of a graph. Such graphs can be defined as follows.

Definition 2.9. [Weighted and Edge-Labeled Graphs] An *edge-labeled graph* or a *weighted graph* is a triple $G = (E, V, w)$ where $w \colon E \to L$ 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 2.10. An example directed weighted graph.



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

2.3.3 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.

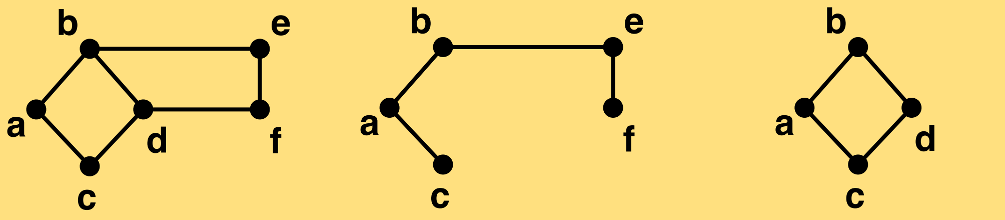
Definition 2.11. [Subgraph] Let $G = (V, E)$ and $H = (V', E')$ be two graphs. H is a subgraph of if $V' \subseteq V$ and $E' \subseteq E$.

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 2.12. [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 2.13. Some vertex induced subgraphs:



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

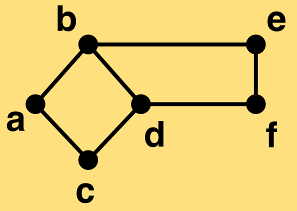
Although we will not use it in this book, it is also possible to define an induced subgraph in terms of a set of edges by including in the graph all the vertices incident on the edges.

Definition 2.14. [Edge-Induced Subgraph] The subgraph of $G = (V, E)$ induced by $E' \subseteq E$ is a graph $H = (V', E')$ where $V' = \cup_{e \in E} e$.

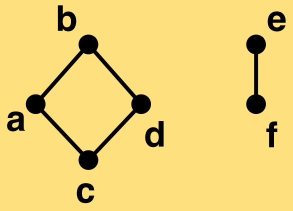
2.3.4 Connectivity and Connected Components

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 2.15. Two example graphs shown. The first in connected; the second is not.



Graph 1



Graph 2

An important subgraph of an undirected graph is a ***connected component*** of a graph.

Definition 2.16. [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 Example 2.15, the first graph has one connected component (hence it is connected); the second has two connected components.

Using vertex-induced subgraphs, we can specify a connected component of a graph by simply specifying the vertices in the component.

Example 2.17. Connected components of our second example graph Example 2.15 can be specified as $\{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$ and $\{\texttt{e}, \texttt{f}\}$.

2.3.5 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, $B \in P$. We define a ***partition of a graph*** as a partition of its vertex set. More precisely, given graph $G = (V, E)$, we define a partition of G as a set of graphs $P = \{G_1 = (V_1, E_1) \ldots G_k = (V_k, E_k)\}$, where $\{V_1, \ldots, V_k\}$ is a (set) partition of V and $G_1, \ldots, G_k$ are vertex-induced subgraphs of G with respect to $V_1, \ldots, V_k$ respectively. As in set partitioning, we use the term ***part*** or ***block*** to refer to each vertex-induced subgraph $G_1, \ldots, G_k$.

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.

2.3.6 Trees

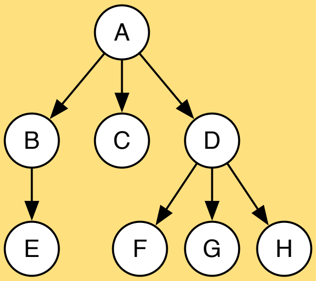
An undirected graph is a tree if it does not have cycles and it is connected. A ***rooted tree*** is a tree with a distinguished root node that can be used to access all other nodes (Definition 2.18). An example of a rooted tree along with the associated terminology is given in Example 2.19.

Definition 2.18. [Rooted Tree] 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.

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 2.19. An example of a rooted tree:



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

.

Chapter 3

SPARC: A Strict Language for Parallel Computing

To describe the algorithms covered in this book, we use a pseudocode notation that is based on a language, which we call SPARC. 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.

3.1 Functional Algorithms

Many of the algorithms in this book are *purely functional*. To understand what this means, let's review first what it means for an algorithm to be pure and functional. In our review below, we use an informal style.

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, printing on the screen, or writing to a file. For example, consider a function that given a natural number n as an argument computes and returns the n^{th} Fibonacci number, and also updates the argument n by overwriting it with the return value. This function has the side effect of changing the value of its argument. We say that a computation is ***pure*** if it doesn't perform any side effects. Pure computations return a value without performing any side effects. In contrast an ***impure*** or ***imperative*** computation can perform side effects. The Fibonacci function described above is impure. Pure computations correspond closely with mathematical systems or notation, where for example, determining the result of a calculation of formula does not affect the result of another. This 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 visible to

Remark 3.1. [The lambda calculus]
As with most functional programming languages, the ML class of languages are based on the ***lambda calculus*** (or λ calculus), a computational model developed by Alonzo Church in 1932. The lambda calculus is a very simple language consisting of expressions e which can only have three forms:

$x,$	:	a *variable name*,
$(\lambda\, x\, .\, e)$	:	a *function definition*, where x is the argument and e is the body, or
$e_1\; e_2$	:	a *function application*, where e_1 and e_2 are expressions;

and effectively only a single rule for processing expressions, called ***beta reduction***. For any function application for which the left hand expression is a function definition, 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. Computation in the lambda calculus consists of applying beta reduction until there is nothing left to reduce.

In the early 30s Church argued that anything that can be "effectively computed" can be computed with the lambda calculus, and therefore that 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***.

Although the lambda calculus allows beta reduction to be applied in any order, most functional programming languages use a specific order. The ML class of languages use call-by-value, which means that for a function application $(\lambda\, x\, .\, e_1)\; e_2$, the expression e_2 must be evaluated to a value before applying beta reduction. Other languages, such as Haskell, allow for the beta reduction before e_2 becomes a value. 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. Call-by-name is inefficient since it creates redundant computations, while call-by-need is both inherently sequential and not well suited for analyzing costs. In this book we, therefore, only use call-by-value. All these reduction orders are Church-Turing complete.

the outside world or is not used by any other computation, 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.

A programming language is called ***functional***, if it does not restrict the use of functions any more than the use of other values such as natural numbers. This principle is sometimes referred to as "functions as first-class values" or "functions as first-class citizens". Functional programming languages do not distinguish between a function and other values: both can be used as building blocks of values, stored in memory, and passed as arguments to functions, etc. For example, a function that returns the number `1` under some argument and the natural number `1` are the same. Treating functions as values leads to a powerful way to code. For example, we can write ***higher-order functions*** that take functions as their arguments.

Combining the concepts of purity and functional programming, the term ***purely functional algorithm*** refers to an algorithm that is both pure and functional, i.e., it is described in a functional language and it avoids use of observable side effects such as imperative updates to memory locations that can be observed and can thus have an effect on other computations. Purely functional algorithms are particularly useful for parallelism for two important reasons.

First, purely functional algorithms are safe for parallelism. In particular, parts of the algorithm may be executed in parallel without affecting each other. In contrast, in imperative programming, the programmer must take care that side effects don't alter the meaning of the program in unintended ways. The problem is that preventing side effects from altering the meaning of the program is very difficult, because 1) depending on the exact execution order (timing) of components, side effects may cause a computation to return different results at different times, and 2) there are exponentially many different orderings. For example, consider three tiny functions, 5-lines each, that read and write from the same memory location. To verify that these three functions execute correctly in parallel, the programmer may have to consider all of the $15!/(5!)^3 = 756756$, different orders in which the lines of these functions may interleave during evaluation. It is nearly impossible for any human being to comprehend such numbers of different possibilities. In a highly parallel system where there are usually many more than functions, this problem can quickly overwhelm any human programmer.

Second, 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.

We note that coding a purely functional algorithm does not require a purely functional programming language. In fact, a purely functional algorithm can be coded in essentially any programming language—one just needs to be much more 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 purely functional algorithms. The techniques that we describe thus are applicable to imperative programming languages as well.

Remark 3.2. [Race conditions]
Side effects that alter the result of the computation based on the evaluation order (timing) are called ***race conditions***. 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. 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.”

Remark 3.3. [Heisenbug]
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.

3.2 The SPARC Language

We describe the syntax and the semantics of the core subset of the SPARC language. ***Syntax*** refers to the structure of the program itself, while ***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 talk about cost semantics in Chapter 4. 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.

3.2.1 Syntax and Semantics

Definition 3.4 shows the syntax of SPARC. A SPARC program is an expression, whose syntax, describe the computations that can be expressed in SPARC. When evaluated 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.

Identifiers. In SPARC, variables, type constructors, and data constructors are given a name, or an ***identifier***. An identifer 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.

Definition 3.4. [SPARC expressions]

Identifier	id	$:=$	$\ldots$	
Variables	x	$:=$	id	
Type Constructors	$tycon$	$:=$	id	
Data Constructors	$dcon$	$:=$	id	
Patterns	p	$:=$	x	variable
		$\mid$	(p)	parenthesization
		$\mid$	p_1, p_2	pair
		$\mid$	$dcon(p)$	data pattern
Types	τ	$:=$	$\mathbb{Z} \mid \mathbb{B}$	base type
		$\mid$	$\tau[*\tau]^+ \mid \tau \to \tau$	products and functions
		$\mid$	$tycon$	type constructors
		$\mid$	dty	data types
Data Types	dty	$:=$	$dcon\ [\textbf{of}\ \tau]$	
		$\mid$	$dcon\ [\textbf{of}\ \tau] \mid dty$	
Values	v	$:=$	$0 \mid 1 \mid -1 \mid 2 \mid -2 \ldots$	integers
		$\mid$	$\texttt{true} \mid \texttt{false}$	booleans
		$\mid$	$\texttt{not} \mid \ldots$	unary operations
		$\mid$	$\wedge \mid \texttt{plus} \mid \ldots$	binary operations
		$\mid$	v_1, v_2	pairs
		$\mid$	(v)	parenthesization
		$\mid$	$dcon(v)$	constructed data
		$\mid$	$\lambda\, p\, .\, e$	anonymous functions
Expression	e	$:=$	x	variables
		$\mid$	v	values
		$\mid$	$e_1\ \mathrm{op}\ e_2$	infix operations
		$\mid$	$e_1\ ,\ e_2$	sequential pair
		$\mid$	$e_1 \mid\mid e_2$	parallel pair
		$\mid$	(e)	parenthesization
		$\mid$	$\textbf{case}\ e_1\ [\mid p\ \textbf{=>}\ e_2]^+$	case
		$\mid$	$\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3$	if then else
		$\mid$	$e_1\ e_2$	function application
		$\mid$	$\textbf{let}\ b^+\ \textbf{in}\ e\ \textbf{end}$	local bindings
Bindings	b	$:=$	$x(p) = e$	function binding
		$\mid$	$p = e$	variable binding
		$\mid$	$\textbf{type}\ tycon = \tau$	type binding, synonyms
		$\mid$	$\textbf{type}\ tycon = dty$	type binding, datatype

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.

Types. Types of SPARC include base types such as integers $\mathbb{Z}$, booleans $\mathbb{B}$, product types such as $\tau_1 * \tau_2 * \ldots \tau_n$, function types $\tau_1 \to \tau_2$ with domain τ_1 and range τ_2, as well as user defined 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 = Point2D of ℤ ⋆ ℤ
           | Point3D of ℤ ⋆ ℤ ⋆ ℤ
```

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 int ⋆ plist.
```

Based on this definition the list

```
Cons (Point2D (0,0),  Cons (Point2D (0,1), Cons (Point2D (0,2), Nil)))
```

defines a list consisting of three points.

Example 3.5. [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.

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

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 int_option = INone | ISome of ℤ.
```

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

Values and 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.

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 $\wedge$ 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 $\boldsymbol{\lambda}\, p\ .\, e$ is a function whose arguments are specified by the pattern p, and whose body is the expression e. For example, the function $\boldsymbol{\lambda}\, x\ .\, x\texttt{+1}$ takes a single variable as an argument and adds one to it. The function $\boldsymbol{\lambda}\, (x\, ,\, y)\ .\, x$ takes a pairs as an argument and returns the first component of the pair.

Expressions, denoted by e and variants (with subscript, superscript, prime), are defined inductively, because in many cases, an expression contains other expressions.

An ***infix expression***, $e_1\ \texttt{op}\ e_2$, involve two expressions and an infix operator `op`. The infix operators include $+$ (plus), $-$ (minus), $\times$ (multiply), / (divide), $<$ (less), $>$ (greater), $\vee$ (or), and $\wedge$ (and). For all these operators the infix expression $e_1\ \texttt{op}\ e_2$ 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

$$3 + 4 \times 5$$

the $\times$ has a higher precedence than $+$ and therefore the expression is equivalent to $3 + (4 \times 5)$. Furthermore all operators are left associative unless stated otherwise, i.e., that is to say that $a\ \texttt{op}_1\ b\ \texttt{op}_2\ c = (a\ \texttt{op}_1\ b)\ \texttt{op}_2\ c$ if $\texttt{op}_1$ and $\texttt{op}_2$ have the same precedence. For example

$$5 - 4 + 2$$

will evaluate to $(5 - 4) + 2 = 3$ not $5 - (4 + 2) = -1$ since $-$ and $+$ have the same precedence.

Expressions include two special infix operators: "," and "||", for generating ordered pairs, or tuples, either sequentially or in parallel. The ***comma*** operator "," 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 "||", as in the infix expression $(e_1 \mid\mid e_2)$, 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, as 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 3.6. The expression

$$\boldsymbol{\lambda}\,(x\texttt{,}y)\ \ \texttt{.}(x\ \star\ x\texttt{,}\ \ y\ \star\ y)$$

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

$$\boldsymbol{\lambda}\,(x\texttt{,}y)\ \ \texttt{.}(x\ \star\ x\ ||\ y\ \star\ y)$$

is a function that take 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.

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.

An ***if-then-else expression***, **if** e_1 **then** e_2 **else** e_3, 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.

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, lets 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 callSPARC a ***strict*** language. In strict 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 3.7. The expression
$(\boldsymbol{\lambda}\,(x, y)\ \texttt{.}\,x/y)\ (8, 2)$
evaluates to 4 since 8 and 2 are bound to x and y, respectively, and then divided.

The expression
$(\ \boldsymbol{\lambda}\,(f, x)\ \texttt{.}\,f(x, x)\ )\ (\texttt{plus}, 3)$
evaluates to 6 since f is bound to the function `plus`, x is bound to 3, and then `plus` is applied to the pair $(3, 3)$.

The expression $(\ \boldsymbol{\lambda}\,\texttt{x}\ \texttt{.}(\ \boldsymbol{\lambda}\,\texttt{y}\ \texttt{.x + y}\ )\ )$ 3 evaluates to a function that adds 3 to any integer.

The ***let expression***, **let** b^+ **in** e **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. Each ***variable binding***, $p = e$, consists of a ***pattern***, p, on the left and an expression, e, on the right. The expression is evaluated and its value is assigned to the variable(s) in the pattern. The value of the expression must therefore match the structure of the pattern. For example if the pattern on the left is a pair of variables (x, y) then the expression on the right must evaluate to a pair. The two elements of the pair are assigned to the variables x and y, respectively. Each ***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. A let expression **let** b^+ **in** e **end** evaluates to the result of evaluating e giving the variable bindings defined in b. Each ***type binding*** can equate a type to a base type or a data type.

Example 3.8. In the following expression

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

Line 2 binds the variable x to $2 + 3 = 5$; Line 3 defines a function $f(w)$ which returns a pair; Line 4 applies the function f to $x - 1 = 4$ returning the pair $(4 \times 4, 4 - 2) = (16, 2)$, which y and z are bound to, respectively (i.e., $y = 16$ and $z = 2$); and finally in Line 6 x, y and z are added giving $5 + 16 + 2$. The result of the expression is therefore 23.

We need to 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 $x = \lambda\ p\,.\,e$, since 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 3.9. The expression:

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

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

Syntax 3.10. [While loops] The ***while loop*** can appear as one of the bindings b in a **let** expression and has the syntax the following syntax.

$$\begin{array}{lll} xs & = & \textbf{start}\ \ p\ \ \textbf{and} \\ & & \textbf{while}\ \ e_{continue}\ \ \textbf{do} \\ & & \quad b^+ \end{array}$$

Here xs are the result variables holding the values computed by the while loop, the pattern p is the initial value for xs. Such a **while** loop evaluates by setting xs to pattern p and then evaluating the loop until $e_{continue}$ evaluates to `false`. In a typical use the body of the loop b^+ defines the variables xs, whose final value will be the value of xs when the loop terminates.

We define the while loop syntax as equivalent to the following pair of bindings.

$$\begin{array}{lll} xs & = & \textbf{start}\ \ p\ \ \textbf{and} \\ & & \textbf{while}\ \ e_{continue}\ \ \textbf{do} \\ & & \quad b^+ \end{array} \qquad \equiv \qquad \begin{array}{l} f\ \ xs\ = \\ \quad \textbf{if}\ \ \texttt{not}\ \ e_{continue}\ \ \textbf{then}\ \ xs \\ \quad \textbf{else}\ \ \textbf{let}\ \ b^+\ \ \textbf{in}\ \ f\ \ xs\ \ \textbf{end} \\ \\ xs\ =\ f\ \ p \end{array}$$

Here xs, p, $e_{continue}$ and b^+ are substituted verbatim. The loop is expressed as a function that takes xs as an argument and runs the body of the loop until the expression $e_{continue}$ becomes false at which time the variables xs is returned. The variables xs are passed from one iteration of the while to the next each of which might redefine them in the bindings. After the loop terminates, the variables take on the value they had at the end of the last iteration.

Loops. SPARC does not have explicit syntax for loops but loops can be implemented with recursion. Throughout the book, we use the following syntactic sugar for expressing while loops. Syntax 3.10 defines the syntax of the while loops. In the definition, the expressions $e_{continue}$ determines the termination condition for the loop, while the bindings b^+ constitute the body of the loop. In a typical use, the body of the loop assigns to the variables of xs, effectively determining the return value of the while loop.

When evaluated a while loop starts by matching the variables xs to the pattern p and then continues to evaluate the while loop in the usual fashion. It first checks the value of $e_{continue}$, if it is false, then the evaluation completes. If not, then the bindings in b^+, which can use the variables xs, are evaluated. Having finished the body, evaluation jumps to the beginning of the **while** and evaluates the termination condition $e_{continue}$, and continues on executing the loop body and so on.

Example 3.11. The following code sums the squares of the integers from 1 to n.

```
sumSquares(n) =
   let
      (s, n) =
         start (0, n) and
         while n > 0 do
            s = s + n * n
            n = n - 1
   in s end
```

By definition it is equivalent to the following code.

```
sumSquares(n) =
   let
      f(s, n) = if not (n > 0) then (n, s)
                else let
                       s = s + n * n
                       n = n - 1
                     in f(s, n) end
      (s, n) = f(0, n)
   in s end
```

Example 3.12. The piece of code below illustrates an example use of data types and higher-order functions.

```
let
  type point = Point2D of ℤ * ℤ
             | Point3D of ℤ * ℤ * ℤ

  inject3D (Point2D (x,y)) = Point3D (x,y,0)

  project2D (Point3D (x,y,z)) = Point2D (x,y)

  compose f g = f g

  p0 = (0,0)
  q0 = project3D p0
  p1 = (compose project2D inject3D) 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 `inject3D` takes a 2D point and transforms it to a 3D point by mapping it to a point on the $z = 0$ plane. The function `project2D` 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 id 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 3.13. 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 ℤ | Node of (tree, ℤ, tree)

find (t, x) =
  case x
  | 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 3.14.
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.

3.2.2 Type System of SPARC

Type me if you can.

.

Part II

Fundamentals

Chapter 4

Algorithm Design and Analysis

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

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

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

4.1 Asymptotic Complexity

If we analyze an algorithm precisely, we usually end up with an equation in terms of a variable characterizing the input. For example, by analyzing the work of the algorithm A for problem P in terms of its input size n, we may obtain the equation: $W_A(n) = 2n\lg n + 3n + 4\lg n + 5$. By applying the analysis method to another algorithm, algorithm B, we may derive the equation: $W_B(n) = 6n + 7\lg^2 n + 8\lg n + 9$.

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

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

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

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

In algorithm design, the three most important asymptotic functions are the "Big-Oh" $O(\cdot)$, "Omega" $\Omega(\cdot)$ and "Theta" $(\Theta(\cdot))$. We also discuss some important conventions that we will follow when doing analysis and using these notations. All of these asymptotic functions are defined based on the notion of asymptotic dominance, which we define below. Throughout this chapter and more generally in this book, the cost functions must be mappings from the domain of natural numbers to real numbers. Such functions are sometimes called ***numeric functions***.

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

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

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

Upper bounds and the Big-Oh: O$(\cdot)$. The asymptotic expression $O(f(n))$, read "order of $f(n)$", or "big-oh of $f(n)$" denotes the set of all functions that are asymptotically dominated by the function $f(n)$. This means that the set consists of the functions that grow at the same or slower rate than $f(n)$. We write $g(n) \in O(f(n))$ to refer to a function $g(n)$ that is in the set $O(f(n))$. We think of $f(n)$ being an ***upper bound*** for $g(n)$ because $f(n)$ grows faster than $g(n)$ as n increases.

Definition 4.2. For a function $g(n)$, we say that $g(n) \in O(f(n))$ if there exist positive constants n_0 and c such that for all $n \geq n_0$, we have $g(n) \leq c \cdot f(n)$.

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

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

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

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

Lower bounds and the Omega notation: $\boldsymbol{\Omega}(\cdot)$. The "big-Oh" notation gives us a way to upper bound a function but it says nothing about lower bounds. For lower bounds, we use the "Omega" notation: the expression $\Omega(f(n))$ denotes the set of all functions that asymptotically dominate the function $f(n)$. Intuitively this means that the set consists of the functions that grow faster than $f(n)$. We write $g(n) \in \Omega(f(n))$ to refer to a function $g(n)$ that is in the set $\Omega(f(n))$. We think of $f(n)$ being a ***lower bound*** for $g(n)$.

Definition 4.4. For a function $g(n)$, we say that $g(n) \in \Omega(f(n))$ if there exist positive constants n_0 and c such that for all $n \geq n_0$, we have $c.f(n) \leq |g(n)|$.

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

Definition 4.5. For a function $g(n)$, we say that $g(n) \in \Theta(f(n))$ if there exist positive constants n_0, c_1, and c_2 such that for all $n \geq n_0$, we have $c_1 \cdot f(n) \leq |g(n)| \leq c_2 \cdot f(n)$.

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

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

4.2 Cost Models: Machine and Language Based

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

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

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

There are certain advantages and disadvantages to both models.

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

The advantage to language-based models is that they simplify analysis of algorithms. The disadvantage is that the predicted cost bounds may not precisely reflect the cost observed when the algorithm is executed on actual hardware. This imprecision of the language model, however, can be minimized and even eliminated by defining the model to be consistent with the machine model and the programming-language environment assumed such as the compiler and the run-time system. When analyzing algorithms in a language-based model we don't need to care about how the language compiles or runs on the machine. Costs are defined directly in the language, specifically its syntax and its dynamic semantics that specifies how to evaluate the expressions of the language. We thus simply consider the algorithm as expressed and analyze the cost by applying the cost function provided by the model.

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

Remark 4.6. We note that both machine models and language-based models usually abstract over existing architectures and programming languages respectively. This is necessary because we wish to our cost analysis to have broader relevance than just a specific architecture or programming language. For example, machine models are usually defined to be valid over many different architectures such as an Intel Nehalem or AMD Phenom. Similarly, language-based models are defined to be applicable to a range of languages. In this book, we use an abstract language that is essentially lambda calculus with some syntactic sugar. As you may know the lambda calculus can be used to model many languages.

4.2.1 The RAM Model for Sequential Computation

Traditionally, algorithms have been analyzed in the Random Access Machine (RAM)[1] model. In this model, a machine consists of a single processor that can access unbounded memory; the memory is indexed by the non-negative integers. The processor interprets sequences of machine instructions (code) that are stored in the memory. Instructions include basic arithmetic and logical operations (e.g. $+$, $-$, $\star$, and, or, not), reads from and writes to arbitrary memory locations, and conditional and unconditional jumps to other locations in the code. Each instruction takes unit time. The execution-time, or simply ***time*** of a computation is measured in terms of the number of instructions executed by the machine.

This model is quite adequate for analyzing the asymptotic runtime of sequential algorithms; most work on sequential algorithms to date has used this model. It is therefore important to understand the model, or at least know what it is. One reason for the RAM's success is that it is relatively easy to reason about the cost of algorithms because algorithmic pseudo code and sequential languages such as C and C++ can easily be mapped to the model. The model is suitable for deriving asymptotic bounds (i.e., using big-O, big-Theta and big-Omega) but not for trying to predict exact runtimes. The reason for this is that on a real machine not all instructions take the same time, and furthermore not all machines have the same instructions.

One problem with the RAM model is that it assumes that accessing all memory locations has the same cost. On real machines this is not the case. In fact, there can be a factor of over 100 between the time for accessing a word of memory from the first level cache and accessing it from main memory. Various extensions to the RAM model have been developed to account for this cost. For example one variant is to assume that the cost for accessing the i^{th} memory location is $f(i)$ for some function f, e.g. $f(i) = \lg(i)$. Fortunately, however, most of the algorithms that turn out to be good in these more detailed models are also good in the RAM. Therefore analyzing algorithms in the simpler RAM model is often a reasonable approximation to analyzing in the more refined models. Hence the RAM has served quite well despite not fully accounting for the variance in memory costs. The model we use in this book also does not account for the variance in memory costs, but as with the RAM the costs can be refined.

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

4.2.2 The Parallel RAM Model

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

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

Since it is synchronous and it requires the algorithm designer to map computation to processors (perform manual scheduling), the PRAM model is can be awkward to work with. For simple parallel loops over n elements we could imagine dividing up the elements evenly among the processors—about n/p each, although there is some annoying rounding required since n is typically not a multiple of p. If the cost of each iteration of the loop is different then we would further have to add some load balancing. In particular simply giving n/p to each processor might be the wrong choice—one processor could get stuck with all the expensive iterations. For computations with nested parallelism, such as divide-and-conquer algorithms the mapping is much more complicated, especially given the highly synchronous nature of the model.

Even though we don't use the PRAM model, most of the ideas presented in this book also work with the PRAM, and many of them were originally developed in the context of the PRAM.

4.2.3 The Work-Span Model

In this book, we use a language-based cost model based on work and span to analyze parallel algorithms. As discussed in Section 4.2, have to be careful about the cost model to make sure that it can be mapped to real hardware by implementing the necessary compilation and runtime system support. Indeed, for the cost-model that we describe here, this is the case (see Section 4.2.4 for more details).

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

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

Example 4.7.

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

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

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

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

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

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

Definition 4.8 shows the precise definitions of the work and span of SPARC, our language for describing algorithms. In the definition and throughout this book, we write $W(e)$ for the work of the expression and $S(e)$ for its span. Both work and span are cost functions that map an expression to a cost measured as in units of time. As common in language-based models, the definition follows the definition expressions for SPARC (Chapter 3). We make one simplifying assumption in the presentation: instead of considering general bindings, we only consider the case where a single variable is bound to the value of the expression.

Eval()

Definition 4.8. [SPARC Cost Model] The work and span of SPARC expressions (Chapter 3) are defined as follows. The notation Eval(e) evaluates the expression e and returns the result, and the notation $[v/x]\ e$ indicates that all free (unbound) occurrences of the variable x in the expression e are replaced with the value v.

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

span

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

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

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

The additional 1 accounts for computation of the sum.

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

Example 4.9. Let expressions compose sequentially.

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

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

Example 4.10. The expression $(\texttt{fib}(6) \mid\mid \texttt{fib}(7))$ runs the two calls to `fib` in parallel and returns the pair $(8, 13)$. It does work

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

and span

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

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

Remark 4.11. In purely functional programs, it is always safe to run things in parallel if there is no explicit sequencing. Since in SPARC, we evaluate e_1 and e_2 sequentially, the span of the expression is calculated in the same way:

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

Note that this does not mean that the span and the work of the expressions are the same!

Since SPARC is purely functional language, we could have in fact evaluated e_1 and e_2 in parallel, wait for the to complete and perform the summation. In this case the span of would have been

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

Note that since we have to wait for both of the expressions to complete, we take the maximum of their span. Since we can perform the final summation serially after they both return, we add the 1 to the final span.

In this book, however, to make it more clear whether expressions are evaluated sequentially or in parallel we will assume that expressions are evaluated in parallel only when indicated by the syntax, i.e., when they are composed with the explicit parallel form.

Parallelism: An additional notion of cost that is important in comparing algorithms is the ***parallelism*** of an algorithm. Parallelism, sometimes called ***average parallelism***, is simply defined as the work over the span:

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

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

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

Suppose $n = 10{,}000$ and if $W(n) = \Theta(n^3) \approx 10^{12}$ and $S(n) = \Theta(n \lg n) \approx 10^5$ then $\mathbb{P}(n) \approx 10^7$, which is a lot of parallelism. But, if $W(n) = \Theta(n^2) \approx 10^8$ then $\mathbb{P}(n) \approx 10^3$, which is much less parallelism. The decrease in parallelism is not because the span was large, but because the work was reduced.

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

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

Designing parallel algorithms. In parallel-algorithm design, we aim to keep parallelism as high as possible but without increasing work. In general the goals in designing efficient algorithms are

1. first priority: to keep work as low as possible, and
2. second priority: keep parallelism as high as possible (and hence the span as low as possible).

In this book we will mostly cover work-efficient algorithms where the work is the same or close to the same as the best sequential time. Indeed this will be our goal throughout the course. Now among the algorithm that have the same work as the best sequential time we will try to achieve the greatest parallelism.

4.2.4 Scheduling

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

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

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

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

Greedy scheduling. We say that a scheduler is ***greedy*** if whenever there is a processor available and a task ready to execute, then it assigns the task to the processor and start running it immediately. Greedy schedulers have a very nice property that is summarized by the greedy scheduling principle.

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

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

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

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

Therefore we have

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

We can thus see that a greedy scheduler does reasonably close to the best possible. In particular $\frac{W}{P} + S$ is never more than twice $\max(\frac{W}{P}, S)$ and when $\frac{W}{P} \gg S$ the difference between the two is very small. Indeed we can rewrite equation 4.1 above in terms of the parallelism $\mathbb{P} = W/S$ as follows

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

Therefore as long as $\mathbb{P} \gg P$ (the parallelism is much greater than the number of processors) then we get near perfect speedup. Therefore $\mathbb{P}$ gives us a rough upper bound on the number of processors we can effectively use.

Example 4.16. As an example, consider the mergeSort algorithm for sorting a sequence of length n. The work is the same as the sequential time, which you might know is

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

We will show that the span for mergeSort is

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

The parallelism is therefore

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

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

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

Remark 4.17. No real scheduler is fully greedy. This is because there is overhead in scheduling the job and thus there will surely be some delay from when a subcomputation becomes ready until when it starts up. Such overheads can be thought as a form of ***scheduling friction***. In practice, the efficiency of a scheduler is therefore quite important to achieving good efficiency. In addition to friction, scheduling can cause memory effects: moving a subcomputation from one processor to another might lead to additional data movement. Because of friction and memory effects, the greedy scheduling principle should only be viewed as a rough estimate in much the same way that the RAM model or any other computational model should be just viewed as an estimate of real time.

4.3 Design Techniques

One of the most difficult and probably most important tasks for any practitioner of computer-science is designing their own algorithms. Beginners often feel that they do not even know where to start when designing an algorithm. In this section, we will briefly describe some of the key techniques used for algorithm design. These techniques are usually a good starting point when designing an algorithm. Even if a particular technique does not lead to an algorithm with the desired properties, it can still add to our understanding of the problem and lead us to a better algorithm. We note that our review of these techniques is not meant to be formal, but rather presented more as a guideline.

4.3.1 Brute Force

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

For example, to sort a set of keys, we can try all permutations of those keys and test each one to see if it is sorted. We are guaranteed to find at least one that is sorted. However, there are $n!$ permutations and $n!$ is very large even for modest n. For example, using Sterling's approximation, we know that $100! \approx \frac{100}{e}^{100} \geq 10^{100}$. There are only about 10^{80} atoms in the universe so there is no feasible way we could apply the brute force method directly. We thus conclude that this approach to solving the sorting problem is not "tractable" for large problems, but it might be a viable approach for small problems. In some cases the number of candidate solutions is much smaller.

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

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

Despite its inefficiency, there are two reasons for why brute-force algorithms are often helpful. First, the brute-force technique can be very useful is when testing the correctness of more efficient algorithms. Even if inefficient for large n the brute-force technique could work well for testing small inputs, especially because brute-force algorithms tend to be simple. The sec-

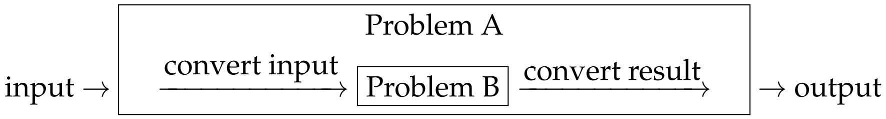


Figure 4.1: An illustration of a reduction from problem A to problem B.

ond reason is that brute-force algorithms are often easy to design and are a good starting point toward a more efficient algorithm.

Note that when solving ***optimization problems***, i.e., problems where we care about the optimal value of a solution, the problem definition might only return the optimal value and not the "underlying" solution that gives that value. For example, a shortest path problem on graphs might be defined such that it just returns the shortest distance between two vertices, but not a path that has that distance. Similarly when solving ***decision problems***, i.e., problems where we care about whether a solutions exists or not, the problem definition might only return a Boolean indicating whether a solution exists and not the solution. For both optimization and decision problems of this kind when we say try all "underlying solutions" we do not mean try all possible return values, which might just be a number or a Boolean, we mean try all possible solutions that lead to those values, e.g., all paths from a to b.

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

4.3.2 Reduction

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

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

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

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

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

4.3.3 Inductive Techniques

The idea behind inductive techniques is to solve one or more smaller instances of the same problem, typically referred to as ***subproblems***, to solve the original problem. The technique most often uses recursion to solve the subproblems. Common techniques that fall in this category include the following.

Divide and conquer. Divide the problem on size n into $k > 1$ independent subproblems on sizes $n_1, n_2, \ldots n_k$, solve the problem recursively on each, and combine the solutions to get the solution to the original problem. Figure 4.2 illustrates the approach for $k = 3$. Since the subproblems are independent, they can be solved in parallel.

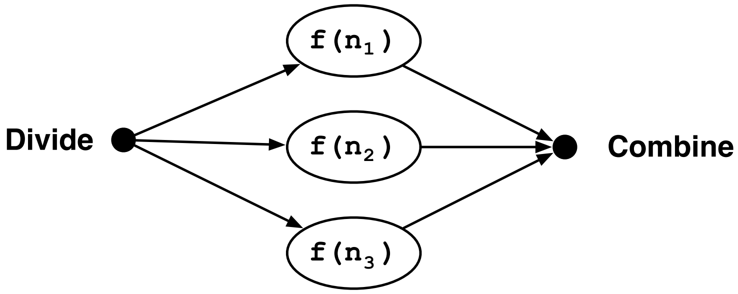


Figure 4.2: Structure of a divide-and-conquer algorithm illustrated ($k = 3$). The independent problems $f(n_1)$, $f(n_2)$, and $f(n_3)$ can be solved in parallel.

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

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

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

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

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

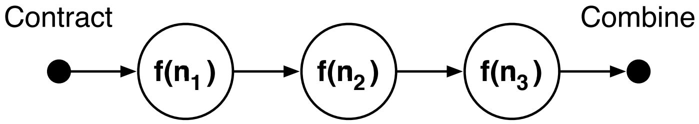


Figure 4.3: Structure of a contraction algorithm illustrated.

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

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

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

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

Contraction. For a problem of size n generate a significantly smaller (contracted) instance (e.g., of size $n/2$), solve the smaller instances recursively, and then use the results to solve the original problem by combining the results from the smaller instances. Contraction only differs from divide and conquer in that it allows there to be only one independent subproblem to be solved at a time, though there could be multiple dependent subproblems to be solved one after another (sequentially).

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

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

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

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

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

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

4.3.4 Randomization

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

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

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

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

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

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

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

Example 4.26. [Symmetry breaking] We often perform randomized symmetry breaking when walking on a crowded sidewalk with many people coming in our direction. With each person we encounter, we may pick a random direction to turn and the other person responds, or the other way. If we recognize each other late, we may end up in a situation where the randomness becomes more apparent, as we attempt to get past each other but make the same (really opposite) decisions. Since we make essentially random decisions, the symmetry is eventually broken—or we run into each other.

4.4 Cost Analysis with Recurrences

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

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

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

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

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

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

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

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

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

When stating recursions, we may use asymptotic notation to express certain terms such as the $O(n)$ in our example. How do you perform calculations with such terms? The trouble is that if you add any two $O(n)$ terms what you get is a $O(n)$ but you can't do that addition a non-constant many times and still have the result be $O(n)$. To prevent mistakes in calculations, we often replace such terms with a non-asymptotic term and do our calculations with that term. For example, we may replace $O(n)$ with n, $2n$, $2n + \lg n + 3$, $3n + 5$, or with something parametric

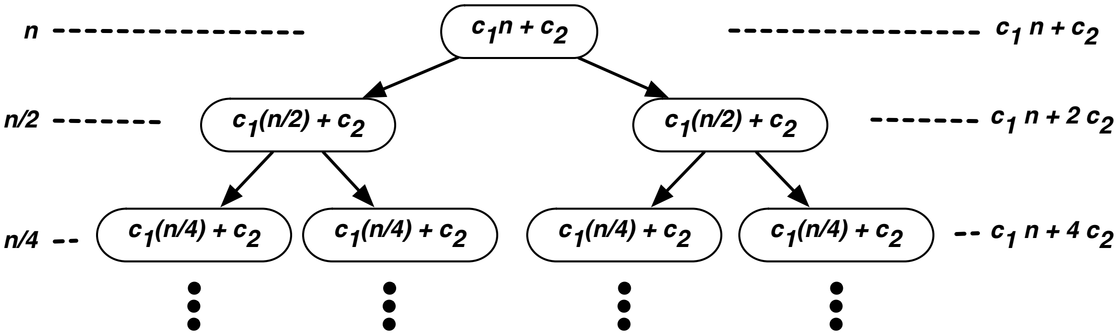


Figure 4.4: Recursion tree for the recursion $W(n) \leq 2W(n/2)+c_1m+c_2$. Each level is annotated with the problem size and the cost at that level.

such as $c_1n + c_2$ where c_1 and c_2 are constants. Such kinds of replacement may introduce some more imprecision to our calculations but again they usually don't matter as they change the closed-form solution by a constant factor.

The Tree Method. Using the recursion $W(n) = 2W(n/2) + O(n)$, we will review the tree method . Our goal is to derive a closed form solution to this recursion.

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

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

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

where c_1 and c_2 are constants. We now draw a tree to represent the recursion. Since there are two recursive calls, the tree is a binary tree, where each node has 2 children, whose input is half the size of the size of the parent node. We then annotate each node in the tree with its cost noting that if the problem has size m, then the cost, excluding that of the recursive calls, is at most $c_1 \cdot m + c_2$. Figure 4.4 shows the recursion tree annotated with costs.

To apply the tree method, there are some key questions we should ask ourselves to aid drawing out the recursion tree and to understand the cost associated with the nodes:

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

- What is the total cost across level i?

Our answers to these questions lead to the following analysis: We know that level i (the root is level $i = 0$) contains 2^i nodes, each costing at most $c_1(n/2^i) + c_2$. Thus, the total cost in level i is at most

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

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

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

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

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

The Brick Method, a Variant of the Tree Method. The tree method involves determining the depth of the tree, computing the cost at each level, and summing the cost across the levels. Usually we can easily figure out the depth of the tree and the cost of at each level relatively easily—but then, the hard part is taming the sum to get to the final answer.

It turns out that there is a special case in which the analysis becomes simpler: when the costs at each level grow geometrically, shrink geometrically, or stay approximately equal. By recognizing whether the recurrence conforms with one of these cases, we can almost immediately determine the asymptotic complexity of that recurrence.

The vital piece of information is the ratio of the cost between adjacent levels. Let L_i denote the total cost at level i of the recursion tree. Based on this, we check if L_i are consistent with one of the three cases and determine the bound as shown in Definition 4.27.

You might have seen the "master method" for solving recurrences in previous classes. We do not like to use it since it only works for special cases and does not give an intuition of what is going on. However, we will note that the three cases of the master method correspond to special cases of leaves dominated, balanced, and root dominated.

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

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

Definition 4.27. [Brick Method]

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

Leaves Dominated

Each level is larger than the level before it by at least a constant factor. That is, there is a constant $\rho > 1$ such that for all level i, $L_{i+1} \geq \rho \cdot L_i$. The total cost is $O(L_d)$.

```
   ++
  ++++
 ++++++
++++++++
```

Balanced

All levels have approximately the same cost. The total cost is $O(d \cdot \max_i L_i)$.

```
++++++++
++++++++
++++++++
++++++++
```

Root Dominated

Each level is smaller than the level before it by at least a constant factor. That is, there is a constant $\rho < 1$ such that for all level i, $L_{i+1} \leq \rho \cdot L_i$. The total cost is $O(L_0)$.

```
++++++++
 ++++++
  ++++
   ++
```

where c_1 and c_2 are constants.

Besides using the recursion tree method, can also arrive at the same answer by mathematical induction. If you want to go via this route (and you don't know the answer a priori), you'll need to guess the answer first and check it. This is often called the "substitution method." Since this technique relies on guessing an answer, you can sometimes fool yourself by giving a false proof. The following are some tips:

1. Spell out the constants. Do not use big-O—we need to be precise about constants, so big-O makes it super easy to fool ourselves.
2. Be careful that the induction goes in the right direction.
3. Add additional lower-order terms, if necessary, to make the induction go through.

Let's now redo the recurrences above using this method. Specifically, we'll prove the following theorem using (strong) induction on n.

Theorem 4.28. *Let a constant $k > 0$ be given. If $W(n) \leq 2W(n/2) + k \cdot n$ for $n > 1$ and $W(1) \leq k$ for $n \leq 1$, then we can find constants κ_1 and κ_2 such that*

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

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

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

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

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

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

4.5 What makes a good solution?

When you encounter an algorithms or a data-structures problem, you can look into your bag of techniques and, with practice, you will find a good solutions to the problem. When we say a ***good solution*** we mean:

1. A solution that is correct, and
2. a solution that has the lowest cost possible.

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

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

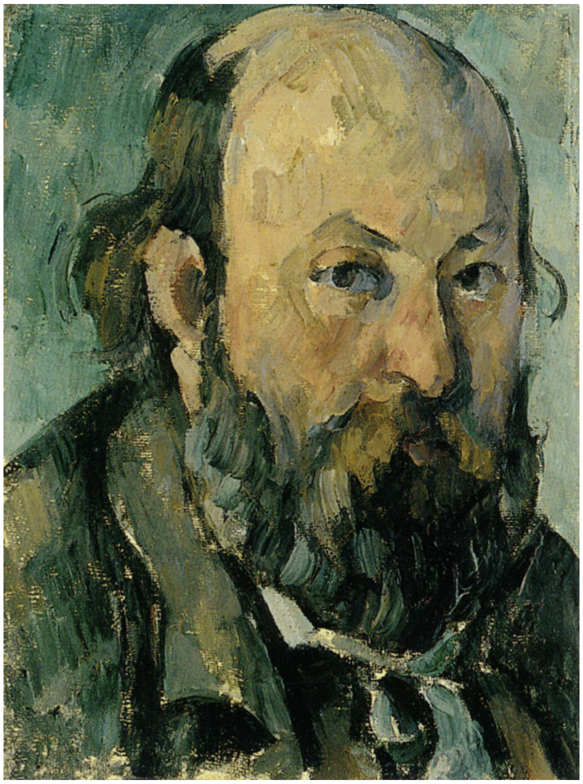


Figure 4.5: Abstraction is a powerful technique in computer science. One reason why is that it enables us to use our intelligence more effectively allowing us not to worry about all the details or the reality. Paul Cezanne noticed that all reality, as we call it, is constructed by our intellect. Thus he thought, I can paint in different ways, in ways that don't necessarily mimic vision, and the viewer can still create a reality. This allowed him to construct more interesting realities. He used abstract, geometric forms to architect reality. Can you see them in his self-portrait? Do you think that his self-portrait creates a reality that is much more three dimensional, with more volume, more tactile presence than a 2D painting that would mimic vision? Cubists such as Picasso and Braque took his ideas on abstraction to the next level.

4.6 Problems

4-1 Closed form solutions

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

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

4-2 Substitution Method
Show that $W(n) = 2W(n/2) + \log n \in O(n)$ by using tree of substitution method.

4-3 Bricks and trees
Derive the brick method from the tree method.

4-4 2-Optimality of Brent's Theorem
Prove that the bound given by Brent's theorem is within a factor two of optimal.

4-5 Order statistics by reduction to sorting
Suppose that you have an algorithm that can, for given a comparison function, comparison sort a sequence of numbers in $\Theta(n \log n)$ work and $\Theta(\log^2 n)$ span. Using the problem reduction technique:

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

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

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

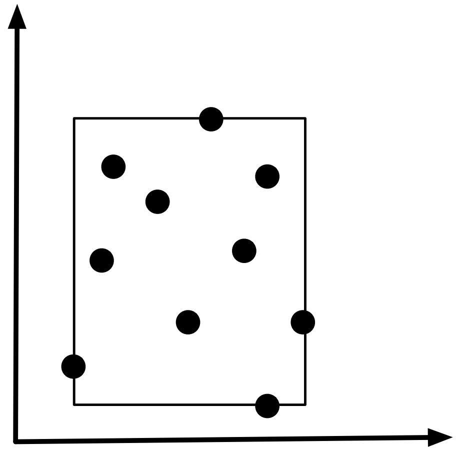
4-7 Graph connectivity
Suppose that you know how to solve the problem of determining whether there is a path from any two vertices in a given undirected graph as long as all the vertices in the graph have degree 3 or less. Using reduction, solve the problem of determining whether any two vertices are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

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

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

- Design a sorting algorithm by reduction to the convex hull problem.
- State the work and the span of your algorithms in terms of the work and the span of a convex hull algorithm.

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



4-11 Academic life

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

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

4-12 Greedy dining

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

Chapter 5

Example: Genome Sequencing

The human genome is the full nucleic acid sequence for humans; it consists 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". The 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; it is expected that its deeper understanding will lead to insights into the nature of life.

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 some of the algorithms behind the results—and the power of problem abstraction which will reveal surprising connections between seemingly unrelated problems. 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.

5.1 The Shotgun Method

What makes sequencing the genome hard 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 and then reassemble the pieces.

Primer walking. A technique called "primer walking" can be used to cut the DNA strands into consecutive fragments and sequence each one. The process is slow and sequential because the result of one fragment is used to construct (in a lab) the molecule needed to find the following fragment. One possible way to parallelize primer walking is to divide the genome into many fragments and sequence them all in parallel. The shortcoming of this approach is that we don't know how to put them together, because we have mixed up the fragments and lost their order. This approach can be viewed as making a jigsaw puzzle out a picture that we have not seen before; it can be difficult and even impossible to solve such a jigsaw puzzle.

Example 5.1. 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 are connected. If we had some additional information about how to connect them, however, we could imagine solving this problem just as we solve a jigsaw puzzle. One way to get additional information on joining the fragments is to make multiple copies of the original sequence and generate many fragments that overlap. Overlaps between fragments then be used to connect them. This is the idea behind the shotgun (sequencing) method, which was the primary method used to first sequence the human genome.

Example 5.2. For example, 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 reverse the cut.

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).
3. Sequence each of the short fragments, which can be performed in parallel.
4. Reconstruct the original genome from the fragments.

Steps 1–3 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.

5.2 Defining the Problem

Given that there might be a many solutions and that we may not always hope to find the actual genome, we wish to define a problem that makes precise the "best solution" that we can find.

It is not easy to define the solution precisely. This is why, it can be as difficult and important to formulate a problem as it is to solve it. But as we will see, we can come pretty close to a realistic solution.

Ockham chooses a razor

Figure 5.1: William of Occam (or Ockham, 1287-1347) posited that among competing hypotheses that explain some data, the one with the fewest assumptions or shortest description is the best one. The term "razor" apparently refers to shaving away unnecessary assumptions, although here is a more modern take on it.

To start with, note that since the fragments all come from the original genome, the result should at least contain all of them. In other words, it is a ***superstring*** of the fragments. There can, however, be multiple superstrings for any given set of fragments. Which one should we pick? How about the shortest superstring? This would give us the simplest explanation, which is often desirable. The principle of selecting the simplest explanation is often referred to as ***Occam's razor***.

Problem 5.3. [Shortest Superstring (SS) Problem] Given an alphabet set Σ and a set of finite-length strings $s \subseteq \Sigma^*$, return a shortest string r that contains every $x \in A$ as a substring of r.

In the definition the notation Σ^*, the "Kleene star", means the set of all possible non-empty strings consisting of characters Σ. Note that, for a string s to be a ***substring*** of another string r, s must as a contiguous block in r. That is, "ag" is a substring of "ggag" but is not a substring of "attg".

For genome sequencing, we have $\Sigma = \{a, c, g, t\}$. We have thus converted a vague problem, sequencing the genome, into a concrete problem, the SS problem. As suggested by the discussion thus far and further discussed at the end of this chapter, the SS problem might not be exactly the right abstraction for the application of sequencing the genome, but it is a good start.

Having specified the problem, we are ready to design an algorithm, in fact a few algorithms, for solving it.

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. We now consider three algorithmic techniques and apply them to this problem, deriving an algorithm from each.

5.3 Brute-Force Algorithm 1

As applied to the genome-sequencing problem, the brute-force technique involves trying all candidate superstrings of the input fragments and selecting the shortest one.

One idea is to consider all strings $x \in \Sigma^*$, and for each x to 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, however, is that there are an infinite number of strings in Σ^* so we cannot check them all. Fortunately, 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. 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.

5.4 Understanding the Structure of the Problem

To improve over the brute force-algorithm described above, let's take a closer look at the problem to make some observations.

Observation 1: Snippets. First, observe that we can ignore strings that are contained in other strings, because they don't contribute to the solution. For example, if we have gagtat, ag, and gt, we can throw out ag and gt. In the context of the genome sequencing problem, we will refer to the fragments that are not contained in others as ***snippets***.

Observation 2: Ordering of the snippets. Since no snippet can be contained in 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. In other words, we only need to consider permutations of the snippets.

Example 5.4. In our example, we had the following fragments.

$$\boxed{\text{catt}}\ \boxed{\text{ag}}\ \boxed{\text{gagtat}}$$
$$\boxed{\text{cat}}\ \boxed{\text{tagg}}\ \boxed{\text{ag}}\ \boxed{\text{tat}}$$
$$\boxed{\text{ca}}\ \boxed{\text{tta}}\ \boxed{\text{gga}}\ \boxed{\text{gtat}}$$

Our snippets are now:

$$S = \left\{\text{catt}, \text{gagtat}, \text{tagg}, \text{tta}, \text{gga}\right\}.$$

The other fragments $\{$cat, ag, tat, ca, 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.

In the following discussion we will use $n = |S|$, i.e. the number of snippets, and $m = \sum_{i=1}^{n} |S_i|$, i.e. the total size of the snippets.

5.5 Brute Force Algorithm 2

Based on our new understanding of the problem, let's consider a second brute force solution. In addition to requiring less computational work, this solution will also help us understand other possible solutions. In the last section, we concluded that we only have to consider permutations of the snippets. We can thus consider all permutations, and find the permutation that gives us the shortest superstring. The question remains of how, once we pick a permutation, we then find the shortest superstring for that permutation. For this purpose we will use the following theorem.

Theorem 5.5 (Removing Overlap)**.** *Given any start ordering of the snippets $s_1, s_2, \ldots, s_n$, removing the maximum overlap between each adjacent pair of snippets (s_i, s_{i+1}) gives the shortest superstring of the snippets for that permutation.*

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 $s_1, \ldots, s_i$ starting in that order. We refer to this superstring as r_i. We now prove that the theorem it 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 $s_0, \ldots, s_{i+1}$ because it includes r_i, which by induction is a superstring of $s_0, \ldots, s_i$, and because it includes s_{i+1}. It is also be the shortest since r_i is the shortest for $s_1, \ldots s_i$ 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. □

Example 5.6. For our running example, consider the following permutation

catt $\underline{\text{tt}}$a $\underline{\text{ta}}$gg $\underline{\text{gg}}$a $\underline{\text{ga}}$gtat

When the maximum overlaps are removed (the excised parts are underlined) we get cattaggagtat, which is indeed the shortest superstring for the given permutation. In this case, it is also the overall shortest superstring.

Since the shortest superstring naturally defines a total order on the snippets, we can compute the shortest superstring by trying all permutations of the snippets, finding the shortest superstring for that permutation by removing the overlaps, and pick the overall shortest. Let's make our algorithm more precise and calculate the work and span of this algorithm. First we need an algorithm for finding the maximum overlap between two strings s and t. To this end, we can again use the brute force-technique: consider each suffix of s and check if it is a prefix of t and select the longest such suffix that is also a prefix. The work of this algorithm is $|s| \cdot |t|$, i.e., the product of the lengths of the strings. The span for checking a particular suffix is also a prefix is $O(\lg t)$ by using a divide-and-conquer algorithm. Since we can try all positions in parallel, the span is thus $O(\lg |t|)$. The span for selecting the maximum over all matches is $O(\lg s)$. The total span is thus $O(\lg s + \lg t) = O(\lg s + t)$.

Now that we know how to find the overlap reasonably efficiently, we can try each permutation, and eliminate overlaps. This would, however, require repeatedly computing the overlap between many of the same snippets. Since we only remove the overlap between successive snippets, there are only $O(n^2)$ pairs that we have to consider. We can thus ***stage*** the algorithm into two stages.

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

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 $s = \{s_1, \ldots, s_n\}$. Let $m = \sum_{x \in S} |x|$. Using our algorithm, $\texttt{overlap}(x, y)$ for finding the maximum overlap between two strings x and y, we have

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

and since all pairs can be considered in parallel,

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

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. Therefore the cost for handling each permutation is $O(n)$ work and $O(\lg n)$ span. Unfortunately, there are $n!$ permutations to consider. 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 $100! \approx 10^{158}$ combinations, 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.

Unfortunately the SS problem turns out to be NP-hard, although we will not show this. When a problem is NP hard, it means that there are *instances* of the problem that are difficult to solve. NP-hardness doesn't rule out the possibility of algorithms that quickly compute near optimal answers or algorithms that perform well on 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.

For this particular 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. We will discuss such an algorithm in Section 5.7.

Remark 5.7. The technique of staging used above is a key technique in algorithm design as well as implementation. The basic idea is to identify a computation that is repeated many times in the future and pre-compute it, storing it in some fast lookup data structure. Later instances of that computation can then be recalled via lookup instead of re-computing every time it is needed.

5.6 Reduction to the Traveling-Salesperson Problem

Another technique in algorithm design is to reduce an algorithmic problem to another problem that we know how to solve. It is sometimes quite surprising that problems that seem very different can be reduced to each other. Here, we shall reduce the shortest superstring problem to another seemingly unrelated problem: the traveling salesperson (TSP) problem. The TSP problem is a canonical NP-hard 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. The asymmetric TSP problem 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. Recall that a cycle is a path in a graph that starts and ends at the same vertex and that a ***Hamiltonian cycle*** is a cycle that visits every vertex exactly once. The symmetric version of the problem can be viewed as considering only graphs where for each arc (u, v), there is also a reverse arc (v, u) with the same weight.

Problem 5.8. [The Asymmetric Traveling Salesperson (aTSP) Problem] 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 idea behind the reduction of the SS problem to the TSP problem comes from something that we have learned from our brute-force algorithm: the shortest superstring problem can be

HELP! WE'RE LOST!
HELP "CAR 54"...AND WIN CASH
54...$1,000 PRIZES
ONE...$10,000 GRAND PRIZE
START and FINISH
Lewiston
Portland
Butte
Boise
Twin Falls
Redding
Reno
Salt Lake City
Gustine
Lone Pine
Mexican Hat
Colorado Springs
Marble Canyon
Manuelito
Truth or Consequences
Blunt
Lincoln
Wichita
Kansas City
Amarillo
Little Rock
Baton Rouge
La Crosse
Chicago
Marion
Indianapolis
Erie
Carlisle
Wana
Wilkesboro
Chattanooga
Barnwell
Bainbridge
Map by Reed McNally
OFFICIAL RULES ON REVERSE SIDE
© PROCTER & GAMBLE 1962

Help Toody and Muldoon find the shortest round trip route to visit all 33 locations shown on the map.

All you do is draw connecting straight lines from location to location to show the shortest round trip route.

HERE'S THE CORRECT START...

Begin at Chicago, Illinois. From there, lines show correct route as far as Erie, Pennsylvania. Next, do you go to Carlisle, Pennsylvania or Wana, West Virginia? Check the easy instructions on back of this entry blank for details.

Figure 5.2: A poster from a contest run by Proctor and Gamble in 1962. The goal was to solve a 33 city instance of the TSP. Gerald Thompson, a Carnegie Mellon professor, was one of the winners.

solved by trying all permutations. In particular we shall make the TSP problem try all the permutations for us. For the reduction, we set up a graph so that each valid Hamiltonian cycle corresponds to a permutation. The graph is ***complete***: it contains an arc between any two vertices, guaranteeing the existence of a Hamiltonian cycle.

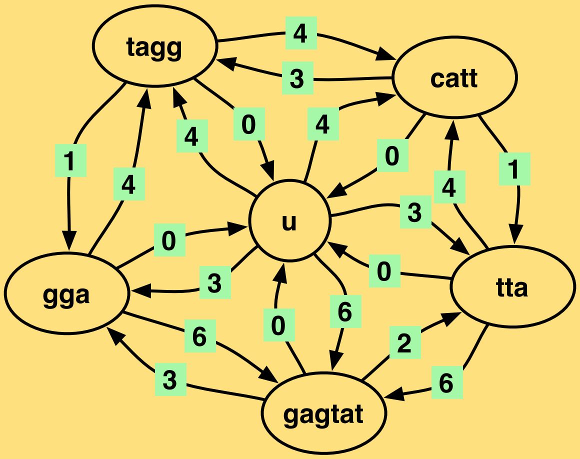
To specify the reduction, let $\texttt{overlap}(s_i, s_j)$ denote the maximum overlap for s_i followed by s_j. For example, for "tagg" and "gga", we have $\texttt{overlap}$ ("tagg","gga") $= 2$. Now we build a graph $D = (V, A)$.

- 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 $w_{i,j} = |s_j| - \texttt{overlap}(s_i, s_j)$. 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, |"gga"| $-$ $\texttt{overlap}$("tagg", "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 5.9. To see this reduction in action, the snippets in our running example, $\Big\{$catt, gagtat, tagg, tta, gga$\Big\}$ 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 the 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.

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.

Example 5.10. [Hardness of shortest superstring] We can also establish that the shortest superstring problem is NP-hard by using the same idea as in the reduction described above but by reversing the direction of the reduction and reducing TSP to the shortest superstring problem. To see this, note that for any cycle that starts at the source vertex, we have a corresponding permutation. The total weight of the arcs on such a cycle is exactly the length of the shortest superstring with overlaps removed. Thus, if we have an algorithm for finding the shortest superstring, then we can solve the TSP problem. Since constructing the graph only requires polynomial work this reduction is efficient. But we know that TSP problem is NP hard. We thus conclude that the shortest superstring problem is also NP hard.

Remark 5.11. 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, then B must be NP-hard (NP-complete).

5.7 Greedy Algorithm

We now consider our final algorithm-design technique, the "greedy" technique for solving the SS problem. When applying the greedy design technique, we consider the current solution at hand and make a greedy ***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.

The greedy technique is a heuristic that in some cases returns an optimal solution, but in many cases it does not. Nevertheless, greedy algorithms are popular because they tend to be simple. We note that, for a given problem there might be several greedy approaches that depend on the types of steps and 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.

The pseudocode for our algorithm is given in Algorithm 5.12. The algorithm relies on a function `overlap` (x, y) to compute the overlap of the strings x and y, and the function `join` (x, y), which places x after y and removes the maximum overlap. For example, `join` ("tagg", "gga") $=$ "tagga". 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

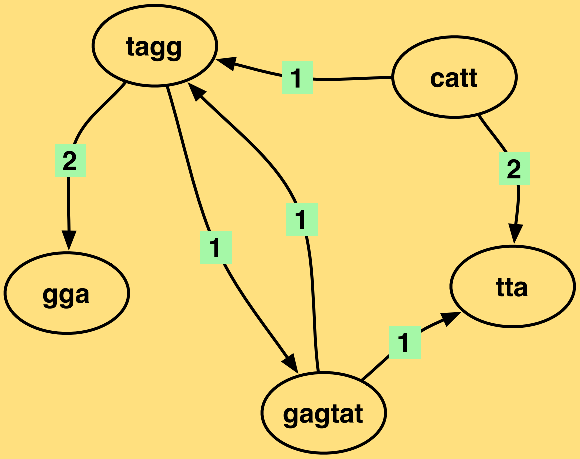
Algorithm 5.12. [Greedy Approximate SS]

```
greedyApproxSS $S$ =
  if $|S| = 1$ then
    return $x \in S$
  else
    let
      $T = \{ (\texttt{overlap}(x,y), x, y) \; : \; x \in S, y \in S, x \neq y \}$
      $(\_, x, y) = \arg\max_{(o,\_,\_)\in T} o$
      $z = \texttt{join}(x,y)$
      $S' = S \cup \{z\} \setminus \{x,y\}$
    in
      greedyApproxSS $S'$
    end
```

in S that have the maximum overlap. It does this by first calculating the overlap for all pairs (Line 6) and then picking the one of these that has the maximum overlap (Line 7). Note that T is a set of triples each corresponding to an overlap and the two strings that overlap. The notation $\arg\max_{(o,\_,\_)\in T} o$ 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 $z = \texttt{join}(x,y)$ 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.

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 5.13. Consider the snippets in our running example,
$\left\{ \text{catt, gagtat, tagg, tta, gga} \right\}$.
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,
- join catt and tta to obtain catta,
- join gagtat and tagga to obtain gagtatagga, and
- join gagtatagga and catta to obtain gagtataggacatta.

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 reduction. Let's analyze the work and the span of the greedy algorithm.

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 the $\arg\max$ for finding the maximum overlap can be computed in $O(m^2)$ work and $O(\lg m)$ span using a simple reduce. The other steps have less work and span. Therefore, not including the recursive call each call to `greedyApproxSS` costs 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 is therefore highly parallel. There are ways to make the algorithm more efficient, but leave that as an exercise to the reader.

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***.

Remark 5.14. 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 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. Don't fret: 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 generate strings up to the length of the repeats, and the double barreled method can put them together.

5.8 Problems

5-1 String concatenation
You are given a set of strings $s_1, \ldots, s_n$ and a target string t.

- Use the brute-force technique to come up with an algorithm that gives you a subset of the strings and the specific order in which they can be concatenated to obtain the target string.
- Analyze the work of your algorithm.
- Analyze the span of your algorithm.

5-2 Compact string concatenation
You are given a set of strings $s_1, \ldots, s_n$ and another target string t. Use the brute-force technique to come up with an algorithm that gives you a subset of the strings and the specific order in which they can be concatenated while removing overlaps to obtain the target string. For example if you concatenate ″parallel″ and ″elision″, you will get ″parallelision″. Analyze the work and the span of your algorithm.

5-3 Start positions of snippets
Consider the set of snippets for a genome sequence, and let s be a superstring of the snippets, such that each snippet is contained in the superstring as substring. Prove than no two snippets can start at the same position in s.

5-4 End positions of snippets
Consider the set of snippets for a genome sequence, and let s be a superstring of the snippets, such that each snippet is contained in the superstring as substring. Prove than no two snippets can end at the same position in s.

5-5 Substring check
Given a string s of length n, and a string t of length m: design an algorithm that checks whether s is a substring of t, and analyze the work and span of your algorithms.

5-6 Hamiltonian paths on complete graphs
Prove or disprove: a Hamiltonian path on a complete directed graph corresponds to a permutation of vertices. A permutation of vertices corresponds to a Hamiltonian path.

5-7 Hamiltonian cycle with start
Prove using the reduction technique that if the TSP problem in NP hard, then so is the problem of finding the shortest Hamiltonian cycle that starts and ends at a specified vertex.

5-8 Contained Strings
In the greedy algorithm `greedyApproxSS` (Algorithm 5.12), we remove x, y from the set of

strings but do not remove any strings from s that are contained within xy = join (x, y). Argue why there cannot be any such strings.

5-9 Correctness of greedy approximation
Prove that algorithm greedyApproxSS (Algorithm 5.12) returns a string that is a superstring of all original strings.

5-10 Exactness of greedy approximation
Give an example input for which greedyApproxSS (Algorithm 5.12) does not return the shortest superstring.

5-11 Improved greedy
Improve the greedy algorithm's cost bounds by presenting a more efficient implementation.

5-12 Generous grandmother
Your rich grandmother enjoys collecting precious items, such as jewelry and gold-plated souvenirs. When you visit her for Spring break (instead of going to Cancun), she is very pleased and decides to reward you. She gives you a bag and a scale and instructs you to take anything you want as long as the bag does not hold any more than 10 pounds. Delighted by the surprising (based on your parents' stories of their childhood) generosity of your grandmother, you also realize that your grandmother forgot to take the price tags off the items, which gives you an idea about the value of these items.

- Design a brute force algorithm for selecting the items to take away with you. Is your algorithm optimal?
- What is the work and span of your brute-force algorithm?
- Design a greedy algorithm for selecting the items to take away with you. Why is your algorithm greedy?
- Is your greedy algorithm optimal?
- What is the work and span of your greedy algorithm?

.

Chapter 6

Sequences

A sequence is an ordered set, i.e., is a collection of elements that are totally ordered. Computer scientists use sequence data structures such as arrays and lists to represent many different sorts of data. In this chapter, we specify a sequence abstract data type and consider several cost specifications for it.

6.1 Definitions

Mathematically, a sequence is an enumerated collection, or an ordered set. As with a set, a sequence has ***elements***. The ***length*** of the sequence is the number of elements in the sequence. A sequence can be finite or infinite; in this book, we only consider finite ones. As an ordered set, 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. We define sequences mathematically as a function, whose domain is natural numbers.

Definition 6.1. An α ***sequence*** is a mapping (function) from $\mathbb{N}$ to α with domain $\{0, \ldots, n-1\}$ for some $n \in \mathbb{N}$.

This mathematical definition might seem pedantic but it is useful for at least a couple reasons: it allows for a concise but yet precise definition of the semantics of the functions on sequences, and we will see, relating sequences to mappings creates a symmetry with the abstract data types such as tables or dictionaries for representing mappings. An important point to notice in the definition is that sequences are parametrized by the type (i.e., set of possible values) of their elements.

Example 6.2. Let $A = \{0, 1, 2, 3\}$ and $B = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. The relation

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

is a function from A to B with domain $\{0, 1, 3\}$ since each element only appears once on the left. It is, however, not a sequence since there is a gap in the domain.

The relation

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

from A to B is a sequence because its domain matches A. The first element of the sequence is $\mathtt{a}$ and thus has rank 0. The second element is $\mathtt{b}$ and has rank 1. The length of the sequence is 4.

Mathematicians use a special notation for writing sequences. We shall also use such a notation, writing for example the sequence $\{(1, \mathtt{b}), (3, \mathtt{a}), (2, \mathtt{a}), (0, \mathtt{a})\}$ as $\langle\, \mathtt{a}, \mathtt{b}, \mathtt{a}, \mathtt{a} \,\rangle$.

Syntax 6.3. [Sequences and Indexing] The notation $\langle\, a_0, a_1, \ldots, a_{n-1} \,\rangle$ is shorthand for the sequence $\{(0, a_0), (1, a_1), \ldots, ((n-1), a_{n-1})\}$. For a sequence a, we write $a[i]$ to refer to the element of a at position i and $a[l, \ldots, h]$ to refer to the subsequence of a restricted to the position between l and h.

Since they occur frequently, 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 double-quote syntax for them, e.g., "$c_0 c_1 c_2 \ldots c_{n-1}$" is a string consisting of the n characters $c_0 \ldots c_{n-1}$.

Example 6.4. Some example sequences follow.

- Consider the integer sequence (or $\mathbb{Z}$ sequence)
  $a = \langle\, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 \,\rangle$.
  We have $a[0] = 2$, $a[2] = 5$, and $a[1, \ldots, 4] = \langle\, 3, 5, 7, 11 \,\rangle$.
- A character sequence, or a string: $\langle\, \texttt{'s'}, \texttt{'e'}, \texttt{'q'} \,\rangle \equiv \texttt{"seq"}$.
- An (integer ** string) sequence: $\langle\, (10, \texttt{"ten"}), (1, \texttt{"one"}), (2, \texttt{"two"}) \,\rangle$.
- A (string sequence) sequence: $\langle\, \langle\, \texttt{"a"} \,\rangle, \langle\, \texttt{"nested"}, \texttt{"sequence"} \,\rangle \,\rangle$.
- A function sequence, or more specifically a ($\mathbb{Z} \to \mathbb{Z}$) sequence:

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

6.2 The Sequence Abstract Data Type

ADT 6.5 shows the interface of an abstract data type for sequences. Sequences are one of the most prevalent ADT's used in this book, and more generally in computing. For economy in writing, communication, and cognition, we adopt a small extension of the standard mathematical notation on sequences, called ***sequence comprehensions***. Syntax 6.6 shows this notation on sequences. In the rest of this section, we describe the semantics of sequences and the comprehension-based syntax. Throughout, we use e and its variants to for SPARC expression. When specifying the semantics of operation, we rely on the mathematical definition of sequences as a partial function whose domain is natural numbers.

The sequence ADT can be broadly divided into several categories.

- Operations such as `nth` and `length` that return an element of the sequence or a particular property of it.
- Constructors that create sequences such as `empty`, `singleton`, and `tabulate`.
- Operations such as `map`, `filter` that operate on each element of a sequence independently in parallel.
- Operations such as `append` and `flatten` that operate on sequences as a whole.
- Operations such as `update` and `inject` that updates the elements of a sequence.
- Operations such as `iterate`, `reduce`, and `scan` that ***aggregate*** information over the elements of the sequence.

Abstract Data Type 6.5. [Sequences] For a value type α, the ***sequence data type*** is the type $\mathbb{S}_\alpha$ consisting of the set of all α sequences, and the following values and functions on $\mathbb{S}_\alpha$. In the specifiction $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ and $\mathcal{O} = \{less, greater, equal\}$.

`length`	:	$\mathbb{S}_\alpha \to \mathbb{N}$
`nth`	:	$\mathbb{S}_\alpha \to \mathbb{N} \to \alpha$
`empty`	:	$\mathbb{S}_\alpha$
`singleton`	:	$\alpha \to \mathbb{S}_\alpha$
`tabulate`	:	$(\mathbb{N} \to \alpha) \to \mathbb{N} \to \mathbb{S}_\alpha$
`map`	:	$(\alpha \to \beta) \to \mathbb{S}_\alpha \to \mathbb{S}_\beta$
`subseq`	:	$\mathbb{S}_\alpha \to \mathbb{N} \to \mathbb{N} \to \mathbb{S}_\alpha$
`append`	:	$\mathbb{S}_\alpha \to \mathbb{S}_\alpha \to \mathbb{S}_\alpha$
`filter`	:	$(\alpha \to \mathbb{B}) \to \mathbb{S}_\alpha \to \mathbb{S}_\alpha$
`flatten`	:	$\mathbb{S}_{\mathbb{S}_\alpha} \to \mathbb{S}_\alpha$
`update`	:	$\mathbb{S}_\alpha \to (\mathbb{N} \times \alpha) \to \mathbb{S}_\alpha$
`inject`	:	$\mathbb{S}_\alpha \to \mathbb{S}_{\mathbb{N} \times \alpha} \to \mathbb{S}_\alpha$
`isEmpty`	:	$\mathbb{S}_\alpha \to \mathbb{B}$
`isSingleton`	:	$\mathbb{S}_\alpha \to \mathbb{B}$
`iterate`	:	$(\alpha \times \beta \to \alpha) \to \alpha \to \mathbb{S}_\beta \to \alpha$
`reduce`	:	$(\alpha \times \alpha \to \alpha) \to \alpha \to \mathbb{S}_\alpha \to \alpha$
`scan`	:	$(\alpha \times \alpha \to \alpha) \to \alpha \to \mathbb{S}_\alpha \to (\mathbb{S}_\alpha \times \alpha)$
`collect`	:	$(\alpha \times \alpha \to \mathcal{O}) \to \mathbb{S}_{\alpha \times \beta} \to \mathbb{S}_{\alpha \times \mathbb{S}_\beta}$

Length and indexing. Given a sequence a, `length` a, also written $|a|$, returns the length of a. The function `nth` returns the element of a sequence at a specified index, e.g. `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.

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., `singleton` 1 evaluates to $\langle\, 1 \,\rangle$.

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

$$\texttt{tabulate}\;(f : (\mathbb{N} \to \alpha))\;(n : \mathbb{N}) : \mathbb{S}_\alpha = \langle\, f(0), f(1), \ldots, f(n-1) \,\rangle\,.$$

Syntax 6.6. [Syntax for Sequences] The table below defines the syntax for the sequence ADT used throughout this book . 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 is a SPARC expression whose value is a sequence, p is a SPARC pattern that binds one or more variables.

$\lvert e_s \rvert$	$\equiv$	`length` (e_s)
$e_s[i]$	$\equiv$	`nth` (e_s)
$\langle\,\rangle$	$\equiv$	`empty`
$\langle\, e \,\rangle$	$\equiv$	`singleton` (e)
$\langle\, e : 0 \leq i < e_n \,\rangle$	$\equiv$	`tabulate` $(\lambda\, i \,.\, e)\; e_n$
$\langle\, e : p \in e_s \,\rangle$	$\equiv$	`map` $(\lambda\, p \,.\, e)\; e_s$
$\langle\, x \in e_s \mid e \,\rangle$	$\equiv$	`filter` $(\lambda\, x \,.\, e)\; e_s$
$A[e_l, \cdots, e'_n]$	$\equiv$	`subseq` $(A, e_l, e'_n - e_l + 1)$
$e_s \mathbin{++} e'_s$	$\equiv$	`append` $e_s\; e'_s$

We use the following syntax for `tabulate` operation

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

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, as in:

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

Map. A common operation on sequences is to apply some computation to each element of a sequence. For example we might want to add five to each element of a sequence. For this purpose, we can use the operation `map`, which 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. We can specify the behavior of `map` as follows

$$\texttt{map}\; (f : \alpha \to \beta)\; (a : \mathbb{S}_\alpha) : \mathbb{S}_\beta = \{(i, f\; x) : (i, x) \in a\}$$

or equivalently as

$$\texttt{map}\; (f : \alpha \to \beta)\; \langle\, a_1, \ldots, a_{n-1} \,\rangle : \mathbb{S}_\alpha) : \mathbb{S}_\beta = \langle\, f(a_1), \ldots, f(a_{n-1}) \,\rangle\,.$$

As with `tabulate`, in `map`, the function f can be applied to all the elements of the sequence in parallel. As we will see in the cost model, this means the span of the function is the maximum of the spans of the function applied at each location, instead of the sum.

We use the following syntax for the `map` function

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

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

Filter. To filter out elements from a given sequence, we can use the function `filter`. The function takes a Boolean function f and a sequence a as arguments and applies f to each element of a, returning the sequence consisting exactly of those elements of $s \in a$ for which $f(s)$ returns true, and maintaining the order of the elements returned. We can specify the behavior of `filter` as follows

$$\texttt{filter}\ (f : \alpha \to \mathbb{B})\ (a : \mathbb{S}_\alpha) : \mathbb{S}_\alpha = \{(|\{(j, y) \in a \mid j < i \wedge f\ y\}|, x) : (i, x) \in a \mid f\ x\}.$$

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

We use the following syntax for the `filter` function

$$\langle\, x \in e_s \mid e \,\rangle \equiv \texttt{filter}\ (\lambda\, x\,.\, e)\ e_s,$$

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 use them together, as in:

$$\langle\, e : x \in e_s \mid e_f \,\rangle \equiv \texttt{map}\ (\lambda\, x\,.\, e)\ (\texttt{filter}\ (\lambda\, x\,.\, e_f)\ e_s).$$

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 6.7. Given the function `fib` i, which returns the i^{th} Fibonacci number, the expression:

$$a = \langle\, \texttt{fib}\, i : 0 \le i < 9 \,\rangle$$

is equivalent to

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

When evaluated, it returns the sequence

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

The expression

$$\langle\, x^2 : x \in a \,\rangle$$

is equivalent to

$$\texttt{map}\ (\lambda\, x \,.\, x^2)\ a.$$

When evaluated it returns the sequence:

$$\langle\, 0, 1, 1, 4, 25, 64, 169, 441, 1156 \,\rangle\,.$$

Given the function `isPrime` x which checks if x is prime, the expression

$$\langle\, x \;\; \texttt{:} \;\; x \in a \;\mid\; \texttt{isPrime} \;\; x \,\rangle$$

is equivalent to

$$\texttt{filter}\ \texttt{isPrime}\ a.$$

When evaluated, it returns the sequence $\langle\, 2, 5, 13 \,\rangle$.

Subsequences. The `subseq`(a, i, j) function extracts a contiguous subsequence 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

$$\texttt{subseq}\ (a : \mathbb{S}_\alpha)\ (i : \mathbb{N})\ (j : \mathbb{N}) : \mathbb{S}_\alpha = \{(k - i, x) : (k, x) \in a \mid i \le k < i + j\}.$$

We use the following syntax for denoting subsequences

$$a[e_i, \ldots, e_j] \equiv \texttt{subseq}(a, e_i, e_j - e_i + 1).$$

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 functions such as splitHead, splitMid, take, and drop for these purposes. Since all of these are trivially expressible in terms of subsequences, we omit their discussion for simplicity.

Append and flatten. For constructing large sequences from smaller ones, the sequence ADT provides the functions append and flatten. The function append (a, b) appends the sequence b after the sequence a. More precisely, we can specify append as follows

$$\texttt{append}\,(a : \mathbb{S}_\alpha)\ (b : \mathbb{S}_\alpha) : \mathbb{S}_\alpha = a \cup \{(i + |a|, x) : (i, x) \in b\}$$

We write $a \mathbin{++} b$ as a short form for append a b.

To append more than two sequences the flatten a function takes a sequence of sequences and flattens them. If the input is a sequence $a = \langle\, a_1, a_2, \ldots, a_n \,\rangle$ it appends all the a_i's. We can specify flatten more precisely as follows

$$\texttt{flatten}\,(a : \mathbb{S}_{\mathbb{S}_\alpha}) : \mathbb{S}_\alpha \qquad = \{(i + \sum_{(k,c)\in a, k<j} |c|, x) \ \ : (i, x) \in b, (j, b) \in a\}.$$

Example 6.8. The append operation $\langle\, 1, 2, 3 \,\rangle \mathbin{++} \langle\, 4, 5 \,\rangle$ yields the sequence $\langle\, 1, 2, 3, 4, 5 \,\rangle$.

The flatten operation flatten $\langle\, \langle\, 1, 2, 3 \,\rangle, \langle\, 4 \,\rangle, \langle\, 5, 6 \,\rangle \,\rangle$ yields the sequence $\langle\, 1, 2, 3, 4, 5, 6 \,\rangle$.

Updates and injections. 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 can specify update as follows

$$\begin{array}{ll} \texttt{update} & (a : \mathbb{S}_\alpha)\ (i : \mathbb{N}, x : \alpha) : \mathbb{S}_\alpha \\ & = \begin{cases} \{(j, y) : (j, y) \in a \mid j \neq i\} \cup \{(i, x)\} & \text{if } 0 \leq i < |a| \\ a & \text{otherwise.} \end{cases} \end{array}$$

To update multiple positions at once, we can use inject. The function inject(a, b) takes a sequence b of location-value pairs and updates each location with its associated value. If any locations are out of range, that pair does nothing. If multiple locations are the same, one of the updates take effect.

In the case of duplicates in the update sequence b, i.e., multiple updates to the same position, we leave it unspecified which update takes effect. The operation `inject` may thus treat duplicate updates non-deterministically.

Example 6.9. Given the string sequence

$$a = \langle \texttt{"the"}, \texttt{"cat"}, \texttt{"in"}, \texttt{"the"}, \texttt{"hat"} \rangle,$$

$$\texttt{update}\ a\ (1, \texttt{"rabbit"})$$

magically yields

$$\langle \texttt{"the"}, \texttt{"rabbit"}, \texttt{"in"}, \texttt{"the"}, \texttt{"hat"} \rangle$$

since location 1 is updated with `"rabbit"`. The expression

$$\texttt{inject}\ a\ \langle (4, \texttt{"log"}), (1, \texttt{"dog"}), (6, \texttt{"hog"}), (4, \texttt{"bog"}), (0, \texttt{"a"}) \rangle$$

could yield

$$\langle \texttt{"a"}, \texttt{"dog"}, \texttt{"in"}, \texttt{"the"}, \texttt{"bog"} \rangle$$

since location 0 is updated with `"a"`, location 1 with `"dog"`, and location 4 with `"bog"`. It could also yield

$$\langle \texttt{"a"}, \texttt{"dog"}, \texttt{"in"}, \texttt{"the"}, \texttt{"log"} \rangle$$

The entry with location 6 is ignored since it is out of range for a.

Collect. The primitive `collect` is useful when elements of a sequence are "keyed", making it possible to associate data with some key. Such pairs consisting of a key and a value are sometimes called ***key-value*** pairs. Given a sequence of key-value pairs, we might want to ***collect*** together all the values for a given key. Collecting values together based on a key is very common in processing databases. In relational database languages such as SQL it is referred to as "Group by". More generally it has many applications.

We will use the function `collect` for this purpose, and it is part of the sequence library. Its type signature is

$$\texttt{collect} : (cmp : \alpha \times \alpha \to \mathcal{O}) \to (a : \mathbb{S}_{\alpha \times \beta}) \to \mathbb{S}_{\alpha \times \mathbb{S}_\beta}.$$

Here the ”order set” $\mathcal{O} = \{\texttt{less}, \texttt{equal}, \texttt{greater}\}$.

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 6.10. The following sequence shows a sequence of key-value pairs consisting of our students from last semester and the classes they take.

$$\begin{array}{lll} kv & = \langle & (\texttt{"jack"}, \texttt{"15210"}), (\texttt{"jack"}, \texttt{"15213"}) \\ & & (\texttt{"mary"}, \texttt{"15210"}), (\texttt{"mary"}, \texttt{"15213"}), (\texttt{"mary"}, \texttt{"15251"}), \\ & & (\texttt{"peter"}, \texttt{"15150"}), (\texttt{"peter"}, \texttt{"15251"}), \\ & & \ldots\rangle. \end{array}$$

We can determine the classes taken by each student by using `collect`˜*cmp*, where *cmp* is a comparison function for strings

$$\begin{array}{lll} \texttt{collect}\ cmp\ kv & = \langle & (\texttt{"jack"}, \langle \texttt{"15210"}, \texttt{"jack"}, \texttt{"15213"}, \ldots\rangle) \\ & & (\texttt{"mary"}, \langle \texttt{"15210"}, \texttt{"15213"}), \texttt{"15251"}, \ldots\rangle), \\ & & (\texttt{"peter"}, \langle \texttt{"15150"}, \texttt{"15251"}, \ldots\rangle), \\ & & \ldots\rangle. \end{array}$$

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.

Checking for empty and singularity sequences. Many algorithms need to distinguish trivial sequences such as empty sequences and singular sequences, which contain only one element, from other non-trivial sequences. The functions `isEmpty` and `isSingular` can be used to check for this purpose; the functions return respectively return `true` if the sequence is empty or singular and return `false` otherwise.

Aggregation with `iterate`. Iteration is a key concept in computing, and specifically in algorithm design. Iteration involves a sequence of steps, taken one after another, where each step transforms the state from the previous step. Iteration is therefore an inherently sequential process. The function `iterate` can be used to create a computation that iterates over a sequence while accumulating information. It starts with an initial state and a sequence, and on each step updates the state based on the next element of the sequence. The function `iterate` has the type signature

$$\texttt{iterate}\ (f\texttt{:}\ \alpha \times \beta \to \alpha)\ (x\texttt{:}\ \alpha)\ (a\texttt{:}\ \mathbb{S}_\beta)\ \texttt{:}\ \alpha$$

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.

We can define the semantics of `iterate` inductively as follows.

$$\texttt{iterate}\ f\ x\ a = \begin{cases} x & \text{if } |a| = 0 \\ f(x, a[0]) & \text{if } |a| = 1 \\ \texttt{iterate}\ f\ (f(x, a[0]))(a[1, \ldots, |a|-1]) & \text{otherwise.} \end{cases}$$

The function `iterate` computes its final result by computing a new state for each element of the function

$$\begin{aligned} x_0 &= x \\ x_1 &= f(x_0, a[0]) \\ x_2 &= f(x_1, a[1]) \\ &\vdots \\ x_n &= f(x_{n-1}, a[n-1]), \end{aligned}$$

where $n = |a|$ and the final result is x_n.

We can similarly define a variant, `iteratePrefixes`, which takes the same arguments but returns all the intermediate values computed as a sequence, i.e., $\langle\, x_0, x_1, \ldots, x_{n-1}\,\rangle$.

As an application of iteration, consider the problem of finding whether a string (sequence) of left and right parentheses is properly matched or nested. We say a string is matched if it can be described recursively as

$$p = \langle\,\rangle \mid p\,p \mid \texttt{'('}\ p\ \texttt{')'},$$

where $\langle\,\rangle$ is the empty sequence, $p\ p$ indicates appending two strings of matched parentheses (recursively defined), and $\texttt{'('}\ p\ \texttt{')'}$ indicates the string starting with `'('` followed by a matched string p followed by `')'`.

Problem 6.11. [Parentheses matching] The *parentheses matching* problem requires determining whether a given a string of parentheses is matched.

There are a several algorithms for solving this problem. Here we consider a linear-work sequential algorithm based on `iterate`. In Chapter 8, we present a divide-and-conquer algorithm that requires no more work asymptotically, but has a low span. Using iteration, we can solve the problem by starting at the beginning of the sequence with a counter set to zero and iterating through the elements one by one. If we ever see a left parenthesis we increment the counter and whenever we see a right parenthesis we decrement the counter. A sequence of parentheses can only be matched if the count ends at 0 since being matched requires that there are an equal number of right and left parentheses. However ending with a count of 0 is not adequate since the string `"))(("` has count 0 but is obviously not matched. It also has to be the case that the count can never go below 0 during the iterations. This observation leads to the following algorithm.

Algorithm 6.12.

```
matchParens $a$ =
let
  count $(s, x)$ =
    case $(s, x)$
    | (None, _) => None
    | (Some $n$, ')') => if $(n = 0)$ then None else Some $(n - 1)$
    | (Some $n$, '(') => Some $(n + 1)$
in
   (iterate count (Some 0) $a$) = Some 0
end
```

The algorithm starts with the state `Some` 0 and increments or decrements the counter on a left and right parenthesis, respectively. If the iterations ever encounter a right parenthesis when the count is zero, this indicates the count will go below zero, and at this point the state is changed to `None`, which is propagated through the rest of the iterations to the result. Therefore at the end if the state is `Some` 0 then the counter never went below zero and ended up at zero so the parentheses must be matched.

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 use the associativity of the addition operations to sum up the elements of a sequence in parallel.

Aggregation with `reduce`. Reduction is a key concept in the design of parallel algorithms. 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.

Definition 6.13. A function $f : \alpha \to \alpha$ is associative if $f(f(x, y), z) = f(x, f(y, z))$ for all x, y and z of type α.

Associativity implies that when applying f to some values, the order in which the applications are performed does not matter. Note that associativity does not mean that you can reorder the arguments to a function (that would be commutativity).

Many functions are associative. For example, 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. An important class of operations that are not associative is floating-point operations. These operations are typically not associative because performing a set of operations in different orders can lead to different results because of loss of precision.

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

$$\texttt{reduce}\ (f\texttt{:}\ \alpha \times \alpha \to \alpha)\ (id\texttt{:}\ \alpha)\ (a : \mathbb{S}_\alpha)\ \texttt{:}\ \alpha$$

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 in equal to iteration. We can define the behavior of `reduce` inductively as follows

$$\texttt{reduce}\ f\ id\ a = \begin{cases} id & \text{if } |a| = 0 \\ a[0] & \text{if } |a| = 1 \\ f\Big(\texttt{reduce}\ f\ id\ (a[0,\ldots,\lfloor\frac{|a|}{2}\rfloor - 1]), & \\ \quad \texttt{reduce}\ f\ id\ (a[\lfloor\frac{|a|}{2}\rfloor,\ldots,|a|-1]\Big) & \text{otherwise.} \end{cases}$$

The function `reduce` is more restrictive than `iterate` since 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

$$\texttt{reduce}\ f\ id\ a = \texttt{iterate}\ f\ id\ a.$$

Example 6.14. The expression
`reduce append` $\langle\,\rangle$ $\langle$`"another","way","to","flatten"`$\rangle$
evaluates to
`"anotherwaytoflatten"`.

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 describe in Section 6.4, the span of `iterate` is linear in the size of the input, whereas the span of `reduce` is logarithmic.

The results of `reduce` and `iterate` may differ when the combining function is non-associative. In this case, the order in which the reduction is performed determines the result; because the function is non-associative, different orderings may lead to different answers. To deal properly with functions that are non-associative, the specification above therefore 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. Note that 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.

Aggregation with scan. 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. For example, in our parenthesis matching algorithm (Algorithm 6.12), we used this property crucially by defining our function to propagate a mismatched parenthesis forward in the computation. We now describe a function called `scan` that allows using the results of intermediate computations and also does so in parallel.

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

$$\texttt{scan}\ (f\texttt{:}\ \alpha\ \times\ \alpha\ \rightarrow\ \alpha)\ (id\texttt{:}\ \alpha)\ (a : \mathbb{S}_\alpha)\ \texttt{:}\ (\mathbb{S}_\alpha\ \times\ \alpha)\texttt{,}$$

where f is an associative function, a is the sequence, and id is the left identity element of f. The expression `scan`$f\,a$ returns the "sum" with respect to f of all prefixes of the input sequence a. For this reason, the `scan` operation is sometimes called the ***prefix sums*** operation. We can define `scan` inductively as follows

$$\texttt{scan}\ f\ id\ a = (\langle\, \texttt{reduce}\ f\ id\ a[0\ldots i] : 0 \le i < |a|\,\rangle\texttt{,}\ \texttt{reduce}\ f\ id\ a).$$

Note that 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") for this purpose.

Example 6.15. 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` $+\ 0\ \langle\, 0, 1, 2\,\rangle$ can be written as

$$
\begin{array}{lllll}
\texttt{scan}\ +0\ \langle\, 0,1,2\,\rangle & = & (& \langle & \texttt{reduce}\ +0\ \langle\,\rangle\,, \\
 & & & & \texttt{reduce}\ +0\ \langle\, 0\,\rangle\,, \\
 & & & & \texttt{reduce}\ +0\ \langle\, 0,1\,\rangle \\
 & & & \rangle\,, & \\
 & & & & \texttt{reduce}\ +0\ \langle\, 0,1,2\,\rangle \\
 & &) & & \\
 & = & & & (\langle\, 0,0,1\,\rangle\,,3).
\end{array}
$$

The computation `scanI` $+\ 0\ \langle\, 0, 1, 2\,\rangle$ can be written as

$$
\begin{array}{llll}
\texttt{scanI}\ +0\ \langle\, 0,1,2\,\rangle & = & \langle & \texttt{reduce}\ +0\ \langle\, 0\,\rangle\,, \\
 & & & \texttt{reduce}\ +0\ \langle\, 0,1\,\rangle\,, \\
 & & & \texttt{reduce}\ +0\ \langle\, 0,1,2,\,\rangle \\
 & & \rangle & \\
 & = & & \langle\, 0,1,3\,\rangle\,.
\end{array}
$$

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 6.16. [Copy scan] Scan is useful when we want pass information along the sequence. For example, suppose you have some "marked" elements that you would like to copy across to their right until they reach another marked element. For example, suppose that we are given a sequence of type $\mathbb{S}_{\mathbb{N}}$ consisting only of natural numbers and asked to return a sequence of the same length where each element receives the previous positive value. For the example, for input $\langle\, 0,\ 7,\ 0,\ 0,\ 3,\ 0\,\rangle$, the result should be $\langle\, 0,\ 0,\ 7,\ 7,\ 7,\ 3\,\rangle$.

Using a sequential loop or `iterate` would be easy. To solve this problem using `scan` we need a combining function f. Consider the function

$$\texttt{skipZero}\ (x, y) \;=\; \textbf{if}\ \ y > 0\ \ \textbf{then}\ \ y\ \ \textbf{else}\ \ x\,.$$

The function passes its right argument if it is positive, otherwise it passes on the left argument.

To be used in a scan it needs to be associative. In particular we need to show that for all x, y and z, we have

$$\texttt{skipZero}(x, \texttt{skipZero}(y, z)) = \texttt{skipZero}(\texttt{skipZero}(x, y), z).$$

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 that either ordering returns z. For the cases that $z = 0$ and y is positive, it is likewise easy to verify that both orderings give y. Finally, for the cases that both $y = z = 0$ and x is positive they both return x, and for all being zero, the ordering returns zero.

To use `skipZero` as part of the scan operation, we need to find its left identity. We can see that for any natural number y

$$\texttt{skipZero}\ (0, y) \;=\; y\,,$$

and that for any natural number x

$$\texttt{skipZero}\ (x, 0) \;=\; x\,.$$

Thus 0 is the left identity for `skipZero`.

Remark 6.17. 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 operations allow using two important algorithm-design techniques: `reduce` operation allows expressing divide-and-conquer algorithms and the `scan` operation allows expressing a iterative algorithms.

6.3 Sequence Comprehensions

Notation such as $\{x^2 : x \in a \mid \texttt{isPrime}\ a\}$ in which one set is defined in terms of the elements of other sets, and conditions on them is referred to as a ***set comprehensions***. The example can be read as: the set of squares of the primes in the set a. Comprehensions are commonly used in mathematics, because of the economy of expression and "comprehension" that they offer. In this book we use the comprehension syntax for a similar reason: they allow expressing algorithms in clear, concise notation. For example, the syntax for sequences shown in Syntax 6.6, as well as sets which will see later in Chapter 13, are based on set comprehensions.

In the examples shown thus far, we have mostly used sequences in "flat" fashion, without nesting sequence primitives within each other. In this section, we consider more uses of sequence comprehensions that involve nesting. As our first example, suppose that we wish to create a sequence consisting of points in two dimensional space (x, y) whose coordinates are natural numbers that satisfy the conditions that $0 \leq x \leq n-1$ and $1 \leq y \leq n$. For example, for $n = 3$, we would like to construct the sequence $\langle\, (0,1),(0,2),(1,1),(1,2),(2,1),(2,2)\,\rangle$. The code below shows one way to generate such a sequence.

```
points2D n =
  flatten (tabulate (λx.tabulate (λy.(x,y+1)) n) n)
```

The algorithm first generates a sequence of the form

$$
\begin{array}{ll}
\langle & \langle\,(0,1),(0,2),\ldots,(0,n)\,\rangle \\
& \langle\,(1,1),(1,2),\ldots,(1,n)\,\rangle \\
& \vdots \\
& \langle\,(n-1,1),(n-1,2),\ldots,(n-1,n)\,\rangle \\
\rangle & ,
\end{array}
$$

and then concatenates the inner sequences by using `flatten`.

Using our sequence comprehension notation, we can express the same code more succinctly as

```
points2D n =
  flatten ⟨⟨(x,y) : 1 ≤ y ≤ n⟩ : 0 ≤ x ≤ n − 1⟩.
```

We simplify this notation a bit more and write the same algorithm as

```
points2D n =
    ⟨(x,y) : 1 ≤ y ≤ n, 0 ≤ x ≤ n − 1⟩.
```

In this notation, we allow the expression to define the members of a sequence by using multiple variables. Note that there is an implicit `flatten` in such notation. We will have to remember this point, as we analyze algorithms that range over multiple sequences.

The notation generalizes to arbitrary levels of nesting. For example, we may want to add one more dimension to point sequences by considering points in three dimensional space, (x, y, z), whose coordinates are natural numbers that satisfy the conditions that $0 \leq x \leq n-1$, $1 \leq y \leq n$, $2 \leq z \leq n+1$. We can write the code for this by nesting the sequences in three levels as follows.

```
points3D n =
  flatten $\langle\,\langle\,\langle\,(x,y,z) : 2 \leq z \leq n+1\,\rangle : 1 \leq y \leq n\,\rangle : 0 \leq x \leq n-1\,\rangle$
```

or more succinctly as

```
points3D n =
  $\langle\,(x,y,z) : 2 \leq z \leq n+1, 1 \leq y \leq n, 0 \leq x \leq n-1\,\rangle$.
```

We can also nest other sequence operations. For example, suppose that we wish to compute the Cartesian product of two sequences, e.g., given $a = \langle\,1, 2\,\rangle$, and
$b = \langle\,3.0, 4.0, 5.0\,\rangle$.
The Cartesian product, $a \times b = \langle\,(1, 3.0), (1, 4.0), (1, 5.0), (2, 3.0), (2, 4.0), (2, 5.0)\,\rangle$. We can write the code for a function for computing as

```
CartesianProduct $(a,b)$ =
  flatten (map ($\boldsymbol{\lambda}$ x.map ($\boldsymbol{\lambda}$ y.(x,y)) b) a).
```

or equivalently as

```
CartesianProduct $(a,b)$ =
  $\langle\,(x,y) : x \in a, y \in b\,\rangle$.
```

Note that the resulting sequence is ordered by the natural lexicographic generalization of the ordering of all the sequences involved.

In general, we can `tabulate`, `map` and `filter` over any number of sequences.

Syntax 6.18. [Comprehensions for multiple sequences] We can sample from any finitely many sequences and compute an expression in terms of their elements, while also filtering the elements using finitely many expressions:

$$\langle\, e : x_1 \in e_1, x_2 \in e_2 \ldots, x_n \in e_n \mid e'_1, e'_2 \ldots e'_m \,\rangle\,.$$

We can also allow variable binding involving ranges of natural numbers, as for example, can be used by `tabulate`. Specifically, $x_i \in e_i$ could be replaced by $e_j \leq i \leq e_k$, where e_j and e_k are expressions whose values are natural numbers and i is a variable.

Example 6.19. 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. We can writes this simply by adding the filtering predicates `isEven`, which holds for even numbers, and `isVowel`, which holds for vowels.

$$\langle\, (x, y) : x \in a, y \in b \mid \texttt{isEven}\ x,\ \texttt{isVowel}\ y \,\rangle\,.$$

Example 6.20. Let's say we want to generate all contiguous subsequences of a sequence a. Each sequence can start at any position $0 \leq i < |a|$, and end at any position $i \leq j < |a|$. We can do this with the following pseudocode

$$\langle\, a\,\langle\, i, \ldots, j \,\rangle : 0 \leq i < |a|, i \leq j < |a| \,\rangle\ ,$$

which is equivalent to

```
flatten (tabulate ($\boldsymbol{\lambda}\, i.$ tabulate ($\boldsymbol{\lambda}\, j.$ a[$i, \ldots, i+j$]) $|a| - i - 1$)
                    $|a|$).
```

This example shows again that comprehensions can be quite convenient.

Remark 6.21. [Comprehensions] Syntax based on set comprehensions is included in many programming languages either directly for sets (e.g., SETL), or for other collections of values such as lists, sequences, or mappings (e.g. Python, Haskell and Javascript). We should note, however, that the syntax is not uniform among the languages. Indeed even among texts on set theory in mathematics the syntax for set comprehensions varies significantly. In our usage, we try to be self-consistent, but necessarily we are not always consistent with usage found elsewhere. To be precise we always view comprehensions as syntactic sugar for some specific function, and always define the translation between the two, as we do in Syntax 6.6).

6.4 Cost Specification

So far in this chapter, we have only specified the behavior of the operations in the sequence ADT. In this section, we consider several different cost specifications for the sequence ADT that are implemented by using arrays, trees, and lists. The cost specifications indicate the cost for the class of implementations that can achieve these cost bounds, keeping in mind that there can be many specific implementations that match the bounds. For example, for the tree-sequence specification, an implementations can use one of many balanced binary tree data structures available. To apply the cost bounds, 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.

Since there usually are many ways to implement an ADT specification, there can be multiple cost specifications for an ADT. We say that one cost specification ***dominates*** another if for every function its asymptotic costs are no higher. Of the two specifications we consider, none dominates another but there are trade-offs: while some operations may be cheaper in one specification and other operations may be more expensive.

Such trade-offs are common and should be considered when selecting which cost specification to use. When designing an algorithm, our goal would be to choose the specification that minimizes the cost for the algorithm. For example, as we will see soon, 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. Conversely, if the algorithm mostly uses `append` and `update`, then tree-sequence specification would be better. After we decide the specification to use, what remains is to select the implementation that matches the specification, which can include additional considerations.

When presenting the cost bounds, we consider the aggregation operations separately, because the cost of such operations depend on the nature of the aggregation performed.

6.4.1 Array Sequences

Cost Specification 6.22 shows the costs for array sequences. Simple operations such as `length` as well as `isEmpty` and `singleton` and `isSingleton` all require constant work and span. Since arrays support random access to any element in constant time, the function `nth` takes constant work and span. For the three operations `tabulate`, `map`, and `filter` the work includes the sum of the work of applying f at each position, as well as an additional unit cost, for the operation itself. In all three operations it is possible to apply the function f in parallel since there is no dependency among the positions. Therefore the span of the functions is the maximum of the span of applying f at each position. The operations `map` and `tabulate` incur an additional unit overhead in the span, but for `filter` the overhead is logarithmic, because `filter` requires compacting or packing the chosen elements contiguously into the result array.

The operation `subseq` has constant work and span. The operation `append` requires work

Cost Specification 6.22. [Array Sequences] We specify the ***array-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
`length` a	1	1
`singleton` x		
`isSingleton` x		
`isEmpty` x		
`nth` a i	1	1
`tabulate` f n	$1 + \sum_{i=0}^{n} W(f(i))$	$1 + \max_{i=0}^{n} S(f(i))$
`map` f a	$1 + \sum_{x \in a} W(f(x))$	$1 + \max_{x \in a} S(f(x))$
`filter` f a	$1 + \sum_{x \in a} W(p(x))$	$\log\lvert a\rvert + \max_{x \in a} S(f(x))$
`subseq` a (i,j)	1	1
`append` a b	$1 + \lvert a\rvert + \lvert b\rvert$	1
`flatten` a	$1 + \lvert a\rvert + \sum_{x \in a} \lvert x\rvert$	$1 + \log\lvert a\rvert$
`update` a (i,x)	$1 + \lvert a\rvert$	1
`inject` a b	$1 + \lvert a\rvert + \lvert b\rvert$	1
`collect` f a	$1 + W(f) \cdot \lvert a\rvert \log\lvert a\rvert$	$1 + S(f) \cdot \log^2\lvert a\rvert$
`iterate` f x a	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
`reduce` f x a	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$\log\lvert a\rvert \cdot \max_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
`scan` f a	$\lvert a\rvert$	$\log\lvert a\rvert$

proportional to the length of the sequences they are working on and 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. The operations `update` and `inject` both require work proportional to the length of the sequences they are working on, but can be implemented in constant span. 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. In Section 6.6, we describe single-threaded array sequences that will allow us under certain restrictions to update a sequence in constant work. 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.

Example 6.23. As an example of `tabulate` and `map`, we have

$$\begin{aligned} W\left(\left\langle\, i^2 : 0 \le i < n \,\right\rangle\right) &= O\left(1 + \sum_{0=1}^{n-1} O\left(1\right)\right) = O\left(n\right) \\ S\left(\left\langle\, i^2 : 0 \le i < n \,\right\rangle\right) &= O\left(1 + \max_{i=0}^{n-1} O\left(1\right)\right) = O\left(1\right) \end{aligned}$$

since the work and span for i^2 is $O\left(1\right)$.

As an example of `filter`, we have

$$\begin{aligned} W\left(\left\langle\, x : x \in a \mid x < 27 \,\right\rangle\right) &= O\left(1 + \sum_{i=0}^{|a|-1} O\left(1\right)\right) = O\left(|a|\right) \\ S\left(\left\langle\, x : x \in a \mid x < 27 \,\right\rangle\right) &= O\left(\lg|a| + \max_{i=0}^{|a|-1} O\left(1\right)\right) = O(\lg|a|) \end{aligned}$$

Example 6.24. Consider the code from Example 6.20:

$$e = \langle\, a[i, \ldots, j] : 0 \le i < |a|, i \le j < |a| \,\rangle\,,$$

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 $a[i, \ldots, j]$ (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 $||a|| = n(n+1)/2 = O(n^2)$, and $|a| = n$. The total work and span are therefore

$$\begin{array}{rcl} W\,(e) & = & O(|a|^2), \text{ and} \\ S\,(e) & = & O(\lg |a|). \end{array}$$

The cost of aggregation operations, `iterate`, `reduce`, and `scan` are somewhat more difficult to specify because they depend on the functions supplied as arguments and more specifically on the intermediate values computed during evaluation.

Cost of `iterate`. The cost of `iterate` depends not only on the arguments but also the intermediate values computed during evaluation. Example 6.25 shows an example where the length of the sequence being iterated over and that of the results are the same but the costs differ due to intermediate values.

Example 6.25. Consider appending the following sequence of strings using `iterate`:

$$\texttt{iterate append " "} \langle \texttt{"abc"}, \texttt{"d"}, \texttt{"e"}, \texttt{"f"} \rangle.$$

If we only count the work of `append` operations performed during evaluation, we obtain a total work of 19, because the following `append` operations are performed

1. `"abc"` ++ `"d"` (work 5),
2. `"abcd"` ++ `"e"` (work 6), and
3. `"abcde"` ++ `"f"` (work 7).

Consider now appending the following sequence of strings, which is a permutation of the previous, using `iterate`:

$$\texttt{iterate append " "} \langle \texttt{"d"}, \texttt{"e"}, \texttt{"f"}, \texttt{"abc"} \rangle$$

If we only count the work of `append` operations using the array-sequence specification, we obtain a total work of 15, because the following `append` operations are performed

1. `"d"` ++ `"e"` (work 3),
2. `"de"` ++ `"f"`, (work 4) and
3. `"def"` ++ `"abc"` (work 7).

In summary, we have used iteration over two sequences both with 4 elements and obtained different costs even though the sequences are permutations of each other (their elements have the same length). This is because the total cost depends on the intermediate values generated during computation.

To specify its cost, we will consider the intermediate values from the specification of `iterate`, reproduced here for convenience.

$$\texttt{iterate}\ f\ x\ a = \begin{cases} x & \text{if } |a| = 0 \\ f(x, a[0]) & \text{if } |a| = 1 \\ \texttt{iterate}\ f\ (f(x, a[0]))(a[1, \ldots, |a| - 1]) & \text{otherwise.} \end{cases}$$

Consider evaluation of `iterate` $f\ v\ A$ and let $\mathcal{T}(\texttt{iterate}\ f\ v\ A)$ 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`. We define the cost of `iterate` as the sum of these calls.

Cost Specification 6.26. [Cost for `iterate`] Consider evaluation of `iterate` $f\ v\ a$ and let $\mathcal{T}(\texttt{iterate}\ f\ v\ a)$ denote the set of calls (trace) to $f(\cdot,\cdot)$ performed along with the arguments. The work and span are

$$\begin{aligned} W\left(\texttt{iterate}\ f\ x\ a\right) &= O\left(1 + \textstyle\sum_{f(y,z)\in\mathcal{T}(\texttt{iterate}\ f\ x\ a)} W\left(f(y,z)\right)\right), \text{ and} \\ S\left(\texttt{iterate}\ f\ x\ a\right) &= O\left(1 + \textstyle\sum_{f(y,z)\in\mathcal{T}(\texttt{iterate}\ f\ x\ a)} S\left(f(y,z)\right)\right). \end{aligned}$$

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

$$\texttt{iterSort}\ a\ =\ \texttt{iterate mergeOne}\ \langle\,\rangle\ a.$$

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

$$W\left(\texttt{iterSort}\ a\right) \ \leq\ \textstyle\sum_{i=1}^{n-1} c \cdot (1+i) \ = O(n^2).$$

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 is logarithmic. We can calculate the total span as

$$S\left(\texttt{iterSort}\ a\right) \ \leq\ \textstyle\sum_{i=1}^{n-1} c \cdot \lg\left(1+i\right) \ = O(n \lg n).$$

Since average parallelism, $W\left(n\right)/S\left(n\right) = O(n/\lg n)$, we see that the algorithm has a reasonable amount of parallelism. Unfortunately, it does much too much work.

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, as shown by Example 6.27.

Example 6.27. 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. `"abc"` ++ `"d"` (work 5),
2. `"abcd"` ++ `"e"` (work 6), and
3. `"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. `"e"` ++ `"f"` (work 3),
2. `"d"` ++ `"ef"`, (work 4) and
3. `"abc"` ++ `"def"` (work 7).

To specify the cost of reduce, we therefore consider its trace based on its specification as gives in Section 6.2, reproduced below for convenience.

$$\texttt{reduce}\ f\ id\ a = \begin{cases} id & \text{if } |a| = 0 \\ a[0] & \text{if } |a| = 1 \\ f\Big(\texttt{reduce}\ f\ id\ (a[0,\ldots,\lfloor\frac{|a|}{2}\rfloor - 1]), & \\ \quad \texttt{reduce}\ f\ id\ (a[\lfloor\frac{|a|}{2}\rfloor,\ldots,|a| - 1]\Big) & \text{otherwise.} \end{cases}$$

Cost Specification 6.28. [Cost for `reduce`] Consider evaluation of `reduce` $f\ x\ a$ and let $\mathcal{T}(\texttt{reduce}\ f\ x\ a)$ denote the set of calls to $f(\cdot,\cdot)$ performed along with the arguments. The work and span are defined as

$$\begin{aligned} W\left(\texttt{reduce}\ f\ x\ a\right) &= O\left(1 + \textstyle\sum_{f(y,z)\in\mathcal{T}(\texttt{reduce}\ f\ x\ a)} W\left(f(y,z)\right)\right), \text{ and} \\ S\left(\texttt{reduce}\ f\ x\ a\right) &= O\left(\lg|a| \cdot \max_{f(y,z)\in\mathcal{T}(\texttt{reduce}\ f\ x\ a)} S\left(f(y,z)\right)\right). \end{aligned}$$

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` (ADT 6.5) is at most $O(\lg|a|)$ deep. Since each node in the recursion tree has span at most $\max_{f(y,z)} S\left(f(y,z)\right)$, any root-to-leaf path, has at most $O(\lg|a| \cdot \max_{f(a,b)} S\left(f(a,b)\right))$ 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 6.29. [Cost for `scan`] Consider evaluation of `scan` $f\ x\ a$). For both the array-sequence and tree-sequence specification

$$\begin{aligned} W\left(\texttt{scan}\ f\ x\ a\right) &= O(|a|) \\ S\left(\texttt{scan}\ f\ x\ a)\right) &= O(\lg|a|). \end{aligned}$$

6.4.2 Tree Sequences

The costs for tree sequences is given in Cost Specification 6.30. 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 not. 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

Cost Specification 6.30. [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
`length` a	1	1
`singleton` x		
`isSingleton` x		
`isEmpty` x		
`nth` $a\ i$	$\log\lvert a\rvert$	$\log\lvert a\rvert$
`tabulate` $f\ n$	$1 + \sum_{i=0}^{n} W\left(f(i)\right)$	$1 + \log n + \max_{i=0}^{n} S\left(f(i)\right)$
`map` $f\ a$	$1 + \sum_{x \in a} W\left(f(x)\right)$	$1 + \log\lvert a\rvert + \max_{x \in a} S\left(f(x)\right)$
`filter` $f\ a$	$1 + \sum_{x \in a} W\left(f(x)\right)$	$1 + \log\lvert a\rvert + \max_{x \in a} S\left(f(x)\right)$
`subseq`(a, i, j)	$1 + \log(\lvert a\rvert)$	$1 + \log(\lvert a\rvert)$
`append` $a\ b$	$1 + \lvert\log(\lvert a\rvert/\lvert b\rvert)\rvert$	$1 + \lvert\log(\lvert a\rvert/\lvert b\rvert)\rvert$
`flatten` a	$1 + \lvert a\rvert \log\left(\sum_{x \in a} \lvert x\rvert\right)$	$1 + \log(\lvert a\rvert + \sum_{x \in a} \lvert x\rvert)$
`inject` $a\ b$	$1 + (\lvert a\rvert + \lvert b\rvert)\log\lvert a\rvert$	$1 + \log(\lvert a\rvert + \lvert b\rvert)$
`collect` $f\ a$	$1 + W\left(f\right) \cdot \lvert a\rvert \log\lvert a\rvert$	$1 + S\left(f\right) \cdot \log^2\lvert a\rvert$
`iterate` $f\ x\ a$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W\left(f(y,z)\right)$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} S\left(f(y,z)\right)$
`reduce` $f\ x\ a$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W\left(f(y,z)\right)$	$\log\lvert a\rvert \cdot \max_{f(y,z) \in \mathcal{T}(-)} S\left(f(y,z)\right)$
`scan` $f\ a$	$\lvert a\rvert$	$\log\lvert a\rvert$

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 within 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.

6.4.3 List Sequences

The costs for tree sequences is given in Cost Specification 6.31. 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.

Remark 6.32. Since they are serial, list-based sequences are usually ineffective for parallel algorithm design.

6.5 An Example: Primes

We now give some more involved examples of how to use sequences and analyze work and span. As usual, the ratio of work and span gives us the parallelism of the algorithm. As our example, we consider the problem of finding prime numbers, more precisely defined as follows.

Problem 6.33. [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. If n is not prime, then it has a

Cost Specification 6.31. [List Sequences] We specify the ***array-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
`length` a `singleton` x `isSingleton` x `isEmpty` x	1	1
`nth` $a\ i$	i	i
`tabulate` $f\ n$	$1 + \sum_{i=0}^{n} W\left(f(i)\right)$	$1 + \sum_{i=0}^{n} S\left(f(i)\right)$
`map` $f\ a$	$1 + \sum_{x \in a} W\left(f(x)\right)$	$1 + \sum_{x \in a} S\left(f(x)\right)$
`filter` $f\ a$	$1 + \sum_{x \in a} W\left(p(x)\right)$	$1 + \sum_{x \in a} S\left(p(x)\right)$
`subseq` $a\ (i, j)$	$1 + i$	$1 + i$
`append` $a\ b$	$1 + \lvert a \rvert$	$1 + \lvert a \rvert$
`flatten` a	$1 + \lvert a \rvert + \sum_{x \in a} \lvert x \rvert$	$1 + \lvert a \rvert + \sum_{x \in a} \lvert x \rvert$
`update` $a\ (i, x)$	$1 + \lvert a \rvert$	$1 + \lvert a \rvert$
`inject` $a\ b$	$1 + \lvert a \rvert + \lvert b \rvert$	$1 + \lvert a \rvert + \lvert b \rvert$
`collect` $f\ a$	$1 + W\left(f\right) \cdot \lvert a \rvert \lg \lvert a \rvert$	$1 + S\left(f\right) \cdot \lvert a \rvert \lg \lvert a \rvert$
`iterate` $f\ x\ a$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W\left(f(y, z)\right)$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} S\left(f(y, z)\right)$
`reduce` $f\ x\ a$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W\left(f(y, z)\right)$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} S\left(f(y, z)\right)$
`scan` $f\ a$	$\lvert a \rvert$	$\lvert a \rvert$

divisor that is at most $\sqrt{n}$ since for any $i \times j = n$, either i or j has to be less than or equal to $\sqrt{n}$. We therefore can check if n is a prime by checking whether any i, $2 \leq i \leq \sqrt{n}$ is a divisor of n. We can write such an algorithm using sequences as follows. For simplicity we assume throughout that $n \geq 2$.

Algorithm 6.34.

```
isPrime $n$ =
  let
    $\mathit{all} = \langle\, n \bmod i : 1 \leq i \leq \lfloor \sqrt{n} \rfloor \,\rangle$
    $\mathit{divisors} = \langle\, x : x \in \mathit{all} \mid x = 0 \,\rangle$
  in
    $|\mathit{divisors}| = 1$
  end
```

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

$$W_{\texttt{isPrime}}(n) = O\left(1 + \sum_{i=1}^{\lfloor \sqrt{n} \rfloor} O(1)\right) = O\left(\sqrt{n}\right).$$

Similarly we have for span:

$$S_{\texttt{isPrime}}(n) = O\left(\lg \sqrt{n} + \max_{i=1}^{\lfloor \sqrt{n} \rfloor} O(1)\right) = O(\lg n).$$

The $\lg \sqrt{n}$ additive terms comes from the cost specification for `filter`.

Since parallelism is the ratio of work to span, the algorithm `isPrime` has parallelism of

$$O\left(\frac{\sqrt{n}}{\lg \sqrt{n}}\right).$$

This is not an abundant amount of parallelism but still plenty for all practical purposes, because work is quite small.

Now that we can test for primality of an 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.

Algorithm 6.35.

```
primesBF $n$ =
   let
      $all = \langle\, i : 1 < i < n \,\rangle$
      $primes = \langle\, x : x \in all \mid \texttt{isPrime}\ x \,\rangle$
   in
      $primes$
   end
```

Let's analyze work and span, again us array sequences. Constructing the sequence all using `tabulate` requires linear work. The work of filtering through it is the sum of the work of the calls to `isPrime`; thus we have

$$\begin{aligned} W_{\texttt{primesBF}}(n) &= O\left(\sum_{i=2}^{n-1} 1 + W_{\texttt{isPrime}}(i)\right) \\ &= O\left(\sum_{i=2}^{n-1} 1 + \sqrt{i}\right) \\ &= O\left(n^{3/2}\right). \end{aligned}$$

Similarly, the span is dominated by the maximum of the span of calls to `isPrime` and a logarithmic additive term.

$$\begin{aligned} S_{\texttt{primesBF}}(n) = &= O\left(\lg n + \max_{i=2}^{n} S_{\texttt{isPrime}}(i)\right) \\ &= O\left(\lg n + \max_{i=2}^{n} \lg i\right) \\ &= O(\lg n) \end{aligned}$$

The parallelism is hence

$$\frac{W_{\texttt{primesBF}}(n)}{S_{\texttt{primesBF}}(n)} = n^{3/2} / \lg n.$$

This is plenty of parallelism but comes at the expense of a large amount of work. We can improve the work for the algorithm significantly. Intuitively, we can see that the algorithm does a lot of redundant work, because it repeatedly performs primality checks for the same numbers. To check whether a number m is prime, the algorithm checks its divisors, it then checks essentially the same divisors for multiples of m, such as $2m, 3m, \ldots$, which largely overlap, because if a number divides m, it also divides its multiples.

We can eliminate this redundancy by more actively eliminating numbers that are composites. 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 `inject`. 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. The code for this algorithm is shown below.

Algorithm 6.36.

```
primeSieve $n$ =
  let
    $cs$ = $\langle i * j : 0 \leq i \leq \lfloor \sqrt{n} \rfloor, 2 \leq j \leq n/i \rangle$
    $sieve$ = $\langle$ ($x$, false) : $x \in cs$ $\rangle$
    $all$ = $\langle$ true : $i \leq 0 < n$ $\rangle$
    $isPrime$ = inject $all$ $sieve$
    $primes$ = $\langle i : 2 \leq i < n \mid isPrime[i] = true \rangle$
  in
    $primes$
  end
```

The work and span for calculating `primeSieve` is similar to the analysis for finding all subsequences in Example 6.24. 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 span since it just a multiply. The work for generating the sequence of composites is linear in the total number of such composites m. The span is $O(\lg m)$ because of the nested sequence and the implied `tabulate`. Constructing the sieve requires linear work in its length, which is m, and constant span.

- The work of `inject` 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 $\lceil \sqrt{n} \rceil$ have, i.e.,

$$\begin{aligned}
m &= \sum_{i=2}^{\lfloor\sqrt{n}\rfloor} \left\lceil \frac{n}{i} \right\rceil \\
&\leq (n+1) \sum_{i=2}^{\lfloor\sqrt{n}\rfloor} \frac{1}{i} \\
&= (n+1)H(\lfloor\sqrt{n}\rfloor) \\
&\leq (n+2)\ln n^{1/2} \\
&= \frac{n+2}{2}\ln n.
\end{aligned}$$

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

$$W_{\texttt{primeSieve}}(n) = O(n \lg n), \text{ and } S_{\texttt{primeSieve}}(n) = O(\lg n).$$

This is a significant improvement in the work.

The work can actually be further improved by noticing that when computing the composites, we only need to consider the multiples of prime numbers. For example we don't need to consider the multiples of 6 since all multiples of 6 are also multiples of 2 and of 3. Based on this observation, it is possible to improve the work to $O(n \lg\lg n)$ without changing the span. We leave this as an exercise.

Remark 6.37. The algorithm for computing primes described here dates back to antiquity and attributed to Eratosthenes of Cyrene, a Greek mathematician.

6.6 Single-Threaded Array Sequences

In this course we will be using purely functional code because it is safe for parallelism and enables higher-order design of algorithms by use of higher-order functions. It is also easier to reason about formally, and is just cool. For many algorithms using the purely functional version makes no difference in the asymptotic work bounds—for example quickSort and mergeSort use $\Theta(n \log n)$ work (expected case for quickSort) whether purely functional or imperative. However, in some cases purely functional implementations lead to up to a $O(\log n)$ factor of additional work. To avoid this we will slightly cheat in this class and allow for benign "effect" under the hood in exactly one ADT, described in this section. These effects do not affect the observable values (you can't observe them by looking at results), but they do affect cost analysis—and if you sneak a peak at our implementation, you will see some side effects.

The issue has to do with updating positions in a sequence. In an imperative language updating a single position can be done in "constant time". In the functional setting we are not allowed to change the existing sequence, everything is persistent. This means that for a sequence of length n an update can either be done in $\Theta(n)$ work with an arraySequence (the whole sequence has to be copied before the update) or $\Theta(\log n)$ work with a treeSequence (an update involves traversing the path of a tree to a leaf). In fact you might have noticed that our sequence interface does not even supply a function for updating a single position. The reason is both to discourage sequential computation, but also because it would be expensive.

Consider a function `update` (i, v) S that updates sequence S at location i with value v returning the new sequence. This function would have cost $\Theta(|S|)$ in the arraySequence cost specification. Someone might be tempted to write a sequential loop using this function. For example for a function $f : \alpha \to \alpha$, a `map` function can be implemented as follows:

```
fun map $f$ $S$ =
  iterate ($\boldsymbol{\lambda}$(($i$,$S'$),$v$).($i+1$, update ($i$,$f(v)$) $S'$))
          (0,$S$)
          $S$
```

This code iterates over S with i going from 0 to $n - 1$ and at each position i updates the value S_i with $f(S_i)$. The problem with this code is that even if f has constant work, with an `arraySequence` this will do $\Theta(|S|^2)$ total work since every update will do $\Theta(|S|)$ work. By using a `treeSequence` implementation we can reduce the work to $\Theta(|S| \log |S|)$ but that is still a factor of $\Theta(\log |S|)$ off of what we would like.

In the class we sometimes do need to update either a single element or a small number of elements of a sequence. We therefore introduce an ADT we refer to as a *Single Threaded Sequence* (`stseq`). Although the interface for this ADT is quite straightforward, the cost specification is somewhat tricky. To define the cost specification we need to distinguish between the latest "copy" of an instance of an `stseq`, and earlier copies. Basically whenever we update a sequence we create a new "copy", and the old "copy" is still around due to the persistence in functional languages. The cost specification is going to give different costs for updating the latest copy and old copies. Here we will only define the cost for updating and accessing the latest copy, since this is the only way we will be using an `stseq`. The interface and costs is as follows:

	Work	Span
fromSeq $S : \alpha$ seq $\rightarrow \alpha$ stseq Converts from a regular sequence to a stseq.	$O(\|S\|)$	$O(1)$
toSeq $ST : \alpha$ stseq $\rightarrow \alpha$ seq Converts from a stseq to a regular sequence.	$O(\|S\|)$	$O(1)$
nth $ST\ i : \alpha$ stseq $\rightarrow$ int $\rightarrow \alpha$ Returns the i^{th} element of ST. Same as for seq.	$O(1)$	$O(1)$
update $ST\ (i, v) :\alpha$ stseq $\rightarrow$ (int $\times\ \alpha$) $\rightarrow \alpha$ stseq Replaces the i^{th} element of ST with v.	$O(1)$	$O(1)$
inject $ST\ I : \alpha$ stseq $\rightarrow$ (int ** α) seq $\rightarrow \alpha$ stseq For each $(i, v) \in I$ replaces the i^{th} element of ST with v.	$O(\|I\|)$	$O(1)$

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 covert an stseq back to a sequence (using toSeq).

In the cost specification the work for both nth and update is $O(1)$, which is about as good as we can get. Again, however, this is only when S is the latest version of a sequence (i.e. noone 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.

Now with an stseq we can implement our map as follows:

Algorithm 6.38.

```
map $f$ $S$ =
  let
    $S'$ = StSeq.fromSeq $S$
    $R$ =  iterate $(\boldsymbol{\lambda}((i, S''), v).(i+1,$ StSeq.update $S''$ $(i, f(v))))$
                   $(0, S')$
                   $S$
  in
    StSeq.toSeq $R$
  end
```

This implementation first converts the input sequence to an stseq, then updates each element of the stseq, and finally converts back to a sequence. Since each update takes constant work, and assuming the function f takes constant work, the overall work is $O(n)$. The span is also

$O(n)$ since `iter` is completely sequential. This is therefore not a good way to implement `map` but it does illustrate that the work of multiple updates can be reduced from $\Theta(n^2)$ on array sequences or $O(n \log n)$ on tree sequences to $O(n)$ using an `stseq`.

Implementing Single Threaded Sequences. You might be curious about how single threaded sequences can be implemented so they act purely functional but match the cost specification. Here we will just briefly outline the idea.

The trick is to keep two copies of the sequence (the original and the current copy) and additionally to keep a "change log". The change log is a linked list storing all the updates made to the original sequence. When converting from a sequence to an `stseq` the sequence is copied to make a second identical copy (the current copy), and an empty change log is created. A different representation is now used for the latest version and old versions of an `stseq`. In the latest version we keep both copies (original and current) as well as the change log. In the old versions we only keep the original copy and the change log. Let's consider what is needed to update either the current or an old version. To update the current version we modify the current copy in place with a side effect (non functionally), and add the change to the change log. We also take the previous version and mark it as an old version by removing its current copy. When updating an old version we just add the update to its change log. Updating the current version requires side effects since it needs to update the current copy in place, and also has to modify the old version to mark it as old and remove its current copy.

Either updating the current version or an old version takes constant work. The problem is the cost of `nth`. When operating on the current version we can just look up the value in the current copy, which is up to date. When operating on an old version, however, we have to go back to the original copy and then check all the changes in the change log to see if any have modified the location we are asking about. This can be expensive. This is why updating and reading the current version is cheap ($O(1)$ work) while working with an old version is expensive.

In this course we will use `stseq`s for some graph algorithms, including breadth-first search (BFS) and depth-first search (DFS), and for hash tables.

6.7 Problems

6-1 Subsequences

Recall that for a sequence a, $a[i, \ldots, j]$ is the subsequence starting at position i and ending at position j. Can you compute $a[i, \ldots, j]$ using tabulate?

6-2 Matching parentheses correctly

Prove that Algorithm 6.12, which uses `iterate`, solves the parentheses matching problem.

6-3 Equivalence of reduce and iterate

Prove that `reduce` and `iterate` are equivalent when the function being applied is associative.

6-4 Parentheses matching with scan

Give an algorithm for the parentheses matching problem using `scan` instead of `iterate`.

6-5 Map and tabulate

Can we implement `map` by using `tabulate`? What is the cost of your implementation using array and tree sequences?

6-6 Cost of collect

Describe how to implement `collect` in a way that is consistent with the specified costs.

6-7 Map and tabulate

We have seen in this chapter that `map` can be implemented using `tabulate`. Is this implementation consistent with the array-sequence and the tree-sequence cost specifications?

Exercise 6.39. Recall the `mergeOne` function that we considered in Section 6.4.

- Prove that `mergeOne` is associative.
- Show that we can use `mergeOne` and `reduce` to implement a sorting algorithm.
- Analyze the work and span of the resulting algorithm.
- What algorithm does this algorithm resemble?

6-8 Primes Revisited

Describe an algorithm for computing prime numbers up to n in $O(\lg \lg n)$ work and $O(\lg n)$ span.

Chapter 7

Algorithm-Design Technique: Contraction

Contraction, an inductive technique for designing parallel algorithms, is probably one of the most important algorithm-design techniques. Like divide-and-conquer algorithms, contraction algorithms involve solving a smaller instance of the same problem, but unlike in divide-and-conquer, there is only one subproblem to solve at a time. A contraction algorithm for problem P has the following structure.

Base Case: If the problem instance is sufficiently small, then compute and return the answer, possibly using another algorithm.

Inductive Step: If the problem instance is sufficiently large, then apply the following three steps (possibly multiple times).

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.
3. ***Expand:*** use the solution to solve the original instance.

Contraction algorithms have several nice properties. First, their inductive structure enables establishing correctness and efficiency properties in a relatively straightforward fashion using principles of induction. For example, to prove a contraction algorithm correct, 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. Similarly, the work and span of a contraction algorithm can be expressed as a recursive (inductive) relation, which essentially reflects the structure of the algorithm itself. We then establish the bounds for work and span by using known, well-understood techniques for solving recursive relations as briefly discussed in Chapter 4. Second, contraction algorithms can be 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 expansions steps are efficient. Similarly, if size of the problem instance decreases geometrically, and if contraction and expansion steps low spans, then the algorithm will have a low span, and thus high parallelism.

In this chapter, we will consider several applications of the contraction technique and analyze the resulting algorithms.

7.1 Example 1: Implementing Reduce with Contraction

As our first example, we describe how to perform reduction on a sequence by using contraction. Recall that the type signature for `reduce` is as follows.

`reduce` $(f\texttt{:}\ \alpha \times \alpha \rightarrow \alpha)\ (id\texttt{:}\ \alpha)\ (A\texttt{:}\ \mathbb{S}_\alpha)\texttt{:}\ \alpha$

where f is an associative function, A is the sequence, and id is the identity element of f. Even though we have defined `reduce` broadly for both associative and non-associative functions, throughout this chapter, we restrict ourselves the case with associative functions, i.e., we assume that the function f is associative.

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. In the case of `reduce`, there are many ways to perform such a contraction. For example, we can reduce the problem to a smaller problem by removing the first element of the sequence. This approach, however, would not result in a geometric reduction in the size of the problem instance. 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

$$\langle\, 3, 5, 7, 5 \,\rangle\,.$$

By using this contraction step, we have reduced the input size by a factor of two. Note that the contraction step can be performed in parallel, because each pair can be considered independently in parallel.

Having reduced the size of the problem with the contraction step, we next solve the resulting problem by invoking the same algorithm and apply expansion to construct the final result. It is not difficult to see that by solving the smaller problem, we have actually solved the original problem, because the sum of the sequence remains the same as that of the original. Thus, the expansion step requires no additional computation. We can thus express our algorithm as follows; for simplicity, we assume that the input size is a power of two.

Algorithm 7.1. [Reduce with contract]

```
(* Assumption: $|A|$ is a power of 2 *)
reduce_contract $f$ $id$ $A$ =
  if isSingleton $A$ then
    $A[0]$
  else
    let B = $\langle\, f(A[2i], A[2i+1]) : 0 \leq i < \lfloor |A|/2 \rfloor \,\rangle$
    in reduce_contract $f$ $id$ $B$ end
```

We can usually express the work and span of a contraction algorithm using a recursive relation. 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 follows.

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

It is not difficult to solve this recursive formula to prove that the algorithm therefore performs $O(n)$ work. 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 follows.

$$S(n) = S(n/2) + 1.$$

The algorithm therefore performs $O(\log n)$ span.

7.2 Example 2: Implementing Scan with Contraction

As an even more interesting example, we describe how to implement the scan sequence primitive efficiently by using contraction. Recall that the scan function has the type signature

$$\texttt{scan}\ (f\texttt{:}\ \alpha \times \alpha \to \alpha)\ (id\texttt{:}\ \alpha)\ (A\texttt{:}\ \mathbb{S}_\alpha)\ \texttt{:}\ (\mathbb{S}_\alpha \times \alpha)$$

where f is an associative function, A is the 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. For example, 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

$$\texttt{scan}\ f\ id\ A = h\,(\texttt{iterate}\ g\ (\langle\ \rangle\ , id)\ A)\,,$$

where $g((B, b), a) = ((\texttt{append}\,\langle\, b\,\rangle\, B), f(b, a))$
and
the function $h(B, b) = ((\texttt{reverse B}), b)$, where `reverse` reverses a sequence. This algorithm, while correct, is almost entirely sequential, leaving no room for parallelism.

Considering the fact that it has to compute some value for each prefix, it may seem difficult to give a parallel algorithm for `scan`. At first glance, 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. It is this apparent dependency that makes `scan` so powerful. Indeed, we often use `scan` when it seems we need a function that depends on the results of other elements in the sequence.

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 reduction in Algorithm 7.1 and see how far that takes use. 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

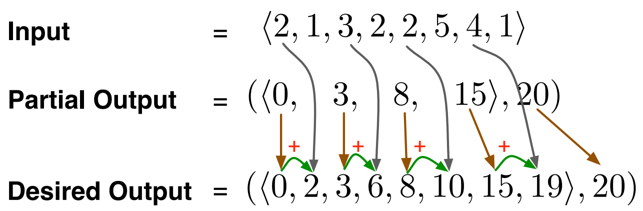
$$(\langle\, 0, 3, 8, 15\,\rangle\,, 20).$$

Notice that in this sequence, the elements in the first, third, etc., positions are actually consistent with the result expected:

$$(\langle\, 0, 2, 3, 6, 8, 10, 15, 19\,\rangle\,, 20).$$

The reason for why half of the elements is correct is 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 the position that do not fall in between a pair. Thus, we can use an expansion step to compute the missing items. It is actually quite simple: all we have to do is to compute the missing elements by applying the function element-wise to the elements of the input at odd positions in the input sequence and the results of the recursive call to `scan`.

To illustrate, the diagram below shows how to produce the final output sequence from the original sequence and the result of the recursive call:


Input = ⟨2, 1, 3, 2, 2, 5, 4, 1⟩
Partial Output = (⟨0, 3, 8, 15⟩, 20)
+
+
+
+
Desired Output = (⟨0, 2, 3, 6, 8, 10, 15, 19⟩, 20)

This leads to the following code. The algorithm we present works for when n is a power of two.

Algorithm 7.2. [Scan Using Contraction, for powers of 2]

```
(* Assumption: $|A|$ is a power of two. *)
scan $f$ $id$ $A$ $=$
  case $|A|$
  | 0 => $(\langle\,\rangle, id)$
  | 1 => $(\langle\, id\,\rangle, A[0])$
  | n =>
    let
      $A' = \langle\, f(A[2i], A[2i+1]) : 0 \le i < n/2 \,\rangle$
      $(r, t)$ = scan $f$ $id$ $A'$
    in
      $(\langle\, p_i : 0 \le i < n \,\rangle, t)$, where $p_i = \begin{cases} r[i/2] & \texttt{even}(i) \\ f(r[i/2], A[i-1]) & \texttt{otherwise} \end{cases}$
    end
```

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 follows.

$$W(n) = W(n/2) + n, \text{ and } S(n) = S(n/2) + 1,$$

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. Indeed, 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`. They yield $O(n)$ work and $O(\log n)$ span.

7.3 Problems

7-1 Ranking sequences

Given a sequence of n natural numbers and and a rank $r < n$, you want to find the element of the sequence with the given rank. For example if the input is $\langle\, 1, 3, 2, 4 \,\rangle$ and you are asked to find the element with rank 0 then the answer is 1 because 1 is the element with rank 0, i.e., the smallest element.

- Using the contraction technique, design an algorithm for finding the element of the sequence with a given rank.
- Analyze the worst-case work and span of your algorithm.
- Analyze the best-case work and span of your algorithm.

Chapter 8

Divide and Conquer

Divide and conquer is one of the most important algorithm-design techniques that can be used to solve a variety of computational problems. The structure of a divide-and-conquer algorithm for a problem P has the following form.

Base Case: When the instance I of the problem P is sufficiently small, compute the answer $P(I)$ perhaps by using a different algorithm.

Inductive Step:

1. **Divide** I into some number of smaller instances of the same problem P.
2. **Recurse** on each of the smaller instances to obtain their answers.
3. **Combine** the answers to produce an answer for the original instance I.

Divide-and-Conquer has several nice properties. First, 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 the base case is correct. Then, we 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 a correct answer. Second, divide-and-conquer technique can lead to efficient algorithms. 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. This brings us to the third nice property, which is that the work and span for a divide-and-conquer algorithm can be expressed as a mathematical equation called recurrence. Such recurrences can be solved without too much difficulty, making analyzing the work and span of many divide-and-conquer algorithms reasonably straightforward (Chapter 4). Finally, divide-and-conquer is naturally yields parallel algorithms, because we can solve the subinstances in parallel. This can lead to significant amount of parallelism, because each inductive step can create more instances to solve in parallel. For example, even if we only divide the problem instance into two sub-instances, each of those sub-instances will themselves generate two more sub-instances, leading to a geometric progression, which quickly accumulates.

In this chapter we apply the divide-and-conquer design technique to several problems and analyze their costs by using recurrences. We also discuss another design-technique, called strengthening, that allows us to apply divide-and-conquer to a wider variety of problems.

8.1 Example I: Sequence Reduce

As a simple example let's start with how we may implement `reduce` using divide and conquer.

Algorithm 8.1. [Reduce via divide and conquer]

```
reduce_dc $f$ $id$ $A$ =
  if isEmpty $A$ then
    $I$
  else if isSingleton $A$ then
    $A[0]$
  else
    let
      $(L, R)$ = splitMid $A$
      $(a, b)$ = (reduce_dc $f$ $id$ $L$ || reduce_dc $f$ $id$ $R$)
    in
      $f(a, b)$
    end
```

We can actually write the recursion for work and span as follows:

$$\begin{aligned} W(n) &= 2W(n/2) + O(1) \in O(n) \\ S(n) &= SW(n/2) + O(1) \in O(\log n). \end{aligned}$$

8.2 Example II: MergeSort

Mergesort and Quicksort are perhaps the canonical examples of divide-and-conquer. They both solve the sorting problem:

Definition 8.2. [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. $R_i \leq R_{i+1}, 0 < i \leq |A| - 1$.

Both Mergesort and Quicksort use $\Theta(n \log n)$ work, which is optimal for the comparison sorting problem. What is interesting is that one of them, Mergesort, has a trivial divide step and interesting combine step, while the other, Quicksort, has an interesting divide step but trivial combine step. We will cover Quicksort in Chapter 11 on randomized algorithms, since it involves randomization. In Mergesort the divide step simply consists of splitting the input sequence. As we will see this is actually common in several divide-and-conquer algorithms. Mergesort can be defined as follows.

Algorithm 8.3. [Mergesort]

```
merge_sort $A$ =
  **if** isEmpty $A$ or isSingleton $A$ **then**
    $A$
  **else**
    **let**
      $(L, R)$ = splitMid $A$
      $(L', R')$ = (merge_sort L || merge_sort R)
    **in**
      merge $(L', R')$
    **end**
```

In this algorithm 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. In the code, if the sequence is larger than one it is split in two approximately equal sized parts, each part is recursively sorted, and the results are merged. Recall that merging takes two sorted sequences and merges them into a single sorted sequence with the same elements. Also note that the two recursive calls are made in parallel. To prove correctness we first note that the base case is certainly correct. Then by induction, roughly, we note that L and R together contain exactly the same elements as A, that by induction L' and R' are sorted versions of L and R, and finally that merge(L', R') will therefore be a sorted version of A.

Since in divide-and-conquer algorithms, the subproblems can be solved independently, the work and span of divide-and-conquer algorithms can be described using simple recurrences. In particular for a problem of size n is broken into k subproblems of size $n_1, \ldots, n_k$, then the work is

$$W(n) \;=\; W_{\text{divide}}(n) \;+\; \sum_{i=1}^{k} W(n_i) \;+\; W_{\text{combine}}(n) + 1$$

and the span is

$$S(n) \;=\; S_{\text{divide}}(n) \;+\; \max_{i=1}^{k} S(n_i) \;+\; S_{\text{combine}}(n) + 1$$

Note that the work recurrence is simply adding up the work across all components.

More interesting is the span recurrence. First, note that a divide and conquer algorithm has to split a problem instance into subproblems before these subproblems are recursively solved. We therefore have to add the span for the divide step. The algorithm can then execute all the subproblems in parallel. We therefore take the maximum of the span for these subproblems. Finally after all the problems complete we can combine the results. We therefore have to add in the span for the combine step.

The work and span recurrence for a divide-and-conquer algorithm usually follows the recursive structure of the algorithm, but is a function of size of the arguments instead of the actual values. The resulting formulas are usually in the form of familiar recurrences such as $W(n) = 2W(n/2) + O(n)$. We will encounter many such recurrences in this book.

Assuming a linear-work, logarithmic-span algorithm for merging sorted sequences, the Mergesort algorithm leads to a recurrence of the form $W(n) = 2W(n/2) + O(n)$. This corresponds to the fact that for an input of size n, Mergesort makes two recursive calls of size $n/2$, and also does $O(n)$ work to merge the resulting sorted sequences. Similarly we can write a recurrence for the span as $S(n) = \max(S(n/2), S(n/2)) + O(\log n) = S(n/2) + O(\log n)$. This accounts for the $O(\log n)$ span of the merge algorithm, and the the fact that two recursive calls are made in parallel.

8.3 Example III: Sequence Scan

Let's consider a divide-and-conquer solution to the problem of scanning over a sequence. We can divide the sequence in two halves, solve each half, and then put the results together. Putting the results together is the tricky part. 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 (L, sL) and (R, sR), such that $((L, sL) = (\langle\, 0, 2, 3, 6 \,\rangle, 8)$ and $(R, sR) = (\langle\, 0, 2, 7, 11 \,\rangle, 12)$. Note that L already gives us the first half of the solution. To compute the second half, observe that in calculating R in the second half, we started with the identity instead of the sum of the first half, sL. Therefore, if we add the sum of the first half, sL, to each element of R, we get the desired result. This leads to the following algorithm:

Algorithm 8.4. [Scan using divide and conquer]

```
scan $f$ $id$ $A$ =
  if isEmpty $A$ then
    $(\langle\,\rangle, I)$
  else if isSingleton $A$ then
    $(\langle\, id \,\rangle, A[0])$
  else
    let
      $(AL, AR)$ = splitMid $A$
      $((L, sL), (R, sR))$ = (scan $f$ $id$ $AL$ || scan $f$ $id$ $AR$)
      $R' = \langle\, f(sL, x) : x \in R \,\rangle$
     in
       (append $(L, R')$, $sL + sR$)
     end
```

Observe that this algorithm takes advantage of the fact that id is really the "identity" for f, i.e. $f(id, x) = x$.

We now consider the work and span for the algorithm. Note that the combine step requires a map to add sL to each element of R, 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:

$$W(n) = 2W(n/2) + O(n) \in O(n \log n)$$

$$S(n) = S(n/2) + O(1) \in O(\log n).$$

Although this is much better than $O(n^2)$ work, we know that we can do better by using contraction (Chapter 7).

8.4 Example IV: Euclidean Traveling Salesperson Problem

We'll now turn to another example of divide and conquer. In this example, we will apply it to devise a heuristic method for an **NP**-hard problem. The problem we're concerned with is a variant of the traveling salesperson problem (TSP) from Chapter 5. This variant is known as the Euclidean traveling salesperson (eTSP) problem because in this problem, the points (aka. cities, nodes, and vertices) lie in a Euclidean space and the distance measure is the Euclidean measure. More specifically, we're interested in the planar version of the eTSP problem, defined as follows:

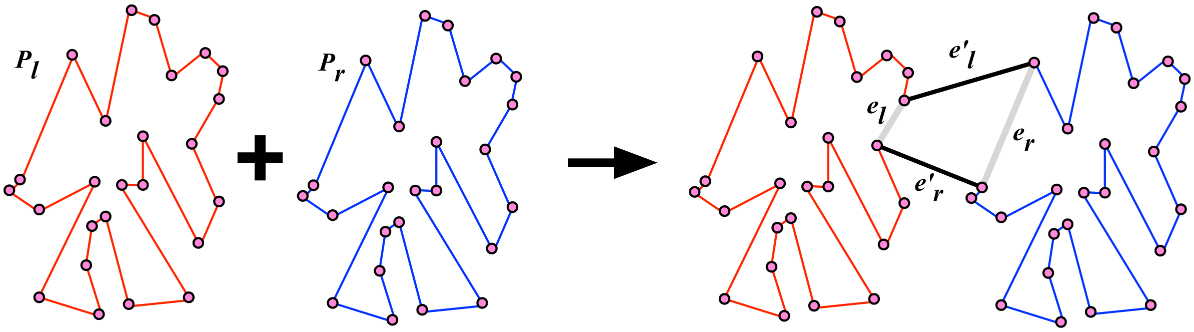
Definition 8.5. [The Planar Euclidean Traveling Salesperson Problem] Given a set of points P in the 2-d plane, the *planar Euclidean traveling salesperson* (eTSP) problem is to find a tour of minimum total distance that visits all points in P exactly once, where the distance between points is the Euclidean (i.e. ℓ_2) distance.

Not counting bridges, 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, it is **NP**-hard, but this problem is easier[1] to approximate.

Here is a heuristic divide-and-conquer algorithms that does quite well in practice. In a few weeks, we will see another algorithm based on Minimum Spanning Trees (MST) that gives a constant-approximation guarantee. This divide-and-conquer algorithm is more interesting than the ones we have done so far because it does work both before and after the recursive calls. Also, as we will see, the recurrence it generates is root dominated.

The basic idea is to split the points by a cut in the plane, solve the TSP on the two parts, and then somehow merge the solutions. For the cut, we can pick a cut that is orthogonal to the coordinate lines. In particular, we can find in which of the two dimensions the points have a larger spread, and then find the median point along that dimension. We'll split just below that point.

To merge the solutions we join the two cycles by swapping a pair of edges.



To choose which swap to make, we consider all pairs of edges of the recursive solutions consisting of one edge $e_\ell = (u_\ell, v_\ell)$ 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:

$$\texttt{swapCost}((u_\ell, v_\ell), (u_r, v_r)) = \|u_\ell - v_r\| + \|u_r - v_\ell\| - \|u_\ell - v_\ell\| - \|u_r - v_r\|$$

where $\|u - v\|$ is the Euclidean distance between points u and v.

The pseudocode for the algorithm is shown below.

[1] 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.

Algorithm 8.6.

```
eTSP($P$) =
  case ($|P|$)
  | 0,1 => raise TooSmall
  | 2 => $\langle (P[0], P[1]), (P[1], P[0]) \rangle$
  | n =>
    let
      ($P_\ell$, $P_r$) = splitLongestDim($P$)
      ($L, R$) = (eTSP($P_\ell$) || eTSP($P_r$))
      $(c, (e, e'))$ = minVal_first $\{$(swapCost($e, e'$), $(e, e')$) $: e \in L, e' \in R\}$
    in
      swapEdges(append($L, R$), $e, e'$)
    end
```

The function `minVal`$_{\texttt{first}}$ uses the first value of the pairs to find the minimum, and returns the (first) pair with that minimum. The function `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.

Cost analysis. Now let's analyze the cost of this algorithm in terms of work and span. We have

$$\begin{aligned} W(n) &= 2W(n/2) + O(n^2) \\ S(n) &= S(n/2) + O(\log n) \end{aligned}$$

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.

In anticipation of recurrences that you'll encounter later in class, we'll attempt to solve a more general form of recurrences. Let $\varepsilon > 0$ be a constant. We'll solve the recurrence

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

by the substitution method.

Theorem 8.7. *Let $\varepsilon > 0$. If $W(n) \le 2W(n/2) + k \cdot n^{1+\varepsilon}$ for $n > 1$ and $W(1) \le k$ for $n \le 1$, then for some constant κ,*

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

Proof. Let $\kappa = \frac{1}{1-1/2^\varepsilon} \cdot k$. The base case is easy: $W(1) = k \le \kappa_1$ as $\frac{1}{1-1/2^\varepsilon} \ge 1$. 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 argued that

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

□

Solving the recurrence directly. Alternatively, we could use the tree method and evaluate the sum directly. As argued before, the recursion tree here has depth $\log n$ 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

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

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

8.5 Strengthening

In most divide-and-conquer algorithms you have encountered so far, the subproblems are occurrences of the problem you are solving. For example, in sorting the subproblems are smaller sorting instances. This is not always the case. Often, you will need more information from the subproblems to properly combine the results. In this case, you'll need to ***strengthen*** the problem definition. If you have seen the approach of strengthening an inductive hypothesis in a proof by induction, it is very much an analogous idea. Strengthening involves defining a problem that solves more than what you ultimately need, but makes it easier or even possible to use solutions of subproblems to solve the larger problem.

8.5.1 Divide and Conquer with Reduce

Let's look back at divide-and-conquer algorithms you have encountered so far. Many of these algorithms have a "divide" step that simply splits the input sequence in half, proceed to solve the subproblems recursively, and continue with a "combine" step. This leads to the following structure where everything except what is in boxes is generic, and what is in boxes is specific to the particular algorithm.

```
myDC $A$ =
   if isEmpty $A$ then
      $\boxed{\texttt{emptyVal}}$
   else if isSingleton $A$ then
      $\boxed{\texttt{base}}(A[0])$
   else
      let $(L, R)$ = splitMid $A$ in
            $(L', R')$ = (myDC $L$ || myDC $R$)
      in
         $\boxed{\texttt{my\_combine}}$ $(L', R')$
      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:

$$\texttt{reduce}\ \boxed{\text{my\_combine}}\ \boxed{\text{emptyVal}}\ (\texttt{map}\ x \in S\ (\boxed{\text{base}}\ x))$$

or equivalently as

$$\texttt{reduce}\ \boxed{\texttt{my\_combine}}\ \boxed{\texttt{emptyVal}}\ (\texttt{map}\ \boxed{\texttt{base}}\ S)$$

Stylistic Notes. We have just seen that we could spell out the divide-and-conquer steps in detail or condense our code into just a few lines that take advantage of the almighty reduce. *So which is preferable*, using the divide-and-conquer code or using reduce? We believe this is a matter of taste. Clearly, your reduce code will be (a bit) shorter, and for simple cases easy to write. But when the code is more complicated, the divide-and-conquer code is easier to read, and it exposes more clearly the inductive structure of the code and so is easier to prove correct.

Restriction. You should realize, however, that 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 Quicksort as 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.

8.6 Problems

8-1 Parallel merge

Give a parallel algorithm for merging two sorted sequences. If the sequences have length m and n, then your algorithm should perform $O(n + m)$ work in $O(\log{(n + m)}$ span.

8-2 Binary expression trees

A binary expression tree is a tree where each internal node is a binary operation such as multiply or add and each leaf is a natural number.

```
type binop = Plus | Minus | Multiply | Divide
type exp_tree = Leaf of int
              | Node of (exp_tree * binop * exp_tree)

function evaluate (t: exp_tree) = ... (* your code here *)
```

- Using divide-and-conquer design an algorithm for evaluating a given expression tree to find the value represented by the expression.
- What is the worst-case work and span of your algorithm?
- What is the best-case work and span of your algorithm?

Chapter 9

Maximum Contiguous Subsequences

In this chapter, we apply the algorithm-design techniques considered thus far to a well-known problem, that of finding contiguous subsequence of a string that sums of to the maximal value. In order to exercise our vocabulary for algorithm design, we intentionally identify the techniques being used, sometimes at a level of precision that may, especially in subsequent reads, feel pedantic.

9.1 The Problem

A ***subsequence*** b of a sequence a is a sequence that can be derived from a by deleting elements of a without changing the order of remaining elements. For example, $\langle\, 0, 2, 4 \,\rangle$ is a subsequence of $\langle\, 0, 1, 2, 2, 3, 4, 5 \,\rangle$. A contiguous subsequence is a subsequence that appears contiguously in the original sequence. For example, $\langle\, 0, 2, 4 \,\rangle$ is a not subsequence of $\langle\, 0, 1, 2, 2, 3, 4, 5 \,\rangle$ but $\langle\, 2, 3, 4 \,\rangle$ is.

Definition 9.1. [Contiguous subsequence] For any sequence a of n elements, the subsequence $b = a[i, \ldots, j]$, $0 \leq i \leq j < n$, consisting of the elements of a at positions $i, i+1, \ldots, j$ is a contiguous subsequence of b.

The maximum-contiguous-subsequence problem requires finding the subsequence of a sequence of integers with maximum total sum.

Definition 9.2. [The Maximum Contiguous-Subsequence-Sum (MCS2) Problem] Given a sequence of numbers, the maximum contiguous-subsequence-sum problem requires finding the sum of the contiguous subsequence with the largest value, i.e.,

$$\mathsf{MCS2}\,(a) = \max \left\{ \sum_{k=i}^{j} a[k] \;:\; 0 \leq i, j \leq |a| - 1 \right\}.$$

For an empty sequence, the maximum contiguous subsequence sum is $-\infty$.

While we only consider sequences of numbers 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.

Before we proceed to design algorithms for the MCS2 problem, let's first establish a lower bound for the amount of work required to solve this problem. Observe that to solve the MCS2 problem, we need to inspect, at the very least, every element of the sequence. This requires linear work in the length of the sequence. We thus have a lower bound of $\Omega(n)$ for the work needed to solve MCS2.

Remark 9.3. [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 [a] 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.

[a] Jon Bentley, Programming Pearls (1st edition), page 76.

9.2 Brute Force

To solve the MCS2 problem, we can apply our most basic algorithm-design technique, the brute force method, which requires trying out all possible solutions. To apply this technique, we first identify the structure of the output, which in this case, is just a number. Thus technically speaking, we will enumerate all numbers and, for each number, check that there is a contiguous subsequence that matches that number until we find the largest number with a matching subsequence. Unfortunately such an algorithm would not terminate, because we may never know when to stop unless we know the result a priori, which we don't.

We can, however bound the sum, by adding up all positive numbers in the sequence and using that bound; clearly the sum of the maximum contiguous subsequence cannot exceed such an optimistic bound. But unfortunately such a bound can be large and cause the cost bounds to depend on the elements of the sequence rather than its length.

We can overcome this small challenge by changing our problem slightly to return a different result. More precisely, we can reduce MCS2 problem to another closely related problem: ***maximum-contiguous-subsequence***, in short ***MCS***, problem, which requires not finding the sum but the sequence itself. This reduction is quite simple: since they both operate on the same input, there is no need to convert the input, to compute the output all we have to do is sum up the elements in the sequence returned by the MCS problem. Since all we have to do is to compute the sum, which can find by using `reduce` in $O(n)$ work and $O(\log n)$ span, the work and span of the reduction is $O(n)$ and $O(\log n)$ respectively.

Thus, all we have to do now is to solve the MCS problem. We can again apply the brute-force-technique by enumerating all possible results. This time, however, it is easier to enumerate all possible results, which are contiguous subsequences of the input sequence. Since such sequences can be represented by a pair of integers (i, j), $0 \leq i \leq j < n$, we can generate all such integer pairs, compute the sum for each sequence, and pick the largest.

We thus completed our first solution. We used the reduction and the brute-force techniques. But our algorithm for solving the maximum-contiguous-subsequence problem has an important redundancy: to find the solution, it computes the result for the MCS2 problem and takes a sum, which is already computed by the MCS2 algorithm. We can eliminate this redundancy by strengthening the problem and requiring it to return the sum in addition to the subsequence. This way we can reduce the problem to the strengthened problem and compute the result in constant work.

The resulting algorithm can be specified as follows

Algorithm 9.4. [Brute Force 1]

```
BFMCS2 $a$ =
  let
    $b$ = $\langle$ reduce + 0 $a[i, \ldots, j]$ : $0 \leq i \leq j < n$ $\rangle$
  in
    reduce max $-\infty$ $b$
  end.
```

We can analyze the work and span of the algorithm by appealing to our cost bounds for `reduce`,`subseq`, and `tabulate`. Since `subseq` is somewhat more expensive with three sequences, we can use the array-sequence cost specification, where is has constant work and span.

$$\begin{aligned} W(n) &= 1 + \sum_{1 \le i \le j \le n} W_{\text{reduce}}(j-i) \le 1 + n^2 \cdot W_{\text{reduce}}(n) = 1 + n^2 \cdot \Theta(n) = \Theta(n^3) \\ S(n) &= 1 + \max_{1 \le i \le j \le n} S_{\text{reduce}}(j-i) \le 1 + S_{\text{reduce}}(n) = \Theta(\log n) \end{aligned}$$

These are cost bounds for enumerating over all possible subsequences and computing their sums. The final step of the brute-force solution is to find the maximum over these $\Theta(n^2)$ combinations. Since the reduce for this step has $\Theta(n^2)$ work and $\Theta(\log n)$ span[1], the cost of the final step is subsumed by other costs analyzed above. Overall, we have an $\Theta(n^3)$-work $\Theta(\log n)$-span algorithm.

Summary 9.5. When trying to apply the brute-force technique to the MCS2 problem, we encountered a problem. We solved this problem by reducing MCS2 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.

9.3 Brute Force Refined with a Reduction

Using the brute-force technique, we developed an algorithm that has low span. The algorithm, however, performs $\Theta(n^3)$ work, which is large. Our goal in this section is to reduce the work performed by the algorithm by a linear factor. Let's first notice that the algorithm does in fact perform a lot of redundant work, because algorithm repeats the same work many times. To see this let's consider the subsequences that start at some location, for example in the middle. For each position the algorithm considers a number of subsequnces that differ by "one" element in their ending positions. In other words many sequences actually overlap but the algorithm does not take advantage of such overlaps.

We can take advantage of such overlaps by computing all subsequences that start at a given position together. Let's call the problem of computing the maximum contiguous subsequence of a sequence that start a given position as the ***Maximum-Contiguous-Sum-with-Start***, abbreviated ***MCS3***, problem.

We can solve this problem by first computing the sum for all subsequences that start at the given position by using a scan and then computing their maximum. This algorithm can be written as follows

[1] Note that it takes the maximum over $\binom{n}{2} \le n^2$ values, but since $\log n^a = a \log n$, this is simply $\Theta(\log n)$

Algorithm 9.6. [Brute Force MCS3 Algorithm]

```
BFMCS3 $a$ $i$ =
   let
      $b$ = scan + 0 $a[i, \ldots, (|a| - 1)]$
   in
      reduce max $-\infty$ $b$
   end.
```

Since the algorithm performs a scan and a reduce, it performs (wost-case) $\Theta(n)$ work in $\Theta(\log n)$ span, for any starting position.

We can use this algorithm to find a more efficient brute-force algorithm for MCS2 by reducing that problem to it: we can try all possible start positions, solve the MCS3 problem for each, and pick the maximum of all the solutions:

Algorithm 9.7. [Brute Force 2]

```
RBFMCS2 $a$ =
   reduce max $-\infty$ $\langle$ BFMCS3 $a$ $i$ : $0 \leq i < n \rangle$.
```

This algorithm performs worst-case $\Theta(n^2)$ work in $\Theta(\log n)$ span, delivering a linear-factor improvement in work.

9.4 Reduction and Iteration

In the previous section, we reduced the MCS2 problem to the MCS3 problem, which we solved using scan. Let's now consider a closely related problem: ***Maximum-Contiguous-Subsequence at Ending***, i.e., the ***MCS2E*** problem, which requires finding the maximum contiguous subsequence ending at a specified end position. We can reduce the MCS2 problem to MCS2E problem by solving the MCS2E problem for each position and taking the maximum over all solutions.

We can solve the MCS2E problem by following a similar strategy to the MCS3 problem but we would have to reverse the list and assume furthermore that our addition operation, with which we compute sums, is commutative. Reversing is easy but we might not want to make further assumptions about the addition operation.

There is a better way. 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 $M_{i+1} = M_i \mathbin{++} \langle\, a[i] \,\rangle$ or $M_{i+1} = \langle\, a[i] \,\rangle$, depending on the sum for each.

We can now use this insight to solve MCS2 problem using iteration: we can iterate over the sequence and solve the MCS2E problem for each ending position and take the maximum over all positions. The SPARC code for this algorithm is shown in Algorithm 9.8. Note that we use the function `iteratePrefixes` to iterate over the input sequence and construct a sequence whose i^{th} position contains the solution to the MCS2E problem at that position.

Algorithm 9.8. [MCS2 with iteration]

```
IterativeMCS2 $a$ =
  let
    $f$ $(sum, x)$ =
      if $sum + x \geq x$ then
        $sum + x$
      else
        $x$
    $b$ = iteratePrefixes $f$ $-\infty$ $a$
  in
    reduce max $-\infty$ $b$
  end.
```

Let's analyze the work and span of this algorithm. To do this, we first have to decide the cost specification of sequences that we want to use. Since the algorithm only uses `iteratePrefixes` and `reduce`, we can use array sequences. Since the function f is constant work and span function, we have $W(n) = O(n)$ and $S(n) = O(n)$. Using iteration, we designed an algorithm that is work efficient, which performs asymptotically optimal work. But unfortunately the algorithm has no parallelism.

9.5 Contraction

We have been able to obtain a work-efficient algorithm for the MCS2 problem by using the brute force and reductions techniques but the algorithm has a large span. We will now design a work-efficient and low-span algorithm for the MCS2 problem using the `scan` operation.

We start with a key observation: any contiguous subsequence of a given sequence can be expressed in terms of the difference between two prefixes of the sequence. More precisely, the subsequence $A[i, \ldots, j]$ is equivalent to the subsequence $A[0, \ldots, j]$ "minus" $A[0, \ldots, i-1]$, where the operation "minus" can be specified precisely depending on the nature of the problem. For example, in MCS2 problem, we can find the sum of the elements in a contiguous subsequence `reduce +` $0\, a[i, \ldots, j]$ as follows

$$\texttt{reduce + } 0\, a[i, \ldots, j] = (\texttt{reduce + } 0\, a[0, \ldots, j]) - (\texttt{reduce + } 0\, a[0, \ldots, i-1])$$

where the "-" is the substraction operation on integers.

Based on this observation, we can solve the MCS2E problem. To see how, consider an ending position j and suppose that you 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 $A[i, \ldots, j]$ 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 i starts at position j which has the minimum of all prefixes up to i. We can compute the minimum prefix that comes before i by using just another scan. Furthermore, we can compute the minimum preceeding prefix for all positions in a single scan. Example **??** illustrates an example. The algorithm below shows a scan-based algorithm based on these insights.

Algorithm 9.9. [Scan-based MCSS]

```
ScanMCSS $a$ =
  let
    $(b, v)$ = scan + 0 $a$
    $c$ = append $b$ $\langle v \rangle$
    $(d, \_)$ = scan min $\infty$ $c$
    $e$ = $\langle\, c[i] - d[i] : 0 < i < |a| \,\rangle$
  in
    reduce max $-\infty$ $e$
  end
```

Given the costs for scan and the fact that addition and minimum take constant work, this algorithm has $\Theta(n)$ work and $\Theta(\log n)$ span. Since, we have to inspect each element of the sequence at least once to solve the MCS2 problem, this algorithm is work optimal.

Example 9.10. Consider the sequence a

$$a = \langle\, 1, -2, 0, 3, -1, 0, 2, -3 \,\rangle\,.$$

Compute

$$\begin{array}{lcl} (b, v) & = & \texttt{scan} + 0\, a \\ c & = & \texttt{append}\, b\, \langle\, v \,\rangle\,. \end{array}$$

We have $c = \langle\, 0, 1, -1, -1, 2, 2, 1, 1, 3, 0 \,\rangle$.

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

$$(d, \_) = \texttt{scan}\, \texttt{min}\, \infty\, c$$

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

$$\begin{array}{lcl} e & = & \langle\, c[i] - d[i] : 0 < i < |a| \,\rangle \\ & = & \langle\, 1, -1, 0, 3, 2, 2, 4, 1 \,\rangle\,. \end{array}$$

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 is e to compute the result

$$\texttt{reduce}\, \texttt{max}\, -\infty\, e = 4.$$

9.6 Divide And Conquer

To apply the divide-and-conquer technique, 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 9.11. 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.

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 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 in Algorithm 9.12.

Algorithm 9.12. [Simple Divide-and-Conquer MCSS]

```
DCMCS2 $a$ =
  if  $|a| = 0$  then
    $-\infty$
  else if $|a| = 1$ then
    $a[0]$
  else
    let
      $(b, c)$   = splitMid a
      $(m_b, m_c) = ($DCMCS2 $b$ || DCMCS2 $c)$
      $m_{bc}$ = bestAcross $(b, c)$
    in
      $\max\{m_b, m_c, m_{bc}\}$
    end
```

The problem of finding the maximum subsequence spanning the cut is closely related to two problems that we have seen already: Maximum-Contiguous-Subsequence Sum with Start, MCS3, and Maximum-Contiguous-Subsequence Sum at Ending, MCS2E.

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 MCS3 problem at position 0. Similarly, the suffix for the left part is exactly an instance of MCS2E problem.

Example 9.13. In Example 9.11 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 $3 + 1 = 4$.

Correctness. As described in Chapter 8, to prove a divide-and-conquer algorithm correct, we can use the technique of strong induction, which enables to assume that the theorem that we are trying to prove remains correct for all smaller subproblems. We now present such a correctness proof for the algorithm DCMCS2.

Theorem 9.14. *Let a be a sequence. The algorithm DCMCS2 returns the maximum contiguous subsequence sum in a gives 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 MCS2 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) = splitMid a`). Furthermore, let $a[i, \ldots, j]$ 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 $a[i, \ldots, j]$ lies with respect to b and c:

- If the sequence $a[i, \ldots, j]$ 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 $a[i, \ldots, j]$ 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 $a[i, \ldots, j]$ 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. □

Cost analysis. What is the work and span of our divide-and-conquer algorithm? Before we analyze the cost, let's first remark that it turns out that we can compute the maximum prefix and suffix sums in parallel by using a primitive called `scan` in $O(n)$ work and $O(\log n)$ span. Note also that `splitMid` requires $O(\log n)$ work and span using array or tree sequences. We thus have the following recurrences with array-sequence or tree-sequence specifications

$$\begin{aligned} W(n) &= 2W(n/2) + \Theta(n) \\ S(n) &= S(n/2) + \Theta(\log n). \end{aligned}$$

Using the definition of big-Θ, we know that

$$W(n) \quad \leq \quad 2W(n/2) + k_1 \cdot n + k_2,$$

where k_1 and k_2 are constants. By using the tree method, we can conclude that $W(n) = \Theta(n \lg n)$ and $S(n) = \log^2 n$.

We can also arrive at the same answer by mathematical induction. If you want to go via this route (and you don't know the answer a priori), you'll need to guess the answer first and check it. This is often called the "substitution method." Since this technique relies on guessing an answer, you can sometimes fool yourself by giving a false proof. The following are some tips:

1. Spell out the constants. Do not use the asymptotic notation—we need to be precise about constants, the asymptotic notation makes it super easy to fool ourselves.

2. Be careful that the inequalities always go in the right direction.

3. Add additional lower-order terms, if necessary, to make the induction go through.

Let's now redo the recurrences above using the substitution method. Specifically, we'll prove the following theorem using (strong) induction on n.

Theorem 9.15. *Let a constant $k > 0$ be given. If $W(n) \leq 2W(n/2) + k \cdot n$ for $n > 1$ and $W(1) \leq k$ for $n \leq 1$, then we can find constants κ_1 and κ_2 such that*

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

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

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

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

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

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

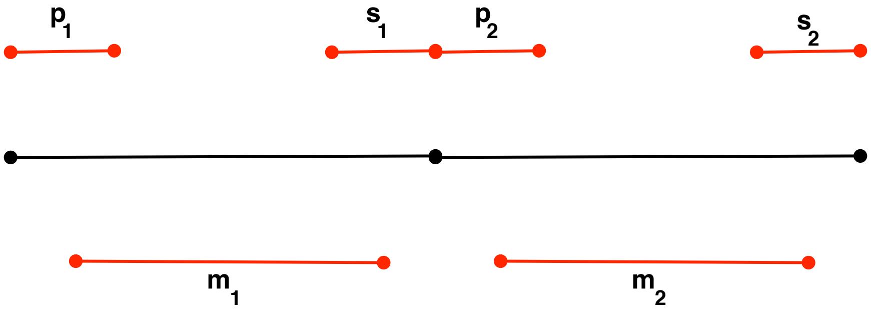


Figure 9.1: Solving the MCS2PS problem with divide and conquer.

9.7 Divide And Conquer with Strengthening

Our first divide-and-conquer algorithm performs $O(n \log n)$ work, which is $O(\log n)$ factor more than the optimal. In this section, we shall reduce the work to $O(n)$ by being more careful about avoiding redundant work. 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. Thus, the algorithm does redundant work by computing them again.

Since these should be computed as part of solving the subproblems, we should be able to return them from the recursive calls. In other words, we want to strengthen the problem so that it returns the maximum prefix and suffix. Since this problem, which we shall call ***MCS2PS***, matches the original MCS2 problem in its input and returns strictly more information, solving MCS2 using MCS2PS is trivial. We can thus focus on solving the MCS2PS problem.

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. This problem is easy to fix 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

$$(\max(s_1 + p_2, m_1, m_2), \max(p_1, t_1 + p_2), \max(s_1 + t_2, s_2), t_1 + t_2).$$

This gives the following algorithm.

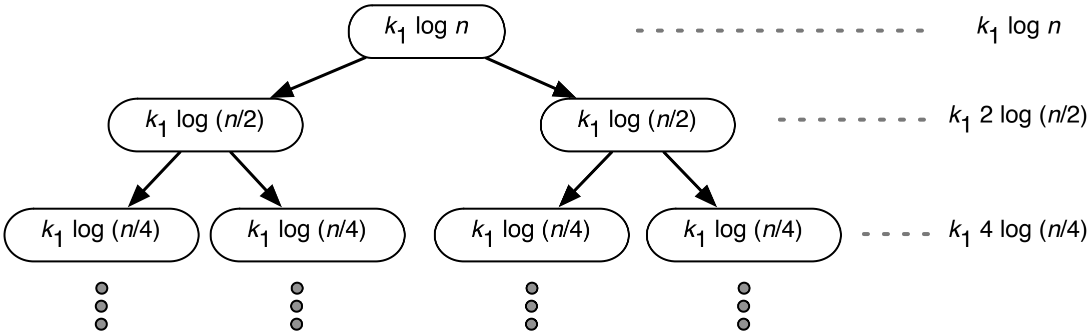
Algorithm 9.16. [Linear Work Divide-and-Conquer MCSS]

```
DCSMCS2′ $a$ =
  if $|a| = 0$ then
    $(-\infty, -\infty, -\infty, 0)$
  else if $|a| = 1$ then
    $(a[0], a[0], a[0], a[0])$
  else
    let
      $(b, c)$ = splitMid $a$
      $((m_1, p_1, s_1, t_1), (m_2, p_2, s_2, t_2))$ = (DCSMCS2′ $b$ || DCSMCS2′ $c$)
    in
      (max $(s_1 + p_2, m_1, m_2)$,    (* overall mcss *)
       max $(p_1, t_1 + p_2)$,        (* maximum prefix *)
       max $(s_1 + t_2, s_2)$,       (* maximum suffix *)
       $t_1 + t_2$)                  (* total sum *)
    end
  $(m,\_,\_,\_)$ = DCSMCS2′ $a$
in $m$ end

DCSMCS2 $a$ =
  let $(m,\_,\_,\_)$ = DCSMCS2′ $a$
  in $m$ end
```

Cost Analysis. Since `splitMid` requires $O(\log n)$ work and span in both array and tree sequences, we have

$$\begin{aligned} W(n) &= 2W(n/2) + O(\log n) \\ S(n) &= S(n/2) + O(\log n). \end{aligned}$$

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

$$W(n) \;\; \leq \;\; \sum_{i=0}^{\log n} k_1 2^i \log(n/2^i)$$

It is not so obvious to what this sum evaluates. The substitution method seems to be more convenient. We'll make a guess that $W(n) \leq \kappa_1 n - \kappa_2 \log n - k_3$. More formally, we'll prove the following theorem:

Theorem 9.17. *Let $k > 0$ be given. If $W(n) \leq 2W(n/2) + k \cdot \log n$ 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*

$$W(n) \;\leq\; \kappa_1 \cdot n - \kappa_2 \cdot \log n - \kappa_3.$$

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

$$\begin{aligned}
W(n) &\leq 2W(n/2) + k \cdot \log n \\
&\leq 2(\kappa_1 \tfrac{n}{2} - \kappa_2 \log(n/2) - \kappa_3) + k \cdot \log n \\
&= \kappa_1 n - 2\kappa_2(\log n - 1) - 2\kappa_3 + k \cdot \log n \\
&= (\kappa_1 n - \kappa_2 \log n - \kappa_3) + (k \log n - \kappa_2 \log n + 2\kappa_2 - \kappa_3) \\
&\leq \kappa_1 n - \kappa_2 \log n - \kappa_3,
\end{aligned}$$

where the final step uses the fact that $(k \log n - \kappa_2 \log n + 2\kappa_2 - \kappa_3) = (k \log n - k \log n + 2k - 2k) = 0 \leq 0$ by our choice of κ's. □

Finishing the tree method. It is possible to solve the recurrence directly by evaluating the sum we established using the tree method. We didn't cover this in lecture, but for the curious, here's how you can "tame" it.

$$\begin{aligned} W(n) &\leq \sum_{i=0}^{\log n} k_1 2^i \log(n/2^i) \\ &= \sum_{i=0}^{\log n} k_1 \left(2^i \log n - i \cdot 2^i\right) \\ &= k_1 \left(\sum_{i=0}^{\log n} 2^i\right) \log n - k_1 \sum_{i=0}^{\log n} i \cdot 2^i \\ &= k_1(2n-1)\log n - k_1 \sum_{i=0}^{\log n} i \cdot 2^i. \end{aligned}$$

We're left with evaluating $s = \sum_{i=0}^{\log n} i \cdot 2^i$. Observe that if we multiply s by 2, we have

$$2s = \sum_{i=0}^{\log n} i \cdot 2^{i+1} = \sum_{i=1}^{1+\log n} (i-1)2^i,$$

so then

$$\begin{aligned} s &= 2s - s = \sum_{i=1}^{1+\log n} (i-1)2^i - \sum_{i=0}^{\log n} i \cdot 2^i \\ &= ((1+\log n) - 1)\, 2^{1+\log n} - \sum_{i=1}^{\log n} 2^i \\ &= 2n \log n - (2n-2). \end{aligned}$$

Substituting this back into the expression we derived earlier, we have $W(n) \leq k_1(2n-1)\log n - 2k_1(n \log n - n + 1) \in O(n)$ because the $n \log n$ terms cancel.

Part III

Randomization

Chapter 10

Probability Theory

We present a brief review of basic discrete probability theory that we shall use in this book. The chapter is organized into three sections. In the first section, we describe probabilistic models, events, probabilities and conditional probabilities. In the second section, we describe random variables, which correspond closely with events, because as events, they can be thought as assigning probabilities to subsets of the sample space. The structure of this section follows that of the first section, reflecting various forms of probability laws on events to random variables. In the third and final part, we describe expectations, which can be used to summarize random variables by their average or mean value.

10.1 Probabilistic Models

Let's begin with an example.

Example 10.1. 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? You can probably figure out that the answer is

$$\frac{\text{\# of outcomes that sum to } 4}{\text{\# of total possible outcomes}} = \frac{3}{36} = \frac{1}{12}$$

since there are three ways the dice could sum to 4 (1 and 3, 2 and 2, and 3 and 1), out of the $6 \times 6 = 36$ total possibilities.

Probability theory is a mathematical study of uncertain situations such as the dice game in our example. In probability theory, we think of a game such as our dice game as an ***experiment***, because it is an "experiment" that we can repeat to obtain a number of outcomes.

The idea behind probability theory is to use a ***probabilistic model*** that models the situation in precise, mathematical terms. A probability model consists of a ***sample space*** and a ***probability function***, also called ***probability law*** that satisfies certain axioms that we shall see.

Sample Spaces and Events. A ***sample space*** Ω is an arbitrary and possibly infinite (but countable) set of possible outcomes of a probabilistic experiment. For the dice game, the sample space is the 36 possible outcomes of the dice. An ***event*** is any subset of Ω and most often representing some property common to multiple outcomes. We typically denote events by capital letters from the start of the alphabet, e.g. A, B, C.

Definition 10.2. [Probabilistic Model] A probabilistic model for an experiment consists of a ***sample space*** that represents the outcomes of that experiment and a ***probability law*** or ***probability function*** that assigns to an event A a value representing the likelihood of that event occurring.

Probability law must satisfy the following axioms.

- **Nonnegativitiy:** $\mathbf{Pr}\left[A\right] \in [0, 1]$.
- **Additivity:** for any two disjoint events A and B, $\mathbf{Pr}\left[A \cup B\right] = \mathbf{Pr}\left[A\right] + \mathbf{Pr}\left[B\right]$.
- **Normalization:** $\mathbf{Pr}\left[\Omega\right] = 1$.

Note that when defining the probabilistic model, we have not specified carefully what kinds of events, we may be interested in, because they may differ based on the experiment and what we are interested in. We do, however, must take care when setting up the probabilistic model to reason about the experiment correctly. For example, each element of the sample space must correspond to one unique 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.

When working with discrete probabilistic models, it is common practice to consider each outcome on its own and assign to it a probability. The probability for any event can then be computed by using the additivity property of the probability law and summing up the probabilities of the outcomes constituting the event.

Example 10.3. For our example of throwing two dice, the sample space consists of all of the 36 possible pairs of values of the dice:

$$\Omega = \{(1,1),(1,2),\ldots,(2,1),\ldots,(6,6)\}.$$

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

$$A = \{(d_1,d_2) \in \Omega \mid d_1 = 3\} = \{(3,1),(3,2),(3,3),(3,4),(3,5),(3,6)\}.$$

The event that "the dice sum to 4" corresponds to the set

$$B = \{(d_1,d_2) \in \Omega \mid d_1 + d_2 = 4\} = \{(1,3),(2,2),(3,1)\}.$$

Assuming the dice are unbiased, the probability law over the sample space can be written as

$$\mathbf{Pr}\left[x\right] = 1/36.$$

The probability of the event A (that the first dice is 3) is thus

$$\mathbf{Pr}\left[A\right] = \sum_{x \in A} \mathbf{Pr}\left[x\right] = \frac{6}{36} = \frac{1}{6}.$$

If the dice were biased so the probability that it shows up with a particular value is proportional to the value, then the probability function would be $\mathbf{Pr}\left[(x,y)\right] = \frac{x}{21} \times \frac{y}{21}$, and the probability of the event B (that the dice add to 4) would be

$$\mathbf{Pr}\left[B\right] = \sum_{x \in B} \mathbf{Pr}\left[x\right] = \frac{1 \times 3 + 2 \times 2 + 3 \times 1}{21 \times 21} = \frac{10}{441}.$$

Properties of Probability Laws. Given a probabilistic model, we can prove several properties of probability laws 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 $\mathbf{Pr}\left[A\right] \leq \mathbf{Pr}\left[B\right]$,

- $\mathbf{Pr}\left[A \cup B\right] = \mathbf{Pr}\left[A\right] + \mathbf{Pr}\left[B\right] - \mathbf{Pr}\left[A \cap B\right]$.

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 $A_0, A_2, \ldots, A_{n-1}$ the bound is:

$$\mathbf{Pr}\left[\bigcup_{0 \le i < n} A_i\right] \le \sum_{i=0}^{n-1} \mathbf{Pr}\left[A_i\right]$$

This bound is true unconditionally. To see why the bound holds we note that the primitive 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 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. $1/n^4$.

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 the one of dice has come up 1? Conditional probability has many practical applications in the real world. For example, given that 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 10.4. For a given probabilistic model consisting of a sample space and probability function, we define the conditional probability of an event A given B, as the probability of A occurring given that B occurs as

$$\mathbf{Pr}\left[A \mid B\right] = \frac{\mathbf{Pr}\left[A \cap B\right]}{\mathbf{Pr}\left[B\right]}.$$

The conditional probability measures the probability that the event A occurs given that B does. It is defined only when $\mathbf{Pr}\left[B\right] > 0$.

Conditional probability satisfies the three axioms of probability laws and thus itself a probability law.

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 10.5. Consider throwing two fair dice and calculate the probability that the first dice comes us 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,

$$\mathbf{Pr}\left[A \mid B\right] = \frac{\mathbf{Pr}\left[A \cap B\right]}{\mathbf{Pr}\left[B\right]} = \frac{|A \cap B|}{|B|} = \frac{1}{3}.$$

Example 10.6. Consider throwing two fair dice and calculate the probability that the first dice comes up 1 given that the second dice comes up 2. Let A be the event that the first dice comes up 1 and let B the event that the second dice comes up 2. We can write A and B in terms of outcomes as

$$\begin{aligned} A &= \{(1,1),(1,2),(1,3),(1,4),(1,5),(1,6)\} \text{ and} \\ B &= \{(1,2),(2,2),(3,2),(4,2),(5,2),(6,2)\} \text{ and} \end{aligned}$$

We thus have $A \cap B = \{(1,2)\}$. Since each outcome is equally likely,

$$\mathbf{Pr}\left[A \mid B\right] = \frac{\mathbf{Pr}\left[A \cap B\right]}{\mathbf{Pr}\left[B\right]} = \frac{|A \cap B|}{|B|} = \frac{1}{6}.$$

Since $\mathbf{Pr}\left[A\right] = \frac{|A|}{|\Omega|} = \frac{1}{6}$, we have concluded that $\mathbf{Pr}\left[A \mid B\right] = \mathbf{Pr}\left[A\right]$, that is A is independent of B. We shall see more about independence later in this section.

Total Probability Law. 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 10.7 (Total Probability Law)**.** *Consider a probabilistic model with sample space Ω and let $A_0, \ldots, A_{n-1}$ be a partition of Ω such that $\mathbf{Pr}\left[A_i\right] > 0$ for all $0 \le i < n$. For any event B the following holds*

$$\begin{aligned} \mathbf{Pr}\left[B\right] &= \mathbf{Pr}\left[B \cap A_0\right] + \mathbf{Pr}\left[B \cap A_1\right] + \ldots + \mathbf{Pr}\left[B \cap A_{n-1}\right] \\ &= \mathbf{Pr}\left[A_0\right]\mathbf{Pr}\left[B \mid A_0\right] + \mathbf{Pr}\left[A_1\right]\mathbf{Pr}\left[B \mid A_1\right] + \ldots + \mathbf{Pr}\left[A_{n-1}\right]\mathbf{Pr}\left[B \mid A_{n-1}\right] \end{aligned}$$

Example 10.8. 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 $\mathbf{Pr}\left[A_0\right] = 0.6$ and $\mathbf{Pr}\left[A_1\right] = 0.4$. Let B the event that you react, we know that $\mathbf{Pr}\left[B \mid A_0\right] = 0.8$ and $\mathbf{Pr}\left[B \mid A_1\right] = 0.1$.

We want to calculate $\mathbf{Pr}\left[B\right]$, which by total probability theorem we know to be

$$\begin{aligned}
\mathbf{Pr}\left[B\right] &= \mathbf{Pr}\left[B \cap A_0\right] + \mathbf{Pr}\left[B \cap A_1\right] \\
&= \mathbf{Pr}\left[A_0\right]\mathbf{Pr}\left[B \mid A_0\right] + \mathbf{Pr}\left[A_1\right]\mathbf{Pr}\left[B \mid A_1\right]. \\
&= 0.6 \cdot 0.8 + 0.4 \cdot 0.1 \\
&= 0.52.
\end{aligned}$$

Independence. It is sometimes important to talk 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 10.9. Two events A and B are ***independent*** if

$$\mathbf{Pr}\left[A \cap B\right] = \mathbf{Pr}\left[A\right] \cdot \mathbf{Pr}\left[B\right].$$

We say that multiple events $A_0, \ldots, A_{n-1}$ are ***mutually independent*** if and only if, for any non-empty subset $I \subseteq \{0, \ldots, n-1\}$,

$$\mathbf{Pr}\left[\bigcap_{i \in I} A_i\right] = \prod_{i \in I} \mathbf{Pr}\left[A_i\right].$$

Recall that $\mathbf{Pr}\left[A \mid B\right] = \frac{\mathbf{Pr}[A \cap B]}{\mathbf{Pr}[B]}$ when $\mathbf{Pr}\left[B\right] > 0$. Thus if $\mathbf{Pr}\left[A \mid B\right] = \mathbf{Pr}\left[A\right]$ then $\mathbf{Pr}\left[A \cap B\right] = \mathbf{Pr}\left[A\right] \cdot \mathbf{Pr}\left[B\right]$. We can thus define independence in terms of conditional probability but this works only when $\mathbf{Pr}\left[B\right] > 0$.

Example 10.10. 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

$$\begin{aligned} & \textbf{Pr}[A] \times \textbf{Pr}[B] &= \quad & \tfrac{1}{6} \times \tfrac{1}{6} &= \tfrac{1}{36} \\ = \quad & \textbf{Pr}[A \cap B] &= \quad & \textbf{Pr}[\{(1,1)\}] &= \tfrac{1}{36}\ . \end{aligned}$$

However, the event $C \equiv \{X = 4\}$ (the dice add to 4) is not independent of A since

$$\begin{aligned} & \textbf{Pr}[A] \times \textbf{Pr}[C] &= \quad & \tfrac{1}{6} \times \tfrac{3}{36} &= \tfrac{1}{72} \\ \neq \quad & \textbf{Pr}[A \cap C] &= \quad & \textbf{Pr}[\{(1,3)\}] &= \tfrac{1}{36}\ . \end{aligned}$$

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 $\frac{1}{6}$).

10.2 Random Variables

A ***random variable*** X is a real-valued function on the outcomes of an experiment, i.e., $X : \Omega \to \mathbb{R}$, i.e., it assigns a real number to each outcome. For a sample space there can be many random variables, each keeping track of some quantity of a probabilistic experiment. 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 10.11. Consider rolling a dice. The sample space is $\Omega = \{1, 2, 3, 4, 5, 6\}$. We can define a random variable X to map each primitive event to 0 if the dice comes up an even number, or 1 if the dice comes up an odd number. This random number is an indicator number indicating whether the dice comes up an odd or an even number.

A random variable is called an ***indicator random variable*** if it takes on the value 1 when some condition is true and 0 otherwise.

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:

$$\{X = x\} \equiv \{y \in \Omega \mid X(y) = x\}\ .$$

Example 10.12. For throwing two dice, we can define random variable as the sum of the two dice:

$$X(d_1, d_2) = d_1 + d_2.$$

We can define an indicator random variable to getting doubles (the two dice have the same value):

$$Y(d_1, d_2) = \begin{cases} 1 & \text{if } d_1 = d_2 \\ 0 & \text{if } d_1 \neq d_2 \end{cases}$$

Using our shorthand, the event $\{X = 4\}$ corresponds to the event "the dice sum to 4".

Remark 10.13. The term random variable might seem counter-intuitive since it is actually a function not a variable, and it is not really random at all since it is a well defined deterministic function on the sample space. However if you think of it in conjunction with the random experiment that selects a primitive element, then it is a variable that takes on its value based on a random process.

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.,

$$\mathbf{P}_X(x) = \mathbf{Pr}\left[\{X = x\}\right].$$

Example 10.14. The probability mass function for the indicator random variable X indicating whether the outcome of a roll of dice is comes up even is

$$\begin{aligned} \mathbf{P}_X(0) &= \mathbf{Pr}\left[\{X = 0\}\right] = \mathbf{Pr}\left[\{1,3,5\}\right] = 1/2, \text{ and} \\ \mathbf{P}_X(1) &= \mathbf{Pr}\left[\{X = 1\}\right] = \mathbf{Pr}\left[\{2,4,6\}\right] = 1/2. \end{aligned}$$

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

$$\begin{aligned} \mathbf{P}_X(3) &= \mathbf{Pr}\left[\{X = 3\}\right] = \mathbf{Pr}\left[\{1,3\}\right] = 1/3, \text{ and} \\ \mathbf{P}_X(7) &= \mathbf{Pr}\left[\{X = 7\}\right] = \mathbf{Pr}\left[\{2,4,5,6\}\right] = 2/3. \end{aligned}$$

Note that much like a probability law, a probability mass function is a non-negative function. It is also additive in a similar sense: for any x and x', the events $\{X = x\}$ and $\{X = x'\}$ are disjoint. Thus for any set $\bar{x}$ of values of X, we have

$$\mathbf{Pr}\left[X \in \bar{x}\right] = \sum_{x \in \bar{x}} \mathbf{P}_X(x).$$

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 probabilistic model, i.e., same sample space and probability law. 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.,

$$\mathbf{P}_{X,Y}(x,y) = \mathbf{Pr}\left[\{X = x\} \cap \{Y = y\}\right] = \mathbf{Pr}\left[X = x, Y = y\right].$$

Here $\mathbf{Pr}\left[X = x, Y = y\right]$ is shorthand for $\mathbf{Pr}\left[\{X = x\} \cap \{Y = y\}\right]$.

Given joint probability mass function for a pair of random variables X and Y, we can calculate the probability mass function for any of them, which is also called the ***marginal probability mass function*** as follows

$$\mathbf{P}_X(x) = \sum_y \mathbf{P}_{X,Y}(x,y), \text{ and } \qquad \mathbf{P}_Y(y) = \sum_x \mathbf{P}_{X,Y}(x,y).$$

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.

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 $0 \le x \le n$ is

$$\mathbf{P}_X(x) = \binom{n}{x} p^x \left(1-p\right)^{n-x}.$$

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 $0 \le x$ is

$$\mathbf{P}_X(x) = (1-p)^{x-1}p.$$

Functions of random variables. A real-valued function f of random variable is also a random variable. We can thus define new random variables by applying a function on a known random variable. 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

$$\mathbf{P}_Y(y) = \sum_{x \mid f(x)=y} \mathbf{P}_X(x).$$

Similarly, for a function of two random variables $Z = g(X,Y)$ defined on the same probabilistic model, we can write the probability mass function as

$$\mathbf{P}_Z(z) = \sum_{(x,y) \mid g(x,y)=z} \mathbf{P}_{X,Y}(x,y).$$

Example 10.15. 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 10.16. Consider the random variable X with the probability mass function

$$\mathbf{P}_X(x) = \begin{cases} 0.25 & \text{if} \quad x = -2 \\ 0.25 & \text{if} \quad x = -1 \\ 0.25 & \text{if} \quad x = 0 \\ 0.25 & \text{if} \quad x = 1 \end{cases}$$

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

$$\mathbf{P}_Y(y) = \begin{cases} 0.25 & \text{if} \quad y = 0 \\ 0.5 & \text{if} \quad y = 1 \\ 0.25 & \text{if} \quad y = 4. \end{cases}$$

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 probabilistic model, we define the ***conditional probability mass function*** of X conditioned on A as

$$\mathbf{P}_{X \mid A} = \mathbf{Pr}\left[X = x \mid A\right] = \frac{\mathbf{Pr}\left[\{X = x\} \cap A\right]}{\mathbf{Pr}\left[A\right]}.$$

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., $\mathbf{P}_{X \mid A}(x) = 1$. 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 10.17. 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

$$\begin{aligned} \mathbf{P}_{X \mid A} x &= \tfrac{\mathbf{Pr}[\{X=x\}\cap A]}{\mathbf{Pr}[A]} \\ &= \begin{cases} \frac{1/36}{1/6} = 1/6 & \text{if } x = 7, \ldots, 12. \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Since random variable 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 probabilistic model. We define the ***conditional probability mass function*** of X with respect to Y as

$$\mathbf{P}_{X \mid Y}(x \mid y) = \mathbf{Pr}\left[X = x \mid Y = y\right].$$

We can rewrite this as

$$\begin{aligned} \mathbf{P}_{X \mid Y}(x \mid y) &= \mathbf{Pr}\left[X = x \mid Y = y\right] \\ &= \tfrac{\mathbf{Pr}[X=x, Y=y]}{\mathbf{Pr}[Y=y]} \\ &= \tfrac{\mathbf{P}_{X,Y}(x,y)}{\mathbf{Pr}[Y=y]}. \end{aligned}$$

Consider the function $\mathbf{P}_{X \mid Y}(x \mid y)$ 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

$$\mathbf{P}_{X \mid Y}(x \mid y) = 1.$$

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

$$\begin{aligned}
&\mathbf{P}_{X,Y}(x,y) = \mathbf{P}_X(x)\mathbf{P}_{Y \mid X}(y \mid x)\\
&\mathbf{P}_{X,Y}(x,y) = \mathbf{P}_Y(y)\mathbf{P}_{X \mid Y}(x \mid y)
\end{aligned}$$

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:

$$\mathbf{P}_X(x) = \sum_y \mathbf{P}_{X,Y}(x,y) = \mathbf{P}_Y(y)\mathbf{P}_{X \mid Y}(x \mid y).$$

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

$$\text{for all } x : \textbf{Pr}\left[X = x \wedge A\right] = \textbf{Pr}\left[X = x\right]\textbf{Pr}\left[A\right] = \mathbf{P}_X(x)\,\textbf{Pr}\left[A\right].$$

When $\textbf{Pr}\left[A\right]$ is positive, this is equivalent to

$$\mathbf{P}_{X \mid A}(x) = \mathbf{P}_X(x).$$

Generalizing this to random variables, we can define a random variable X an ***independent*** of another random variable Y if

$$\text{for all } x, y : \mathbf{P}_{X,Y}(x,y) = \mathbf{P}_X(x)\mathbf{P}_Y(y).$$

By the definition of probability mass function, this condition is the same as the condition that $\{X = x\}$ and $\{Y = y\}$ are independent.

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.

10.3 Expectation

The ***expectation*** of a random variable X in a probabilistic model $(\Omega, \textbf{Pr})$ is the sum of the random variable over the primitive events weighted by their probability, specifically:

$$\underset{\Omega,\textbf{Pr}}{\textbf{E}}\left[X\right] = \sum_{y \in \Omega} X(y) \cdot \textbf{Pr}\left[\{y\}\right].$$

For convenience, we usually drop the $(\Omega, \textbf{Pr})$ subscript on $\textbf{E}$ since it is clear from the context.

Example 10.18. Assuming unbiased dice ($\mathbf{Pr}\left[(d_1,d_2)\right] = 1/36$), the expectation of the random variable X representing the sum of the two dice is:

$$\mathbf{E}\left[X\right] = \sum_{(d_1,d_2)\in\Omega} X(d_1,d_2) \times \frac{1}{36} = \sum_{(d_1,d_2)\in\Omega} \frac{d_1+d_2}{36} = 7.$$

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 $\mathbf{Pr}[(d_1,d_2)] = (d_1/21) \times (d_2/21)$ and:

$$\mathbf{E}\left[X\right] = \sum_{(d_1,d_2)\in\Omega} \left((d_1+d_2) \times \frac{d_1}{21} \times \frac{d_2}{21}\right) = 8\,\frac{2}{3}.$$

It is usually more natural to define expectations in terms of the probability mass function of the random variable

$$\mathbf{E}\left[X\right] = \sum_x x \cdot \mathbf{P}_X(x).$$

Example 10.19. The expectation of an indicator random variable X is the probability that the associated predicate is true (i.e. that $X = 1$):

$$\begin{aligned}\mathbf{E}\left[X\right] &= 0 \cdot \mathbf{P}_X(0) + 1 \cdot \mathbf{P}_X(1). \\ &= \mathbf{P}_X(1).\end{aligned}$$

Example 10.20. 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 10.21. 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

$$\begin{array}{rcl} \mathbf{P}_X(x) & = & \sum_{x=1}^{\infty} x \cdot (1-p)^{x-1}p \\ & = & p \cdot \sum_{x=1}^{\infty} x \cdot (1-p)^{x-1} \end{array}$$

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 $E[X] = p$.

Example 10.22. 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 $\mathbf{E}[X] = 0 + 1 \cdot 3/8 + 2 * 1/16 = 7/8$.

Markov's Inequality. Consider a non-negative random variable X. We can ask how much larger can X's maximum value be than its expected value. With small probability it can be arbitrarily much larger. However, since 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. This gives us for a non-negative random variable X the inequality:

$$\mathbf{Pr}\left[X \geq \beta\,\mathbf{E}[X]\right] \leq \frac{1}{\beta}$$

or equivalently (by substituting $\beta = \alpha/\,\mathbf{E}[X]$),

$$\mathbf{Pr}[X \geq \alpha] \leq \frac{\mathbf{E}[X]}{\alpha}$$

which is known as Markov's inequality.

Composing Expectations. Recall that functions or random variables are themselves random variables (defined on the same probabilistic model), 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 probabilistic model, 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.

Example 10.23. An important special case of functions of random variables is the linear functions. For example, let $Y = f(X) = aX + b$, where $a, b \in \mathbb{R}$.

$$\begin{array}{rcl}
\mathbf{E}\left[Y\right] = \mathbf{E}\left[f(X)\right] & = & \mathbf{E}\left[aX + b\right] \\
 & = & \sum_x f(x)\mathbf{P}_X(x) \\
 & = & \sum_x (ax + b)\mathbf{P}_X(x) \\
 & = & a\sum_x x\mathbf{P}_X(x) + b\sum_x \mathbf{P}_X(x) \\
 & = & a\,\mathbf{E}\left[X\right] + b.
\end{array}$$

Example 10.24. [Linearity of Expectations] 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

$$\mathbf{E}\left[Z\right] = \mathbf{E}\left[aX + bY + c\right] = a\,\mathbf{E}\left[X\right] + b\,\mathbf{E}\left[Y\right] + c.$$

The proof of this statement is relatively simple.

$$\begin{array}{rcl}
\mathbf{E}\left[Z\right] & = & \mathbf{E}\left[aX + bY + c\right] \\
 & = & \sum_{x,y} (ax + by + c)\mathbf{P}_{X,Y}(x,y) \\
 & = & a\sum_{x,y} x\mathbf{P}_{X,Y}(x,y) + b\sum_{x,y} y\mathbf{P}_{X,Y}(x,y) + \sum_{x,y} c\mathbf{P}_{X,Y}(x,y) \\
 & = & a\sum_{x}\sum_{y} x\mathbf{P}_{X,Y}(x,y) + b\sum_{y}\sum_{x} y\mathbf{P}_{X,Y}(x,y) + \sum_{x,y} c\mathbf{P}_{X,Y}(x,y) \\
 & = & a\sum_{x} x\sum_{y} \mathbf{P}_{X,Y}(x,y) + b\sum_{y} y\sum_{x} \mathbf{P}_{X,Y}(x,y) + \sum_{x,y} c\mathbf{P}_{X,Y}(x,y) \\
 & = & a\sum_{x} x\mathbf{P}_{X}(x) + b\sum_{y} y\mathbf{P}_{Y}(y) + c \\
 & = & a\,\mathbf{E}\left[X\right] + b\,\mathbf{E}\left[Y\right] + c.
\end{array}$$

An interesting consequence of this proof is that the random variables X and Y does not have be defined on the same probabilistic model and sample 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 $\mathbf{E}\left[X\right] \times \mathbf{E}\left[Y\right] = \mathbf{E}\left[X \times Y\right]$? 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):

$$\begin{aligned}
\mathbf{E}[X] \times \mathbf{E}[Y] &= \left(\sum_x x\,\mathbf{Pr}[\{X = x\}]\right)\left(\sum_y y\,\mathbf{Pr}[\{Y = y\}]\right) \\
&= \sum_x \sum_y (xy\,\mathbf{Pr}[\{X = x\}]\,\mathbf{Pr}[\{Y = y\}]) \\
&= \sum_x \sum_y (xy\,\mathbf{Pr}[\{X = x\} \cap \{Y = y\}]) \qquad \text{due to independence} \\
&= \mathbf{E}[X \times Y]
\end{aligned}$$

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 10.25. In Example 10.18 we analyzed the expectation on X, the sum of the two dice, by summing across all 36 primitive events. This was particulary 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:

$$\mathbf{E}[X] = 2\,\mathbf{E}[X_1] = 2 \times \sum_{d \in \{1,2,3,4,5,6\}} d \times \frac{d}{21} = 2 \times \frac{1+4+9+16+25+36}{21} = 8\,\frac{2}{3}\,.$$

Conditional Expectation. As we saw in the previous section, conditional probability mass functions are the same as ordinary probability mass functions but are obtained by normalizing the a probability mass function over a particular event or random variable. We can thus define expectations over conditional probability mass functions. For example, the conditional expectations of a random variable X given a positive-probability event A is defined as

$$\mathbf{E}[X \mid A] = \sum_x x\mathbf{P}_{X \mid A}(x).$$

Similarly, the conditional expectations of a random variable X given that another random variable Y has value y

$$\mathbf{E}[X \mid Y = y] = \sum_x x\mathbf{P}_{X \mid Y=y}(x).$$

Note that this definition is directly implied by the definition on the events because the random variable Y taking on the value y corresponds to an event.

A very useful tool in calculating the expectation of a random variable is conditioning the random variable on another random variable and then summing up the conditioned expectations. This is known as the ***total expectation theorem*** and is stated as follows:

Example 10.26. Suppose we toss n coins, where each coin has a probability p of coming up heads. What is the expected value of the random variable X denoting the total number of heads?

Solution I: We will apply the definition of expectation directly. This will rely on some messy algebra and useful equalities you might or might not know, but don't fret since this not the way we suggest you do it.

$$\begin{aligned}
\mathbf{E}[X] &= \sum_{k=0}^{n} k \cdot \mathbf{Pr}[\{X = k\}] \\
&= \sum_{k=1}^{n} k \cdot p^k (1-p)^{n-k} \binom{n}{k} \\
&= \sum_{k=1}^{n} k \cdot \frac{n}{k} \binom{n-1}{k-1} p^k (1-p)^{n-k} && [\text{ because } \binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} \text{]} \\
&= n \sum_{k=1}^{n} \binom{n-1}{k-1} p^k (1-p)^{n-k} \\
&= n \sum_{j=0}^{n-1} \binom{n-1}{j} p^{j+1} (1-p)^{n-(j+1)} && [\text{ because } k = j+1 \text{]} \\
&= np \sum_{j=0}^{n-1} \binom{n-1}{j} p^{j} (1-p)^{(n-1)-j} \\
&= np(p + (1-p))^n && [\text{ Binomial Theorem]} \\
&= np
\end{aligned}$$

That was pretty tedious :(

Solution II: We'll use linearity of expectations. Let $X_i = \mathbb{I}\{i\text{-th coin turns up heads}\}$. That is, 1 if the i-th coin turns up heads and 0 otherwise. Clearly, $X = \sum_{i=1}^{n} X_i$. So then, by linearity of expectations,

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbf{E}[X_i].$$

What is the probability that the i-th coin comes up heads? This is exactly p, so $\mathbf{E}[X] = 0 \cdot (1-p) + 1 \cdot p = p$, which means

$$\mathbf{E}[X] = \sum_{i=1}^{n} \mathbf{E}[X_i] = \sum_{i=1}^{n} p = np.$$

Example 10.27. A coin has a probability p of coming up heads. What is the expected value of Y representing the number of flips until we see a head? (The flip that comes up heads counts too.)

Solution I: We'll directly apply the definition of expectation:

$$\begin{aligned}\mathbf{E}[Y] &= \sum_{k\geq 1} k(1-p)^{k-1}p \\ &= p\sum_{k=0}^{\infty}(k+1)(1-p)^k \\ &= p\cdot\frac{1}{p^2} && \text{[by Wolfram Alpha, though you should be able to do it.]} \\ &= 1/p\end{aligned}$$

Solution II: Alternatively, we'll write a recurrence for it. As it turns out, we know that with probability p, we'll get a head and we'll be done—and with probability $1-p$, we'll get a tail and we'll go back to square one:

$$\mathbf{E}[Y] = p\cdot 1 + (1-p)\Big(1+\mathbf{E}[Y]\Big) = 1 + (1-p)\,\mathbf{E}[Y] \implies \mathbf{E}[Y] = 1/p.$$

by solving for $\mathbf{E}[Y]$ in the above equation.

The expectation of a random variable X

$$\mathbf{E}[X] = \sum_y \mathbf{P}_Y(y)\,\mathbf{E}[X \mid Y = y].$$

10.4 Problems

10-1 Probability Law Properties
Consider two events A and B in a probabilistic model and prove that

- if $A \subseteq B$, then $\mathbf{Pr}[A] \leq \mathbf{Pr}[B]$,
- $\mathbf{Pr}[A \cup B] = \mathbf{Pr}[A] + \mathbf{Pr}[B] - \mathbf{Pr}[A \cap B]$.
- $\mathbf{Pr}[A \cup B] \leq \mathbf{Pr}[A] + \mathbf{Pr}[B]$.

10-2 Union Bound
Prove the union bound.

10-3 Conditional Probability
Prove that conditional probability satisfies the three axioms of probability laws.

10-4 Disjointness and independence
Are disjoint events independent? Prove or disprove.

10-5 Unexpected independence
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. Show that these events are independent.

10-6 Conditional Maximum
Suppose that we roll a dice twice and that the first roll turns up 3. What is the probability that minimum of the two rolls is 4?

10-7 Conditional Minimum
Suppose that we roll a dice twice and that the first roll turns up 3. What is the probability that minimum of the two rolls is 5?

10-8 Marginal probability
Recall that we can calculate the marginal probability mass function for a random variable X from the probability mass function for a pair of random variables X and Y defined on the same probabilistic model as

$$\mathbf{P}_X(x) = \sum y \mathbf{P}_{X,Y}(x,y).$$

Prove that this equation is correct.

10-9 Number of trials to success
Consider performing at most n independent Bernoulli trials with a probability of success $p = 1/2$. What is the conditional probability mass function for the Bernoulli random variable, i.e., number of trials, given that the experiments ends in success.

10-10 Expected maximum
What is the expected maximum value of throwing two dice?

10-11 Independent dice
For throwing two dice, are the two random variables X and Y in Example 10.12 independent.

10-12 Expectation of univariate function
Let Y be a random variable defined as $Y = f(X)$ in terms of a real-value function of the random variable X. Prove that

$$E[Y] = E[f(X)] = \sum_x f(x)\mathbf{P}_X(x).$$

.

Chapter 11

Randomized Algorithms

The theme of this chapter is ~~*madendrizo*~~ *randomized algorithms*. These are algorithms that make use of randomness in their computation. You might know of quicksort, which is efficient on average when it uses a random pivot, but can be bad for any pivot that is selected without randomness.

Even though analyzing randomized algorithms can be difficult, randomization turns out to be a crucial technique for algorithm design, making the extra effort well worth. For example, for some problems randomized algorithms are simpler or faster than non-randomized algorithms. The problem of primality testing (PT), which is to determine if an integer is prime, is a good example. In the late 70s Miller and Rabin developed a famous and 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 much more complicated and computationally more costly than the randomized version. Hence in practice everyone still uses the randomized version.

There are many other problems in which a randomized solution is simpler or cheaper than the best non-randomized solution. In this chapter, after covering the prerequisite background, we will consider some such problems. The first we will consider is the following simple problem:

> **Question:** How many comparisons do we need to find the top two largest numbers in a sequence of n distinct numbers?

Without the help of randomization, there is a trivial algorithm for finding the top two largest numbers in a sequence that requires about $2n - 3$ comparisons. We show, however, that if the order of the input is randomized, then the same algorithm uses only $n + O(\log n)$ comparisons in expectation (on average). This matches a more complicated deterministic version based on tournaments.

Randomization plays a particularly important role in developing parallel algorithms, and analyzing such algorithms introduces some new challenges. In this chapter we will look at two randomized algorithms with significant parallelism: one for finding the k^{th} order statistics of a sequences, and the other is quicksort. In future chapters we will cover many other randomized algorithms.

In this book we require that randomized algorithms always return the correct answer, but their costs (work and span) will depend on random choices. Such algorithms are sometimes called ***Las Vegas algorithms***. Algorithms that run in a fixed amount of time, but may or may not return the correct answer, depending on random choices, are called ***Monte Carlo*** algorithms.

11.1 Expectation versus High Probability

In analyzing costs for a randomized algorithms there are two types of bounds that are useful: expected bounds, and high-probability bounds.

Expected bounds tell us about the average cost across all random choices made by the algorithm. For example if an algorithm has $\Theta(n)$ expected work, it means that on averaged over all random choices it makes in all runs, the algorithm performs $\Theta(n)$ work. Since 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.

High-probability bounds on the other hand tell us that it is very unlikely that the cost will be above some bound. For a problem of size n we say that some property is true with high probability if it is true with probability $1 - 1/n^k$ for some constant $k > 1$. This means the inverse is true with very small probability $1/n^k$. Now if we had n experiments each with inverse probability $1/n^k$ we can use the union bound to argue that the total inverse probability is $n \cdot 1/n^k = 1/n^{k-1}$. This means that for $k > 2$ the probability $1 - 1/n^{k-1}$ is still true with high probability. High-probability bounds are typically stronger than expectation bounds.

Expected bounds are quite convenient when analyzing work (or running time in traditional sequential algorithms). This is because the linearity of expectations (Chapter 10) 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. Unfortunately this kind of composition does not work when analyzing the span of an algorithm, because this requires taking the maximum of random variables, rather than their sum. For example, if we had n tasks each of which has expected span of 2 units of time, we cannot say that the expected span across all tasks is 2 units. It could be that most of the time each task has a span of 2 units, but that once with probability $1/n$, the task requires n units. The expected span for each task is still close to 2 units but if we have n tasks chances are high that one task will take n units and the expected maximum will be close to n rather than 2. We therefore cannot compose the expected span from each task by taking a maximum.

Figure 11.1: 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 dice tosser rolls a dice immediately before the race is started. The dice indicates how many initial laps everyone has to run. When the first person is about to complete the laps, a second dice is thrown indicating how many more laps everyone has to run. Clearly, some understanding of probability can help one decide how to practice for the race and how fast to run at the start. Thanks to Tom Murphy for the design of the 2007 T-shirt.

Unlike expected bounds, high-probability bounds can allow us to bound span. For example, lets say we know that every task finishes in 2 units of time with probability $1 - 1/n^5$, or equivalently that each task takes more than 2 units of time with probability $1/n^5$ and takes at most n units of time otherwise. Now with n tasks the probability that there will be at least one that requires more than 2 units of time is at most $1/n^4$ by union bound. Furthermore, when it does, the contribution to the expectation is $1/n^3$. Because of these properties of summing vs. taking a maximum, in this book we often analyze work using expectation, but analyze span using high probability.

11.2 Finding The Two Largest

The max-two problem is to find the two largest elements from a sequence of n (unique) numbers. Lets consider the following simple iterative algorithm for the problem.

Algorithm 11.1. [Iterative Max-Two]

```
1  max2 $a$ =
2  let
3     update $((m_1, m_2), v)$ =
4        if $v \leq m_2$ then $(m_1, m_2)$
5        else if $v \leq m_1$ then $(m_1, v)$
6        else $(v, m_1)$
7     init = if $a[0] \geq a[1]$ then $(a[0], a[1])$ else $(a[1], a[0])$
8  in
9     iter update init $a[2, \ldots, n-1]$
10 end
```

In the following analysis, we will be meticulous about constants. This iterative algorithm requires up to $1+2(n-2) = 2n-3$ comparisons since there is one comparison in `init` and since each of the $n-2$ `update`'s requires up to two comparisons. On the surface, this may seem like the best one can do. Surprisingly, there is a divide-and-conquer algorithm that uses only about $3n/2$ comparisons (exercise to the reader). More surprisingly still is the fact that it can be done in $n + O(\log n)$ comparisons. But how?

A closer look at the analysis above reveals that we were pessimistic about the number of comparisons; not all elements will get past the "if" statement in Line **??**; therefore, only some of the elements will need the comparison in Line **??**. But we didn't know how many of them, so we analyzed it in the worst possible scenario.

Let's try to understand what's happening better by looking at the worst-case input. It is not difficult to convince yourself that an increasing sequence of length n, e.g., $\langle 1, 2, 3, \ldots, n \rangle$ leads to $2n-3$ comparisons. As we iterate from left to right, we find a new maximum for each new element—this new element gets compared in both Lines 4 and 5.

But perhaps it's unlikely to get such a deliberately structured sequence if we consider the elements in random order. With only 1 in $n!$ chance, a sequence will be fully sorted. You can work out the probability that the random order will result in a sequence that looks "approximately" sorted, and it would not be too high. Thus we can reasonably hope to save a lot of comparisons in Line **??** by considering elements in random order.

Let's thus analyze the following algorithm: on input a sequence t of n elements:

1. Let $a = \texttt{permute}(t, \pi)$, where π is a random permutation (i.e., we choose one of the $n!$ permutations).

2. Run algorithm `max2` on a.

Note that we don't need to explicitly construct a. All we need instead is to pick a random element that hasn't been considered and consider that element next. For the analysis, it is convenient to describe the process in terms of a randomly permuted sequence.

After applying the random permutation we have that our sample space Ω corresponds to each permutation. Since there are $n!$ permutations on a sequence of length n and each has equal probability, we have $|\Omega| = n!$ and $\textbf{Pr}\left[x\right] = 1/n!, x \in \Omega$. However, as we will see, we do not really need to know this, all we need to know is what fraction of the sample space obeys some property.

Let i be the position in a (indexed from 1 to n). Now let X_i be an indicator random variable denoting whether Line **??** and hence its comparison gets executed for the value at S_i (i.e., Recall that an indicator random variable is actually a function that maps each primitive event (each permutation in our case) to 0 or 1. In particular given a permutation, it returns 1 iff for that permutation the comparison on Line 5 gets executed on iteration i. Lets say we want to now compute the total number of comparisons. We can define another random variable (function) Y that for any permutation returns the total number of comparisons the algorithm takes on that permutation. This can be defined as

$$Y = \underbrace{1}_{\text{Line } 7} + \underbrace{n-2}_{\text{Line } 4} + \underbrace{\sum_{i=3}^{n} X_i}_{\text{Line } 5} .$$

We are interested in computing the expected value of Y. By linearity of expectation, we have

$$\begin{aligned} \textbf{E}\left[Y\right] &= \textbf{E}\left[1 + (n-2) + \sum_{i=3}^{n} X_i\right] \\ &= 1 + (n-2) + \sum_{i=3}^{n} \textbf{E}\left[X_i\right]. \end{aligned}$$

Our tasks therefore boils down to computing $\textbf{E}\left[X_i\right]$ for $i = 3, \ldots, n$. To compute this expectation, we ask ourselves: *What is the probability that $a_i > m_2$?* A moment's thought shows that the condition $a_i > m_2$ holds exactly when a_i is either the largest element or the second largest element in $\{a_1, \ldots, a_i\}$. So ultimately we're asking: what is the probability that a_i is the largest or the second largest element in randomly-permuted sequence of length i?

To compute this probability, we note that each element in the sequence is equally likely to be anywhere in the permuted sequence (we chose a random permutation. In particular, if we look at the k-th largest element, it has $1/i$ chance of being at a_i. (You should also try to work it out using a counting argument.) Therefore, the probability that a_i is the largest or the second largest element in $\{a_1, \ldots, a_i\}$ is $\frac{1}{i} + \frac{1}{i} = \frac{2}{i}$, so

$$\textbf{E}\left[X_i\right] = 1 \cdot \tfrac{2}{i} = 2/i.$$

Plugging this into the expression for $\mathbf{E}[Y]$, we obtain

$$\begin{aligned}
\mathbf{E}[Y] &= 1 + (n-2) + \sum_{i=3}^{n} \mathbf{E}[X_i] \\
&= 1 + (n-2) + \sum_{i=3}^{n} \frac{2}{i} \\
&= 1 + (n-2) + 2\Big(\tfrac{1}{3} + \tfrac{1}{4} + \ldots \tfrac{1}{n}\Big) \\
&= n - 4 + 2\Big(1 + \tfrac{1}{2} + \tfrac{1}{3} + \tfrac{1}{4} + \ldots \tfrac{1}{n}\Big) \\
&= n - 4 + 2H_n,
\end{aligned}$$

where H_n is the n-th Harmonic number. But we know that $H_n \leq 1 + \lg n$, so we get $\mathbf{E}[Y] \leq n - 2 + 2\lg n$. We can also use the following bound on Harmonic sums:

$$H(n) = O(\lg n + 1),$$

or more precisely

$$H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n} = \ln n + \gamma + \varepsilon_n,$$

where γ is the Euler-Mascheroni constant, which is approximately $0.57721\cdots$, and $\varepsilon_n \sim \frac{1}{2n}$, which tends to 0 as n approaches ∞. This shows that the summation and integral of $1/i$ are almost identical (up to a small adative constant and a low-order vanishing term).

Remark 11.2. Reducing the number of comparisons by approximately a factor of two might not lead to a significant gain in performance in practice. For example, if the comparison function is a constant-time, simple comparison function the $2n-3$ algorithm and the $n-1+2\log n$ algorithm are unlikely to be significant. For most cases, the $2n-3$ algorithm might in fact be faster to due better locality.

The point of this example is to demonstrate the power of randomness in achieving something that otherwise seems impossible—more importantly, the analysis hints at why on a typical "real-world" instance, the $2n-3$ algorithm does much better than what we analyzed in the worst case (real-world instances are usually not adversarial).

11.3 Order statistics

In statistics, computing the order statistics of sample, which we may represent as a sequence, has many important applications. We can precisely state the problem as follows.

Problem 11.3. [Order statistics] Given an a sequence and an integer k where $0 \leq k < |a|$, and a comparison $<$ defining a total ordering over the elements of the sequence, find the k^{th} order statistics, i.e., k^{th} smallest element, in the sequences.

We can solve this problem by sorting first and selecting the k^{th} element but this would require $O(n \log n)$ work, assuming that comparisons require constant work. We wish to do better; in particular we would like to achieve linear work and still achieve $O(\log^2 n)$ span. For the purposes of simplicity, let's assume that sequences consist of unique elements and consider the following simple algorithm. Based on the contraction design technique, the algorithm uses randomization to contract the problem to a smaller instance.

Algorithm 11.4. [contracting k^{th} smallest]

```
select a k =
let
  $p = a[0]$
  $\ell = \langle\, x \in a \mid x < p \,\rangle$
  $r = \langle\, x \in b \mid x > p \,\rangle$
in
  if $(k < |\ell|)$ then select $\ell$ $k$
  else if $(k < |a| - |r|)$ then $p$
  else select $r$ $(k - (|a| - |r|))$
```

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 recursively explores that side. When exploring the right side, r, the parameter k needs to be adjusted by since all elements less or equal to the pivot p are being thrown out: there are $|a| - |r|$ such elements.

As written the algorithm picks as pivot the first key in the sequence instead of a random key. As with the two-largest problem, we can add randomness by first randomly permuting a sequence t to generate the input sequence a and then applying `select` on a. This is equivalent to randomly picking a pivot at each step of contraction.

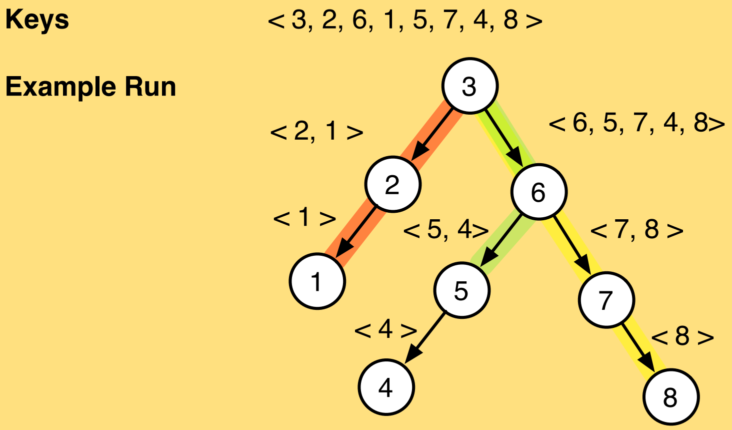
Let's analyze the work and span of the randomized algorithm where we pick pivots uniformly randomly. Let $n = |a|$ and define $X(n) = \max\{|\ell|, |r|\}/|a|$, which is the fractional size of the larger side. Notice that X is an upper bound on the fractional size of the side the algorithm actually recurs into. Now since lines 3 and 4 are simply two `filter` calls, we have the following recurrences:

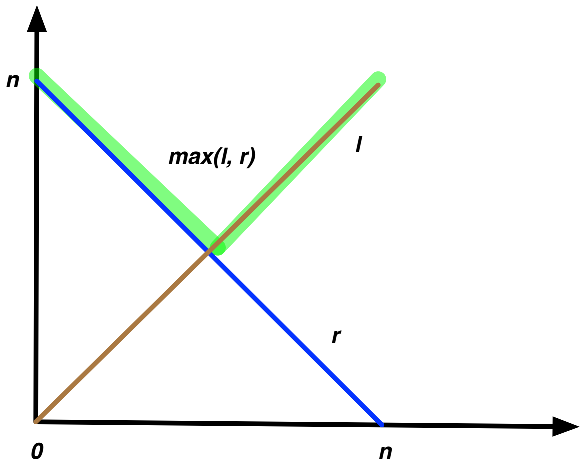
$$\begin{array}{lcllcl} W(n) & \leq & W(X(n) \cdot n) & + & O(n) \\ S(n) & \leq & S(X(n) \cdot n) & + & O(\log n) \end{array}$$

Let's first look at the work recurrence. Specifically, we are interested in $\mathbf{E}\left[W(n)\right]$. First, let's try to get a sense of what happens in expectation.

The key quantity in bounding th expectation is bounding $\mathbf{E}\left[X(n)\right]$. To this end, let's none first that all pivots are equally likely. We can thus draw the following plot of the size of ℓ and size of r as a function of where the pivot belongs in the sorted order of a.

Example 11.5. 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 eight-order statistics, $k = 7$.





If the pivot is at the start then ℓ is empty and $|r| = |a| - 1$, and if the pivot is at the end then r is empty and $|\ell| = |a| - 1$. Since the probability that we land on a point on the x axis is $1/n$, we can write the expectation for $X(n)$ as

$$\mathbf{E}\left[X(n)\right] = \frac{1}{n}\sum_{i=0}^{n-1}\frac{\max\{i, n-i-1\}}{n} \le \frac{1}{n}\sum_{j=n/2}^{n-1}\frac{2}{n}\cdot j \le \frac{3}{4}$$

(Recall that $\sum_{i=x}^{y} i = \frac{1}{2}(x+y)(y-x+1)$.)

This calculation tells us that in expectation, $X(n)$ is a constant fraction smaller than 1, so intuitively in calculating the work we should have a nice geometrically decreasing sum that adds up to $O(n)$. It is not quite so simple, however, since the constant fraction is only in expectation. It could also be we are unlucky for a few contraction steps and the sequences size hardly goes down at all. We will cover other algorithms on graphs that have the same property, i.e. that the size goes down by an expected constant factor on each contraction step. The following theorem shows 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. 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.

Theorem 11.6. *Starting with size n, the expected size of a in algorithm* `select` *after d recursive calls is $\left(\frac{3}{4}\right)^d n$.*

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 size of the input to the d^{th} recursive call and let Z the pivot chosen at the d^{th} call. 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

$$\begin{array}{rcl} E[Y_{d+1}] & = & \sum_{y,z} y f(y,z) \mathbf{P}_{Y,Z}(y,z) \\ & = & \sum_{y} \sum_{z} y f(y,z) \mathbf{P}_{Y}(y) \mathbf{P}_{Z \mid Y}(z \mid y) \\ & = & \sum_{y} y \mathbf{P}_{Y}(y) \sum_{z} f(y,z) \mathbf{P}_{Z \mid Y}(z \mid y) \\ & \leq & \sum_{y} y \mathbf{P}_{Y}(y)\, \mathbf{E}\left[X(y)\right]. \\ & \leq & \frac{3}{4} \sum_{y} y \mathbf{P}_{Y}(y). \\ & \leq & \frac{3}{4}\, \mathbf{E}\left[Y_d\right]. \end{array}$$

Note that we have used the bound

$$\mathbf{E}\left[X(y)\right] = \sum_{z} f(y,z) \mathbf{P}_{Z \mid Y}(z \mid y) \leq \frac{3}{4},$$

which we established above.

We thus conclude that $\mathbf{E}\left[Y_{d+1}\right] = \frac{3}{4}\, \mathbf{E}\left[Y_d\right]$, which this trivially solves to the bound given in the theorem, since at $d = 0$ the input size is n. □

The work at each level of the recursive calls is linear in the size of the input and thus can be written as $W_{\texttt{select}}(n) \leq k_1 n + k_2$, where n is the input size. Since 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, and thus, we can bound the expected work as

$$\begin{aligned}
\mathbf{E}\left[W_{\texttt{select}}(n)\right] &\le \sum_{i=0}^{n}(k_1\,\mathbf{E}\left[Y_i\right] + k_2) \\
\mathbf{E}\left[W_{\texttt{select}}(n)\right] &\le \sum_{i=0}^{n}(k_1 n\left(\frac{3}{4}\right)^i + k_2) \\
&\le k_1 n\left(\sum_{i=0}^{n}\left(\frac{3}{4}\right)^i\right) + k_2 n \\
&\le 4k_1 n + k_2 n \\
&\in O(n).
\end{aligned}$$

Expected Span. We can bound the span of the algorithm by $O(n \lg n)$ trivially 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 Theorem 11.6 to bound the number of levels taken by `select` more tightly using a high probability bound.

Consider depth $d = 10\lg n$. At this depth, the expected size upper bounded by $n\left(\frac{3}{4}\right)^{10\lg n}$. With a little math this is equal to $n \times n^{-10\lg(4/3)} \approx n^{-3.15}$. Now, by Markov's inequality, if the expected size is at most $n^{-3.15}$ then the probability of having size at least 1 is bounded by

$$\mathbf{Pr}\left[Y_{10\lg n} \ge 1\right] \le E[Y_{10\lg n}]/1 = n^{-3.15}.$$

In applying Markov's inequality, we choose 1, because we know that the algorithm terminates for that input size. By increasing the constant factor from 10 to 20 would decrease the probability to $n^{-7.15}$, which is extremely unlikely: for $n = 10^6$ this is 10^{-42}. We have therefore shown that the number of steps is $O(\log n)$ with high probability. Each step has span $O(\log n)$ so the overall span is $O(\log^2 n)$ with high probability.

Using the high probability bound, we can bound the expected span by using the total expectation theorem. For brevity let the random variable Y be defined as $Y = Y_{10\lg n}$,

$$\begin{aligned}
\mathbf{E}\left[S\right] &= \textstyle\sum_y \mathbf{P}_Y(y)\,\mathbf{E}\left[S \mid Y = y\right]. \\
&= \textstyle\sum_{y \le 1} \mathbf{P}_Y(y)\,\mathbf{E}\left[S \mid Y = y\right] + \sum_{y>1} \mathbf{P}_Y(y)\,\mathbf{E}\left[S \mid Y = y\right] \\
&\le (1 - n^{-3.5})O(\lg^2 n) + n^{-3.5}O(n) \\
&= O(\lg^2 n).
\end{aligned}$$

The expected bound follows by the fact that with high probability the depth of the recursive calls is $O(\lg n)$ and that each recursive call has $O(\lg n)$ span, because it requires a sequences `filter`. The span for the case when the span is not greater that $10\lg n$ contributes only a constant value to the expectation as long as it is a polynomial that is less that $n^{3.5}$.

In summary, we have shown than the `select` algorithm on input of size n does $O(n)$ work in expectation and has $O(\log^2 n)$ span with high probability. As mentioned at the start of the chapter, we will typically be analyzing work using expectation and span using high probability.

Algorithm 11.7. [Quicksort]

```
quicksort $a$ =
if $|a| = 0$ then $a$
else
   let
      $p$ = pick a pivot from $a$
      $a_1 = \langle\, x \in a \mid x < p \,\rangle$
      $a_2 = \langle\, x \in a \mid x = p \,\rangle$
      $a_3 = \langle\, x \in a \mid x > p \,\rangle$
      $(s_1, s_3)$ = (sort $a_1$ || sort $a_3$)
   in
      $s_1$ ++ $a_2$ ++ $s_3$
   end
```

11.4 Quicksort

Moving on to a more complex algorithm, let's analyze the work and span of the randomized quicksort algorithm. In later chapters we will see that the analysis of quicksort presented here is 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.

Consider the quicksort algorithm given in Algorithm 11.7. In this algorithm, we intentionally leave the pivot-choosing step unspecified because the property we are discussing holds regardless of the choice of the pivot.

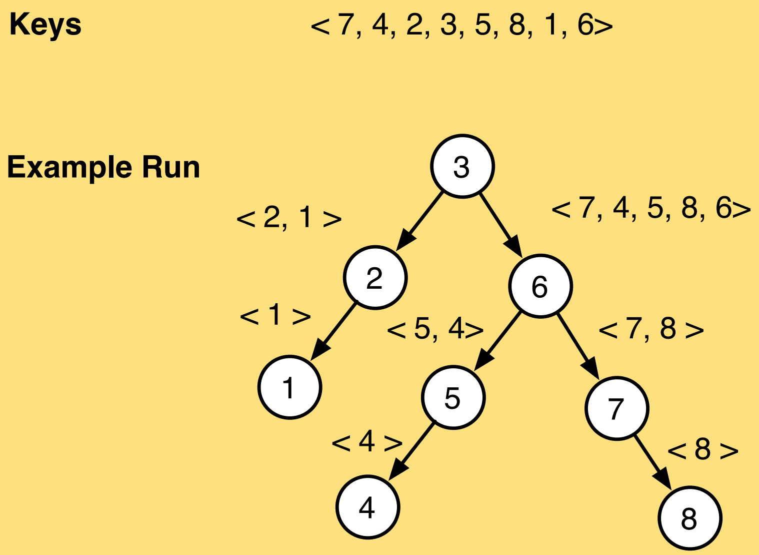
There is plenty of parallelism in this version quicksort. There is both parallelism due to the two recursive calls and in the fact that the filters for selecting elements greater, equal, and less than the pivot can be parallel.

Note that each call to quicksort either makes no recursive calls (the base case) or two recursive calls. The call tree is therefore binary. We will often find it convenient to map the run of a quicksort to a binary-search tree (BST) representing the recursive calls along with the pivots chosen. We will sometimes refer to this tree as the *call tree* or ***pivot tree***. We will use this call-tree representation to reason about the properties of quicksort, e.g., the comparisons performed, its span. An example is shown in Example 11.8.

Let's consider some strategies for picking a pivot.

- **Always pick the first element:** If the sequence is sorted in increasing order, then picking the first element is the same as picking the smallest element. We end up with a lopsided recursion tree of depth n. The total work is $O(n^2)$ since $n-i$ keys will remain at level i and

Example 11.8. An example run of quicksort along with its pivot tree.



hence we will do $n-i-1$ comparisons at that level for a total of $\sum_{i=0}^{n-1}(n-i-1)$. Similarly, if the sequence is sorted in decreasing order, we will end up with a recursion tree that is lopsided in the other direction. In practice, it is not uncommon for a sort function input to be a sequence that is already sorted or nearly sorted.

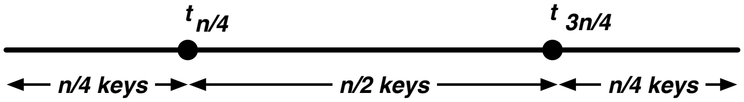
- **Pick the median of three elements:** Another strategy is to take the first, middle, and the last elements and pick the median of them. For sorted lists the split is even, so each side contains half of the original size and the depth of the tree is $O(\log n)$. Although this strategy avoids the pitfall with sorted sequences, it is still possible to be unlucky, and in the worst-case the costs and tree depth are the same as the first strategy. This is the strategy used by many library implementations of quicksort. Can you think of a way to slow down a quicksort implementation that uses this strategy by picking an adversarial input?

- **Pick an element randomly:** It is not immediately clear what the depth of this 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 hope that this strategy will result in a tree of depth $O(\log n)$ in expectation or with high probability. Indeed, picking a random pivot gives us expected $O(n \log n)$ work and $O(\log^2 n)$ span for quicksort and an expected $O(\log n)$-depth tree, as we will show.

Analysis of Quicksort

To develop some intuition for the span analysis, let's consider the probability that we split the input sequence more or less evenly. If we select a pivot that is greater than $t_{n/4}$ and less than

$t_{3n/4}$ then $X(n)$ is at most $3n/4$. Since all keys are equally likely to be selected as a pivot this probability is $\frac{3n/4-n/4}{n} = 1/2$. The figure below illustrates this.



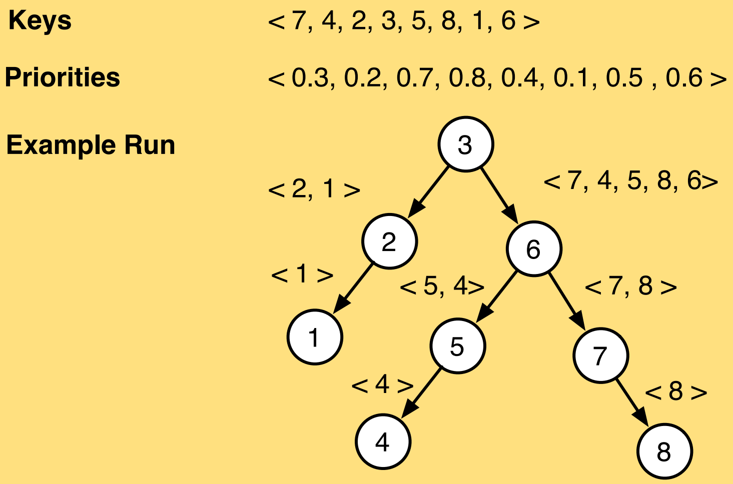
This observations implies that at each level of the call tree (every time a new pivot is selected), the size of the input to both calls decrease by a constant fraction (of $3/4$). At every two levels, the probability that the input size decreases by $3/4$ is the probability that it decreases at either step, which is at least $1 - \frac{1}{2} \cdot \frac{1}{2} = \frac{3}{4}$, etc. More generally, after m such steps, the probability that the input size decreases by a factor of $3/4$ is $1 - \frac{1}{2}^m$. Thus the probability that the input size decreases by a factor of $3/4$ approaches 1 quickly. For example if $m = 10$ then this probability is 0.999. Thus we can conclude that quicksort behaves like a balanced divide and conquer algorithm and it should complete after $c \log n$ levels for some constant c.

We now make this intuition more precise. There are many methods of analysis that we can use. In the rest of this section, we consider one in detail, which is based on counting, and outline another, which is based establishing a recurrence, which can then be solved.

For the analysis, we assume a priority-based selection technique for pivots. At the start of the algorithm, we assign each key a random priority uniformly at random from the real interval $[0, 1]$ such that each key has a unique priority. We then pick in Line 5 the key with the highest priority. Notice that once the priorities are decided, the algorithm is completely deterministic. In addition, we assume a version of quicksort that compares the pivot p to each key in S once (instead of 3 times, once to generate each of a_1, a_2, and a_3).

Exercise 11.9. Rewrite the quicksort algorithm so to use the comparison once when comparing the pivot with each key at a recursive call.

Example 11.10. Quicksort with priorities and its call tree, which is a binary-search-tree, illustrated.



Exercise 11.11. Convince yourself that the two presentations of randomized quicksort are fully equivalent (modulo the technical details about how we might store the priority values).

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.
- Since, the pivot is not sent as part of the input to a recursive call, a key is selected to be a pivot at most once.
- Each key is selected to be pivot.

Based on these observations, we conclude that each pair of keys is compared at most once. [1]

Expected work for Quicksort. We are now ready to analyze the expected work of randomized quicksort by counting how many comparisons `quicksort` it makes in expectation. We introduce a random variable

$$Y(n) \quad = \quad \text{number of comparisons } \texttt{quicksort} \text{ makes on input of size } n,$$

[1] We need only the first two observations to establish this conclusion.

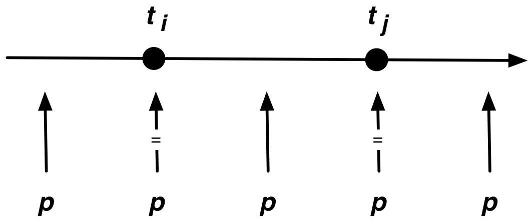


Figure 11.2: The possible relationships between the selected pivot p, t_i and t_j illustrated.

and we are interested in finding an upper bound on $\mathbf{E}\left[Y(n)\right]$. In particular we will show it is in $O(n \log n)$. $\mathbf{E}\left[Y(n)\right]$ will not depend on the order of the input sequence.

Consider the final sorted order of the keys $t = \texttt{sort}(a)$ and let p_i be the priority we chose for the element t_i. Consider two positions $i, j \in \{1, \ldots, n\}$ in the sequence t and define following random variable

$$X_{ij} \quad = \quad \begin{cases} 1 & \text{if } t_i \text{ and } t_j \text{ are compared by quicksort} \\ 0 & \text{otherwise} \end{cases}$$

Since in any run of quicksort, each pair of keys is compared at most once, $Y(n)$ is equal to the sum of all X_{ij}'s, i.e.,

$$Y(n) \quad \leq \quad \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$$

Note that we only need to consider the case that $i < j$ since we only want to count each comparison once. By linearity of expectation, we have

$$\mathbf{E}\left[Y(n)\right] \leq \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbf{E}\left[X_{ij}\right]$$

Since each X_{ij} is an indicator random variable, $\mathbf{E}\left[X_{ij}\right] = \mathbf{Pr}\left[X_{ij} = 1\right]$.

To compute the probability that t_i and t_j are compared (i.e., $\mathbf{Pr}\left[X_{ij} = 1\right]$), let's take a closer look at the quicksort algorithm to gather some intuitions. Notice that the first recursive call takes as its pivot p the element with highest priority. Then, it splits the sequence into two parts, one with keys larger than p and the other with keys smaller than p. For each of these parts, we run `quicksort` recursively; therefore, inside it, the algorithm will pick the highest priority element as the pivot, which is then used to split the sequence further.

For any one call to `quicksort` there are three possibilities (illustrated in Figure 11.2) for X_{ij}, where $i < j$:

- The pivot (highest priority element) is either t_i or t_j, in which case t_i and t_j are compared and $X_{ij} = 1$.

- The pivot is element between t_i and t_j, in which 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 $X_{ij} = 0$.
- The pivot is less than t_i or greater than t_j. Then 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`.

With this intuition in mind, we can establish the following claim.

Claim 11.12. For $i < j$, t_i and t_j are compared if and only if p_i or p_j has the highest priority among $\{p_i, p_{i+1}, \ldots, p_j\}$.

Proof. Assume first that t_i (t_j) has the highest priority. In this case, all the elements in the subsequence $t_i \ldots t_j$ will move together in the call 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. □

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,

$$\begin{aligned}
\mathbf{E}[X_{ij}] &= \mathbf{Pr}[X_{ij} = 1] \\
&= \mathbf{Pr}[p_i \text{ or } p_j \text{ is the maximum among } \{p_i, \ldots, p_j\}] \\
&= \frac{2}{j - i + 1}.
\end{aligned}$$

The bound indicates that the closer two keys are in the sorted order (t) the more likely it is that they are compared. For example, the keys t_i is compared to t_{i+1} with probability 1. It is easy to understand why if we consider the corresponding pivot tree. One of t_i and t_{i+1} must be an ancestor of the other in the pivot tree: 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.

Hence, the expected number of comparisons made in randomized `quicksort` is

$$\begin{aligned}
\mathbf{E}\left[Y(n)\right] &\le \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbf{E}\left[X_{ij}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \\
&\le 2n \sum_{i=1}^{n-1} H_n \\
&= 2nH_n \in O(n \log n).
\end{aligned}$$

Note that in the derivation of the asymptotic bound, we used the fact that $H_n = \ln n + O(1)$.

Indirectly, we have also shown that the average work for the basic deterministic `quicksort` (always pick the first element) is also $O(n \log n)$. Just shuffle the data 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 $O(n \log n)$ on average.

Expected span of Quicksort. 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 $O(\log n)$—needed for the filter. We argue that the depth of the pivot tree is $O(\log n)$ by relating it to the number of contraction steps of the randomized `select` we considered in Section **??**. We refer to 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.

Claim 11.13. 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 the say that the distribution of pivots selected along the path and the sizes of each problem is identical.

The reason this is true, is 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 that $10 \lg n$ is tiny. 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 $10 \lg n$, which is $1/n^{3.5}$. The total probability that any of the n leaves have depth larger than $10 \lg n$ is

$$\mathbf{Pr}\left[\text{depth of quicksort pivot tree} > 10 \lg n\right] \leq \frac{n}{n^{3.15}} = \frac{1}{n^{2.15}}.$$

We thus have our high probability bound on the depth of the pivot tree.

The overall span of randomized `quicksort` is therefore $O(\log^2 n)$ with high probability. As in `select`, we can establish an expected bound by using the total expectation theorem. We leave this as an exercise to the reader.

Alternative Analysis. 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:

$$Y(n) \quad = \quad Y(X(n)) + Y(n - X(n) - 1) + n - 1$$

where the random variable $Y(n)$ is the size of the set a_1 (we use $X(n)$ instead of Y_n to avoid double subscripts). We can now write an equation for the expectation of $X(n)$.

$$\begin{aligned}
\mathbf{E}\left[Y(n)\right] &= \mathbf{E}\left[Y(X(n)) + Y(n - X(n) - 1) + n - 1\right] \\
&= \mathbf{E}\left[Y(X(n))\right] + \mathbf{E}\left[Y(n - X(n) - 1)\right] + n - 1 \\
&= \frac{1}{n}\sum_{i=0}^{n-1}\left(\mathbf{E}\left[Y(i)\right] + \mathbf{E}\left[Y(n-i-1)\right]\right) + n - 1
\end{aligned}$$

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.

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, respectfully. Let the random variable $X(n) = \max\{|a_1|, |a_2|\}$, 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

$$S(n) = S(X(n)) + O(\log n).$$

For the analysis, we shall condition the span on the random variable denoting the size of the maximum half and apply the total expectation theorem.

$$\mathbf{E}\left[S(n)\right] = \sum_{m=n/2}^{n} \mathbf{Pr}\left[X(n) = m\right] \cdot \mathbf{E}\left[S(n) \mid (X(n) = m)\right].$$

The rest is algebra

$$\begin{aligned}
\mathbf{E}\left[a_n\right] &= \sum_{m=n/2}^{n} \mathbf{Pr}\left[M(n) = m\right] \cdot \mathbf{E}\left[S(n) \mid (M(n) = m)\right] \\
&\leq \mathbf{Pr}\left[X(n) \leq \frac{3n}{4}\right] \cdot \mathbf{E}\left[S(\frac{3n}{4})\right] + \mathbf{Pr}\left[X(n) > \frac{3n}{4}\right] \cdot \mathbf{E}\left[S(n)\right] + c \cdot \log n \\
&\leq \frac{1}{2}\,\mathbf{E}\left[S(\frac{3n}{4})\right] + \frac{1}{2}\,\mathbf{E}\left[S(n)\right] \\
&\implies \mathbf{E}\left[S(n)\right] \leq \mathbf{E}\left[S(\frac{3n}{4})\right] + 2c\log n.
\end{aligned}$$

This is a recursion in $\mathbf{E}\left[S(\cdot)\right]$ and solves easily to $\mathbf{E}\left[S(n)\right] = O(\log^2 n)$.

Remark 11.14. Quicksort 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 the most famous researchers in probability theory. The analysis we will cover is different from what Hoare used in his original paper, although we will mention how he did the analysis. It is interesting that while Quicksort is often used as an quintessential example of a recursive algorithm, at the time, no programming language supported recursion and Hoare spent significant space in his paper explaining how to simulate recursion with a stack.

We note that our presentation of quicksort algorithm shown in Algorithm 11.7 differs from Hoare's original version which sequentially partitioned the input by using two fingers that moved 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.

.

Chapter 12

Binary Search Trees

Searching is one of the most important operations in computer science. Of the many search data structures that have been designed and are used in practice, search trees, more specifically balanced binary search trees, occupy a coveted place because of their broad applicability to many different sorts of problems. For example, in this book, we rely on binary search trees to implement set and table abstract data types (Chapter 13), which are then used in the implementation of many algorithms, including for example graph algorithms.

If we are interested in searching a static or unchanging collection of elements, then we can use a simpler data structure such as sequences. For example, we can use a sequence with the array-based cost specification to implement an efficient search function by representing the collection as a sorted sequence and by using binary search. Such an implementation would yield a logarithmic-work search operation. If, however, we want to support dynamic collections, where for example, we insert new elements and delete existing elements, sequences would require linear work. Binary search trees, or ***BSTs*** for short, make it possible to compute with dynamic collections by using insertions, deletions, as well as searches all in logarithmic number of tree 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 update operations, such as union and difference, which can be used to insert and delete (respectively) many elements at once.

The rest of this chapter is organized as follows. We first define binary search trees (Section 12.1) and present an ADT for them (Section 12.2). We then present a parametric implementation of the ADT (Section 12.4) by using only two operations, `split` and `join`, which respectively split a tree at a given key and join two trees. In Section 12.5, we present a cost specification based on the parametric implementation, which achieves strong bounds as long as the `split` and `join` operations have logarithmic work and span. As a result, we are able to reduce the problem of implementing the BST ADT to the problem of implementing just the functions `split` and `join`. We finish the chapter by presenting a specific instance of the parametric implementation using Treaps (Section 12.6). Other possible implementation techniques are described in Section 12.3

12.1 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 2.18). A ***full binary tree*** is a rooted tree, where each node is either a ***leaf***, which has no children, or an ***internal node***, which have a left and a right child (Definition 12.1).

Definition 12.1. [Full Binary Tree] A ***full binary tree*** is an ordered rooted tree in which every internal node has exactly two children: the first or the ***left child*** and the second or the ***right child***. The ***left subtree*** of a node is the subtree rooted at the left child, and the ***right subtree*** the one rooted at the right child.

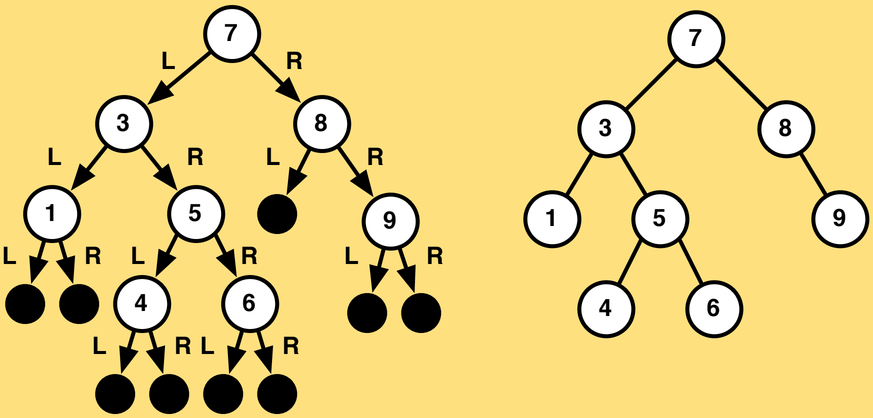
A binary search tree is a full binary tree, where each internal node u has a unique key k such that each node in its left subtree has a key less than k and each node in its right subtree has a key greater that x. Formally, we can define binary search trees as follows.

Definition 12.2. [Binary Search Tree (BST)] A ***binary search tree*** (BST) over a totally ordered set S is a full binary tree that satisfies the following conditions.

1. There is a one-to-one mapping $k(v)$ from internal tree nodes to elements in S.
2. for every u in the left subtree of v, $k(u) < k(v)$
3. for every u in the right subtree of v, $k(u) > k(v)$

In the definition, conditions 2 and 3 are referred to as the ***BST property***. We often refer to the elements of S in a BST as keys, and use $\mathsf{dom}(T)$ to indicate the domain (keys) in a BST T. The ***size*** of a BST is the number of keys in the tree, i.e. $|S|$.

Example 12.3. An example binary search tree over the set of natural numbers $\{1, 3, 4, 5, 6, 7, 8, 9\}$ is shown below.



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—for every node, the keys in the left subtree are less, and the ones in the right subtree are greater.

In the illustration on the left, the edges are oriented away from the root. They could have also been oriented towards the root. When illustrating binary search trees, we usually replace the directed arcs with undirected edges, leaving the orientation to be implicit. We also draw the left and right subtrees of a node on its left and right respectively. Following this convention, we can draw the tree on the left above as shown an the right. We use this convention in future figures.

12.2 The BST Abstract Data Type

ADT 12.4 describes an ADT for BSTs parametrized by a totally ordered key set. We briefly describe this ADT and present some examples. As we shall see, the BST ADT can be implemented in many ways. In order to present concrete examples, we assume an implementation but do not specify it.

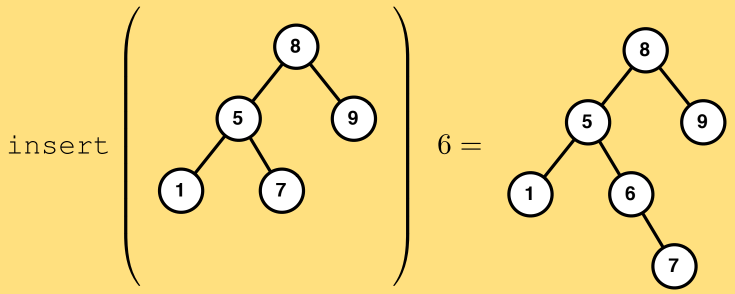
The ADT supports the constructor operations `empty` and `singleton` for creating an empty BST and BST with a single key. The function `find` searches for a given key and returns a boolean indicating success.

The functions `insert` and `delete` insert and delete a given key into or from the BST.

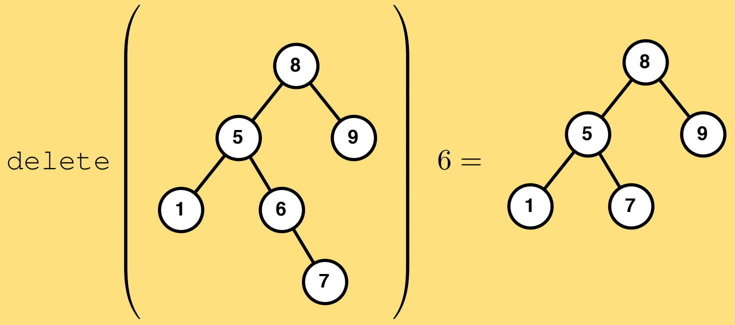
Abstract Data Type 12.4. [BST] For a universe of totally ordered keys $\mathbb{K}$, the BST ADT consists of a type $\mathbb{T}$ representing a power set of keys and the functions specified as follows. In the specification, $[\![T]\!]$ denotes the set of keys in the tree T.

`empty`	:	$\mathbb{T}$
`singleton`	:	$\mathbb{K} \to \mathbb{T}$
`find`	:	$\mathbb{T} \to \mathbb{K} \to \mathbb{B}$
`delete`	:	$\mathbb{T} \to \mathbb{K} \to \mathbb{T}$
`insert`	:	$\mathbb{T} \to \mathbb{K} \to \mathbb{T}$
`intersection`	:	$\mathbb{T} \to \mathbb{T} \to \mathbb{T}$
`difference`	:	$\mathbb{T} \to \mathbb{T} \to \mathbb{T}$
`union`	:	$\mathbb{T} \to \mathbb{T} \to \mathbb{T}$
`split`	:	$\mathbb{T} \to \mathbb{K} \to (\mathbb{T} \times \mathbb{B} \times \mathbb{T})$
`join`	:	$\mathbb{T} \to \mathbb{T} \to \mathbb{T}$

Example 12.5. Inserting the key 6 into the input tree returns a new tree including 6.



Example 12.6. Deleting the key 6 from the input tree returns a tree without it.



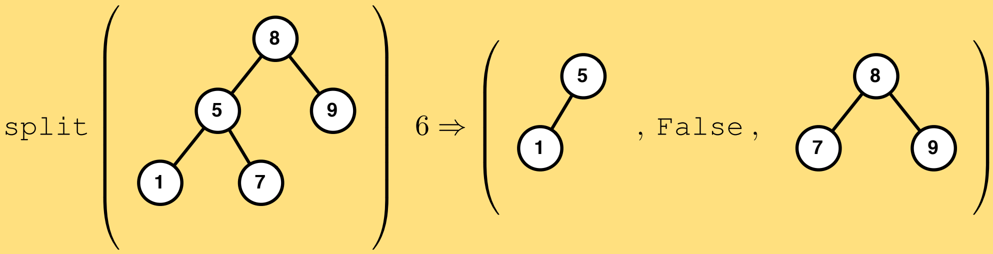
The function `union` takes two BSTs and returns a BST that contains all the keys in them; `union` is an aggregate insert operation. The function `intersection` takes two BSTs and returns a BST that contains the keys common in both. The function `difference` takes two BSTs t_1 and

t_2 and returns a BST that contains the keys in t_1 that are not in t_2; `difference` is an aggregate delete operation.

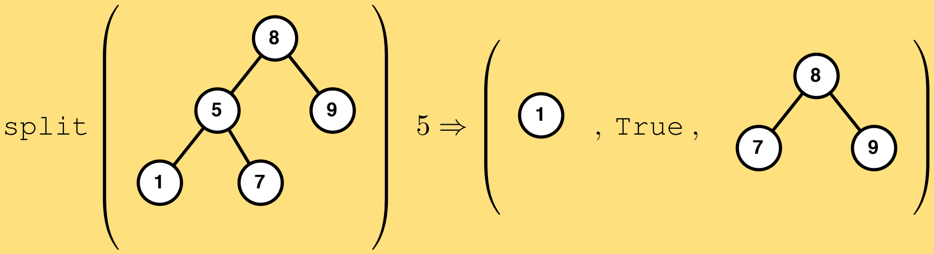
The function `split` takes a tree t and a key k and splits t into two trees: one consisting of all the keys of t less than k, and another consisting of all the keys of t greater than k. It also returns a Boolean value indicating whether k appears in t. The exact structure of the trees returned by `split` 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 12.7. The function `split` illustrated.

- Splitting the input tree at 6 yields two trees, consisting of the keys less that 6 and those greater that 6, indicating also that 6 is not in the input tree.

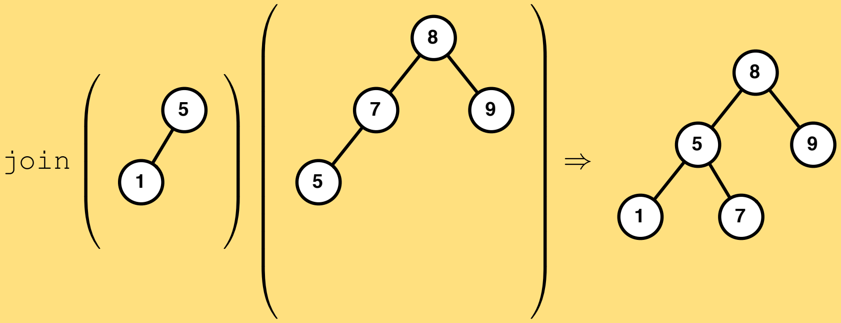


- Splitting the input tree at 5 yields two trees, consisting of the keys less than 5 and those greater than 5, indicating also that 5 is found in the input tree.



The function `join` 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 `join` 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 joined.

Example 12.8. The function join illustrated.



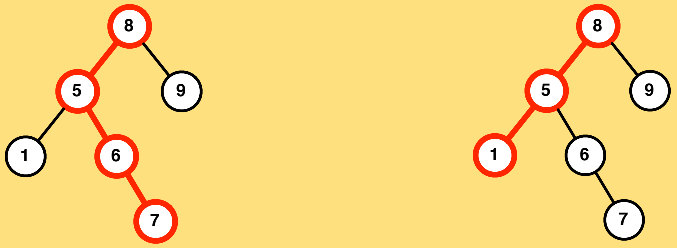
12.3 Implementation via Balancing

The main idea behind the implementation of BSTs is to organize the keys such that

1. a specific key can be located by following a branch in the tree, performing key comparisons along the way, and
2. a set of keys that constitute a contiguous range in a sorted order of keys in the tree can be moved as a chunk by performing constant work.

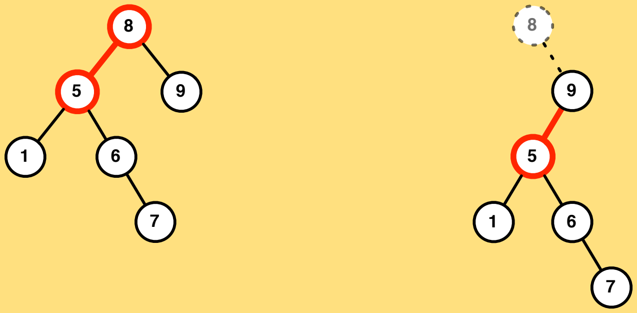
To see how we can search in a tree, consider searching for a key k in a tree t whose root is r. We can start at the root r and if k equals the key at the root, $k(r)$, then we have found our key, otherwise if $k < k(r)$, then we know that k cannot appear in the right subtree, so we only need to search the left subtree, and if $k > k(r)$, 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. In both cases we have followed a single path through the BST starting at the root.

Example 12.9. A successful search for 7 and an unsuccessful search for 4 in the given tree. Search paths are highlighted.



To see how we can operate on a range of keys, note first that each subtree in a binary tree contains all the keys that are within a specific range. We can find such a range by performing a search as described—in fact, a search as described identifies a possibly empty range. Once we find a range of keys, we can operate on them as a group by handling their root. For example, we can move the whole subtree to another location by linking the root to another parent.

Example 12.10. Consider the tree shows below on the left, we can handle all the keys that are less than 8 by holding the subtree rooted at 5, the left child of 8. For example, we can make 5 the left child of 9 and delete 8 from the tree. Note that if 8 remains in the tree, the resulting tree would not be a valid BST.



By finding a range of keys by traversing a path in the BST, and by moving ranges with constant work, it turns out to be possible to implement all the operations in the BST ADT efficiently as long as the paths traversed are not too long. One way to guarantee absence of long paths is to make sure that the tree remains balanced, i.e., the longest paths have approximately the same length. A binary tree is defined to be ***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. We refer to a BST data structure as *nearly balanced* or simply as *balanced* if all trees with n elements have height $O(\lg n)$, perhaps in expectation or with high probability.

There are many balanced BST data structures. 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 data structures:

1. ***AVL trees*** are the earliest nearly balanced BST data structure (1962). It maintains the invariant that the two children of each node differ in height by at most one, which in turn implies approximate 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 approximate balance with high-probability.

5. ***Splay trees*** are an amortized data structure that does not guarantee approximate balance, but instead guarantees that for any sequence of m insert, find and delete operations each does $O(\lg n)$ amortized work.

There are dozens of other balanced BST data structures (e.g. scapegoat trees and skip lists), as well as many that allow larger degrees, including 2–3 trees, brother trees, and B trees. In this chapter we will cover Treaps.

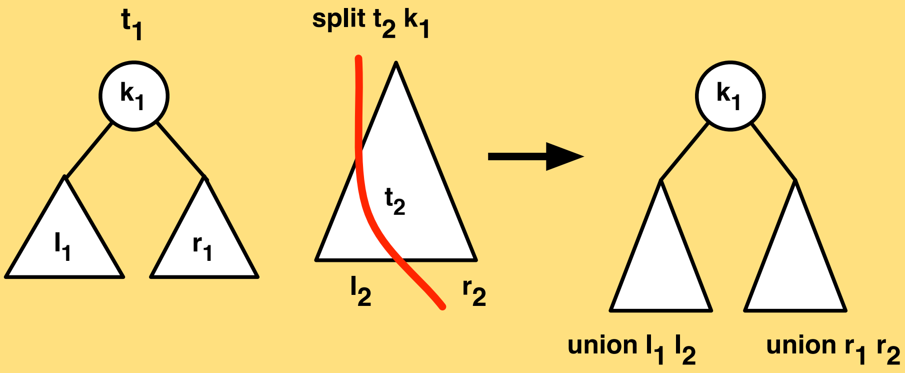
12.4 A Parametric Implementation

We describe a minimalist implementation of the BST ADT based on two functions, `split` and `join`. Since the implementation depends on just these two functions, we refer to it as a parametric implementation. Data Structure 12.11 illustrates the parametric implementation, assuming that the implementation of `split` and `join` are supplied. The implementation defines the tree type as consisting of leaves or internal nodes with left and right subtrees and a key. The auxiliary function, `joinM` takes two trees t_1 and t_2 and a "middle" key k that is sandwiched between the two trees—that is k is greater than all the keys in t_1 and less than all the keys in t_2—and returns a tree that contains all the keys in t_1 and t_2 as well as k.

The function `find` is easily implementable with a `split`, which indicates whether the key used for splitting is found in the tree or not. To implement `insert` of a key k into a tree, we first `split` the tree at k and then join the two returned trees along with key k using `joinM`. To implement `delete` of a key k from a tree, we first `split` the tree at k and then join the two returned trees with `join`. If the key k was found, this gives us a tree that does not contain the k; otherwise we obtain a tree of the same set of keys (though the structure of the tree may be different internally depending on the implementation of `split` and `join`).

The implementation of `union` uses divide and conquer. 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. There are different ways to select the key k used to split the tree. One way is to use the key at the root of one of the two trees, for example the first tree, and split the second tree with it; this is the approach take in the parametric implementation.

Example 12.12. The union of tree t_1 and t_2 illustrated.



The implementation of `intersection` uses a divide-and-conquer approach similar to that of `union`. As in `union`, we split both trees by using the key k_1 at the root of the first tree, and compute intersections recursively. We then compute 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 left and right subtrees recursively and is guaranteed to find all shared keys because the `split` operation places all keys less than and greater than the given key to two separate trees.

Exercise 12.13. Prove correct the functions `intersection`, `difference`, and `union`.

Data Structure 12.11. [Implementing the BST ADT with `split` and `join`]

```
type $\mathbb{T}$ = Leaf | Node of ($\mathbb{T} \times \mathbb{K} \times \mathbb{T}$)
split $t$ $k$ = ... (* as given *)
join $t_1$ $t_2$ = ... (* as given *)

joinM $t_1$ $k$ $t_2$ = join $t_1$ (join (singleton $k$) $t_2$)
empty = Leaf
singleton ($k$) = Node(Leaf,$k$,Leaf)
find $t$ $k$ = let (_,$v$,_) = split $t$ $k$  in $v$ end
delete $t$ $k$ = let ($l$,_,$r$) = split $t$ $k$ in join $l$ $r$ end
insert $t$ $k$ = let ($l$,_,$r$) = split $t$ $k$ in joinM $l$ $k$ $r$ end

intersect $t_1$ $t_2$ =
  case ($t_1$,$t_2$)
  | (Leaf,_) => Leaf
  | (_,Leaf) => Leaf
  | (Node ($l_1$,$k_1$,$r_1$),_) =>
    let ($l_2$,$b$,$r_2$) = split $t_2$ $k_1$
        ($l$,$r$) = (intersect $l_1$ $l_2$) || (intersect $r_1$ $r_2$)
    in if $b$ then joinM $l$ $k_1$ $r$ else join $l$ $r$ end

difference $t_1$ $t_2$ =
  case ($t_1$, $t_2$)
  | (Leaf,_) => Leaf
  | (_,Leaf) => $t_1$
  | (Node ($l_1$,$k_1$,$r_1$),_) =>
    let ($l_2$,$b$,$r_2$) = split $t_2$ $k_1$
         ($l$,$r$) = (difference $l_1$ $l_2$) || (difference $r_1$ $r_2$)
    in if $b$ then join $l$ $r$ else joinM $L$ $k_1$ $r$ end

union $t_1$ $t_2$ =
  case ($t_1$,$t_2$)
  | (Leaf,_) => $t_2$
  | (_,Leaf) => $t_1$
  | (Node ($l_1$,$k_1$,$r_1$),_) =>
    let ($l_2$,_,$r_2$) = split $t_2$ $k_1$
         ($l$,$r$) = (union $l_1$ $l_2$) || (union $r_1$ $r_2$)
    in joinM $l$ $k_1$ $r$ end
```

Cost Specification 12.14. [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.

	Work	*Span*
`empty`	$O(1)$	$O(1)$
`singleton` k	$O(1)$	$O(1)$
`split` $t\ k$	$O(\lg \lvert t \rvert)$	$O(\lg \lvert t \rvert)$
`join` $t_1\ t_2$	$O(\lg(\lvert t_1 \rvert + \lvert t_2 \rvert))$	$O(\lg(\lvert t_1 \rvert + \lvert t_2 \rvert))$
`find` $t\ k$	$O(\lg \lvert t \rvert)$	$O(\lg \lvert t \rvert)$
`insert` $t\ k$	$O(\lg \lvert t \rvert)$	$O(\lg \lvert t \rvert)$
`delete` $t\ k$	$O(\lg \lvert t \rvert)$	$O(\lg \lvert t \rvert)$
`intersect` $t_1\ t_2$	$O\left(m \cdot \lg \frac{n+m}{m}\right)$	$O(\lg n)$
`difference` $t_1\ t_2$	$O\left(m \cdot \lg \frac{n+m}{m}\right)$	$O(\lg n)$
`union` $t_1\ t_2$	$O\left(m \cdot \lg \frac{n+m}{m}\right)$	$O(\lg n)$

12.5 Cost Specification

There are many ways to implement an efficient data structure that matches our BST ADT, many of these implementation 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 12.14 shows the costs for the BST ADT as 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). As may be expected the cost of `empty` and `singleton` are constant.

For the rest of the operations, we justify the cost bounds by assuming the existence of logarithmic time `split` and `join` operations, and by using our parametric implementation described above. The work and span costs of find, insert, and delete are determined by the split and join operation and are thus logarithmic in the size of the tree.

The cost bounds on union, intersection, and difference, which are similar are more difficult to see. Let's analyze the cost for union as implemented by the parametric implementation. It is easy to apply a similar analysis to intersection and difference.

Consider now a call to union with parameters t_1 and t_2. To simplify the analysis, we will make the following assumptions:

1. t_1 is perfectly balanced (i.e., the left and right subtrees of the root have size at most $|t_1|/2$),
2. each time a key from t_1 splits t_2, it splits the tree in exactly in half, and
3. $|t_1| < |t_2|$.

Later we will relax these assumptions. Let us define $m = |t_1|$ and $n = |t_2|$ (recall the size of a tree is the number of keys in it). With these assumptions and examining the algorithm we can then write the following recurrence for the work of union:

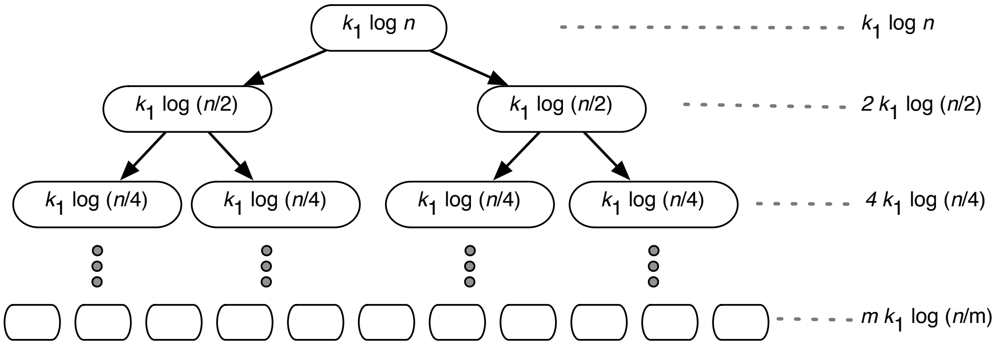
$$\begin{aligned} W_{\text{union}}(m,n) &\leq 2W_{\text{union}}(m/2,n/2) + W_{\text{split}}(n) + W_{\text{join}}(n+m) + O(1) \\ &\leq 2W_{\text{union}}(m/2,n/2) + O(\lg n) \ . \end{aligned}$$

The size for join is the sum of the two sizes, $m+n$, but since $m \leq n$, $O(\lg(n+m))$ is equivalent to $O(\lg n)$. We also have the base case

$$\begin{aligned} W_{\text{union}}(1,n) &\leq 2W_{\text{union}}(0,n/2) + W_{\text{split}}(n) + W_{\text{join}}(n) + O(1) \\ &\leq O(\lg n) \ . \end{aligned}$$

The final inequality holds because $2W_{\text{union}}(0,n) = O(1)$.

We can draw the recursion tree showing the work performed by the splitting of t_2 and by the joining of the results as follows. For simplicity of the argument, let's assume that the leaves of the tree correspond to the case for $m = 1$.



Brick method. Let's analyze the structure of the recursion tree shown above. We can find the number of leaves in the tree by examining the work recurrence. Notice that in the recurrence, the tree bottoms out when $m = 1$ and before that, m always gets split in half (remember that t_1 is perfectly balanced). The tree t_2 does not affects the shape of the recursion tree or the stopping condition. Thus, there are exactly m leaves in the tree. In fact, the recursion can be rewritten as a recursion of the form $W(m) = 2W(m/2) + \ldots$, which means that there are m leaves. By the same reasoning, we can see that the leaves are $(1 + \lg m)$ deep.

Let's now determine the size of t_2 at the leaves. We have m keys in t_1 to start with, and they split t_2 evenly all the way down to the level of the leaves (by assumption). Thus, the leaves have all the same size of $\frac{n}{m}$. Therefore, each leaf adds a $O(\lg(1 + \frac{n}{m}))$ term to the work (the $1+$ is needed to deal with the case that $n = m$). Since there are m leaves, the whole bottom level costs $O(m \lg(1 + \frac{n}{m}))$.

We will now prove that the cost at the bottom level is indeed asymptotically the same as the total work. In other words, the tree is leaves-dominated. It is possible to prove that the tree is leaves-dominated by computing the ratio of the work at adjacent levels, i.e., the ratio $\frac{2^{i-1} \lg n/2^{i-1}}{2^i \lg n/2^i} = \frac{1}{2} \frac{\lg n - i + 1}{\lg n - i}$, where $i \le \lg m < \lg n$. This ratio is less than 1 for all levels except for the last level, where by taking $i = \lg n - 1$ we have

$$\frac{1}{2} \frac{\lg n - i + 1}{\lg n - i} \le \frac{1}{2} \frac{1}{\lg n - \lg n + 1 + 1} \lg n - \lg n + 1 = \frac{1}{1}.$$

Thus the total work is asymptotically dominated by the total work of the leaves, which is $O\left(m \lg n/m\right)$.

Direct derivation. We can establish the same fact more precisely. Let's 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

$$\sum_{i=0}^{b} 2^i \lg \frac{n}{2^i} = \lg n \sum_{i=0}^{b} 2^i - \sum_{i=0}^{b} i\, 2^i. = a \sum_{i=0}^{b} 2^i - \sum_{i=0}^{b} i\, 2^i.$$

Let's now focus 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

$$
\begin{array}{rl}
= & \sum_{i=0}^{b} \left((2^{b+1}-1)-(2^i-1)\right). \\
= & (b+1)(2^{b+1}-1) - \sum_{i=0}^{b} (2^i - 1) \\
= & (b+1)(2^{b+1}-1) - \left(2^{b+1}-1-(b+1)\right) \\
= & (b+1)(2^{b+1}-1) - \left(2^{b+1}-1-(b+1)\right) \\
= & b\,2^{b+1}+1.
\end{array}
$$

Let's now go back and plug this into the original work bound and simplify

$$
\begin{array}{rcl}
W(n,m) & = & \sum_{i=0}^{b} 2^i \lg \frac{n}{2^i} \\
& = & a\sum_{i=0}^{b} 2^i - \sum_{i=0}^{b} i\,2^i \\
& = & a\,(2^{b+1}-1) - (b\,2^{b+1}+1) \\
& = & a\,2^{b+1} - a - b\,2^{b+1} - 1 = 2m(a-b) - a - 1 \\
& = & 2m(\lg n - \lg m) - a - 1 = 2m \lg \frac{n}{m} - a - 1 \\
& = & O\left(m \lg \frac{n}{m}\right).
\end{array}
$$

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.

Removing the Assumptions. Of course, in reality, our keys in t_1 won't split subtrees of t_2 in half every time. But it turns out that any unevenness in the splitting only helps reduce the work—i.e., the perfect split is the worst case. We won't go through a rigorous argument, but if we keep the assumption that t_1 is perfectly balanced, then the shape of the recursion tree stays the same. What is now different is the cost at each level. Let us try to analyze the cost at level i. At this level, there are $k = 2^i$ nodes in the recursion tree. Say the sizes of t_2 at these nodes are $n_1, \ldots, n_k$, where $\sum_j n_j = n$. Then, the total cost for this level is

$$
c \cdot \sum_{j=1}^{k} \lg(n_j) \quad \leq \quad c \cdot \sum_{j=1}^{k} \lg(n/k) = c \cdot 2^i \cdot \lg(n/2^i),
$$

where we used the fact that the logarithm function is concave[1]. Thus, the tree remains leaf-dominated and the same reasoning shows that the total work is $O(m \lg(1 + \frac{n}{m}))$.

Still, in reality, t_1 doesn't have to be perfectly balanced as we assumed. To generalize the analysis, we just need a tree with $O(\lg m)$ height. Thus, we only need t_1 to be approximately balanced.

[1] This is also known as Jensen's inequality.

Finally, we assumed that t_1 is larger that t_2. If it is smaller, then we can reverse the order of arguments, so in this case, there is no loss of generality. 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 this is all.

We end by remarking that as described, the span of `union` is $O(\lg^2 n)$, but this can be improved to $O(\lg n)$ by changing the algorithm slightly.

In summary, `union` can be implemented in $O(m \lg(1 + \frac{n}{m}))$ work and span $O(\lg n)$.

Essentially the same analysis applies to the functions `intersection` and `difference`, whose structures are the same as `union`, except for an additional constant work and span for the conditional (**if**) expression.

12.6 Treaps

Our parametric implementation established an interesting fact: to implement the BST ADT efficiently, we only need to provide efficient `split` and `join` operations. In this section, we present a data structure called *Treaps* that can support `split` and `join` operations in expected logarithmic 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 with high probability.

The idea behind Treaps is to associate a uniformly randomly selected priority to each key and maintain a priority order between keys in addition to the binary-search-tree order. The priority order between keys resemble the order used in binary heaps, leading to the name “Tree Heap” or “Treap.” We define Treaps as follows.

Definition 12.15. [Treap] A Treap is a binary search tree over a set K along with a ***priority*** for each key given by

$$p : \mathbb{K} \to \mathbb{Z} \,,$$

that in addition to satisfying the BST property on the keys K, satisfies the heap property on the priorities $p(k), k \in K$, i.e., for every internal node v with left and right children u and w:

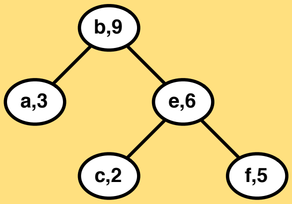
$$p(k(v)) \geq p(k(u)) \text{ and } p(k(v)) \geq p(k(w)),$$

where $k(v)$ denotes the key of a node.

Example 12.16. 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.

Exercise 12.17. Prove that if the priorities are unique, then there is exactly one tree structure that satisfies the Treap properties.

So how do we assign priorities? As we briefly suggested in the informal discussion above, it turns out that if the priorities are selected uniformly randomly then the tree is guaranteed to be near balanced, i.e. $O(\lg |S|)$ height, with high probability. We will show this shortly.

The second idea behind Treaps is to update the tree structure according to new priorities efficiently by performing local reorganizations. Based on our parametrized implementation, we can give an implementation for the BST ADT with Treaps simply by implementing the `split` and `join` functions. Data Structure 12.18 shows such an implementation. For the implementation we assume, without loss of generality, that the priorities are integers. We present only the code for `split` and `join`; the rest of the implementation is essentially the same as in Data Structure 12.11 with the only exception that since the nodes now carry priorities, we will need to account for them as we pattern match on nodes and create new ones. In implementing the rest of the functions, there is no interesting operations on priorities: they simply follow the key that they belong to.

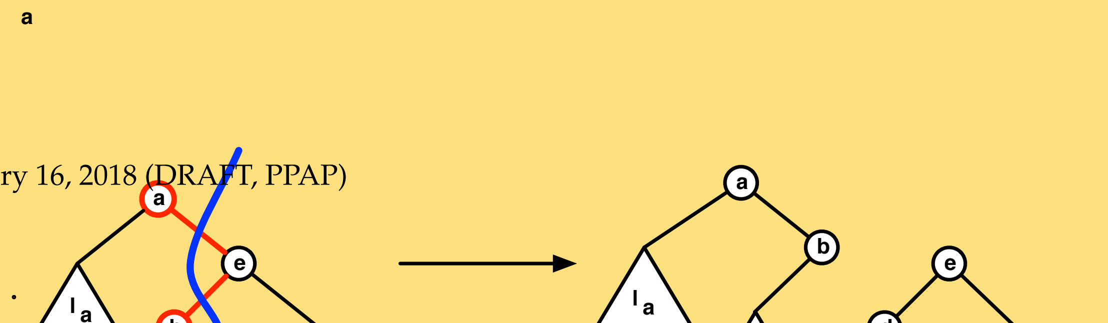
To implement the function `singleton`, we rely on a function `randomInt`, which when called returns a (pseudo-)random number. Such functions are broadly provided by programming languages.

The `split` algorithm recursively traverses the tree from the root to the key k splitting along the path, and then when returning from the recursive calls, it puts the subtrees back together. When putting back the trees along the path being split through, the function does not have to compare priorities because `Node` on Lines 14 and 18, the priority p' is the highest priority in the input tree T and is therefore larger than the priorities of either of the subtrees on the left and right. Hence `split` maintains the heap property of treaps.

Data Structure 12.18. [Implementing BST with Treaps]

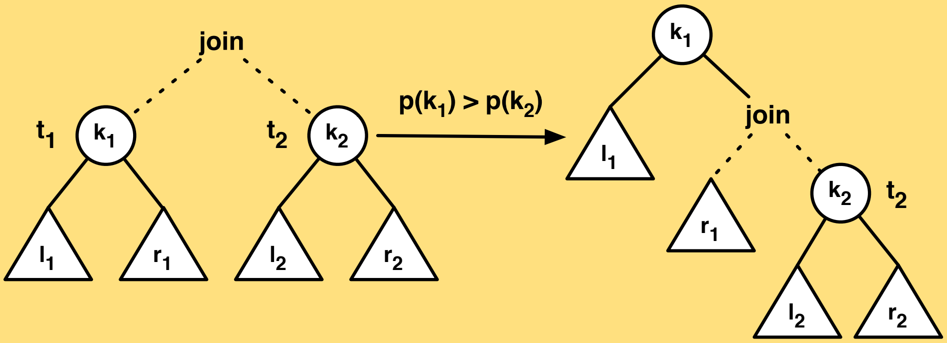
```
type 𝕋 = Leaf | Node of (𝕋 × 𝕂 × ℤ × 𝕋)

let empty = Leaf

singleton k = Node(Leaf,k,randomInt(),Leaf)

split t k =
  case t
  | Leaf => (Leaf, False, Leaf)
  | Node (l,k',p',r) =
      case compare (k,k')
      | LESS =>
          let (l',x,r') = split l k
          in (l',x,Node(r',k',p',r)) end
      | EQUAL => (l,true,r)
      | GREATER =>
          let (l',x,r') = split r k
          in (Node (l,k',p',l'),x,r') end

join t1 t2 =
  case (t1,t2)
  | (Leaf, _) => t2
  | (_ , Leaf) => t1
  | (Node (l1,k1,p1,r1), Node (l2,k2,p2,r2)) =>
      if (p1 > p2) then
        Node (l1,k1,p1, join r1 t2)
      else
        Node (join t1 l2, k2,p2,r2)
end
```

Example 12.19. A `split` operation on a Treap and key c, which is not in the Treap. The `split` traverses the path $\langle a, e, b, d \rangle$ turning right at a and b (Line 16 of the Data Structure 12.18) and turning left at e and d (Line 12). The pieces are put back together into the two resulting trees on the way back up the recursion.



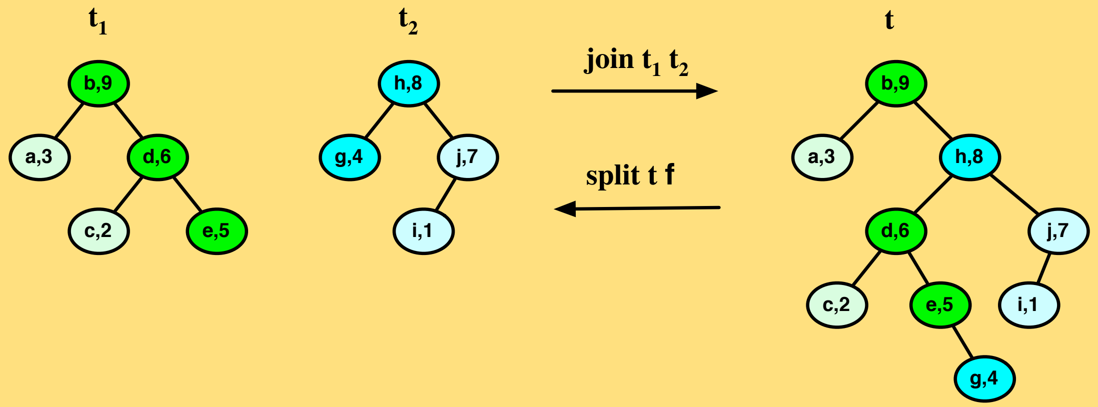
Unlike the implementation of `split`, the implementation of `join`(L, R) operates on priorities in order to ensure that the resulting Treap satisfies the heap priority of Treaps. Specifically, given two trees, `join` first compares the priorities of the two roots, making the larger priority the new root. It then recursively joins the Treaps consisting of the other tree and the appropriate side of the new root. This is illustrated in the following example:

Example 12.20. An illustration of `join`t_1 t_2 on Treaps. If $p(k_1) > p(k_2)$, then the function recurs with `join`(R_1, t_2) and the result becomes the right child of k_1.



The path from the root to the leftmost node in a BST is called the ***left spine***, and the path from the root to the rightmost node is called the ***right spine***. The function `join` t_1 t_2 merges the right spine of t_1 with the left spine of t_2 based on the priority order. This ensures that the priorities are in decreasing order down the path.

Example 12.21. An illustration of `join` for Treaps applied to t_1 and t_2 in more detail. The right spine of t_1 consisting of $(b, 9)$, $(d, 6)$ and $(e, 5)$ is merged by priority with the left spine of t_2 consisting of $(h, 8)$ and $(g, 4)$. Note that splitting the result with the key f will return the original two trees.



Let's bound now the work for `split` and `join`. Each one does constant work on each recursive call. For `split` each recursive call goes to one of the children, so the number of recursive calls is at most the height of t. For `join` each recursive call either goes down one level in t_1 or one level in t_2. Therefore the number of recursive calls is bounded by the sum of the heights of the two trees. Hence the work of `split` $t\ k$ is $O(h(t))$ and the work of `join`(t_1, m, t_2) is $O(h(t_1) + h(t_2))$. Thus all that is left to do is to bound the height of a Treap.

Analysis of randomized Treaps. We can analyze the height of a Treap by relating them to quicksort, which we analyzed in Chapter 11. In particular consider the following variant of quicksort.

Algorithm 12.22. Treap Generating Quicksort

```
qsTree $a$ =
  if $|a| = 0$ then Leaf
  else let
    $x$ = the key $k \in a$ for which $p(k)$ is the largest
    $a_1$ = $\langle\, y \in a \mid y < x \,\rangle$
    $a_2$ = $\langle\, y \in a \mid y > x \,\rangle$
    $(l, r)$ = (qsTree $a_1$ ) || (qsTree $a_2$)
  in
    Node $(l, x, r)$
  end
```

This algorithm is almost identical to our previous quicksort except that it uses `Node` instead of `append` on Line 9, `Leaf` instead of an empty sequence in the base case, and, since it is generating a set, it needs only keep one copy of the keys equal to the pivot.

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. In Chapter 11, we proved that if we pick the priorities at random, the recursion depth is $O(\lg n)$ with high probability. Therefore we know that the height of a Treap is $O(\lg n)$ with high probability.

12.7 Augmenting Trees

Thus far in this chapter, the only interesting information that we stored in BSTs were keys. While such trees can be useful, we sometimes wish to augment trees with more information. In this section, we describe how we might augment BSTs with additional information such as key-value pairs, subtree sizes, and reduced values in general.

12.7.1 Augmenting with Key-Value Pairs

Perhaps the simplest form of augmentation involves storing in the BST a key-value pair instead of just a key. Implementing BSTs augmented with key-value pairs is relatively straightforward by updating the relevant parts of the ADT. For example, to accommodate the key, we can change the BST data type to a key-value pair, and update the implementation of the functions to pass the value around with the key as needed, making sure that a key-value pair is never separated. For functions such as `find` and `split` that may return the value, we make sure to do so.

12.7.2 Augmenting with Size

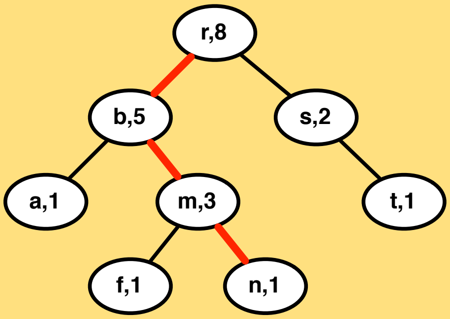
As a more complex augmentation, we might want to associate with each node in the tree a size field that tells us how large the subtree rooted at that node is. As a motivating example for this form of augmentation, suppose that we wish to extend the BST ADT (ADT 12.4) with the following additional functions.

- 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. For example, we can use them to implement the sequence interface discussed in Chapter 6.

If we have a way to count the number of nodes in a subtree, then we can easily implement these functions. Algorithm 12.23 shows such an implementation by using a size operation for computing the size of a tree, written $|t|$ for tree t. 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 12.24. 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.



Algorithm 12.23. [Rank]

```
rank $t$ $k$ =
  **case** $t$
  | Leaf => 0
  | Node $(l, k', r)$ =>
      **case** compare $(k, k')$
      | LESS => rank $l$ $k$
      | EQUAL => $|l|$
      | GREATER => $|l| + 1+$ rank $r$ $k$

select $t$ $i$ =
  **case** $t$
  | Leaf => raise exception OutOfRange
  | Node $(l, k, r)$ =>
      **case** compare $(i, |l|)$ of
        LESS => select $l$ $i$
        EQUAL => $k$
        GREATER => select $r$ $i - |l| - 1)$
```

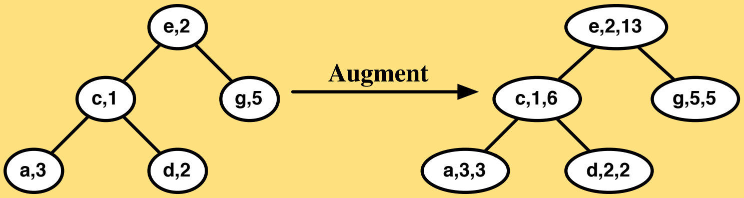
To implement a size-augmented tree, we need to keep `size` field at each node and compute the size of the nodes as they are created. In our parametric implementation, we can incorporate the `size` field by changing the definition of a node and initializing it to 1, when a singleton tree is created. When `split` and `join` functions create a new node, they can compute its size by summing the sizes of its children.

In addition to the `rank` and `select` functions, we can also define the function `splitRank`(T, i), which splits the tree into two by returning 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.

12.7.3 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 function. 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 12.25. 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 it 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 three values whenever creating a node. We can thus change a data structure to support reduced values by changing the way we create nodes. 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.

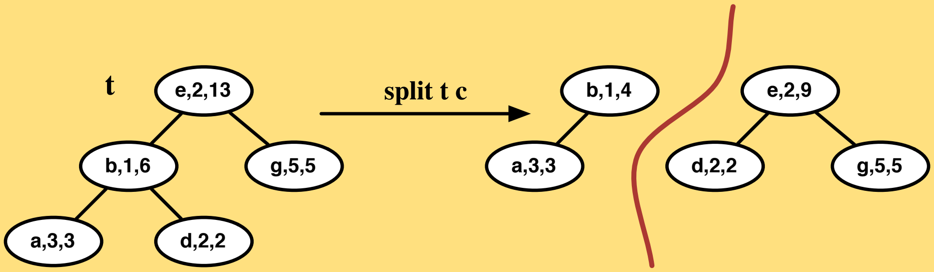
As an example, Figure 12.26 describes an extension of the parametric implementation of Treaps to support reduced values. The description is parametric in the values paired with keys and the function `f` used for reduction. The type for Treaps is extended to store the value paired with the key as well as the reduced value. Specifically, in a `Node`, the first data entry is the value paired by the key and the second is the reduced value.

To compute reduced values as the structure of the tree changes, the implementation relies on an auxiliary function `mkNode` (read "make node") that takes the key-value pair as well as the left and right subtrees and computes the reduced value by applying reducer function to the values of the left and right subtrees as well as the value. The only difference in the implementation of `split` and `join` functions from Chapter 12 is the use of `mkNode` instead of `Node`.

Data Structure 12.26. [Treaps with reduced values]

```
(* type of the reduced value, as specified. *)
type rv = ...
(* associative reducer function, as specified. *)
$f$(x: rv, y: val, z: rv): rv = ...
(* identity for the reducer function, as specified. *)
$id_f$ : rv = ...

type treap =
    Leaf
  | Node of (Treap × key × priority  × (val × rv) × Treap)

rvOf $t$ =
  case $t$
  | Leaf => $id_f$
  | Node (_,_,_(_,w),_) => w

mkNode $(l,k,p,v,r)$ = Node $(l$,$k$,$p$,$(v$,$f$ (rvOf $l$,$v$,rvOf $r$)),$r$)

split $t$ $k$ =
  case t
  | Leaf => (Leaf,false,Leaf)
  | Node $(l,k',p',(v',w'),r)$ =
      case compare $(k,k')$
      | LESS =>
          let $(l',x,r')$ = split $l$ $k$
          in $(l',x,$mkNode $(r',k',p',v',r))$ end
      | EQUAL => ($l$,true,$r$)
      | GREATER =>
          let $(l',x,r')$ = split $r$ $k$
          in (mkNode $(l,k',p',v',l'),x,r')$ end

join $t_1$ $t_2$ =
  case $(t_1,t_2)$ of
     (Leaf, _) => $t_2$
  | (_, Leaf) => $t_1$
  | (Node $(l_1,k_1,p_1,(v_1,w_1),r_1)$, Node $(l_2,k_2,p_2,(v_2,w_2),r_2)$) =>
      if $p_1$ > $p_2$) then mkNode ($l_1$,$k_1$,$p_1$,$v_1$,join $r_1$ $t_2$)
      else mkNode (join $t_1$ $l_2$,$k_2$,$v_2$,$r_2$)
```

Example 12.27. The following diagram shows an example of splitting an augmented tree.



The tree is split by the key c, and the reduced values on the internal nodes need to be updated. This only needs to happen along the path that created the split, which in this case is e, b, and d. The node for d does not have to be updated since it is a leaf. The `makeNode` for e and b are what will update the reduced values for those nodes.

We note that 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 12.28. In an imperative implementation of binary search trees, when a child node is side affected, the reduced values for the nodes on the path from the modified node to the root must be recomputed.

12.8 Problems

12-1 Insert
Design an algorithm for inserting a given key into a BST.

12-2 Delete
Design an algorithm for deleting a given key from a tree.

12-3 Minimum height
Prove that the minimum possible height of a binary search tree with n keys is $\lceil \log_2(n+1) \rceil$.

12-4 Finding Ranges
Given a BST T and two keys $k_1 \leq k_2$ return a BST T' that contains all the keys in T that fall in the range $[k_1, k_2]$.

12-5 Tree rotations
In a BST T where the root v has two children, let u and w be the left and right child of v respectively. You are asked to reorganize T. For each reorganization design a constant work and span algorithm.

- **Left rotation.** Make w the root of the tree.
- **Right rotation.** Make u the root of the tree.

12-6 Size as reduced value
Show that size information can be computed as a reduced value. What is the function to reduce over?

12-7 Implementing `splitRank`
Implement the `splitRank` function.

12-8 Implementing `select`
Implement the `select` function using `splitRank`.

Chapter 13

Sets and Tables

> "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 lead 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, using various composition rules, such as Cartesian products.

Our goals in this chapter 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, are themselves very useful, and hence deserve their own interface. In this book we refer to mappings as tables. In this chapter, we define the interface for sets and tables, specify their cost, and present some examples.

Other chapters cover data structures on binary search trees (Chapter 12) and hashing (Chapter 20) that can be used for implementing the sets and tables interfaces. We also consider many applications of sets and tables. For example, sequences (Chapter 6) are a particular type of table—one where the domain of the tables are the integers from 0 to $n-1$. Other examples include graphs (Chapter 14).

Returning to the discussion in the first paragraph about paradoxes in set theory: in our presentation we define sets as elements from particular universes, so as long as these universes are built up in a certain way, we avoid such paradoxes. However, we put an additional restriction on sets (and tables). In particular, although we allow for infinite universes (countable or uncountable) we limit ourselves to finite sized sets. This means we certainly do not capture all set theory, but the assumption makes everything computationally feasible. Representation of infinite sets is an interesting research topic but beyond the scope of this book.

When using sets in SPARC pseudo-code, we adopt some special notation for sets to simplify their expressions in comprehension.

Syntax 13.1. [Sets] We use the standard set notation $\{e_o, e_1, \cdots, e_n\}$ to indicate a set. The notation $\emptyset$ or $\{\}$ refers to an empty set. We also use the conventional mathematical syntax for set operations 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. We will also use the following shorthands:

$$\bigcup_{s \in S} s \quad \equiv \texttt{reduce union } \emptyset \quad S$$

13.1 Sets: Interface and Specification

For a universe of elements $\mathbb{U}$ (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. In the specification, $\mathbb{N}$ is the natural numbers (non-negative integers) and $\mathbb{B} = \{\texttt{T}, \texttt{F}\}$; for a set A of type $\mathbb{S}$, $[\![A]\!]$ denotes the (mathematical) set of keys in the set.

Recall that 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. (For more background on sets, please see Section 2.1.) Based on these definitions, we define our ADT for sets (ADT 13.2) by assuming that elements of the sets are drawn from a universe $\mathbb{U}$ of elements that can be tested for equality.

The Set ADT consists of basic operations on sets. The operation `size` takes a set and returns the number of elements in the set. The operation `toSeq` converts a set to a sequence by ordering the elements of the set in an unspecified way. Since elements of a set may not accept an ordering relation, the resulting order is arbitrary. This means that `toSeq` is possibly non-deterministic—it could return different orderings in different implementations, or even on different runs of the same implementation. We specify `toSeq` as follows

Abstract Data Type 13.2. [Sets]

`size`	:	$\mathbb{S} \to \mathbb{N}$
`toSeq` A	:	$\mathbb{S} \to \mathit{Seq}$
`empty`	:	$\mathbb{S}$
`singleton`	:	$\mathbb{U} \to \mathbb{S}$
`fromSeq`	:	$\mathit{Seq} \to \mathbb{S}$
`filter`	:	$((\mathbb{U} \to \mathbb{B}) \to \mathbb{S}) \to \mathbb{S}$
`intersection`	:	$\mathbb{S} \to \mathbb{S} \to \mathbb{S}$
`difference`	:	$\mathbb{S} \to \mathbb{S} \to \mathbb{S}$
`union`	:	$\mathbb{S} \to \mathbb{S} \to \mathbb{S}$
`find`	:	$\mathbb{S} \to \mathbb{U} \to \mathbb{B}$
`delete`	:	$\mathbb{S} \to \mathbb{U} \to \mathbb{S}$
`insert`	:	$\mathbb{S} \to \mathbb{U} \to \mathbb{S}$
`iterate`	:	$(\alpha * \mathbb{U} \to \alpha) \to \alpha \to \mathbb{S} \to \alpha$
`reduce`	:	$(\mathbb{U} \times \mathbb{U} \to \mathbb{U}) \to \mathbb{U} \to \mathbb{S} \to \mathbb{U}$

$$\texttt{toSeq}\ (\{\, x_0, x_1, \ldots, x_n \,\}\ \texttt{:}\ \mathbb{S})\texttt{:}\ \mathit{seq}\ =\ \langle\, x_0, x_1, \ldots, x_n \,\rangle\texttt{.}$$

Several operations enable constructing sets. The function `empty` returns an empty set; we can specify it as follows.

$$\texttt{empty}\ \texttt{:}\ \mathbb{S}\ =\ \emptyset$$

The function `singleton` constructs a singleton set from a given element.

$$\texttt{singleton}\ (x\ \texttt{:}\ \mathbb{U})\ \texttt{:}\ \mathbb{S}\ =\ \{\,\mathrm{x}\,\}$$

The operation `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).

$$\texttt{fromSeq}\ (A\texttt{:}\ \mathit{seq})\ \texttt{:}\ \mathbb{S}\ =\ \texttt{range}\ A$$

Several operations operate on sets to produce new sets. The operation `filter` selects the elements of a sequence that satisfy a given Boolean function, i.e.,

$$\texttt{filter}\ (f\ :\ \mathbb{U} \rightarrow \mathbb{B})\ (A\ :\ \mathbb{S})\ :\ \mathbb{S}\ =\ \{\, x \in A \mid f(x) \,\}.$$

The operations `intersection`, `difference`, and `union` perform the corresponding set operation on their arguments:

$$\begin{array}{l}
\texttt{intersection}\ (A\ :\ \mathbb{S})\ (B\ :\ \mathbb{S})\ :\ \mathbb{S}\ =\ A \cap B \\
\texttt{difference}\ (A\ \ :\ \mathbb{S})\ (B\ :\ \mathbb{S})\ :\ \mathbb{S}\ =\ A \setminus B. \\
\texttt{union}\ (A\ :\ \mathbb{S})\ (B\ :\ \mathbb{S})\ :\ \mathbb{S}\ =\ A \cup B
\end{array}$$

We refer to the operations `intersection`, `difference`, and `union` as ***bulk updates***, because they allow updating with a large set of elements "in bulk."

The operations `find`, `insert`, and `delete` are singular versions of the bulk operations `intersection`, `union`, and `difference` respectively. The `find` operation checks whether an element is in a set—it is the basic membership test for sets.

$$\texttt{find}\ (A\ \ :\ \mathbb{S})\ (x\ :\ \mathbb{U})\ :\ \mathbb{B}\ =\ \begin{cases} \texttt{true} & \text{if } x \in A \\ \texttt{false} & \text{otherwise} \end{cases}$$

We can also specify the `find` operation is in terms of set intersection:

$$\texttt{find}\ (A\ :\ \mathbb{S})\ (x\ :\ \mathbb{U})\ :\ \mathbb{B}\ =\ |A \cap \{x\}| = 1.$$

The operations `delete` and `insert` delete an existing element from a set, and insert a new element into a set, respectively:

$$\begin{array}{l}
\texttt{delete}\ (A\ \ :\ \mathbb{S})\ (x\ \ :\ \mathbb{U})\ :\ \mathbb{S}\ =\ A \setminus \{x\}. \\
\texttt{insert}\ (A\ :\ \mathbb{S})\ (x\ :\ \mathbb{U})\ :\ \mathbb{S}\ =\ A \cup \{x\}
\end{array}$$

The next two operations enable aggregating over sets, via iteration and reduction. The operation `iterate` can be used to iterate over a set while accumulating information. The operation takes an initial value x, a function f, and a set A, and iterates over the elements of A in some unspecified, possibly non-deterministic, order, to produce a final value.

$$\begin{array}{ll}
\texttt{iterate} & (f : \alpha * \mathbb{U} \rightarrow \alpha) \rightarrow (\text{x}\colon \alpha)(A : \mathbb{S}) : \alpha \\
& = \begin{cases} x & \text{if } |A| = 0 \\ \texttt{iterate}\ f\ (f(v,y))\ (A \setminus \{y\}) & \text{otherwise, where } y \in A. \end{cases}
\end{array}$$

The function `iterate` computes its final result by computing a new state for each element of the set $A = \{a_0, \ldots a_n\}$

$$\begin{array}{rcl} x_0 & = & x \\ x_1 & = & f(x_0, a_0) \\ x_2 & = & f(x_1, a_1) \\ & \vdots & \\ x_n & = & f(x_{n-1}, a_n). \end{array}$$

To perform a reduction over a set, the ADT supplies the `reduce` operation. The operation takes associative binary operation f along with the left identity element for the operation id, and applies f to the elements in the set to produce the "sum" of the elements. The identity of f satisfies the constraint that $f(id, x) = x$ for all $x \in \mathbb{U}$. For a set $A = \{a_0, \ldots, a_n\}$, we can define the behavior of `reduce` inductively as follows.

$$\begin{array}{ll} \texttt{reduce} & (f : \mathbb{U} * \mathbb{U} \to \mathbb{U})\ (id : \mathbb{U})\ (A : \mathbb{S}) : \mathbb{U} \\ & = \begin{cases} id & \text{if } |A| = 0 \\ A[0] & \text{if } |A| = 1 \\ f(x, y) & \text{if } |A| = 2, \text{ and } A = \{x, y\} \\ f(\texttt{reduce}\ f\ id\ \{a_0, \ldots, a_m\} & \\ \quad \texttt{reduce}\ f\ id\ \{a_{m+1}, \ldots, a_n\} & \text{otherwise, where } m = \lceil n/2 \rceil. \end{cases} \end{array}$$

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 13.3. Convince yourself that there is no way to create an infinite sized set using the interface and with finite work.

Example 13.4. Some operations on sets:

$$\begin{array}{rcl} |\{a, b, c\}| & = & 3 \\ \{x \in \{4, 11, 2, 6\} \mid x < 7\} & = & \{4, 2, 6\} \\ \texttt{find}\ \{6, 2, 9, 11, 8\}\ 4 & = & \texttt{false} \\ \{2, 7, 8, 11\} \cup \{7, 9, 11, 14, 17\} & = & \{2, 7, 8, 9, 11, 14, 17\} \\ \texttt{toSeq}\ \{2, 7, 8, 11\} & = & \langle 8, 11, 2, 7 \rangle \\ \texttt{fromSeq}\ u\ \langle 2, 7, 2, 8, 11, 2 \rangle & = & \{8, 2, 11, 7\} \end{array}$$

Remark 13.5. You may notice that the interface does not contain a `map` function. If we try to generalize the notion of `map` from sequences, a `map` function does not make sense in the context of a set: if we interpret map to take in a collection, apply some function to each element and return a collection of the same structure. 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, which doesn't match the `map` paradigm (which traditionally preserves the structure and size).

Remark 13.6. 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.

13.2 Cost Specification for Sets

Sets can be implemented in several ways. The most common efficient ways use hashing or balanced trees. An interesting aspect of both of these implementations is that they require more that just equality on the element (or the universe $\mathbb{U}$): an implementation based on balanced search trees requires a total ordering on the elements of $\mathbb{U}$ and hence a comparison; an implementation based on hashing does not require comparison, but requires the ability to hash the elements of $\mathbb{U}$. For the purposed of the ADT, we consider the operations equality, comparison, and hashing, to be implicitly defined on the universe $\mathbb{U}$.

There are various cost tradeoffs in the implementations based on balanced binary search trees and hashing. We consider a cost model based on a balanced-tree implementation. Since a balanced tree implementation requires comparisons inside the various set operations, the cost of these comparisons affects the work and span. For the specification presented here Cost Specification 13.7, we assume that the comparison operation requires constant work and span.

Cost Specification 13.7. [Tree Sets]

The cost specification for tree-based implementation of sets follow. For the specification, we define $n = \max(|A_1|, |A_2|)$ and $m = \min(|A_1|, |A_2|)$.

	Work	*Span*
`size` A `singleton` x	1	1
`toSeq` A	$\sum_{x \in A} \|A\|$	$\lg \|A\|$
`filter` f A	$\sum_{x \in A} W(f(x))$	$\lg \|A\| + \max_{x \in A} S(f(x))$
`intersection` A_1 A_2 `difference` A_1 A_2 `union` A_1 A_2	$m \cdot \lg(1 + \frac{n}{m})$	$\lg(n + m)$
`find` A e `delete` A x `insert` A x	$\lg \|A\|$	$\lg \|A\|$

Let us 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 singleton 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 operation will require searching the tree from the root to some node. We cover such an implementation in Chapter 12.

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 when one of the sets is a single element and the other when both sets are equal length. In the first case the bulk operations are doing the same thing as the singleton operations. Indeed if we implement the singleton operations on a set A using the bulk ones, as suggested a few pages ago, 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:

$$O\left(\lg\left(1 + \frac{|A|}{1}\right)\right) = O(|A|)\,.$$

Now let's consider the second case when both sets have equal length, say n. In this case we have $m = n$ giving

$$W(n) = O\left(n \cdot \lg\left(1 + \frac{n}{n}\right)\right) = O(n).$$

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 operations in terms of the singleton ones by iteration. Since it uses iteration, however, the resulting algorithms are sequential. Furthermore, they are 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`. In designing parallel algorithms, we therefore prefer to use the bulk operations `intersection`, `difference`, and `union` instead of `find`, `delete`, and `insert` when possible.

Example 13.8. One way to convert a sequence to a set would be to insert the elements one by one, which can be coded as

$$\texttt{fromSeq}\, A = \texttt{iterate Set.insert}\ \emptyset\ A$$

However, this implementation is sequential. We can write a parallel function as follows.

$$\texttt{fromSeq}\, A = \texttt{reduce Set.union}\ \emptyset\ \langle\, \{x\} : x \in A \,\rangle$$

13.3 Tables: Interface

We now consider the table interface, which is an abstract data type for the mathematical notion of a mapping. Recall that in set theory a mapping (or function) is a set of key-value pairs in which each key only appears once in the set. Mappings allow us to associate data with keys, which is very useful in the design of algorithms. We already discussed how sequences are the special case of mapping where the keys are integers in the range $\{0, \ldots, n\}$ for some n.

Since we have an ADT for sets we could represent a table as a set of key-value pairs as in their mathematical definition, e.g., $\{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}$. However, there are many unique features of tables that warrant having a separate interface. For example, we would like the interface to ensure that each key only appears once in a table, and we would like a method for efficiently looking up a value based on a key.

Our ADT for tables is defined in ADT 13.9. We write

$$\{k_1 \mapsto v_1, k_2 \mapsto v_2, \ldots\}$$

Abstract Data Type 13.9. [Tables]
For a universe of keys $\mathbb{K}$, 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 along with functions below. Throughout the set $\mathbb{S}$ denotes the powerset of the keys $\mathbb{K}$, $\mathbb{N}$ are the natural numbers (non-negative integers), and $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$.

`size`	:	$\mathbb{T} \to \mathbb{N}$
`empty`	:	$\mathbb{T}$
`singleton`	:	$\mathbb{K} \times \mathbb{V} \to \mathbb{T}$
`tabulate`	:	$(\mathbb{K} \to \mathbb{V}) \to \mathbb{S} \to \mathbb{T}$
`map`	:	$(\mathbb{V} \to \mathbb{V}) \to \mathbb{T} \to \mathbb{T}$
`filter`	:	$(\mathbb{K} \times \mathbb{V} \to \mathbb{B}) \to \mathbb{T} \to \mathbb{T}$
`intersection`	:	$(\mathbb{V} \times \mathbb{V} \to \mathbb{V}) \to \mathbb{T} \to \mathbb{T} \to \mathbb{T}$
`union`	:	$(\mathbb{V} \times \mathbb{V} \to \mathbb{V}) \to \mathbb{T} \to \mathbb{T} \to \mathbb{T}$
`difference`	:	$\mathbb{T} \to \mathbb{T} \to \mathbb{T}$
`find`	:	$\mathbb{T} \to \mathbb{K} \to (\mathbb{V} \cup \bot)$
`delete`	:	$\mathbb{T} \to \mathbb{K} \to \mathbb{T}$
`insert`	:	$(\mathbb{V} \times \mathbb{V} \to \mathbb{V}) \to \mathbb{T} \to (\mathbb{K} \times \mathbb{V}) \to \mathbb{T}$
`restrict`	:	$\mathbb{T} \to \mathbb{S} \to \mathbb{T}$
`subtract`	:	$\mathbb{T} \to \mathbb{S} \to \mathbb{T}$

for a table that maps k_i to v_i instead of $\{(k_1, v_1), (k_2, v_2), \ldots\}$ since it allows us to distinguish between tables and a set of pairs.

The operation `size` returns the size of the table, defined as the number of key-value pairs, i.e.,

$$\texttt{size}\,(A : \mathbb{T}) : \mathbb{N} = |A|.$$

The operation `empty` generates an empty table, i.e.,

$$\texttt{empty} : \mathbb{T} = \emptyset.$$

The operation `singleton` generates an table consisting of a single key-value pair, i.e.,

$$\texttt{singleton}\,(k : \mathbb{K}, v : \mathbb{V}) : \mathbb{T} = \{k \mapsto v\}\,.$$

Larger tables can be created by using the `tabulate` operation, which takes a function and a key set and creates a table by applying the function to each element of the set, i.e.,

$$\texttt{tabulate}\,(f : \mathbb{K} \to \mathbb{V})(A : \mathbb{S}) : \mathbb{T} = \{k \mapsto f(k) : k \in A\}\,.$$

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:

$$\texttt{map}\,(f : \mathbb{V} \to \mathbb{V})\,(A : \mathbb{T}) : \mathbb{T} = \{k \mapsto f(v) : k \mapsto v \in A\}\,.$$

The function `filter` selects the key-value pairs in a table that satisfy a given function:

$$\texttt{filter}\,(f : \mathbb{K} \times \mathbb{V} \to \mathbb{B})\,(A : \mathbb{T}) : \mathbb{T} = \{(k \mapsto v) \in A \mid f(k, v)\}\,.$$

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 operation takes a ***conflict-resolution*** function f of type $\mathbb{V} \times \mathbb{V} \to \mathbb{V}$ as an argument and uses it to assign a value to the key by passing to f the existing and the new value. We specify intersection as

$$\begin{array}{ll} \texttt{intersection} & (f : \mathbb{V} \times \mathbb{V} \to \mathbb{V})\,(A_1 : \mathbb{T})\,(A_2 : \mathbb{T}) : \mathbb{T} \\ & = \{k \mapsto f(v_1, v_2) : (k \mapsto v_1) \in A_1 \cap (k \mapsto v_2) \in A_2\}\,. \end{array}$$

The function `difference` substracts one table from another by throwing away all entries in the first table whose key appears in the second.

$$\texttt{difference}\,(A_1 : \mathbb{T})\,(A_2 : \mathbb{T}) : \mathbb{T} = \{k \mapsto v : (k \mapsto v_1) \in A_1 \text{ and } (k \mapsto v_2) \notin A_2\}\,.$$

The operation `union` unions the key value pairs in two tables into a single table. As with `intersection`, the operation takes a conflict-resolution function to determine the ultimate value of a key that appears is both tables. We specify `union` in terms of the `intersection` and `difference` operations.

$$\begin{array}{rl} \texttt{union} & (A_1 : \mathbb{T})\,(A_2 : \mathbb{T}) : \mathbb{T} \\ = & (\texttt{intersection}\; f\; A_1\; A_2) \;\cup \\ & (\texttt{difference}\; A_1\; A_2) \;\cup \\ & (\texttt{difference}\; A_2\; A_1) \end{array}$$

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 ($\perp$).

$$\texttt{find}\,(A : \mathbb{T})\,(k : \mathbb{K}) : \mathbb{B} = \begin{cases} v & \text{if } k \mapsto v \in A \\ \perp & \text{otherwise} \end{cases}$$

Given a table, the function `delete` deletes a the key-value pair for a specified key from the table:

$$\texttt{delete}\,(A : \mathbb{T})\,(k : \mathbb{K}) = \left\{(k' \mapsto v') \in A \mid k \neq k'\right\}.$$

The operation `insert` inserts a key-value pair into a given table. It can be thought as a singleton version of `union` and specified as such:

$$\texttt{insert}\,(f : \mathbb{V} * \mathbb{V} \to \mathbb{V})\,(A : \mathbb{T})\,(k : \mathbb{K}, v : \mathbb{V}) : \mathbb{T} = \texttt{union}\ f\ A\,(\texttt{singleton}\,(k, v)).$$

The function `restrict` restricts the domain of the table to a given set:

$$\texttt{restrict}\,(A : \mathbb{T})\,(B : \mathit{set}) : \mathbb{T} = \{k \mapsto v \in A \mid k \in B\}.$$

A closely related function `subtract` deletes from a table the entries that belong a specified set:

$$\texttt{subtract}\,(A : \mathbb{T})\,(B : \mathit{set}) : \mathbb{T} = \{(k \mapsto v) \in A \mid k \notin B\}.$$

The functions `intersection` and `resrict` can be viewed as bulk versions of the function `find`. Similarly, the function `difference` and `subtract` can be viewed as the bulk versons of the function `delete`.

In addition to these operations, we can also provide a `collect` operation 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` operation that operate on sequences (Chapter 6) as discussed previously and then use tabulate to make a table out of this sequence. We can also implement it more directly using as follows.

Algorithm 13.10. [`collect` on Tables]

```
collect $A$ = Sequence.reduce ( Table.union Sequence.append )
                              $\{\}$
                              $\langle \{k \mapsto \langle v \rangle\} : (k,v) \in A \rangle$
```

Syntax 13.11. [Tables] In the book we will use the notation

$$\{k_1 \mapsto v_1, k_2 \mapsto v_2, \ldots, k_n \mapsto v_n\}\,,$$

to indicate a table in which each key k_i is mapped to a value v_i. We will also use the following shorthands:

$$\begin{array}{ll}
A[k] & \equiv \texttt{find}(A, k) \\
\{k \mapsto f(x) : k \mapsto x \in A\} & \equiv \texttt{map}\ f\ A \\
\{k \mapsto f(x) : k \in S\} & \equiv \texttt{tabulate}\ f\ S \\
\{(k \mapsto v) \in A \,|\, p(k,v)\} & \equiv \texttt{filter}\ p\ A \\
A \setminus C & \equiv \texttt{subtract}\ A\ C \\
A_1 \cup A_2 & \equiv \texttt{union second}\ A_1\ A_2 \\
\bigcup_{t \in S} t & \equiv \texttt{reduce (union second)}\ \emptyset\ S
\end{array}$$

where `second(a,b) = b`.

Example 13.12. Define tables A_1 and A_2 and set S as follows.

$$\begin{array}{rcl}
A_1 & = & \{a \mapsto 4, b \mapsto 11, c \mapsto 2\} \\
A_2 & = & \{b \mapsto 3, d \mapsto 5\} \\
C & = & \{3, 5, 7\}\,.
\end{array}$$

The examples below show some operations, also using our syntax.

$$\begin{array}{lrcl}
\texttt{find:} & A_1[b] & = & 11 \\
\texttt{filter:} & \{k \mapsto x \in A_1 \mid x < 7\} & = & \{a \mapsto 4, c \mapsto 2\} \\
\texttt{map:} & \{k \mapsto 3 \times v : k \mapsto v \in A_2\} & = & \{b \mapsto 9, d \mapsto 15\} \\
\texttt{tabulate:} & \left\{k \mapsto k^2 : k \in C\right\} & = & \{3 \mapsto 9, 5 \mapsto 25, 9 \mapsto 81\} \\
\texttt{union:} & A_1 \cup A_2 & = & \{a \mapsto 4, b \mapsto 3, c \mapsto 2, d \mapsto 5\} \\
\texttt{union:} & \texttt{union} + (A_1, A_2) & = & \{a \mapsto 4, b \mapsto 14, c \mapsto 2, d \mapsto 5\} \\
\texttt{subtract:} & A_1 \setminus \{b, d, e\} & = & \{a \mapsto 4, c \mapsto 2\}
\end{array}$$

13.4 Cost Specification for Tables

The costs of the table operations are very similar to sets. As with sets there is a symmetry between the three operations `restrict`, `union`, and `subtract`, and the three operations `find`, `insert`, and `delete`, respectively, where the prior three are effectively "parallel" versions of the earlier three.

Cost Specification 13.13. [Tables]

	Work	*Span*
`size` A `singleton` (k, v)	1	1
`filter` $p\ A$	$\displaystyle\sum_{(k \mapsto v) \in A} W(p(k, v))$	$\lg \lvert A \rvert + \displaystyle\max_{(k \mapsto v) \in A} S(f(k, v))$
`map` $f\ A$	$\displaystyle\sum_{(k \mapsto v) \in A} W(f(v))$	$\lg \lvert A \rvert + \displaystyle\max_{(k \mapsto v) \in A} S(f(v))$
`find` $A\ k$ `delete` $A\ k$ `insert` $f\ A\ (k, v)$	$\lg \lvert A \rvert$	$\lg \lvert A \rvert$
`intersection` $f\ A_1\ A_2$ `difference` $A_1\ A_2$ `union` $f\ A_1\ A_2$ `restrict` $A\ C$ `subtract` $A\ C$	$m \lg(1 + \frac{n}{m})$	$\lg(n + m)$

where $n = \max(|A_1|, |A_2|)$ and $m = \min(|A_1|, |A_2|)$ or $n = \max(|A|, |C|)$ and $m = \min(|A|, |C|)$ as applicable. For `union` and `intersection`, we assume that $W(f(\cdot, \cdot)) = S(f(\cdot, \cdot)) = O(1)$.

Remark 13.14. 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 13.15. Tables are similar to sets: they extend sets so that each key now carries a value. Their cost specification and implementations are also similar.

13.5 Example: Indexing and Searching a Collection

As an interesting application of the sets and tables ADT's that we have discussed, let's consider the problem of indexing a set of documents to provide fast and efficient search operations on documents. Concretely 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 operations 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 kind of interface is exactly what we use when we search documents on the web. For example, search engines from companies such as Google and Microsoft allow you to do all of these searches. 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 13.16. As a simple document collection, we can consider the tweets made by some of your friends yesterday.

```
T = ⟨ ("jack", "chess␣is␣fun"),
      ("mary", "I␣had␣fun␣in␣dance␣club␣today"),
      ("nick", "food␣at␣the␣cafeteria␣sucks"),
      ("melissa'',␣"rock climbing was a blast.''),
      ("peter'',␣"I had fun at nick's party'')
      ⟩
```

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 such a 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.

Abstract Data Type 13.17. [Document Index]

```
type word      =  string
type id        =  string
type contents  =  string
type docs
type index
mkIndex        :  id → contents sequence → index
find           :  index → word → docs
lAnd           :  docs → docs → docs
lAndNot        :  docs → docs → docs
or             :  docs → docs → docs
size           :  docs → ℕ
toSeq          :  docs → docs sequence
```

Example 13.18. For our collection of tweets, we can use ADT 13.17 to make an index of these tweets and define a function to find a word in this index as follows.

```
fw : word → docs = find (mkIndex 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 (lAnd (fw "fun") (lOr (fw "food")  (fw "rock")))
```

returns all the documents (tweets) that contain `"fun"` and either `"food"` or `"rock"`, which are ⟨`"jack", "mary"`⟩. The code,

```
size (lAndNot (fw "fun") (fw "climbing"))
```

returns the number of documents that contain ”fun” and not ”climbing”, which is 1.

We can implement this interface using sets and tables. The mkIndex function can be implemented as follows.

Algorithm 13.19.

```
mkIndex $docs$ =
let
  tagWords $id$ $doc$ = $\langle\, (w, id)$ : $w$ \in $tokens\ doc \,\rangle$
  $pairs$ = flatten $\langle$ tagWords $id\ d \mid (id, d) \in docs \,\rangle$
  $words$ = Table.collect $pairs$
in
  $\{\, w \mapsto$ Set.fromSeq $d \mid w \mapsto d \in$ words $\}$
end
```

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 identifier-word pairs. We then use Table.collect to construct a table that maps each identifier to the sequence of words. Finally, for each table entry, we convert the sequence to a set so that we can perform set operations on them.

Example 13.20.
Here is an example of how `mkIndex` works. We start by tagging the words with their document it, 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" ↦ ⟨"mary"⟩),
          ("at" ↦ ⟨"mary", "peter"⟩),
          ...
          ("fun" ↦ ⟨"jack", "mary", "sue", "peter", "john"⟩),
          ...
        }
```

Finally, for each word the sequences of document identifiers is converted to a set. Note the notation that is used to express a map over the elements of a table.

The rest of the interface can be implemented as follows:

Algorithm 13.21.

```
find T v = Table.find T v
lAnd A B = Set.intersection A B
lOr  A B = Set.union A B
lAndNot A B = Set.difference A B
size A = Set.size A
toSeq A = Set.toSeq A
```

Cost. Assuming that all tokens have a length upper bounded by a constant, the cost of `mkIndex` is dominated by the collect, which is basically a sort. The work is therefore $O(n \log n)$ and the span is $O(\log^2 n)$, assuming the words have constant length.

Note that if we perform a `size(fw "red")` the cost is only $O(\log n)$ work and span. It just involves a search and then a length.

If we perform `lAnd (fw "fun") (lOr (fw "courses") (fw "classes"))` the worst case work lAnd span are at most:

$$\begin{aligned} W &= O\left(|\texttt{fw "fun"}| + |\texttt{fw "courses"}| + |\texttt{fw "classes"}|\right) \\ S &= O\left(\log|index|\right) \end{aligned}$$

The sum of sizes is to account for the cost of the `lAnd` and `or` The actual cost could be significantly less especially if one of the sets is very small.

13.6 Ordered Sets and Tables

The set and table interfaces described so far do not include any operations that use the ordering of the elements. This allows the interfaces to be defined on types that don't have a natural ordering, which makes the interfaces to be 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, 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 13.22 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 12).

Abstract Data Type 13.22. [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 $\mathbb{U}$) along with the functions below. In the specification, $\mathbb{N}$ is the natural numbers (non-negative integers) and $\mathbb{B} = \{\mathtt{T}, \mathtt{F}\}$; for a A of type $\mathbb{S}$ $[\![A]\!]$ denotes the (mathematical) set of keys in the tree. We assume max or min of the empty set returns the special element $\bot$.

Include Set ADT (ADT 13.2)

$$
\begin{array}{lcl}
\mathtt{first} & : & \mathbb{S} \to (\mathbb{U} \cup \{\bot\}) \\
\mathtt{first}(A) & = & \min [\![A]\!] \\[1ex]
\mathtt{last} & : & \mathbb{S} \to (\mathbb{U} \cup \{\bot\}) \\
\mathtt{last}(A) & = & \max [\![A]\!] \\[1ex]
\mathtt{previous} & : & \mathbb{S} \to \mathbb{U} \to (\mathbb{U} \cup \{\bot\}) \\
\mathtt{previous}\ A\ k & = & \max \{k' \in [\![A]\!] \mid k' < k\} \\[1ex]
\mathtt{next} & : & \mathbb{S} \to \mathbb{U} \to (\mathbb{U} \cup \{\bot\}) \\
\mathtt{next}\ A\ k & = & \min \{k' \in [\![A]\!] \mid k' > k\} \\[1ex]
\mathtt{split} & : & \mathbb{S} \to \mathbb{U} \to \mathbb{S} \times \mathbb{B} \times \mathbb{S} \\
\mathtt{split}\ A\ k & = & \Big(\{k' \in [\![A]\!] \mid k' < k\}, k \overset{?}{\in} S, \{k' \in [\![A]\!] \mid k' > k\} \Big) \\[1ex]
\mathtt{join} & : & \mathbb{S} \to \mathbb{S} \to \mathbb{S} \\
\mathtt{join}\ A_1\ A_2 & = & A_1 \cup A_2 \text{ assuming } \max [\![A_1]\!] < \min [\![A_2]\!] \\[1ex]
\mathtt{getRange} & : & \mathbb{S} \to \mathbb{U} \times \mathbb{U} \to \mathbb{S} \\
\mathtt{getRange}\ A\ k_1\ k_2 & = & \{k \in [\![A]\!] \mid k_1 \leq k \leq k_2\} \\[1ex]
\mathtt{rank} & : & \mathbb{S} \to \mathbb{U} \to \mathbb{N} \\
\mathtt{rank}\ A\ k & = & |\{k' \in A \mid k' < k\}| \\[1ex]
\mathtt{select}\ A\ i & : & \mathbb{S} \to \mathbb{N} \to (\mathbb{U} \cup \{\bot\}) \\
\mathtt{select}\ A\ i & = & k \in [\![A]\!] \text{ such that } \mathtt{rank}([\![A]\!], k) = i \\
 & & \text{or } \bot \text{ if there is no such } k \\[1ex]
\mathtt{splitRank} & : & \mathbb{S} \to \mathbb{N} \to \mathbb{S} \times \mathbb{S} \\
\mathtt{splitRank}\ A\ i & = & (\{k \in [\![A]\!] \mid k < \mathtt{select}(S, i)\}, \\
 & & \ \{k \in [\![A]\!] \mid k \geq \mathtt{select}(S, i)\})
\end{array}
$$

Example 13.23. Consider the following sequence ordered lexicographically:

$$A = \{\texttt{"artie"}, \texttt{"burt"}, \texttt{"finn"}, \texttt{"mike"}, \texttt{"rachel"}, \texttt{"sam"}, \texttt{"tina"}\}$$

- first A returns ”artie”.
- next A "quinn" or next S "mike" returns "rachel".
- getRange A "blain","quinn" or getRange A "burt","mike" returns

  {"burt","finn","mike"}.

Cost specification. 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 13.24. 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 12).

Cost Specification 13.25. [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 13.7) and tables (Cost Specification 13.13) for the operations supported by sets and tables. The work and span for all the operations in ADT 13.22 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.

13.7 Ordered Tables with Augmented with Reducers

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 13.26. [Reducer-Augmented Ordered Table ADT] For a specified reducer, i.e., an associative function on values $f : \mathbb{V} \times \mathbb{V} \to \mathbb{V}$ and identity element I_f, a reducer-augmented ordered table supports all operations on ordered tables as specified in ADT 13.22 and the following operation

$$\begin{array}{lcl} \texttt{reduceVal} & : & \mathbb{T} \to \mathbb{V} \\ \texttt{reduceVal}\ A & = & \texttt{Table.reduce}\ f\ I_f\ \ A \end{array}$$

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 13.27. [Analyzing Profits at **TRAM⋆LAW**®]

Let's say that based on your expertise in algorithms you are hired as a consultant by the giant retailer **TRAM⋆LAW**®. 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, or for the week after its latest promo. 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:

$$\texttt{reduceVal}(\texttt{getRange}(T, t_1, t_2))$$

This will take $O(\lg n)$ work, which is much cheaper than 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 13.28. [A Jig with **QADSAN**®] Now in your next consulting job **QADSAN**® hires you to more efficiently support queries on the 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.

13.8 Problems

13-1 Cost of bulk operations

Show that the work of the bulk operations, `intersection`, `difference`, and `union` is $O(n+m)$, where n and m are the sum of the sizes of the inputs.

13-2 Implementing fromSeq

What is the work and span of each of the implementations of `fromSeq` in Example 13.8

13-3 Union on Different Sizes

Based on the cost specification for Sets, what is the assymptotic work and span for taking the union of two sets one of size n and the other of size $\sqrt{n}$? You can assume the comparison for sets takes constant work. Please simplify as much as possible.

13-4 Cost of Table Collect

The following code implements `Table.collect`(S).

$$\texttt{Seq.reduce (Table.union Seq.append)}\ \langle\,\rangle\ \{k \mapsto \langle\, v\,\rangle : v \in S\}$$

Derive (tight) asymptotic upper bounds on the work and span for the code in terms of $n = |S|$. You can assume the comparison function used by the table takes constant work, and that the Sequence is a array sequence. You don't need a proof.

13-5 Identifying Repeats

Implement a function `identifyRepeats` that given a sequence of integer returns a table that maps each unique element to either `ONE` or `MULTI`. For example

$$\texttt{identifyRepeats}\ \langle\, 7, 2, 4, 2, 3, 2, 1, 3\,\rangle$$

would return $\{1 \mapsto \texttt{ONE}, 2 \mapsto \texttt{MULTI}, 3 \mapsto \texttt{MULTI}, 4 \mapsto \texttt{ONE}, 7 \mapsto \texttt{ONE}\}$. What is the (tight) asymptotic upper bound for work and span for your implementation in terms of $n = |S|$.

13-6 Union on Multiple Sets

The following code takes the union of a sequence of sets:

$$\textbf{fun}\ \texttt{UnionSets}(S)\ =\ \texttt{Seq.reduce Set.Union}\ \{\}\ S$$

For example

$$\texttt{UnionSets}\ \langle\, \{5, 7\}, \{3, 1, 5, 2\}, \{2, 5, 8\}\,\rangle$$

would return the set $\{1, 2, 3, 5, 7, 8\}$. Derive (tight) asymptotic upper bounds for the work and span for `UnionSets` in terms of $n = |S|$ and $m = \sum_{x \in S} |x|$. You can assume the comparison used for the sets takes constant work. Please keep your analysis to under a page. We do not need a formal proof. An explanation based on the tree-method is sufficient.

13-7 Range query

Now lets say that **QADSAN**® also wants to support queries that given a time range return the maximum increase in stock value during that range (i.e. the largest difference between two trades in which the larger amount comes after the smaller amount). What function would you use for f to support such queries in $O(\log n)$ work.

13-8 Intervals
After your two consulting jobs, you are taking 15-451, with Professor Mulb. On a test he asks you to describe a data structure for representing an abstract data type called interval tables. An interval is a region on the real number line starting at x_l and ending at x_r, and an interval table supports the following operations on intervals:

$\texttt{insert}(A, I)$	:	$\mathbb{T} \times (\texttt{real} \times \texttt{real}) \rightarrow \mathbb{T}$	*insert interval I into table A*
$\texttt{delete}(A, I)$	:	$\mathbb{T} \times (\texttt{real} \times \texttt{real}) \rightarrow \mathbb{T}$	*delete interval I from table A*
$\texttt{count}(A, x)$	:	$\mathbb{T} \times \texttt{real} \rightarrow \texttt{int}$	*return the number of intervals crossing x in A*

How would you implement this?

13-9 Domain search
Continuing on our document indexing example discussed in Section 13.5, suppose that you wish to extend the kind of queries to include domain search. A domain search specifies a ”domain” that identifies a subset of the documents and requires performing all the queries with respect to that domain. For example, the domain search ”site:cs.cmu.edu parallel algorithms” on a search engine would search all pages that are in the ”cs.cmu.edu” domain for the keywords ”parallel algorithms”.

Part IV

Graphs

Chapter 14

Graphs and their Representation

14.1 Graphs and Relations

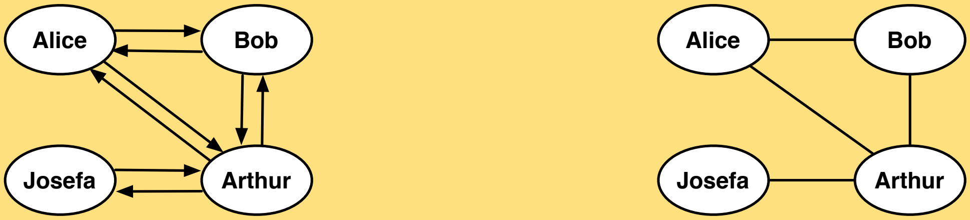
Graphs (sometimes referred to as networks) are one of the most important abstractions in computer science. What makes graphs important is that they represent relationships. As you will (or might have) discover (discovered already) relationships between things from the most abstract to the most concrete, e.g., mathematical objects, people, events, are what makes everything interesting. Considered in isolation, hardly anything is interesting. For example, considered in isolation, there would be nothing interesting about a person. It is only when you start considering his or her relationships to the world around, the person becomes interesting. Even at a biological level, what is interesting are the relationships between cells, molecules, and the biological mechanisms.

Other abstractions such as trees can also represent relationships, but only certain ones. Graphs are more interesting because they can represent any relationships—they are far more expressive. For example, in a tree, there cannot be cycles or multiple paths between two nodes. Here, what we mean by a relationship is essentially anything that we can represent abstractly by the mathematical notion of a relation. A *relation* is defined as a subset of the Cartesian product of two sets.

To represent a relation with a graph, we construct a graph, whose vertices represent the domain and the range of the relationship and connect the vertices with edges as described by the relation.

Example 14.1. You can represent the friendship relation between people as a subset of the Cartesian product of the people, e.g, {(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 each edge denotes a pair of the members of the relation of the form (a, b) and (b, a).



In some cases, it is possible to label the vertices of the graphs with natural numbers starting from 0. More precisely, an ***enumerated graph*** is a graph $G = (V, E)$ where $V = \{0, 1, \ldots, n-1\}$. As we shall see, such graphs can be more efficient to represent than general graphs, where we may not assume enumeration.

In order to be able to use graph abstractions, we need to set up some definitions and introduce some terminology. Please see Section 2.3 to review the basic definitions and concepts involving graphs. In the rest of this chapter, we assume familiarity with basic graph theory and discuss representation techniques for graphs as well as applications of graphs.

14.2 Representing Graphs

To choose an efficient and fast representation for graphs, we need to determine first the kinds of operations that we intend to support. For example we might want to perform the following operations on a graph $G = (V, E)$.

(1) Map over the vertices $v \in V$.

(2) Map over the edges $(u, v) \in E$.

(3) Map over the (in and 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 vertices.

(7) Insert or delete edges.

Representing graphs for parallel algorithms. To enable parallel algorithm design, in this book, we represent graphs by using the abstract data types that we have seen such as sequences, sets, and tables. This strategy allows us to select the best implementation (data structure) that meets the needs of the algorithm at the lowest cost. In the discussion below, we mostly consider directed graphs. To represent undirected graphs one can, for example, keep each edge in both directions, or in some cases just keep it in one direction. For the following discussion, consider a graph $G = (V, E)$ with n vertices and m edges.

Edge Sets. The simplest representation of a graph is based on its definition as a set of vertices V and a set of directed edges $E \subseteq V \times V$. If we use the set ADT, the keys for the edge set are simply pairs of vertices. The set could be implemented as a list, an array, a tree, or a hash table.

Example 14.2. In the edge-set representation, we can represent the directed graph of friends Example 14.1 by using a set of the edges:

$$\begin{array}{l} \{ \\ \quad (\text{Alice}, \text{Bob}), (\text{Alice}, \text{Arthur}), (\text{Bob}, \text{Alice}), (\text{Bob}, \text{Arthur}), \\ \quad (\text{Arthur}, \text{Josefa}), (\text{Arthur}, \text{Bob}), (\text{Arthur}, \text{Alice}), (\text{Josefa}, \text{Arthur}) \\ \}. \end{array}$$

Consider, for example, the tree-based cost specification for sets given in Chapter 13. For m edges this would allow us to determine if an arc (u, v) is in the graph with $O(\lg m)$ work using a find, and allow us to insert or delete an arc (u, v) in the same work. We note that we will often use $O(\lg n)$ instead of $O(\lg m)$, where n is the number of vertices. This is fine, because $m \leq n^2$, which means that $O(\lg m)$ implies $O(\lg n)$.

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 requires a filter based on checking if the first element of each pair matches v:

$$\{(x, y) \in E \mid v = x\}$$

For m edges this requires $\Theta(m)$ work and $O(\lg n)$ span, which is not efficient in terms of work. In fact, just about any representation of sets would require at least $O(m)$ work.

Cost Specification 14.3. [Edge Sets for Graphs] The cost for various graph operations assuming a tree-based cost model for sets. Assumes the function being mapped uses constant work and span, and that when mapping over the neighbors of a vertex, we have already found the neighbors for that vertex.

	Edge Set	
	work	*span*
$(u, v) \overset{?}{\in} G$	$O(\lg n)$	$O(\lg n)$
map over edges	$O(m)$	$O(\lg n)$
find neighbors	$O(m)$	$O(\lg n)$
map over neighbors	$O(d_G(v))$	$O(\lg n)$
degree of vertex v	$O(m)$	$O(\lg n)$

Adjacency Tables. To access neighbors more efficiently, we can use adjacency tables. The *adjacency table* representation is a table that maps every vertex to the set of its (out) neighbors. This is simply an edge-set table. In this representation, we can access efficiently the out neighbors of a vertex by performing a table lookup. Assuming the tree-based cost model for tables, this requires $O(\lg n)$ work and span.

Example 14.4. The adjacency table representation for the directed graphs representation of the friends relationship in Example 14.1 is

$$
\begin{array}{l}
\{ \\
\quad \text{Alice} \mapsto \{\text{Arthur}, \text{Bob}\}\,, \\
\quad \text{Bob} \mapsto \{\text{Alice}, \text{Arthur}\}\,, \\
\quad \text{Arthur} \mapsto \{\text{Alice}, \text{Josefa}\}\,, \\
\quad \text{Josefa} \mapsto \{\text{Arthur}\} \\
\}
\end{array}
$$

We can check if a particular arc (u, v) is in the graph by first pulling out the adjacency set for u, and then using a find operation to determine if v is in the set of neighbors. The operations thus requires $O(\lg n)$ work and span using a tree-based cost model. Similarly inserting an arc, or deleting an arc requires $O(\lg n)$ work and span. The cost of finding, inserting or deleting an edge is therefore the same as with edge sets. Note that in general, once the neighbor set has been pulled out, we can apply a constant work function over the neighbors in $O(d_G(v))$ work and $O(\lg d_G(v))$ span.

Adjacency Sequences. For enumerated graphs $G = (V, E)$, where the vertices are labeled with the natural numbers $0 \ldots (|V| - 1)$, we can use sequences to improve efficiency of the adjacency table representation by using sequences for both tables and sets. This representation allow for fast random access, requiring only $O(1)$ work to access the i^{th} element rather than $O(\lg n)$. For example, we can find the out neighbors of a vertex in $O(1)$ work and span. Certain other operations, such as subselecting vertices, however, is more expensive. Because of the reduced cost of access, we sometimes use adjacency sequences to represent a graph.

Example 14.5. We can relabel the directed graph in Example 14.1 by assigning the labels $0, 1, 2, 3$ to Alice, Arthur, Bob, Josefa respectively. We can represent the resulting enumerated graph with the following adjacency sequence:

$$
\begin{array}{l}
\langle \\
\quad \langle\, 1, 2 \,\rangle\,, \\
\quad \langle\, 0, 2, 3 \,\rangle\,, \\
\quad \langle\, 0, 1 \,\rangle\,, \\
\quad \langle\, 1 \,\rangle \\
\rangle.
\end{array}
$$

Costs. The cost of edge sets and adjacency tables is summarized with the following cost specification.

Cost Specification 14.6. [Adjacency tables and sequences] The cost for various graph operations assuming a tree-based cost model for tables and sets and an array-based cost model for sequences. Assumes the function being mapped uses constant work and span and assumes that when mapping over the neighbors of a vertex, we have already found the neighbors for that vertex.

	Adjacency Table		**Adjacency Sequence**	
	work	*span*	*work*	*span*
$(u, v) \overset{?}{\in} G$	$O(\lg n)$	$O(\lg n)$	$O(d_G(u))$	$O(\lg d_G(u))$
map over edges	$O(m)$	$O(\lg n)$	$O(m)$	$O(1)$
find neighbors	$O(\lg n)$	$O(\lg n)$	$O(1)$	$O(1)$
map over neighbors	$O(d_G(v))$	$O(\lg n)$	$O(d_G(v))$	$O(1)$
degree of vertex v	$O(\lg n)$	$O(\lg n)$	$O(1)$	$O(1)$

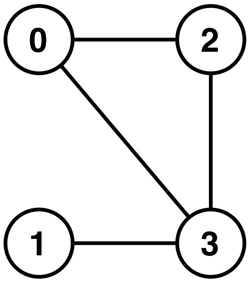


Figure 14.1: An undirected graph.

14.2.1 Traditional Representations for Graphs

Traditionally, graphs are represented by using one of the four standard representations, which we review briefly below. Of these representations, edge lists and adjacency lists can be viewed as implementations of edge sets and adjacency tables, by using lists to implement sets. For the following discussion, consider a graph $G = (V, E)$ with n vertices and m edges. As we consider different representations, we illustrate how the graph shown in Figure 14.1 is represented using each one.

Adjacency matrix. Assign a unique label from 0 to $n-1$ to each vertex and construct an $n \times n$ matrix of binary values in which location (i, j) is 1 if $(i, j) \in E$ and 0 otherwise. Note that for an undirected graph the matrix is symmetric and 0 along the diagonal. For directed graphs the 1s can be in arbitrary positions.

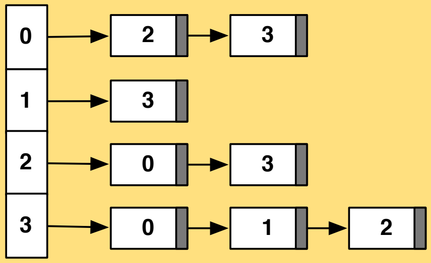
Example 14.7. Using an adjacency matrix, the graph in Figure 14.1 is represented as follows.

$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

The disadvantage of adjacency matrices is their space demand of $\Theta(n^2)$. Graphs are often sparse, with far fewer edges than $\Theta(n^2)$.

Adjacency list. Assign a unique label from 0 to $n-1$ to each vertex and construct an array A of length n where each entry $A[i]$ contains a pointer to a linked list of all the out-neighbors of vertex i. In an undirected graph with edge $\{u, v\}$ the edge will appear in the adjacency list for both u and v.

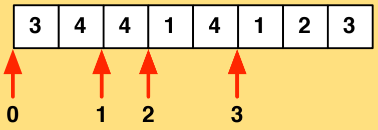
Example 14.8. Using adjacency lists, the graph Figure 14.1 is represented as follows.



Adjacency lists are not well suited for parallelism since the lists require that we traverse the neighbors of a vertex sequentially.

Adjacency array. Similar to an adjacency list, an adjacency array keeps the neighbors of all vertices, one after another, in an array `adj`; and separately, keeps an array of indices that tell us where in the `adj` array to look for the neighbors of each vertex.

Example 14.9. Using an adjacency array, the graph Figure 14.1 is represented as follows.



The disadvantage of this approach is that it is not easy to insert new edges.

Edge list. A list of pairs $(i, j) \in E$. As with adjacency lists, this representation is not good for parallelism.

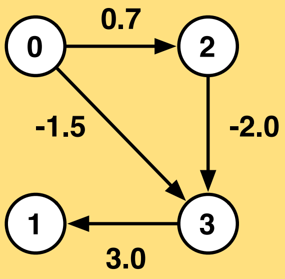
14.3 Weighted Graphs and Their Representation

Many applications of graphs require associating weights or other values with the edges of a graph. Such graphs can be defined as follows.

Definition 14.10. [Weighted and Edge-Labeled Graphs] An *edge-labeled graph* or a *weighted graph* is a triple $G = (E, V, w)$ where $w\colon E \to L$ 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 14.11. An example directed weighted graph.



We described three different representations of graphs suitable for parallel algorithms: edge sets, adjacency tables, and adjacency sequences. We can extend each of these representations to support edge-labeling by separately representing the function from edges to labels using a table (mapping) that maps each edge (or arc) to its value. This representation allows looking up the edge value of an edge $e = (u, v)$ by using a table lookup. We call this an *edge-label table*.

Example 14.12. For the weighted graph in Example 14.11, the edge-label table is:

$$W = \{(0,2) \mapsto 0.7,\ (0,3) \mapsto -1.5,\ (2,3) \mapsto -2.0,\ (3,1) \mapsto 3.0\}$$

A nice property of edge-label tables is that they work uniformly with all graph representations, and they are clean since they separates the edge labels from the structural information. However keeping a separate edge-label table creates redundancy, wasting space and possibly requiring extra work to access the edge labels. The redundancy can be avoided by storing the edge labels directly with the edge.

For example, instead of using edge sets, we can use edge-label tables (mapping edges to their values). Similarly, when using adjacency tables, we can replace each set of neighbors with a table mapping each neighbor to the label of the edge to that neighbor. Finally, we can extend an adjacency sequences by creating a sequence of neighbor-value pairs for each out edge of a vertex. This is illustrated in the following example.

Example 14.13. For the weighted graph in Example 14.11, the adjacency table representation is

$$G = \{1 \mapsto \{2 \mapsto 0.7, 3 \mapsto -1.5\}, 3 \mapsto \{3 \mapsto -2.0\}, 4 \mapsto \{1 \mapsto 3.0\}\},$$

and the adjacency sequence representation is

$$G = \langle\, \langle\, (2, 0.7),\ (3, -1.5)\,\rangle, \langle\ \rangle, \langle\, (3, -2.0)\,\rangle, \langle\, (1, 3.0)\,\rangle\,\rangle\,.$$

14.4 Applications of Graphs

Since they are powerful abstractions, graphs can be very important in modeling data. In fact, many problems can be reduced to known graph problems. Here we outline just some of the many applications of graphs.

1. *Social network graphs: to tweet or not to tweet.* Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. 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, or even who might be a good match for who, or is that whom.

2. *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 map programs such as Google maps, Bing maps and now Apple IOS 6 maps (well perhaps without the public transport) to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.

3. *Utility graphs.* The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them. 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.

4. *Document link graphs.* The best known 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.

5. *Protein-protein interactions graphs.* Vertices represent proteins and edges represent interactions between them that carry out some biological function in the cell. 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.

6. *Network packet traffic graphs.* Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them. Such graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity.

7. *Scene graphs.* In graphics and computer games scene graphs represent the logical or spacial relationships between objects in a scene. Such graphs are very important in the computer games industry.

8. *Finite element meshes.* 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. The elements along with the connections between adjacent elements forms a graph that is called a finite element mesh.

9. *Robot planning.* Vertices represent states the 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.

10. *Neural networks.* 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.

11. *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 (yes, I have no idea what that means).

12. *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.

13. *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.

14. *Graphs in compilers.* Graphs are used extensively in compilers. They can be used for type inference, for so called data flow analysis, register allocation and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.

15. *Constraint graphs.* Graphs are often used to represent constraints among items. 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.

16. *Dependence graphs.* Graphs can be used to 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.

.

Chapter 15

Graph Search

The term ***graph search*** or ***graph traversal*** refers to a class of algorithms that systematically explore the vertices and edges of a graph. Graph-search algorithms can be used to solve many interesting problems on graphs, or compute many interesting properties of graphs; they are indeed at the heart of many graph algorithms. In this chapter, we introduce the concept of a graph search, describing a general algorithm for it. We then consider further specializations of this general algorithm, specifically the breadth-first search (BFS), depth-first search (DFS), and priority-first search (PFS) algorithms, each of which is particularly useful in computing certain properties of graphs. Graph search can be applied to both directed and undirected graphs.

As an example of what graph search may be used for consider the following problem. 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. Then we have:

Problem 15.1. [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.

Another problem we may wish to solve with graph search is to determine the shortest path from a vertex to all other vertices. This is covered in the next chapter.

As the problems above suggest, a graph search usually starts at a specific vertex, and also sometimes at a set of vertices. We refer to such a vertex (vertices) as the ***source*** vertex (vertices). Graph search then searches out from this source (or these sources) and ***visits*** neighbors of vertices that have already been visited. To ensure efficiency, graph search algorithms usually keep track of the vertices that have already been visited to avoid visiting a vertex for a second time. We will refer to these as the ***visited set*** of vertices, and often denote them with the variable X. Since we only visit un-visited neighbors of the visited set it can be useful to keep track of all such vertices. We will refer to these vertices as the frontier. More formally:

Definition 15.2. For a graph $G = (V, E)$ and a visited set $X \subset V$, the ***frontier set*** is the set of un-visited out neighbors of of X, i.e. the set $N_G^+(X) \setminus X$.

Recall that $N_G^+(v)$ are the outgoing 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). We often denote the frontier set with the variable F. Assuming we start at a single source vertex s, we can now write a generic graph-search algorithm as follows.

Algorithm 15.3. [Graph Search]

```
graphSearch $(G, s)$ =
  let
    $(X, F)$ =
      start  $(\{\}, \{s\})$ and
      while  $(|F| > 0)$ do
        choose $U \subseteq F$ such that $|U| \geq 1$
        visit $U$
        $X = X \cup U$              % Add newly visited vertices to $X$
        $F = N_G^+(X) \setminus X$    % The next frontier, i.e. neighbors of $X$ not in $X$
  in $X$ 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 an iteration of the while loop. 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. Note that when visiting the set of vertices U, we can usually visit them in parallel. The algorithm terminates when there are no more vertices to visit—i.e., the frontier is empty. This does not mean that it visits all the vertices: vertices that are not reachable from the source will never be visited by the algorithm. In fact, the function `graphSearch` returns exactly the set of vertices that are reachable from the source vertex s. Hence `graphSearch` solves the reachability problem, and this is true for any choice of U on line 6. In many applications of graph search, we keep track of other information in addition to the reachable vertices.

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 techniques. In this chapter we consider three methods for selecting the subset, each leading to a specific graph-search technique. Selecting all of the vertices in the frontier leads to breadth-first search. Selecting the single most recently "seen" vertex from the frontier leads to depth-first search (when visiting a vertex we "see" all its neighbors). Selecting the highest-priority vertex (or vertices) in the frontier, by some definition of priority, leads to priority-first search. Note that breadth-first search is parallel since it can visit many vertices at once (the whole frontier), while depth-first-search is sequential since it only visits one vertex at a time.

Since graph search always visits an out-neighbor of the visited set, we can associate every visited vertex with an in-neighbor from which it was visited. When a vertex is visited there might be many in-neighbors that have already been visited, so we have to choose one. The mapping of vertices to their chosen in-neighbor defines a tree over the reachable vertices with the source s as the root. In particular the parent of each vertex is its in-neighbor that is chosen when visited. The source s has no parent and is therefore the root. This tree is referred to as the ***graph search tree***. We will see that it is useful in various applications.

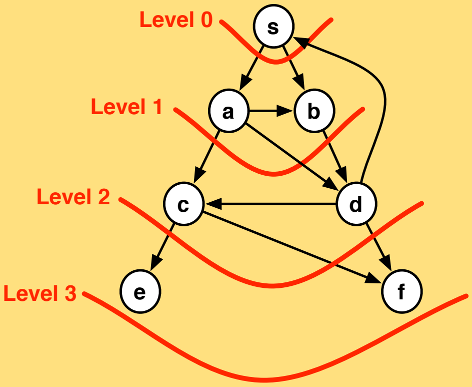
15.1 Breadth-First Search (BFS)

The ***breadth-first search*** (BFS) algorithm is a particular graph-search algorithm that can be applied to solve a variety of problems. Beyond solving the reachability problem, it can be used to find the shortest (unweighted) path from a given 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.

To understand how BFS operates, consider a graph $G = (V, E)$ and a source vertex $s \in V$. Define the ***level*** of a vertex $v \in V$ 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 $\delta_G(s, v)$ (we often drop the G in the subscript when it is clear from the context).

Breadth first search (BFS) starts at the given *source* vertex s and explores the graph outward in all directions in increasing order of levels. It first visits all the vertices that are the out-neighbors of s (i.e. have distance 1 from s, and hence are on level 1), and then continues on to visit the vertices that are on level 2 (have distance 2 from s), then level 3, and so on.

Example 15.4. A graph and its levels illustrated. BFS visits the vertices on levels 0, 1, 2, and 3 in that order.



Since f is on level 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 $\langle\, s, b, d, f \,\rangle$.

As mentioned in the introduction, the BFS-algorithm is a specialization of the graph-search algorithm (Algorithm 15.3) for which all frontier vertices are visited on each round. The algorithm can thus be defined as follows:

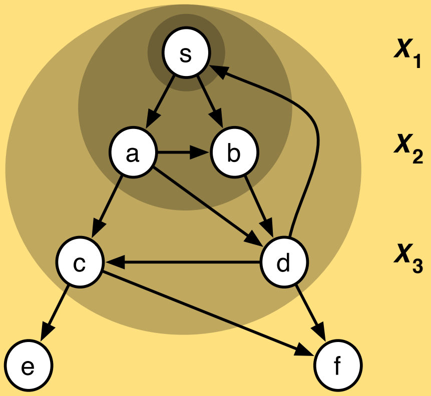
Algorithm 15.5. [BFS: reachability and radius]

```
BFSReach $(G = (V, E), s)$ =
  let
    $(X, F, i)$ =
      start $(\{\}, \{s\}, 0)$ and
      while $(|F| > 0)$
        invariant: $X = \{u \in V \mid \delta_G(s, u) < i\} \wedge$
                   $F = \{u \in V \mid \delta_G(s, u) = i\}$
        $X = X \cup F$
        $F = N_G^+(F) \setminus X$
        $i = i + 1$
  in $(X, i)$ end
```

In addition to the visited set X and frontier F maintained by the general graph search, the algorithm maintains the level i. It searches the graph level by level, starting at level 0 (the source s), and visiting one level on each round of the while loop. We refer to all the vertices visited before level (round) i as X_i. Since at level i we visit vertices at a distance i, and since we visit levels in increasing order, the vertices in X_i are exactly those with distance less than i from the source.

At the start of round i the frontier F_i contains all un-visited neighbors of X_i, which are the vertices in the graph with distance exactly i from s. In each round, the algorithm visits all the vertices in the frontier and marks newly visited vertices by adding the frontier to the visited set, i.e, $X_{i+1} = X_i \cup F_i$. To generate the next set of frontier vertices, the algorithm takes the neighborhood of F and removes any vertices that have already been visited, i.e., $F_{i+1} = N^+_G(F) \setminus X_{i+1}$. Algorithm 15.5 just keeps track of the visited set X, the frontier F, and the level i, but, as we will see, in general BFS-based algorithms can keep track of other information.

Example 15.6. 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). The neighbors $N_G(F_1)$ are the central vertex s and those in F_2. Notice that vertices in a frontier can share the same neighbors (e.g., a and b share d), which is why $N_G(F)$ is defined as the *union* of neighbors of the vertices in F to avoid duplicate vertices.



To prove that the algorithm is correct we need to prove the invariant that is stated in the algorithm.

Lemma 15.7. In `BFSReach` the invariants $X = \{v \in V_G \mid \delta_G(s,v) < i\}$ and $F = \{v \in V_G \mid \delta_G(s,v) = i\}$ are maintained.

Proof. This can be proved by induction on the level i. For the base case $i = 0$, and we have

$X_0 = \{\}$ and $F_0 = \{s\}$. This is true since 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 $i+1$. For F_{i+1}, the algorithm takes all neighbors of F_i and removes the set X_{i+1}. Since all vertices F_i have distance i from s, by assumption, then a neighbor v of F must have $\delta_G(s,v)$ of no more than $i+1$. Furthermore, all vertices of distance $i+1$ must be reachable from a vertex at distance i. Therefore, the out-neighbors of F_i contain all vertices of distance $i+1$ and only vertices of distance at most $i+1$. When removing X_{i+1} we are left with all vertices of distance $i+1$, as needed. □

To see that the algorithm returns all reachable vertices, note that if a vertex v is reachable from s, then there is a path from s to v and the vertices on that path have have distances $0, 1, 2, 3, \ldots, \delta(s,v)$. Therefore for all rounds of `BFSReach` where $i \leq \delta(s,v)$ the frontier F is non empty, and the algorithm continues to the next round, eventually reaching v on round $\delta(s,v)$, which is at most $|V|$. Note that the algorithm also returns i, which is the maximum distance from s to any reachable vertex in G.

Exercise 15.8. 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?

15.2 Shortest Paths and Shortest-Path Trees

Thus far we have used BFS for reachability. As you might have noticed, however, our `BFSReach` algorithm is effectively calculating the distance to each of the reachable vertices as it goes along since all vertices v that are visited on level i have $\delta(s,v) = i$. But, the algorithm does not store this information. It is relatively straightforward to extend BFS to keep track of the distances. For example the following algorithm takes a graph and a source and returns a table mapping every reachable vertex v to $\delta_G(s,v)$.

Algorithm 15.9. [BFS-based Unweighted Shorted Paths]

```
BFSDistance $(G, s)$ =
  let
    $(X, F, i)$ =
      start $(\{\}, \{s\}, 0)$ and
      while $(|F| > 0)$
        $X = X \cup \{v \mapsto i : v \in F\}$
        $F = N_G^+(F) \setminus \texttt{domain}(X)$
        $i = i + 1$
  in $(X, i)$ end
```

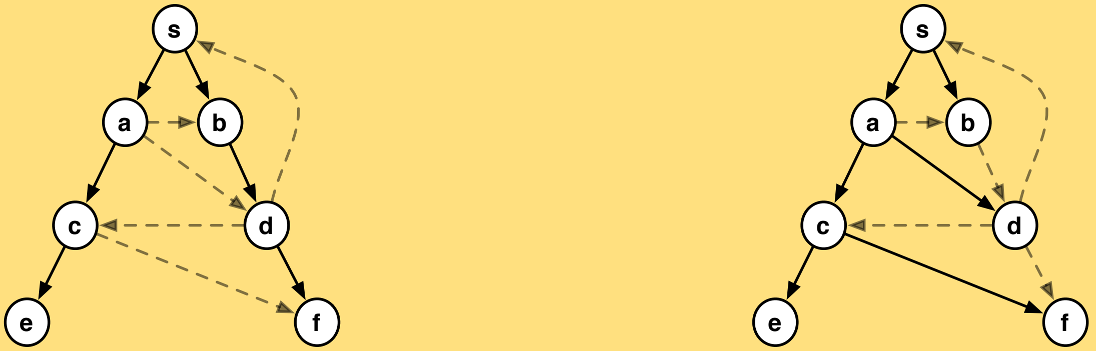
In the algorithm the table X is used both to keep track of the visited vertices and for each of these vertices to keep its distance from s. When visiting the vertices in the frontier we simply add them to X with distance i.

Sometimes in addition to the shortest distance to each vertex from s, it might be useful to know a path (i.e. the vertices along the path) which has that distance. For example if your favorite map program told you it was 2.73 miles to your destination, but did not tell you how to get there it would not be too useful. With hardly any additional work it turns out we can generate a data structure during BFS that allows us to quickly extract a shortest path from s to any reachable vertex v. The idea is to generate a graph-search tree, as defined in the introduction of this chapter—i.e., the parent of each vertex v is an in-neighbor of v that was already visited when v is visited. When such a tree is generated with BFS, the path in the tree from the root s to a vertex v is a shortest path from s to v.

To convince yourself the graph-search tree contains shortest paths, consider a vertex v visited at level i. All in-neighbors already visited when v is visited must be at level $i - 1$. This is because if they were at a lower level then v would be visited earlier (contradiction). If they are at a higher level, then they are not yet visited (contradiction again). Therefore the parent u of v in the tree must be exactly one closer to s. The parent of u must again be one closer. Therefore the path up the tree is a shortest path (at least if we assume edges are unweighted). We therefore call a graph-search tree generated by BFS the unweighted ***shortest-path tree***.

We can represent a shortest-path tree with a table mapping each reachable vertex to its parent in the tree. Given such a shortest-paths tree, we can then compute the shortest path to a particular vertex v by walking in the tree from v up to the root s, and then reversing the order of the result. Note that there could be more than one shortest-path tree, because there can be multiple paths of equal length from the source to a vertex. For example 15.10 shows two possible trees that differ while still yielding the same shortest distances.

Example 15.10. An undirected graph and two possible BFS trees with distances from s. Non-tree edges, which are edges of the graph that are not on a shortest paths are indicated by dashed lines.



Exercise 15.11. Modify Algorithm 15.9 to return a shortest-path tree represented as a table mapping each vertex to its parent. The source s can map to itself.

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 the next chapter 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.

15.3 Cost of BFS

The cost of BFS depends on the particular representation that we choose for graphs. In this section, we consider two representations, one using tree-based sets and tables, and the other using based on single-threaded array sequences. For a graph with m edges and n vertices, the first requires $O(m \log n)$ the second $O(m)$ work. The span depends on the how many rounds the while loop makes, which equals the largest distance of any reachable vertex from the source. We will refer to as d. The span with tree-based sets and tables is $O(d \log^2 n)$ (i.e. $O(\log^2 n)$ per level) and with array sequences it is $O(d \log n)$ (i.e. $O(\log n)$ per level).

When analyzing the cost of BFS with either representation, a natural method is to sum the work and span over the rounds of the algorithm, each of which correspond to a single iteration of the while loop. 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 for BFS observing that BFS visits each vertex at most once, and since the algorithm only looks at a vertices out-edges when visiting it, the algorithm also only uses every edge at most once.

Cost with BST-Sets and BST-Tables

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 $F' = N \setminus F$. 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., $\|F\| = \sum_{v \in F}(1 + d_G^+(v))$. The costs for each round are then

	Work	Span
$X \cup F$	$O(\|F\| \log n)$	$O(\log n)$
$N_G^+(F)$	$O(\\|F\\| \log n)$	$O(\log^2 n)$
$N \setminus X$	$O(\\|F\\| \log n)$	$O(\log n)$.

The first and last lines fall directly out of the tree-based cost specification for the set ADT. The second line is a bit more involved. The union of out-neighbors is implemented as

$$\textbf{let}\ N_G^+(F) = \texttt{Table.reduce Set.Union \{\} (Table.restrict}\ G\ F\texttt{)}$$

Let $G_F = \texttt{Table.restrict}\ G\ F$. The work to find G_F is $O(|F| \log n)$. For the cost of the union, note that the set union results in a set whose size is no more than the sizes of the sets unioned. The total work per level of reduce is therefore no more than $\|F\|$. Since there are $O(\log n)$ such levels, the work is bounded by

$$\begin{aligned} W\,(\texttt{Table.reduce})\ \texttt{union}\ \{\}\ G_F) &= O\left(\log |G_F| \sum_{v \mapsto N(v) \in G_F} (1 + |N(v)|)\right) \\ &= O\left(\log n \cdot \|F\|\right) \end{aligned}$$

and span is bounded by

$$S\,(\texttt{reduce union}\ \{\}\ G_F) = O(\log^2 n)$$

since each union has span $O(\log n)$ and the reduction tree is bounded by $\log n$ depth.

Focusing on a single round, we can see that the cost per vertex and edge visited in that round is $O(\log n)$. 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 $W_v = W_e = O(\log n)$. Since the total work is $W = W_v n + W_e m$ (recall that $n = |V|$ and $m = |E|$), we thus conclude that

$$\begin{aligned} W_{BFS}(n, m, d) &= O(n \log n + m \log n) \\ &= O(m \log n), \text{ and} \\ S_{BFS}(n, m, d) &= O(d \log^2 n). \end{aligned}$$

We drop the $n \log n$ term in the work since for BFS we cannot reach any more vertices than there are edges.

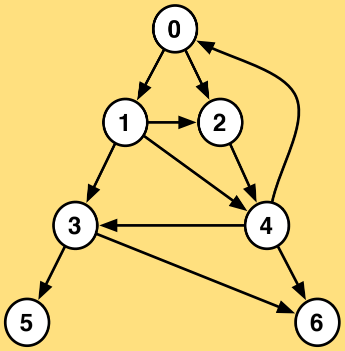
Notice that span depends on d. In the worst case $d \in \Omega(n)$ and BFS is sequential. Many real-world graphs, however, have low diameter; for such graphs BFS has good parallelism.

Cost with Single-Threaded Sequences

Consider an enumerated graph $G = (V, E)$ where $V = \{0, 1, \ldots, n-1\}$. We can represent enumerated graphs as a sequence of sequences A, where $A[i]$ is a sequence representing the out-arcs of vertex i. If the out-arcs are ordered, we can order them accordingly; if not, we can choose an arbitrary order.

Example 15.12. The enumerated graph below can be represented as

$$\langle \langle 1, 2 \rangle, \langle 2, 3, 4 \rangle, \langle 4 \rangle, \langle 5, 6 \rangle, \langle 3, 6 \rangle, \langle \ \rangle, \langle \ \rangle \rangle.$$



This representation supports constant-work lookup operations for finding the out-edges (or out-neighbors) of a vertex. Since the graph does not change during BFS, this representation suffices for implementing BFS. In addition to performing lookups in the graph, the BFS algorithm also needs to determine whether a vertex is visited or not by using the visited set X. Unlike the graph, the visited set X changes during the course of the algorithm. We therefore use a single-threaded sequence of length $|V|$ to mark which vertices have been visited. By using `inject`, we can mark vertices in constant work per update. For each vertex, 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.

The sequence-based BFS algorithm is shown in Algorithm 15.13. On entering the main loop, the sequence X contains the parents for both the visited and the frontier vertices instead of just for the visited vertices. The frontier F is represented a sequence of vertices. Each iteration of the loop starts by visiting the vertices in the frontier (in this generic algorithm this includes no computation). Next, it computes the sequence N of the neighbors of the vertices in the frontier, and updates the visited set X to map the vertices in the next frontier to their parents. Note that a vertex in N can have several in-neighbors from F; the function `inject` selects one of these as parent. The iteration completes by computing the next frontier. Since `inject` guarantees a single parent, each vertex is included at most once.

Algorithm 15.13. [BFS Tree]

```
BFSTree $(G, s)$ =
  let
    $X_0$ = STSeq.fromSeq $\langle \texttt{None} : v \in \langle 0, \ldots, |G| - 1 \rangle \rangle$
    $X$ = STSeq.update $X_0$ $(s, \texttt{Some } s)$

    (* Perform BFS. *)
    $(X, F)$ =
      start $(X, \langle s \rangle)$ and
      while $(|F| > 0)$
        (* Visit $F$ *)
        (* Compute next frontier and update visited. *)
        $N =$ Seq.flatten $\langle \langle (u, \texttt{Some } v) : u \in G[v] \mid X[u] = \texttt{None} \rangle : v \in F \rangle$
        $X =$ STSeq.inject $X$ $N$
        $F = \langle u : (u, v) \in N \mid X[u] = \texttt{Some } v \rangle$
  in
    STSeq.toSeq $X$
  end
```

All the work is done in Lines 12 and 13, and Line 14. Also note that the `STSeq.inject` on Line 13 is always applied to the most recent version. We can write out the following table of costs.

	X: $STSequence$		X: $Sequence$	
Line	Work	Span	Work	Span
8	$O(\|F_i\|)$	$O(\log n)$	$O(\|F_i\|)$	$O(\log n)$
9	$O(\|F_i\|)$	$O(1)$	$O(n)$	$O(1)$
10	$O(\|F_i\|)$	$O(\log n)$	$O(\|F_i\|)$	$O(\log n)$
total across all d rounds	$O(m)$	$O(d \log n)$	$O(m + nd)$	$O(d \log n)$

In the table, d is the number of rounds (i.e. the shortest path length from s to the reachable vertex furthest from s). The last two columns indicate the costs for when `X` is implemented as a regular sequence (with array-based costs) instead of a single-threaded sequence. The big difference is the cost of `inject`. As before the total work across all rounds is calculated by noting that every out-edge is only processed in one frontier, so $\sum_{i=0}^{d} \|F_i\| = m$.

15.4 Depth-First Search

So far we have seen that breadth-first search (BFS) is effective in solving certain problems on graphs, such as finding the shortest paths from a source. We now look at how another graph search algorithm called *depth-first search*, or *DFS* for short, is more effective for other problems such as topological sorting, and cycle detection on directed graphs.

An example application: Topological Sorting

As an example, we 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. Figure 15.1 illustrates the tasks that a climber must complete, along with the dependencies between them, as a directed graph, where vertices represent tasks and arcs represent dependencies 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.

Graphs such as these that represents tasks and dependencies between them are sometimes called ***dependency graphs***. An important property of dependency graphs is that they have no cycles. In general, we refer to a directed graph without cycles as a ***directed acyclic graph*** or a ***DAG*** for short.

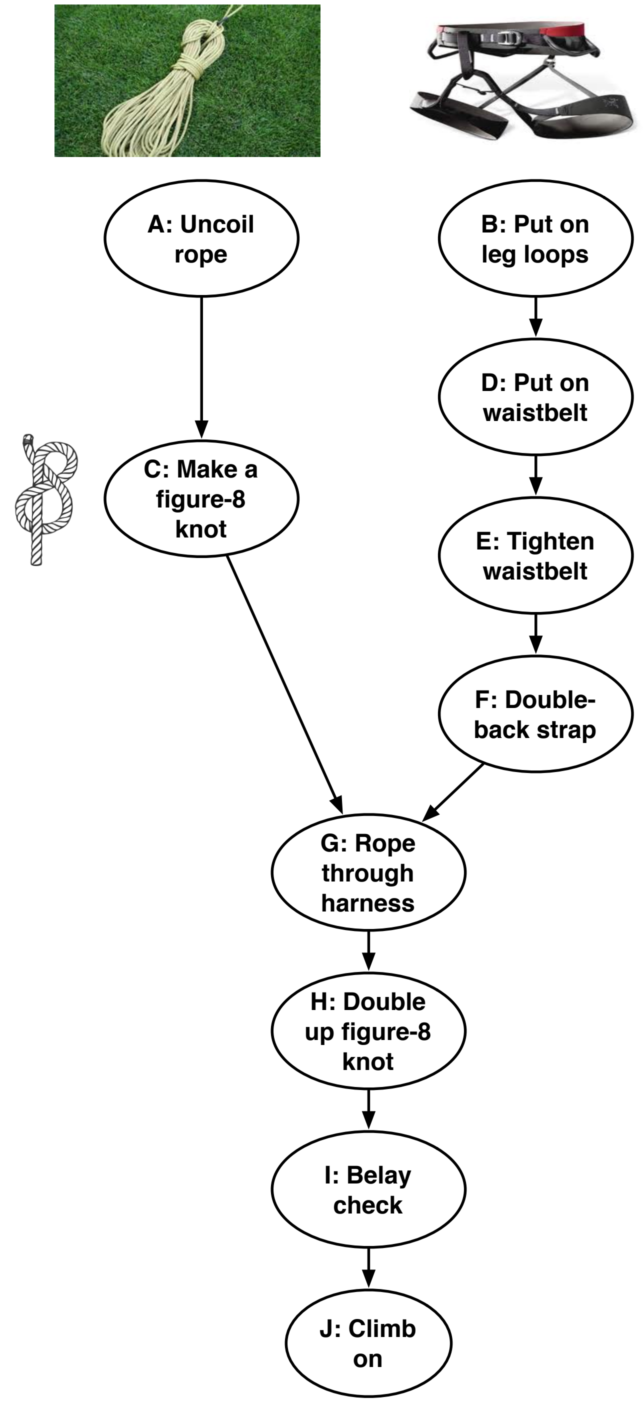


Figure 15.1: A simplified DAG for tying into a rope with a harness.

Since a climber can only perform one of these tasks at a time (at least without help), her actions are naturally ordered. We call a total ordering of the vertices of a DAG that respects all dependencies a topological sort.

Definition 15.14. [Topological Sort of a DAG] The topological sort of a DAG (V, E) is a total ordering, $v_1 < v_2 \ldots < v_n$ of the vertices in V such that for any edge $(v_i, v_j) \in E$, we have $v_i < v_j$.

There are many possible topological orderings for the DAG in Figure 15.1. 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, E, F in that order), prepare the rope (tasks A and then C), rope through, and finally complete the knot, get her gear checked, and climb on (tasks G, H, I, J, in that order).

We will soon see how to use DFS as an algorithm to solve topological sorting. BFS cannot be used to implement topological sort, because BFS visits vertices in the order of their distance from the source. This can break dependencies, because dependencies on the longer paths may be ignored. In our example, the BFS algorithm could ask the climber to rope through the harness (task G) before fully putting on the harness.

Depth-First Search (DFS)

Recall that in graph search, we can choose any (non-empty) subset of the vertices on the frontier to visit in each round. The DFS algorithm is a specialization of graph search that picks the vertex that is most recently added to the frontier. Intuitively, when a vertex is visited, we can think of "seeing" all the neighbors in some order; the DFS algorithm visits the most recently seen vertex that is in the frontier (i.e. that has not already been visited).

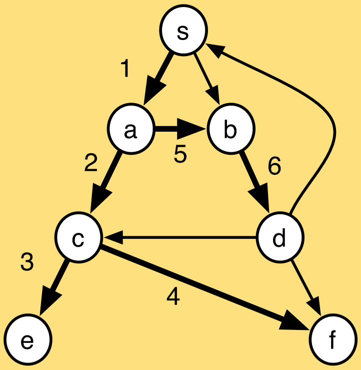
One way to find the most recently seen vertex is to time-stamp the neighbors when we visit a vertex, and then use these time stamps by always picking the most recent (largest) one. Since time stamps increase monotonically and since we always visit the largest one, we can implement this approach by using a stack, which implicitly keeps track of the ordering of the timestamps (more recently added vertices are closer to the top of the stack). We can refine this solution one-more level and implicitly represent the stack by using recursion. Algorithm 15.15 below defines such an algorithm based on this idea.

Algorithm 15.15. [DFS reachability]

```
reachability $(G, s)$ =
  let DFS $(X, v)$ =
        if $v \in X$ then $X$
        else iterate DFS $(X \cup \{v\})$ $(N_G(v))$
  in DFS $(\{\}, s)$ end
```

The algorithm maintains a set of visited vertices X. In DFS if the vertex v has not already been visited, then the algorithms marks it as visited by adding it to the set X (i.e. $X \cup \{v\}$), and then iterates over the neighbors of v, running DFS on each. The algorithm returns all visited vertices, which is the final value of X. As with BFS, this will be the vertices reachable from s (recall that all graph search techniques we consider identify the reachable vertices). Note, however, that unlike the previous algorithms we have shown for graph search, the frontier is not maintained explicitly. Instead it 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 15.16. An example of DFS on a graph where the out-edges are ordered counterclockwise, starting from the left.



v	X
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	$\{s, a, c, e, f, b, d\}$
f	$\{s, a, c, e, f, b, d\}$
s	$\{s, a, c, e, f, b, d\}$
b	$\{s, a, c, e, f, b, d\}$

Each row corresponds to one call to DFS in the order they are called, and v and X are the arguments to the call. 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.

Exercise 15.17. Convince yourself that using generic graph search where the frontier is represented as a stack visits the vertices in the same order as the recursive implementation of DFS.

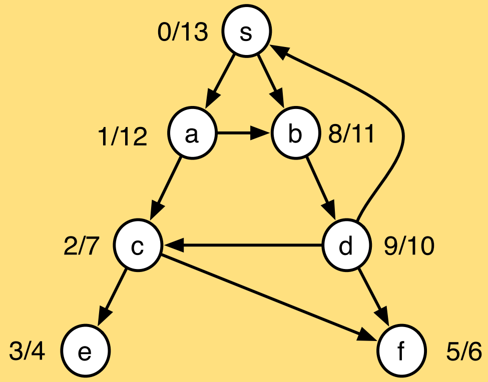
The recursive formulation of DFS has an important property—it makes it easy not just to identify when a vertex is first visited (i.e., when adding v to X), but also to identify when everything that is reachable from v has been visited, which occurs when the `iterate` completes. As we will see, many applications of DFS require us to do something at each of these points.

15.5 DFS Numbers and the DFS Tree

Applications of DFS usually require us to perform some computation at certain points during search. To develop some intuition for the structure of DFS and the important points during a DFS, think of curiously browsing photos of your friends in a photo sharing site. For the purposes of this exercise, let's suppose that you have a list of your friends in front of you and also a bag of colorful pebbles. You start by visiting the site of a friend and put a white pebble next to his name to remember that you have visited their site. You then realize that some other friends are tagged in your friend's photos, so you visit the site of one, marking again their name with a white pebble. Then in that new site, you see other friends tagged and visit the site of another and again pebble their name white. Of course, you are careful not to revisit friends that you have already visited, which are easily noticeable thanks to the pebbles. When you finally reach a site that contains no one you know that you have not already visited, you are ready to press the back button. However, before you press the back button you change the pebble for that friend from white to red to indicate that you have completely explored their site and everything reachable from them. You then press the back button, which moves you to the site you visited immediately before first visiting the current one (also the most recently visited site that still has a white pebble on it). You now check if another friend that you have not visited is tagged on that site. If there is, you visit and so on. When done visiting all neighbors you change that site from a white to a red pebble, and hit the back button again. This continues until you change the site of your original friend from a white pebble to a red pebble. This process of turning a vertex white and then later red is a conceptual way to identify two important points in the search.

To make precise this notion of turning a vertex white and red in DFS, we can assign two timestamps to each vertex. The time at which a vertex receives its white pebble is called the ***discovery time***. The time at which a vertex receives its red pebble is called ***finishing time***. We refer to the timestamps as ***DFS numbers***.

Example 15.18. A graph and its DFS numbers illustrated; t_1/t_2 denotes the timestamps showing when the vertex gets its white (discovered) and red pebble (finished) respectively.



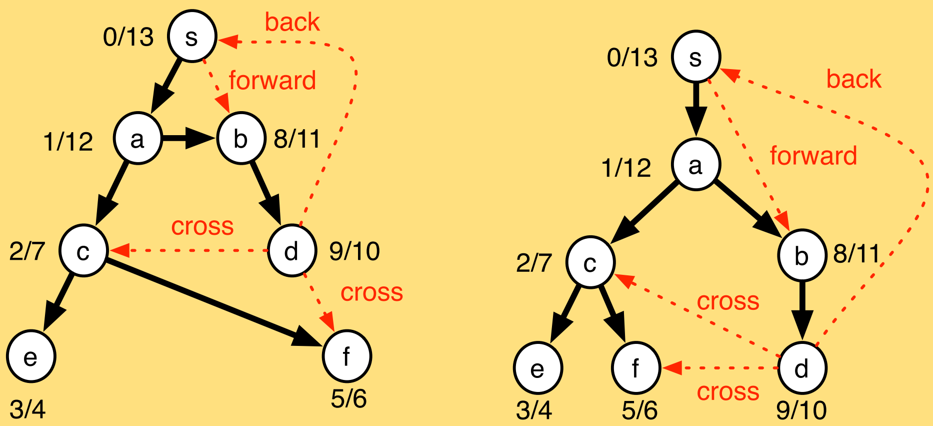
Note that vertex a gets a finished timed of 12 since it does not finish until everything reachable from its two out neighbors, c and b, have been fully explored. Vertices d, e and f have no un-visited out neighbors, and hence their finishing time is one more than their discovery time.

Given a graph and a DFS of the graph, it is can be useful to classify the edges of into categories as follows.

Definition 15.19. [Tree and Non-Tree Edges in DFS] We call an edge (u, v) a *tree edge* if v receives its white pebble when the edge (u, v) was traversed. Tree edges define the *DFS tree*, which is a special case of a graph-search tree for DFS. The rest of the edges in the graph, which are non-tree edges, can further be classified as 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 15.20. Tree edges (black), and non-tree edges (red, dashed) illustrated with the original graph and drawn as a tree.



Exercise 15.21. How can you determine by just using the DFS numbers of the endpoints of an edge whether it is a cross edge, forward edge, or backward edge?

It also useful to consider the point at which we ***revisit*** a vertex in DFS and find that it has already been visited. We can rewrite our DFS algorithm (Algorithm 15.15) so that the discover, finish and revisit times are made clear as follows.

Algorithm 15.22. [DFS with discover, finish and revisit]

```
reachability $(G, s)$ =
  let DFS $(X, v)$ =
    if $v \in X$ then $X$    % revisit $v$
    else let
      $X'$ = $X \cup \{v\}$    % discover and visit $v$
      $X''$ = iterate DFS $(X \cup \{v\})$ $(N_G(v))$
    in $X''$   % finish $v$
  in DFS $(\{\}, s)$ end
```

15.6 Applications of DFS

We now consider three applications of DFS: topological sorting (as introduced earlier), finding cycles in undirected graphs and finding cycles in directed graphs. These applications make use of the categorization of edges, the DFS numbering and the notion of discovering, finishing and revisiting.

Topological sorting

The DFS numbers have many interesting properties. One of these properties, established by the following lemma, makes it possible to use DFS for topological sorting.

Lemma 15.23. When running DFS on a DAG, if a vertex u is reachable from v then u will finish before v finishes.

Proof. This lemma might seem obvious, but we need to be a bit careful. We consider two cases.

1. u is discovered before v. In this case u must finish before v is discovered otherwise there would be a path from u to v and hence a cycle.
2. v is discovered 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.

□

Intuitively, the lemma holds because DFS fully searches any un-visited vertices that are reachable from a vertex before returning from that vertex (i.e., finishing that vertex). 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, since they must finish before v finishes. This is exactly the property we require from a topological sort.

Algorithm 15.24 gives an implementation of topological sort. Instead of generating DFS numbers, and sorting them, which is a bit clumsy, it maintains a sequence S and whenever finishing a vertex v, appends v to the front of S, i.e. $\texttt{cons}(v, S)$. This gives the same result since it orders the vertices by reverse finishing time. The main difference from the `reachability` version of `DFS` is that we thread the sequence S, as indicated by the underlines. In the code we have marked the discovery (Line 7) and finish points (Line 9). The vertex is added to the front of the list at the finish. The last line iterates over all the vertices to ensure they are all included in the final topological sort. Alternatively we could have added a "start" vertex and an edge from it to every other vertex, and then just searched from the start. The algorithm just returns the sequence S (the second value), throwing away the set of visited vertices X.

Algorithm 15.24. [Topological Sort]

```
topSort $(G = (V, E))$ =
  let DFS $((X, \underline{S}), v)$ =
      if $v \in X$ then
        $(X, \underline{S})$    % Revisit $v$
      else
        let
          $X' = X \cup \{v\}$                                     % Discover $v$
          $(X'', \underline{S'})$ = iterate DFS $(X', \underline{S})$ $(N_G^+(v))$
        in $(X'', \underline{\texttt{cons}(v, S')})$ end                    % Finish $v$
  in second (iterate DFS $(\{\}, \underline{\langle\,\rangle})$ $V$) end
```

Cycle Detection in Undirected Graphs

The second problem we consider is determining if there are cycles in an undirected graph. Given a graph $G = (V, E)$ the *cycle detection* problem is to determine if there are any cycles in the graph. The problem is different depending on whether the graph is directed or undirected. We first consider the undirected case, and then the directed case.

How would we modify the generic DFS algorithm above to solve this problem? A key observation is that in an undirected graph if DFS$'$ ever arrives at a vertex v a second time, and the second visit is coming from another vertex u (via the edge (u, v)), then there must be two paths between u and v: the path from u to v implied by the edge, and a path from v to u followed by the search between when v was first visited and u was visited. Since there are two distinct paths, there is a "cycle". Well not quite! Recall that in an undirected graph a cycle must be of length at least 3, so we need to be careful not to consider the two paths $\langle u, v \rangle$ and $\langle v, u \rangle$ implied by the fact the edge is bidirectional (i.e. a length 2 cycle). It is not hard to avoid these length two cycles by removing the parent from the list of neighbors. These observations lead to the following algorithm.

Algorithm 15.25. [Undirected cycle detection]

```
undirectedCycle $(G = (V, E))$ =
  let
    $s$ = a new vertex          % used as top level parent

    DFS $\underline{p}$ $((X, \underline{C}), v)$ =
      if $(v \in X)$ then
        $(X, \underline{true})$                  % revisit $v$
      else
        let
          $X'$ = $X \cup \{v\}$          % discover $v$
          $(X'', \underline{C'})$ = iterate (DFS $\underline{v}$) $(X', \underline{C})$ $(N_G(v)\underline{\setminus\{p\}})$
        in $(X'', \underline{C'})$ end              % finish $v$

  in second (iterate (DFS $\underline{s}$) $(\{\}, \underline{false})$ $V$) end
```

The algorithm iterates over all vertices in the final line in case the graph is not connected. It uses a "dummy" vertex s as the parent at the top level. The key differences from the generic DFS are underlined. The variable C is a Boolean variable indicating whether a cycle has been found so far. It is initially set to `false` and set to `true` if we find a vertex that has already been visited. The extra argument p to DFS$'$ is the parent in the DFS tree, i.e. the vertex from which the search came from. It is needed to make sure we do not count the length 2 cycles. In particular we remove p from the neighbors of v so the algorithm does not go directly back to p from v. The parent is passed to all children by "currying" using the partially applied (DFS$'$ v). If the code executes the `revisit` v line then it has found a path of length at least 2 from v to p and the length 1 path (edge) from p to v, and hence a cycle.

Cycle Detection in Directed Graphs

We now consider cycle detection but in the directed case. This can be an important preprocessing step for topological sort since topological sort will return garbage for a graph that has cycles. Here is the algorithm:

Algorithm 15.26. [Directed cycle detection]

```
directedCycle $(G = (V,E))$ =
  let DFS $\underline{Y}$ $((X,\underline{C}),v)$ =
      if $(v \in X)$ then
         $(X,\underline{C \lor Y[v]})$           % revisit $v$
      else let
           $X' = X \cup \{v\}$           % discover $v$
           $\underline{Y' = Y \cup \{v\}}$
           $(X'',\underline{C'})$ = iterate (DFS $Y'$) $(X',\underline{C})$ $(N_{G'}(v))$
         in $(X'',\underline{C'})$ end           % finish $v$
  in second(iterate (DFS {}) $(\{\},\underline{false})$ $V$) end
```

The differences from the generic version are once again underlined. In addition to threading a Boolean value C through the search that keeps track of whether there are any cycles, it maintains the set Y of ancestors in the DFS tree. In particular when visiting a vertex v, and before recursively visiting its neighbors, we add v to the set Y. To see how maintaining the ancestors helps recall that a *back edge* in a DFS search is an edge that goes from a vertex v to an ancestor u in the DFS tree. We can then make use of the following theorem.

Theorem 15.27. *A directed graph $G = (V, E)$ has a cycle if and only if a DFS over the vertices has a back edge.*

Exercise 15.28. Prove this theorem.

Our algorithm for directed cycles therefore simply needs to check if there are any back edges, which is only true if it revisits an ancestor, i.e., a vertex in Y.

Higher-Order DFS

As already described there is a common structure to all the applications of DFS—they all do their work either when "discovering" a vertex, when "finishing" it, or when "revisiting" it, i.e. attempting to visit when already visited. This suggests that we might be able to derive a generic version of DFS in which we only need to supply functions for these three components. This is indeed possible by having the user define a state of type α that can be threaded throughout search, and then supplying and an initial state and the following three functions. More specifically, each function takes the state, the current vertex v, and the parent vertex p in the DFS tree, and returns an updated state. The `finish` function takes both the discover and the finish state. The algorithm for generalized DFS for directed graphs can then be written as:

Algorithm 15.29. [Generalized directed DFS]

```
directedDFS (revisit,discover,finish) $(G, \Sigma_0, s)$ =
  let
    DFS  $p$ $((X, \Sigma), v)$ =
      if $(v \in X)$ then
        $(X, \underline{\texttt{revisit}}\ (\Sigma, v, p))$
      else
        let
          $\Sigma' = \underline{\texttt{discover}}\ (\Sigma, v, p)$
          $X' = X \cup \{v\}$
          $(X'', \Sigma'') =$ iterate (DFS $v$) $(X', \Sigma')$ $(N_G^+(v))$
          $\Sigma''' = \underline{\texttt{finish}}\ (\Sigma', \Sigma'', v, p)$
        in $(X'', \Sigma''')$ end
  in
    DFS $s$ $((\emptyset, \underline{\Sigma_0}), s)$
  end
```

At the end, DFS returns an ordered pair (X, Σ) : Set $\times\ \alpha$, which represents the set of vertices visited and the final state Σ. The generic search for undirected graphs is slightly different since we need to make sure we do not immediately visit the parent from the child. As we saw this causes problems in the undirected cycle detection, but it also causes problems in other algorithms. The only necessary change to the directedDFS is to replace the $(N_G^+(v))$ at the end of Line 10 with $(N_G^+(v) \setminus \{p\})$.

With this generic algorithm we can easily define our applications of DFS. For undirected cycle detection we have:

Algorithm 15.30. [Undirected Cycles with generalized undirected DFS]

```
$\Sigma_0 =$ false : bool
revisit(_) = true
discover(fl,_,_) = fl
finish(_,fl,_,_) = fl
```

For topological sort we have.

Algorithm 15.31. [Topological sort with generalized directed DFS]

```
$\Sigma_0 = [\ ]$ : vertex list
revisit($L, \_, \_$) $=$ $L$
discover($L, \_, \_$) $=$ $L$
finish($\_, L, v, \_$) $=$ $v :: L$
```

For directed cycle detection we have.

Algorithm 15.32. [Directed cycles with generalized directed DFS]

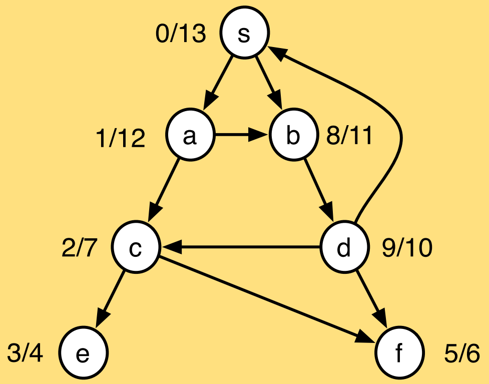
```
$\Sigma_0$ $=$ ($\{\}$, false) : Set $\times$ bool
revisit (($S$, fl), $v$, $\_$) $=$ ($S$,  fl $\vee$ ($S[v]$))
discover (($S$, fl), $v$, $\_$) $=$ ($S \cup \{v\}$ , fl)
finish (($S$, $\_$), ($\_$, fl), $v$, $\_$) $=$ ($S$, fl)
```

For these last two cases we need to also augment the graph with the vertex s and add the edges to each vertex $v \in V$. Note that none of the examples actually use the last argument, which is the parent. There are other examples that do.

15.7 Cost of DFS

At first sight, we might think that DFS can be parallelized by searching the out edges in parallel. This might work if the searches on each out edge never "meet up" as would be the case for a tree. However, 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 (discovered) once, and in DFS it is also important that the earlier out-edge discovers any shared vertices, not the later one.

Example 15.33. Consider the example graph drawn below.



If we search the out-arcs 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 15.34. 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.

We therefore assume there is no parallelism in DFS and focus on the work. The work required by DFS will depend on the data structures used to implement the set, but generally we can bound work by counting the number of operation and multiplying this count by the cost of each operation. In particular we have the following lemma.

Lemma 15.35. For a graph $G = (V, E)$ with m edges, and n vertices, `DFS` function in Algorithms 15.15 and 15.24 will be called at most $n + m$ times and a vertex will be discovered (and finished) at most n times.

Proof. Since each vertex is added to X when it is first discovered, every vertex can only be discovered once. It follows that every out edge will only be traversed once, invoking a call to `DFS`. Therefore `DFS` is called at most m times through an edge. In topological sort, an additional n initial calls are made starting at each vertex. □

Each call to `DFS` performs one find for $X[v]$. Every time the algorithm discovers a vertex, it performs one insertion of v into X. In total, the algorithm therefore performs at most n

insertions and $m + n$ finds. This results in the following cost specification.

Cost Specification 15.36. [DFS] The DFS algorithm on a graph with m out edges, and n vertices, and using the tree-based cost specification for sets runs in $O((m + n) \log n)$ work and span. Later we will consider a version based on single threaded sequences that reduces the work and span to $O(n + m)$.

DFS with Single-Threaded Arrays

Here is a version of DFS using adjacency sequences for representing the graph and ST sequences for keeping track of the visited vertices.

Algorithm 15.37. [DFS with single threaded arrays]

```
directedDFS ($G$:(int seq) seq,$\Sigma_0 : \alpha$, $s$:int) =
  let
    DFS $p$ (($X$:bool stseq, $\Sigma : \alpha$, $v$:int) =
      if ($X[v]$) then
        ($X$, revisit ($\Sigma, v, p$))
      else
        let
          $X'$ = STSeq.update $X$ ($v$,true)
          $\Sigma' =$ discover ($\Sigma, v, p$)
          ($X'', \Sigma''$) = iterate (DFS $v$) ($X', \Sigma'$) ($G[v]$)
          $\Sigma''' =$ finish ($\Sigma', \Sigma'', v, p$)
        in ($X'', \Sigma'''$) end

    $X_{init}$ = STSeq.fromSeq ⟨ false : $v \in \langle 0, \ldots, |G| - 1 \rangle$ ⟩
  in
    DFS $s$ (($X_{init}, \Sigma_0$), $s$)
  end
```

If we use an stseq for X (as indicated in the code) then this algorithm uses $O(m)$ work and span. However if we use a regular sequence, it requires $O(n^2)$ work and $O(m)$ span.

Exercise 15.38. Convince yourself that the $O(m)$ bound above is correct, e.g., not $O(n + m)$.

15.8 Priority-First Search

The graph search algorithm that we described does not specify the vertices to visit next (the set U). This is intentional, because graph search algorithms such as breadth-first search and depth-first search differ exactly on which vertices they visit next. Many graph-search algorithms can be viewed as visiting vertices in some priority order. To see this, suppose that a graph-search algorithm assigns a priority to every vertex in the frontier. We can imagine the algorithm assigning a priority to a vertex v when it inserts v into the frontier. Now instead of choosing some unspecified subset of the frontier to visit next, the algorithm picks the highest (or the lowest) priority vertices. Effectively we change Line 6 in the `graphSearch` algorithm to:

```
Choose $U$ as the highest priority vertices in $F$
```

We refer to such an algorithm as a *priority-first search (PFS)* or a *best-first search*. The priority order can either be determined statically (a priori) or it can be generated on the fly by the algorithm.

Priority-first search is a greedy technique since 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. As you will see soon, several famous graph algorithms are instances of priority-first search, e.g., Dijkstra's algorithm for finding single-source shortest paths and Prim's algorithm for finding Minimum Spanning Trees.

.

Chapter 16

Shortest Paths

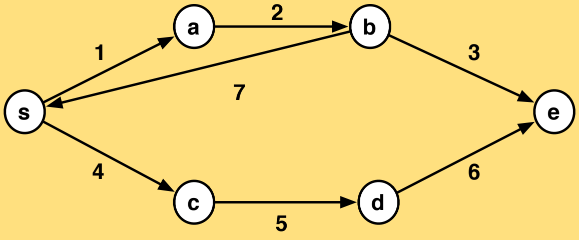
Given a graph where edges are labeled with weights (or distances) and a source vertex, what is the shortest path between the source and some other vertex? Problems requiring us to answer such queries are broadly known as ***shortest-paths problems***. Shortest-paths problem come in several flavors. The ***single-source shortest path*** problem requires finding the shortest paths between a given source and all other vertices; the ***single-pair*** shortest path problem requires finding the shortest path between given a source and a given destination vertex; the ***all-pairs shortest path problem*** requires finding the shortest paths between all pairs of vertices.

In this chapter, we consider the single-source shortest-paths problem and discuss two algorithms for this problem: Dijkstra's and Bellman-Ford's algorithms. Dijkstra's algorithm is more efficient but it is mostly sequential and it works only for graphs where edge weights are nonnegative. Bellman-Ford's algorithm is a good parallel algorithm and works for all graphs but performs significantly more work.

16.1 Shortest Weighted Paths

Consider a weighted graph $G = (V, E, w)$, where V is the set of vertices, E is the set of edges, and $w\colon E \to \mathbb{R}$ 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 $(u, v) \notin E$. We define the ***weight of a path*** as the sum of the weights of the edges along that path.

Example 16.1. 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.



For a weighted graph $G(V, E, w)$ a shortest weighted path from vertex u to vertex v is a path from u to v with minimal weight. There might be multiple paths with equal weight; if so they are all shortest weighted paths from u to v. We use the term ***distance*** from u to v, written $\delta_G(u, v)$, to refer to the weight of a shortest path from u to v. If there is no path from u to v, then we define the distance to be infinity, i.e., $\delta_G(u, v) = \infty$.

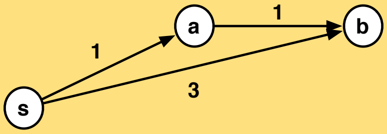
Recall that a path allows for repeated vertices—a simple path does not. If we allow for negative weight edges then it is possible to create shortest paths of infinite length (in edges) and whose weight is $-\infty$. In Example 16.1 if we change the weight of the edge (b, s) to -7 the shortest path between s and e has weight $-\infty$ since a path can keep going around the cycle $\langle\, s, a, b, s \,\rangle$ reducing its weight by 4 each time around. For this reason, when computing shortest paths we will need to be careful about cycles of negative weight. As we will discover, even if there are no negative weight cycles, negative edge weights make finding shortest paths more difficult. For this reason, we will first consider the problem of finding shortest paths when there are no negative edge weights.

In this chapter, we are interested in finding single-source shortest paths as defined in Problem 16.2 below. Note that the problem requires finding only one of the possibly many shortest paths between two vertices. In some cases, we only care about the distance $\delta(u, v)$ but not the path itself.

Problem 16.2. [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.

In Section 15.1 we saw how Breadth-First Search (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. Ignoring weight and using BFS, unfortunately, does not work on weighted graphs.

Example 16.3. To see why BFS does not work, consider the following directed graph with 3 vertices:



BFS first visits b and then a. When it visits b, it assigns b an incorrect weight of 3. Since BFS never visit b again, it will not find the shortest path going trough a, which happens to be shorter.

The reason why BFS works on unweighted graphs is quite interesting and helpful for understanding other shortest path algorithms. The key idea behind BFS is to visit vertices in order of their distance from the source, visiting closest vertices first, and the next closest, and so on. More specifically, for each frontier F_i (at round i), BFS has the correct distance from the source to each vertex in the frontier. It then can determine the correct distance for unencountered neighbors that are distance one further away (on the next frontier). We will use a similar idea for weighted paths.

16.2 Dijkstra's Algorithm

Consider a variant of the SSSP problem, where all the weights on the edges are non-negative (i.e. $w : E \to \mathbb{R}^+$). We refer to this as the **SSSP$^+$** ***problem***. Dijkstra's algorithm solves the SSSP$^+$ problem. It is an important algorithm, because it is efficient, and because it is an elegant example of a greedy algorithm that can find optimal results. In this section, we are going to (re-)discover this algorithm by taking advantage of properties of graphs and shortest paths.

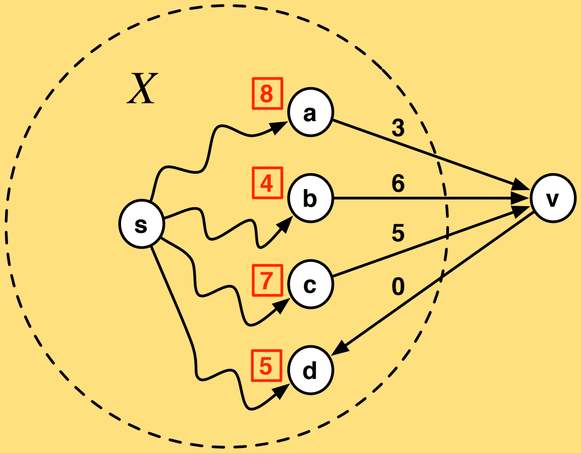
Let's start by noting that since no edges have negative weights, there cannot be a negative-weight cycle. One can therefore never make a path shorter by visiting a vertex twice—i.e., a path that cycles back to a vertex cannot have less weight than the path that ends at the first visit to the vertex. When searching for a shortest path, we thus have to consider only the simple paths.

Let us start with a brute-force algorithm for the SSSP$^+$ problem, that, for each vertex, considers all simple paths between the source and the vertex and selects the shortest such path. Unfortunately there can be a large number (exponential in the number of vertices) of paths between any pair of vertices, so any algorithm that tries to look at all paths is not likely to scale beyond very small instances. We therefore have to try to reduce the work. Toward this end let us observe that the brute-force algorithm does redundant work, because it does not take advantage of a crucial property of shortest paths: any sub-path of a shortest path is a shortest path (between its end vertices). We refer to this property as the ***sub-paths property*** of shortest paths. This property means that we can build shortest paths from smaller shortest paths. We are going to use this property to derive both Dijkstra's algorithm, and also Bellman-Ford's algorithm, which solves the SSSP problem on general graphs, allowing for negative edge weights.

Example 16.4. [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.

To see how sub-paths property can be helpful, consider the graph in Example 16.5 and suppose that an oracle has told us the shortest paths to all vertices except for the vertex v. 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 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. Recall that we have decided to find only the shortest paths that are simple, which cannot go through v itself.

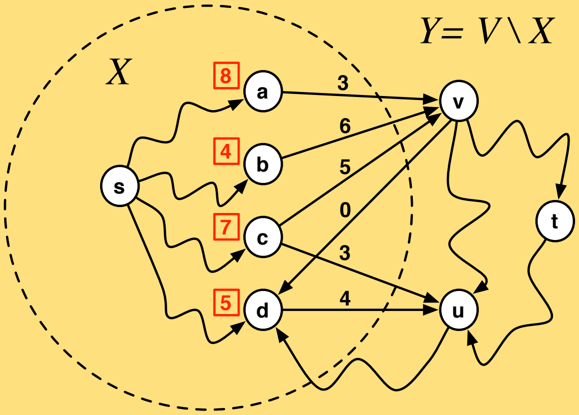
Example 16.5. In the graph G shown below, suppose that we have found the shortest paths from the source s to all the other vertices except for v; each vertex is labeled with its distance to s. The weight of the shortest path to v
is $\min\left(\delta_G(s,a)+3, \delta_G(s,b)+6, \delta_G(s,c)+5\right)$. The shortest path goes through the vertex $(a, b,$ or $c)$ that minimizes the weight, which in this case is vertex b.



Let's try to generalize the argument and consider the case where the oracle tells us the shortest paths from s to 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. $V \setminus X$. Consider 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. As in graph search, we call the set of vertices that are neighbors of X but not in X, i.e. $N^+(X) \setminus X$, the frontier.

It should be clear that any path that leaves X has to 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. Consider the vertex $v \in Y$ that has an edge to some already visited vertex $x \in X$ and that minimizes the sum $\delta_G(s, x) + w(x, v)$. Since all paths to Y must go through the frontier when exiting X, and since edges are non-negative, a sub-path cannot be longer than the full path. Thus, no other vertex in Y can be closer to the source than x. See Example 16.6. Furthermore, since all other vertices in Y are farther than v and since all edges are non-negative, the shortest path for v is $\delta_G(s, x) + w(x, v)$.

Example 16.6. 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 shortest path from the source s to a vertex u in Y is the path to vertex d with weight 5 followed by the edge (d, u) with weight 4 and hence total weight 9. If edge weights are non-negative there cannot be any shorter way to get to u, whatever $w(v, u)$ is, therefore we know that $\delta(s, u) = 9$.



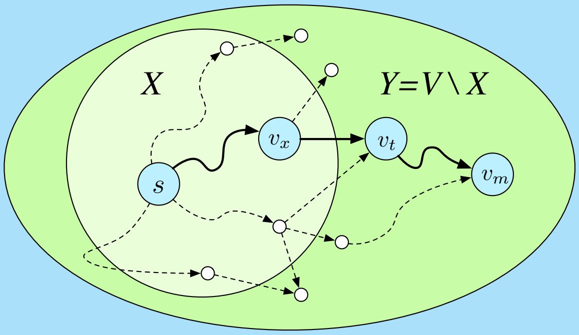
The intuition that we have developed thus far is stated more precisely and proved in Lemma 16.7. The Lemma tells us that once we know the shortest paths to a set X we can add more vertices to X with their shortest paths. This gives us a crank that can churn out at least one new vertex on each round. Dijkstra's algorithm simply does exactly this: it turns the crank until all vertices are visited.

You 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 search is a graph search, where each round visits the frontier vertices 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 weight consisting of a path to $x \in X$ and an additional edge from x

Lemma 16.7. [Dijkstra's Property] Consider a (directed) weighted graph $G = (V, E, w)$, $w\colon E \to \mathbb{R}^*$, 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

$$p(v) \equiv \min_{x \in X}(\delta_G(s, x) + w(x, v))$$

then $\min\limits_{y \in Y} p(y) = \min\limits_{y \in Y} \delta_G(s, y)$.



In 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 (see the figure). 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. □

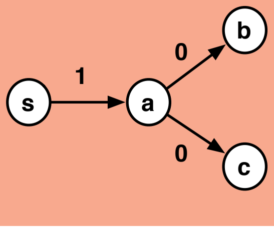
Implication: *This gives us a way to easily find $\delta_G(s, v)$ for at least one vertex $v \in Y$. In particular for all $v \in Y$ where $p(v) = \min_{y \in Y} p(y)$, it must be the case that $p(v) = \delta_G(s, v)$ since there cannot be a shorter path. Also we need only consider the frontier vertices in calculating $p(v)$, since for others in Y, $p(y)$ is infinite.*

to v. In other words, this algorithm is actually an instance of priority-first search. We are now ready to define precisely Dijkstra's algorithm.

Algorithm 16.8. [Dijkstra's Algorithm] For a weighted graph $G = (V, E, w)$ and a source s, Dijkstra's algorithm is priority search on G starting at s with $d(s) = 0$, using priority $p(v) = \min\limits_{x \in X}(d(x) + w(x, v))$ (to be minimized), and setting $d(v) = p(v)$ when v is visited.

Note that Dijkstra's algorithm will visit vertices in non-decreasing shortest-path weight since on each round it visits unvisited vertices that have the minimum shortest-path weight from s.

Remark 16.9. It may be tempting to think that Dijkstra's algorithm visits vertices strictly in increasing order of shortest-path weight from the source, visiting vertices with equal shortest-path weight on the same round. This is not true. To see this consider the example below and convince yourself that it does not contradict our reasoning.



Lemma 16.10. 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 16.7, $p(v) = \delta_G(s, v)$, 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. □

16.2.1 Implementation and the Cost of Dijkstra's Algorithm

We can implement Dijkstra's algorithm efficiently using a priority queue to maintain $p(v)$ as shown in Algorithm 16.11. The algorithm uses a priority queue that supports `deleteMin` and `insert` operations. Note that this algorithm only adds one vertex at a time even if there are multiple vertices with equal distance.

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 (Line 8 in the code) and, if it hasn't already been visited, visits it by adding $(x \mapsto d)$ to the table of visited vertices (Line 15), and then adds all its neighbors v to Q along with the priority $d(x) + w(x, v)$ (i.e. the distance to v through x) (Lines 16 and 17). 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, but what is important is that the minimum distance will always be pulled out first. Line 11 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.

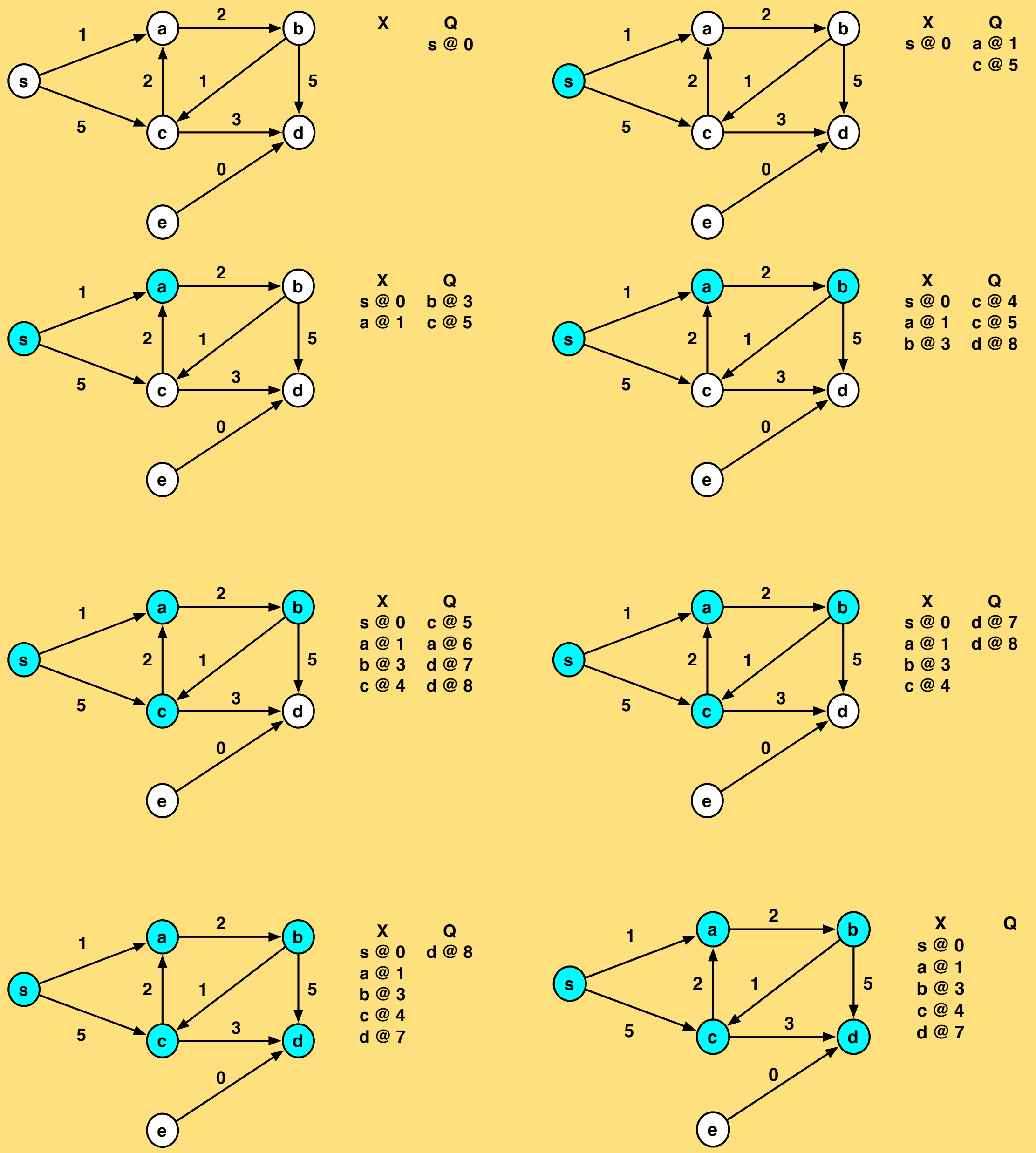
Algorithm 16.11. [Dijkstra's Algorithm using Priority Queues]

```
dijkstraPQ $(G, s)$ =
let
  % requires:
      $\forall (x \mapsto d) \in X, d = \delta_G(s, x)$
      $\{(d, y) \in Q \mid y \in V \setminus X\} = \{(d + w(x, y), y) : \; (x \mapsto d) \in X,$
                                                  $y \in N_G^+(x) \setminus X\}$
  % returns: $\{x \mapsto \delta_G(s, x) : x \in R_G(s)\}$
  function dijkstra $(X, Q)$ =
    case $\boxed{\texttt{PQ.deleteMin}}(Q)$ of
      (None, _) => $X$
     | (Some $(d, v), Q'$) =>
        if $\boxed{(v, \_) \in X}$ then
          dijkstra $(X, Q')$
        else
          let
            $X' = \boxed{X \cup \{(v, d)\}}$
            relax $(Q, (u, w)) = \boxed{\texttt{PQ.insert}}(d + w, u)\; Q$
            $Q'' = \boxed{\texttt{iterate}}$ relax $Q' \; \boxed{N_G^+(v)}$
          in dijkstra $(X', Q'')$ end
in
  dijkstra $(\{\}, \boxed{\texttt{PQ.insert}}(0, s)\; \{\})$
end
```

There are a couple other variants on Dijkstra's algorithm using priority queues. Firstly we could check inside the `relax` function whether u is already in X and if so not insert it into the priority queue. This does not affect the asymptotic work bounds but probably would give some improvement in practice. Another variant is to decrease 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.

To analyze the work and span of priority-queue based implementation of Dijkstra's algorithm shown in Algorithm 16.11, let's first consider the priority queue ADT's that we use. For the priority queue, we assume `PQ.insert` and `PQ.deleteMin` have $O(\log n)$ work and span. Since Dijkstra's algorithm makes no updates to the graph, we can represent the input graph simply either by using a table or a sequence, mapping vertices to their out-neighbors along with the weight of the corresponding edge. As we shall see, it suffices to use the tree based costs for tables. To represent the mapping of visited vertices to their distances, we can use a table, an array sequence, or a single-threaded array sequences.

Example 16.12. 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.



Operation	Line	# of calls	PQ	Tree Table	Array	ST Array
`deleteMin`	Line 8	$O(m)$	$O(\log m)$	-	-	-
`insert`	Line 16	$O(m)$	$O(\log m)$	-	-	-
`find`	Line 11	$O(m)$	-	$O(\log n)$	$O(1)$	$O(1)$
`insert`	Line 15	$O(n)$	-	$O(\log n)$	$O(n)$	$O(1)$
$N_G^+(v)$	Line 17	$O(n)$	-	$O(\log n)$	$O(1)$	-
`iterate`	Line 17	$O(m)$	-	$O(1)$	$O(1)$	-

Figure 16.1: The costs for the important steps in the algorithm `dijkstraPQ`.

To analyze the work, we calculate the work for each different kind of operation and sum them up to find the total work. Figure 16.1 summarizes the costs of the operations, along with the number of calls made to each operation.

The algorithm includes a box around each operation on the graph G, the set of visited vertices X, or the priority queue `PQ`. The `PQ.insert` in Line 20 is called only once, so we can safely ignore it. Of the remaining operations, The `iterate` and $N_G^+(v)$ on Line 17 are on the graph, Lines 11 and 15 are on the table of visited vertices X, and Lines 8 and 16 are on the priority queue Q.

We can calculate the total number of calls to each operation by noting that the body of the let starting on Line 14 is only run once for each vertex. Thus, Line 15 and $N_G^+(v)$ on Line 17 are only called $O(n)$ times. All other operations are performed once for each edge. The total work for Dijkstra's algorithm using a tree table is therefore $O(m \log m + m \log n + m + n \log n)$. Since $m \leq n^2$, the total work is $O(m \log n)$.

Since the algorithm is sequential, the span is the same as the work.

Based on the table one should note that when using either tree tables or single threaded sequences, the cost is no more than the cost of the priority queue operations. Therefore there is no asymptotic advantage of using one over the other; there might, however, be differences in constant factors. One should also note that using regular purely functional arrays is not a good idea, because the cost is then dominated by the insertions and the algorithm runs in $\Theta(n^2)$ work.

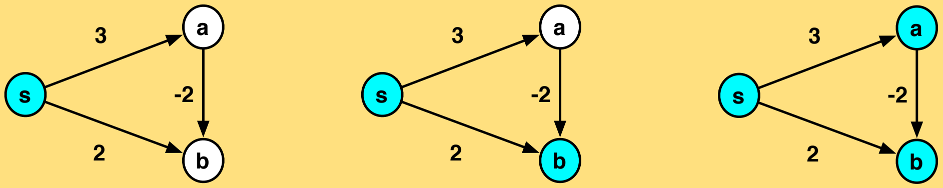
16.3 The Bellman Ford Algorithm

We now turn to solving the single source shortest path problem in the general case where we allow negative weights in the graph. One might ask how negative weights make sense. When considering distances on a map for example, negative weights do not make sense (at least without time travel), but many other problems reduce to shortest paths. In such reductions negative weights do show up.

Before proceeding we note that if there is a negative weight cycle (the sum of weights on the cycle is negative) reachable from the source, then there cannot be a finite-distance solution to the single-source shortest path problem, as discussed earlier. In such a case, we would like the algorithm to indicate that such a cycle exists and terminate.

Recall that in our development of Dijkstra's algorithm we assumed non-negative edge weights. This both allowed us to only consider simple paths (with no cycles) but more importantly played a critical role in arguing the correctness of Dijkstra's property. More specifically, Dijkstra's algorithm is based on the assumption that the shortest path to the vertex v in the frontier that is closest to the set of visited vertices, whose distances have been determined, can be determined by considering just the incoming edges of v. With negative edge weights, this is not true anymore, because there can be a shorter path that ventures out of the frontier and then comes back to v.

Example 16.13. 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 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).

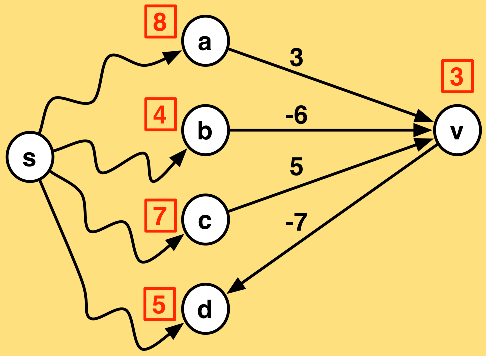
A property we can still take advantage of, however, is that the sub-paths of a shortest paths themselves are shortest. Dijkstra's algorithm exploits this property by building longer paths from shorter ones, i.e., by building shortest paths in non-decreasing order. With negative edge weights this does not work anymore, because paths can get shorter as we add edges.

But there is another way to use the same property: building paths that contain more and more edges. To see how, suppose that you have found the shortest paths from source to all vertices with k or fewer edges. We can compute the shortest paths with $k+1$ or fewer edges by extending all paths by one edge if this is beneficial. To find the shortest paths with at most $k+1$ edges, all we need to do is consider each vertex v and all its incoming edges and pick the shortest path to that vertex that arrives at a neighbor u using k or fewer edges and then takes the edge (u, v). To make this more precise, define ***k-distance***, written $\delta_G^k(s,t)$, as distance from s to t considering all paths with at most k edges, i.e., the shortest weighted path from s to t using at most k edges.

Example 16.14. 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 $\delta^k_G(s,u)$. The weight of the shortest path to v using $k+1$ or fewer edges is

$$\min\left(\delta^k_G(s,v), \min \delta^k_G(s,a)+3, \delta^k_G(s,b)-6, \delta^k_G(s,c)+5\right.$$

The shortest path with at most $k+1$ edges has weight -2 and goes through vertex b.



Based on this idea, we can construct paths with greater number of edges. We start by determining $\delta^0_G(s,v)$ for all $v \in V$. 5 Since no vertex other than the source is reachable with a path of length 0, $\delta^0_G(s,v) = \infty$ for all vertices other than the source. Then we determine $\delta^1_G(s,v)$ for all $v \in V$, and iteratively, $\delta^{k+1}_G(s,v)$ based on all $\delta^k_G(s,v)$ for all $k > 1$. To calculate the updated distances with at most $k+1$ edges the algorithm, we can use the shortest path with k edges to each of its in-neighbors and then adds in the weight of the one additional edge. More precisely, for each vertex v,

$$\delta^{k+1}(v) = \min(\delta^k(v), \min_{x \in N^-(v)} (\delta^k(x) + w(x,v))).$$

Recall that $N^-(v)$ indicates the in-neighbors of vertex v.

Since the new distance for a vertex is calculated in terms of the distances from most recent iteration, if the distances for all vertices remain unchanged, then they will continue remaining unchanged after one more iteration. It is therefore unnecessary to continue iterating after the distances converge—we can stop when distances converge. Convergence, however is not guaranteed. For example, if we have a negative cycle reachable from the source then, some distances continue to decrease at each iteration. We can detect negative cycles by noticing that the distances do not converge even after $|V|$ iterations. This is because, a simple (acyclic) path in a graph can include at most $|V|$ edges and, in the absence of negative-weight cycles, there always exist an acyclic shortest path.

Algorithm 16.15 defines the Bellman Ford algorithm based on these ideas. The algorithm runs until convergence or until $|V|$ iterations have been performed. If $|V$ iterations have been per-

Algorithm 16.15. [Bellman Ford]

```
BellmanFord $(G = (V, E), s)$ =
let
    % requires: $\forall v \in V, D_v = \delta^k_G(s, v)$
    BF $(D, k)$ =
    let
        $D' = \{v \mapsto \min(D[v], \min_{u \in N^-_G(v)}(D[u] + w(u, v))) : v \in V\}$
    in
      if $(k = |V|)$ then None
      else if (all$\{D[v] = D'[v] : v \in V\}$) then Some $D$
      else $BF(D',\ k + 1)$
    end

    $D = \{v \mapsto\ \infty : v \in V \setminus \{s\}\} \cup \{s \mapsto 0\}$

in $BF(D, 0)$ end
```

formed and the distances did not converge, then in Line 8, the algorithm returns `None`. This indicates that the algorithm has detected a negative weight cycle. An illustration of the algorithm over several steps is shown in Example 16.16.

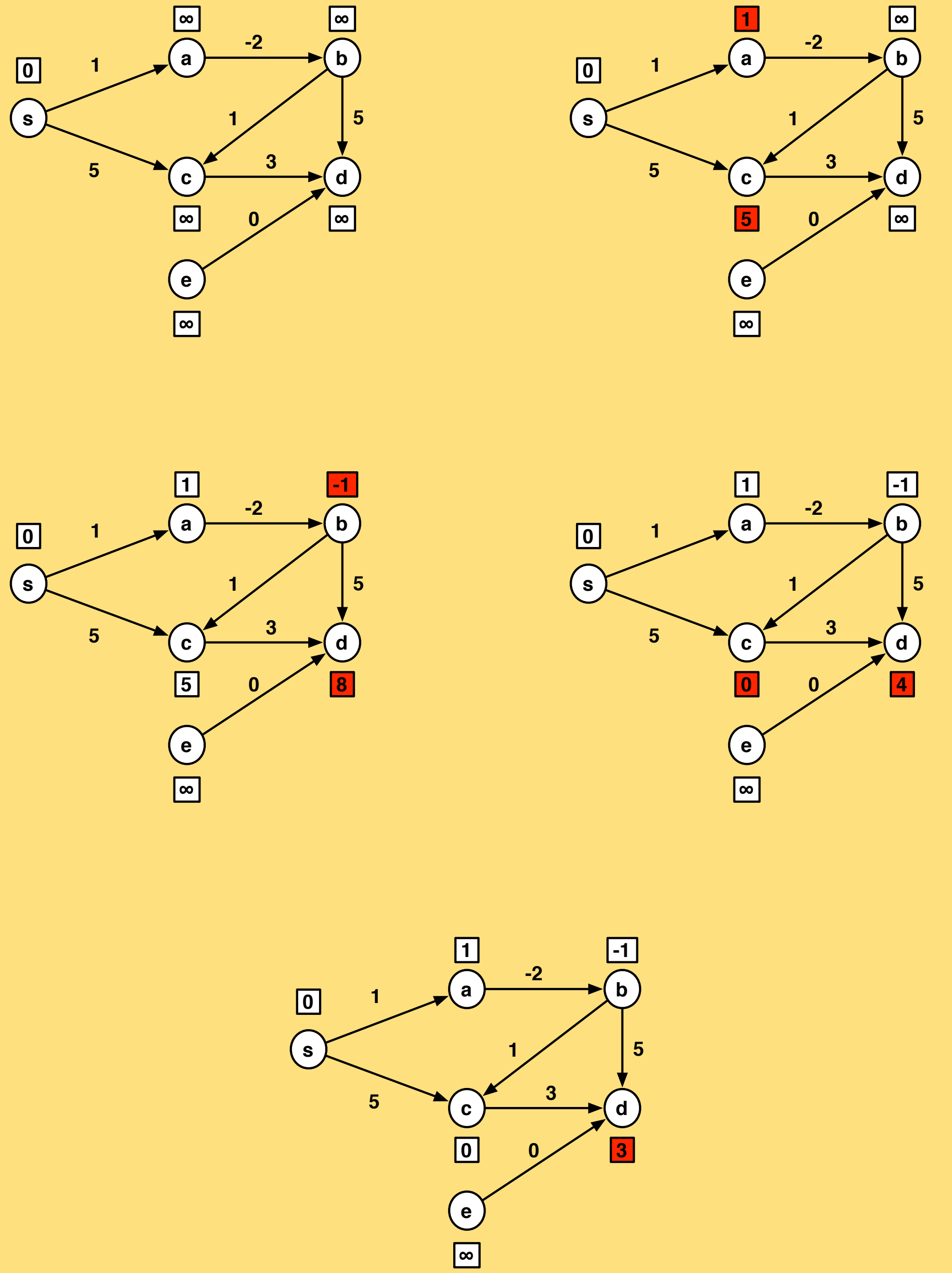
Theorem 16.17. *Given a directed weighted graph $G = (V, E, w)$, $w : E \to R$, and a source s, the* `BellmanFord` *algorithm returns either $\delta_G(s, v)$ for all vertices reachable from s, or indicates that there is a negative weight-cycle in G that is reachable from s.*

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. □

16.3.1 Cost of Bellman-Ford

For the cost analysis cost of Bellman-Ford, we consider two different representations for graphs, one using tables and the other using sequences. For a table-based representation of the graph, we use a table mapping each vertex to a table of neighbors along with their real-valued weights. We represent the distances D as a table mapping vertices to their distances.

Example 16.16. Several steps of the Bellman Ford algorithm. The numbers with squares indicate the current distances and highlight those that has changed on each step.



Let's consider the cost of one call to BF, not including the recursive calls. The only nontrivial computations are on Lines 6 and 9. Line 6 consists of a tabulate over the vertices. As the cost specification for tables indicate, to calculate the work for a tabulate, we take the sum of the work for each vertex, and for the span we take the maximum of the spans, and add $O(\log n)$. Now consider what the algorithm does for each vertex. First, it has to find the neighbors in the graph (using a `find G v`). This requires $O(\log |V|)$ work and span. Then the algorithm performs a map over the in-neighbors. Each instance of this map requires finding in the distance table to obtain $D[u]$, finding in the weight table, the weight, and an addition operation. The find operations take $O(\log |V|)$ work and span. Finally there is a reduce for finding the shortest path through an in-neighbor. The reduce takes $O(1 + |N_G(v)|)$ work and $O(\log |N_G(v)|)$ span. For each vertex the value calculated by mapping over the neighbors is compared against the current distance $D[v]$ and the minimum is taken. This requires $O(\lg |V|)$ work and span. Using $n = |V|$ and $m = |E|$, we can write the work as follows

$$\begin{aligned}
W_{BF}(n,m) &= O\left(\sum_{v \in V}\left(\log n + |N_G^-(v)| + \sum_{u \in N_G^-(v)} (1 + \log n)\right)\right) \\
&= O\left((n+m)\log n\right).
\end{aligned}$$

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 as follows

$$\begin{aligned}
S_{BF}(n,m) &= O\left(\max_{v \in V}\left(\log n + \log |N_G^-(v)| + \max_{u \in N_G^-(v)} (1 + \log n)\right)\right) \\
&= O(\log n).
\end{aligned}$$

The work and span of Line 9 is simpler to analyze since it only involves a tabulate and a reduction: it requires $O(n \log n)$ work and $O(\log n)$ span.

Since the number of calls to BF is bounded by n, as discussed earlier. Since the calls to `BF` are performed sequentially, we can multiply the work and span for each call by the number of calls to compute the total work and span, which, assuming $m \geq n$, are

$$\begin{aligned}
W_{BF}(n,m) &= O(nm \log n) \\
S_{BF}(n,m) &= O(n \log n).
\end{aligned}$$

Let's now analyze the cost with a sequence representation of graphs. If we assume that the graphs is unemerable, then the vertices are identified by the integers $\{0, 1, \ldots, |V| - 1\}$ and we can use sequences to represent the graph. Instead of using a `find` for a table, which requires $O(\log n)$ work, we can use `nth` (indexing) requiring only $O(1)$ work. This improvement in costs can be applied for looking up in the graph to find the neighbors of a vertex, and looking up in the distance table to find the current distance. By using the improved costs we get:

$$\begin{aligned}
W_{BF}(n,m) &= O\left(\sum_{v \in V}\left(1 + |N_G^-(v)| + \sum_{u \in N_G^-(v)} 1\right)\right) \\
&= O(n+m) \\
S_{BF}(n,m) &= O\left(\max_{v \in V}\left(1 + \log|N_G^-(v)| + \max_{u \in N_G^-(v)} 1\right)\right) \\
&= O(\log n)
\end{aligned}$$

and hence the overall complexity for `BellmanFord` with array sequences is and assuming $m \geq n$,

$$\begin{aligned}
W(n,m) &= O(nm) \\
S(n,m) &= O(n \log n)
\end{aligned}$$

By using array sequences we have reduced the work by a $O(\log n)$ factor.

16.4 Problems

16-1 Sub-paths
Prove that the sub-paths property holds for any graph, also in the presence of negative weights.

16-2 Distances and Paths
We mentioned that we are sometimes only interested in the weight of a shortest path, rather than the path itself. For a weighted graph $G = (V, E, w)$, assume you are given the distances $\delta_G(s,v)$ for all vertices $v \in V$. For a particular vertex u, describe how you could determine the vertices P in a shortest path from s to u in $O(p)$ work, where $p = \sum_{v \in P} d^-(v)$.

16-3 Dijkstra and BFS
Consider a graph G where all edges have weight 1. The following is true: When run on G, Dijkstra's can algorithm visits the vertices in the same order as BFS does. Explain your answer.

16-4 Parallel Dijkstra
As we have seen, there is not much parallelism in Dijkstra's algorithm. One way to increase parallelism in Dijkstra is to remove all vertices that have the same priority together and visit them all at once. Carefully describe an algorithm that can perform such visits in parallel and the priority-queue ADT and the data structure needed by the algorithm. What is the work and span of your algorithm?

16-5 Dijkstra with Negative Edge Weights
Given a graph G with possibly negative edge weights, one idea would be to find the smallest negative edge weight k and create a new graph G' by adding to each edge weight the value $|k|$ to ensure that all edge weights are non-negative. We can now run Dijkstra's algorithm on G' from any given source s to find the shortest paths to all the other vertices from s. Since by adding the same constant value $|k|$ to each edge, we have not changed the relative comparison between edges. We might thus conclude that the shortest paths in G' and G are the same. Is this correct? Prove or disprove by creating an example. Is there a relationship between shortest paths in G and G'?

.

Chapter 17

Graph Contraction and Connectivity

In earlier chapters, we have mostly covered techniques for solving problems on graphs that were developed in the context of sequential algorithms. Some of them are easy to parallelize while others are 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 limited parallelism in Dijkstra's algorithm, but there was plenty of parallelism in the Bellman-Ford algorithm. In this chapter we cover a technique called "graph contraction" that was specifically designed to be used in parallel algorithms and that allows obtaining poly-logarithmic span for certain graph problems.

Let's start with a definition of 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 $B \in P$; each susbet $B \in P$ is called a part or block.

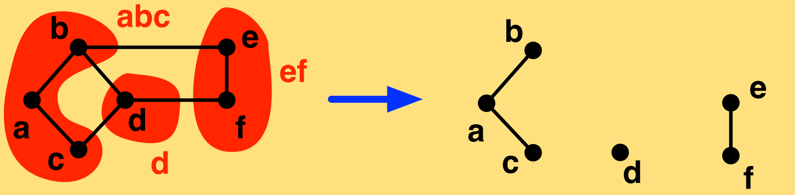
Definition 17.1. Given a graph G, a ***graph partition*** of G is a collection of graphs $H_0 = (V_0, E_0), \ldots, H_{k-1} = (V_{k_1}, E_{k-1})$ such that $\{V_0, \ldots, V_{k-1}\}$ is a set partition of V and $H_0, \ldots, H_{k-1}$ are vertex-induced subgraphs of G with respect to $V_0, \ldots, V_{k-1}$. Each subgraph H_i is called a part of G.

In other words, a graph partition is a collection of graphs induced by vertex-induced graphs of a set partition of its vertices.

In a graph partition, we can distinguish between 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 $v_2 \in V_i$, and by construction $\{v_1, v_2\} \in E_i$. We call an edge $\{v_1, v_2\}$ a ***cut edge***, if $v_1 \in V_i$ and $v_2 \in V_j$ and $i \neq j$. 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.

Sometimes it is useful to give a name or a label to each part in a graph partition. A graph partition can then be described as a set of labels for the parts and ***partition map*** that maps each vertex to the label of its part. The labels can be chosen arbitrarily but sometimes it is conceptually and computationally easier to use a vertex inside a part as a name (representative) for that part.

Example 17.2. The partition $\{\{\mathtt{a},\mathtt{b},\mathtt{c}\},\{\mathtt{d}\},\{\mathtt{e},\mathtt{f}\}\}$ of the vertices $\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{e},\mathtt{f}\}$, defines three parts as vertex-induced subgraphs.



The edges $\{\mathtt{a},\mathtt{b}\}$, $\{\mathtt{a},\mathtt{c}\}$, and $\{\mathtt{e},\mathtt{f}\}$ are internal edges, and the edges $\{\mathtt{c},\mathtt{d}\}$, $\{\mathtt{b},\mathtt{d}\}$, $\{\mathtt{b},\mathtt{e}\}$ and $\{\mathtt{d},\mathtt{f}\}$ are cut edges.

By labeling the parts "abc", "d" and "ef", we can specify the graph partition with following partition map:
$(\{\mathtt{abc},\mathtt{d},\mathtt{ef}\},\ \{\mathtt{a}\mapsto\mathtt{abc},\mathtt{b}\mapsto\mathtt{abc},\mathtt{c}\mapsto\mathtt{abc},\mathtt{d}\mapsto\mathtt{d},\mathtt{e}\mapsto\mathtt{ef},\mathtt{f}\mapsto\mathtt{ef}\})$.

Instead of assigning a fresh label to each part, we can choose a representative vertex. For example, by picking a, d, and c as representatives, we can represent the partition above using the following partition map

$$(\{\mathtt{a},\mathtt{d},\mathtt{e}\},\{\mathtt{a}\mapsto\mathtt{a},\mathtt{b}\mapsto\mathtt{a},\mathtt{c}\mapsto\mathtt{a},\mathtt{d}\mapsto\mathtt{d},\mathtt{e}\mapsto\mathtt{e},\mathtt{f}\mapsto\mathtt{e}\}).$$

17.1 Graph Contraction

Graph contraction is a 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 is important, since divide-and-conquer can be difficult to apply in graph problems efficiently. This is because divide and conquer usually requires partitioning graphs into smaller graphs in a balanced fashion such that the number of cut edges is minimized. Since graphs can be highly irregular, however, they can be difficult to partition. In fact, graph partitioning problems are typically NP-hard. The rest of this section describes the graph-contraction design technique and two approaches to graph-contraction: edge contraction and star contraction.

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 (Design Technique 17.3) as follows.

Algorithm-Design Technique 17.3. [Graph Contraction]

Base case: If the graph is small (e.g., it has no edges), then compute the desired result.

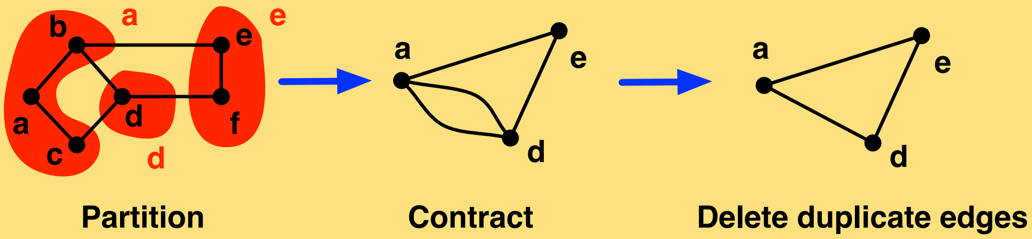
Inductive case:

- Contract the graph into a smaller quotient graph.
  - Partition the graph into parts.
  - Contract each part to a single super-vertex.
  - Drop internal edges.
  - Reroute cut edges to corresponding super-vertices.
- Recursively solve the problem for the quotient graph.
- Using the result for the quotient graph, compute the result for the input graph.

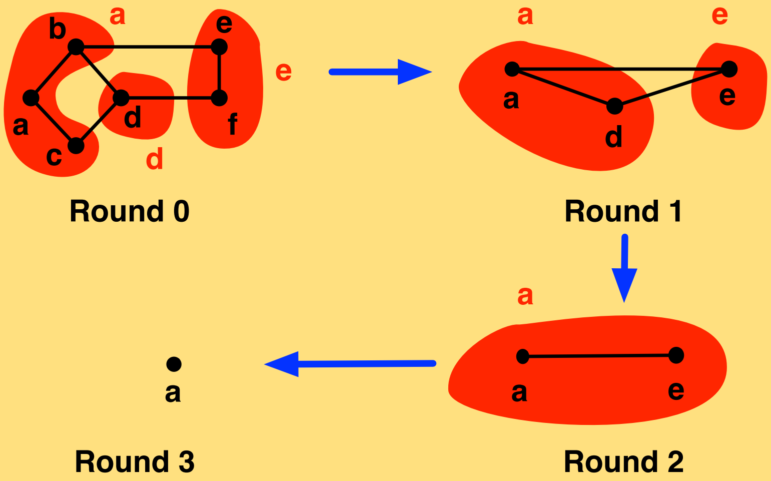
The key step of graph contraction is the construction of the quotient. To this end, we partition the graph and construct a quotient graph, where each part in the partition is represented by a vertex in the quotient graph. We can construct the quotient graph by creating a ***super-vertex*** for each partition We then consider each edge (u, v) in the graph. If the edge is an internal edge, then we skip it. Otherwise, we create a new edge between the super-vertices representing the parts containing u and v. Since there can be many cut edges between two parts, we may create multiple edges between two super-vertices. We can remove such edges or leave them in the graph, in which case we would be working with multigraphs. In this chapter, we shall remove duplicate edges, because this is simpler for our purposes. The process of identifying a partition and updating the edges is called a ***round of graph contraction***. In a graph contraction, rounds are repeated until there are no edges left.

An important property of graph contraction is that it is guided by a graph partition. Since parts in a graph partition are disjoint, each vertex in the graph is mapped to one unique vertex in the quotient graph.

Example 17.4. One round of graph contraction:



Contracting a graph down to a single vertex in three rounds:

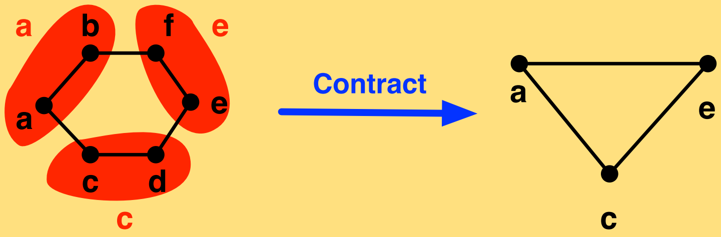


As described, the graph-contraction technique is generic in the kind of graph partition used for constructing the quotient graph. In the rest of this chapter, we will consider two techniques, edge partitioning and star partitioning, and the resulting graph-contraction algorithms.

17.1.1 Edge Partition and Edge Contraction

Edge partitioning is a graph-partitioning technique. In an ***edge partition***, each part is either a single vertex or two vertices connected by an edge. We use the term ***edge contraction*** to refer to a graph contraction performed by using edge partitions.

Example 17.5. An example edge partition in which every part consists of two vertices and an edge between them. Contracting the graph based on this partition yields a quotient graph with half as many vertices and edges.

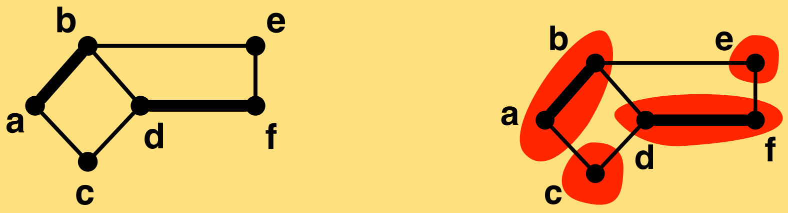


Note that in general, parts cannot be just pairs of vertices, because the graph might not have an even number of vertices, but even if it does (no pun intended), it is likely that it cannot be partitioned into a set of pairs joined by edges.

We can construct an edge partition by selecting an ***independent edge set***, or ***vertex matching***, where no two edges share a vertex, and placing all the remaining vertices that are not incident an a selected edges into singleton sets. The problem of finding a vertex matching is called the ***vertex-matching problem***.

Definition 17.6. 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 share a vertex.

Example 17.7. A vertex matching for our favorite graph (highlighted edges) and the corresponding parts.

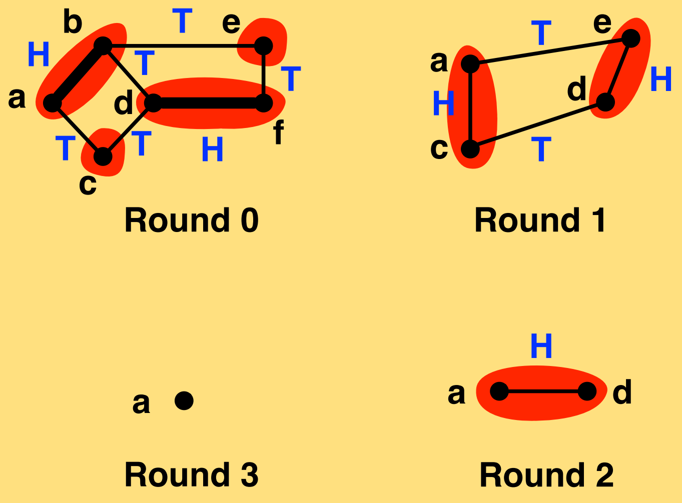


The vertex matching defines four parts (circled), two of them defined by the edges in the matching, $\{$a, b$\}$ and $\{$d, f$\}$, and two of them are the unmatched vertices c and e.

The problem of finding the largest vertex matching for a graph is called the ***maximum vertex matching*** problem. Many algorithms for this well-studied problem have been proposed, including one that can solve the problem in $O(\sqrt{|V|}|E|)$ work. For graph contraction, we do not need a maximum matching but one that it is sufficiently large. For example, we can use a greedy algorithm to construct a vertex matching by going through the edges one by one maintaining an initially empty set M and for each edge, if no edge in M is already incident on its endpoints then add it to M, otherwise toss it. The problem with this approach is that it is sequential since each decision depends on previous decisions. To find the vertex matching in parallel, we will need to make local decisions at each vertex. One possibility is for each vertex to select one of its neighbors to match with. Such a selection can be done 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. We can use randomization to break the symmetry. For example, we can flip a coin for each edge (u, v) in parallel and select the edge, effectively matching u and v, if the coin for the edge comes up heads and all the edges incident on u and v flip tails. This guarantees that a vertex is matched with at most one other vertex.

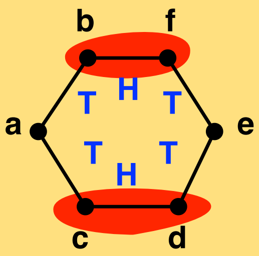
Example 17.8. [Edge contraction]

An example edge contraction illustrated.



Let us analyze how effective this approach is in selecting a reasonably large matching. We first consider cycle graphs, consisting of a single cycle and no other edges. In such a graph every vertex has exactly two neighbors.

Example 17.9. 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 $\{\texttt{c}, \texttt{d}\}$ and $\{\texttt{b}, \texttt{f}\}$ are selected.

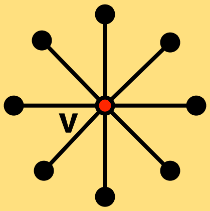
We want to determine the probability that an edge is selected in such a graph. Since the coins are flipped independently at random, and each vertex has degree two, the probability that an edge picks heads and its two adjacent edges pick tails is $\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}$. To analyze the number of edges 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 deleted is $\frac{m}{8}$ (note, $m = n$ in a cycle graph).

In the chapter on randomized algorithms Section **??** we argued 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 $O(\log n)$ 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 $k \log n$ rounds the probability that the problem size is a least 1 is very small. For a cycle graph, this technique leads to an algorithm for graph contraction with linear work and $O(\log^2 n)$ span—left as an exercise.

There are several ways to improve the number of deleted 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 deleted edges to $\frac{m}{4}$. Another way is let each edge pick a random number in some range and then select and 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 edges contracted in a circle to $\frac{m}{3}$.

Although edge contraction works quite well with cycle graphs, or sequentially with the appropriate data structures, it does not work well for arbitrary graphs. The problem is in edge contraction, only one edge incident on a vertex can be contracted in each round. Therefore vertices with high degrees, will have to contract their neighbors one at a time. The example below illustrates a particularly difficult graph, called a star graph, for edge contraction.

Example 17.10. A star graph with center v and eight satellites.



More precisely, we can define a star graph as follows.

Definition 17.11. [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. $E = \{\{v, u\} : u \in V \setminus \{v\}\}$.

Note that a single vertex and a single edge are both star graphs.

It is not difficult to convince ourselves that on a star graph with n vertices—1 center and $n - 1$ satellites—any edge partitioning algorithm will take $\Omega(n)$ rounds. To fix this problem we need to be able to form parts that consist of more than just edges.

Remark 17.12. An abstract data type called disjoint sets is often used to contract graphs sequentially. Disjoint sets supply two functions: ***union***, which joins two components, and ***find***, which finds what component a vertex is in. In our framework, the union operation is simply edge contraction across a single edge, and the find is just a lookup in the partition map. Semantically for a partition map P we can define `union` as:

$$\begin{array}{rcl} \texttt{union}(P, u, v) & = & \{u' \mapsto \textbf{if } (v' = u) \textbf{ then } v \textbf{ else } v' \\ & & \;\; : (u' \mapsto v') \in P\} \end{array}$$

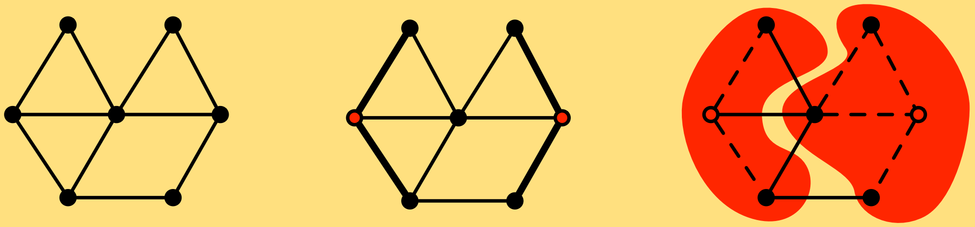
where here we have made v the new representative of the super-vertex $\{u, v\}$, and have updated all vertices that used to point to u to now point to v. Implementing the `union` this way, however, is inefficient since it can require updating a lot of vertices. It turns out that the operations can be can be implemented much more efficiently. Indeed one can implement a data structure that only requires amortized $O(\alpha(n))$ work per operation, where $\alpha(n)$ (the inverse Ackermann function) is a function that is very close ot $O(1)$, and n is the number of operations.

17.1.2 Star Partition and Star Contraction

In an edge partition, if there is an edge incident on a vertex v is selected as a part, then none of the other edges incident on v can be their own part. This limits the effectiveness of edge partitioning, because it is unable to contract significantly graphs with high-degree vertices. In this section, we describe an alternative technique: star partition.

In ***star partitioning***, we partition a graph G by selecting parts of G to correspond to subgraphs of G that are stars. For example in Example 17.13, the whole graph can be selected as a single partition. We refer to a graph contraction where parts are selected by star partitioning a ***star contraction***.

Example 17.13. In the graph shown below on the left, we first find two disjoint stars. We then partition the graph into two parts, each defined as the vertex-induced subgraphs of the two stars.

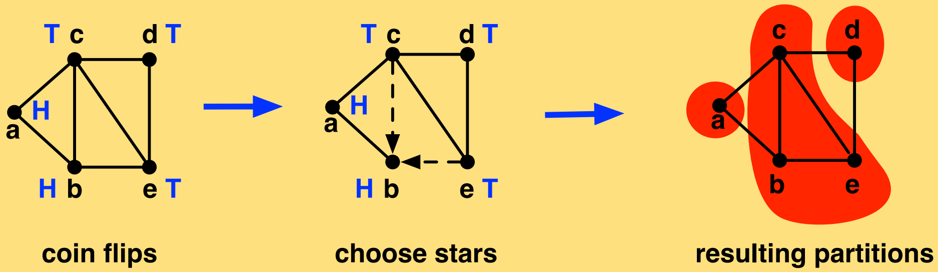


A graph
Its two stars.
Centers highlighed.
The corresponding star partition.
Solid edges are cut edges.
Dashed edges are internal edges.

As with edge partitions, it is possible to construct star partitions sequentially. One approach proceeds by adding start incrementally: start with an arbitrary vertex v and make v the center of a star; then, attach as satellites all the unattached neighbors of v; remove now v and its satellites from the graph, and recursively repeat the same process with the remaining graph. Note that a vertex is the simplest form of a star. So if we have isolated vertices remaining, each will form a ***singleton star*** consisting of a single vertex.

We can also construct star partitions in parallel by making local independent decisions at each vertex. As in edge partitioning, we can use 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 it attempts to become a satellite by finding a neighbor that is a center. If no such neighbor exists (all neighbors have flipped tails or the vertex is isolated), then the vertex becomes a center. If a vertex has multiple centers as neighbors, it can pick one arbitrarily. This approach might not be optimal in the sense that it might not create the smallest number of stars but, as we shall see, this is acceptable for our purposes, because we only need to reduce the size of the graph by a constant factor.

Example 17.14. An example star partition. 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 flipped heads, then it becomes a singleton star (e.g., vertex d). We end up with 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 parts, which consists of the subgraphs induced by each star.



To specify the star-partition algorithm, we need a source of randomness. For this, we assume that each vertex is given a (potentially infinite) sequence of random and independent coin flips. The i^{th} element of the sequence can be accessed via the function

$$\texttt{heads}(v, i) : V \times \mathbb{Z} \to \mathbb{B},$$

which returns `true` if the i^{th} flip on vertex v is heads and false otherwise. Since most machines don't have true sources of randomness, in practice this can be implemented with a pseudorandom number generator or even with a good hash function.

Algorithm 17.15 illustrates the code for star partitioning. The function `starPartition` takes as argument a graph and a round number, and returns 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 directed 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. More specifically, the union is shorthand for the code

$$\texttt{Set.reduce (Table.union } (\boldsymbol{\lambda}\,\texttt{(x,y) => x)) } \emptyset \ \{\{u \mapsto v\} : (u, v) \in TH\}.$$

This completes the selection of satellites and their centers. The algorithm next determines the set of super vertices, which can become the vertices of the quotient graph in contraction, as all the vertices minus the satellites. To complete the process, the algorithm determines each super vertex and matches it with itself (Line 10). This effectively promotes 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.

Algorithm 17.15. [Star Partition]

```
starPartition $(G = (V, E), i)$ =
let
  (* Find the arcs from satellites to centers *)
  $TH$ = $\{(u, v) \in E \mid \neg \texttt{heads}(u, i) \wedge \texttt{heads}(v, i)\}$

  (* Partition map from satellites to centers *)
  $P$ = $\bigcup_{(u,v) \in TH} \{u \mapsto v\}$

  (* Super vertices are centers and unmatched ones *)
  $V'$ = $V \setminus \texttt{domain}(P)$

  (* Map super-vertices to themselves *)
  $P'$ = $\{u \mapsto u : u \in V'\}$

in $(V', P' \cup P)$ end
```

Example 17.16. The star-partition algorithm proceeds on the graph in Example 17.14 as follows. First, it computes

$$TH = \{(\texttt{c}, \texttt{a}), (\texttt{c}, \texttt{b}), (\texttt{e}, \texttt{b})\},$$

as the edges from satellites to centers. Now, it converts each edge into a singleton table, and merges all the tables into the table, which is going to become part of the partition map:

$$P = \{\texttt{c} \mapsto \texttt{b}, \texttt{e} \mapsto \texttt{b}\}.$$

Note that the edge $(\texttt{c}, \texttt{a})$ has been removed since when uniting the tables, we selects only one element for each key in the domain. Now for all remaining vertices $V' = V \setminus \texttt{domain}(P) = \{\texttt{a}, \texttt{b}, \texttt{d}\}$ we map them to themselves, giving:

$$P' = \{\texttt{a} \mapsto \texttt{a}, \texttt{b} \mapsto \texttt{b}, \texttt{d} \mapsto \texttt{d}\}.$$

The vertices in P' are the centers. Finally we merge P and P' to give to construct the partition map

$$P' \cup P = \{\texttt{a} \mapsto \texttt{a}, \texttt{b} \mapsto \texttt{b}, \texttt{c} \mapsto \texttt{b}, \texttt{d} \mapsto \texttt{d}, \texttt{e} \mapsto \texttt{b}\}.$$

Analysis of Star Partitioning. By examining the algorithm, we can conclude the following work and span bounds for the star partitioning algorithm.

Theorem 17.17 (Star Partitioning)**.** *Based on the array-based cost specification for sequences and single-threaded sequences, the cost of* `starPartition` *is $O(n + m)$ work and $O(\log n)$ span for a* *graph with n vertices and m edges.*

Let's 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 computed by the algorithm. The following lemma shows that on a graph with $n_\bullet$ non-isolated vertices, the size of P and thus the number of satellites is at least $n_\bullet/4$ in expectation.

Lemma 17.18. For a graph G with $n_\bullet$ non-isolated vertices, the expected number of satellites in a call to `starPartition` (G, r) with any r 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 $v \in$ `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, $\mathbf{Pr}\left[R_v\right] \geq \mathbf{Pr}\left[T_v\right] \mathbf{Pr}\left[H_u\right] = 1/4$. By the linearity of expectation, the expected number of satellites is

$$\begin{aligned}\mathbf{E}\left[\sum_{v:v \text{ non-isolated}} \mathbb{I}\left\{R_v\right\}\right] &= \sum_{v:v \text{ non-isolated}} \mathbf{E}\left[\mathbb{I}\left\{R_v\right\}\right] \\ &\geq n_\bullet/4.\end{aligned}$$

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. □

Analysis of Graph Contraction with Star Partitioning. For the analysis of star contraction, i.e., graph contraction with star partitioning, 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.

$$S(n_\bullet) = \begin{cases} S(n'_\bullet) + O(\log n) & \text{if} \quad n_\bullet > 0 \\ 1 & \text{otherwise.} \end{cases}$$

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. Since $\mathbf{E}\left[X\right] = n_\bullet/4$, $\mathbf{E}\left[n'_\bullet\right] = 3n/4$. This is a familiar recurrence, which we know solves to $O(\log^2 n_\bullet)$, and thus $O(\log^2 n)$, in expectation.

As for work, ideally, 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 one can remove a constant fraction of the non-isolated vertices on one round of star contraction, we have not shown anything about how many edges we remove. Since 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. Consider, for example, following sequence of rounds.

round	vertices	edges
1	n	m
2	$n/2$	$m - n/2$
3	$n/4$	$m - 3n/4$
4	$n/8$	$m - 7n/8$

In this example, 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 $O(m \log n)$. Indeed, this is the best bound we can show asymptotically.

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:

$$W(n_\bullet, m) \leq \begin{cases} W(n'_\bullet, m) + O(n_\bullet + m) & \text{if} \qquad n_\bullet > 1 \\ 1 & \text{otherwise.} \end{cases}$$

This recursion solves to $\mathbf{E}\left[W(n_\bullet, m)\right] = O(n_\bullet + m \log n_\bullet) = O(n + m \log n)$. 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 $O(n \log n)$. This analysis gives us the following theorem.

Theorem 17.19. *For a graph $G = (V, E)$, we can contract the graph into a number of isolated vertices in $O(|V| + |E| \log |V|)$ work and $O(\log^2 |V|)$ span.*

17.2 Connectivity via Graph Contraction

The ***(graph) connectivity*** problem requires determining the connected components of a graph.

Problem 17.20. [The Graph Connectivity (GC) Problem] Given an undirected graph $G = (V, E)$ return all of its connected components (maximal connected subgraphs).

A graph connectivity algorithm could return the connected components of a graph, by, for example, specifying the set of vertices in each component.

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. This creates the first component. Move onto the next vertex, and if it has not already been searched, then search from that vertex to create the second component. Repeat until all the vertices are considered.

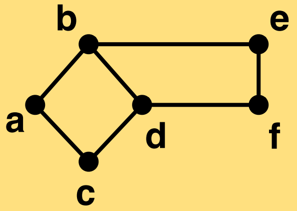
Algorithm 17.22. [Counting Components using Graph Contraction]

```
1 countComponents (G = (V, E)) =
2    if |E| = 0 then |V|
3    else
4       let
5          (V', P) = starPartition (V, E)
6          E' = {(P[u], P[v]) : (u, v) ∈ E | P[u] ≠ P[v]}
7       in
8          countComponents (V', E')
9       end
```

Using graph search leads to perfectly sensible sequential algorithms for graph connectivity, but they are not good parallel algorithms. When the diameter of a graph is small, we may use BFS to perform each graph search, but we still have to iterate over the components one by one. Thus the span in the worst case can be linear in the number of components, which can be large.

We would like to find a parallel algorithm for connectivity that has a small span an all graphs. To this end, we use the graph-contraction technique with star partitioning. To specify the algorithm, 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.

Example 17.21. The representation of an undirected graph as a set of ordered pairs, with each edge appearing in both directions.



$$
\begin{aligned}
V &= \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}, \texttt{f}\} \\
E &= \{(\texttt{a}, \texttt{b}), (\texttt{b}, \texttt{a}), (\texttt{b}, \texttt{d}), (\texttt{b}, \texttt{e}), (\texttt{e}, \texttt{b}), (\texttt{d}, \texttt{b}), (\texttt{d}, \texttt{f}), (\texttt{a}, \texttt{c}), \\
&\qquad (\texttt{c}, \texttt{a}), (\texttt{c}, \texttt{d}), (\texttt{d}, \texttt{c}), (\texttt{d}, \texttt{f}), (\texttt{f}, \texttt{d}), (\texttt{e}, \texttt{f}), (\texttt{f}, \texttt{e})\}
\end{aligned}
$$

Algorithm 17.22 illustrates a graph-contraction algorithm for determining the number of connected components in a graph. Each contraction on Line 5 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 6 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 $P[u] \neq P[v]$.

Having computed the quotient graph, the algorithm recursively solves the problem on it. Recursion bottoms out when the graph contains no edges, in which case, each component has been contracted down to a singleton vertex, and thus the number of vertices in the contracted graph is equal to the number of components in the input graph.

Example 17.23. The values of V', P, and E' after each round of the contraction shown in Example 17.4.

$$\begin{array}{lrcl}
 & V' & = & \{\mathtt{a}, \mathtt{d}, \mathtt{ef}\} \\
\text{round 1} & P' & = & \{\mathtt{a} \mapsto \mathtt{a}, \mathtt{b} \mapsto \mathtt{a}, \mathtt{c} \mapsto \mathtt{a}, \mathtt{d} \mapsto \mathtt{d}, \mathtt{e} \mapsto \mathtt{e}, \mathtt{f} \mapsto \mathtt{e}\} \\
 & E' & = & \{(\mathtt{a}, \mathtt{e}), (\mathtt{e}, \mathtt{a}), (\mathtt{a}, \mathtt{d}), (\mathtt{d}, \mathtt{a}), (\mathtt{d}, \mathtt{e}), (\mathtt{e}, \mathtt{d})\} \\
 & & & \\
 & V' & = & \{\mathtt{a}, \mathtt{e}\} \\
\text{round 2} & P' & = & \{\mathtt{a} \mapsto \mathtt{a}, \mathtt{d} \mapsto \mathtt{abcd}, \mathtt{e} \mapsto \mathtt{e}\} \\
 & E' & = & \{(\mathtt{a}, \mathtt{e}), (\mathtt{e}, \mathtt{a})\} \\
 & & & \\
 & V' & = & \{\mathtt{a}\} \\
\text{round 3} & P' & = & \{\mathtt{a} \mapsto \mathtt{a}, \mathtt{e} \mapsto \mathtt{a}\} \\
 & E' & = & \{\}
\end{array}$$

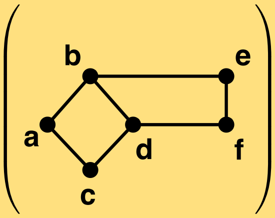
Our previous algorithm just counted the number of components. We can modify the algorithm slightly to compute the components themselves instead of returning their count. To this end, we are going to construct the mapping from vertices to their components recursively. This is possible because we can obtain the mapping by composing the mapping from vertices to their super-vertices and the mapping from super-vertices to their components, which we obtain recursively. Algorithm 17.24 shows the algorithm.

Algorithm 17.24. [Contraction-based graph connectivity]

```
1  connectedComponents $(G = (V, E)) =$
2    if $|E| = 0$ then
3      $(V, \{v \mapsto v : v \in V\})$
4    else
5      let
6        $(V', P) =$ starPartition $(V, E)$
7        $E' = \{(P[u], P[v]) : (u, v) \in E \mid P[u] \neq P[v]\}$
8        $(V'', C) =$ connectedComponents $(V', E')$
9      in
10       $(V'', \{v \mapsto C[s] : (v \mapsto s) \in P\})$
11   end
```

Example 17.25.

`connectedComponents`



might return:

$$(\{\mathtt{a}\},\ \{\mathtt{a} \mapsto \mathtt{a}, \mathtt{b} \mapsto \mathtt{a}, \mathtt{c} \mapsto \mathtt{a}, \mathtt{d} \mapsto \mathtt{a}, \mathtt{e} \mapsto \mathtt{a}, \mathtt{f} \mapsto \mathtt{a}\})$$

since 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.

The only differences from `countComponents` are a modification to the base case, and the extra line (Line 10) after the recursive call. 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, when returning from the recursion, Line 10 updates the mapping P from vertices to super-vertices by looking up the component that the super-vertex belongs to, which is given by C. This simply involves the look up $C[s]$ for every $(v \mapsto s) \in P$. Note that if you view a mapping as a function, then this is equivalent to function composition, i.e., $C \circ P$.

Example 17.26. Consider our example graph (Example 17.25), and assume that `starPartition` returns:

$$\begin{aligned} V' &= \{\mathtt{a},\mathtt{d},\mathtt{e}\} \\ P &= \{\mathtt{a} \mapsto \mathtt{a}, \mathtt{b} \mapsto \mathtt{a}, \mathtt{c} \mapsto \mathtt{a}, \mathtt{d} \mapsto \mathtt{d}, \mathtt{e} \mapsto \mathtt{e}, \mathtt{f} \mapsto \mathtt{e}\}. \end{aligned}$$

This pairing corresponds to the case where a, d and e are chosen an centers. Since the graph is connected, the recursive call to `connectedComponents`(V', E') will map all vertices in V' to the same vertex. Lets say this vertex is `a` giving:

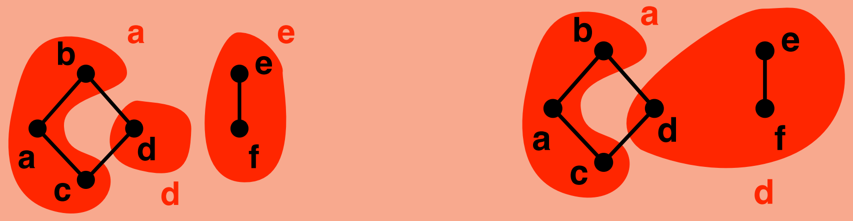
$$\begin{aligned} V'' &= \{\mathtt{a}\} \\ P' &= \{\mathtt{a} \mapsto \mathtt{a}, \mathtt{d} \mapsto \mathtt{a}, \mathtt{e} \mapsto \mathtt{a}\} \ . \end{aligned}$$

Now $\{v \mapsto P'[s] : (v \mapsto s) \in P\}$ 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:

$$\{\mathtt{a} \mapsto \mathtt{a}, \mathtt{b} \mapsto \mathtt{a}, \mathtt{c} \mapsto \mathtt{a}, \mathtt{d} \mapsto \mathtt{a}, \mathtt{e} \mapsto \mathtt{a}, \mathtt{f} \mapsto \mathtt{a}\} \ .$$

Cost of connected-components algorithms.

Remark 17.27. In general the graph contraction techniques does not specify how to partition the graph. In this chapter, we considered two techniques for graph partitioning. Depending on the problem, other techniques can be used. For graph contraction to be applicable to a problem, however, it is important that the quotient graph satisfy certain properties. For example, when solving graph connectivity with the algorithms described here, we have to be careful that the graph partition maintains connectivity: a subgraph should be connected in the quotient graph, if and only if it was connected in the input graph. To ensure this, we will need to use a graph-partition algorithm that ensures that each part is connected in the input graph. For example, the pictures below illustrate two graph partitions. The first graph partition maintains connectivity, the second one does not.



The partitioning on the left is appropriate for graph contraction since each partition is connected. The partition on the right is not since `d` is not connected to `e` and `f`.

17.3 Forest Contraction and Tree Contraction

Suppose that we want to contract a forest of trees instead of a general graph as we considered thus far. Since forests are graphs, we can use the same `starPartition` algorithm to contract the forest. Since a forest of n vertices has at most $n - 1$ edges, we obtain a better work bound, because the number of edges decrease geometrically (in expectation) in each round, as do the number of vertices. The overall expected work is therefore a geometric sum of the form:

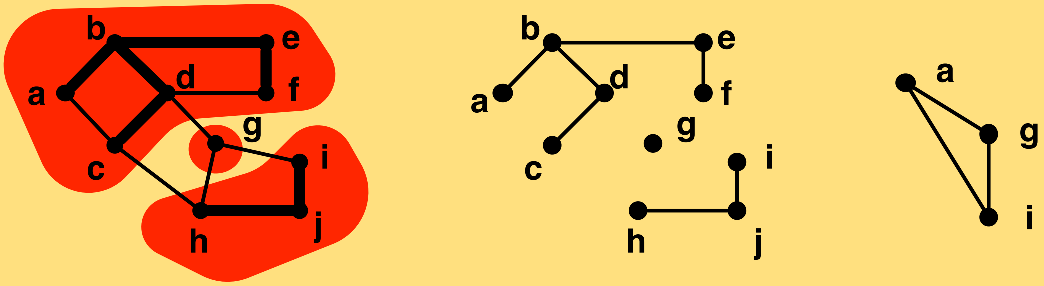
$$\mathbf{E}\left[W(n,m)\right] = \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^{i} kn = O(n),$$

instead of $O(m \log n)$ for general graphs. The span is not affected. The same bound applies if the forest is connected, and thus is a tree.

For a graph $G = (V, E)$ consider a subset of edges $F \subset E$ that forms a forest (i.e., has no cycles). Such a set of edges partitions the graph G, where parts are defined as the subgraphs induced by the trees in F. We can thus contract a graph by identifying a forest F, and then use `connectedComponents` (V, T), which does linear work as explained above, instead of our `starPartition` routine. This corresponds to ***tree partitioning*** instead of star or edge partitioning, which are special kinds of trees. We will use this idea in an algorithm for Minimum

Spanning Trees described in Chapter 18.

Example 17.28. A graph and a subset of the edges F (highligted) consisting of three disjoint trees illustrated in the middle diagram. Each tree induces a part in the original graph (red blobs).



If we run `connectedComponents` on F, then are left with the desired partitioning with super-vertices $\{\texttt{a}, \texttt{g}, \texttt{i}\}$ and the mapping:

$$\{\texttt{a} \mapsto \texttt{a}, \texttt{b} \mapsto \texttt{a}, \texttt{c} \mapsto \texttt{a}, \texttt{d} \mapsto \texttt{a}, \texttt{e} \mapsto \texttt{a}, \texttt{f} \mapsto \texttt{a}, \texttt{g} \mapsto \texttt{g}, \texttt{h} \mapsto \texttt{i}, \texttt{i} \mapsto \texttt{u}, \texttt{j} \mapsto \texttt{i}\}.$$

Using this partition, we can compute a quotient graph in the usual way by re-routing edges to the super-vertices. The resulting quotient graph is illustrated on the right.

17.4 Problems

17-1
There are 18 subgraphs for a triangle consisting of three vertices and three edges connecting them, including the empty graph and the graph itself. List them all.

17-2
In star contraction, what is the probability that a vertex with degree d is removed.

17-3
Find an example graph, where star-based graph contraction removes a small number of edges on each round.

17-4
Describe how to construct a graph that exhibits the worst-case behavior for Theorem 17.19.

17-5
Is the star contraction algorithm work-optimal for a dense graph with $\Omega(n^2)$ edges? Prove or disprove.

Chapter 18

Minimum Spanning Trees

In this chapter we cover a important graph problem, Minimum Spanning Trees (MST). The MST of an undirected, weighted graph is a tree that spans the graph while minimizing the total weight of the edges in the tree. We first define spanning tree and minimum spanning trees precisely and then present two sequential algorithm and one parallel algorithm, which are respectively Kruskal's, Prim's, and Borůvka's. All of these algorithms utilize an important cut property, which we also desribe.

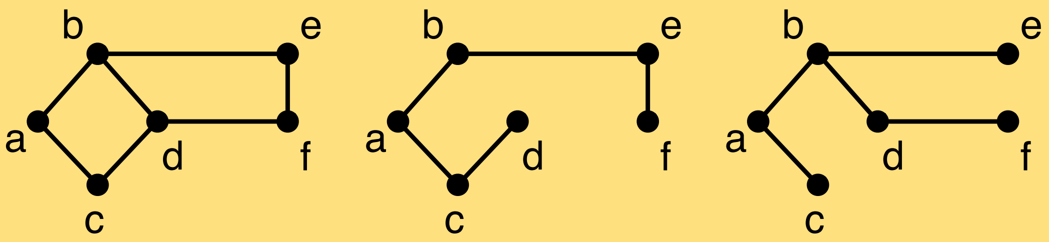
18.1 Minimum Spanning Trees

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 "touching" all the vertices. We call such a tree a spanning tree.

Definition 18.1. For a connected undirected graph $G = (V, E)$, a spanning tree is a tree $T = (V, E')$ with $E' \subseteq E$.

Note that a graph can have many spanning trees, but all have $|V|$ vertices and $|V| - 1$ edges.

Example 18.2. A graph on the left, and two possible spanning trees.



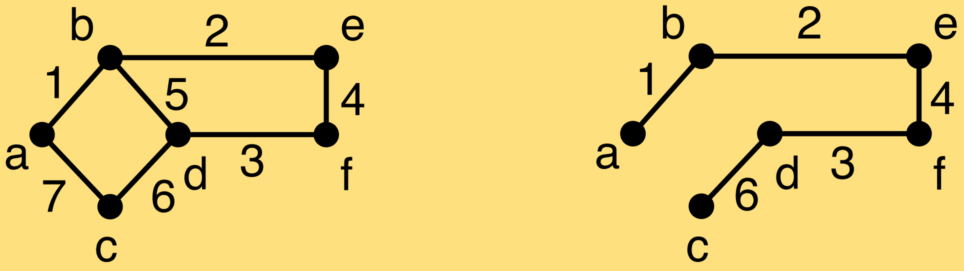
One way to generate a spanning tree is simply to do a graph search. For example the DFS-tree of a DFS is a spanning tree, as it finds a path from a source to all the vertices. Similarly, we can construct a spanning tree based on BFS, by adding each edge that leads to the discovery of an unvisited vertex to the tree. DFS and BFS are work-efficient algorithms for computing spanning trees but as we discussed they are not good parallel algorithms. Another way to generate a spanning tree is to use graph contraction, which as we have seen can be done in parallel. The idea is to use star contraction and add all the edges that are selected to define the stars throughout the algorithm to the spanning tree.

Recall that a graph has many spanning trees. 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 18.3. 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 18.4. A graph (left) and its MST (right).



Example 18.5. Minimum spanning trees have many interesting applications. One example concerns the design of a network. 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 the possible connection between rooms as a graph, where vertices represent rooms and weighted edges represent possible connections along with their cost (weight). To minimize the cost of the wiring, you could find a minimum spanning tree of the graph.

18.2 Algorithms for Minimum Spanning Trees

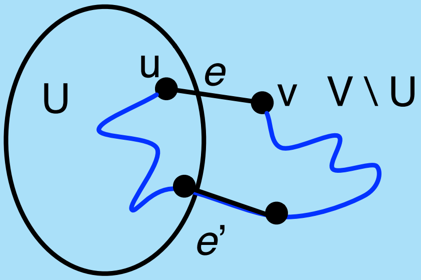
There are several algorithms for computing minimum spanning trees. They all, however, are based on the same underlying property about cuts in a graph, which we will refer to as the ***light-edge property***. Intuitively, the light-edge property (precisely defined below) states that if you partition the graph into two, the minimum edge between the two parts has to be in the MST. The light-edge property gives a way to identify algorithmically the edges of an MST.

In our discussion we will assume that all edges have distinct weights. This assumption causes no loss-of-generality, because light-edge property allows us to break ties arbitrarily, which we can take advantage of by for example breaking ties based on some arbitrary ordering of edges such as their position in the input. A simplifying consequence of this assumption is that the MST of a graph with distinct edge weights is unique.

Definition 18.6. [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 $(U, V \setminus U)$, which is called the ***cut***. We refer to the edges between the two parts as the ***cut edges*** written $E(U, \overline{U})$, where $\overline{U} = V \setminus U$.

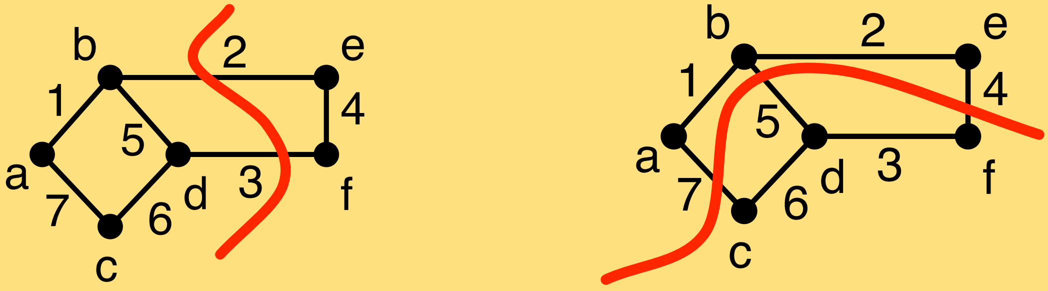
The subset U used in the definition of a cut might include a single vertex v, in which case the cut edges would be all edges incident on v. But the subset U must be a proper subset of V (i.e., $U \neq V$). We sometimes say that a cut edge ***crosses*** the cut.

Lemma 18.8. [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 in the minimum spanning tree MST(G) of G.



Proof. The proof is by contradiction. Assume the minimum-weighted edge $e = (u, v)$ is not in the MST. Since the MST spans the graph, there must be some simple path P connecting u and v in the MST (i.e., consisting of just edges in the MST). The path must cross the cut between U and $V \setminus U$ at least once since u and v are on opposite sides. Let e' be an edge in P that crosses the cut. By assumption the weight of e' is larger than that of e. Now, insert e into the graph—this gives us a cycle that includes both e and e'—and remove e' from the graph to break the only cycle and obtain a spanning tree again. Now, since the weight of e is less than that of e', the resulting spanning tree has a smaller weight. This is a contradiction and thus e must have been in the tree. □

Example 18.7. Two example cuts. For each cut, we can find the lightest edge that crosses that cut, which are the edges with weight 2 (left) and 4 (right) respectively.



An important implication of the light-edge property as proved in Lemma 18.8 is that any minimum-weight edge that crosses a cut can be immediately added to the MST. In fact, all of the three algorithms that we will consider in this chapter 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. Borůvka's algorithm constructs a tree in parallel by considering the cut defined by each and every vertex. In the next section, we briefly review Kruskal's and

Prim's algorithm and spend most of our time on a parallel variant of Borůvka's algorithm.

Remark 18.9. Even though Borůvka'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.

18.2.1 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".

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 as described in the last chapter 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.

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***.

Algorithm 18.10. [Union-Find Kruskal]

```
kruskal $(G = (V, E, w))$ =
  let
    $U$ = iterate insert $\emptyset$ $V$                    (* initialize UF *)
    $E'$ = sort $(E, w)$                               (* sort the edges *)
    addEdge $((U, T),\ e = (u, v))$ =
      let
        $u'$ = find $(U, u)$
        $v'$ = find $(U, v)$
      in
        if $(u' = v')$ then
          $(U, T)$          (* if $u$ and $v$ are connected, skip *)
        else
          (* contract $e$ and add it to $T$ *)
          (union $(U, u', v'), T \cup e$ )

     end
  in
    iterate addEdge $(U, \emptyset)$ $E'$
  end
```

The ADT for a union-find data structure consists of the following operations on a union-find structure U:

- insert$U$ v inserts the vertex v into U,
- union U (u, v) joins the two elements u and v into a single super-vertex,
- find$U$ v returns the super-vertex in which v belongs, possibly itself,
- equals$u$ v returns true if u and v are the same super-vertex. Now we can simply process the edges in increasing order. This idea gives Algorithm 18.10.

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.

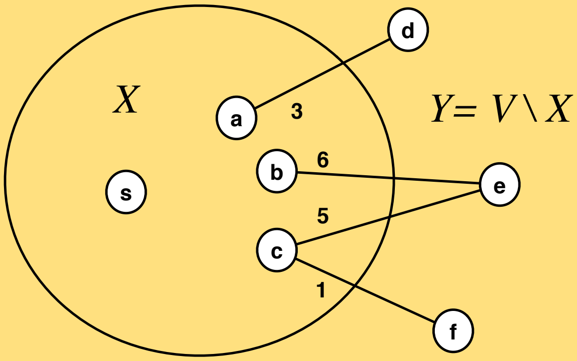
18.2.2 Prim's Algorithm

Prim's algorithm performs a priority-first search to construct the minimum spanning tree. The idea is that if we have already visited a set X, then by the light-edge property the minimum weight edge with one of its endpoint in X and the other in $V \setminus X$ must be in the MST (it is a minimum cross edge from X to $V \setminus X$). We can therefore add it to the MST and include the other endpoint in X. This leads to the following definition of Prim's algorithm:

Algorithm 18.11. [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 $T = \emptyset$, using priority $p(v) = \min_{x \in X} w(x, v)$ (to be minimized), and setting $T = T \cup \{(u, v)\}$ when visiting v where $w(u, v) = p(v)$.

When the algorithm terminates, T is the set of edges in the MST.

Example 18.12. A step of Prim's algorithm. Since the edge $(\texttt{c}, \texttt{f})$ has minimum weight across the cut (X, Y), the algorithm will "visit" $\texttt{f}$ adding $(\texttt{c}, \texttt{f})$ to T and $\texttt{f}$ to X.



Exercise 18.13. Carefully prove the correctness of Prim's algorithm by induction.

Interestingly this algorithm is quite similar to Dijkstra's algorithm for shortest paths. The only differences are (1) we start at an arbitrary vertex instead of at a source, (2) that $p(v) = \min_{x \in X}(x, v)$ instead of $\min_{x \in X}(d(x) + w(x, v))$, and (3) we maintain a tree T instead of a table of distances $d(v)$. Because of the similarity we can basically use the same priority-queue implementation as in Dijkstra's algorithm and it runs with the same $O(m \log n)$ work bounds.

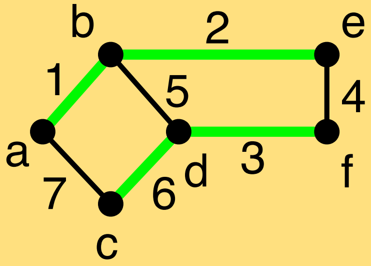
Remark 18.14. Prim's algorithm was invented in 1930 by Czech mathematician Vojtech Jarnik and later independently in 1957 by computer scientist Robert Prim. Edsger Dijkstra's rediscovered it in 1959 in the same paper he described his famous shortest path algorithm.

18.2.3 Borůvka'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 Borůvka. As Kruskal's and Prim's, Borůvka's algorithm constructs the MST by inserting light edges but unlike them, it inserts many light edges at once.

To see how we can select multiple light edges, recall that 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 an any vertex is in the MST. We call such edges *vertex-joiners*.

Example 18.15. The vertex-joiners of the graph are highlighted.

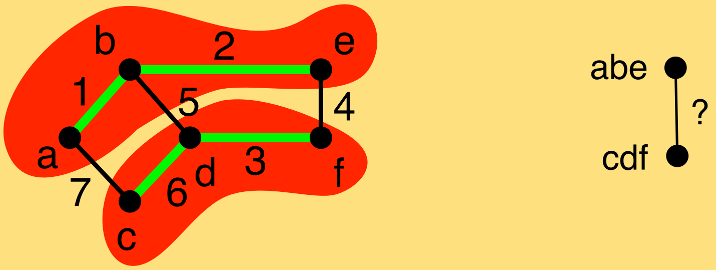


Since we know that the vertex-joiners are all in the MST, we can insert them into it in parallel by letting each vertex pick its vertex-joiner. In the example above, the vertices a and b both pick edge {a, b}, vertex c picks {c, d}, d, vertex f picks {d, f}, and e picks {e, b}.

Sometimes just one round of picking vertex-joiners will select all the MST edges and would generate a complete solution. In most cases, however, the minimum-weight edges on their own do not form a spanning tree. In the example above, the edge (e, f), which is in the MST, is not selected (neither e nor f pick it). To see how we can proceed, note that the vertex-joiners define a partition of the graph—all the vertices are in a part. Consider now the edges that remain internal to a partition but are not vertex joiners. Such an edge cannot be in the MST, because inserting it into the MST would create a cycle. The edges that cross the partitions, however, must be considered as they can indeed be in the MST.

One way to eliminate the internal edges from consideration, while keeping the cross edges is to perform a graph contraction based on the partitioning defined by the vertex-joiners. Recall that in graph contraction, all we need is a partitioning of the graph into disjoint connected subgraphs. Given such a partitioning, we then replace each subgraph (partition) with a super-vertex and relabel the edges. This is repeated until no edges remain.

Example 18.16. Contraction along the minimum edges. Note that there are redundant edges between the two partitions.



When performing graph contraction, we have to be careful about redundant edges. In our discussion of graph contraction in Chapter 17, used 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.

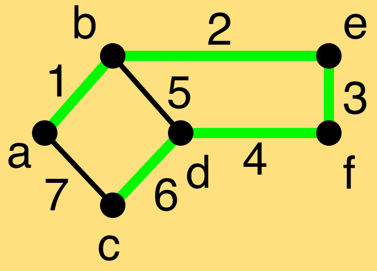
What we just covered is exactly Borůvka's idea. He did not discuss implementing the contraction in parallel. At the time, there were not any computers let alone parallel ones. We are glad that he has left us something to do. In summary, Borůvka's algorithm can be described as follows.

Algorithm 18.17. [Borůvka] 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.

Cost of Borůvka by using tree contraction. We now consider the efficiency of this algorithm. We first focus on the number of rounds of contraction and then consider how to implement the contraction. Since contracting an edge removes exactly one vertex (contraction of and edge can be viewed as folding one endpoint into the other), if k edges are selected then k vertices are removed. Since each vertex picks a vertex joiner 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 joiners. Therefore at least $n/2$ vertex joiners are picked and thus $n/2$ vertices will be removed. Consequently, Borůvka's algorithm will take at most $\log_2 n$ rounds of selecting vertex-joiners and contracting based on the partitioning defined by them.

To contract the partition defined by the vertex-joiners, we cannot use edge or star contraction, because the parts may not correspond to an edge or a star.

Example 18.18. An example where minimum-weight edges give a non-star tree. Note that we have in fact picked a minimum spanning tree in one round by picking the vertex joiners for each vertex.

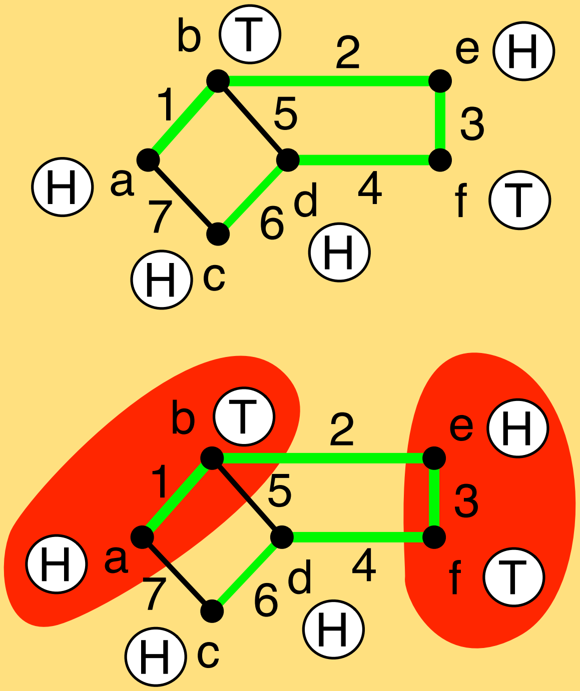


It turns out, vertex joiners form a forest (a set of trees). Therefore, each part is a tree and thus we want to contract trees. By removing all edges that are not vertex-joiners, we can contract a part by applying star contraction to it. Furthermore, since when doing a star contraction on a tree, it remains a tree on each step, 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. As discussed earlier for redundant edges we want to keep the minimum weight such edge. There are various ways to do this, including keeping the redundant edges. Keeping the edges turns out to be an effective solution, and allows the updating the edges to be done in $O(m)$ work. Assuming redundant edges, the minimum into each component can still be done with $O(m)$ work, as described below. Since there are at most $\log n$ rounds, Borůvka's algorithm will run in $O(m \log n)$ work and $O(\log^3 n)$ span.

Cost of Borůvka by using star contraction. We now describe how to improve the span of Borůvka by a logarithmic factor by interleaving steps of star contraction with steps of finding the vertex-joiners, instead of fully contracting the trees defined by the vertex-joiners. The idea is to apply randomized star contraction on the subgraph induced by the vertex-joiners, instead of considering the whole graph as in conventional star contraction. Intuitively, this is correct because we only have to care about vertex-joiners (all other edges cannot be in the MST). As we will show, on each round, we will still be able to reduce 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 18.19. An example of Borůvka with star contraction.



For a set of vertex-joiners jE, consider the subgraph $H = (V, jE)$ of G and apply one step of the star contraction on H. To apply star contraction, we can modify our `starContract` routine so that after flipping coins, the tails only hook across their minimum-weight edge. The modified algorithm for star contraction is as follows. In the code w stands for the weight of the edge (u, v).

Algorithm 18.20. [Star Contraction along Vertex-Joiners]

```
joinerStarContract $(G = (V, E), i)$ =
  let
    jE = vertexJoiners $(G)$
    $P$ = $\{u \mapsto (v, w) \in$ jE $\mid \neg$heads$(u, i) \wedge$ heads$(v, i)\}$
    $V'$ = $V \setminus$ domain$(P)$
  in
    $(V', P)$
  end
```

where (vertexJoiners$G$) finds the vertex-joiners out of each vertex v.

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 still removes $n/4$ vertices in expectation on each step:

Lemma 18.21. For a graph G with n non-isolated vertices, let X_n be the random variable indicating the number of vertices removed by joinerStarContract(G, r). Then, $\mathbf{E}\left[X_n\right] \geq n/4$.

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 jE. 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 $v \in$ 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 $(v, u) \in$ minE. 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, $\mathbf{Pr}\left[R_v\right] \geq \mathbf{Pr}\left[T_v\right] \mathbf{Pr}\left[H_u\right] = 1/4$. By the linearity of expectation, we have that the number of removed vertices is

$$\mathbf{E}\left[\sum_{v:v \text{ non-isolated}} \mathbb{I}\left\{R_v\right\}\right] = \sum_{v:v \text{ non-isolated}} \mathbf{E}\left[\mathbb{I}\left\{R_v\right\}\right] \geq n/4$$

since we have n vertices that are non-isolated. □

This means that this MST algorithm will take only $O(\log n)$ rounds, just like our other graph contraction algorithms.

Final Things. There is a little bit of trickiness since, as the graph contracts, the endpoints of each edge changes. Therefore, if we want to return the edges of the minimum spanning tree,

Algorithm 18.22. [Borůvka’s based on Star Contraction]

```
vertexJoiners $E$ =
  let
    $ET = \{(u,v,w,l) \mapsto \{u \mapsto (v,w,l)\} : (u,v,w,l) \in E\}$
    joinEdges $((v_1,w_1,l_1),(v_2,w_2,l_2)) =$
      if $(w_1 \leq w_2)$ then $(v_1,w_1,l_1)$ else $(v_2,w_2,l_2)$
  in
    reduce (union joinEdges) {} $ET$
  end


joinerStarContract $(G = (V,E), i)$
  let
    $minE$ = vertexJoiners $G$
    $P = \{(u \mapsto (v,w,\ell)) \in minE \mid \neg\texttt{heads}(u,i) \wedge \texttt{heads}(v,i)\}$
    $V' = V \setminus \texttt{domain}(P)$
    in
      $(V',P)$
    end


MST $((V,E),T,i) =$
  if $(|E| = 0)$ then $T$
  else
  let
    $(V',PT) =$ joinerStarContract $((V,E),i)$
    $P = \{u \mapsto v : u \mapsto (v,w,\ell) \in PT\} \cup \{v \mapsto v : v \in V'\}$
    $T' = \{\ell : u \mapsto (v,w,\ell) \in PT\}$
    $E' = \{(P[u],P[v],w,l) : (u,v,w,l) \in E \mid P[u] \neq P[v]\}$
  in
    MST $((V',E'),T \cup T',i+1)$
  end
```

they might not correspond to the original endpoints. To deal with 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` × `vertex` × `weight` × `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 `joinerStarContract` that appears in Algorithm 18.22.

The function `vertexJoiner`(G) in Line 12 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. By Lemma 18.8, since all these edges are minimum out of the vertex, they are safe to add to the MST. Line 13 then picks from these edges the edges that go from a tail to a head, and therefore generates a mapping from tails to heads along minimum edges, creating stars. Finally, Line 14 removes all vertices that are in this mapping to star centers.

This 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 in Algorithm 18.22 The MST algorithm is called by running `MST`$(G, \emptyset, r)$. As an aside, we know that T is a spanning forest on the contracted nodes.

Finally we describe how to implement `minEdges`(G), which returns for each vertex the minimum edge incident on that vertex. There are various ways to do this. One way is to make a singleton table for each edge and then merge all the tables with an appropriate function to resolve collisions. Algorithm 18.22 gives code that merges edges by taking the one with lighter edge weight.

If using sequences for the edges and vertices an even simpler way is to presort the edges by decreasing weight and then use `inject`. Recall that when there are collisions at the same location `inject` will always take the last value, which will be the one with minimum weight.

18.3 Minimum Spanning Trees and the Travel Salesperson Problem

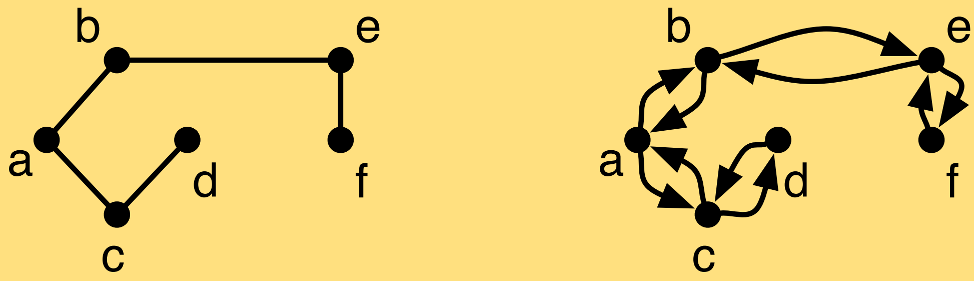
Bounding TSP with MST. There is an interesting connection between minimum spanning trees and the symmetric Traveling Salesperson Problem (TSP), an NP-hard problem. Recall that in TSP problem, we are given a set of n cities (vertices) and are interested in finding a tour that visits all the vertices exactly once and returns to the origin. For the symmetric case 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. It is however not a tree but a cycle. Since each vertex is visited once, however, dropping any edge would yield a spanning tree. Thus a solution to the TSP problem cannot have less total weight than that of a minimum spanning tree. In other words, the weight of a MST yields a lower bound on the solution to the symmetric TSP problem for graphs with non-negative edge weights.

Approximating TSP with MST. It turns out that 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 MST 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, $w(a,c) \leq w(a,b) + w(b,c)$. This restriction holds for most applications of the TSP problem and is referred to as the *metric TSP* problem. It also implies that edge weights are non-negative. We would now like a way to use the MST to generate a path to take as an approximate solution to the 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.

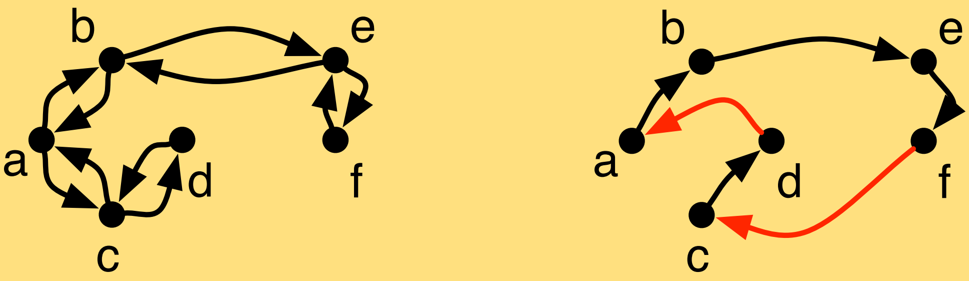
Given a minimum spanning tree T we can start at any vertex s and take a path based on the depth-first search on the tree from s. In particular whenever we visit a new vertex v from vertex u we traverse the edge from u to v and when we are done visiting everything reachable from v we then back up on this 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 18.23. 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`.



Now, recall that in the TSP problem it is assumed that there is an edge between every pair of vertices. Since 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 when 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 18.24. 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.



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. Since 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 18.25. 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.

18.4 Problems

18-1 Trees and edges
Prove that any tree with n vertices has $n - 1$ edges.

18-2 MST with star contraction
Work out the details of the algorithm for spanning trees using graph contraction with star partitions (as mentioned in Section 18.1) and prove that it produces a spanning tree.

18-3 Network design
Prove that the network with the minimum cost in Example 18.5 is indeed an MST of the graph.

18-4 MST with priority queues
Write out the pseudocode for a Priority Queue based implementation of Prim's algorithm that runs in $O(m \log n)$ work.

18-5 Unique MST's
Prove that a graph with distinct edge weights has a unique minimum spanning tree.

18-6 Acyclic vertex-joiners
Prove that the vertex-joiners selected in any round Borůvka's algorithm form a forest. Recall that we are assuming that no two edge weights are equal.

.

Part V

Parallelism in Practice

Chapter 19

Dynamic Programming

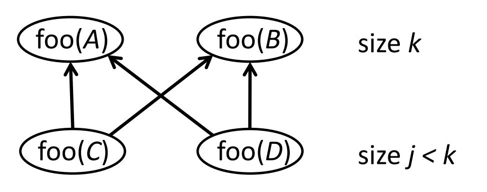
> “An interesting question is, ’Where did the name, dynamic programming, come from?’ The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. 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 16 is named after Richard Bellman and Lester Ford. In fact that algorithm can be viewed as a dynamic program. Although the quote is an interesting bit of history it does not tell us much about dynamic programming. But perhaps the quote will make you feel better about the fact that the term has little intuitive meaning.

In this book, as commonly used in computer science, we will use the term dynamic programming to mean an algorithmic technique in which (1) one constructs the solution of a larger problem instance by composing solutions to smaller instances, and (2) the solution to each smaller instance can be used in multiple larger instances.

Dynamic programming is another example of an inductive technique where an algorithm relies on putting together smaller parts to create a larger solution. The correctness then follows by induction on problem size. The beauty of such techniques is that the proof of correctness parallels the algorithmic structure. So far the inductive techniques we have covered are divide-and-conquer, the greedy method, and contraction. In the greedy method and contraction each instance makes use of only a single smaller instance. In the case of divide-and-conquer, as with dynamic programming, we made use of multiple smaller instances to solve a single larger instance. However in divide-and-conquer we have always assumed the solutions are solved independently, and therefore to calculate the work we added up the work from each recursive call. However, what if two instances of size k, for example, both need the solution to the same instance of size $j < k$?

Example 19.1. Two smaller instance of the problem (function call) `foo` being shared by two larger instances.

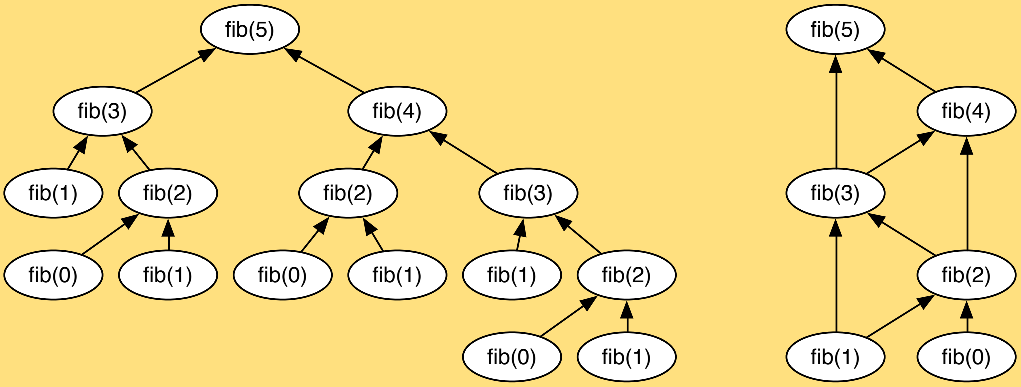


It is all about sharing. Although sharing the results in this simple example will make at most a factor of two difference in work, in general sharing the results of subproblems can make an exponential difference in the work performed. It turns out there are many practical problems where sharing results of subinstances is useful and can make a significant differences in the work used to solve a problem. We will go through several of these examples in this chapter.

Example 19.2. Consider the following algorithm for calculating the Fibonacci numbers.

```
fib($n$) =
  if ($n \leq 1$) then 1
  else fib($n-1$) + fib($n-2$)
```

This recursive algorithm takes exponential work in n as indicated by the recursion tree below on the left for 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.

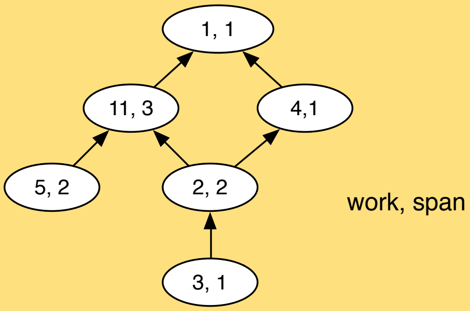
With divide-and-conquer the composition of a problem instance in terms of smaller instances is typically described as a tree, and in particular the so called recursion tree. With dynamic programming, to account for sharing, the composition can instead be viewed as a Directed Acyclic Graph (DAG). Each 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 dependences 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.

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.

Example 19.3. 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 $1 + 2 + 3 + 1 = 7$.

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 Fortunately, most dynamic programs have significant parallelism. Some, however, do not have much parallelism.

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, of course, is to think inductively—how can you solve an instance of a problem by composing the solutions to smaller instances? Once an inductive solution is formulated you can 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.

It turns out that most problems that can be tackled with dynamic programming solutions are optimization or decision problems. An *optimization problem* is one in which we are trying to find 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. Therefore when you see an optimization or enumeration problem you should think about possible dynamic programming solutions.

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 weighs on edges are defined in some problem specific way.

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.

19.1 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:

Problem 19.4. 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 19.5. `subsetSum`$(\{1,4,2,9\},8)$ returns `false` since there is no subset of 1, 4, 2, and 9 that adds up to 8. However, `subsetSum`$(\{1,4,2,9\},12)$ returns `true` since $1+2+9=12$.

In the general case when k is unconstrained, the 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.

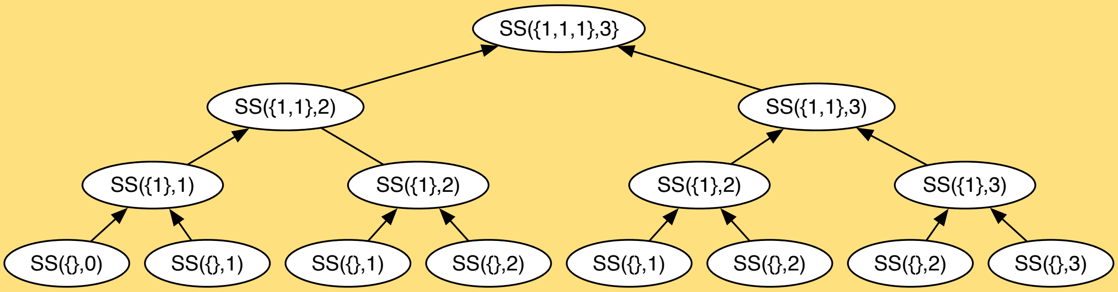
Since we do not know whether to add an element or not, we could try both cases, i.e. finding a sum with and without that element. This leads to a divide-and-conquer approach for solving SS(S,k) in which we pick one element a out of the set S (any will do), and then make two recursive calls, one with 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 19.6. [Recursive Subset Sum]

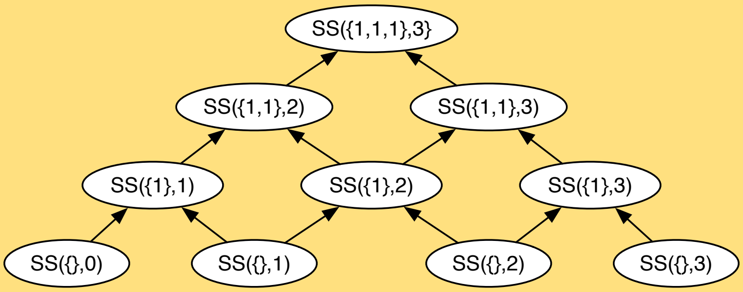
```
SS($S,k$) =
  case ($S$, $k$) of
    (_, 0) => true
  | (Nil, _) => false
  | (Cons($a,R$), _) =>
      if ($a > k$) then SS($R$, $k$)
      else (SS($R, k-a$) or SS($R,k$))
```

Lines 3 and 4 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

Example 19.7. Consider $\texttt{SS}(\{1,1,1\},3)$. This clearly should return true since $1+1+1=3$. The recursion tree is as follows.



There are many calls to $\texttt{SS}$ in this tree with the same arguments. In the bottom row, for example there are three calls each to $\texttt{SS}(\emptyset,1)$ and $\texttt{SS}(\emptyset,2)$. If we coalesce the common calls we get the following DAG:



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 $\texttt{SS}$, 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.

What is the work of this recursive algorithm? Well, it leads to a binary recursion tree that might be $n = |S|$ deep. This would imply something like 2^n work. This is not good. The key observation, however, is that there is a large amount of sharing of subproblems, as can be seen in Example 19.7. 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 $\texttt{SS}(S,k)$ 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 $\texttt{SS}$ (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:

$$W(\texttt{SS}(S,k)) = O(k|S|)$$

To calculate the span we need 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:

$$S(\texttt{SS}(S,k)) = O(|S|)$$

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 19.8. [Recursive Subset Sum (Indexed)]

```
SS($S$, $k$) = let
    % Determine SS($S[0,\ldots,i-1], j$)
    % $0 \le i \le |S|$ and $0 \le j \le k$, so at most $(|S|+1) \times (k+1)$ distinct calls
    SS'($i$, $j$) =
        case ($i$, $j$) of
            (_, 0) => true
          | (0, _) => false
          | ($i,k$) => if ($S[i-1] > j$) then SS'($i-1$, $j$)
                       else (SS'($i-1$, $j-S[i-1]$) or SS'($i-1$, $j$))
  % Calculates subset sum over all elements of $S$
  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 19.9. 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.

19.2 Minimum Edit Distance

The second problem we consider is a optimization problem, the minimum edit distance problem.

Problem 19.10. The minimum edit distance (MED) problem 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 19.11. Consider the sequence

$$S = \langle\, A, B, C, A, D, A \,\rangle$$

we could convert 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$(S, T) = 3$.

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[1]. 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 use 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 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.

[1] Alternatively it might store the new version, but use the differences to encode the old version.

Remark 19.12. 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.

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 19.13. 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, recur 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):

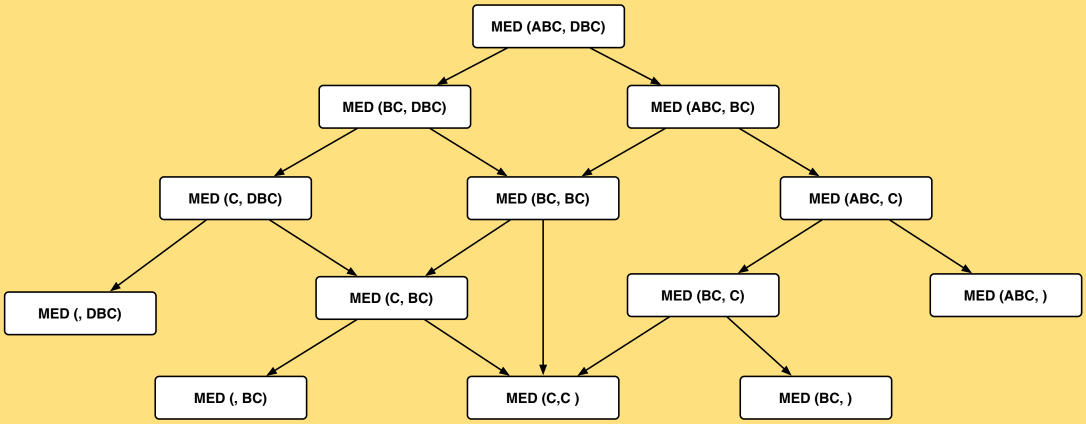
Algorithm 19.14. [Recursive solution to the MED problem]

```
MED($S$,$T$) =
  case ($S$, $T$) of
    (_, Nil) => $|S|$
  | (Nil, _) => $|T|$
  | (Cons($s$,$S'$), Cons($t$,$T'$)) =>
      if ($s = t$) then MED($S'$,$T'$)
      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 $|S|$ insertions. 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.

As with the recursive solution to the subset sum problem, 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. However, again, there are many calls to MED with the same arguments. We thus view the computation as a DAG in which each vertex corresponds to call to MED with distinct arguments. An edge is placed from u to v if the call v uses u.

Example 19.15. 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$(\langle A, B, C \rangle, \langle B, C \rangle)$

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

$$W(\texttt{MED}(S,T)) = O(|S||T|)$$

and

$$S(\texttt{MED}(S,T)) = O(|S|+|T|).$$

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 19.16. [Recursive indexed solution to the MED problem]

```
MED$(S,T)$ = let
   % Return MED($S[0,\ldots,i-1], T[0,\ldots,j-1]$)
   % $0 \le i \le |S|$ and $0 \le j \le |T|$, so at most $(|S|+1) \times (|T|+1)$ distinct arguments
   MED'$(i,j)$ =
      case $(i,j)$ of
         $(i,0)$ = $i$
      | $(0,j)$ = $j$
      | $(i,j)$ = if $(S[i-1] = T[i-1])$ then MED'$(i-1, j-1)$
                  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 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.

19.3 Top-Down Dynamic Programming

So far we have assumed some sort of magic identified the shared subproblems in our recursive codes and avoided recomputation. We are now ready to cover one of the techniques, the top-down approach to implementing dynamic-programming algorithms.

The top-down approach is based on running the recursive code as is, generating implicitly the recursion structure from the root of the DAG down to the leaves. Each time a solution to a smaller instance is found for the first time it generates a mapping from the input argument to its solution. This way when we come across the same argument a second time we can just look up the solution. This process is called *memoization*, and the table used to map the arguments to solutions is called a *memo table*.

The tricky part of memoization is checking for equality of arguments since the arguments might not be simple values such as integers. Indeed in our examples so far the arguments have all involved sequences. We could compare the whole sequence element by element, but that would be too expensive.

You might think that we can do it by comparing "pointers" to the values. But this does not work since the sequences can be created separately, and even though the values are equal, there could be two copies in different locations. Comparing pointers would say they are not equal and we would fail to recognize that we have already solved this instance. There is actually a very elegant solution that fixes this issue called *hash consing*. Hash consing guarantees that there is a unique copy of each value by always hashing the contents when allocating a memory location and checking if such a value already exists. This allows equality of values to be done in constant work. The approach only works in purely functional languages and needs to be implemented as part of the language runtime system (as part of memory allocation). Unfortunately no language does hash consing automatically, so we are left to our own devices.

The most common way to quickly test for equality of arguments is to use a simple type, such as an integer, as a *surrogate* to represent the input values. The property of these surrogates is that there needs to be a 1-to-1 correspondence between the surrogates and the argument values—therefore if the surrogates match, the arguments match. The user is responsible for guaranteeing this 1-to-1 correspondence.

Consider how we might use memoization in the dynamic program we described for minimum edit distance (MED). You have probably covered memoization before, but you most likely did it with side effects. Here we will do it in a purely functional way, which requires that we "thread" the table that maps arguments to results through the computation. Although this threading requires a few extra characters of code, it is safer for parallelism.

Recall that MED takes two sequences and on each recursive call, it uses suffixes of the two original sequences. To create integer surrogates we can simply use the length of each suffix. There is clearly a 1-to-1 mapping from these integers to the suffixes. MED can work from either end of the string so instead of working front to back and using suffix lengths, it can work back to front and use prefix lengths—we make this switch since it simplifies the indexing. This leads to the following variant of our MED code. In this code, we use prefixes instead of suffixes to simplify the indexing.

```
MED($S, T$) = let
  MED'($i, j$) =
    case ($i, j$) of
```

```
          $(i, 0) \;=\; i$
        | $(0, j) \;=\; j$
        | $(i, j) \;=\;$ if $(S[i-1] = T[j-1])$ then MED' $(i-1, j-1)$
                                   else $1 + \min($MED' $(i, j-1)$, MED' $(i-1, j))$
  in
    MED' $(|S|, |T|)$
  end
```

You should compare this with the purely recursive code for MED. Apart from the use of prefixes to simplify indexing, the only real difference is replacing S and T with i and j in the definition of MED$'$. The i and j are therefore the surrogates for S and T respectively. They represent the sequences $S\,\langle\, 0, \ldots, i-1\,\rangle$ and $T\,\langle\, 0, \ldots, j-1\,\rangle$ where S and T are the original input strings.

So far we have not added a memo table, but we can now efficiently store our solutions in a memo table based on the pair of indices (i, j). Each pair represents a unique input. In fact since the arguments range from 0 to the length of the sequence we can actually use a two dimensional array (or array of arrays) to store the solutions.

To implement the memoization we define a memoization function:

```
  memo $f$ $(M, a)$ =
    case find$(M, a)$ of
      SOME$(v)$ => $v$
    | 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. We can now write MED using memoization as shown in Figure 19.1.

Note that the memo table M is threaded throughout the computation. In particular every call to MED not only takes a memo table as an argument, it also returns a memo table as a result (possibly updated). Because of this passing, the code is purely functional. The problem with the top-down approach as described, however, is that it is inherently sequential. By threading the memo state we force a total ordering on all calls to MED. It is easy to create a version that uses side effects, as you did in 15-150 or as is typically done in imperative languages. In this case calls to MED can be made in parallel. However, then one has to be very careful since there can be race conditions (concurrent threads modifying the same cells). Furthermore if two concurrent threads make a call on MED with the same arguments, they can and will often both end up doing the same work. There are ways around this issue which are also fully safe—i.e., from the users point of view all calls look completely functional—but they are beyond the scope of this course.

```
MED$(S,T)$ = let
  MED'$(M,(i,j))$ =
      case $(i,j)$ of
        | $(M,(0,j))$ = $(M,j)$
        | $(M,(i,0))$ = $(M,i)$
        | $(M,(i,j))$ =
            if $(S[i-1] = T[j-1])$ then memo MED'$(M,(i-1,j-1))$
            else let
              $(M',v_1)$ = memo MED'$(M,(i,j-1))$
              $(M'',v_2)$ = memo MED'$(M',(i-1,j))$
            in $(M'',1+\min(v_1,v_2))$ end
in
  MED'$(\{\},(|S|,|T|))$
end
```

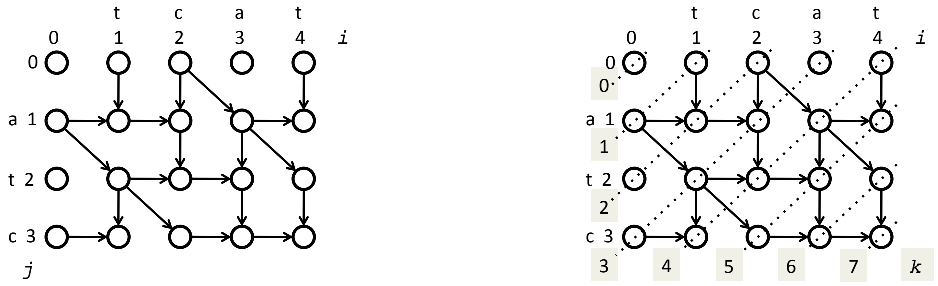
Figure 19.1: The memoized version of Minimum Edit Distance (MED).

19.4 Bottom-Up Dynamic Programming

The other technique for implementing dynamic-programming algorithms is the bottom-up technique. Instead of simulating the recursive structure, which starts at the root of the DAG, when using this technique, we start at the leaves of the DAG and fills in the results in some order that is consistent with the DAG–i.e. for all edges (u,v) it always calculates the value at a vertex u before working on v. Because of this careful scheduling, all values will be already calculated when they are needed.

The simplest way to implement bottom-up dynamic programming is to do some form of systematic traversal of a DAG. It is therefore useful to understand the structure of the DAG. For example, consider the structure of the DAG for minimum edit distance. In particular let's consider the two strings $S =$ `tcat` and $T =$ `atc`. We can draw the DAG as follows where all the edges go down and to the right:

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. We Consider MED$(4,3)$. The characters $S[4]$ and $T[3]$ are not equal so the recursive calls are to MED$(3,3)$ and MED$(4,2)$. This corresponds to the vertex to the left and the one above. Now if we consider MED$(4,2)$ the characters $S[4]$ and $T[2]$ are equal so the recursive call is to 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. This tells us quite a bit about the DAG. In particular it tells us that it is safe to process the vertices by first traversing the first row from left to right, and then the second row, and so on. It is also safe to traverse the first column from top to bottom and then the second



(a)

(b)

Figure 19.2: The DAG for MED on the strings "tcat" and "atc" (a) and processing it by diagonals (b).

```
MED$(S,T)$ = let
  MED'$(M,(i,j))$ =
    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
      $s = \max(0, k - |T|)$
      $e = \min(k, |S|)$
      $M' = M \cup \{(i, k-i) \mapsto MED(M,(i,k-i)) : i \in \{s, \ldots, e\}\}$
     in
      diagonals$(M', k+1)$
     end
in
  diagonals$(\{\}, 0)$
end
```

Figure 19.3: The dynamic program for MED based on the bottom-up approach using diagonals.

column and so on. In fact it is safe to process the diagonals in the / direction from top left moving to the bottom right. In this case each diagonal can be processed in parallel.

In general when applying MED(S,T) we can use an $|T| \times |S|$ array to store all the partial results. We can then fill the array either by row, column, or diagonal. Using diagonals can be coded as shown in Figure 19.3.

The code uses a table M to store the array entries. In practice an array might do better. Each round of `diagonals` processes one diagonal and updates the table M, starting at the leaves at top left. The figure below shows these diagonals indexed by k on the left side and at the bottom. We note that the index calculations are a bit tricky (hopefully we got them right). Notice that the size of the diagonals grows and then shrinks.

19.5 Optimal Binary Search Trees

We have talked about using BSTs for storing an ordered set or table. The cost of finding an key is proportional to the depth of the key in the tree. In a fully balanced BST of size n the average depth of each key is about $\log n$. Now suppose you have a dictionary where you know probability (or frequency) that each key will be accessed—perhaps the word "of" is accessed much more often than "epistemology". The goal is find a static BST with the lowest overall access cost. That is, make a BST so that the more likely keys are closer to the root and hence the average access cost is reduced. This line of reasoning leads to the following problem:

Definition 19.17. The *optimal binary search tree* (OBST) problem is given an ordered set of keys S and a probability function $p : S \to [0 : 1]$:

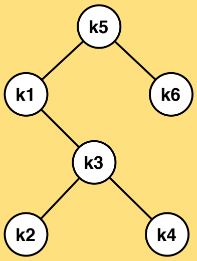
$$\min_{T \in \texttt{Trees}(S)} \left(\sum_{s \in S} d(s, T) \cdot p(s) \right)$$

where $\texttt{Trees}(S)$ 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 19.18. 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 19.19. Find another tree with equal cost.

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 19.20. Write a recurrence for the total number of distinct binary search trees with n keys.

Since we are considering binary search trees, one of the keys must be the root of the optimal tree. Suppose S_r is that root. An important observation is that both of its subtrees must be optimal, which is a common property of optimization problems: The optimal solution to a problem contains optimal solutions to subproblems. This *optimal substructure* property is often a clue that either a greedy or dynamic programming algorithm might apply.

Which key should be the root of the optimal tree? A greedy approach might be to pick the key k with highest probability and put it at the root and then recur on the two sets less and greater than k. You should convince yourself that this does not work. Since we cannot know in advance which key should be the root, let's try all of them, recursively finding their optimal subtrees, and then pick the best of the $|S|$ possibilities.

With this recursive approach, how should we define the subproblems? Let S be all the keys placed in sorted order. Now any subtree of a BST on S must contain 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 i and going to j (inclusive of both). Not surprisingly we will use the pair (i, j) to be the surrogate for $S_{i,j}$.

Now Let's consider how to calculate the cost give the solution to two subproblems. For subproblem $S_{i,j}$, assume we pick key S_r $(i \leq r \leq j)$ as a the root. We can now solve the OSBT problem on the prefix $S_{i,r-1}$ and suffix $S_{r+1,i}$. We therefore might consider adding these two solutions and the cost of the root ($p(S_r)$) to get the cost of this solution. This, however, is wrong. The problem is that by placing the solutions to the prefix and suffix as children of S_r we have increased the depth of each of their keys by 1. 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 therefore have:

$$\begin{aligned}
\texttt{Cost}(T) &= \sum_{s\in T} d(s,T)\cdot p(s)\\
&= p(S_r) + \sum_{s\in T_L}(d(s,T_L)+1)\cdot p(s) + \sum_{s\in T_R}(d(s,T_R)+1)\cdot p(s)\\
&= \sum_{s\in T} p(s) + \sum_{s\in T_L} d(s,T_L)\cdot p(s) + \sum_{s\in T_R} d(s,T_R)\cdot p(s)\\
&= \sum_{s\in T} p(s) + \texttt{Cost}(T_L) + \texttt{Cost}(T_R)
\end{aligned}$$

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 the following recursive definition:

Algorithm 19.21. [Recursive Optimal Binary Search Tree]

```
$\texttt{OBST}(S)$ =
  if $|S| = 0$ then $0$
  else $\sum_{s\in S} p(s) + \min_{i\in\langle 1\ldots|S|\rangle}\left(\texttt{OBST}(S_{1,i-1}) + \texttt{OBST}(S_{i+1,|S|})\right)$
```

Exercise 19.22. How would you return the optimal tree in addition to the cost of the tree?

As in the examples of subset sum and minimum edit distance, if we execute the recursive program directly `OBST` it will require exponential work. Again, however, we can take advantage of sharing among the calls to `OBST`. To bound the number of vertices in the corresponding DAG we need to count the number of possible arguments to `OBST`. Note that every argument is a contiguous subsequence from the original sequence S. A sequence of length n has only $n(n+1)/2$ contiguous subsequences since there are n possible ending positions and for the i^{th}

end position there are i possible starting positions ($\sum_{i=1}^{n} i = n(n+1)/2$). Therefore the number of possible arguments is at most $O(n^2)$. Furthermore the heaviest path of vertices in the DAG is at most $O(n)$ since recursion can at most go n levels (each level removes at least one key).

Unlike our previous examples, however, the cost of each vertex in the DAG (each recursive in our code not including the subcalls) is no longer constant. The subsequence computations $S_{i,j}$ can be done in $O(1)$ work each (think about how) but there are $O(|S|)$ of them. Similarly the sum of the $p(s)$ will take $O(|S|)$ work. To determine the span of a vertex we note that the $\min$ and sum can be done with a reduce in $O(\log|S|)$ span. Therefore the work of a vertex is $O(|S|) = O(n)$ and the span is $O(\log n)$. Now we simply multiply the number of vertices by the work of each to get the total work, and the heaviest path of vertices by the span of each vertex to get the span. This give $O(n^3)$ work and $O(n \log n)$ span.

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 ($A_1 \times A_2 \times \cdots A_n$) 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 $2 \times 10, 10 \times 2$, 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.

19.6 Coding optimal BST

As with the MED problem we first replace the sequences in the arguments with integers. In particular we describe any subsequence of the original sorted sequence of keys S to be put in the BST by its offset from the start (i, 1-based) and its length l. We then get the following recursive routine.

Algorithm 19.23. [Recursive Optimal Binary Search Tree (indexed)]

```
OBST($S$) = let
    % Determine OBST($S[i, \ldots, i + l - 1]$)
    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($1, |S|$)
end
```

This modified version can now more easily be used for either the top-down solution using memoization or the bottom-up solution. In the bottom-up solution we note that we can build a table with the columns corresponding to the i and the rows corresponding to the l. Each of them range from 1 to n ($n = |S|$). It would as follows:

```
 1 2 ... n
1         /
2       /
.     /
.   /
n /
```

The table is triangular since as l increases the number of subsequences of that length decreases. This table can be filled up row by row since every row only depends on elements in rows above it. Each row can be done in parallel.

19.7 Problems with Efficient Dynamic Programming Solutions

There are many problems with efficient dynamic programming solutions. Here we list just some of them to give a sense of what these problems are.

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 S1 and S2 be interleaved into S3
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 x1/x2/x3.../xn 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 20

Hashing

hash: transitive verb[1]

1. (a) to chop (as meat and potatoes) into small pieces
   (b) confuse, muddle
2. ...

This is the definition of hash from which the computer term was derived. The idea of hashing as originally conceived was to take values and to chop and mix them to the point that the original values are muddled. The term as used in computer science dates back to the early 1950's.

More formally the idea of hashing is to approximate a random function $h : \alpha \to \beta$ from a source (or universe) set U of type α to a destination set of type β. Most often the source set is significantly larger than the destination set, so the function not only chops and mixes but also reduces. In fact the source set might have infinite size, such as all character strings, while the destination set always has finite size. Also the source set might consist of complicated elements, such as the set of all directed graphs, while the destination are typically the integers in some fixed range. Hash functions are therefore many to one functions.

Using an actual randomly selected function from the set of all functions from α to β is typically not practical due to the number of such functions and hence the size (number of bits) needed to represent such a function. Therefore in practice one uses some pseudorandom function.

Exercise 20.1. How many hash functions are there that map from a source set of size n to the integers from 1 to m? How many bits does it take to represent them? What if the source set consists of character strings of length up to 20? Assume there are 100 possible characters.

[1] Merriam Websters

Why is it useful to have random or pseudo random functions that map from some large set to a smaller set? Generally such functions might be used to hide data, to reorder data in a random order, or to spread data smoothly over a range. Here we consider some applications of each.

1. We saw how hashing can be used in treaps. In particular we suggested using a hash function to hash the keys to generate the "random" priorities. Here what was important is that the ordering of the priorities is somehow random with respect to the keys. Our analysis assumed the priorities were truly random, but it can be shown that a limited form of randomness that arise out of relatively simple hash functions is sufficient.

2. In cryptography hash functions can be used to hide information. One such application is in digital signatures where a so-called secure hash is used to describe a large document with a small number of bits.

3. A one-way hash function is used to hide a string, for example 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. 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.

4. String commitment protocols use hash functions to hide to 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 mean time 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 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.

There is a deep and interesting theory of hash functions. Depending on the application, the needs of the hash function are very different. We will not cover the theory here but you will likely see it in more advanced algorithms classes.

For hash table applications a hash function should have the following properties:

- It should distribute the keys evenly so that there are not many collisions.
- It should be fast to compute.

Here we consider some simple ones. For hashing integers we can use

$$h(x) = (ax + b) \bmod p$$

where $a \in [1, \ldots, p-1]$, $b \in [0, \ldots, p-1]$, and p is a prime. This is called a linear congruential hash function has some nice properties that you will likely learn about in 15-451.

For strings we can simply use a polynomial

$$h(S) = \left(\sum_{i=1}^{|S|} s_i a^i \right) \bmod p$$

Sequentially, Horner's method avoids computing a^i explicitly. In parallel, simply use scan with multiplication. This hash function tends to mix bits from earlier characters with bits in later characters.

In our analysis we will assume that we have hash functions with the following idealized property called *simple uniform hashing*: The hash function uniformly distributes the n keys over the range $[0, \ldots, m-1]$ and the hash value for any key is independent of the hash value for any other key.

20.1 Hash Tables

Hash tables are used when an application needs to maintain a dynamic set where the main operations are `insert`, `find` and `delete`. Hash tables can implement the abstract data types `Set` and `Table`. Unlike binary search trees, which require the universe of keys has a total ordering, hash tables do not. A total ordering enables the additional ordering operations provided by the Ordered Set abstract data type.

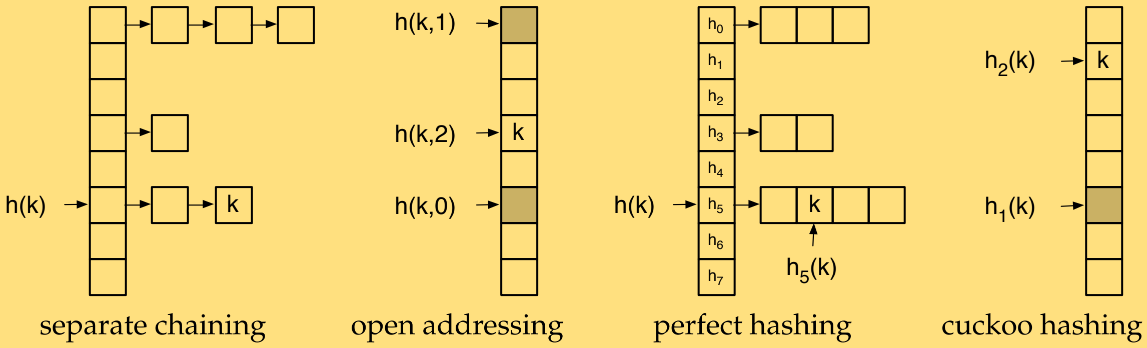
The main issue with hash table is collisions, where two keys hash to the same location. Is it possible to avoid collisions? Not if we don't know the set of keys in advance. Since the size of the table T is much smaller than the universe of keys U, $|T| \ll |U|$, there must exist at least two keys that map to the same index, by the *Pigeonhole Principle*: If you put more than m items into m bins, then at least one bin contains more than one item. For a particular hash function, the subset of keys $K \subset U$ that we want to hash may or may not cause a collision even when the number of keys is much smaller than the size of the table. Therefore, for general purpose dynamic hash tables we have to assume that collisions will occur.

How likely is there at least one collision? This is the same question as the birthday paradox: When there a n or more people in a room, what is the chance that two people have the same birthday? It turns out that for a table of size 365 you need only 23 keys for a 50% chance of a collision, and as little as 60 keys for a 99% chance. More interestingly, when hashing to m locations, you can expect a collision after only $\sqrt{\frac{1}{2}\pi\, m}$ insertions, and can expect every location in the table to have an element after $\Theta(m \log m)$ insertions. The former is related to the *birthday paradox*, whereas the latter is related to the *coupon collector* problem.

There are several well-studied collision resolution strategies:

- **Separate chaining**: For each bin store a linked list of all keys that hash to that bin.
- **Open addressing**: Place all keys directly into bins (one key per bin), but if multiple keys hash to the same bin, then all but one of them in "nearby" bins.
- **Perfect hashing**: When the keys are known in advance, it is possible to construct hash functions that avoid collisions entirely, by using a two level hash function.
- **Multiple-choice hashing and Cuckoo hashing**: A special case of open addressing in which every key is placed in only one of two locations $h_1(k)$ or $h_2(k)$.

Example 20.2. Different types of hash tables. The grey indicates the location is already full with another key.



separate chaining open addressing perfect hashing cuckoo hashing

We will consider the first two in this lecture.

In our discussion, we will assume we have a set of n keys K that we want to store and a hash function $h : \texttt{key} \rightarrow [0, \ldots, m-1]$ for some m.

20.2 Separate Chaining

In 15-122 you studied hash tables using separate chaining. As you may recall, the idea is to maintain an array of linked lists. All keys that hash to the same location (bin) in the sequence are maintained in a linked list. To find a key a key k, go to the location $h(k)$, and search the list belonging to that location for that key. To insert a key, first try to find the key. If the key is in the list, then possibly replace it with the new value (depending on the semantics of insert), otherwise add the key to either the start or the end of the list.

Exercise 20.3. Write pseudocode for inserting to, searching from and deleting from a separate chaining hash table.

The costs of these operations is related to the average length of a chain, which is n/m when there are n keys in a table with m chains. We call $\lambda = n/m$ the *load factor* of the table.

When searching for a key the key might or might not be in the table. We refer to the two cases as a *successful search* and an *unsuccessful search*, respectively. Assuming that we can compute $h(k)$ in $O(1)$ work, we then have:

Claim 20.4. For simple uniform hashing, an *unsuccessful search* takes expected $\Theta(1 + \lambda)$ work.

Proof. The average length of a list is λ. If we search for a key that is not in the table, we need to search the whole list to determine the key is not in the table. Including the cost of computing $h(k)$, the total work is $\Theta(1 + \lambda)$. □

Claim 20.5. For simple uniform hashing, a *successful search* takes expected $\Theta(1 + \lambda)$ work.

Proof. To simplify the analysis, we assume that keys are added at the end of the chain[a]. The cost of a successful search is the same as an unsuccessful search at the time the key was added to the table. That is, when we first insert a key, the cost to insert it is the same as the cost of an unsuccessful search. Suppose this cost is T_k. Subsequent searches for this key is also T_k. When the table has i keys, the cost of inserting a new key is expected to be $(1 + i/m)$. Averaging over all keys, the cost of a successful search is

$$\frac{1}{n}\sum_{i=0}^{n-1}(1 + i/m) = \frac{1}{n}(n + n(n-1)/(2m)) = 1 + (n-1)/(2m) \leq 1 + \lambda/2 = \Theta(1 + \lambda)$$

□

[a] The average successful search time is the same whether new keys are added to the front of the end of the chain.

That is, for successful searches we examine half the list on average and unsuccessful searches the full list. If $n = O(m)$ then with simple uniform hashing, all operations have expected $O(1)$ work and span. Even more importantly, some chains can be long, $O(n)$ in the worst case, but it is expectation they will have length λ. The advantage of separate chaining is that it is not particularly sensitive to the size of the table. If the number of keys is more than anticipated, the cost of search becomes only somewhat worse. If the number is less, then only a small amount of space in the table is wasted and the cost of search is faster.

20.3 Open Address Hash Tables

The next technique does not need any linked lists but instead stores every key directly in the locations of an array, which we will refer to as cells. Open address hashing using so-called linear probing has an important practical advantage over separate chaining: it causes fewer cache misses since typically all locations that are checked are on the same cache line.

The basic idea of open addressing is to maintain an array that is some constant factor larger than the number of keys and to store all keys directly in this array. Every cell in the array is either empty or contains a key.

To decide to which cells to assign a key, open addressing uses an ordered sequence of locations in which the key can be stored. In particular let's assume we have a function $h(k, i)$ that returns the i^{th} location for key k. We refer to the sequence

$$\langle\, h(k, 0), h(k, 1), h(k, 2), \ldots \,\rangle$$

as the *probe* sequence. We will get back to how the probe sequence might be assigned, but let's first go through how these sequences are used. When inserting a key the basic idea is to try each of the locations in order until we find a cell that is empty, and then insert the key at that location. Sequentially, insert would look like:

Algorithm 20.6. [Insertion into an Open Address Hash Table]

```
function insert($T, k$) = let
  function insert'($T, k, i$) =
    case $T[h(k,i)]$ of
      None => update($T, h(k,i), k$)
    | Some($k'$) =>
        if $(k = k')$ then $T$
        else insert'($T, k, i+1$)
in insert'($T, k, 0$) end
```

Example 20.7. Suppose the hash table has the following keys:

	0	1	2	3	4	5	6	7
T =		B			E	A		F

Now if for a key D we had the probe sequence $\langle 1, 5, 3, \cdots \rangle$, then we would find location 1 and 5 full (with B and E) and place D in location 3 giving:

	0	1	2	3	4	5	6	7
T =		B		D	E	A		F

Note that for the update operation to be constant work and span, T must be a single threaded array. Also, the insertion algorithm will loop forever if all locations are full. Such an infinite loop can be prevented by ensuring that $h(k, i)$ tries every location as i is incremented, and checking when the table is full. Also, as described, the algorithm will insert the same key multiple times over. This problem is easily corrected by checking if the key is in the table and if so returning immediately.

To search we have the following code:

Algorithm 20.8. [Find in an Open Address Hash Table]

```
function find($T, k$) = let
  function find'($T, k, i$) =
    case $T[h(k, i)]$ of
      None => false
    | Some($k'$) =>
        if (k = k') then true
        else find'($T, k, i + 1$)
in find'($T, k, 0$) end
```

Example 20.9. In the table

	0	1	2	3	4	5	6	7
T =		B		D	E	A		F

if key E has the probe sequence $\langle 7, 4, 2, \cdots \rangle$, find($T$, E) would first search location 7, which is full, and then location 4 where it finds E.

Let's say we deleted A from the table above, and then searched for D. Will find locate it? No, it will stop looking once it finds the empty cell where A was. One solution might be to rehash everything after a delete. But that would be an extremely expensive operation for every delete. An alternative is to use what is called a *lazy delete*. Instead of deleting the key, simply replace the key with a special Hold value. That is, introduce an entry data type:

```
datatype $\alpha$ entry = Empty | Hold | Full of $\alpha$
```

For find, simply skip over a Hold entry and move to the next probe. insert(v) can replace the first Hold entry with Full(v). But if insert needs to check for duplicate keys, it first needs to search for the key. If it finds the key it overwrites it with the new value. Otherwise it continues until it finds an empty cell, at which point it can replace the first Hold in the probe sequence.

The main concern with lazy deletes is that they effectively increase the load factor, increasing the cost of the hash table operations. If the load factor becomes large and performance degrades, the solution is to rehash everything to a new larger table. The table should be a constant fraction larger each time the table grows so as to maintain amortized constant costs.

Now let's consider some possible probe sequences we can use. Ideally, we would like a key to use any of the possible $m!$ probe sequences with equal probability. This ideal is called *uniform hashing*. But uniform hashing is not practical. Common probe sequences, which we will consider next, are

- linear probing
- quadratic probing
- double hashing

20.3.1 Linear Probing

In linear probing, to insert a key k, it first checks $h(k)$ and then checks each following location in succession, wrapping around as necessary, until it finds an empty location. That is, the i^{th} probe is

$$h(k, i) = (h(k) + i) \mod m.$$

Each position determines a single probe sequence, so there are only m possible probe sequences.

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 be lengthen further.

What is the impact of clustering for an unsuccessful search? Let's consider two extreme examples when the table is half full, $\lambda = 1/2$ (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

$$1 + [n + (n - 1) + (n - 2) + ... + 1]/m = 1 + n(n + 1)/2m \approx n/4$$

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.

We won't attempt to analyze the cost of successful and unsuccessful searches, as considering cluster formation during linear probing is quite difficult. We make the following claim:

Claim 20.10. When using linear probing in a hash table of size m that contains $n = \lambda m$ keys, the average number of probes needed for an unsuccessful search or an insert is

$$\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

and for a successful search is

$$\frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right).$$

As you can see from the following table, which shows the expected number of probes under uniform hashing, the performance of linear probing degrades significantly when the load factor increases:

λ	1/4	1/2	2/3	3/4	9/10
successful	1.2	1.5	2.0	3.0	5.5
unsuccessful	1.4	2.5	5.0	8.5	50.5

Linear probing is quite competitive, though, 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.

20.3.2 Quadratic Probing

Quadratic probe sequences cause probes to move away from clusters, by making increasing larger jumps. The i^{th} probe is

$$h(k,i) = (h(k) + i^2) \mod m.$$

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 make the following guarantee.

Claim 20.11. 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. (by contradiction) Consider two probe locations $h(k)+i^2$ and $h(k)+j^2, 0 \le i, j < \lceil m/2 \rceil$. Suppose the locations are the same but $i \ne j$. Then

$$\begin{aligned} h(k) + i^2 &\equiv (h(k) + j^2) \mod m \\ i^2 &\equiv j^2 \mod m \\ i^2 - j^2 &\equiv 0 \mod m \\ (i-j)(i+j) &\equiv 0 \mod m \end{aligned}$$

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. Contradiction.

Thus the first $\lceil m/2 \rceil$ probes are distinct and guaranteed to find an empty location. □

Computing the next probe is only slightly more expensive than linear probing as it can be computed without using multiplication:

$$\begin{aligned} h_i - h_{i-1} &\equiv (i^2 - (i-1)^2) \mod m \\ h_i &\equiv (h_{i-1} + 2i - 1) \mod m \end{aligned}$$

Unfortunately, requiring that the table remains less than half full makes quadratic probing space inefficient.

20.3.3 Double Hashing

Double hashing uses two hash functions, 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 i^{th} probe is

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod m.$$

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 $h_2(k)$, every cell is probed if $h_2(k)$ is relatively prime to m. Two possible ways to ensure $h_2(k)$ is relatively prime to m are, either make $m = 2^k$ and design $h_2(k)$ so it is always odd, or make m prime and ensure $h_2(k) < m$. Of course, $h_2(k)$ cannot equal zero.

Double hashing behaves quite closely to uniform hashing for careful choices of h_1 and h_2. Under uniform hashing the average number of probes for an unsuccessful search or an insert is at most

$$1 + \lambda + \lambda^2 + \ldots = \left(\frac{1}{1-\lambda}\right)$$

and for a successful search is at most

$$\frac{1}{\lambda}\left(1 + \ln\left(\frac{1}{1-\lambda}\right)\right).$$

The former bound is because the probability of needing more than i probes is at most λ^i. A search always needs one probe, and with probability λ needs a second probe, and with probability λ^2 needs a third probe, and so on. The bound for a successful search for a key k follows the same probe sequences as when it was first inserted. So if k was the $(j+1)^{th}$ key inserted the cost for inserting it is at most $1/(1-j/m)$. Therefore the average cost of a successful search is at most

$$\begin{aligned}\frac{1}{n}\sum_{j=0}^{n-1}\frac{1}{1-j/m} = \frac{m}{n}\sum_{j=0}^{n-1}\frac{1}{m-j} \\ &= \tfrac{1}{\lambda}\left(\sum_{j=0}^{m}\frac{1}{j} + \sum_{j=0}^{m-n}\frac{1}{j}\right) \\ &= \tfrac{1}{\lambda}(H_m - H_{m-n}) \\ &\leq \tfrac{1}{\lambda}(\ln m + 1 - \ln(m-n)) \\ &= \tfrac{1}{\lambda}\left(1 + \ln\left(\tfrac{1}{1-\lambda}\right)\right)\end{aligned}$$

The table below shows the expected number of probes under the assumption of uniform hashing and is the best one can expect by open addressing.

λ	1/4	1/2	2/3	3/4	9/10
successful	1.2	1.4	1.6	1.8	2.6
unsuccessful	1.3	2.0	3.0	4.0	10.0

Comparing these numbers with the numbers in the table for linear probing, the linear probing numbers are remarkable close when the load factor is 50% or below.

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.

20.4 Hash Table Summary

Hashing is a classic example of a space-time tradeoff: increase the space so table operations are faster; decrease the space but table operations are slower.

Separate chaining is simple to implement and is less sensitive to the quality of the hash function or load factors, so it is often the choice when it is unknown how many and how frequently keys may be inserted or deleted from the hash table. On the other hand open addressing can be more space efficient as there are no linked lists. Linear probing has the advantage that it has small constants and works well with caches since the locations checked 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%. Although double hashing reduces clustering, so high load factors are possible, finding suitable pairs of hash functions is somewhat more difficult and increases the cost of a probe.

20.5 Parallel Hashing

In the parallel context, instead of inserting, finding or deleting one key at a time, each operation takes a set of keys. Since a hash function distributes keys across slots in the table, we can expect many keys will be hashed to different locations. The idea is to use open addressing in multiple rounds. For `insert`, each round attempts to write the keys into the table at their appropriate hash position. For any key that cannot be written because another key is already there, the key continues for another round using its next probe location. Rounds repeat until all keys are written to the table.

In order to prevent writing to a position already occupied in the table, we introduce a variant of the `inject` function. The function

$$\texttt{injectCond}(I, S)\ :\ (\texttt{int}\ \times\ \alpha)\ \texttt{seq}\ \times\ (\alpha\ \texttt{option})\ \texttt{seq}\ \rightarrow\ (\alpha\ \texttt{option})\ \texttt{seq}$$

takes a sequence of index-value pairs $\langle\,(i_1, v_1), \ldots, (i_n, v_n)\,\rangle$ and a target sequence S and conditionally writes each value v_j into location i_j of S. In particular it writes the value only if the location is set to None. If there are two or more values with the same index (a conflict) then it conditionally writes the value only for the *first* occurrence of the index (recall that `inject` uses the *last* occurrence of an index). Resolving conflicts in `injectCond` can be implemented using a parallel primitive called write-with-min[2].

[2] For more information, see the paper *Reducing Contention Through Priority Updates* by Julian Shun, Guy Blelloch, Jeremy Fineman and Phillip Gibbons: http://www.cs.cmu.edu/~jshun/contention.pdf.

Algorithm 20.12. [Parallel Insertion with Open Addressing]

```
function insert($T, K$) = let
  val $i$ = 0
  while $(|K| > 0)$ with $(T, K, i)$
    val $I$ = $\langle\, (h(k,i), k) : k \in K \,\rangle$
    val $T$ = injectCond($I, T$)
    val $K$ = $\langle\, k \in K \mid T[h(k,i)] \neq$ Some($k$) $\rangle$
    val $i$ = $i + 1$
in $T$ end
```

For round i, insert uses each key's i^{th} probe in its probe sequence and attempts to write the key to the table. To see whether it successfully wrote a key to the table, it reads the value written to the table and checks if it is the same as the key. In this way it can filter out all keys that it successfully wrote to the table. It repeats the process on the keys it didn't manage to write, using the keys' $(i + 1)$ probes.

Example 20.13. Consider a table with the following entries before round i:

	0	1	2	3	4	5	6	7
T =		A		B			D	C

If $K = \langle\, \text{E}, \text{F} \,\rangle$, $h(\text{E}, 0) = 1$, and $h(\text{F}, 0) = 2$, then on the first round $I = \langle\, (1, \text{E}), (2, \text{F}) \,\rangle$ and the injectCond would fail to write E to index 1 but would succeed in writing F to index 2 resulting in the following table:

	0	1	2	3	4	5	6	7
T =		A	F	B			D	C

On the next iteration we have $K = \langle\, \text{E} \,\rangle$ and $i = i + 1 = 1$.

Note that if T is implemented using a single threaded array, then parallel insert basically does the same work as the sequential version adding the keys one by one. The only difference is that the parallel version may add keys to the table in a different order than the sequential.

Example 20.14. With linear probing, the parallel version adds F first using 1 probe and then adds E at index 4 using 4 probes:

T_P =

0	1	2	3	4	5	6	7
	A	F	B	E		D	C

Whereas, the sequential version might add E first using 2 probes, and then F using 3 probes:

T_S =

0	1	2	3	4	5	6	7
	A	E	B	F		D	C

Both versions make 5 probes in the table.

Since we showed that, with suitable hash functions and load factors, the expected cost of insert is $O(1)$, the expected work for the parallel version is $O(|K|)$. In addition, in each round, the expected size of K decreases by a constant fraction, so the span is $O(\log |K|)$.

Exercise 20.15. Assuming uniform hashing and a hash table of size $2|K|$, prove that the expected size of K decreases by a constant fraction in each call to `insert'`.

Exercise 20.16. Describe how to do hash table searches in parallel and deletions in parallel.

20.6 Comparison to Tree Tables

- Searches are faster in expectation
- Insertions are faster in expectation
- Map, reduce, and filter remain linear work
- Union/merge can be slower.

In contrast to a tree table, a hash table cannot support range queries efficiently since it does not maintain any order on the keys.

.

Chapter 21

Priority Queues

We have already discussed and used priority queues in a few places in this class. We used them as an example of an abstract data type. We also used them in the priority-first graph search to implement Dijkstra's algorithm, and Prim's algorithm for minimum spanning trees.

Abstract Data Type 21.1. [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 (partial list):

$$\begin{array}{lcl}
\texttt{empty} & : & \mathbb{T} \\
\llbracket\texttt{empty}\rrbracket & = & \{\} \\
\texttt{singleton} & : & \mathbb{S} \to \mathbb{T} \\
\llbracket\texttt{singleton}e\rrbracket & = & \{e\} \\
\texttt{findMin}Q & : & \mathbb{T} \to \mathbb{S} \\
\llbracket\texttt{findMin}Q\rrbracket & = & \min\llbracket Q\rrbracket \\
\texttt{insert} & : & \mathbb{S} \to \mathbb{T} \\
\llbracket\texttt{insert}(Q,e)\rrbracket & = & \llbracket Q\rrbracket \cup \{e\} \\
\texttt{deleteMin} & : & \mathbb{T} \to \mathbb{T} \times (\mathbb{S} \cup \{\bot\}) \\
\llbracket\texttt{deleteMin}Q\rrbracket & = & \begin{cases} (\llbracket Q\rrbracket, \bot) & \llbracket Q\rrbracket = \{\} \\ (\llbracket Q\rrbracket \setminus \{\min\llbracket Q\rrbracket\}, \min\llbracket Q\rrbracket) & \textit{otherwise} \end{cases} \\
\texttt{meld} & : & \mathbb{T} \times \mathbb{T} \to \mathbb{T} \\
\llbracket\texttt{meld}\rrbracket(Q_1, Q_2) & = & \llbracket Q_1\rrbracket \cup \llbracket Q_2\rrbracket
\end{array}$$

As you might have seen in other classes, a priority queue can also be used to implement an $O(n \log n)$ work (time) version of selection sort, often referred to as heapsort. The sort can be implemented as:

Algorithm 21.2. [Heapsort]

```
sort $S$ =
let
  pq = Sequence.iter PQ.insert PQ.empty $S$
  hsort pq =
    case PQ.deleteMin pq
    | NONE => $\langle\,\rangle$
    | SOME($v$,pq′) => Seq.append $\langle v \rangle$ (hsort pq′)
in
  hsort pq
end
```

Priority queues also have applications elsewhere, including

- Huffman Codes
- Clustering algorithms
- Event simulation
- Kinetic algorithms

Priority queues can be implemented by using a variety of data structures. 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. Another possibility would be to use balanced binary search trees (e.g., treaps), which would support both operations in $O(\lg n)$ work and span. As we shall see, a binary heap data structure, implemented with (mutable) arrays, can support both operations in $O(\log n)$ span. Binary heaps have the slight advantage over binary search trees of supporting an additional operations, `findMin`, which returns the element with the minimum priority in $O(1)$ work and span.

21.1 Binary Heaps

A *min-heap* is a rooted tree such that the key stored at every node is less 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. A *search-tree* is a rooted tree such that the key sorted at every node is greater than (or equal to) all the keys in its left subtree and less than all the keys in its right subtree. Heaps maintain only a partial ordering, whereas search trees maintain a total ordering.

A binary heap is a particular implementation 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 an array, and the index of the a parent or child node is a simple computation. Recall that operations on a binary heap first restore the shape property, and then the heap property.

To build a priority queue, we can insert one element at a time into the priority queue as we did in heap sort above. With both balanced binary search trees and binary heaps, the cost is $O(n \log n)$. Can we do better? For heaps, yes, build the heap recursively. If the left and right children are already heaps, we can just "sift down" the element at the root:

Algorithm 21.3.

```
fromSeq $S$ =
  let
    heapify $(S, i)$ =
      if $(i \geq |S|/2)$ then $S$
      else
        let
          $S'$ = heapify $(S,\ 2 \times i + 1)$
          $S''$ = heapify $(S',\ 2 \times i + 2)$
        in siftDown $(S'',\ i)$ end
  in heapify $(S, 0)$ end
```

With ST-sequences, `siftDown` does $O(\log n)$ work on a subtree of size n. Therefore, `fromSeq` has work

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

We can build a binary heap in parallel with ST-sequences. If you consider S as a complete binary tree, the leaves are already heaps. The next level up of this tree, the roots of the subtrees violate the heap property and need to be sifted down. Since the two children of each root are heaps, the result of sift down is a heap. That is, on each level of the complete tree, fix the heaps at that level by sifting down the elements at that level. The code below, for simplicity, assumes $|S| = 2^k - 1$ for some k:

Algorithm 21.4.

```
fromSeq $S$ =
let
  heapify $(S, d)$ =
    let $S'$ = siftDown $(S,\ [2^d - 1, \ldots, 2^{d+1} - 2],\ d)$
    in if $(d = 0)$ then $S'$
       else heapify $(S',\ d - 1)$
    end
in heapify $(S,\ \log_2 n - 1)$ end
```

There is a subtly with this parallel `siftDown`. It too needs to work one layer of the binary tree at a time. That is, it takes a sequence of indices corresponding to elements at level d and determines if the those elements need to swap with elements at level $d + 1$. It does the swaps using `inject`. Then it calls `siftDown` recursively using the indices to where the elements at d moved to in level $d + 1$, if indeed they moved down. When it reaches the leaves it returns the updated ST-sequence.

This parallel version does the same work as the sequential version. But now span is $O(\log n)$ at each of the $O(\log n)$ layers of the tree. More specifically, the span of the level d heapify operation satisfies this recurrence:

$$S(d) = S(d - 1) + O((\log n) - d)$$

Since d starts at $\log n$, this sums to $O(\log^2 n)$.

In summary, the table below shows that a binary heap is an improvement over more general purpose structures used for implementing priority queues. The shape property of a binary heap, though, limits its ability to implement other useful priority queue operations efficiently. Next, we will a more general priority queue, meldable ones.

Implementation	**`findMin`**	**`deleteMin`**	**`insert`**	**`fromSeq`**
sorted linked list	$O(1)$	$O(1)$	$O(n)$	$O(n \log n)$
unsorted linked list	$O(n)$	$O(n)$	$O(1)$	$O(n)$
balanced search tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n \log n)$
binary heap	$O(1)$	$O(\log n)$	$O(\log n)$	$O(n)$

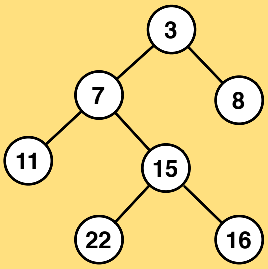
21.2 Meldable Priority Queues

Recall that, much earlier in the course, we introduced a meldable priority queue as an example of an abstract data type. It includes the `meld` operation, which is analogous to `merge` for binary search trees; It takes two meldable priority queues and returns a meldable priority queue that contains all the elements of the two input queues.

Today we will discuss one implementation of a meldable priority queue, which has the same work and span costs as binary heaps, but also has an efficient operation `meld`. This operation has work and span of $O(\log n + \log m)$, where n and m are the sizes of the two priority queues to be merged.

The structure we will consider is 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. The goal is to make the `meld` fast, and in particular run in $O(\log n)$ work. First, let's consider how we could use `meld` and what might be an implementation of `meld` on a heap.

Example 21.5. An example min-heap illustrated.



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's consider how to implement the three operations `deleteMin`, `insert`, and `fromSeq` on a heap. Like `join` for treaps, 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`:

```
(* Insert all keys into priority queue *)
val pq = Seq.reduce Q.meld Q.empty (Seq.map Q.singleton S)
```

In this way, we can insert multiple keys into a heap in parallel: Simply build a heap as above and then meld the two heaps. There is no real way, however, to remove keys in parallel unless we use something more powerful than a heap.

The only operation we need to care about, therefore, is the `meld` operation. Suppose that we are given twe heaps to meld. By inspecting the roots of the heaps, we can determine that the smaller one will be the root of the new heaps. 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- and the right-subtrees of the chosen root, and the other tree. Let's 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.

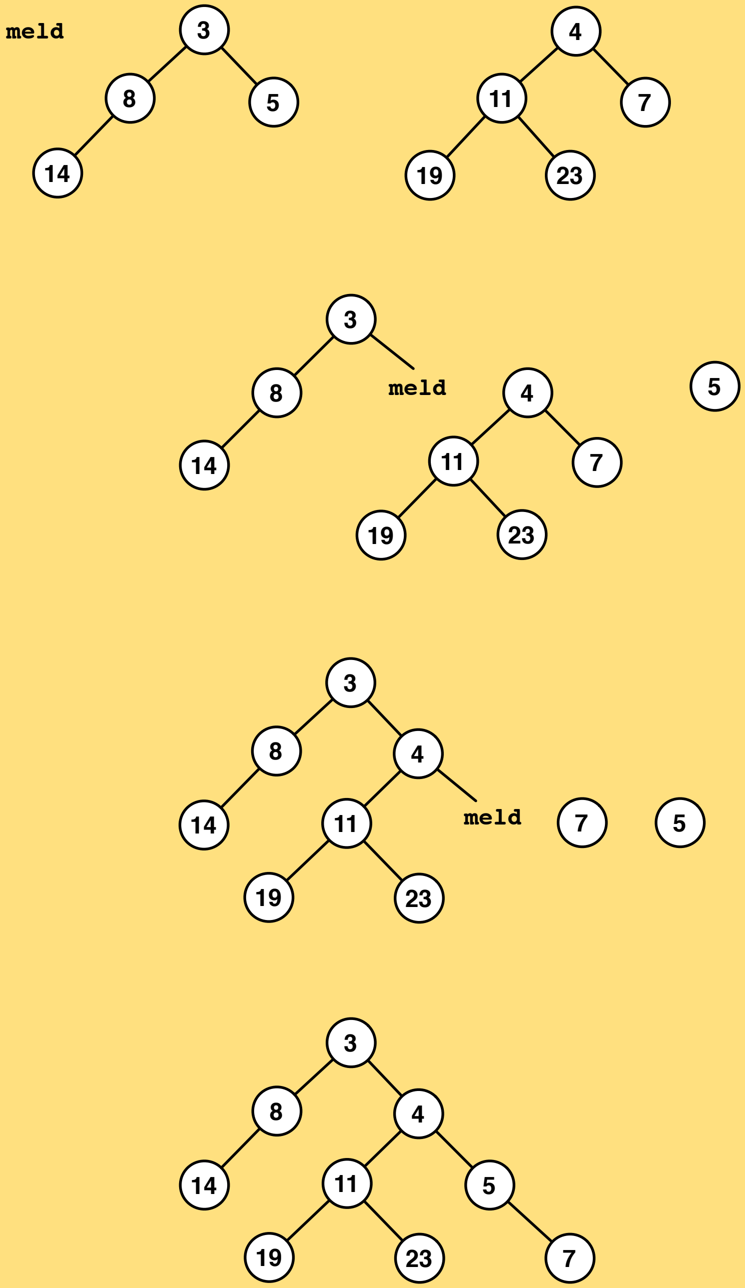
This algorithm leads to the code shown below (Algorithm 21.6)

Algorithm 21.6. ˜

```
datatype PQ = Leaf | Node of (key × PQ × PA)

meld $(A, B)$ =
   case $(A, B)$
   |  (_, Leaf) => $A$
   | (Leaf, _) => $B$)
   | (Node ($k_a$, $L_a$, $R_a$), Node ($k_b$, $L_b$, $R_b$)) =>
         case Key.compare ($k_a$, $k_b$) of
            Less => Node ($k_a$, $L_a$, meld($R_a$, $B$))
          | _ => Node ($k_b$, $L_b$, meld($A$, $R_b$))
```

The 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 21.7. An example `meld` operations on two heaps illustrated.



21.3 Leftist Heaps

It turns out there is a relatively easy fix to this imbalance problem. The idea is to keep the trees so that the trees are always deeper on the left than the right. In particular, we define the *rank* of a node x as

$$\mathsf{rank}(x) = \#\text{ of nodes on the right spine of the subtree rooted at } x,$$

and more formally:

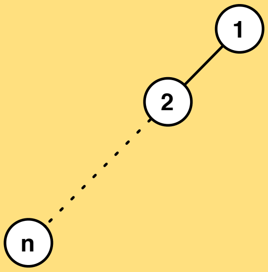
$$\begin{aligned}\mathsf{rank}(\mathsf{leaf}) &= 0\\ \mathsf{rank}(\mathsf{node}(\_,\_,R) &= 1 + \mathsf{rank}(R)\end{aligned}$$

Now we require that all nodes of a leftist heap have the "leftist property". That is, if $L(x)$ and $R(x)$ are the left and child children of x, then we have:

Leftist Property: For any node x in a leftist heap, $\mathsf{rank}(L(x)) \geq \mathsf{rank}(R(x))$

This is why the tree is called leftist: for each node in the heap, the rank of the left child must be at least the rank of the right child. Note that this definition allows an extremely unbalanced tree such as the one shown below.

Example 21.8. An example leftist heap.



This is OK since we only ever traverse the right spine of a tree, which in this case has length 1.

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. In this way, all update operations we care about can be supported efficiently. We'll make this idea precise in the following lemma which will prove later; we'll see how we can take advantage of this fact to support fast meld operations.

Lemma 21.9. In a leftist heap with n entries, the rank of the root node is at most $\log_2(n+1)$.

In words, this lemma says *leftist heaps have a short right spine,* about $\log n$ in length. To get good efficiency, we should take advantage of it. Notice that unlike the binary search tree property, the heap property gives us a lot of freedom in working with left and right child of a node (in particular, they don't need to be ordered in any specific way). Since the right spine is short, our meld algorithm should, when possible, try to work down the right spine. With this rough idea, if the number of steps required to meld is proportional to the length of the right spine, we have an efficient algorithm that runs in about $O(\log n)$ work.

To make use of ranks we add a rank field to every node and make a small change to our code to maintain the leftist property: the meld algorithm below effectively traverses the right spines of the heaps A and B. (Note how the recursive call to meld are only with either (R_a, B) or (A, R_b).)

Data Structure 21.10. [Leftist Heap]

```
datatype PQ =
    Leaf
  | Node of (int × key × PQ × PQ)

rank Leaf = 0
 | rank (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 $(A, B)$
  | (_, 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 $(A, R_b))$
```

Note that the only real difference is that we now use 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. It also maintains the rank value on each node.

Theorem 21.11. *If A and B are leftists heaps then the* meld(A, B) *algorithm runs in* $O(\log(|A|) +$

$\log(|B|)$ *work and returns a leftist heap containing the union of A and 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 $\mathsf{rank}(A) + \mathsf{rank}(B)$ steps, and each step does constant work. Since both trees are leftist, by Lemma 21.9, 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 routine guarantees that the rank of the left branch is at least as great as the rank of the right branch. □

Before proving Lemma 21.9 we will 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'll 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 $\mathsf{rank}(L(x)) \geq \mathsf{rank}(R(x)) = r - 1$. As the size of the tree rooted x is $1 + |L(x)| + |R(x)|$, the smallest size this tree can be is

$$n(r) = 1 + n(\mathsf{rank}(L(x))) + n(\mathsf{rank}(R(x))) \\ \geq 1 + n(r-1) + n(r-1) = 1 + 2 \cdot n(r-1).$$

Unfolding the recurrence, we get $n(r) \geq 2^r - 1$, which proves the claim.

Proof of Lemma 21.9. To prove that 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 $r \leq \log_2(n+1)$. □

21.4 Summary of Priority Queues

Already, we have seen a handful of data structures that can be used to implement a priority queue. Let's look at the performance guarantees they offer.

Implementation	`insert`	`findMin`	`deleteMin`	`meld`
(Unsorted) Sequence	$O(n)$	$O(n)$	$O(n)$	$O(m+n)$
Sorted Sequence	$O(n)$	$O(1)$	$O(n)$	$O(m+n)$
Balanced Trees (e.g. Treaps)	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(m \log(1+\frac{n}{m}))$
Leftist Heap	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log m + \log n)$

Indeed, a big win for leftist heap is in the super fast `meld` operation—logarithmic as opposed to roughly linear in other data structures.

.

Appendix A

Algorithm-Design Technique: Partitioning

Partitioning technique for algorithm design generalizes the divide-and-conquer technique. It allows us to divide the problem instance into not just a constant number of smaller instances but into many more sub-instances and solve each sub-instance independently in parallel, perhaps by using another technique. As in divide and conquer, we compute the solution for the original instance by combining the solutions to the sub-instances.

Base Case: When the instance I of the problem P is sufficiently small, compute the answer $P(I)$ perhaps by using a different algorithm.

Partition: Partition I into a number of smaller sub-instances of the same problem P.

Solve: Solve each sub-instance to obtain their solutions.

Combine: Combine the answers to produce a solution for the original instance I.

Partitioning is probably one of the most important parallel algorithm-design techniques. Like divide-and-conquer algorithms, partitioning algorithms involve solving a smaller instance of the same problem, but unlike in divide-and-conquer, there are many more possibly significantly smaller instances, and thus we can solve each smaller instance possibly by using the same or a different algorithm.

The key advantage of the partitioning technique over divide-and-conquer is that it can reduce the size of the problem instance by a large, non-constant amount in a single parallel step. This can lead to parallel algorithms with much smaller span.

In this chapter, we will consider several applications of the partitioning technique and analyze the resulting algorithms.

As our running example, we consider how to merge two sorted sequences by using partitoning. We can give a type signature for `merge` as follows.

```
merge (f: α * α → bool) (A: 𝕊_α) (B: 𝕊_α): 𝕊_α
```

where f is a comparison function that returns `true` if its first argument is less that its second. The second and third arguments are sequences.

The crux in using the partitoning technique is to design an algorithm for reducing an instance of the problem to a significantly smaller instance by performing a parallel partitoning step. In the case of `merge`, there are many ways to perform such a partitioning.

In the rest of this chapter, we consider three different applications of the partitioning technique, each of which yield three different results. For simplicity, we make several assumptions;

- We use n and m to refer to the length of the sequences, A and B to be merged, i.e., $n = |A|$ and $m = |B|$. We assume that n and m are powers of 2.
- We assume the sequences to be merged consist of unique elements and use

A.1 Example 1: Partitioning into Two

Let's consider partitioning the sequences into two subsequences.

Let's consider the middle element a of A and position $n_0 = n/2$, i.e., $a = A[n_0]$. We can partition B with a by performing a binary search on B to find the position m_0, such that for all $b = B[i]$, $b < a$ if $i < m_0$ and $b > a$ if $i > m_0$. This gives us two sub-instances to solve:

- $A[0 \ldots n_0 - 1]$ and $B[0 \ldots m_0 - 1]$
- $A[n_0 \ldots n - 1]$ and $B[m_0 \ldots m]$.

We can now solve these sub-instances recursively and concatenate the resulting solutions to solve the original instance.

Exercise A.1. Prove that this algorithm performs $O(n + m)$ work in $O(\lg^2 n + m)$ span.

A.2 Example 2: Partitioning into Many

Let's consider partitioning the sequences into many subsequences. The reason for why this could be effective is that we might be able to reduce the problem size into a very small size that we can solve immediately by using a serial algorithm.

For example, let's partition the sequence A into $p = n/\lg n$ subsequences at positions $n_0 = \lg n, n_1 = 2\lg n, \ldots, n_{p-2} = (p-1)\lg n$ and let $aa_0 = A[\lg n], aa_1 = A[2\lg n], \ldots aa_{p-2} = A[(p-1)\lg n$. We can now partition B into smaller sequences by using each element to perform a binary search to determine the position that separates the array into parts that are smaller and bigger than the element. Let $m_0 \ldots m_{p-2}$ denote the positions computed for $aa_0 \ldots a_{p_2}$.

INSERT PICTURE.

We now have p subproblem defined by

- $A[0 \ldots n_0 - 1]$-$B[0 \ldots m_0 - 1]$,
- $A[n_0 \ldots n_1 - 1]$-$B[m_0 \ldots m_0 - 1]$
- $\vdots$
- $A[n_{p-2} \ldots n - 1]$-$B[m_{p-2} \ldots m]$.

Let's analyze the work and span of the partitioning step. Selecting the p elements to split requires $O(n)$ work and $O(1)$ span. The binary searchers require $O(p \lg m)$ work and $O(\lg m)$ span. Creating the subproblems requires $O(n+m)$ work and $O(1)$ span. Thus the total work is $O(n + m + p\lg m)$ and the total span is $O(\lg m)$. To bound the total work, note that $p \lg m = \frac{n}{\lg n} \lg m = n \log_n m \leq m$ because this means that we are effectively encoding the number m in base n but then multiplying by the base n the length of the corresponding instead taking exponents, which is bound to give us a smaller number, except perhaps for same small values of n and m. Thus we conclude that the total work is $O(n+m)$ and the span is $O(\lg m)$.

We know something important about these subinstances: that each subsequence of A has exactly $\lg n$ elements. Thus, if the corresponding subsequence of B has also $O(\lg n)$ elements, for some constant of our choosing, then we can merge them serially in $O(\lg n)$ work and span.

Suppose, we get lucky and all of the sequences and we perform all of the p merges serially. We can then bound the total work as $O(n+m)$ and the span as $O(\lg n + \lg m) = O(\lg n + m)$ (because each term is bounded by $\lg n + m$).

If we are not lucky, then there is a simple solution. Our partitioning technique always produces sub-instances for the A array that are only log-size. Thus, we can apply the same partitioning technique to each subinstance but this time by reversing the place of A and B. This application will give us subsequences that are bounded both by $\lg n$ and $\lg m$ and will take $O(\lg \lg n)$ span and $O(n+m)$ work in total.

After running the partitioning step for a second time, we are now guaranteed to have subproblem, whose total size is bounded by $\lg n + \lg m$, we can thus merge the them all in $O(\lg n + m)$ work and span.

Thus summing over the two partitions and the merge, we have performed $O(n+m)$ work in $O(\lg n + m)$ span. This algorithm is asymptotically work efficient and has a very low span.

A.3 Example 3: Partitioning into Not as Many

Let's consider again partitioning the sequences into many subsequences, but not quite as many. Specifically, let's partition the sequence A into $p = \sqrt{n}$ subsequences at positions $n_0 = n\sqrt{n}, n_1 = 2n\sqrt{n}, \ldots, n_{p-2} = (p-1)n\sqrt{n}$ and let $aa_0 = A[\sqrt{n}], aa_1 = A[2\sqrt{n}], \ldots aa_{p-2} = A[(p-1)\sqrt{n}$. We can now partition B into smaller sequences by using each element to perform a binary search to determine the position that separates the array into parts that are smaller and bigger than the element. Let $m_0 \ldots m_{p-2}$ denote the positions computed for $aa_0 \ldots aa_{p_2}$.

INSERT PICTURE.

We now have p subproblem defined by

- $A[0 \ldots n_0 - 1]$-$B[0 \ldots m_0 - 1]$,
- $A[n_0 \ldots n_1 - 1]$-$B[m_0 \ldots m_0 - 1]$
- $\vdots$
- $A[n_{p-2} \ldots n - 1]$-$B[m_{p-2} \ldots m]$.

Let's analyze the work and span of the partitioning step. Selecting the p elements to split requires $O(p)$ work and $O(1)$ span. The binary searchers require $O(p \lg m)$ work and $O(\lg m)$ span. Creating the subproblems requires $O(p)$ work and $O(1)$ span. Thus the total work is $O(p + p \lg m)$ and the total span is $O(\lg m)$. To bound the total work, note that $p \lg m = \frac{n}{\lg n} \lg m = n \log_n m \leq m$ because this means that we are effectively encoding the number m in base n but then multiplying by the base n the length of the corresponding instead taking exponents, which is bound to give us a smaller number, except perhaps for same small values of n and m. Thus we conclude that the total work is $O(\sqrt{n} + m)$ and the span is $O(\lg m)$.

Since we have partitioned our sequence only $\sqrt{n}$ subsequences, the subinstances are quite large, so we can't solve them sequentially without taking a hit on the span. But, we can solve each subproblem recursively by using the same algorithm. This thus gives a complete algorithm: if the problem is small, that is A is say less than 16 elements, then solve directly by finding the rank of the elements of A in B and then constructing the output. Otherwise, partition A into $\sqrt{n}$ pieces and find the corresponding sub-sequences of B in $O(\sqrt{n} + m)$ work and $O(\lg m)$ span. Finally, recursively solve the resulting sub-instances and construct the final solution.

We can write the span of this algorithm as

$$S(n, m) = S(\sqrt{n}, m) + \lg m.$$

We can expand this as

$$S(n,m) = \sum_{i=1}^{\lg \lg n} \lg m,$$

which solves to $S(n,m) = O(\lg \lg n \cdot \lg m)$.

Similarly, we can write the work of this algorithm as

$$W(n,m) = \sqrt{n} \cdot W(\sqrt{n}, m) + \sqrt{n} + m.$$

But this is a big overapproximation, because it does not take into account the fact that B in partitioned into pieces. We can observe, Observe now that the worst-case partitioning for B is the even partitioning. Thus, we have

$$W(n,m) = \sqrt{n} \cdot W(\sqrt{n}, \sqrt{m}) + \sqrt{n} + m.$$

This can be bounded by $O(n + m \lg \lg n)$.

Exercise A.2. Work out the missing details of the analysis above. Can you improve on the bound? Perhaps by changing the algorithm to make it work efficient?

.

Index