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 *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 $\sum_{i=1}^{n}\sum_{j>i} X_{ij}$. 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

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

It follows that the expected number of comparisons is bounded above by $2nH_n$, where H_n is the *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 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.

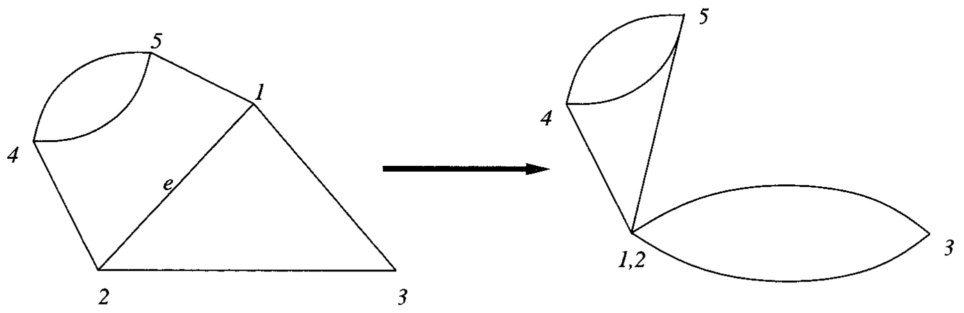

Figure 1.1: A step in the min-cut algorithm; the effect of contracting edge $e = (1,2)$ is shown.

► **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? $\mathcal{E}_1^x = \mathcal{E}_2$

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 $X_1, X_2, \ldots$,

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

CHAPTER 3

Moments and Deviations

In Chapters 1 and 2, we bounded the expected running times of several randomized algorithms. While the expectation of a random variable (such as a running time) may be small, it may frequently assume values that are far higher. In analyzing the performance of a randomized algorithm, we often like to show that the behavior of the algorithm is good almost all the time. For example, it is more desirable to show that the running time is small with high probability, not just that it has a small expectation. In this chapter we will begin the study of general methods for proving statements of this type. We will begin by examining a family of stochastic processes that is fundamental to the analysis of many randomized algorithms: these are called *occupancy problems*. This motivates the study (in this chapter and the next) of general bounds on the probability that a random variable deviates far from its expectation, enabling us to avoid such custom-made analyses. The probability that a random variable deviates by a given amount from its expectation is referred to as a *tail probability* for that deviation. Readers wishing to review basic material on probability and distributions may consult Appendix C.

3.1. Occupancy Problems

We begin with an example of an *occupancy problem*. In such problems we envision each of m indistinguishable objects ("balls") being randomly assigned to one of n distinct classes ("bins"). In other words, each ball is placed in a bin chosen independently and uniformly at random. We are interested in questions such as: what is the maximum number of balls in any bin? what is the expected number of bins with k balls in them? Such problems are at the core of the analyses of many randomized algorithms ranging from data structures to routing in parallel computers. Later, in Section 3.6, we will encounter a variant of the occupancy problem, known as the *coupon collector's problem*; in

Chapter 4, we will apply sophisticated techniques to various random variables arising in occupancy problems.

Our discussion of the occupancy problem will illustrate a recurrent tool in the analysis of randomized algorithms: that *the probability of the union of events is no more than the sum of their probabilities.* This is a special case of the Boole-Bonferroni Inequalities (Proposition C.2) and can be formally stated as follows: for arbitrary events $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$, not necessarily independent,

$$\mathbf{Pr}[\cup_{i=1}^{n}\mathcal{E}_i] \leq \sum_{i=1}^{n} \mathbf{Pr}[\mathcal{E}_i].$$

This principle is extremely useful because it assumes nothing about the dependencies between the events. Thus, it enables us to analyze phenomena involving events with very complicated interactions, without having to unravel the interactions.

Consider first the case $m = n$. For $1 \leq i \leq n$, let X_i be the number of balls in the ith bin. Following Example 1.1, we have $\mathbf{E}[X_i] = 1$ for all i. Yet we do not expect that during a typical experiment every bin receives exactly one ball. Rather, we expect some bins to have no balls at all, and others to have many more than one.

Let us try now to make a statement of the form "with very high probability, no bin receives more than k balls," for a suitably chosen k. Let $\mathcal{E}_j(k)$ denote the event that bin j has k or more balls in it. We concentrate on analyzing $\mathcal{E}_1(k)$. The probability that bin 1 receives exactly i balls is

$$\binom{n}{i}\left(\frac{1}{n}\right)^i\left(1-\frac{1}{n}\right)^{n-i} \leq \binom{n}{i}\left(\frac{1}{n}\right)^i \leq \left(\frac{ne}{i}\right)^i\left(\frac{1}{n}\right)^i = \left(\frac{e}{i}\right)^i.$$

The second inequality results from an upper bound for binomial coefficients (Proposition B.2). Thus,

$$\mathbf{Pr}[\mathcal{E}_1(k)] \leq \sum_{i=k}^{n}\left(\frac{e}{i}\right)^i \leq \left(\frac{e}{k}\right)^k\left(1+\frac{e}{k}+\left(\frac{e}{k}\right)^2+\cdots\right). \tag{3.1}$$

Let $k^* = \lceil (3\ln n)/\ln\ln n \rceil$. Then,

$$\mathbf{Pr}[\mathcal{E}_1(k^*)] \leq \left(\frac{e}{k^*}\right)^{k^*}\frac{1}{1-e/k^*} \leq n^{-2}.$$

The same computation tells us that this upper bound applies to $\mathbf{Pr}[\mathcal{E}_i(k^*)]$ for all i, but can we say that *no bin* is likely to have more than k^* balls in it? For this we invoke the principle mentioned at the beginning of this section: the probability of the union of the events $\mathcal{E}_i(k^*)$ is no more than their sum. We obtain that

$$\mathbf{Pr}[\cup_{i=1}^{n}\mathcal{E}_i(k^*)] \leq \sum_{i=1}^{n}\mathbf{Pr}[\mathcal{E}_i(k^*)] \leq \frac{1}{n}.$$

Thus we have established:

Theorem 3.1: *With probability at least* $1 - 1/n$, *no bin has more than* $k^* = (e \ln n)/\ln\ln n$ *balls in it.*

Interestingly, when m is of the order of $n \log n$, the bin with the most balls has about the same number of balls as the expected number of balls in any bin. This phenomenon is exploited in a number of randomized algorithms (see, for instance, Section 4.2).

Exercise 3.1: For $m = n \log n$, show that with probability $1 - o(1)$ every bin contains $O(\log n)$ balls.

We turn to a classic combinatorial problem. Suppose that m balls are randomly assigned to n bins. We study the probability of the event that they all land in distinct bins. The special case $n = 365$ is popular in mathematical lore as the *birthday problem.* The interpretation is that the 365 days of the year correspond to 365 bins, and the birthday of each of m people is chosen independently and uniformly from all 365 days (ignoring leap years). How large must m be before two people in the group are likely to share their birthdays?

Consider the assignment of the balls to the bins as a sequential process: we throw the first ball into a random bin, then the second ball, and so on. For $2 \le i \le m$, let $\mathcal{E}_i$ denote the event that the ith ball lands in a bin not containing any of the first $i-1$ balls. We will bound $\mathbf{Pr}[\cap_{i=2}^m \mathcal{E}_i]$ from above. From (1.6), we can write

$$\mathbf{Pr}[\cap_{i=2}^m \mathcal{E}_i] = \mathbf{Pr}[\mathcal{E}_2]\mathbf{Pr}[\mathcal{E}_3 \mid \mathcal{E}_2]\mathbf{Pr}[\mathcal{E}_4 \mid \mathcal{E}_2 \cap \mathcal{E}_3] \cdots \mathbf{Pr}[\mathcal{E}_m \mid \cap_{i=2}^{m-1} \mathcal{E}_i].$$

Now, it is easy to compute $\mathbf{Pr}[\mathcal{E}_i \mid \cap_{j=2}^{i-1} \mathcal{E}_j]$: this is simply the probability that the ith ball lands in an empty bin given that the first $i-1$ all fell into distinct bins, and is thus $1 - (i-1)/n$. Making use of the fact that $1 - x \le e^{-x}$, we have

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

Thus, we see that for m equal to $\lceil \sqrt{2n} + 1 \rceil$, the probability that all m balls land in distinct bins is at most $1/e$; as m increases beyond this value, the probability drops rapidly.

3.2. The Markov and Chebyshev Inequalities

We have seen above that making statements about the probability that a random variable deviates far from its expectation may involve a detailed, problem-specific analysis. Often, one can avoid such detailed analyses by resorting to general inequalities on such tail probabilities.

We begin with the Markov inequality, a fundamental tool we will invoke repeatedly when we develop more sophisticated bounding techniques. Let X be a discrete random variable and $f(x)$ be any real-valued function. Then the expectation of $f(X)$ is given by (see Appendix C)

$$\mathbf{E}[f(X)] = \sum_x f(x)\mathbf{Pr}[X = x].$$

Theorem 3.2 (Markov Inequality): *Let Y be a random variable assuming only non-negative values. Then for all $t \in \mathbb{R}^+$,*

$$\mathbf{Pr}[Y \geq t] \leq \frac{\mathbf{E}[Y]}{t}.$$

Equivalently,

$$\mathbf{Pr}[Y \geq k\mathbf{E}[Y]] \leq \frac{1}{k}.$$

PROOF: Define a function $f(y)$ by $f(y) = 1$ if $y \geq t$, and 0 otherwise. Then $\mathbf{Pr}[Y \geq t] = \mathbf{E}[f(Y)]$. Since $f(y) \leq y/t$ for all y,

$$\mathbf{E}[f(Y)] \leq \mathbf{E}\left[\frac{Y}{t}\right] = \frac{\mathbf{E}[Y]}{t},$$

and the theorem follows. □

This is the tightest possible bound when we know only that Y is non-negative and has a given expectation. Unfortunately, the Markov inequality by itself is often too weak to yield useful results. The following exercise may help the reader appreciate this; it shows that the Markov inequality is tight only for rather uninteresting distributions.

Exercise 3.2: Given a positive integer k, describe a random variable X assuming only non-negative values, such that

$$\mathbf{Pr}[X \geq k\mathbf{E}[X]] = \frac{1}{k}.$$

The following generalization of Markov's inequality underlies its usefulness in deriving stronger bounds.

Exercise 3.3: Let Y be any random variable and h any non-negative real function. Show that for all $t \in \mathbb{R}^+$,

$$\mathbf{Pr}[h(Y) \geq t] \leq \frac{\mathbf{E}[h(Y)]}{t}.$$

We now show that the Markov inequality can be used to derive better bounds on the tail probability by using more information about the distribution of the random variable. The first of these is the Chebyshev bound, which is based on the knowledge of the variance of the distribution; we will apply this to the analysis of a simple randomized selection algorithm.

For a random variable X with expectation μ_X, its *variance* σ_X^2 is defined to be $\mathbf{E}[(X-\mu_X)^2]$. The *standard deviation* of X, denoted σ_X, is the positive square root of σ_X^2. (See Appendix C.)

Theorem 3.3 (Chebyshev's Inequality): *Let X be a random variable with expectation μ_X and standard deviation σ_X. Then for any $t \in \mathbb{R}^+$,*

$$\mathbf{Pr}[|X-\mu_X| \geq t\sigma_X] \leq \frac{1}{t^2}.$$

PROOF: First, note that

$$\mathbf{Pr}[|X-\mu_X| \geq t\sigma_X] = \mathbf{Pr}[(X-\mu_X)^2 \geq t^2\sigma_X^2].$$

The random variable $Y = (X-\mu_X)^2$ has expectation σ_X^2, and applying the Markov inequality to Y bounds this probability from above by $1/t^2$. □

3.3. Randomized Selection

We now consider the use of random sampling for the problem of selecting the kth smallest element in a set S of n elements drawn from a totally ordered universe. We assume that the elements of S are all distinct, although it is not very hard to modify the following analysis to allow for multisets. Let $r_S(t)$ denote the rank of an element t (the kth smallest element has rank k) and let $S_{(i)}$ denote the ith smallest element of S. We extend the use of this notation to subsets of S as well. Thus we seek to identify $S_{(k)}$.

In Step 1 (see following page), we sample with replacement: for instance, if an element s of S is chosen to be in R on the first of our $n^{3/4}$ drawings, the remaining $n^{3/4}-1$ drawings are all as likely to pick s again as any other element in S. This style of sampling appears to be wasteful, but we employ it here because it keeps our analysis clean. Sampling without replacement would result in a marginally sharper analysis, but in practice this may be slightly harder to implement: throughout the sampling process, we would have to keep track of the elements chosen so far.

Figure 3.1 illustrates Step 3, where small elements are at the left end of the picture and large ones at the right. Determining (in Step 4) whether $S_{(k)} \in P$ is easy since we know the ranks $r_S(a)$ and $r_S(b)$ and we compare either or both of these to k, depending on which of the three **if** statements in Step 4 we execute. The sorting in Step 5 can be performed in $O(n^{3/4}\log n)$ steps.

Algorithm LazySelect:

```
Input: A set $S$ of $n$ elements from a totally ordered universe, and an integer $k$
    in $[1, n]$.
Output: The $k$th smallest element of $S$, $S_{(k)}$.
1. Pick $n^{3/4}$ elements from $S$, chosen independently and uniformly at random
   with replacement; call this multiset of elements $R$.
2. Sort $R$ in $O(n^{3/4} \log n)$ steps using any optimal sorting algorithm.
3. Let $x = kn^{-1/4}$. For $\ell = \max\{\lfloor x - \sqrt{n} \rfloor, 1\}$ and $h = \min\{\lceil x + \sqrt{n} \rceil, n^{3/4}\}$, let
   $a = R_{(\ell)}$ and $b = R_{(h)}$. By comparing $a$ and $b$ to every element of $S$, determine
   $r_S(a)$ and $r_S(b)$.
4. if $k < n^{1/4}$, then $P = \{y \in S \mid y \le b\}$;
   else if $k > n - n^{1/4}$, let $P = \{y \in S \mid y \ge a\}$;
   else if $k \in [n^{1/4}, n - n^{1/4}]$, let $P = \{y \in S \mid a \le y \le b\}$;
   Check whether $S_{(k)} \in P$ and $|P| \le 4n^{3/4} + 2$. If not, repeat Steps 1–3 until such
   a set $P$ is found.
5. By sorting $P$ in $O(|P| \log |P|)$ steps, identify $P_{(k - r_S(a) + 1)}$, which is $S_{(k)}$.
```

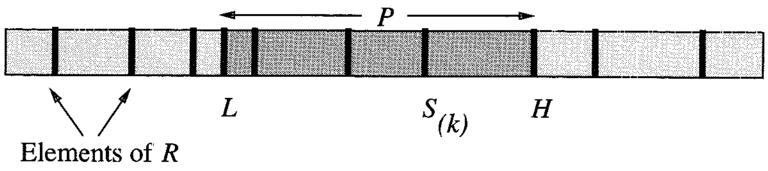

Figure 3.1: The LazySelect algorithm.

Thus the idea of the algorithm is to identify two elements a and b in S such that both of the following statements hold with high probability:

1. The element $S_{(k)}$ that we seek is in P.
2. The set P of elements between a and b is not very large, so that we can sort P inexpensively in Step 5.

We examine how either of these requirements could fail. We focus on the most interesting case when $k \in [n^{1/4}, n - n^{1/4}]$, so that $P = \{y \in S \mid a \le y \le b\}$; the analysis for the other two cases of Step 4 is similar and in fact somewhat simpler.

If the element a is greater than $S_{(k)}$ (or if b is smaller than $S_{(k)}$), we fail because P does not contain $S_{(k)}$. For this to happen, fewer than ℓ of the samples in R should be smaller than $S_{(k)}$ (respectively, at least h of the random samples should be smaller than $S_{(k)}$). We will bound the probability that this happens using the Chebyshev bound.

The second type of failure occurs when P is too big. To study this, we define $k_\ell = \max\{1, k - 2n^{3/4}\}$ and $k_h = \min\{k + 2n^{3/4}, n\}$. To obtain an upper bound on the probability of this kind of failure, we will be pessimistic and say that failure occurs if either $a < S_{(k_\ell)}$ or $b > S_{(k_h)}$. We prove that this is also unlikely, again using the Chebyshev bound. Before we perform this analysis, we establish an important property of independent random variables. Recall the definition of a joint density function $p(x, y)$ for random variables X and Y (Definition C.9).

► **Definition 3.1:** Let X and Y be random variables and $f(x, y)$ be a function of two real variables. Then,

$$\mathbf{E}[f(X,Y)] = \sum_{x,y} f(x,y)p(x,y).$$

For independent random variables X and Y we have from Proposition C.6

$$\mathbf{E}[XY] = \mathbf{E}[X]\mathbf{E}[Y]. \tag{3.2}$$

Lemma 3.4: *Let $X_1, X_2, \ldots, X_m$ be independent random variables. Let $X = \sum_{i=1}^m X_i$. Then $\sigma_X^2 = \sum_{i=1}^m \sigma_{X_i}^2$.*

PROOF: Let μ_i denote $\mathbf{E}[X_i]$, and $\mu = \sum_{i=1}^m \mu_i$. The variance of X is given by

$$\mathbf{E}[(X-\mu)^2] = \mathbf{E}[(\sum_{i=1}^m (X_i - \mu_i))^2].$$

Expanding the latter and using linearity of expectations, we obtain

$$\mathbf{E}[(X-\mu)^2] = \sum_{i=1}^m \mathbf{E}[(X_i-\mu_i)^2] + 2\sum_{i<j} \mathbf{E}[(X_i-\mu_i)(X_j-\mu_j)].$$

Since all pairs X_i, X_j are independent, so are the pairs $(X_i - \mu_i), (X_j - \mu_j)$. By (3.2), each term in the latter summation can be replaced by $\mathbf{E}[(X_i - \mu_i)]$ $\mathbf{E}[(X_j - \mu_j)]$. Since $\mathbf{E}[(X_i - \mu_i)] = \mathbf{E}[X_i] - \mu_i = 0$, the latter summation vanishes. It follows that

$$\mathbf{E}[(X-\mu)^2] = \sum_{i=1}^m \mathbf{E}[(X_i-\mu_i)^2] = \sum_{i=1}^m \sigma_{X_i}^2.$$

□

As in the analysis of **RandQS** in Chapter 1, we measure the running time of **LazySelect** in terms of the number of comparisons performed by it.

Theorem 3.5: *With probability $1 - O(n^{-1/4})$, **LazySelect** finds $S_{(k)}$ on the first pass through Steps 1–5, and thus performs only $2n + o(n)$ comparisons.*

PROOF: The time bound is easily established by examining the algorithm; Step 3 requires $2n$ comparisons, and all other steps perform $o(n)$ comparisons, provided the algorithm finds $S_{(k)}$ on the first pass through Steps 1–5. We now consider the first mode of failure listed above: $a > S_{(k)}$ because fewer than ℓ of the samples in R are less than or equal to $S_{(k)}$ (so that $S_{(k)} \notin P$). Let $X_i = 1$ if the ith random sample is at most $S_{(k)}$, and 0 otherwise; thus $\mathbf{Pr}[X_i = 1] = k/n$, and $\mathbf{Pr}[X_i = 0] = 1 - k/n$. Let $X = \sum_{i=1}^{n^{3/4}} X_i$ be the number of samples of R that are at most $S_{(k)}$. Note that we really do mean the number of samples, and not the number of distinct elements. The random variables X_i are *Bernoulli trials* (Appendix C): each may be thought of as the outcome of a coin toss. Then, using Lemma 3.4 and the variance of a Bernoulli trial with success probability p

$$\mu_X = \frac{kn^{3/4}}{n} = kn^{-1/4},$$

and

$$\sigma_X^2 = n^{3/4}\left(\frac{k}{n}\right)\left(1 - \frac{k}{n}\right) \leq \frac{n^{3/4}}{4}.$$

This implies that $\sigma_X \leq n^{3/8}/2$. Applying the Chebyshev bound to X,

$$\mathbf{Pr}[|X - \mu_X| \geq \sqrt{n}] \leq \mathbf{Pr}[|X - \mu_X| \geq 2n^{1/8}\sigma_X] = O\left(n^{-1/4}\right).$$

An essentially identical argument shows that

$$\mathbf{Pr}[b < S_{(k)}] = O\left(n^{-1/4}\right).$$

Since the probability of the union of events is at most the sum of their probabilities, the probability that either of these events occurs (causing $S_{(k)}$ to lie outside P) is $O(n^{-1/4})$.

Now for the second mode of failure – that P contains more than $4n^{3/4} + 2$ elements. For this, the analysis is very similar to that above in studying the first mode of failure, with k_ℓ and k_h playing the role of k. The analysis shows that $\mathbf{Pr}[a < S_{(k_\ell)}]$ and $\mathbf{Pr}[b > S_{(k_h)}]$ are both $O(n^{-1/4})$ (the reader should verify these details). Adding up the probabilities of all of these failure modes, we find that the probability that Steps 1–3 fail to find a suitable set P is $O(n^{-1/4})$. □

Exercise 3.4: The failure probability can be driven down further at the expense of increased running time. For a suitable definition of the $o(n)$ term, give an upper bound on the probability that the algorithm does not find $S_{(k)}$ in $cn + o(n)$ steps for $c > 2$.

Exercise 3.5: Theorem 3.5 tells us that the probability that **LazySelect** terminates in $2n + o(n)$ steps goes to 1 as $n \to \infty$. Suggest a modification in the algorithm that brings the constant in the linear term down to 1.5 from 2. We will refine this further in Problem 4.15.

This adds to the significance of **LazySelect**: the best known deterministic selection algorithms use $3n$ comparisons in the worst case and are quite complicated to implement. Further, it is known that any deterministic algorithm for finding the median requires at least $2n$ comparisons, so we have a randomized algorithm that is both fast and has an expected number of comparisons that is provably smaller than that of any deterministic algorithm. The high probability bound of the previous exercise can be easily converted into a bound on the expected running time:

Exercise 3.6: Show that as a direct corollary of Theorem 3.5, the expected running time of the **LazySelect** algorithm is $2n + o(n)$.

Consider what happens when we modify **LazySelect** to be recursive as follows: in Step 5, instead of sorting P we recursively use **LazySelect** to find $P_{(k-r_S(a)+1)}$. In this recursive version, the size of the candidate set P in which we are seeking $S_{(k)}$ is shrinking as the recursion proceeds. Using our analysis we can prove that at a typical stage of recursion the probability of failure at that stage is $O(|P|^{-1/4})$. But $|P|$ is diminishing, so that this probability of failure is rising as the algorithm proceeds! Thus, when the candidate set is down to a constant size, for instance, the failure probability is up to a constant and there is very little we can do about it. This is a fundamental barrier, not a weakness of our analysis. This is a typical problem with recursive randomized algorithms, and rears its head again in parallel randomized algorithms (where we always try to break a problem into smaller sub-problems) as well. A standard solution is to stop the recursion when the problem size is down to a certain size, and switch to a different, more expensive but deterministic technique – as we did by sorting in Step 5 of **LazySelect**.

3.4. Two-Point Sampling

We have so far been making use of the fact that the variance of the sum of *independent* random variables equals the sum of their variances. In fact, we can make a stronger statement. Let X and Y be discrete random variables defined on the same probability space. The *joint density function* of X and Y is the function

$$p(x,y) = \mathbf{Pr}[\{X = x\} \cap \{Y = y\}].$$

Thus $\mathbf{Pr}[Y = y] = \sum_x p(x,y)$, and

$$\mathbf{Pr}[X = x \mid Y = y] = \frac{p(x,y)}{\mathbf{Pr}[Y = y]}.$$

These definitions extend to a set $X_1, X_2, \ldots$ of more than two random variables. Such a set of random variables is said to be *pairwise independent* if for all $i \neq j$, and $x, y \in \mathbb{R}$,

$$\mathbf{Pr}[X_i = x \mid X_j = y] = \mathbf{Pr}[X_i = x].$$

We will use the result from the following exercise.

Exercise 3.7: Let n be a prime number and $\mathbb{Z}_n$ denote the ring of integers modulo n. For a and b chosen independently and uniformly at random from $\mathbb{Z}_n$, let $Y_i = ai + b \bmod n$. Show that for $i \not\equiv j \pmod{n}$, Y_i and Y_j are uniformly distributed on $\mathbb{Z}_n$ and pairwise independent. (Make use of the fact that in the field $\mathbb{Z}_n$, given fixed values for y_i and y_j, we can solve $y_i \equiv ai+b \pmod{n}$ and $y_j \equiv aj+b \pmod{n}$ uniquely for a and b.)

The following exercise is similar to Lemma 3.4.

Exercise 3.8: Let $X_1, X_2, \ldots, X_m$ be *pairwise* independent random variables, and $X = \sum_{i=1}^{m} X_i$. Show that $\sigma_X^2 = \sum_{i=1}^{m} \sigma_{X_i}^2$.

We now consider an application of these concepts to the reduction of the number of random bits used by ***RP*** algorithms (see Definition 1.8). Consider an ***RP*** algorithm A for deciding whether input strings x belong to a language L. Given x, A picks a random number r from the range $\mathbb{Z}_n = \{0, \ldots, n-1\}$, for a suitable choice of a prime n, and computes a binary value $A(x,r)$ with the following properties:

- If $x \in L$, then $A(x,r) = 1$ for at least half the possible values of r.
- If $x \notin L$, then $A(x,r) = 0$ for all possible choices of r.

For a randomly chosen r, $A(x,r) = 1$ is conclusive proof that $x \in L$, while $A(x,r) = 0$ is evidence that $x \notin L$.

For any $x \in L$, we refer to the values of r for which $A(x,r) = 1$ as *witnesses* for x; clearly, at least $n/2$ of the n possible values of r are witnesses. Of course, for $x \notin L$, there are no witnesses at all. The definition allows different $x \in L$ to have different sets of witnesses. Generally, n will be too large for us to test efficiently all the n potential witnesses for a given input x. However, for any $x \in L$, a random choice of r is a witness with probability at least $1/2$.

The fear is that $x \in L$ but the randomly chosen value of r yields $A(x,r) = 0$. However, we can drive down this probability of incorrectly classifying x by picking $t > 1$ values $r_1, \ldots, r_t$ independently from the range $\mathbb{Z}_n$, and computing $A(x, r_i)$ for all of them – in other words, by performing t independent iterations of the algorithm A on the same input x. If for any i we obtain $A(x, r_i) = 1$, we declare that x is in L, else we declare that x is not in L. By the independence of the trials, we are guaranteed that the probability of incorrectly classifying an input $x \in L$ (by declaring that it is not in L) is at most 2^{-t}.

Choosing t independent random numbers is expensive in that it requires $\Omega(t \log n)$ random bits. Suppose instead that we are only willing to use $O(\log n)$ random bits. In particular suppose that we wish to use only two independent samples from $\mathbb{Z}_n$. For a, b chosen independently from $\mathbb{Z}_n$, the naive usage of a and b as potential witnesses, i.e., computing $A(x,a)$ and $A(x,b)$, yields an upper bound of only 1/4 on the probability of incorrect classification. Here is a better scheme: let $r_i = ai + b \bmod n$, and compute $A(x, r_i)$ for $1 \le i \le t$. As before, if for any i we obtain $A(x, r_i) = 1$, we declare that x is in L, else we declare that x is not in L. What is the probability of incorrectly classifying any input x? We show that this probability is much smaller than 1/4.

We need to worry about the possibility of making error only in the case where the input x is in L. Our analysis will be insensitive to the actual values of r in $\mathbb{Z}_n$ which are witnesses for x; we will only rely on the fact that at least half the values of r are witnesses. Clearly $A(x, r_i)$ is a random variable over the probability space of pairs a and b chosen independently from $\mathbb{Z}_n$. By the result of Exercise 3.7, the random r_i's are pairwise independent and, therefore, so are the random variables $A(x, r_i)$, for $1 \le i \le t$. Let $Y = \sum_{i=1}^{t} A(x, r_i)$. Assuming that $x \in L$, $\mathbf{E}[Y] \ge t/2$ and $\sigma_Y^2 \le t/4$, or $\sigma_Y \le \sqrt{t}/2$. The probability that the pairwise independent iterations produce an incorrect classification corresponds to the event $\{Y = 0\}$, and

$$\mathbf{Pr}[Y = 0] \le \mathbf{Pr}[|Y - \mathbf{E}[Y]| \ge t/2].$$

By the Chebyshev inequality, the latter is at most $1/t$. Thus, the error probability is at most $1/t$, which is a considerable improvement over the error bound of 1/4 achieved by the naive use of a and b. This improvement is sometimes referred to as *probability amplification.*

For a random variable X with expectation μ_X, we define the kth *central moment* to be $\mu_X^k = \mathbf{E}[(X - \mu_X)^k]$, if it exists (Appendix C). For example, the variance is the second central moment.

Exercise 3.9: The use of the variance of a random variable in bounding its deviation from its expectation is called *the second moment method.* In an analogous fashion, we can speak of the *kth moment method*: let k be even, and suppose we have a random variable X for which $\mu_X^k = \mathbf{E}[(X - \mu_X)^k]$ exists. Show that

$$\mathbf{Pr}[|X - \mu_X| > t\sqrt[k]{\mu_X^k}\,] \le \frac{1}{t^k}.$$

Why is the kth moment method difficult to invoke for odd values of k?

The second moment method is generally useful for a random variable X if σ_X is $o(\mu_X)$. In a manner similar to "two-point" sampling (the name comes from the independent choice of two points a and b from which the r_i are derived), one can speak of k-point sampling for $k > 2$. The reader is referred to Appendix C for a further discussion of k-wise independence.

3.5. The Stable Marriage Problem

Consider a society in which there are n men (denoted by capital letters A,B,C, ...) and n women (denoted by a,b,c...). A *marriage* M is a 1-1 correspon-