Part VII

Randomized Algorithms

This part of the book is devoted to randomized algorithms. A randomized algorithm is simply an algorithm that uses a source of random bits, allowing it to make random moves. Randomized algorithms are extremely popular in computer science because (1) they are highly efficient (have low runtimes) on every input, and (2) they are often quite simple.

As we'll see, while randomized algorithms are very simple to state, analyzing their correctness and runtime will utilize all the probability tools that we have learned so far, plus some new tools.

Chapter 21 covers randomized algorithms of the Las Vegas variety. These algorithms always produce the correct answer, but their runtime depends on the random bits.

Next, in Chapters 22 and 23 we cover randomized algorithms of the Monte Carlo variety. These algorithms are extremely fast, regardless of the random bits. However, they return the correct answer only some fraction of the time, where the fraction depends on the random bits.

We only provide the briefest introduction to randomized algorithms in the text. The exercises offer many more examples and illustrate further directions. There are also several textbooks that are devoted entirely to randomized algorithms; see for example [21, 41, 53, 54].

21 Las Vegas Randomized Algorithms

This chapter introduces randomized algorithms. We start with a discussion of the differences between randomized algorithms and deterministic algorithms. We then introduce the two primary types of randomized algorithms: Las Vegas algorithms and Monte Carlo algorithms. This chapter and its exercises will contain many examples of randomized algorithms, all of the Las Vegas variety. In Chapter 22 we will turn to examples of the Monte Carlo variety.

21.1 Randomized versus Deterministic Algorithms

In deriving the runtime of an algorithm, we typically assume that there is an *adversary* who provides the input, and we consider the runtime of the algorithm on this input.

A **deterministic algorithm** always follows the same sequence of steps, and the adversary knows what steps the algorithm takes. Thus, the adversary can feed the algorithm a "worst-case input" on which it will take an exceptionally long time. The **runtime** of the algorithm is specifically defined as the runtime on that worst-case input.

By contrast, a **randomized algorithm** is an algorithm that makes use of a random sequence of bits in deciding what to do next. The adversary still gets to choose which input to feed the algorithm. However, because the randomized algorithm makes **random moves**, it is very hard for an adversary to defeat – that is, there often is no longer a worst-case input.

This brings us to the **primary advantage** of randomized algorithms: they are likely to be very *efficient* (low runtime) on *every* input. The adversary is powerless when the algorithm is randomized since the particular steps that the algorithm will take depends on random numbers. This makes it hard for the adversary to foil a randomized algorithm with a bad input that takes a long time.

When we say that randomized algorithms are "likely" to be efficient on every input, we mean that the randomness is over the string of random bits; one could always have a very poor choice of random bits which results in inefficiency. Randomized algorithms are often much faster than deterministic ones because they don't have a worst-case input. That said, because the algorithm uses random bits, the execution time of the algorithm can vary even on the same fixed input; that is, the execution time on a given input is a random variable (r.v.).

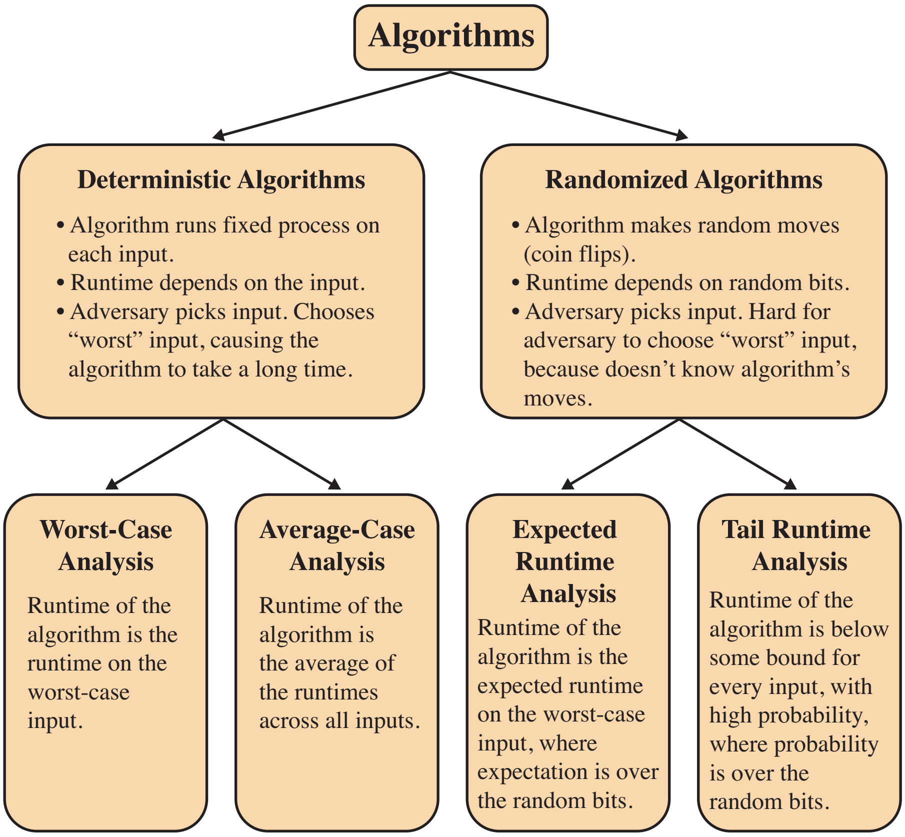
Algorithms
Deterministic Algorithms
• Algorithm runs fixed process on each input.
• Runtime depends on the input.
• Adversary picks input. Chooses "worst" input, causing the algorithm to take a long time.
Randomized Algorithms
• Algorithm makes random moves (coin flips).
• Runtime depends on random bits.
• Adversary picks input. Hard for adversary to choose "worst" input, because doesn't know algorithm's moves.
Worst-Case Analysis
Runtime of the algorithm is the runtime on the worst-case input.
Average-Case Analysis
Runtime of the algorithm is the average of the runtimes across all inputs.
Expected Runtime Analysis
Runtime of the algorithm is the expected runtime on the worst-case input, where expectation is over the random bits.
Tail Runtime Analysis
Runtime of the algorithm is below some bound for every input, with high probability, where probability is over the random bits.

Figure 21.1 *Deterministic versus randomized algorithms.*

It is important not to confuse randomized algorithms with the **average-case analysis of deterministic algorithms**. In average-case analysis, the *input* is drawn from a distribution, and the goal is to show that the algorithm is efficient in expectation over all the inputs. That is, while there may be some bad inputs on which the deterministic algorithm takes a really long time, if those inputs occur with low probability, then we can say that the deterministic algorithm performs well in expectation, where expectation is taken over the space of *all* inputs. When we talk about average-case analysis we are no longer talking about an adversary providing the input, but rather we can think of having a random input.

In the exercises we will see examples of both randomized algorithms and average-case analysis, so that you can see the difference between the two.

A **secondary advantage** of randomized algorithms is that they are often much *simpler* than deterministic algorithms. In fact, many randomized algorithms sound impossibly stupid, but work well and are very easy to describe.

21.2 Las Vegas versus Monte Carlo

There are two types of randomized algorithms, which are actually quite different.

A **Las Vegas** algorithm will always produce the correct answer. However, its running time on a given input is variable, depending on the sequence of random bits. Although for some random bits its running time is high, its average running time is hopefully low (where the average is taken over the sequence of random bits).

A **Monte Carlo** algorithm typically runs in a fixed amount of time, which is very short and is typically independent of the random choices made. However, it only gives the correct answer some fraction of the time. For example, a Monte Carlo algorithm may only produce the correct answer half the time. This may seem really stupid. What's the point of having an algorithm that gives the wrong answer? However, it's not as bad as it seems: The error probability depends on the particular random bits. Hence, runs are independent of each other and one can improve the correctness by running the algorithm multiple times (with freshly drawn random bits).

An example of a Monte Carlo algorithm is the Stochastic Gradient Descent (SGD) algorithm, used extensively in machine learning for finding the minimum of a multi-dimensional function. SGD reduces computation time over traditional Gradient Descent by only doing the needed minimization computations at a few randomly selected points. While the result may not always be correct, it's extremely fast.

We now present some examples of Las Vegas algorithms, which always produce the correct answer. In the chapter we will concentrate on expected runtime; however, the exercises will also consider the tail of the runtime distribution.

Our first randomized algorithm is Randomized Quicksort. Before we describe it, it helps to review Deterministic Quicksort.

21.3 Review of Deterministic Quicksort

Quicksort is an efficient algorithm for sorting a list of n numbers: $a_1, a_2, \ldots, a_n$. Throughout our discussion we will assume for convenience that these numbers are distinct. The **size** of the problem is the number of elements in the list being sorted, namely n. The **runtime** of the algorithm is the number of comparisons needed to sort the list. Throughout, we will use

$$C(n) = \text{number of comparisons needed when the problem size is } n.$$

In **Deterministic Quicksort**, the first element in the list (a_1) is designated as the *pivot*. All elements in the list are then compared with the pivot. Those elements less than the pivot are put into list $L1$, and those greater than the pivot are put into list $L2$, creating the list:

$$L1,\ a_1,\ L2.$$

Quicksort is then recursively applied to list $L1$ to obtain $L1s$ (sorted version of $L1$) and is recursively applied to list $L2$ to obtain $L2s$. The list returned is then

$$L1s,\ a_1,\ L2s.$$

Question: What is an example of a *bad* input list for Deterministic Quicksort?

Answer: In a sorted list, the pivot is always the smallest element in the list. Now all the elements end up in just *one* of the sublists, which is bad, because the size of the problem shrinks too slowly, resulting in high runtime.

Question: How many comparisons are needed in the case of a bad input list?

Answer: In the first step we compare the pivot with $n - 1$ elements. We then end up with a sublist of length $n - 1$, which requires $C(n - 1)$ comparisons to sort. Hence:

$$C(n) = (n - 1) + C(n - 1),$$

where $C(1) = 0$. Consequently $C(n) = O(n^2)$ on this bad input list.

Question: What is an example of a *good* input list for Deterministic Quicksort?

Answer: Ideally, we would like the pivot element to always be the median of the list. For example, consider the list:

$$\{5, 3, 2, 4, 7, 6, 8\},$$

which splits into:

$$\{3, 2, 4\}, 5, \{7, 6, 8\}$$

which further divides into:

$$\{2\}, 3, \{4\}, 5, \{6\}, 7, \{8\}.$$

Question: What is the number of comparisons needed by Deterministic Quicksort on a good input list?

Answer: Since the good input splits the list into two even lists at each step, we have approximately (ignoring rounding up or down):

$$\begin{aligned} C(n) &= n - 1 + 2C(n/2) \\ &= (n-1) + 2\left(n/2 - 1 + 2C(n/4)\right) \\ &= (n-1) + (n-2) + 4C(n/4) \\ &= (n-1) + (n-2) + 4\left(n/4 - 1 + 2C(n/8)\right) \\ &= (n-1) + (n-2) + (n-4) + 8C(n/8). \end{aligned}$$

Continuing in this fashion, we have that:

$$\begin{aligned} C(n) &= (n-1) + (n-2) + (n-4) + (n-8) + \cdots \\ &= n \lg n - \left(1 + 2 + 4 + \cdots + \frac{n}{2} + n\right) \\ &= n \lg n - 2n + 1 \\ &= O(n \lg n). \end{aligned}$$

21.4 Randomized Quicksort

We'd like the running time of Quicksort to be $O(n \lg n)$ on *every* input list. But how can we achieve this? The adversary can always choose to give us a bad input list that forces the running time to $O(n^2)$.

The solution is to use a randomized algorithm. Our **Randomized Quicksort** algorithm is identical to Deterministic Quicksort, except that the *pivot position* is chosen at random in each step. This makes it impossible for the adversary to give us a bad input list, which is the point of using randomness!

We will now prove that the expected running time of Randomized Quicksort is $O(n \lg n)$ on every input. Here, "expectation" is over all sequences of random pivot positions. In Exercise 21.13 you will invoke the Chernoff bound to show that *with high probability (w.h.p.)* the running time of Randomized Quicksort is $O(n \ln n)$ on every input.

Theorem 21.1 (Randomized Quicksort runtime) *Given any input list of n distinct elements, Randomized Quicksort will make $O(n \lg n)$ comparisons in expectation.*

Proof: Let $a_1, a_2, a_3, \ldots, a_n$ be an input. Let $s_1 < s_2 < s_3 < \ldots < s_n$ be the sorted version of this input. For $i < j$, let X_{ij} be an indicator random variable that takes on the value 1 if s_i and s_j are ever compared during the running of the algorithm and 0 otherwise. Note that s_i and s_j are compared at most once. Then, invoking Linearity of Expectation, we have:

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

$$\mathbf{E}[C(n)] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbf{E}\left[X_{ij}\right].$$

Question: What is $\mathbf{E}\left[X_{ij}\right]$, namely the probability that s_i and s_j are compared?

Hint: Think about the following *sorted sublist*: $S = \left[s_i, s_{i+1}, s_{i+2}, \ldots, s_j\right]$ and condition on which element in S is the first to be chosen to be a pivot.

Answer: At any moment of time before one of the elements of S has been chosen as a pivot, all the elements of S must be in the same sublist. Now consider that moment when one of the elements of S is first chosen as a pivot. If the pivot element chosen is s_i, then s_i will get compared with all the elements in S, and hence s_i and s_j will get compared. The argument is the same if the pivot element chosen is s_j. On the other hand, if any element of S other than s_i or s_j is chosen as the pivot, then after the pivot operation, s_i and s_j will end up in different sublists and will never get compared. Hence,

$$\mathbf{P}\left\{s_i \text{ and } s_j \text{ get compared}\right\} = \frac{2}{j-i+1}.$$

We thus have:

$$\begin{aligned}\mathbf{E}[C(n)] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbf{E}\left[X_{ij}\right] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= 2 \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{1}{k} \qquad \text{where } k = j-i+1 \\ &\le 2 \sum_{i=1}^{n} \sum_{k=2}^{n} \frac{1}{k}.\end{aligned}$$

Now, recalling the fact from (1.16) that

$$\sum_{i=1}^{n} \frac{1}{i} < 1 + \ln n,$$

we have:

$$\mathbf{E}\left[C(n)\right] \leq 2 \sum_{i=1}^{n} \sum_{k=2}^{n} \frac{1}{k} < 2 \sum_{i=1}^{n} (1 + \ln n - 1) = 2n \ln n.$$

We have thus shown that $\mathbf{E}\left[C(n)\right] = O(n \ln n) = O(n \lg n)$ as desired. ■

Summary: At this point, we have seen that Deterministic Quicksort, where the pivot is always chosen to be the first element of the list, has a worst-case input which forces $O(n^2)$ comparisons. By contrast, Randomized Quicksort, where the pivot is chosen randomly, has no worst-case input, and has an average runtime of $O(n \lg n)$, where this average is taken over the random choice of the pivot.

Question: Our analyses of both Deterministic Quicksort and Randomized Quicksort were *worst-case analyses* because the adversary was allowed to pick the worst possible input. What is meant by *average-case analysis of Quicksort*?

Answer: In average-case analysis, we are once again running Deterministic Quicksort, with our pivot always chosen to be the first element in the list, for example. However, rather than the input being chosen by an adversary, we assume that we have a *random input* – that is, a randomly ordered list. We derive the expected runtime, where the expectation is over the random ordering of the list.

Question: What is the runtime of Deterministic Quicksort under average-case analysis?

Answer: Because the input is randomly chosen, the adversary has no control over the first element in each sublist. So in each round, our pivot is effectively a random element in the list. Thus the computation of expected runtime is identical to what we saw for Randomized Quicksort, where we pick the pivot at random. Hence the expected runtime of the average-case analysis of Deterministic Quicksort is also $O(n \lg n)$.

21.5 Randomized Selection and Median-Finding

In the ***k*-Select** problem, we are given an unsorted list and asked to find the kth smallest element in the list. We'll assume that the list has n elements: $a_1, a_2, \ldots, a_n$. Again, for convenience, we assume that these numbers are distinct.

We will also ignore floors and ceilings in our discussion, so as to keep the notation from getting out of hand.

Question: What's an obvious way to solve k-Select in $O(n \lg n)$ time?

Answer: Sort the list, using Randomized Quicksort, and then return the kth element in the sorted list.

Our goal is to solve k-Select in $O(n)$ time.

Question: For certain values of k, it should be obvious how to achieve $O(n)$ time. What are these values?

Answer: If $k = 1$, then we can solve the problem just by walking through the list and keeping track of the smallest element so far. Similarly for $k = n$.

When $k = \frac{n}{2}$ (also known as the **Median-Select** problem), it is not at all obvious how to achieve $O(n)$ time.

We will present a very simple Las Vegas randomized algorithm for achieving $O(n)$ time on *every* input in expectation. The idea is to use random pivots as we did in the Randomized Quicksort algorithm. However, unlike the case of Quicksort, the pivot will allow us to throw away a part of the list.

Imagine that we start with a list of n elements, and our goal is to find the kth smallest element. We now pick a pivot at random. Suppose that our pivot happens to be the ith smallest element in the list, s_i. In $O(n)$ time, we can subdivide the list into $L1$, those $i - 1$ elements smaller than our pivot, and $L2$, those $n - i$ elements bigger than our pivot. Our kth smallest element is either in $L1$ or $L2$, or it is equal to the pivot (if $k = i$).

Question: If $k < i$, then our problem reduces to ...

Answer: Finding the kth element in $L1$, a list of size $i - 1$.

Question: If $k > i$, then our problem reduces to ...

Answer: Finding the $(k - i)$th element in $L2$, a list of size $n - i$.

We refer to the above algorithm as **Randomized *k*-Select**.

Before we write up the formal analysis, let's do a quick thought-experiment.

Question: Suppose that the pivot element always exactly splits the list in half. How many comparisons, $C(n)$, will be needed by our algorithm?

Answer: We need $n - 1$ comparisons to split the list. After splitting the list, we'll have reduced the problem to selection in a list of length $n/2$. Ignoring floors and ceilings, we have:

$$\begin{aligned} C(n) &= (n-1) + C(n/2) \\ &= (n-1) + (n/2 - 1) + C(n/4) \\ &< n + n/2 + n/4 + n/8 + \cdots + 1 \\ &\leq 2n. \end{aligned}$$

So $C(n) = O(n)$ if the pivot is always picked *optimally*.

We will now show that, if we pick a *random pivot*, we can still achieve $O(n)$ comparisons. Here, expectation is over the choice of the random pivot. Our derivation is an *upper bound* because we will assume that we are always reduced to looking at the *longest sublist* of the two randomly created sublists. This time we won't ignore floors and ceilings, so that you can see how to argue this precisely.

Theorem 21.2 (Randomized k-Select runtime) *For any list of n distinct elements, Randomized k-Select makes $\leq cn$ comparisons in expectation, where $c = 4$. This holds for any k.*

Proof: In general when writing a proof, one does not know exactly what the constant c will be. Thus, we will write our proof as if we are not given the value of c, and we will show how we can derive c as part of the proof, to get that $c = 4$.

Since the pivot is chosen randomly, it is equal to the ith smallest element with probability $\frac{1}{n}$. Hence we have:

$$\begin{aligned} \mathbf{E}\left[C(n)\right] &\leq (n-1) + \sum_{i=1}^{n} \mathbf{P}\left\{\text{pivot is } s_i\right\} \cdot \mathbf{E}\left[C\left(\max\{i-1, n-i\}\right)\right] \\ &= (n-1) + \sum_{i=1}^{n} \frac{1}{n} \cdot \mathbf{E}\left[C\left(\max\{i-1, n-i\}\right)\right] \\ &\leq (n-1) + \frac{2}{n} \sum_{i=\lfloor \frac{n}{2} \rfloor}^{n-1} \mathbf{E}\left[C(i)\right]. \end{aligned}$$

We will show that this results in $\mathbf{E}\left[C(n)\right] = O(n)$. We use induction. We claim that $\mathbf{E}\left[C(i)\right] \leq c \cdot i$ for some small integer $c \geq 1$ to be named later, and where $i < n$.

Since $\mathbf{E}\left[C(1)\right] = 0 \leq c \cdot 1$, the base case holds. Assuming that the inductive hypothesis holds for $i \le n-1$, we have:

$$\begin{aligned}
\mathbf{E}\left[C(n)\right] &\le (n-1) + \frac{2}{n}\sum_{i=\lfloor \frac{n}{2} \rfloor}^{n-1} c \cdot i \\
&= (n-1) + \frac{2c}{n} \cdot \frac{(n-1)+\lfloor \frac{n}{2} \rfloor}{2} \cdot \left(n-1-\left\lfloor \frac{n}{2} \right\rfloor + 1\right) \\
&\le (n-1) + \frac{2c}{n} \cdot \frac{(n-1)+\frac{n}{2}}{2} \cdot \left(n - \frac{n-1}{2}\right) \\
&= (n-1) + \frac{c}{n} \cdot \left(\frac{3n}{2} - 1\right) \cdot \frac{n+1}{2} \\
&= (n-1) + \frac{c}{4n} \cdot (3n-2) \cdot (n+1) \\
&= (n-1) + \frac{3cn}{4} + \frac{c}{4} - \frac{2c}{4n}. \qquad (21.1)
\end{aligned}$$

Our goal is to show that $\mathbf{E}\left[C(n)\right] \le cn$. From (21.1), we can see that, if we set $c = 4$, then we have that:

$$\begin{aligned}
\mathbf{E}\left[C(n)\right] &\le (n-1) + \frac{3 \cdot 4 \cdot n}{4} + \frac{4}{4} - \frac{2 \cdot 4}{4n} \\
&= (n-1) + 3n + 1 - \frac{2}{n} \\
&\le 4n.
\end{aligned}$$

So

$$\mathbf{E}\left[C(n)\right] \le 4n$$

is a solution to the original equation. We have thus proven the inductive case. ■

Question: Suppose we want to determine the median of a list of length n. How many comparisons are needed?

Answer: Still $O(n)$. If n is odd, we use Randomized k-Select with $k = (n+1)/2$. We refer to the median-finding algorithm as **Randomized Median-Select**.

21.6 Exercises

21.1 **Creating a fair coin**
You are given a biased coin that returns heads with probability 0.6 and tails otherwise. Let *Biased-Flip* be a routine that flips the biased coin once and returns the output. Design a Las Vegas algorithm, *Fair*, which

outputs heads with probability 0.5 and tails otherwise. Your algorithm, *Fair*, should only make calls to *Biased-Flip* and nothing else.

(a) State your *Fair* algorithm clearly.

(b) Prove that *Fair* outputs heads with probability 0.5 and Tails otherwise.

(c) Derive the expected number of calls to *Biased-Flip* required for *Fair* to produce an output.

21.2 **Creating a three-way fair coin**

Given a function *2WayFair* that returns 0 or 1 with equal probability, implement a Las Vegas function, *3WayFair*, that returns 0, 1, or 2 with equal probability. Aim to use a minimum number of calls to *2WayFair*.

(a) What is the expected number of calls to *2WayFair* made by *3WayFair*?

(b) Explain why *3WayFair* is a Las Vegas algorithm.

(Note: The solution is simple. Do not use any floating point arithmetic.)

21.3 **Nuts-and-bolts problem**

[Proposed by David Wajc] Imagine that you have n nuts, $N_1, N_2, \ldots, N_n$ with distinct sizes: $1, 2, 3, \ldots, n$. You also have n bolts, $B_1, B_2, \ldots, B_n$ with distinct sizes: $1, 2, 3, \ldots, n$, such that there is exactly one bolt that fits each nut. You can't see the nuts or the bolts, but you can perform a "trial" which consists of comparing one nut with one bolt. The result of a single trial is that either (a) they're a perfect fit, or (b) the bolt was too small, or (c) the bolt was too large. *You are not allowed to compare nuts with nuts or bolts with bolts.*

(a) Describe an efficient randomized algorithm for matching all n nuts to the n bolts in as few trials as you can. (Using $\Theta(n^2)$ trials is too many!)

(b) Derive the expected asymptotic running time of your algorithm.

21.4 **Ropes problem**

You have n ropes. Each rope has two ends. Consider the following randomized algorithm: At each step of your algorithm, you pick two random ends (these may be two ends from the same rope, or one end from one rope and one end from another rope), and tie these ends together. Keep going until there are no ends left. What is the expected number of cycles formed? Express your answer using $\Theta(\cdot)$.

21.5 **Uniform sampling from a stream**

[Proposed by David Wajc] Suppose you are walking down a long road, whose length you don't know in advance. Along the road are houses, which you would like to photograph with your very old camera. This old camera allows you to take as many pictures as you want, but only has enough memory to store *one* picture at a time. The street contains n houses, but you don't know n before you reach the end of the street.

Your goal is to end up with one photo in your camera, where that photo is equally likely to show any of the n houses.

One algorithm for achieving this goal is to walk all the way down the street, counting houses, so that we can determine n. Then we roll an n-sided die, where X denotes the roll outcome. Then we walk to the house numbered X and take its picture. However, you're a busy person and you don't want to walk down the street again. Can you achieve your goal by walking up the street only once? This problem is referred to as *uniform sampling from a stream with unknown length.*

(a) Propose a randomized algorithm for uniform sampling from a stream with unknown length. Your algorithm will involve replacing the item stored in memory with some probability as you walk (only once) down the street.

(b) Prove that, for all i, $\mathbf{P}\{i\text{th item is output}\} = \frac{1}{n}$.

21.6 **Pruning a path graph**

[Proposed by Vanshika Chowdhary] Figure 21.2 shows a path graph of n edges. At each round, you select a random edge of those remaining and cut it. Whenever an edge is cut, that edge and everything below that edge falls off. Let X be the number of edges that you cut until the entire path disappears (all edges are gone). What is $\mathbf{E}[X]$?

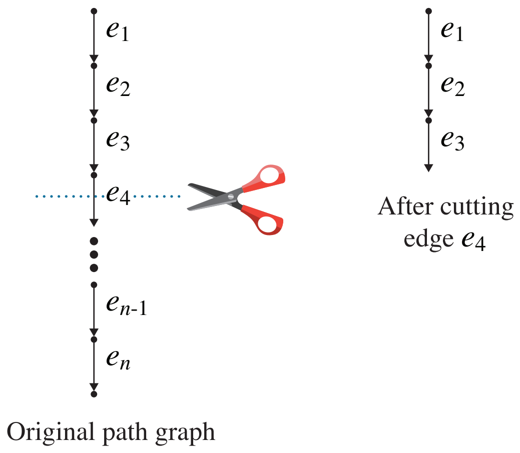

Figure 21.2 *For Exercise 21.6. Path graph with n edges, before and after pruning edge* e_4.

21.7 **Uniform sampling from a stream – generalized**

As in Exercise 21.5, you are walking down a long road with n houses, where you don't know n in advance. This time you have a *new camera* for photographing houses. This new camera has enough memory to store

s photos at a time. You walk all the way down the street *just once* taking photos. By the end of your walk, you want to have stored a random subset of s homes. (Assume $n \geq s$.)

(a) Provide a randomized algorithm for achieving your goal.

(b) Let S denote the set of houses stored in your camera. Prove that, at the end of your walk, each of the n houses has an equal probability of being in S.

21.8 **Finding the max – average-case analysis**

Given an array A of length n containing distinct integers $a_1, a_2, \ldots, a_n$, the *FindMax* algorithm determines the maximum number in A. Assuming the inputs are given in a uniformly random order, what is the expected number of times that *currentMax* is updated? Provide upper and lower bounds for this expression.

Algorithm 21.3 (FindMax($a_1, a_2, \ldots, a_n$))

```
1. currentMax = $-\infty$
2. for $i = 1, \ldots, n$ do
        if $a_i >$ currentMax then currentMax = $a_i$.
3. return currentMax
```

21.9 **Average-case analysis of Move-to-Front**

Suppose you use a linked list to store n items: $a_1, a_2, \ldots, a_n$. Then the time to access the ith stored item in the list is i. If you know that certain items are accessed more frequently, you would like to store them at the *front* of the list, so that their access time is shorter. Unfortunately, you don't know the access probabilities of items, so you use the (deterministic) Move-To-Front (MTF) algorithm: Each time an item is accessed, you append it to the front of the list, so that its access time is 1 (for now). Assume that MTF has been running for a long time. Our goal is to understand the expected time to look up an item in the list, call it $\mathbf{E}[T]$, given that item a_i is accessed with probability p_i.

(a) Prove that

$$\mathbf{E}[T] = 1 + \sum_{i=1}^{n} p_i \sum_{j \neq i} \frac{p_j}{p_j + p_i}. \qquad (21.2)$$

[Hint: Start by conditioning on the item, a_i, being accessed. The position of a_i can be expressed in terms of a sum of X_{ij} indicator random variables, where $\mathbf{E}\left[X_{ij}\right]$ is the probability that item a_j precedes a_i.]

(b) Verify your expression for $\mathbf{E}[T]$ in the case $p_i = \frac{1}{n}, \forall i$.

(c) Suppose that $p_i = C \cdot 2^{-i}$, $i = 1, 2, \ldots, n$, where C is the appropriate normalizing constant. Compute $\mathbf{E}[T]^{\text{MTF}}$ from (21.2), where $n = 5$ (you can write a small program). Now consider the case where we know the p_i's and we arrange the items according to the best arrangement (BA), namely in order of decreasing p_i. How does $\mathbf{E}[T]^{\text{BA}}$ compare with $\mathbf{E}[T]^{\text{MTF}}$?

21.10 **How to find a mate – average-case analysis**
[This is a repeat of Exercise 3.14, which is a nice example of average-case analysis.] Imagine that there are n people in the world. You want to find the best spouse. You date one person at a time. After dating the person, you need to decide if you want to marry them. If you decide to marry, then you're done. If you decide not to marry, then that person will never again agree to marry you (they're on the "burn list"), and you move on to the next person.
Suppose that after dating a person you can accurately rank them in comparison with all the other people whom you've dated so far. You do not, however, know their rank relative to people whom you haven't dated. So, for example, you might early on date the person who is the best of the n, but you don't know that.
Assume that the candidates are randomly ordered. Specifically, assume that each candidate has a unique score, uniformly distributed between 0 and 1. Our goal is to find the candidate with the highest score.

Algorithm 21.4 (Marriage algorithm)
1. *Date $r \ll n$ people. Rank those r to determine the "best of r."*
2. *Now keep dating people until you find a person who is better than that "best of r" person.*
3. *As soon as you find such a person, marry them. If you never find such a person, you'll stay unwed.*

What r maximizes $\mathbf{P}\{\text{end up marrying the best of } n\}$? When using that r, what is the probability that you end up marrying the best person? (In your analysis, feel free to assume that n is large and $H_n \approx \ln(n)$.)

21.11 **Finding the k largest elements**
Given an array A of n distinct elements in random order, we will consider two algorithms which each output the k largest elements in sorted order.
(a) Randomized Algorithm 1 uses Randomized k-Select to find the kth largest element, x. We then walk through the array, keeping only those elements $\geq x$. Finally, we sort these k largest elements via Randomized Quicksort. Derive an asymptotic expression for the expected number of comparisons. Since Algorithm 1 is randomized, the expectation is over the random bits.
(b) Deterministic Algorithm 2 maintains a sorted list at all times, $S = [s_1 > s_2 > \cdots > s_k]$, of the top-$k$-so-far. We start by sorting the first k elements of A via Deterministic Quicksort and calling that S. We now take each element, x, of A, starting with the $(k+1)$th element,

$x = a_{k+1}$, and insert it into its place in S. To do this, we compare x with each element of S starting with s_k and then s_{k-1} (if needed), and then s_{k-2} (if needed) and so on until x finds its place in S. This is the first run. In the second run, we insert the $(k+2)$th element of A into its proper place in S. There will be $n-k$ runs, many of which will not change S at all. Prove that the expected number of comparisons made is $O(n + k^2 \log n)$. Since Algorithm 2 is deterministic, the expectation is over the randomly ordered input.

21.12 **Randomized dominating set**

A *dominating set*, D, in a connected undirected graph $G = (V, E)$, is a set of vertices such that, for each $v \in V$, either $v \in D$ or v is adjacent to some $v' \in D$ (in both cases we say that v is *covered* by D). Assume that $|V| = n$ and that G is *d-regular*, with $d \geq 2$, meaning that each vertex has exactly d neighbors. Our goal is to find the minimum sized D.

(a) Sheng proposes the following randomized algorithm to find a valid D: Each vertex picks a random number in the range $(0, 1)$. For each edge, (i, j), we pick the endpoint with the larger number to be in D. In this way, for every edge (i, j), we are guaranteed that at least one of i and j are in D. What is $\mathbf{E}[|D|]$ found by Sheng's algorithm?

(b) A better randomized algorithm is Algorithm 21.5. Derive $\mathbf{E}[|D|]$ for Algorithm 21.5. Here are some steps:

(i) Express $\mathbf{E}[|D|]$ as a function of the p value in Algorithm 21.5.

(ii) Find the p that minimizes $\mathbf{E}[|D|]$. Express $\mathbf{E}[|D|]$ for this p.

(iii) Prove that $0 < \mathbf{E}[|D|] < n$. What happens to $\mathbf{E}[|D|]$ as d grows large?

Algorithm 21.5 (Dominating Set)

```
1. Given $G = (V, E)$, pick a random subset $D_0 \subseteq V$ where $D_0$
   includes each $v \in V$ with probability $p$.
2. Let $D_1$ be all vertices in $V$ that are not covered by $D_0$.
3. Return $D = D_0 \cup D_1$.
```

21.13 **Bounding the tail of Randomized Quicksort**

Use Chernoff bounds to show that, w.h.p. $(1 - \frac{1}{n})$, Randomized Quicksort requires only $O(n \lg n)$ comparisons to sort a list of length n. Here are some steps to help you:

(a) Consider a particular run of Randomized Quicksort as shown in Figure 21.3. The tree shows the list L at each stage and then shows the sublists L_1 and L_2 under that, separated by the pivot, p. You can imagine drawing such a tree for any instance of Randomized Quicksort. Let T denote the total number of comparisons made by Randomized Quicksort. Explain why T is upper-bounded by the sum of the lengths of all root-to-leaf paths in the ternary tree. Note that pivots count as leaves as well, so every element is eventually a leaf. [Hint: For each leaf, think about the number of comparisons that it's involved in.]

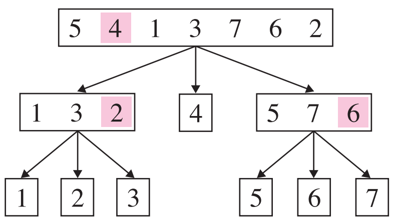

Figure 21.3 *Randomized Quicksort tree. The randomly selected pivot is in pink.*

(b) Now we'll argue that w.h.p. each root-to-leaf path is of length $O(\log n)$. Note: It's fine that some quantities are not integers.

(i) Let's say that a node of the tree is "good" if the randomly chosen pivot separates the current list at the node into two sublists, each of size at most $\frac{3}{4}$ the size of the current list. Otherwise we say that the node is "bad." What is the probability that a node is "good"?

(ii) Let g denote the maximum number of "good" nodes possible along a single root-to-leaf path. What is g as a function of n?

(iii) Consider an arbitrary leaf i. We want to prove that the root-to-leaf path ending in i is not very long. Specifically, show that

$$\mathbf{P}\left\{\text{The root-to-leaf path ending in } i \text{ has length } \geq 6g\right\} \leq \frac{1}{n^2}.$$

Here you're using the g from part (ii). Note that as soon as we see the first g "good" nodes, we'll be down to a single leaf.

(c) We have seen that with probability at least $1 - \frac{1}{n^2}$ a given root-to-leaf path is no longer than $6g$. What probabilistic statement about T follows from this?

21.14 **Randomized AND–OR tree evaluation**

Min–max game trees are often represented by an AND–OR tree on binary inputs, where AND is equivalent to "Min" and OR is equivalent to "Max." In an AND–OR tree, there are alternating levels of ANDs and ORs. The leaves of the tree are all 0's and 1's. Recall that $\text{AND}(a, b) = 1$ only if $a = b = 1$, while $\text{OR}(a, b) = 1$ if either $a = 1$ or $b = 1$ or both. Each node in the tree has a value (computed bottom-up) based on its subtrees; the value of the entire tree is the value of the root node. T_k denotes a tree with k AND levels and k OR levels, having height $2k$ and $2^{2k} = 4^k$ leaves. Figure 21.4 shows T_2.

(a) How many leaves must be evaluated in determining the value of T_k when a deterministic algorithm is used? What exactly will the adversary do to force you to evaluate that many leaves? The adversary

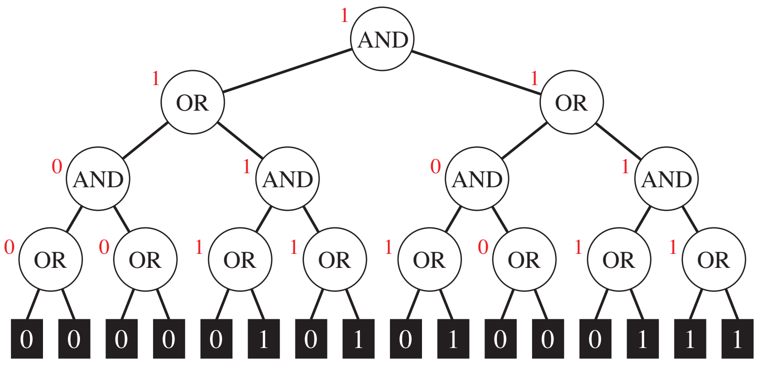

Figure 21.4 *This figure shows T_k, where $k = 2$. This means that there are $k = 2$ AND levels and $k = 2$ OR levels. The height of a T_k tree is $2k$. The values are computed bottom-up and are shown in red at each node. The final value of this tree is* 1.

knows the order in which your algorithm evaluates leaves and will give you the worst-case input.

(b) Consider the following *Randomized AND–OR algorithm*. This algorithm computes the value of each node in the tree, bottom-up. However, it *randomly* considers whether to first look at the left node or the right node, and then it doesn't bother looking at the remaining node unless necessary. Prove that the Randomized AND–OR algorithm requires $\leq 3^k$ leaf evaluations in expectation. Here, expectation is taken over the random bits used by the algorithm. As always, the adversary will try to give you the worst-case input; however, it will have a harder time because your moves are random.

(i) Start with a tree of height 1, consisting of two leaves connected by an OR. How many leaves on average must be evaluated if the value of your tree is 1? How about if the value of your tree is 0?

(ii) Now consider the tree T_k, where $k = 1$. This tree will have a single AND with two ORs underneath. How many leaves in expectation must be evaluated if the value of your tree is 1? What changes if the value of your tree is 0?

(iii) Prove via induction that you can determine the value of T_k in $\leq 3^k$ leaf evaluations in expectation. Do this both when the value of the tree is 1 and when it is 0.

21.15 **Multi-armed chocolate machine**

[Proposed by Weina Wang] A chocolate machine has two arms, as shown in Figure 21.5. If you pull Arm 1, it gives you a chocolate with probability $p_1 = \frac{3}{4}$. If you pull Arm 2, it gives you a chocolate with probability $p_2 = \frac{1}{4}$. Unfortunately, you don't know the values of p_1 and p_2, or which one is bigger.

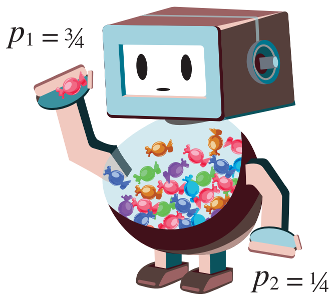

Figure 21.5 *Chocolate machine with two arms.*

Suppose pulling an arm once costs 1 dollar, and you have n dollars in total. Your goal is always to spend your n dollars to maximize the number of chocolates you receive in expectation.

(a) If you knew p_1 and p_2, how would you want to spend your n dollars? Let R^* denote the total number of chocolates you get. What is $\mathbf{E}[R^*]$?

(b) Since you do not know p_1 and p_2, you decide to pull each arm $\frac{n}{2}$ times (assume n is an even number). Let R_{rand} be the total number of chocolates you get. What is $\mathbf{E}[R_{\text{rand}}]$? Compare $\mathbf{E}[R_{\text{rand}}]$ with $\mathbf{E}[R^*]$.

(c) You figure that you can experiment with the arms a bit and decide how to use the rest of the money based on what you see. Suppose you pull each arm once to see which gives you chocolates.
- If one arm gives a chocolate and the other one does not, you use the remaining $n - 2$ dollars on the arm that gives a chocolate.
- Otherwise, you pick an arm uniformly at random and use the remaining $n - 2$ dollars on that arm.

Let R_{informed} be the total number of chocolates you get. What is $\mathbf{E}[R_{\text{informed}}]$? Compare $\mathbf{E}[R_{\text{informed}}]$ with $\mathbf{E}[R^*]$.

(d) You decide to experiment further. Suppose you pull each arm $m = 8 \ln n$ times. Let X and Y be the numbers of chocolates you get from Arm 1 and Arm 2, respectively. Then you do the following:
- If $X \geq Y$, you use the remaining $n - 2m$ dollars on Arm 1.
- Otherwise, you use the remaining $n - 2m$ dollars on Arm 2.

Let $R_{\text{well-informed}}$ denote the total number of chocolates you get. Derive a lower bound on $\mathbf{E}[R_{\text{well-informed}}]$. Show that $\mathbf{E}[R^*] - \mathbf{E}[R_{\text{well-informed}}] = O(\ln n)$.

For more general versions of this problem and more interesting algorithms, check out the multi-armed bandits literature (e.g. [47]).

21.16 **Infinite highway problem**
Imagine an infinitely long one-lane highway, starting at location 0 and

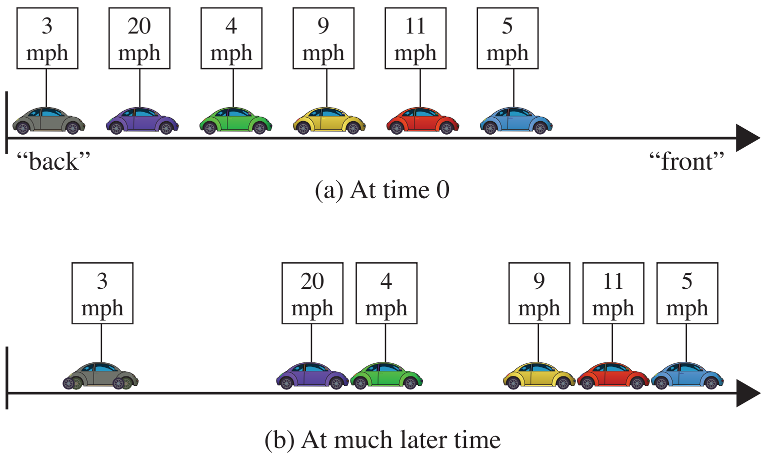

Figure 21.6 *One example of the infinite highway problem from Exercise 21.16.*

extending forever. There are n distinct cars, which start out evenly spaced. Each of the cars moves at some speed drawn independently from $\text{Uniform}(0, 100)$. The cars will drive forever on this one-lane highway, unable to pass each other, and faster cars will eventually get stuck behind slower cars that started in front of them. Over time, the cars will segregate into clusters. Figure 21.6 shows one particular example. Let X denote the number of clusters formed for a general instance of this problem.

(a) What is $\mathbf{E}[X]$?

(b) What is $\mathbf{Var}(X)$?

(c) Prove that X is less than $3\mathbf{E}[X]$ w.h.p. when n is high.

21.17 **Independent set**

[Proposed by Misha Ivkov] Let $G = (V, E)$ be a graph with $n = |V|$ vertices and $m = |E|$ edges. We say that $S \subset V$ is an independent set if no pair of vertices in S are connected by an edge. You will prove that G has an independent set S of size $\geq \frac{n^2}{4m}$. To do this, you will use the **probabilistic method**, which says: To prove that there is an independent set of size $\geq k$ in G, find a randomized algorithm which gives you an independent set of size S, where $\mathbf{E}[S] \geq k$. (Here S is a r.v. which depends on the random bits of the randomized algorithm.) Now you know there must be an independent set of size $\geq k$ in G.

Use the following Randomized Independent Set algorithm:

1. Pick each vertex of V to be in S with probability p.
2. If there exist two vertices in S that share an edge, randomly delete one.

Show that $\mathbf{E}[S] \geq \frac{n^2}{4m}$. Note: You will have to specify p.