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

$$T(n) \leq 2T(n/2) + (c+1)n, \tag{1.1}$$

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 ith 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 $\sum_{i=1}^{n} \sum_{j>i} X_{ij}$. We are interested in the expected number of comparisons, which is clearly

$$\mathbf{E}[\sum_{i=1}^{n} \sum_{j>i} X_{ij}] = \sum_{i=1}^{n} \sum_{j>i} \mathbf{E}[X_{ij}]. \tag{1.2}$$

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,

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

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 ith 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 $S_{(i)}, S_{(i+1)}, \ldots, S_{(j)}$ 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

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

It follows that the expected number of comparisons is bounded above by $2nH_n$, where H_n is the *nth 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 $S_{(1)}, \ldots, S_{(n)}$?

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

Cambridge University Press
978-0-521-47465-8 - Randomized Algorithms
Rajeev Motwani and Prabhakar Raghavan
Excerpt
More information

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

$$\mathbf{Pr}[\mathcal{E}_1 \cap \mathcal{E}_2] = \mathbf{Pr}[\mathcal{E}_1] \times \mathbf{Pr}[\mathcal{E}_2] \tag{1.4}$$

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

$$\mathbf{Pr}[\mathcal{E}_1 \cap \mathcal{E}_2] = \mathbf{Pr}[\mathcal{E}_1 \mid \mathcal{E}_2] \times \mathbf{Pr}[\mathcal{E}_2] = \mathbf{Pr}[\mathcal{E}_2 \mid \mathcal{E}_1] \times \mathbf{Pr}[\mathcal{E}_1], \tag{1.5}$$

where $\mathbf{Pr}[\mathcal{E}_1 \mid \mathcal{E}_2]$ 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).

$$\mathbf{Pr}[\cap_{i=1}^{k}\mathcal{E}_i] = \mathbf{Pr}[\mathcal{E}_1] \times \mathbf{Pr}[\mathcal{E}_2 \mid \mathcal{E}_1] \times \mathbf{Pr}[\mathcal{E}_3 \mid \mathcal{E}_1 \cap \mathcal{E}_2] \cdots \mathbf{Pr}[\mathcal{E}_k \mid \cap_{i=1}^{k-1}\mathcal{E}_i]. \tag{1.6}$$

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.

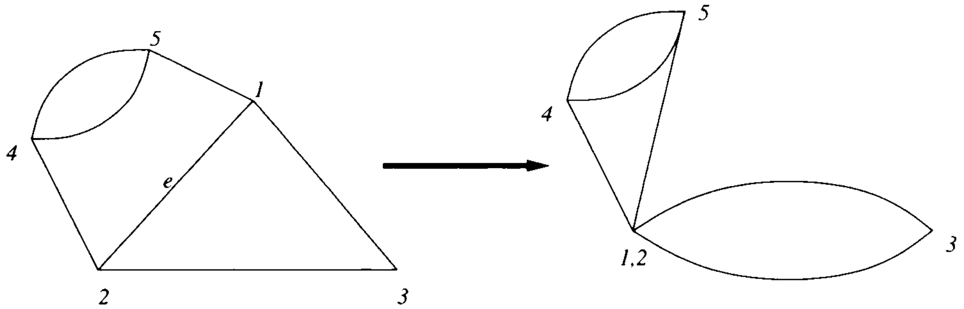

Figure 1.1: A step in the min-cut algorithm; the effect of contracting edge $e = (1, 2)$ is shown.

► **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 ith 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 $\mathbf{Pr}[\mathcal{E}_1] \geq 1 - 2/n$. 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 $\mathbf{Pr}[\mathcal{E}_2 \mid \mathcal{E}_1] \geq 1 - 2/(n-1)$. At the ith 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, $\mathbf{Pr}[\mathcal{E}_i \mid \cap_{j=1}^{i-1} \mathcal{E}_j] \geq 1 - 2/(n-i+1)$. What is the probability that no edge of C is ever picked in the process? We invoke (1.6) to obtain

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

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 $X_1, X_2, \ldots$,

$$\mathbf{E}[\sum_i X_i] = \sum_i \mathbf{E}[X_i]. \tag{1.7}$$

(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 ith 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 ith 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 $S = \{s_1, s_2, \ldots, s_n\}$ 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 $s_{i1}, s_{i2}, \ldots$, 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