

13 Online Algorithms	368
13.1 The Online Paging Problem	369
13.2 Adversary Models	372
13.3 Paging against an Oblivious Adversary	374
13.4 Relating the Adversaries	377
13.5 The Adaptive Online Adversary	381
13.6 The k -Server Problem	384
Notes	387
Problems	389
14 Number Theory and Algebra	392
14.1 Preliminaries	392
14.2 Groups and Fields	395
14.3 Quadratic Residues	402
14.4 The RSA Cryptosystem	410
14.5 Polynomial Roots and Factors	412
14.6 Primality Testing	417
Notes	426
Problems	427
<i>Appendix A</i> Notational Index	429
<i>Appendix B</i> Mathematical Background	433
<i>Appendix C</i> Basic Probability Theory	438
References	447
Index	467

Preface

THE last decade has witnessed a tremendous growth in the area of randomized algorithms. During this period, randomized algorithms went from being a tool in computational number theory to finding widespread application in many types of algorithms. Two benefits of randomization have spearheaded this growth: simplicity and speed. For many applications, a randomized algorithm is the simplest algorithm available, or the fastest, or both.

This book presents the basic concepts in the design and analysis of randomized algorithms at a level accessible to advanced undergraduates and to graduate students. We expect it will also prove to be a reference to professionals wishing to implement such algorithms and to researchers seeking to establish new results in the area.

Organization and Course Information

We assume that the reader has had undergraduate courses in Algorithms and Complexity, and in Probability Theory. The book is organized into two parts. The first part, consisting of seven chapters, presents basic tools from probability theory and probabilistic analysis that are recurrent in algorithmic applications. Applications are given along with each tool to illustrate the tool in concrete settings. The second part of the book also contains seven chapters, each focusing on one area of application of randomized algorithms. The seven areas of application we have selected are: data structures, graph algorithms, geometric algorithms, number theoretic algorithms, counting algorithms, parallel and distributed algorithms, and online algorithms. Naturally, some of the algorithms used for illustration in Part I do fall into one of these seven categories. The book is not meant to be a compendium of every randomized algorithm that has been devised, but rather a comprehensive and representative selection. The Appendices review basic material on probability theory.

We have taught several regular as well as short-term courses based on the material in this book, as have some of our colleagues. It is virtually impossible to cover all the material in the book in a single academic term or in a week's intensive course. We regard Chapters 1–4 as the core around which a course may be built. Following the treatment of this material, the instructor may continue with that portion of the remainder of Part I that supports the material of Part II (s)he wishes to cover. Chapters 5–13 depend only on material in Chapters 1–4, with the following exceptions:

1. Chapter 5 on Probabilistic Methods is a prerequisite for Chapters 6 (Random Walks) and 11 (Approximate Counting).
2. Chapter 6 on Random Walks is a prerequisite for Chapter 11 (Approximate Counting).
3. Chapter 7 on Algebraic Techniques is a prerequisite for Chapters 14 (Number Theory and Algebra) and 12 (Parallel and Distributed Algorithms).

We have included three types of problems in the book. **Exercises** occur throughout the text, and are designed to deepen the reader's understanding of the material being covered in the text. Usually, an exercise will be a variant, extension, or detail of an algorithm or proof being studied. **Problems** appear at the end of each chapter and are meant to be more difficult and involved than the **Exercises** in the text. In addition, **Research Problems** are listed in the Discussion section at the end of each chapter. These are problems that were open at the time we wrote the book; we offer them as suggestions for students (and of course professional researchers) to work on.

Based on our experience with teaching this material, we recommend that the instructor use one of the following course organizations:

- A comprehensive basic course: In addition to Chapters 1–4, this course would cover the material in Chapters 5, 6, and 7 (thus spanning all of Part I).
- A course oriented toward algebra and number theory: Following Chapters 1–4, this course would cover Chapters 7, 14, and 12.
- A course oriented toward graphs, data structures, and geometry: Following Chapters 1–4, this course would cover Chapters 8, 9, and 10.
- A course oriented toward random walks and counting algorithms: Following Chapters 1–4, this course would cover Chapters 5, 6, and 11.

Each of these courses may be pruned and given in abridged form as an intensive course spanning 3–5 days.

Paradigms for Randomized Algorithms

A handful of general principles lies at the heart of almost all randomized algorithms, despite the multitude of areas in which they find application. We briefly survey these here, with pointers to chapters in which examples of these

principles may be found. The following summary draws heavily from ideas in the survey paper by Karp [243].

Foiling an adversary. The classical adversary argument for a deterministic algorithm establishes a lower bound on the running time of the algorithm by constructing an input on which the algorithm fares poorly. The input thus constructed may be different for each deterministic algorithm. A randomized algorithm can be viewed as a probability distribution on a set of deterministic algorithms. While the adversary may be able to construct an input that foils one (or a small fraction) of the deterministic algorithms in the set, it is difficult to devise a single input that is likely to defeat a randomly chosen algorithm. While this paradigm underlies the success of *any* randomized algorithm, the most direct examples appear in Chapter 2 (in game tree evaluation), Chapter 7 (in efficient proof verification), and Chapter 13 (in online algorithms).

Random sampling. The idea that a random sample from a population is representative of the population as a whole is a pervasive theme in randomized algorithms. Examples of this paradigm arise in almost all the chapters, most notably in Chapters 3 (selection algorithms), 8 (data structures), 9 (geometric algorithms), 10 (graph algorithms), and 11 (approximate counting).

Abundance of witnesses. Often, an algorithm is required to determine whether an input (say, a number x) has a certain property (for example, “is x prime?”). It does so by finding a *witness* that x has the property. For many problems, the difficulty with doing this deterministically is that the witness lies in a search space that is too large to be searched exhaustively. However, by establishing that the space contains a large number of witnesses, it often suffices to choose an element at random from the space. The randomly chosen item is likely to be a witness; further, independent repetitions of the process reduce the probability that a witness is not found on any of the repetitions. The most striking examples of this phenomenon occur in number theory (Chapter 14).

Fingerprinting and hashing. A long string may be represented by a short *fingerprint* using a random mapping. In some pattern-matching applications, it can be shown that two strings are likely to be identical if their fingerprints are identical; comparing the short fingerprints is considerably faster than comparing the strings themselves (Chapter 7). This is also the idea behind *hashing*, whereby a small set S of elements drawn from a large universe is mapped into a smaller universe with a guarantee that distinct elements in S are likely to have distinct images. This leads to efficient schemes for deciding membership in S (Chapters 7 and 8) and has a variety of further applications in generating pseudo-random numbers (for example, two-point sampling in Chapter 3 and pairwise independence in Chapter 12) and complexity theory (for instance, algebraic identities and efficient proof verification in Chapter 7).

Random re-ordering. A striking use of randomization in a number of problems in data structuring and computational geometry involves randomly re-ordering the input data, followed by the application of a relatively naive algorithm. After the re-ordering step, the input is unlikely to be in one of the orderings that is pathological for the naive algorithm. (Chapters 8 and 9).

Load balancing. For problems involving choice between a number of resources, such as communication links in a network of processors, randomization can be used to “spread” the load evenly among the resources, as demonstrated in Chapter 4. This is particularly useful in a parallel or distributed environment where resource utilization decisions have to be made locally at a large number of sites without reference to the global impact of these decisions.

Rapidly mixing Markov chains. For a variety of problems involving counting the number of combinatorial objects with a given property, we have approximation algorithms based on randomly sampling an appropriately defined population. Such sampling is often difficult because it may require computing the size of the sample space, which is precisely the problem we would like to solve via sampling. In some cases, the sampling can be achieved by defining a Markov chain on the elements of the population and showing that a short random walk using this Markov chain is likely to sample the population uniformly (Chapter 11).

Isolation and symmetry breaking. In parallel computation, when solving a problem with many feasible solutions it is important to ensure that the different processors are working toward finding the same solution. This requires isolating a specific solution out of the space of all feasible solutions without actually knowing any single element of the solution space. A clever randomized strategy achieves *isolation*, by implicitly choosing a random ordering on the feasible solutions and then requiring the processors to focus on finding the solution of lowest rank. In distributed computation, it is often necessary for a collection of processors to break a deadlock and arrive at a consensus. Randomization is a powerful tool in such deadlock-avoidance, as shown in Chapter 12.

Probabilistic methods and existence proofs. It is possible to establish that an object with certain properties exists by arguing that a randomly chosen object has the properties with positive probability. Such an argument gives no clue as to how to find such an object. Sometimes, the method is used to guarantee the existence of an algorithm for solving a problem; we thus know that the algorithm exists, but have no idea what it looks like or how to construct it. This raises the issue of *non-uniformity* in algorithms (Chapters 2 and 5).

Conventions

Most of the conventions we use are described where they first arise. One worth mentioning here is the issue of *integer breakage*: as long as it does not materially affect the algorithm or analysis being considered (and the intent is unambiguous from the context), we omit ceilings and floors from numbers that strictly should be integers. Thus, we might say “choose \sqrt{n} elements from the set of size n ” even when n is not a perfect square. Our intent is to present the crux of the algorithm/analysis without undue notational clutter from ceilings and floors. The expression $\log x$ denotes $\log_2 x$, and the expression $\ln x$ denotes the natural logarithm of x .

Acknowledgements

This book would not have been possible without the guidance and tutelage of Dick Karp. It was he who taught us this field and gave us invaluable guidance at every stage of the book – from the initial planning to the feedback he gave us from using a preliminary version of the manuscript in a graduate course at Berkeley.

We thank the following colleagues, who carefully read portions of the manuscript and pointed out many errors in early versions: Pankaj Agarwal, Donald Aingworth, Susanne Albers, David Aldous, Noga Alon, Sanjeev Arora, Julien Basch, Allan Borodin, Joan Boyar, Andrei Broder, Bernard Chazelle, Ken Clarkson, Don Coppersmith, Cynthia Dwork, Michael Goldwasser, David Gries, Kazuyoshi Hayase, Mary Inaba, Sandy Irani, David Karger, Anna Karlin, Don Knuth, Tom Leighton, Mike Luby, Keju Ma, Karthik Mahadevan, Colin McDiarmid, Ketan Mulmuley, Seffi Naor, Daniel Panario, Bill Pulleyblank, Vijaya Ramachandran, Raimund Seidel, Tom Shiple, Alistair Sinclair, Joel Spencer, Madhu Sudan, Hisao Tamaki, Martin Tompa, Gert Vegter, Jeff Vitter, Peter Winkler, and David Zuckerman. We apologize in advance to any colleagues whose names we have inadvertently omitted.

Special thanks go to Allan Borodin and the students of his CSC 2421 class at the University of Toronto (Fall 1994), as well as to Gudmund Skovbjerg Frandsen, Prabhakar Ragde, and Eli Upfal for giving us detailed feedback from courses they taught using early versions of the manuscript. Their suggestions and advice have been invaluable in making this book more suitable for the classroom.

We thank Rao Kosaraju, Ron Rivest, Joel Spencer, Jeff Ullman, and Paul Vitanyi for providing us with much help and advice on the process of writing and improving the manuscript.

The first author is grateful to Stanford University for the environment and resources which made this effort possible. Several colleagues in the Computer Science Department provided invaluable advice and encouragement. Don Knuth played the role of mentor and his faith in this project was a tremendous source of encouragement. John Mitchell and Jeff Ullman were especially helpful with the mechanics of the publication process. This book owes a great deal to the students, teaching assistants, and other participants in the various offerings of the course CS 365 (Randomized Algorithms) at Stanford. The feedback from these people was invaluable in refining the lecture notes that formed a partial basis for this book. Steven Phillips made a significant contribution as a teaching assistant in CS 365 on two different occasions. Special thanks are due to Yossi Azar, Amotz Bar-Noy, Bob Floyd, Seffi Naor, and Boris Pittel for their guest lectures and help in preparing class notes. The following students transcribed some lecture notes, and their class participation was vital to the development of this material: Julien Basch, Trevor Bourget, Tom Chavez, Edith Cohen, Anil Gangolli, Michael Goldwasser, Bert Hackney, Alan Hu, Jim Hwang, Vasilis Kallistros, Anil Kamath, David Karger, Robert Kennedy, Sanjeev Khanna,

Daphne Koller, Andrew Kosoresow, Sherry Listgarten, Alan Morgan, Steve Newman, Jeffrey Oldham, Steven Phillips, Tomasz Radzik, Ram Ramkumar, Will Sawyer, Sunny Siu, Eric Torng, Theodora Varvarigou, Eric Veach, Alex Wang, and Paul Zhang.

The research and book-writing efforts of the first author have been supported by the following grants and awards: the Bergmann Award from the US-Israel Binational Science Foundation; an IBM Faculty Development Award; gifts from the Mitsubishi Corporation; NSF Grant CCR-9010517; the NSF Young Investigator Award CCR-9357849, with matching funds from IBM Corporation, Schlumberger Foundation, Shell Foundation, and Xerox Corporation; and various grants from the Office of Technology Licensing at Stanford University.

The second author is indebted to his colleagues at the Mathematical Sciences Department of the IBM Thomas J. Watson Research Center, and to the IBM Corporation for providing the facilities and environment that made it possible to write this book. He also thanks Sandeep Bhatt for his encouragement and support of a course on Randomized Algorithms taught by the author at Yale University; the class notes from that course formed a partial basis for this book.

We are indebted to Lauren Cowles of Cambridge University Press for her editorial help and advice in the preparation of the manuscript; this book has emerged much improved as a result of her untiring efforts.

Rajeev Motwani thanks his wife Asha for her love, encouragement, and cheerfulness; without her distractions this book would have been completed several months earlier. This task would not have been possible without the constant support and faith of his family over the years. Finally, the two mutts Tipu and Noori deserve special mention for giving company during the many late night editing sessions.

Prabhakar Raghavan thanks his wife Srilatha for her love and support, his parents for their inspiration, and his children Megha and Manish for ensuring that there was never a dull moment when writing this book.

World-Wide Web

Current information on this book may be found at the following address on the World-Wide Web:

<http://www.cup.org/Reviews&blurbs/RanAlg/RanAlg.html>

This address may be used for ordering information, reporting errors and viewing an edited list of errors found by other readers.

PART ONE

Tools and Techniques

CHAPTER 1

Introduction

CONSIDER sorting a set S of n numbers into ascending order. If we could find a member y of S such that half the members of S are smaller than y , then we could use the following scheme. We partition $S \setminus \{y\}$ into two sets S_1 and S_2 , where S_1 consists of those elements of S that are smaller than y , and S_2 has the remaining elements. We recursively sort S_1 and S_2 , then output the elements of S_1 in ascending order, followed by y , and then the elements of S_2 in ascending order. In particular, if we could find y in cn steps for some constant c , we could partition $S \setminus \{y\}$ into S_1 and S_2 in $n - 1$ additional steps by comparing each element of S with y ; thus, the total number of steps in our sorting procedure would be given by the recurrence

$$T(n) \leq 2T(n/2) + (c + 1)n, \quad (1.1)$$

where $T(k)$ represents the time taken by this method to sort k numbers on the worst-case input. This recurrence has the solution $T(n) \leq c'n \log n$ for a constant c' , as can be verified by direct substitution.

The difficulty with the above scheme in practice is in finding the element y that splits $S \setminus \{y\}$ into two sets S_1 and S_2 of the same size. Examining (1.1), we notice that the running time of $O(n \log n)$ can be obtained even if S_1 and S_2 are *approximately* the same size – say, if y were to split $S \setminus \{y\}$ such that neither S_1 nor S_2 contained more than $3n/4$ elements. This gives us hope, because we know that every input S contains at least $n/2$ candidate splitters y with this property. How do we quickly find one?

One simple answer is to choose an element of S at random. This does not always ensure a splitter giving a roughly even split. However, it is reasonable to hope that in the recursive algorithm we will be lucky fairly often. The result is an algorithm we call **RandQS**, for Randomized Quicksort.

Algorithm **RandQS** is an example of a *randomized algorithm* – an algorithm that makes random choices during execution (in this case, in Step 1). Let us assume for the moment that this random choice can be made in unit time; we

will say more about this in the Notes section. What can we prove about the running time of **RandQS**?

Algorithm RandQS:

Input: A set of numbers S .

Output: The elements of S sorted in increasing order.

1. Choose an element y uniformly at random from S : every element in S has equal probability of being chosen.
2. By comparing each element of S with y , determine the set S_1 of elements smaller than y and the set S_2 of elements larger than y .
3. Recursively sort S_1 and S_2 . Output the sorted version of S_1 , followed by y , and then the sorted version of S_2 .

As is usual for sorting algorithms, we measure the running time of **RandQS** in terms of the number of comparisons it performs since this is the dominant cost in any reasonable implementation. In particular, our goal is to analyze the *expected* number of comparisons in an execution of **RandQS**. Note that all the comparisons are performed in Step 2, in which we compare a randomly chosen partitioning element to the remaining elements. For $1 \leq i \leq n$, let $S_{(i)}$ denote the element of *rank* i (the i th smallest element) in the set S . Thus, $S_{(1)}$ denotes the smallest element of S , and $S_{(n)}$ the largest. Define the random variable X_{ij} to assume the value 1 if $S_{(i)}$ and $S_{(j)}$ are compared in an execution, and the value 0 otherwise. Thus, X_{ij} is a count of comparisons between $S_{(i)}$ and $S_{(j)}$, and so the total number of comparisons is $\sum_{i=1}^n \sum_{j>i}^n X_{ij}$. We are interested in the expected number of comparisons, which is clearly

$$\mathbf{E}\left[\sum_{i=1}^n \sum_{j>i}^n X_{ij}\right] = \sum_{i=1}^n \sum_{j>i}^n \mathbf{E}[X_{ij}]. \quad (1.2)$$

This equation uses an important property of expectations called *linearity of expectation*; we will return to this in Section 1.3.

Let p_{ij} denote the probability that $S_{(i)}$ and $S_{(j)}$ are compared in an execution. Since X_{ij} only assumes the values 0 and 1,

$$\mathbf{E}[X_{ij}] = p_{ij} \times 1 + (1 - p_{ij}) \times 0 = p_{ij}. \quad (1.3)$$

To facilitate the determination of p_{ij} , we view the execution of **RandQS** as a binary tree T , each node of which is labeled with a distinct element of S . The root of the tree is labeled with the element y chosen in Step 1, the left sub-tree of y contains the elements in S_1 and the right sub-tree of y contains the elements in S_2 . The structures of the two sub-trees are determined recursively by the executions of **RandQS** on S_1 and S_2 . The root y is compared to the elements in the two sub-trees, but no comparison is performed between an element of the left sub-tree and an element of the right sub-tree. Thus, there is a comparison

between $S_{(i)}$ and $S_{(j)}$ if and only if one of these elements is an ancestor of the other.

The in-order traversal of T will visit the elements of S in a sorted order, and this is precisely what the algorithm outputs; in fact, T is a (random) binary search tree (we will encounter this again in Section 8.2). However, for the analysis we are interested in the level-order traversal of the nodes. This is the permutation π obtained by visiting the nodes of T in increasing order of the level numbers, and in a left-to-right order within each level; recall that the i th level of the tree is the set of all nodes at distance exactly i from the root.

To compute p_{ij} , we make two observations. Both observations are deceptively simple, and yet powerful enough to facilitate the analysis of a number of more complicated algorithms in later chapters (for example, in Chapters 8 and 9).

1. There is a comparison between $S_{(i)}$ and $S_{(j)}$ if and only if $S_{(i)}$ or $S_{(j)}$ occurs earlier in the permutation π than any element $S_{(\ell)}$ such that $i < \ell < j$. To see this, let $S_{(k)}$ be the earliest in π from among all elements of rank between i and j . If $k \notin \{i, j\}$, then $S_{(i)}$ will belong to the left sub-tree of $S_{(k)}$ while $S_{(j)}$ will belong to the right sub-tree of $S_{(k)}$, implying that there is no comparison between $S_{(i)}$ and $S_{(j)}$. Conversely, when $k \in \{i, j\}$, there is an ancestor-descendant relationship between $S_{(i)}$ and $S_{(j)}$, implying that the two elements are compared by **RandQS**.
2. Any of the elements $S_{(i)}, S_{(i+1)}, \dots, S_{(j)}$ is equally likely to be the first of these elements to be chosen as a partitioning element, and hence to appear first in π . Thus, the probability that this first element is either $S_{(i)}$ or $S_{(j)}$ is exactly $2/(j - i + 1)$.

We have thus established that $p_{ij} = 2/(j - i + 1)$. By (1.2) and (1.3), the expected number of comparisons is given by

$$\begin{aligned} \sum_{i=1}^n \sum_{j>i}^n p_{ij} &= \sum_{i=1}^n \sum_{j>i}^n \frac{2}{j - i + 1} \\ &\leq \sum_{i=1}^n \sum_{k=1}^{n-i+1} \frac{2}{k} \\ &\leq 2 \sum_{i=1}^n \sum_{k=1}^n \frac{1}{k}. \end{aligned}$$

It follows that the expected number of comparisons is bounded above by $2nH_n$, where H_n is the n th *Harmonic number*, defined by $H_n = \sum_{k=1}^n 1/k$.

Theorem 1.1: *The expected number of comparisons in an execution of **RandQS** is at most $2nH_n$.*

From Proposition B.4 (Appendix B), we have that $H_n \sim \ln n + \Theta(1)$, so that the expected running time of **RandQS** is $O(n \log n)$.

Exercise 1.1: Consider the (random) permutation π of S induced by the level-order traversal of the tree T corresponding to an execution of **RandQS**. Is π uniformly distributed over the space of all permutations of the elements $S_{(1)}, \dots, S_{(n)}$?

It is worth examining carefully what we have just established about **RandQS**. The expected running time *holds for every input*. It is an expectation that depends only on the random choices made by the algorithm, and *not* on any assumptions about the distribution of the input. The behavior of a randomized algorithm can vary even on a single input, from one execution to another. The running time becomes a random variable, and the running-time analysis involves understanding the distribution of this random variable.

We will prove bounds on the performances of randomized algorithms that rely solely on their random choices and not on any assumptions about the inputs. It is important to distinguish this from the *probabilistic analysis of an algorithm*, in which one assumes a distribution on the inputs and analyzes an algorithm that may itself be deterministic. In this book we will generally not deal with such probabilistic analysis, except occasionally when illustrating a technique for analyzing randomized algorithms.

Note also that we have proved a bound on the *expected* running time of the algorithm. In many cases (including **RandQS**, see Problem 4.14), we can prove an even stronger statement: that *with very high probability* the running time of the algorithm is not much more than its expectation. Thus, on almost every execution, independent of the input, the algorithm is shown to be fast.

The randomization involved in our **RandQS** algorithm occurs only in Step 1, where a random element is chosen from a set. We define a randomized algorithm as an algorithm that is allowed access to a source of independent, unbiased, random bits; it is then permitted to use these random bits to influence its computation. It is easy to sample a random element from a set S by choosing $O(\log |S|)$ random bits and then using these bits to index an element in the set. However, some distributions cannot be sampled using only random bits. For example, consider an algorithm that picks a random real number from some interval. This requires infinitely many random bits. While we will usually not worry about the conversion of random bits to the desired distribution, the reader should keep in mind that random bits are a resource whose use involves a non-trivial cost. Moreover, there is sometimes a non-trivial computational overhead associated with sampling from a seemingly well-behaved distribution. For example, consider the problem of using a source of unbiased random bits to sample uniformly from a set S whose cardinality is *not* a power of 2 (see Problem 1.2).

There are two principal advantages to randomized algorithms. The first is performance – for many problems, randomized algorithms run faster than the best known deterministic algorithms. Second, many randomized algorithms are simpler to describe and implement than deterministic algorithms of comparable

performance. The randomized sorting algorithm described above is an example. This book presents many other randomized algorithms that enjoy these advantages.

In the next few sections, we will illustrate some basic ideas from probability theory using simple applications to randomized algorithms. The reader wishing to review some of the background material on the analysis of algorithms or on elementary probability theory is referred to the Appendices.

1.1. A Min-Cut Algorithm

Two events \mathcal{E}_1 and \mathcal{E}_2 are said to be *independent* if the probability that they both occur is given by

$$\Pr[\mathcal{E}_1 \cap \mathcal{E}_2] = \Pr[\mathcal{E}_1] \times \Pr[\mathcal{E}_2] \quad (1.4)$$

(see Appendix C). In the more general case where \mathcal{E}_1 and \mathcal{E}_2 are not necessarily independent,

$$\Pr[\mathcal{E}_1 \cap \mathcal{E}_2] = \Pr[\mathcal{E}_1 | \mathcal{E}_2] \times \Pr[\mathcal{E}_2] = \Pr[\mathcal{E}_2 | \mathcal{E}_1] \times \Pr[\mathcal{E}_1], \quad (1.5)$$

where $\Pr[\mathcal{E}_1 | \mathcal{E}_2]$ denotes the *conditional probability* of \mathcal{E}_1 given \mathcal{E}_2 . Sometimes, when a collection of events is not independent, a convenient method for computing the probability of their intersection is to use the following generalization of (1.5).

$$\Pr[\cap_{i=1}^k \mathcal{E}_i] = \Pr[\mathcal{E}_1] \times \Pr[\mathcal{E}_2 | \mathcal{E}_1] \times \Pr[\mathcal{E}_3 | \mathcal{E}_1 \cap \mathcal{E}_2] \cdots \Pr[\mathcal{E}_k | \cap_{i=1}^{k-1} \mathcal{E}_i]. \quad (1.6)$$

Consider a graph-theoretic example. Let G be a connected, undirected multigraph with n vertices. A *multigraph* may contain multiple edges between any pair of vertices. A *cut* in G is a set of edges whose removal results in G being broken into two or more components. A *min-cut* is a cut of minimum cardinality. We now study a simple algorithm for finding a min-cut of a graph.

We repeat the following step: pick an edge uniformly at random and merge the two vertices at its end-points (Figure 1.1). If as a result there are several edges between some pairs of (newly formed) vertices, retain them all. Edges between vertices that are merged are removed, so that there are never any self-loops. We refer to this process of merging the two end-points of an edge into a single vertex as the *contraction* of that edge. With each contraction, the number of vertices of G decreases by one. The crucial observation is that an edge contraction does not reduce the min-cut size in G . This is because every cut in the graph at any intermediate stage is a cut in the original graph. The algorithm continues the contraction process until only two vertices remain; at this point, the set of edges between these two vertices is a cut in G and is output as a candidate min-cut.

Does this algorithm always find a min-cut? Let us analyze its behavior after first reviewing some elementary definitions from graph theory.

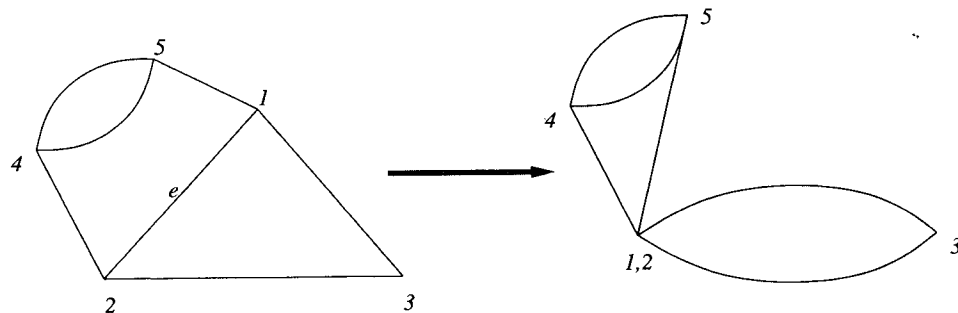


Figure 1.1: A step in the min-cut algorithm; the effect of contracting edge $e = (1, 2)$ is shown.

► **Definition 1.1:** For any vertex v in a multigraph G , the *neighborhood* of v , denoted $\Gamma(v)$, is the set of vertices of G that are adjacent to v . The *degree* of v , denoted $d(v)$, is the number of edges incident on v . For a set S of vertices of G , the neighborhood of S , denoted $\Gamma(S)$, is the union of the neighborhoods of the constituent vertices.

Note that $d(v)$ is the same as the cardinality of $\Gamma(v)$ when there are no self-loops or multiple edges between v and any of its neighbors.

Let k be the min-cut size. We fix our attention on a particular min-cut C with k edges. Clearly G has at least $kn/2$ edges; otherwise there would be a vertex of degree less than k , and its incident edges would be a min-cut of size less than k . We will bound from below the probability that no edge of C is ever contracted during an execution of the algorithm, so that the edges surviving till the end are exactly the edges in C .

Let \mathcal{E}_i denote the event of *not* picking an edge of C at the i th step, for $1 \leq i \leq n-2$. The probability that the edge randomly chosen in the first step is in C is at most $k/(nk/2) = 2/n$, so that $\Pr[\mathcal{E}_1] \geq 1 - 2/n$. Assuming that \mathcal{E}_1 occurs, during the second step there are at least $k(n-1)/2$ edges, so the probability of picking an edge in C is at most $2/(n-1)$, so that $\Pr[\mathcal{E}_2 \mid \mathcal{E}_1] \geq 1 - 2/(n-1)$. At the i th step, the number of remaining vertices is $n-i+1$. The size of the min-cut is still at least k , so the graph has at least $k(n-i+1)/2$ edges remaining at this step. Thus, $\Pr[\mathcal{E}_i \mid \cap_{j=1}^{i-1} \mathcal{E}_j] \geq 1 - 2/(n-i+1)$. What is the probability that no edge of C is ever picked in the process? We invoke (1.6) to obtain

$$\Pr[\cap_{i=1}^{n-2} \mathcal{E}_i] \geq \prod_{i=1}^{n-2} \left(1 - \frac{2}{n-i+1}\right) = \frac{2}{n(n-1)}.$$

The probability of discovering a particular min-cut (which may in fact be the unique min-cut in G) is larger than $2/n^2$. Thus our algorithm may err in declaring the cut it outputs to be a min-cut. Suppose we were to repeat the above algorithm $n^2/2$ times, making independent random choices each time. By (1.4), the probability that a min-cut is not found in any of the $n^2/2$

attempts is at most

$$\left(1 - \frac{2}{n^2}\right)^{n^2/2} < 1/e.$$

By this process of repetition, we have managed to reduce the probability of failure from $1 - 2/n^2$ to a more respectable $1/e$. Further executions of the algorithm will make the failure probability arbitrarily small – the only consideration being that repetitions increase the running time.

Note the extreme simplicity of the randomized algorithm we have just studied. In contrast, most deterministic algorithms for this problem are based on network flows and are considerably more complicated. In Section 10.2 we will return to the min-cut problem and fill in some implementation details that have been glossed over in the above presentation; in fact, it will be shown that a variant of this algorithm has an expected running time that is significantly smaller than that of the best known algorithms based on network flow.

Exercise 1.2: Suppose that at each step of our min-cut algorithm, instead of choosing a random edge for contraction we choose two vertices at random and coalesce them into a single vertex. Show that there are inputs on which the probability that this modified algorithm finds a min-cut is exponentially small.

1.2. Las Vegas and Monte Carlo

The randomized sorting algorithm and the min-cut algorithm exemplify two different types of randomized algorithms. The sorting algorithm *always* gives the correct solution. The only variation from one run to another is its running time, whose distribution we study. We call such an algorithm a *Las Vegas algorithm*.

In contrast, the min-cut algorithm may sometimes produce a solution that is incorrect. However, we are able to bound the probability of such an incorrect solution. We call such an algorithm a *Monte Carlo algorithm*. In Section 1.1 we observed a useful property of a Monte Carlo algorithm: if the algorithm is run repeatedly with independent random choices each time, the failure probability can be made arbitrarily small, at the expense of running time. Later, we will see examples of algorithms in which both the running time and the quality of the solution are random variables; sometimes these are also referred to as Monte Carlo algorithms. For decision problems (problems for which the answer to an instance is YES or NO), there are two kinds of Monte Carlo algorithms: those with *one-sided error*, and those with *two-sided error*. A Monte Carlo algorithm is said to have two-sided error if there is a non-zero probability that it errs when it outputs either YES or NO. It is said to have one-sided error if the probability that it errs is zero for at least one of the possible outputs (YES/NO) that it produces.

We will see examples of all three types of algorithms – Las Vegas, Monte Carlo with one-sided error, and Monte Carlo with two-sided error – in this book.”

Which is better, Monte Carlo or Las Vegas? The answer depends on the application – in some applications an incorrect solution may be catastrophic. A Las Vegas algorithm is by definition a Monte Carlo algorithm with error probability 0. The following exercise gives us a way of deriving a Las Vegas algorithm from a Monte Carlo algorithm. Note that the efficiency of the derivation procedure depends on the time taken to verify the correctness of a solution to the problem.

Exercise 1.3: Consider a Monte Carlo algorithm A for a problem Π whose expected running time is at most $T(n)$ on any instance of size n and that produces a correct solution with probability $\gamma(n)$. Suppose further that given a solution to Π , we can verify its correctness in time $t(n)$. Show how to obtain a Las Vegas algorithm that always gives a correct answer to Π and runs in expected time at most $(T(n) + t(n))/\gamma(n)$.

In attempting Exercise 1.3 the reader will have to use a simple property of the *geometric random variable* (Appendix C). Consider a biased coin that, on a toss, has probability p of coming up HEADS and $1 - p$ of coming up TAILS. What is the expected number of (independent) tosses up to and including the first head? The number of such tosses is a random variable that is said to be *geometrically distributed*. The expectation of this random variable is $1/p$. This fact will prove useful in numerous applications.

Exercise 1.4: Let $0 < \epsilon_2 < \epsilon_1 < 1$. Consider a Monte Carlo algorithm that gives the correct solution to a problem with probability at least $1 - \epsilon_1$, regardless of the input. How many independent executions of this algorithm suffice to raise the probability of obtaining a correct solution to at least $1 - \epsilon_2$, regardless of the input?

We say that a Las Vegas algorithm is an *efficient Las Vegas* algorithm if on any input its expected running time is bounded by a polynomial function of the input size. Similarly, we say that a Monte Carlo algorithm is an *efficient Monte Carlo* algorithm if on any input its worst-case running time is bounded by a polynomial function of the input size.

1.3. Binary Planar Partitions

We now illustrate another very useful and basic tool from probability theory: *linearity of expectation*. For random variables X_1, X_2, \dots ,

$$\mathbf{E}[\sum_i X_i] = \sum_i \mathbf{E}[X_i]. \quad (1.7)$$

(See Proposition C.5.) We have implicitly used this tool in our analysis of **RandQS**. A point that cannot be overemphasized is that (1.7) holds *regardless* of any dependencies between the X_i .

► **Example 1.1:** A ship arrives at a port, and the 40 sailors on board go ashore for revelry. Later at night, the 40 sailors return to the ship and, in their state of inebriation, each chooses a random cabin to sleep in. What is the expected number of sailors sleeping in their own cabins?

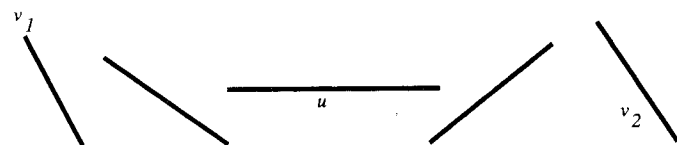
The inefficient approach to this problem would be to consider all 40^{40} arrangements of sailors in cabins. The solution to this example will involve the use of a simple and often useful device called an *indicator variable*, together with linearity of expectation. Let X_i be 1 if the i th sailor chooses her own cabin, and 0 otherwise. Thus X_i indicates whether or not a certain event occurs, and is hence called an indicator variable. We wish to determine the expected number of sailors who get their own cabins, which is $\mathbf{E}[\sum_{i=1}^{40} X_i]$. By linearity of expectation, this is $\sum_{i=1}^{40} \mathbf{E}[X_i]$. Since the cabins are chosen at random, the probability that the i th sailor gets her own cabin is $1/40$, so $\mathbf{E}[X_i] = 1/40$. Thus the expected number of sailors who get their own cabins is $\sum_{i=1}^{40} 1/40 = 1$.

Our next illustration is the construction of a *binary planar partition* of a set of n disjoint line segments in the plane, a problem with applications to computer graphics. A binary planar partition consists of a binary tree together with some additional information, as described below. Every internal node of the tree has two children. Associated with each node v of the tree is a region $r(v)$ of the plane. Associated with each internal node v of the tree is a line $\ell(v)$ that intersects $r(v)$. The region corresponding to the root is the entire plane. The region $r(v)$ is partitioned by $\ell(v)$ into two regions $r_1(v)$ and $r_2(v)$, which are the regions associated with the two children of v . Thus, any region r of the partition is bounded by the partition lines on the path from the root to the node corresponding to r in the tree.

Given a set $S = \{s_1, s_2, \dots, s_n\}$ of non-intersecting line segments in the plane, we wish to find a binary planar partition such that every region in the partition contains at most one line segment (or a portion of one line segment). Notice that the definition allows us to divide an input line segment s_i into several segments s_{i1}, s_{i2}, \dots , each of which lies in a different region. The example of Figure 1.2 gives such a partition for a set of three line segments (dark lines).

Exercise 1.5: Show that there exists a set of line segments for which no binary planar partition can avoid breaking up some of the segments into pieces, if each segment is to lie in a different region of the partition.

Binary planar partitions have two applications in computer graphics. Here, we describe one of them, the problem of *hidden line elimination* in computer

Figure 1.3: An illustration of $\text{index}(u, v)$.

linearity of expectation this equals

$$n + \sum_u \sum_{v \neq u} \Pr[u \rightarrow v] \leq n + \sum_u \sum_{v \neq u} \frac{1}{\text{index}(u, v) + 1}. \quad (1.8)$$

For any line segment u and any finite positive integer i , there are at most two vertices v and w such that $\text{index}(u, v)$ and $\text{index}(u, w)$ equals i . This is because the extension of the segment u along either of the two possible directions will meet any other line segment at most once. Thus, in each of the two directions, there is a total ordering on the points of intersection with other segments and the index values increase monotonically. This implies that

$$\sum_{v \neq u} \frac{1}{\text{index}(u, v) + 1} \leq \sum_{i=1}^{n-1} \frac{2}{i+1}.$$

Combining this with (1.8) implies that the expected size of P_π is bounded above by

$$n + 2 \sum_u \sum_{i=1}^{n-1} \frac{1}{i+1} \leq n + 2nH_n,$$

which is $O(n \log n)$. \square

Note that in computing the expected number of intersections, we only made use of linearity of expectation. We do not require any independence between the events $u \rightarrow v$ and $u \rightarrow w$, for segments u, v , and w . Indeed, these events need not be independent in general.

One way of interpreting Theorem 1.2 is as follows: since the expected size of the binary planar partition constructed by the algorithm is $O(n \log n)$ on any input, there *must exist* a binary autopartition of size $O(n \log n)$ for every input. This follows from the simple fact that any random variable assumes at least one value that is no greater than its expectation (and, indeed, one that is no less than its expectation). Thus we have used a probabilistic argument to assert that a combinatorial object – in this case a binary autopartition of size $O(n \log n)$ – exists with *absolute certainty* rather than with some probability. This is an example of the *probabilistic method in combinatorics*. We will study the probabilistic method in greater detail in Chapter 5.

1.4. A Probabilistic Recurrence

Frequently, we express a random variable of interest as a recurrence in terms of other random variables. In this section, we study one such situation using the **Find** algorithm analyzed in detail in Problem 1.9. The material in this section, although useful, is not an essential prerequisite for subsequent topics and may be omitted in the first reading.

The **Find** algorithm for selecting the k th smallest of a set S of n elements works as follows. We pick a random element y and partition $S \setminus \{y\}$ into two sets S_1 and S_2 (elements smaller and larger than y respectively) as in **RandQS**. Suppose $|S_1| = k - 1$; then y is the desired element and we are done. Otherwise, if $|S_1| \geq k$, we recursively find the k th smallest element of S_1 ; else we recursively find the $(k - |S_1| - 1)$ th smallest element in S_2 .

The expected number of comparisons made by the **Find** algorithm is the subject of Problem 1.9. Suppose instead that we were to ask the following question: what is the expected number of times we make the recursive call in the algorithm? Equivalently, what is the expected number of times we pick a random element in the algorithm? While this question may not be especially important for the **Find** algorithm, it is the kind of question that arises in the analysis of a number of parallel and geometric algorithms. Intuitively, we expect that the size of the residual problem in the **Find** algorithm is divided by a constant factor at each recursive level, so that we expect that the number of recursive invocations is $O(\log n)$. Below, we show that this intuition can be formalized in a general setting.

Let $g(x)$ be a monotone non-decreasing function from the positive reals to the positive reals. Consider a particle whose position changes at discrete time steps and is always at a positive integer. If the particle is currently at position $m > 1$, it proceeds at the next step to the position $m - X$, where X is a random variable ranging over the integers $1, \dots, m - 1$. All we know about X is that $\mathbf{E}[X] \geq g(m)$, and that X is chosen independently of the past. It is clear that the particle will always reach position 1 and the process terminates in that state. The interesting question is, assuming that the particle starts at position n , what is the expected number of steps before it reaches position 1? The reader may associate the position of the particle with the size of the problem in a recursive call of the **Find** algorithm. Although we have more information about the distribution of X in the case of **Find**'s analysis, it turns out that the bound on the expected size of the residual problem suffices for proving the following result.

Theorem 1.3: Let T be the random variable denoting the number of steps in which the particle reaches the position 1. Then, $\mathbf{E}[T] \leq \int_1^n dx/g(x)$.

PROOF: The proof is by induction on n ; let us suppose the theorem holds for values of m smaller than n . Let $f(m) = \int_1^m dx/g(x)$ for $m \geq 1$. We wish to show that $\mathbf{E}[T] \leq f(n)$.

Consider the first step, during which the particle proceeds from position n to position $n - X$, where X is chosen from a distribution for which $\mathbf{E}[X] \geq g(n)$. We have

$$\mathbf{E}[T] \leq 1 + \mathbf{E}[f(n - X)] \quad (1.9)$$

$$= 1 + \mathbf{E}\left[\int_1^n \frac{dy}{g(y)} - \int_{n-X}^n \frac{dy}{g(y)}\right] \quad (1.10)$$

$$= 1 + f(n) - \mathbf{E}\left[\int_{n-X}^n \frac{dy}{g(y)}\right] \quad (1.11)$$

$$\leq 1 + f(n) - \mathbf{E}\left[\int_{n-X}^n \frac{dy}{g(n)}\right] \quad (1.12)$$

$$= 1 + f(n) - \frac{\mathbf{E}[X]}{g(n)} \quad (1.13)$$

$$\leq f(n). \quad (1.14)$$

The inequality (1.12) follows from the assumption that $g(y)$ is non-decreasing, while (1.14) follows from the lower bound on $\mathbf{E}[X]$. \square

Exercise 1.6: If X were to range over all integers having value at most $m-1$ (possibly including negative integers), how would the statement and proof of Theorem 1.3 change?

For the **Find** algorithm, we can show (following the analysis of Problem 1.9) that $g(m) \geq m/4$. We may then apply the above theorem to bound the expected number of recursive calls to **Find** by $4 \ln n$.

Exercise 1.7: What prevents us from using Theorem 1.3 to bound the expected number of levels of recursion in the **RandQS** algorithm?

1.5. Computation Model and Complexity Classes

In this section we discuss models of computation used in this book, and follow this with a review of complexity classes.

1.5.1. RAMs and Turing Machines

Following common practice, throughout this book we use the *Turing machine* model to discuss complexity-theory issues. As is common, however, we switch to the *RAM* (random access machine) as the model of computation when describing and analyzing algorithms (except in the study of parallel and distributed algorithms in Chapter 12, where we define a version of the RAM model for

machines working in parallel). We begin by defining the Turing machine, which is an abstract model of an algorithm.

► **Definition 1.2:** A *deterministic Turing machine* is a quadruple $M = (S, \Sigma, \delta, s)$. Here S is a finite set of states, of which $s \in S$ is the machine's *initial state*. The machine uses a finite set of symbols, denoted Σ ; this set includes special symbols **BLANK** and **FIRST**. The function δ is the *transition function* of the Turing machine, mapping $S \times \Sigma$ to $(S \cup \{\text{HALT}, \text{YES}, \text{NO}\}) \times \Sigma \times \{\leftarrow, \rightarrow, \text{STAY}\}$. The machine has three halting states **HALT** (the *halting state*), **YES** (the *accepting state*), and **NO** (the *rejecting state*) (these are states, but formally not in S).

The input to the Turing machine is generally thought of as being written on a *tape*; unless otherwise specified, the machine may read from and write on this tape. We assume that **HALT**, **YES**, and **NO**, as well as the symbols \leftarrow , \rightarrow , and **STAY**, are not in $S \cup \Sigma$. The machine begins in the initial state s with its *cursor* at the first symbol of the input x (i.e., the left end of the tape); this symbol is always **FIRST**. The rest of the input is a string of finite length from $(\Sigma \setminus \{\text{BLANK}, \text{FIRST}\})^*$; the left-most **BLANK** on the tape identifies the end of the input string.

The transition function dictates the actions of the machine, and may be thought of as its *program*. In each step, the machine reads the symbol α of the input currently pointed to by the cursor; based on this symbol and the current state of the machine, it chooses a next state, a symbol β to be overwritten on α and a cursor motion direction from $\{\leftarrow, \rightarrow, \text{STAY}\}$ (here \leftarrow and \rightarrow specify a motion by one step to the left and right, respectively, while **STAY** specifies that the cursor remain in its present position). The transition function is designed to ensure that the cursor never falls off the left end of the input, identified by **FIRST**. The machine may of course overwrite the **BLANK** symbol.

If the machine halts in the **YES** state, we say that it has *accepted* the input x . If the machine halts in the **NO** state, we say that it has *rejected* the input x . The third halting state, **HALT**, is for the computation of functions whose range is not Boolean; in such cases, the output of the function computation is written onto the tape. An algorithm corresponds to a Turing machine that always halts.

A *probabilistic Turing machine* is a Turing machine augmented with the ability to generate an unbiased coin flip in one step. It corresponds to a randomized algorithm. On any input x , a probabilistic Turing machine accepts x with some probability, and we study this probability.

In the light of these definitions, we may speak of an algorithm accepting or rejecting an input (we visualize the Turing machine underlying the algorithm as accepting or rejecting), and similarly speak of a randomized algorithm accepting or rejecting an input with some probability.

In the RAM model, we have a machine that can perform the following types of operations involving registers and main memory: input-output operations, memory-register transfers, indirect addressing, branching, and arithmetic operations. Each register or memory location may hold an integer that can be accessed as a unit, but an algorithm has no access to the representation of the number.

The arithmetic instructions permitted are $+$, $-$, \times , $/$. In addition, an algorithm can compare two numbers, and evaluate the square root of a positive number.

Two types of RAM models are defined based on the cost used for measuring the running time of a program. In the *unit-cost RAM* (sometimes also called the *uniform RAM*), each instruction can be performed in one time step. This model is believed to be much too powerful since there is no known polynomial-time simulation of this model by Turing machines. This situation arises because the unit-cost RAM, unlike the more restricted Turing machine, is able to use multiplication to quickly compute extremely large integers. However, if we disallow all arithmetic operations besides addition and subtraction, then it is possible to show that the resulting model is equivalent to Turing machines under polynomial-time simulations.

A more realistic version of the RAM is the so-called *log-cost RAM* where each instruction requires time proportional to the logarithm of the size of its operands. It turns out that the log-cost RAM with the complete arithmetic instruction set is equivalent to Turing machines under polynomial-time simulations.

For simplicity, we will work with the general unit-cost RAM model. At the same time, we will avoid misuse of its power by ensuring that in all algorithms under consideration the size of the operands is polynomially bounded in the input size. Thus, our algorithm can be transformed to the log-cost RAM model with only a small (logarithmic in the input size) multiplicative slow-down in the running time. We also assume that the RAM can in a single step choose an element uniformly at random from a set of cardinality polynomial in the size of the problem input. Standard texts on automata and complexity (see the Notes section) give proofs of the following basic fact.

Proposition 1.4: *Any Turing machine computation of length polynomial in the size of the input can be simulated by a RAM computation of length polynomial in the size of the input. Any RAM computation of length polynomial in the size of the input can be simulated by a Turing machine computation of length polynomial in the size of the input.*

1.5.2. Complexity Classes

We now define some basic complexity classes focusing on those involving randomized algorithms. For these definitions, the underlying model of computation is assumed to be the Turing machine, but by the preceding discussion it could be substituted by a log-cost RAM or the restricted form of the unit-cost RAM.

In complexity theory, it is common to concentrate on the decision problem derived from some hard optimization problem. This enables the development of an elegant theoretical framework, and the decision problem is usually not significantly different in structure from its optimization counterpart. For instance, consider the *satisfiability* problem, in which an instance consists of a set of clauses in conjunctive normal form (CNF). Because the satisfiability problem appears at various points in this book, we define some terminology relating

to it. The Boolean inputs are called *variables*, which may appear in either uncomplemented or complemented form in a clause. The uncomplemented or complemented variables in a clause are known as *literals* (respectively, *unnegated* and *negated* literals). A clause is said to be satisfied if at least one of the literals in it is TRUE. A solution consists either of an assignment of Boolean values to the variables that ensures that every clause is satisfied (such an assignment is known as a *truth assignment*), or a negative answer that it is not possible to assign inputs so as to satisfy all the clauses simultaneously. The decision version of this problem, commonly abbreviated SAT, seeks only a YES or NO answer depending on whether or not all the clauses can simultaneously be satisfied, without demanding an assignment of values to the inputs (in case the answer is YES).

► **Example 1.2:** Consider the following instance of satisfiability:

$$(x_1 \vee \bar{x}_2 \vee x_4) \wedge (\bar{x}_3 \vee \bar{x}_4 \vee x_5) \wedge (\bar{x}_1 \vee x_2 \vee x_4 \vee \bar{x}_5).$$

In this example, there are three clauses. The first stipulates that either x_1 should be TRUE, or x_2 should be FALSE, or x_4 should be TRUE. The literal \bar{x}_2 denotes that one way of satisfying the first clause is to set x_2 FALSE. The first two clauses have three literals each, while the third has four. The assignments $x_1 = \text{TRUE}$, $x_3 = \text{FALSE}$, and $x_5 = \text{FALSE}$ suffice to satisfy all the clauses (regardless of the values assigned to x_2 and x_4). Thus the solution to this instance for the decision question (SAT) is YES.

Any decision problem can be treated as a language recognition problem. Fix a finite alphabet Σ , usually $\Sigma = \{0, 1\}$, and let Σ^* be the set of all possible strings over this alphabet. Denote by $|s|$ the length of a string s . A *language* $L \subseteq \Sigma^*$ is any collection of strings over Σ . The corresponding *language recognition* problem is to decide whether a given string x in Σ^* belongs to L . An algorithm solves a language recognition problem for a specific language L by *accepting* (output YES) any input string contained in L , and *rejecting* (output NO) any input string not contained in L . The SAT problem can easily be cast in the form of a language recognition problem by devising a suitable encoding of formulas as bit-strings.

A complexity class is a collection of languages all of whose recognition problems can be solved under prescribed bounds on the computational resources. We are primarily interested in various forms of efficient algorithms, where efficient is defined as being *polynomial time*. Recall that an algorithm has polynomial running time if it halts within $n^{O(1)}$ time on any input of length n . The following definitions list some interesting complexity classes.

► **Definition 1.3:** The class P consists of all languages L that have a polynomial-time algorithm A such that for any input $x \in \Sigma^*$,

- $x \in L \Rightarrow A(x)$ accepts.
- $x \notin L \Rightarrow A(x)$ rejects.

► **Definition 1.4:** The class NP consists of all languages L that have a polynomial-time algorithm A such that for any input $x \in \Sigma^*$,

- $x \in L \Rightarrow \exists y \in \Sigma^*, A(x, y)$ accepts, where $|y|$ is bounded by a polynomial in $|x|$.
- $x \notin L \Rightarrow \forall y \in \Sigma^*, A(x, y)$ rejects.

A useful view of P and NP is the following. The class P consists of all languages L such that for any x in L a proof of the membership x in L (represented by the string y) can be *found* and *verified* efficiently. On the other hand, NP consists of all languages L such that for any x in L , a proof of the membership of x in L can be *verified* efficiently. Obviously, $P \subseteq NP$, but it is not known whether $P = NP$. If $P = NP$, the existence of an efficiently verifiable proof implies that it is possible to actually find such a proof efficiently.

For any complexity class C , we define the complementary class $\text{co-}C$ as the set of languages whose complement is in the class C . That is,

$$\text{co-}C = \{L \mid \bar{L} \in C\}.$$

It is obvious that $P = \text{co-}P$ and $P \subseteq NP \cap \text{co-}NP$. We do not know whether $P = NP \cap \text{co-}NP$ or whether $NP = \text{co-}NP$, although both statements are widely believed to be false.

Likewise, we can define deterministic and non-deterministic complexity classes for different bounds on the running time. Let *exponential time* denote a running time which is $2^{p(n)}$ for some polynomial $p(n)$ in the input size. Allowing exponential time instead of polynomial time in Definitions 1.3 and 1.4 gives us the complexity classes EXP and $NEXP$. Clearly, $EXP \subseteq NEXP$, but once again we do not know whether this inclusion is strict. On the other hand, we do know that if $P = NP$, then $EXP = NEXP$.

We can also define *space* complexity classes by leaving the running time unconstrained and instead placing a bound on the space used by an algorithm. In the case of Turing machines, the space used is determined by the number of distinct positions on the tape that are scanned during an execution; for RAMs, the space requirement is simply the number of words of memory required by an algorithm. In Definitions 1.3 and 1.4, requiring polynomial space instead of polynomial time yields the definition of the class $PSPACE$ and $NPSPACE$. A $PSPACE$ algorithm may run for super-polynomial time. These classes behave differently from the time complexity classes; for example, we know that $PSPACE = NPSPACE$ and $PSPACE = \text{co-}PSPACE$.

We next review the notions of polynomial reductions and completeness for a complexity class.

► **Definition 1.5:** A *polynomial reduction* from a language $L_1 \subseteq \Sigma^*$ to a language $L_2 \subseteq \Sigma^*$ is a function $f : \Sigma^* \rightarrow \Sigma^*$ such that:

1. There is a polynomial-time algorithm that computes f .
2. For all $x \in \Sigma^*$, $x \in L_1$ if and only if $f(x) \in L_2$.

Exercise 1.8: Show that if there is a polynomial reduction from L_1 to L_2 , then $L_2 \in P$ implies that $L_1 \in P$.

► **Definition 1.6:** A language L is *NP-hard* if, for all $L' \in NP$, there is a polynomial reduction from L' to L .

Thus, if any NP -hard decision problem can be solved in polynomial time, then so can all problems in NP .

► **Definition 1.7:** A language L is *NP-complete* if it is in NP and is NP -hard.

Intuitively the decision problems corresponding to NP -complete languages are the “hardest” problems in NP . Note that the notion of NP -completeness applies only to decision problems; the optimization problem corresponding to an NP -complete decision problem is NP -hard, but is not NP -complete because it is not in NP by definition. As with NP , the notions of hardness and completeness can be generalized to any class C , for an appropriate notion of reduction. Unless otherwise specified, the default notion of a reduction is a polynomial reduction, and this is typically used for defining hardness and completeness in complexity classes that are a superset of P , such as $PSPACE$.

We generalize these classes to allow for randomized algorithms. The basic idea is to replace the existential and universal quantifiers in the definition of NP by probabilistic requirements.

► **Definition 1.8:** The class RP (for Randomized Polynomial time) consists of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input x in Σ^* ,

- $x \in L \Rightarrow \Pr[A(x) \text{ accepts}] \geq \frac{1}{2}$.
- $x \notin L \Rightarrow \Pr[A(x) \text{ accepts}] = 0$.

The choice of the bound on the error probability $1/2$ is arbitrary. In fact, as was observed in the case of the min-cut algorithm, independent repetitions of the algorithm can be used to go from the case where the probability of success is *polynomially small* to the case where the probability of error is *exponentially small* while changing only the degree of the polynomial that bounds the running time. Thus, the success probability can be changed to an inverse polynomial function of the input size without significantly affecting the definition of RP .

Observe that an RP algorithm is a Monte Carlo algorithm that can err only when $x \in L$. This is referred to as *one-sided error*. The class $\text{co-}RP$ consists of languages that have polynomial-time randomized algorithms erring only in the

case when $x \notin L$. A problem belonging to both **RP** and **co-RP** can be solved by a randomized algorithm with *zero-sided error*, i.e., a Las Vegas algorithm.

► **Definition 1.9:** The class **ZPP** (for Zero-error Probabilistic Polynomial time) is the class of languages that have Las Vegas algorithms running in expected polynomial time.

Exercise 1.9: Show that $\mathbf{ZPP} = \mathbf{RP} \cap \mathbf{co-RP}$.

Consider now the class of problems that have randomized Monte Carlo algorithms making *two-sided errors*.

► **Definition 1.10:** The class **PP** (for Probabilistic Polynomial time) consists of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input x in Σ^* ,

- $x \in L \Rightarrow \Pr[A(x) \text{ accepts}] > \frac{1}{2}$.
- $x \notin L \Rightarrow \Pr[A(x) \text{ accepts}] < \frac{1}{2}$.

To reduce the error probability of a two-sided error algorithm, we can perform several independent iterations on the same input and produce the output (accept or reject) that occurs in the majority of these iterations. Unfortunately, the definition of the class **PP** is rather weak: because we have no bound on how far from $1/2$ the probabilities are, it may not be possible to use a small number of repetitions of an algorithm A with such two-sided error probability to obtain an algorithm with significantly smaller error probability.

Exercise 1.10: Consider a randomized algorithm with two-sided error probabilities as in the definition of **PP**. Show that a polynomial number of independent repetitions of this algorithm need not suffice to reduce the error probability to $1/4$. (Consider the case where the error probability is $1/2 - 1/2^n$.)

A more useful class of two-sided error randomized algorithms corresponds to the following complexity class.

► **Definition 1.11:** The class **BPP** (for Bounded-error Probabilistic Polynomial time) consists of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input x in Σ^* ,

- $x \in L \Rightarrow \Pr[A(x) \text{ accepts}] \geq \frac{3}{4}$.
- $x \notin L \Rightarrow \Pr[A(x) \text{ accepts}] \leq \frac{1}{4}$.

In a later chapter (see Problem 4.8) we will show that for this class of algorithms the error probability can be reduced to $1/2^n$ with only a polynomial number of iterations. In fact, the probability bounds $3/4$ and $1/4$ can be changed to $1/2 + 1/p(n)$ and $1/2 - 1/p(n)$, respectively, for any polynomially bounded function $p(n)$ without affecting this error reduction property or the definition of the class **BPP** to a significant extent.

The reader is referred to Problems 1.11–1.14 for several basic relationships between these complexity classes. There are several interesting open questions regarding the relationships between these randomized complexity classes, for example:

1. Is $\mathbf{RP} = \mathbf{co-RP}$?
2. Is $\mathbf{RP} \subseteq \mathbf{NP} \cap \mathbf{co-NP}$? (Note that since $\mathbf{co-RP} \subseteq \mathbf{co-NP}$, showing that $\mathbf{RP} = \mathbf{co-RP}$ would imply $\mathbf{RP} \subseteq \mathbf{NP} \cap \mathbf{co-NP}$.)
3. Is $\mathbf{BPP} \subseteq \mathbf{NP}$?

Although these classes are defined in terms of decision problems, they can be used to classify the complexity of a broader class of problems such as search or optimization problems. We will overload our notation a bit by using the complexity class labels for referring to algorithms. For example, **RandQS** will be called a **ZPP** algorithm.

Consider the following decision version of the min-cut problem: given a graph G and integer K , verify that the min-cut size in G equals K . Assume that we have modified (by incorporating sufficiently many repetitions) the Monte Carlo min-cut algorithm to reduce its probability of error below $1/4$. This algorithm can solve the decision problem by computing a cut value k and comparing it with K . This gives a **BPP** algorithm. In the case where K is indeed the min-cut value, the algorithm may not come up with the right value and, hence, may reject the input. Conversely, if the min-cut value is smaller than K , the algorithm may only find cuts of size K and, hence, may accept the input.

We may modify this decision problem: given G and K , verify that the min-cut size in G is *at most* K . Now, the algorithm described above translates into an **RP** algorithm for this problem. In the case where the actual min-cut size C is larger than K , the algorithm will never accept the input. This is because it can only find cuts of size k no smaller than C and hence greater than K .

Notes

The ideas underlying randomized algorithms can be traced back to *Monte Carlo methods* used in numerical analysis, statistical physics, and simulation. In the context of computability theory, the notion of a probabilistic Turing machine was proposed by de Leeuw, Moore, Shannon, and Shapiro [122] and further explored in the pioneering work of Rabin [340] and Gill [166]. Berlekamp [57], Rabin [341], and Solovay and Strassen [382] gave early examples of concrete randomized algorithms. Rabin [341] proposed randomized algorithms for problems in computational geometry and in number theory. Around the same time, Solovay and Strassen [382] gave a randomized Monte

Carlo algorithm for testing for primality; this problem is explored further in Chapter 14, as is the randomized algorithm for factoring polynomials due to Berlekamp [57].

In the last twenty years, the array of techniques for devising and analyzing randomized algorithms has grown. We develop these techniques in the chapters to follow. Karp [243], Maffioli, Speranza, and Vercellis [289], and Welsh [415] give excellent surveys of randomized algorithms. Johnson [220] surveys the probabilistic (or “average-case”) analysis of algorithms (sometimes also referred to as “distributional complexity”), contrasting it with randomized algorithms surveyed in his following bulletin [221].

Our **RandQS** algorithm is based on Hoare’s algorithm [201]. The min-cut algorithm of Section 1.1, together with many variations and extensions, is due to Karger [231].

Monte Carlo methods have been popular in the sciences for over a hundred years now. The classic experiment on approximating the value of π by dropping needles on a sheet of paper with parallel lines is described in an eighteenth-century paper by Buffon [86] (see also Hall [190]). The origin of the modern theory of Monte Carlo methods in the physical sciences is widely attributed to Ulam, von Neumann, and Fermi [116]. The term *Las Vegas algorithm* was introduced by Babai [37], although he uses the term in a slightly different sense. Our usage conforms to the currently accepted notion of a Las Vegas algorithm.

An important issue, alluded to in the discussion following the analysis of **RandQS** but otherwise not covered in detail in this book, is the generation of random samples from various types of distributions. First, there is the question of generating randomness within the inherently deterministic computers that will implement our randomized algorithms. This leads into the area of pseudo-random number generation, which is surveyed in the article by Boppana and Hirschfeld [73] and in Knuth’s book [259]. Even if we assume that a source of truly random bits is available, there is the issue of converting this into the various types of distributions that may be required in randomized algorithms (for example, see Problems 1.2 and 1.3). This problem is studied in the context of Monte Carlo simulations, for example in the work of von Neumann [409, 410], and Knuth [259] covers this in great detail. A comprehensive study of this important family of problems in terms of its computational complexity was undertaken by Knuth and Yao [264]. The complexity of random sampling of combinatorial structures, such as graphs with specified properties, has been studied by Pruhs and Manber [338]; as discussed in Chapter 11, the problem of counting the number of combinatorial structures with specified properties, often a difficult computational problem, can sometimes be reduced to random sampling.

The idea of using independent iterations to reduce the error probability of Monte Carlo algorithms has an analog for Las Vegas algorithms. Alt, Guibas, Mehlhorn, Karp, and Wigderson [25] study the possibility of reducing the probability that the running time of a Las Vegas algorithm substantially exceeds its expected value by employing the following strategy: choose a sequence (T_i) and use independent iterations of the Las Vegas algorithm, aborting the i th iteration in T_i steps, until one of the iterations terminates successfully within the allotted time. These results were strengthened by Luby, Sinclair, and Zuckerman [286], who also considered the minimization of the expected total running time of such strategies.

The material of Section 1.3 is drawn from Paterson and Yao [329]. The **Find** algorithm described in Section 1.4 is due to Hoare [200]. Theorem 1.3 is given in a paper by Karp, Upfal and Wigderson [250]. Karp [244] gives a number of additional results on probabilistic recurrence relations.

The reader is referred to introductory texts on algorithms and complexity such as those by Aho, Hopcroft, and Ullman [5, 6] and Papadimitriou [326] for more details on the Turing machine model and the RAM model. It is known, for instance, that sorting n numbers requires $\Omega(n \log n)$ operations in the RAM model of computation. The books by Bovet and Crescenzi [81] and by Papadimitriou [326] contain a more detailed treatment of the complexity classes described in this chapter.

Problems

- 1.1** (Due to J. von Neumann [409].)
- (a) Suppose you are given a coin for which the probability of HEADS, say p , is *unknown*. How can you use this coin to generate unbiased (i.e., $\Pr[\text{HEADS}] = \Pr[\text{TAILS}] = 1/2$) coin-flips? Give a scheme for which the expected number of flips of the biased coin for extracting one unbiased coin-flip is no more than $1/[p(1-p)]$. (**Hint** : Consider two consecutive flips of the biased coin.)
- (b) Devise an extension of the scheme that extracts the largest possible number of independent, unbiased coin-flips from a given number of flips of the biased coin.
- 1.2** (Due to D.E. Knuth and A. C-C. Yao [264].)
- (a) Suppose you are provided with a source of unbiased random bits. Explain how you will use this to generate uniform samples from the set $S = \{0, \dots, n-1\}$. Determine the expected number of random bits required by your sampling algorithm.
- (b) What is the *worst-case* number of random bits required by your sampling algorithm? Consider the case when n is a power of 2, as well as the case when it is not.
- (c) Solve (a) and (b) when, instead of unbiased random bits, you are required to use as the source of randomness uniform random samples from the set $\{0, \dots, p-1\}$; consider the case when n is a power of p , as well as the case when it is not.
- 1.3** (Due to D.E. Knuth and A. C-C. Yao [264].) Suppose you are provided with a source of unbiased random bits. Provide efficient (in terms of expected running time and expected number of random bits used) schemes for generating samples from the distribution over the set $\{2, 3, \dots, 12\}$ induced by rolling two unbiased dice and taking the sum of their outcomes.
- 1.4**
- (a) Suppose you are required to generate a random permutation of size n . Assuming that you have access to a source of independent and unbiased random bits, suggest a method for generating random permutations of size n . Efficiency is measured in terms of both time and number of random bits. What lower bounds can you prove for this task?
- (b) Consider the following method for generating a random permutation of size n . Pick n random values X_1, \dots, X_n independently from the uniform distribution over the interval $[0, 1]$. Now, the permutation that orders the

random variables in ascending order is claimed to be a random permutation, and it can be determined by sorting the random values. Is the claim correct? How efficient is this scheme?

(c) Consider the following “lazy” implementation of the scheme suggested in (b). The binary representation of the fraction X_i is a sequence of unbiased and independent random bits. At any given stage of the sorting algorithm, we would have chosen only as many bits of each X_i as necessary to resolve all the comparisons performed up to that point. When comparing X_i to X_j , if the current prefixes of their binary expansions do not determine the outcome of the comparisons, then we extend their prefixes by choosing further random bits until this happens. Compute tight bounds on the *expected* number of random bits used by this implementation.

- 1.5** Consider the problem of using a source of unbiased random bits to generate samples from the set $S = \{0, \dots, n-1\}$ such that the element i is chosen with probability p_i . Show how to perform this sampling using $O(\log n)$ random bits per sample, regardless of the values of p_i . Use the result from part (c) of Problem 1.4.
- 1.6** Consider a sequence of n flips of an unbiased coin. Let H_i denote the absolute value of the excess of the number of HEADS over the number of TAILS seen in the first i flips. Define $H = \max_i H_i$. Show that $E[H_i] = \Theta(\sqrt{i})$, and that $E[H] = \Theta(\sqrt{n})$.
- 1.7** Suppose we choose a permutation π of the ordered set $N = \{1, 2, \dots, n\}$ uniformly at random from the space of all permutations of N . Let $L(\pi)$ denote the length of the longest increasing subsequence in permutation π .
- (a) For large n and some positive constant c , prove that $E[L(\pi)] \geq c\sqrt{n}$.
- (b) Is the bound in (a) tight?
- 1.8** Consider adapting the min-cut algorithm of Section 1.1 to the problem of finding an s - t min-cut in an undirected graph. In this problem, we are given an undirected graph G together with two distinguished vertices s and t . An s - t cut is a set of edges whose removal from G disconnects s from t ; we seek an s - t cut of minimum cardinality. As the algorithm proceeds, the vertex s may get amalgamated into a new vertex as a result of an edge being contracted; we call this vertex the s -vertex (initially the s -vertex is s itself). Similarly, we have a t -vertex. As we run the contraction algorithm, we ensure that we never contract an edge between the s -vertex and the t -vertex.
- (a) Show that there are graphs in which the probability that this algorithm finds an s - t min-cut is exponentially small.
- (b) How large can the number of s - t min-cuts in an instance be?
- 1.9** Consider the **Find** algorithm described in Section 1.4 for selecting the k th smallest of a set S of n elements. Show that the algorithm finds the k th smallest element in S in expected time $O(n)$.
- 1.10** Consider the setting of Example 1.1. Show that the probability that no sailor returns to her own cabin approaches $1/e$ as the number of sailors grows large.

- 1.11** Verify the following inclusions:

$$P \subseteq RP \subseteq NP \subseteq PSPACE \subseteq EXP \subseteq NEXP.$$

It is not known whether these inclusions are strict.

- 1.12** Verify the following inclusions:

$$RP \subseteq BPP \subseteq PP.$$

It is not known whether these inclusions are strict.

- 1.13** Show that $PP = \text{co-}PP$ and $BPP = \text{co-}BPP$.

- 1.14** Show that $NP \subseteq PP \subseteq PSPACE$.

- 1.15** (Due to K.-I. Ko [265].) Show that $NP \subseteq BPP$ implies $NP = RP$.