Lecture Notes
CPSC 622/622, Winter 2013
RANDOMIZED ALGORITHMS

Philipp Woelfel

Contents

1 Some Basic Randomized Algorithms

1.1 The Hiring Problem

Suppose a company wants to hire an assistant. There are n candidates for the job. The approach of the company is the following (see Figure 1.1): It interviews each candidate, one after the other. The first candidate it interviews gets hired. Whenever the company interviews another candidate and that candidate is better than the one currently employed, the employed one is fired, and the new candidate is hired. Interviewing candidate i has cost c_i and hiring candidate i has cost c_h. We assume $c_h \gg c_i$.

In the worst-case, all candidates have to be interviewed and all candidates get hired. The worst-case occurs, if the candidates interview in increasing order of quality. The worst-case cost is $\Theta(n \cdot c_i + n \cdot c_h)$.

A Randomized Algorithm. A simple randomized algorithm could first bring the candidates in a random order and then interview them. Since only the relative quality of candidate matters, we can assume w.l.o.g. that the qualities of candidates are represented by distinct integers $1, \ldots, n$. Thus, our randomized algorithm behaves as the deterministic one for an input $(s_1, \ldots, s_n)$ that is a random permutation of the integers $1, \ldots, n$.

Random Variables and their Expectation. If we do this, the number of hirings is not a fixed number, but a number determined at random, i.e., a *random variable*. A *discrete random variable* X over a countable set $C \subseteq \mathbb{R}$ assigns each element $z \in C$ a probability $\text{Prob}(X = z)$, s.t.

1. $0 \leq \text{Prob}(X = z) \leq 1$ for all $z \in C$, and
2. $\sum_{z \in C} \text{Prob}(X = z) = 1$.

The *expectation* of a random variable X is defined as

$$E[X] := \sum_{z \in C} z \cdot \text{Prob}(X = z). \tag{1.1}$$

Algorithm 1: Hire

Input: A sequence $(s_1, \ldots, s_n)$ of $n \geq 1$ positive integers, representing the quality of the n candidates.
Output: $\max\{s_1, \ldots, s_n\}$.

```
1 best := 0.
2 for i = 1, ..., n do
      /* Interview candidate i:                                    */
3     if s_i > best then
          /* Hire candidate i                                      */
4         best := s_i
5     end
6 end
```

Figure 1.1: An Algorithm for the Hiring Problem

One of the most useful properties of random variables is the *linearity of expectation*, which means that for any two random variables X, Y, and any $z \in \mathbb{R}$

$$E[X + Y] = E[X] + E[Y], \quad \text{and} \quad E[z \cdot X] = z \cdot E[X].$$

Analyzing the Randomized Algorithm. Let X be the *random variable* that denotes the number of times we hire a new candidate. We are interested in the expectation of X, which according to (1.1) is

$$E[X] = \sum_{i=1}^{n} i \cdot \text{Prob}(X = i).$$

Let X_i be the random variable that has value 1 if candidate i is hired and value 0, otherwise. (X_i is an *indicator random variable*.) By definition of expectation,

$$E[X_i] = 0 \cdot \text{Prob}(X_i = 0) + 1 \cdot \text{Prob}(X_i = 1) = \text{Prob}(X_i = 1).$$

Then $X = X_1 + \cdots + X_n$, and thus by *linearity of expectation*

$$E[X] = E[X_1 + \cdots + X_n] = E[X_1] + \cdots + E[X_n] = \sum_{1 \leq i \leq n} \text{Prob}(X_i = 1). \tag{1.2}$$

What is the probability that the i-th candidate is better qualified than the first $i-1$ candidates? Choosing the first i random candidates from $\{1,\dots,n\}$ can be done by first choosing a subset $S \subseteq \{1,\dots,n\}$ of cardinality i and then bringing it in a random order. No matter what subset S we choose, the probability that the largest number appears last when we bring it into a random order is $1/i$. Thus, for all $1 \le i \le n$,

$$\text{Prob}(X_i = 1) = \frac{1}{i}.$$

Using (1.2) and Fact 1.1, we get

The Harmonic Series is defined as $H_k := \sum_{1\le i\le k} 1/i$.

Fact 1.1. $\ln(k+1) \le H_k \le \ln k + 1$.

Proof: Since $1/k$ is non-increasing we have

$$H_k = \sum_{1\le i\le k} \frac{1}{i} \le 1 + \int_1^k \frac{1}{x}dx = 1 + \Big[\ln(x)\Big]_1^k = 1 + \ln k.$$

and

$$H_k = \sum_{1\le i\le k} \frac{1}{i} \ge \int_1^{k+1} \frac{1}{x}dx = \ln(k+1).$$

$$E[X] = \sum_{1\le i\le n} \frac{1}{i} = \ln n + O(1)$$

Thus, the *expected* cost of hiring using the hiring algorithm for a random permutation of the input is now $O(c_i \cdot n + c_h \cdot \log n)$.

1.2 Randomized Quicksort

See the textbook, pp. 3-7.

Deterministic Quicksort Recall the deterministic Quicksort algorithm, depicted in Figure 1.2. There is no rule how to choose the pivot element. In practice, the set S is stored in an array, and often a simple rule is applied to choose the pivot, e.g., the last element in the array.

In the worst-case, in each recursive call for a set S, the pivot-element is always the largest one from the set S. Then $S_2 = \emptyset$ and $|S_1| = |S| - 1$. It is not hard to see that the number of comparisons in this case is

$$n-1+n-2+n-3+\cdots+1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Omega(n^2).$$

Algorithm 2: DetQS

Input: A set S of n different numbers, stored in an array
Output: the numbers from S in increasing order

1 Choose an arbitrary *pivot* $y \in S$.
2 $S_1 := \{x \in S \mid x < y\}$; $S_2 := \{x \in S \mid x > y\}$.
3 **if** $S_1 \neq \emptyset$ **then** DetQS(S_1)
4 Output y
5 **if** $S_2 \neq \emptyset$ **then** DetQS(S_2)

Figure 1.2: Deterministic Quicksort

Average Case Running Time The average case running time of an algorithm is the expected running time for a random input. E.g., in the case of Quicksort, the average case running time is the same as the expected running time of a random permutation of the integers $\{1, \ldots, n\}$. It is known that the average case running time of the algorithm DetQS is $\Theta(n \log n)$, not matter how the pivot element is chosen.

The average case running time is often not a very helpful complexity measure, because in practice the data that has to be processed is rarely random. Especially for sorting applications, the data to be sorted is often either partially sorted or inversely sorted or has some other non-random properties. The idea of a randomized algorithm is to use *random choices* to make up for a lack of randomness in the input data. While we usually cannot rely on the randomness in the data, we can "control" the the randomness used by an algorithm.

Randomizing Quicksort It is known that if the input set S is given by an array that is randomly distributed, then the expected number of comparisons is only $O(n \log n)$. We can achieve a similar behaviour even for the worst-case input, by choosing the pivot-element uniformly at random in each round. Consider the randomized version, `RandQS`, in Figure 1.3.

How Likely is the Worst Case? First note that the worst case in which in each recursive call either the largest or smallest element is being chosen is very unlikely. For one iteration in which a set of size k is considered, the probability that this happens is

Algorithm 3: RandQS

Input: A set S of n integers
Output: the numbers from S in increasing order

1 Choose a pivot $y \in S$ **uniformly at random** (each element in S has probability $1/n$ of being chosen).
2 $S_1 := \{x \in S \mid x < y\}$; $S_2 := \{x \in S \mid x > y\}$.
3 **if** $S_1 \neq \emptyset$ **then** RandQS(S_1)
4 Output y
5 **if** $S_2 \neq \emptyset$ **then** RandQS(S_2)

Figure 1.3: Randomized Quicksort

$2/k$. Hence, the probability that this happens in every recursive call is

$$\frac{2}{n} \cdot \frac{2}{n-1} \cdots \frac{2}{2} = \frac{2^{n-1}}{n!} \leq \frac{2^{n/2}}{(n/2)^{n/2}} = \left(\frac{4}{n}\right)^{n/2}.$$

This is an **exponentially small** probability. But the running time can still be very bad, even if the worst case does not occur.

Expected Running Time We are interested in the *expected* number of comparisons in a run of the algorithm. Let $s_1, \ldots, s_n$ be the element from S such that $s_1 \leq s_2 \leq \cdots \leq s_n$. Let $X_{i,j}$, $1 \leq i < j \leq n$, be indicator random variables with

$$X_{i,j} = \begin{cases} 1 & \text{if } s_i \text{ and } s_j \text{ are compared during a run of RandQS}(S) \\ 0 & \text{otherwise.} \end{cases}$$

Note that any two elements can be compared at most once. Hence, the total number of comparisons during a run of RandQS(S) is a random variable X, where

$$X = \sum_{1 \leq i < n} \sum_{i < j \leq n} X_{i,j}.$$

The expected number of comparisons is the expectation of X, $E[X]$, and by *linearity of expectation*, we have

$$E[X] = E\left[\sum_{1 \leq i < n} \sum_{i < j \leq n} X_{i,j}\right] = \sum_{1 \leq i < n} \sum_{i < j \leq n} E[X_{i,j}].$$

Let $p_{i,j}$ be the probability that s_i and s_j are being compared. Then $\text{Prob}(X_{i,j} = 1) = p_{i,j}$. Hence,

$$E[X_{i,j}] = \sum_{z \in \{0,1\}} z \cdot \text{Prob}(X_{i,j} = z) = 0 \cdot \text{Prob}(X_{i,j} = 0) + 1 \cdot \text{Prob}(X_{i,j} = 1) = p_{i,j}.$$

Thus,

$$E[X] = \sum_{1 \le i < n} \sum_{i < j \le n} p_{i,j}. \tag{1.3}$$

To analyze this, we have to determine $p_{i,j}$. Recall that $s_1, \ldots, s_n$ is the sorted list of elements in S. For $1 \le a \le b \le n$ let $S_{a,b} = \{s_a, s_{a+1}, \ldots, s_b\}$. Note that each element is chosen as pivot exactly once.

Lemma 1.2. *It is $X_{i,j} = 1$ if and only if during the run of the algorithm, the first element in $S_{i,j}$ chosen as pivot is in $\{s_i, s_j\}$.*

Proof: During each recursive call for a set S in which pivot $y \in S$ is chosen, the following happens:

1. y gets compared with all elements in $S - \{y\}$, and
2. S gets split up in three subsets, $S_< = \{x \in S \mid x < y\}$, $\{y\}$, and $S_> = \{x \in S \mid x > y\}$.

Then recursive calls are made for $S_<$ and $S_>$. Hence, s_i and s_j get compared only if there is a recursive call made for a set S that contains s_i and s_i, and either s_i or s_j is chosen as a pivot.

Note that due to the way S is split up, all elements in one set for which a recursive call is being made are consecutive elements in the list $s_1, \ldots, s_n$. (This is immediate from a simple induction on the number of recursive calls.) Hence, s_i and s_j get compared if and only if at some point in time a recursive call is made for a set S that contains all of $s_i, s_{i+1}, \ldots, s_j$ and at that point either s_i or s_j is chosen as a pivot. As long as none of $s_i, \ldots, s_j$ is chosen as pivot, all those elements remain in the same set, and s_i and s_j don't get compared.

So now consider the point in time when for the the first time a value $s^* \in \{s_i, \ldots, s_j\}$ is being chosen as pivot. Then the set S for which the recursive call of DetQS(S) is being made when this happens contains $s_i, \ldots, s_j$ as a subset. If $s^* \in \{s_i, s_j\}$, then clearly s^* gets compared with all elements in $S - \{s^*\}$ and thus also with the element in $\{s_i, s_j\} - \{s^*\}$, so s_i and s_j get compared. Otherwise, if $s^* \notin \{s_i, s_j\}$ then the set S is split into the subsets $S_<$ and $S_>$, and $S_<$ contains s_i and $S_>$ contains s_j. In this case, s_i and s_j won't be compared during the remainder of the algorithm. □

The probability that s_i or s_j is the first element in $S_{i,j} = \{s_i, \ldots, s_j\}$ to be chosen as pivot is

$$\frac{2}{|S_{i,j}|} = \frac{2}{j-i+1}.$$

Hence, $p_{i,j} = 2/(j-i+1)$. Using Fact 1.1, we can simplify (1.3) as follows:

$$E[X] = \sum_{1\le i<n} \sum_{i<j\le n} \frac{2}{j-i+1} = \sum_{1\le i<n} \sum_{1\le k\le n-i} \frac{2}{k+1} = \sum_{1\le i<n} \left(\sum_{1\le k\le n-i+1} \frac{2}{k} - 1 \right)$$
$$= 2 \cdot \sum_{1\le i<n} (H_{n-i+1} - 1) = 2 \cdot \sum_{1\le i\le n} (H_i - 1) = \Theta\left(\sum_{2\le i\le n} \log i \right).$$

Clearly,

$$\sum_{2\le i\le n} \log i \le \sum_{2\le i\le n} \log n \le n \cdot \log n.$$

Moreover, for $n \ge 4$, looking only at the largest $n/2$ terms of the following sum:

$$\sum_{2\le i\le n} \log i \ge \sum_{n/2\le i\le n} \log(n/2) = \frac{n}{2} \cdot \big((\log n) - 1 \big) = \Omega(n \cdot \log n).$$

This yields the following theorem.

Theorem 1.3. *The expected number of comparisons in an execution of RandQS is* $\Theta(n \cdot \log n)$.

1.3 Verifying the Equality of Strings

Assume you have large files on your computer at home and on your computer at work. You want to verify that the files are the same on both computers, without transferring the files. More precisely, consider two players, Alice and Bob. Alice has a string $a \in \{0,1\}^n$ and Bob a string $b \in \{0,1\}^n$. How many bits do they have to communicate in order to decide whether $a = b$?

Techniques from *communication complexity* show that if they want to do this deterministically, they have to transfer n bits. The idea is now to compute (short) random fingerprints $f(a)$ and $f(b)$ for a and b, and compare only these fingerprints with each other. Thus, f should be some random function $f : \{0,1\}^n \to \{0,1\}^m$ for $m \ll n$.

But what is a "random function", and how do we describe it? In general, we can use a *family of functions*, $\mathcal{F}$, and pick $f \in \mathcal{F}$ uniformly at random. If we do this, on average we need at least $\lceil \log |\mathcal{F}| \rceil$ bits to describe the function f. Then Alice can send Bob the description of f as well as $f(a)$, and Bob can compute $f(b)$ and compare it to $f(a)$.

Note that Alice and Bob can interpret their strings as integers, i.e., the string $x \in \{0,1\}^n$ is interpreted as the integer $\sum_{i=0}^{n-1} x_i \cdot 2^i$. Therefore, from now on assume that a and b are integers in $\mathbb{N} \cup \{0\}$.

We use the family $\mathcal{F}$ that consists of functions $f_p : \mathbb{N} \to \{0, \ldots, p-1\}$, $f_p(x) = x \bmod p$, where p is a prime in $\{2, \ldots, 2^m - 1\}$ (m depends on the length of Alice's string). Thus, in order to describe f_p, Alice can send a description of the prime p to Bob.

For the ease of discussion assume now that n (the length of Alice's string) and m are fixed and Bob knows these two values. The protocol then works as follows: Alice picks a random prime p from $\{2, \ldots, 2^m - 1\}$. Then Alice computes the fingerprint $f_p(a)$ and sends the pair $\big(p, f_p(a)\big)$ to Bob. Now Bob can compute the fingerprint $f_p(b)$ and compare it with $f_p(a)$. He sends the result of that comparison back to Alice.

In total, at most $2m + 1$ bits need to be transferred. But how do we choose m and what is the error probability?

If $a = b$, then $a \bmod p = b \bmod p$, so Alice and Bob correctly decide that the strings are equal.

If $a \neq b$, then an error occurs if and only if $a \bmod p = b \bmod p$. So we have to bound the probability for the event "$a \bmod p = b \bmod p$" for a randomly chosen p in $\{2, \ldots, 2^m - 1\}$.

Now for any positive integer M let $\lambda(M)$ be the number of primes in $\{2, \ldots, M\}$.

Lemma 1.4. *Let $x, y \in \{0, \ldots, N-1\}$, $x \neq y$. If p is a random prime in $\{2, \ldots, M-1\}$, then*

$$\text{Prob}\big(x \bmod p = y \bmod p\big) \leq \frac{\log N}{\lambda(M)}.$$

Proof: Assume w.l.o.g. that $x > y$ and let $d = x - y$. Then $x \bmod p = y \bmod p$ if and only if $d \bmod p = 0$, i.e., if d is a multiple of p.

Write d as its prime factorization, i.e., $d = p_1 \cdots p_k$, where each p_i, $1 \leq i \leq k$, is a prime divisor of d. Clearly, $k \leq \log d$, because $p_i \geq 2$ for $1 \leq i \leq k$. Hence, if we pick p at random, the probability that $p \in \{p_1, \ldots, p_k\}$ is at most $k/\lambda(M)$.

We get

$$\text{Prob}\big(x \bmod p = y \bmod p\big) \leq \frac{k}{\lambda(M)} \leq \frac{\log d}{\lambda(M)} \leq \frac{\log N}{\lambda(M)}.$$

□

So what is $\lambda(M)$? We need the following famous statement from number theory:

Theorem 1.5.

$$\lim_{M \to \infty} \frac{\lambda(M)}{M/\ln M} = 1,$$

and if $M > 67$, then

$$\lambda(M) > \frac{M}{\ln M} > 1.44 \cdot \frac{M}{\log M}$$

Corollary 1.6. *Whether two strings of length n are equal, can be decided randomly by comparing two strings of length $m \geq 7$, and with a one-sided error probability (false positives) of at most $n \cdot m/2^m$.*

For example, pick $m = \max\{7, \lceil c \cdot \log n \rceil\}$ for some constant c. Then $M = 2^m > 67$, so we can apply the second part of Theorem 1.5. Then the error probability is at most

$$\frac{n \cdot c \cdot \log n}{2^m} = \frac{n \cdot c \cdot \log n}{n^c}.$$

Alice needs to send only $2m = O(\log n)$ bits to Bob (the m-bit prime and the m-bit fingerprint). Hence, using fingerprints of *logarithmic* lengths, we can get arbitrary small *polynomial* error probabilities.

Or, pick $m = \log n \log\log n$. Then the error probability is

$$O\left(\frac{n \log n \log\log n}{2^{\log n \log\log n}}\right) = O\left(\frac{\log n \cdot \log\log n}{n^{\log\log n - 1}}\right) = \frac{1}{n^{\omega(1)}}.$$

In order to make this practical, we have to be able to find a random prime in $\{2, \ldots, M\}$ efficiently. We can simply choose a number uniformly at random from $\{2, \ldots, M\}$ and then test whether it is prime—we need an efficient primality test algorithm, such as the randomized Miller-Rabin test, or the deterministic algorithm by Agrawal, Kayal, and Saxena. If the random number is not prime, we repeat.

The probability that a random number in $\{2, \ldots, M\}$ is prime is

$$\frac{\lambda(M)}{M} = \Theta(1/\log M).$$

So after an expected number of $O(\log M)$ trials, we have found a prime.

2 Basics in Probability Theory I

Note: This is not covered in the textbook, although some of the definitions can be found in Appendix C.

2.1 Probability Space

Definition 2.1. A *discrete probability space* is a countable set Ω with a mapping $\text{Prob} : 2^\Omega \to \mathbb{R}$, such that

(A1) $\text{Prob}(\{w\}) \geq 0$ for all $w \in \Omega$,

(A2) $\text{Prob}(A) = \sum_{w \in A} \text{Prob}(\{w\})$ for all $A \subseteq \Omega$, and

(A3) $\text{Prob}(\Omega) = 1$.

The subsets of Ω are called *events* and the events of cardinality one are called *elementary events*. In order to define a probability space it suffices to define Ω and the mapping Prob for the elementary events. By Axiom (A2), the definition of $\text{Prob}(A)$ for all other events A is then unique.

Example 2.2. A model for the result of a die-roll: Define $\Omega = \{1, \ldots, 6\}$ and $\text{Prob}(\{1\}) = \text{Prob}(\{2\}) = \cdots = \text{Prob}(\{6\}) = 1/6$. The only way to satisfy axiom (A2) is to define $\text{Prob}(A) = |A|/6$ for all $A \subseteq \Omega$. Note that this implies $\text{Prob}(\Omega) = \text{Prob}(\{1, \ldots, 6\}) = 1$, so (A3) is true.

With this definition, the probability that the die shows either 1 or 2 pips is $\text{Prob}(\{1, 2\}) = \text{Prob}(\{1\}) + \text{Prob}(\{2\}) = 1/3$.

Example 2.3. A model for rolling two dice, one red and one blue:

- $\Omega := \{(i, j) \mid 1 \leq i, j \leq 6\}$, $\text{Prob}(A) = |A|/|\Omega|$.
- Thus, $\text{Prob}(\{(i, j)\}) = 1/36$ for all $1 \leq i \leq j \leq 6$.

Example 2.4. A model for rolling two indistinguishable dice:

- $\Omega = \{(i, j) \mid 1 \leq i \leq j \leq 6\}$, $\text{Prob}(A) = |A|/|\Omega|$.
- Thus, $\text{Prob}(\{(i, j)\}) = 1/|\Omega| = 1/(6 + 5 + 4 + 3 + 2 + 1) = 1/21$ for $1 \leq i \leq j \leq 6$.

Theorem 2.5. *Let* (Ω, Prob) *be a discrete probability space. Then for all* $A, B \subseteq \Omega$

1. $0 \leq \text{Prob}(A) \leq 1$,
2. $\text{Prob}(\emptyset) = 0$,
3. $\text{Prob}\left(\overline{A}\right) = 1 - \text{Prob}(A)$,
4. $A \subseteq B \;\Rightarrow\; \text{Prob}(A) = \text{Prob}(B) - \text{Prob}(B - A)$, *and*
5. $\text{Prob}(A \cup B) = \text{Prob}(A) + \text{Prob}(B) - \text{Prob}(A \cap B) \leq \text{Prob}(A) + \text{Prob}(B)$. *(The inequality* $\text{Prob}(A \cup B) \leq \text{Prob}(A) + \text{Prob}(B)$ *is called* union bound*.)*

Example 2.6. Consider the model from Example 2.3 for throwing a blue and a red die. We discuss two events and their probabilities.

- A: *Both dice show an even number of pips.*

  It is $|A| = 9 = 3 \cdot 3$, because there are three possibilities for each of the dice to show an even number of pips. I.e., A contains 9 elementary events. In this model, each elementary event has probability $1/36$. Hence, $\text{Prob}(A) = 9/36 = 1/4$.

- B: *The total number of pips is even.*

  Then $B = B_1 \cup B_2$, where B_1 is the event that both dice show an even number of pips, and B_2 is the event, that both dice show an odd number of pips. Hence, $\text{Prob}(B) = \text{Prob}(B_1 \cup B_2) = \text{Prob}(B_1) + \text{Prob}(B_2) - \text{Prob}(B_1 \cap B_2) = 1/4 + 1/4 - 0 = 1/2$.

2.2 Conditional Probabilities

Definition 2.7. Let (Ω, Prob) be a probability space, and $A, B \subseteq \Omega$. Then

$$\text{Prob}(A \mid B) := \begin{cases} \dfrac{\text{Prob}(A \cap B)}{\text{Prob}(B)} & \text{if } \text{Prob}(B) \neq 0, \text{ and} \\ 0 & \text{if } \text{Prob}(B) = 0. \end{cases}$$

is the *conditional probability* that A occurs, given that B occurs.

Example 2.8. In the analysis of randomized Quicksort, we considered the following problem: Let $S = \{s_1, \ldots, s_n\}$ and $S_{i,j} = \{s_i, \ldots, s_j\}$, $1 \leq i < j \leq n$. Pick elements from S uniformly at random (without replacement), one after the other. What is the probability that s_i or s_j is chosen before any other element in $S_{i,j}$?

Fix some point in time at which the random element is picked from a subset $S' \supseteq S_{i,j}$. Let s be the element we pick. Let A be the event that $s \in \{s_i, s_j\}$, and B be the event that $s \in S_{i,j}$. Then the conditional probability that $s \in \{s_i, s_j\}$ given that $s \in S_{i,j}$ is

$$\text{Prob}(A \mid B) = \frac{\text{Prob}(A \cap B)}{\text{Prob}(B)} = \frac{\text{Prob}(A)}{\text{Prob}(B)} = \frac{2/|S'|}{|S_{i,j}|/|S'|} = \frac{2}{|S_{i,j}|} = \frac{2}{j-i+1}.$$

A useful statement is the Total Probability Theorem.

Theorem 2.9 (Total Probability)**.** *For any two events $A, B \subseteq \Omega$*

$$\text{Prob}(A) = \text{Prob}(A|B) \cdot \text{Prob}(B) + \text{Prob}\big(A|\overline{B}\big) \cdot \text{Prob}\big(\overline{B}\big).$$

Proof. By Definition 2.7 (conditional probabilities)

$$\text{Prob}(A|B) \cdot \text{Prob}(B) + \text{Prob}\big(A|\overline{B}\big) \cdot \text{Prob}\big(\overline{B}\big) = \text{Prob}(A \cap B) + \text{Prob}\big(A \cap \overline{B}\big).$$

Since $A \cap B$ and $A \cap \overline{B}$ are disjoint, by Theorem 2.5

$$\text{Prob}(A \cap B) + \text{Prob}\big(A \cap \overline{B}\big) = \text{Prob}\big((A \cap B) \cup (A \cap \overline{B})\big) = \text{Prob}(A).$$

□

Repeated application of the Total Probability Theorem yields that for any partition $B_1, \ldots, B_k$ of Ω (i.e., $B_1, \ldots, B_k$ are disjoint and their union is Ω), we have

$$\text{Prob}(A) = \sum_{1 \leq i \leq k} \text{Prob}(A|B_i) \cdot \text{Prob}(B_i). \tag{2.1}$$

Definition 2.10. Two events A and B are *independent*, if $\text{Prob}(A|B) = \text{Prob}(A)$.

Theorem 2.11. *Two events A and B are* independent*, if and only if*

$$\text{Prob}(A \cap B) = \text{Prob}(A) \cdot \text{Prob}(B).$$

3 A Min-Cut Algorithm

This is covered in Chapter 1.1 of the textbook, but the algorithm presented here is slightly different.

Definition 3.1. Let $G = (V, E)$ be an undirected graph.

1. A *k-cut* is a partition of V into k disjoint sets $V_1, \ldots, V_k \neq \emptyset$.
2. A *crossing edge* of a k-cut $\{V_1, \ldots, V_k\}$ is an edge $\{a, b\} \in V_i \times (V - V_i)$ for some $1 \leq i \leq k$.
3. A *minimal k-cut* of G is a k-cut that has as few crossing edges as possible.

See Figure 3.1 for a randomized algorithm, Min2Cut, that computes a 2-cut. What is the probability that the output is a *minimal* 2-cut?

Fix an arbitrary minimal 2-cut $\{A, B\}$ of G and let C be the set of its crossing edges and $c = |C|$ the number of the crossing edges. Consider a run of the algorithm, and let e_i, $1 \leq i \leq n-2$, be the edge picked in round i (i.e., when $k = n - i + 1$).

If for the output $Cut_2 = \{V_1, V_2\}$ of the algorithm we have $V_1 = A$ or $V_1 = B$, then the number of crossing edges of that cut is c, and thus $\{V_1, V_2\}$ is a minimal 2-cut. Thus, we prove a lower bound on the probability that V_1 equals either A or B.

Lemma 3.2. *It is $V_1 = A$ or $V_1 = B$, if and only if $e_1, \ldots, e_{n-2} \notin C$.*

Proof: "$\Rightarrow$": If $e_i \in C \subseteq A \times B$ for some $1 \leq i \leq n$, then both endpoints of e_i end up in the same set V_1 or V_2 (when the algorithm terminates), and thus V_1 is neither A nor B.

Algorithm 4: Min2Cut

Input: An undirected graph $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$, $n \geq 2$.

Output: A two-cut $\{V_1, V_2\}$.

1 Let $k := n$ and $Cut_n := \Big\{\{v_1\}, \ldots, \{v_n\}\Big\}$.

2 **while** $k > 2$ **do**

3 Pick an edge e from the crossing edges of the cut Cut_k uniformly at random.

4 Let $U, U' \in Cut_k$ be the disjoint sets such that one endpoint of e is in U and the other in U'.

5 Let $Cut_{k-1} = \big((Cut_k - U) - U'\big) \cup \{U \cup U'\}$. `/* I.e., replace the sets` U `and` U' `in the cut with their union. */`

6 **end**

7 Output the cut Cut_2.

Figure 3.1: Computing a 2-Cut of a graph.

"$\Leftarrow$": Now suppose $e_1, \ldots, e_{n-2} \notin C$. Consider some set C_k computed during the algorithm. We show by induction on decreasing values of k that for each set $U \in C_k$ it holds either $U \subseteq A$ or $U \subseteq B$. For $k = n$ this is true since each set $U \in C_n$ contains exactly one node. Now consider an iteration in which two sets U, U' get merged in line 5. Let e be the edge that was chosen in this round, so e has one endpoint in U and the other in U'. Since $e \notin C$, it cannot have one endpoint in A and the other in B. W.l.o.g. assume both endpoints are in A. By the induction hypothesis, U and U' are subsets of the same set, A or B, but since both have one endpoint from e and that endpoint is in A, we conclude that U and U' are both subsets of A. Hence, $U \cup U' \subseteq A$. It follows that Cut_{k-1} computed in the iteration contains only subsets of A and subsets of B. □

From the above we get

$$\begin{aligned}
\text{Prob}(V_1 = A \vee V_1 = B) &= \text{Prob}(e_1, \ldots, e_{n-2} \notin C) = \\
&\text{Prob}(e_{n-2} \notin C \mid e_1, \ldots, e_{n-3} \notin C) \cdot \text{Prob}(e_1, \ldots, e_{n-3} \notin C) = \\
&\qquad \text{Prob}(e_{n-2} \notin C \mid e_1, \ldots, e_{n-3} \notin C) \cdot \\
&\text{Prob}(e_{n-3} \notin C \mid e_1, \ldots, e_{n-4} \notin C) \cdot \text{Prob}(e_1, \ldots, e_{n-4} \notin C) = \\
\cdots &= \left(\prod_{2 \leq i \leq n-2} \text{Prob}(e_i \notin C \mid e_1, \ldots, e_{i-1} \notin C) \right) \cdot \text{Prob}(e_1 \notin C) \\
&= \prod_{1 \leq i \leq n-2} \text{Prob}(e_i \notin C \mid e_1, \ldots, e_{i-1} \notin C) \quad (3.1)
\end{aligned}$$

For each value of $2 \leq k \leq n$ let m_k be the number of crossing edges in the k-cut that is considered in the iteration of that value of k. Note that in the i-th iteration, i.e., when edge e_i is chosen, we have $k = n - i + 1$. Thus,

$$\text{Prob}(e_i \in C \mid e_1, \ldots, e_{i-1} \notin C) = \frac{c}{m_{n-i+1}},$$

or, equivalently,

$$\text{Prob}(e_i \notin C \mid e_1, \ldots, e_{i-1} \notin C) = 1 - \frac{c}{m_{n-i+1}}.$$

We obtain from (3.1)

$$\text{Prob}(V_1 = A \vee V_2 = B) = \prod_{1 \leq i \leq n-2} \left(1 - \frac{c}{m_{n-i+1}}\right). \tag{3.2}$$

In order to prove a lower bound on this probability, we need to upper bound c/m_{n-i+1}. Recall that m_{n-i+1} is the number of crossing edges of the cut $(V_1, \ldots, V_k)$ considered in the i-th iteration for $k = n - i + 1$. Hence, it is the number of crossing edges in a k-cut. We now relate this number to the number of edges in a minimal 2-cut of the graph:

Observation 3.3. *Let $G = (V, E)$ have a k-cut with m_k crossing edges. Then G has a 2-cut with at most $2m_k/k$ crossing edges.*

Proof: Let $\{V_1, \ldots, V_k\}$ be a k-cut of G. Each of the m_k crossing edge has two endpoints, so there is a total of $2m_k$ endpoints. By averaging over the k sets $V_1, \ldots, V_k$ (pigeon hole principle), it follows that there is a set V_i, such that at most $2m_k/k$ of the crossing edges have endpoints in V_i. Therefore, $(V_i, V - V_i)$ is a 2-cut of G with at most $2m_k/k$ crossing edges. □

Applying Observation 3.3 we have

$$c \leq \frac{2m_k}{k} \quad \text{and thus} \quad \frac{c}{m_{n-i+1}} \leq \frac{2}{n-i+1}.$$

Plugging this into (3.2) yields

$$\begin{aligned}\text{Prob}(V_1 = A \vee V_1 = B) &\geq \prod_{1 \leq i \leq n-2} \left(1 - \frac{2}{n-i+1}\right) = \prod_{1 \leq i \leq n-2} \frac{n-i-1}{n-i+1} \\ &= \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3} = \frac{2}{n(n-1)}.\end{aligned}$$

This is a lower bound on the probability, that the algorithm outputs a minimal 2-cut.

Theorem 3.4. *The algorithm Min2Cut outputs a minimal 2-cut with proability at least*

$$\frac{2}{n(n-1)}.$$

The algorithm Min2Cut *errs* with probability less than $1 - 2/n^2$. In order to achieve a smaller error probability we can repeat the algorithm k times and then choose the minimal cut among all outputs. What is the probability that this new algorithm does not output a minimal cut?

Let A_i, $1 \leq i \leq k$, denote the event, that Min2Cut errs when we run it for the i-th time. Then by Theorem 3.4, $\text{Prob}(A_i) < 1 - 2/n^2$. Whenever we repeat the algorithm, we use random choices *independent* from all previous ones. Thus, the event A_i is independent from the events $A_1, \ldots, A_{i-1}$, and we obtain the following bound for the probability that Min2Cut errs k times:

$$\begin{aligned}\text{Prob}(A_1 \cap \cdots \cap A_k) = \text{Prob}(A_1) \cdots \text{Prob}(A_k) &< (1 - 2/n^2)^k \\ &= \left((1 - 2/n^2)^{n^2/2}\right)^{2k/n^2} < \frac{1}{e^{2k/n^2}}.\end{aligned}$$

(Here we used the fact that $(1 - 1/t)^t < 1/e$ for all $t \geq 1$.)

E.g., if we repeat $n^2/2$ times, then the error probability is less than $1/e$. For $n^3/2$ repetitions the error probability is $1/e^n$, i.e., it is *exponentially small*. But of course, for k repetitions the running time increases by a factor of k. This technique, repeating an algorithm multiple times to reduce the error probability, is called *probability amplification*.

4 Basics in Probability Theory II

4.1 Random Variables

Definition 4.1. A (discrete) *random variable* X over a discrete probability space (Ω, Prob) is a function $X : \Omega \to M$, where $M \subseteq \mathbb{R}$ is a countable set.

Notation. A random variable $X : \Omega \to M$ defines a new probability space (M, p_X), where

$$p_X(\{z\}) = \text{Prob}(\{a \in \Omega \mid X(a) = z\}).$$

We use the following notation for $z \in \mathbb{R}$.

- $\text{Prob}(X = z) := p_X\big(\{z\} \cap M\big)$.
- $\text{Prob}(X \leq z) := p_X(\{y \in \mathbb{R} \mid y \leq z\} \cap M)$.
- Analogously we can define $\text{Prob}(X < z)$, $\text{Prob}(X \geq z)$, $\text{Prob}(X > z)$, etc.

Example 4.2. Let X be the random variable denoting the result of a die roll. The probability space modeling a die roll is (Ω, Prob), where $\Omega = \{1, \ldots, 6\}$ and $\text{Prob}(\{w\}) = 1/6$ for all $w \in \Omega$. The random variable X is the mapping $X : \Omega \to \mathbb{N}$, $z \mapsto z$. Using the notation above, we get for example $\text{Prob}(X = 5) = \text{Prob}(\{5\}) = 1/6$, and $\text{Prob}(X \leq 3) = \sum_{1 \leq i \leq 3} \text{Prob}(X = i) = 3 \cdot 1/6 = 1/2$.

Arithmetic with Random Variables. Let $(\Omega, \mathbf{Pr})$ be a probability space. For two random variables $X : \Omega \to M_X, Y : \Omega \to M_Y$, $M_X, M_Y \subseteq \mathbb{R}$, and a binary operation $\circ$ on $\mathbb{R}$, we define

$$X \circ Y : \Omega \to M', z \mapsto X(z) \circ Y(z),$$
$$\text{where } M' = M_X \circ M_Y = \{x \circ y \mid (x, y) \in M_X \times M_Y\}.$$

Example 4.3. Throw a red and a blue die. Let X and Y denote the number of pips shown by the red and the blue die, resp. Let Z denote the sum of the pips, i.e., $Z = X + Y$. Then we have

$$\text{Prob}(Z = 3) = \text{Prob}(X + Y = 3) = \text{Prob}(\{a \in \Omega \mid X(a) + Y(a) = 3\}) =$$
$$\text{Prob}(\{(1, 2), (2, 1)\}) = 2/36 = 1/18.$$

4.2 Expectation

Recall that for a countable set M a series $\sum_{i\in M} s(i)$ is *absolutely* convergent, if $\sum_{i\in M} |s(i)|$ converges.

Definition 4.4. The *expectation* of a discrete random variable $X : \Omega \to M$, $M \subseteq \mathbb{R}$, is

$$E[X] := \sum_{z\in M} z \cdot \text{Prob}(X = z) \quad \left(= \sum_{w\in\Omega} X(w) \cdot \text{Prob}(\{w\}) \right),$$

if the series is *absolutely convergent*.

If the series in Definition 4.4 does not converge absolutely, then the expectation of X is not defined. Why? If the series is only conditionally convergent (i.e., convergent, but not absolutely convergent), then by the Riemann Series Theorem the terms of the series can be rearranged such that the series converges to any given value, or even diverges. Of course the expectation should not depend on the order of the terms in the series, therefore the expectation is undefined in this case.

Example 4.5. Throw one die and let X be the number of pips it shows. Then the expectation of X is

$$\sum_{1\le i\le 6} i \cdot \frac{1}{6} = \frac{1}{6} \cdot \sum_{1\le i\le 6} i = \frac{1}{6} \cdot 6 \cdot (6+1)/2 = 3.5.$$

Many of the random variables we analyze are running times of randomized algorithms. Such random variables are usually positive integers. If $X > 0$ and $X \in \mathbb{N}$ with probability 1, then there is a very useful characterizations of the expectation.

Theorem 4.6. *Let $X : \Omega \to \mathbb{N} \cup \{0\}$ be a random variable. If $E[X]$ exists, then*

$$E[X] = \sum_{i=0}^{\infty} \text{Prob}(X > i).$$

Proof:

$$\begin{aligned}
E[X] &= \sum_{k=0}^{\infty} k \cdot \text{Prob}(X = k) = \sum_{k=1}^{\infty} k \cdot \text{Prob}(X = k) \\
&= \text{Prob}(X = 1) + 2 \cdot \text{Prob}(X = 2) + 3 \cdot \text{Prob}(X = 3) + \dots \\
&= \sum_{k=1}^{\infty} \text{Prob}(X = k) + \sum_{k=2}^{\infty} \text{Prob}(X = k) + \sum_{k=3}^{\infty} \text{Prob}(X = k) + \dots \\
&= \sum_{i=1}^{\infty} \sum_{k=i}^{\infty} \text{Prob}(X = k) = \sum_{i=1}^{\infty} \text{Prob}(X \geq i) \\
&= \sum_{i=1}^{\infty} \text{Prob}(X > i - 1) = \sum_{i=0}^{\infty} \text{Prob}(X > i)
\end{aligned}$$

□

Example 4.7 (Geometric Distribution)**.** A random variable $X : \Omega \to \mathbb{N}$ *geometrically* distributed with parameter q, if

$$\text{Prob}(X = k) = q(1 - q)^{k-1} \quad \text{for } k \in \mathbb{N}.$$

For example, how often, on average, do we have to repeat the Min2Cut Algorithm in order to find a minimal 2-cut? Let q be the probability that the algorithm outputs a minimal 2-cut, and let X be the number of repetitions, until the Min2Cut Algorithm succeeds. By Theorem 3.4, we have $q \geq 2/\big(n(n-1)\big)$. The probability that the algorithm succeeds after exactly k repetitions, is that it fails $k-1$ times, and then succeeds. Since the repetitions are *independent*,

$$\text{Prob}(X = k) = (1 - q)^{k-1} \cdot q.$$

Thus, the random variable X is geometrically distributed with parameter $q \geq 2/\big(n(n-1)\big)$. So what is $E[X]$?

For a geometrically distributed random variable it holds by Theorem 4.6

$$\begin{aligned}
E[X] = \sum_{k \geq 0} \text{Prob}(X > k) = \sum_{k \geq 0} (1 - q)^k &= \lim_{n \to \infty} \sum_{k=0}^{n} (1 - q)^k \\
&= \lim_{n \to \infty} \frac{(1 - q)^{n+1} - 1}{(1 - q) - 1} = \frac{1}{q}. \quad (4.1)
\end{aligned}$$

Thus, the expected number of repetitions, until the Min2Cut algorithm outputs a minimal 2-cut, is at most $n(n-1)/2$.

Lemma 4.8 (Linearity of Expectation)**.** *Let X and Y be two random variables and $a \in \mathbb{R}$. Then*

$$E[a \cdot X] = a \cdot E[X] \quad \textit{and} \quad E[X + Y] = E[X] + E[Y].$$

Proof: Let $X : \Omega \to M$ and $Y : \Omega \to M'$, $M, M' \subseteq \mathbb{R}$.

1. We have

$$E[a \cdot X] = \sum_{z \in a \cdot M} z \cdot \text{Prob}(a \cdot X = z) = \sum_{z \in a \cdot M} z \cdot \text{Prob}(X = z/a)$$
$$= \sum_{z/a \in M} (z/a) \cdot a \cdot \text{Prob}(X = z/a) \overset{z'=z/a}{=} a \cdot \sum_{z' \in M} z' \cdot \text{Prob}(X = z') = a \cdot E[X].$$

2. Exercise.

Example 4.9. Consider a weighted coin which shows heads with probability p. If we flip the coin n times, how often do we expect it to show heads? Let X be the corresponding random variable. Then $X = X_1 + \cdots + X_n$, where $X_i = 1$ is an indicator random variable with value 1 if the i-th coin flip shows heads and with value 0 otherwise. It holds $E[X_i] = p$ and by linearity of expectation $E[X] = n \cdot p$.

Theorem 4.10. *If X and Y are independent random variables and whose expectations exist, then*

$$E[X \cdot Y] = E[X] \cdot E[Y].$$

(Without proof.)

5 Binary Planar Partitions

See Section 1.3 in the textbook.

Consider a set $S = \{s_1, \ldots, s_n\}$ of line segments in the plane $\mathbb{R}^2$. Given a location p of a camera outside of the convex hull of S, we would like to draw the line segments as they look like from the perspective of the camera. Parts of line segments that are "hidden" behind other line segments should not be visible. The idea is to use an algorithm that first paints the line segments that are the furthest away from the camera, and then paint closer ones, overpainting the hidden line segment parts. We would like to postprocess the set S of line segments in such a way, that given p we can quickly draw the line segments.

During the postprocessing phase we construct a data structure that is called a *binary planar partition*, defined in the following. Let T be a binary tree, where each inner node is labeled with a line in the plane $\mathbb{R}^2$. Then each node in the tree corresponds to a convex subset of $\mathbb{R}^2$ (which we will sometimes call *region*), determined recursively as follows:

1. The root of T corresponds to $\mathbb{R}^2$.
2. Let v be a node labeled with a line ℓ_v, and let $R_v \subseteq \mathbb{R}^2$ be the convex set corresponding to v. The line ℓ_v splits R into two convex subsets R_0 and R_1 (R_0 containing $\ell_v \cap R$), and the children of ℓ_v correspond to these subsets.

Definition 5.1. Let $S = \{s_1, \ldots, s_n\}$ be a set of non-intersecting line segments in the plane. A *binary partition* of S is a binary tree T whose internal nodes are labeled with lines in the plane, such that each region corresponding to a leaf in the tree contains at most one (not necessarily complete) line segment of S.

A binary partition T is an *autopartition* of S, if each node is labeled with a line ℓ such that $s_i \subseteq \ell$ for some $i \in \{1, \ldots, n\}$.

5.1 The Painter's Algorithm.

We compute a binary partition T of S in a preprocessing step. Then, given the location of the camera, p, and the root r of T, we run the algorithm Paint(r, p) (see Algorithm 5).

Algorithm 5: Paint

Input: A vertex u in T and a point $p \in \mathbb{R}^2$

```
1 if u is a leaf then
2 |   paint the part of the line segment in R_u (if any).
3 else
4 |   Let v, w be the children of u, such that R_w is "behind" line ℓ_u from the
  |   perspective of the camera, i.e., R_w is farther away from p than R_v (if p is on
  |   ℓ_u, then the order is arbitrary).
5 |   Paint(w, p)
6 |   Paint(v, p)
7 end
```

It is desirable to have a binary planar partition as small as possible, because the running time of the painter's algorithm grows with the tree size. Since segments are broken into smaller pieces, it is not clear that a solution of size $O(n)$ exists.

5.2 A Randomized Algorithm for Binary Autopartitions

For a line segment s, we denote by $\ell(s)$ the line extending s in both directions. Algorithm 6, RandAuto, computes a binary autopartition of a set of non-intersecting line segments, where $\ell(s) \neq \ell(s')$ for any two segments s, s' from the input. Note that if there are two line segmenst s, s' with $\ell(s) = \ell(s')$ an autopartition might not exist. However, in order to achieve that no two lines $\ell(s), \ell(s')$ are the same, we can rotate each line by a very small (negligible) angle. The assumptions that the line segments are non-intersecting is not problematic either, because two intersecting segments can be cut into four non-intersecting segments.

Theorem 5.2. *The expected size of the tree returned by RandAuto is $O(n \log n)$.*

Proof: Let $ind(s_i, s_j)$ be the number of line segments one intersects if one extends segment s_i in one direction until the extension hits s_j (hitting s_j counts as one intersection). If $\ell(s_i)$ does not intersect s_j, then $ind(s_i, s_j) = \infty$.

We write $s_i \dashv s_j$, if $s_i \neq s_j$, and segment s_j is cut by $\ell(s_i)$ in the partition defined by T. More formally, there is a vertex v in T, labeled with $\ell_v = \ell(s_i)$, such that the part of the line segment of s_j that lies in R_v (i.e., $R_v \cap s_j$) intersects with ℓ_v.

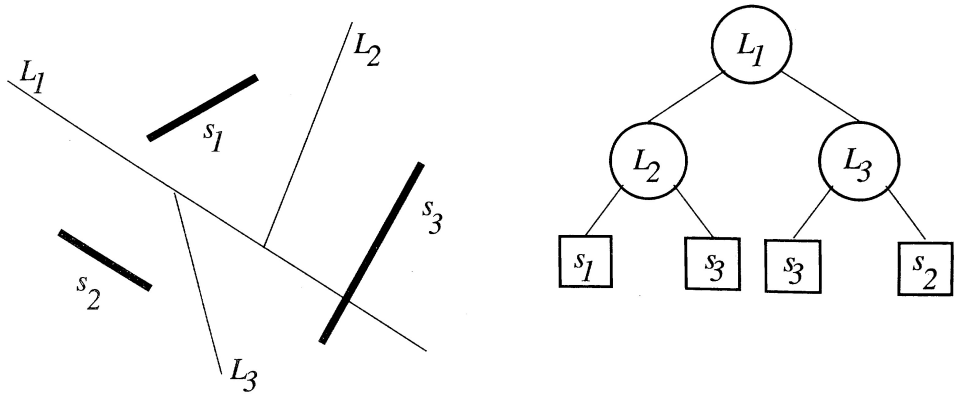


Figure 5.1: An example (taken from the textbook) for a binary planar partition (not an autopartition). The line segments are s_1, s_2 and s_3, and the partition is formed by the lines L_1, L_2 and L_3. Each leaf is labeled by the line segment that the region corresponding to that leaf contains.

Algorithm 6: RandAuto

Input: A set $S = \{s_1, \ldots, s_n\}$ of non-intersecting line segments, where $\ell(s_i) \neq \ell(s_j)$ for all $1 \leq i < j \leq n$.

Output: A binary autopartition T of S.

1 Pick a permutation π of $\{s_1, \ldots, s_n\}$ uniformly at random from the $n!$ permutations.

2 **while** *T has a leaf v corresponding to a convex set R_v that contains more than one segment* **do**

3 Label v with $\ell(s_i)$, where s_i is first segment in the ordering π that intersects R_v.

4 Add the corresponding children to v (choose the children in such a way that the region that does not intersect s_i contains another line segment from R_v).

5 **end**

6 Output T.

We consider the following random variables for $1 \leq i, j \leq n$:

$$C_{i,j} = \begin{cases} 1 & \text{if } s_i \dashv s_j. \\ 0 & \text{otherwise.} \end{cases}$$

The total number of uncut line segments is $n + \sum_{1 \leq i,j \leq n} C_{i,j}$. This is equal to the total number of leaves in the tree—note that every region corresponding to a leaf intersects with *exactly one* line segment.

Thus, the expected number of leaves in the tree is

$$E[n + \sum_{1 \leq i,j \leq n} C_{i,j}] = n + \sum_{1 \leq i,j \leq n} E[C_{i,j}] \tag{5.1}$$

What is the probability that $C_{i,j} = 1$, or, equivalently, that $s_i \dashv s_j$? Consider the one sided extension of s_i that hits s_j. Let $k = ind(s_i, s_j)$, and $r_1, \ldots, r_{k-1}, r_k = s_j$ be the line segments that the extension intersects until it hits s_j. Then none of the segments $\{r_1, \ldots, r_{k-1}, s_j\}$ can appear in π before s_i: Consider the time at which $\ell(s_i)$ is chosen and when it splits a region R_v containing the intersection point of $\ell(s_i)$ and s_j. Then the intersection points of $\ell(s_i)$ and $r_1, \ldots, r_k$ are all in R_v, too. Hence, each of the line segments $r_1, \ldots, r_k$ intersects R_v. Since none of them was chosen, s_i must come before any of them in π.

The probability that s_i comes before all of $r_1, \ldots, r_k$ is $1/(k+1)$. Hence,

$$E[C_{i,j}] = \text{Prob}(C_{i,j} = 1) \leq \frac{1}{ind(s_i, s_j) + 1},$$

and thus by (5.1) the expected number of leaves in T is at most

$$n + \sum_{1 \leq i,j \leq n} E[C_{i,j}] \leq n + \sum_{1 \leq i \leq n} \sum_{1 \leq j \leq n} \frac{1}{ind(s_i, s_j) + 1}.$$

For fixed i and any $1 \leq k \leq n-1$, there are at most two possible values for $j \in \{1, \ldots, n\}$ such that $ind(s_i, s_j) = k$ (extend the segment in both directions). We obtain

$$\begin{aligned} n + \sum_{1 \leq i \leq n} \sum_{1 \leq j \leq n} \frac{1}{ind(s_i, s_j) + 1} \leq n + \sum_{1 \leq i \leq n} \sum_{1 \leq k \leq n-1} \frac{2}{k+1} &= n + 2 \sum_{1 \leq i \leq n} \sum_{2 \leq k \leq n} \frac{1}{k} \\ &= n + 2 \sum_{1 \leq i \leq n} (H_n - 1) = O(n \log n). \end{aligned}$$

Hence, $O(n \log n)$ is an upper bound on the expected number of leaves in the tree T. □

6 Basics in Probability Theory III — Conditional Expectation

Sometimes we want to compute the expectation of one random variable, under the condition that another random variable has a certain value. For example, let X and Y be two discrete random variables. Then given that $Y = y$ for some value $y \in \mathbb{R}$, X has a different probability distribution (unless X and Y are independent). Hence, we obtain a new random variable $Z = (X|Y = y)$, where $\text{Prob}(Z = i) = \text{Prob}(X = i|Y = y)$. This way, the notion $E[X|Y = y]$ is just the expectation of Z, i.e.,

$$E[X|Y = y] = \sum_x x \cdot \text{Prob}(X = x|Y = y).$$

Lemma 6.1. *Let* (Ω, Prob) *be a probability space with two discrete random variables* $X : \Omega \to M$ *and* $Y : \Omega \to M'$*, where* $M, M' \subseteq \mathbb{R}$*. If* $E[X]$ *exists, then*

$$E[X] = \sum_{y \in M'} E[X|Y = y] \cdot \text{Prob}(Y = y).$$

Proof. Since $E[X]$ exists, the sums below converge absolutely. Applying Theorem 2.9, we obtain

$$\begin{aligned} E[X] &= \sum_{x \in M} x \cdot \text{Prob}(X = x) \\ &\overset{(2.1)}{=} \sum_{x \in M} x \cdot \sum_{y \in M'} \text{Prob}(X = x|Y = y) \cdot \text{Prob}(Y = y) \\ &= \sum_{y \in M'} \text{Prob}(Y = y) \sum_{x \in M} x \cdot \text{Prob}(X = x|Y = y) \\ &= \sum_{y \in M'} \text{Prob}(Y = y) E[X|Y = y]. \end{aligned}$$

□

7 Token Processes

7.1 A Simple Board Game

Consider the following game played on a board with n fields, labeled $0, \ldots, n-1$ from left to right. We place a token on the first field with number 0. In several rounds we move the token towards the right, until it falls off the board. The number of positions we are allowed to move the token in the i-th round is a random variable $X_i \in \{1, \ldots, n-1\}$, which is determined by some random experiment. Suppose we cannot make any assumption on the random variables X_i, for example we don't know whether they are independent. However, suppose we know an upper bound on the expected step size X_i. Can we conclude on a lower bound for the number of rounds we have to play this game, until the token falls off the board? Below we discuss a technique that allows us to elegantly analyze this simple board game, but that also has several applications beyond that.

7.2 Wald's Theorem

The following theorem is very useful for analyzing certain random experiments, such as the board game described above.

Theorem 7.1 (Wald's Theorem)**.** *Let $X_1, X_2, \ldots$ be a sequence of non-negative random variables and let $T \in \mathbb{N}$ be a random variable, such that the expectations of T and X_k exist for all k. Let $\sim$ be one of the relations $=, \leq, \geq, <, >$, and suppose that*

$$E[X_k | k \leq T] \sim \mu$$

for some $\mu \in \mathbb{R}$ and for all $k \geq 1$. Then

$$E[X_1 + \cdots + X_T] \sim \mu \cdot E[T].$$

Proof. Let $I_k = 1$ if $T \geq k$ and $I_k = 0$, otherwise. Since $E[X_i]$ exists (the sum is absolutely convergent) and $E[T]$ exists, obviously $E[X_1 + \cdots + X_T]$ exists, too. Therefore,

the sums in the following derivation are all absolutely convergent and can be reordered arbitrarily.

$$\begin{aligned} E[X_1 + \cdots + X_T] &= E\left[\sum_{k\in\mathbb{N}} X_k I_k\right] \\ &= \sum_{k\in\mathbb{N}} E[X_k I_k] \\ &= \sum_{k\in\mathbb{N}} E[X_k I_k | I_k = 0] \cdot \text{Prob}(I_k = 0) + E[X_k I_k | I_k = 1] \cdot \text{Prob}(I_k = 1) \end{aligned}$$

Clearly, $E[X_k I_k | I_k = 0] = E[0] = 0$, and since $I_k = 1$ iff $k \leq T$,

$$\begin{aligned} E[X_1 + \cdots + X_T] &= \sum_{k\in\mathbb{N}} E[X_k I_k | I_k = 1] \cdot \text{Prob}(I_k = 1) \\ &= \sum_{k\in\mathbb{N}} E[X_k \cdot 1 | I_k = 1] \cdot \text{Prob}(I_k = 1) \\ &= \sum_{k\in\mathbb{N}} E[X_k | k \leq T] \cdot \text{Prob}(k \leq T) \\ &= \sum_{k\in\mathbb{N}} E[X_k | k \leq T] \cdot \text{Prob}(k \leq T) \\ &\sim \sum_{k\in\mathbb{N}} \mu \cdot \text{Prob}(k \leq T) \\ &= \mu \sum_{k\in\mathbb{N}} \text{Prob}(k \leq T) \\ &= \mu \sum_{k\geq 0} \text{Prob}(T > k) \\ &= \mu \cdot E[T]. \end{aligned}$$

□

Now consider again the board game as described in the beginning of the previous section. Recall that X_i is the number of positions we can move the token forward in the i-th round and that the expected number of steps we are allowed to make is at most z. To be precise, we only know this for the step sizes as long as the game is not over, i.e., we have

$$E[X_i | i \leq T] \leq z.$$

Let T be the number of rounds until the token falls off the board. This means in T rounds the token is moved to the right by at least n positions in total. Hence, $X_1+\cdots+X_T \geq n$, and thus $E[X_1 + \cdots + X_T] \geq n$. Using Wald's Theorem with "$\leq$" for the relation "$\sim$", we get

$$n \leq E[X_1 + \cdots + X_T] \leq E[T] \cdot z$$

and thus

$$E[T] \geq n/z.$$

7.3 Jensen's Inequality

Another inequality that is often very useful is described by the following theorem. Recall that a function is convex if the graph of the function lies not above below the line segment joining any two points of the graph. More precisely, $f : \mathbb{R} \to \mathbb{R}$ is convex if for any $x_1, x_2 \in \mathbb{R}$ and $t \in [0, 1]$

$$f\big(t \cdot x_1 + (1-t)x_2\big) \leq t \cdot f(x_1) + (1-t) \cdot f(x_2)$$

(See Figure 7.3.)

Theorem 7.2. *Let X be a random variable and $f : \mathbb{R} \to \mathbb{R}$ a function such that $E[X]$ and $E[f(X)]$ exist. Then*

(a) $f(E[X]) \leq E[f(X)]$ if f is convex, and

(b) $f(E[X]) \geq E[f(X)]$ if f is concave.

Proof. Let $\mu = E[X]$. Let $\ell(x) = f(\mu) + \alpha(x - \mu)$ for some α. I.e., the line $\ell(x)$ goes through the point $\big(\mu, f(\mu)\big)$ and has slope α. Since f is convex, we can choose α such that $\ell(x) \leq f(x)$ for all $x \in \mathbb{R}$. (Line ℓ is a supporting line for $f(x)$; if f is differentiable, we can choose $\alpha = f'(\mu)$.) Then

$$E[f(X)] \geq E[\ell(X)] = E[f(\mu) + \alpha(X - \mu)] = f(\mu) + \alpha \cdot (E[X] - \mu) = f\big(E[X]\big).$$

The last equality follows from our choice of $\mu = E[X]$. □

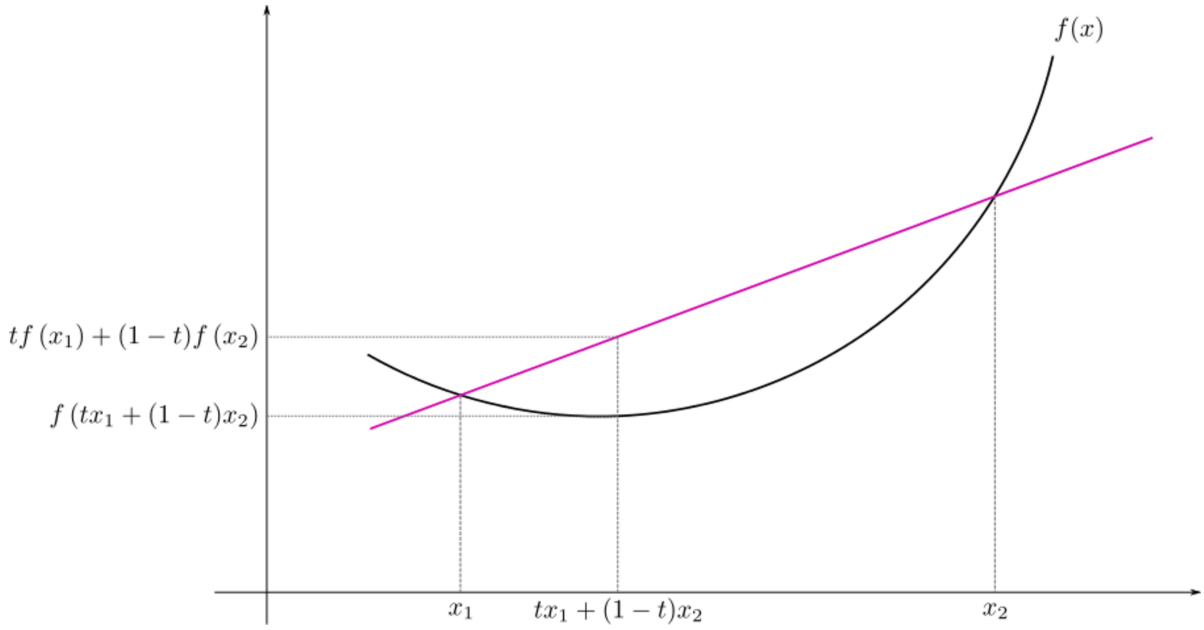


Figure 7.1: A convex function

Function Find(S, k)

Input: A set S of n integers, and an integer $k \in \{1, \ldots, n\}$.
Output: An integer $x \in S$.

1 Pick $y \in S$ uniformly at random.
2 **if** $|S| = 1$ **then return** y
3 $S_< := \{x \in S \mid x < y\}$
4 $S_> := \{x \in S \mid x > y\}$
5 **if** $|S_<| \geq k$ **then** `Find`$(S_<, k)$
6 **if** $|S_<| = k - 1$ **then** `Find` $(\{y\}, 1)$
7 **if** $|S_<| < k - 1$ **then** `Find` $(S_>, k - |S_<| - 1)$

Figure 7.2: Finding the element of a given rank.

7.4 Application: Finding an Element of a Given Rank

Let S be a set of n distinct integers. The *rank* of an element $x \in S$ is the integer k such that x is the k-th smallest element in S. I.e., if $S = \{s_1, \ldots, s_n\}$, where $s_1 < \cdots < s_n$, then s_i has rank i. The algorithm `Find` (see Figure 7.2) returns for a given set S and an integer $k \in \{1, \ldots, |S|\}$ the element of rank k in S.

The algorithm's correctness is clear. Often we have to analyze the number of recursive calls of an algorithm. What is the number of recursive calls of `Find`? Intuitively, during each recursive call the pivot element y should be somewhere in the "middle" of the sorted list of S. Thus, we conjecture that the expected number of recursive calls is $\Theta(\log n)$. In the following, we use Wald's Theorem to prove a lower bound of $\Omega(\log n)$.

Let $S_0 = S$ and $S_1, S_2, S_3, \ldots, S_T$, where $S_T = \{s_k\}$ be the sets used for the recursive calls. (I.e., S_i is the argument of the i-th recursive call.) Further, let $n_i = |S_i|$. Then $n_0 = n$ and $n_T = 1$. Finally, let

$$X_i = \log\left(\frac{n_{i-1}}{n_i}\right) = \log(n_{i-1}) - \log(n_i) \quad \text{for } 1 \leq i \leq T.$$

It is not hard to show that $E[X_i \,|\, i \leq T] = O(1)$:

Claim 7.3. *Consider an arbitrary set S of size $m \geq 2$ and let S' be the set of size m' obtained during the first recursive call of* `Find`(S, k). *Then*

$$E[\log(m/m')] = O(1).$$

A proof of this statement can be found at the end of this section. Hence, for some constant z,

$$E[X_i | i \leq T] \leq z. \tag{7.1}$$

On the other hand, since $n_0 = n$ and $n_T = 1$, we have

$$X_1 + \cdots + X_T = \log\left(\frac{n_0}{n_1}\right) + \cdots + \log\left(\frac{n_{T-1}}{n_T}\right) = \log\left(\frac{n_0}{n_1} \cdots \frac{n_{T-1}}{n_T}\right) = \log\left(\frac{n_0}{n_T}\right) = \log n.$$

In particular, this means that $E[X_1 + \cdots + X_T] = \log n$. We can now apply Wald's Theorem with (with "$\leq$" in place of "$\sim$"), and obtain

$$\log(n) = E[X_1 + \cdots + X_T] \leq E[T] \cdot z,$$

and so the expected number of recursive calls is

$$E[T] \geq \frac{\log n}{z} = \Omega(\log n).$$

Proof of Claim 7.3. If $i = k$, then $m' = 1$ and otherwise $m' \geq 1$. Thus, for $i \in \{1, k, m\}$, $\log(m/m') \leq \log(m)$ and for $i \notin \{1, k, m\}$:

$$\log(m/m') \leq \log\left(\max\{m/(i-1),\, m/(m-i)\}\right).$$

The probability that $i \in \{1, k, m\}$ is at most 3/m, and so we get

$$E\left[\log\left(\frac{m}{m'}\right)\right] \leq \frac{3 \cdot \log m}{m} + \sum_{1<i<m, i\neq k} \frac{1}{m} \cdot \log\left(\max\left(\frac{m}{i-1}, \frac{m}{m-i}\right)\right). \tag{7.2}$$

Clearly, $3 \cdot \log m/m = O(1)$, so it suffices to show that the second part of the above sum is $O(1)$. If i is an integer then $i - 1 < m - i$ if $i \leq m/2$, and $i - 1 \geq m - i$ if $i > m/2$. Hence, $\max\{m/(i-1), m/(m-i)\}$ equals $m/(i-1)$ for $i \leq m/2$ and $m/(m-i)$ for $i > m/2$. Thus we have

$$\begin{aligned}
&\sum_{1<i<m, i\neq k} \frac{1}{m} \cdot \log\left(\max\left(\frac{m}{i-1}, \frac{m}{m-i}\right)\right) \\
&\qquad \leq \sum_{1<i\leq m/2} \frac{\log\left(m/(i-1)\right)}{m} + \sum_{m/2<i<m} \frac{\log\left(m/(m-i)\right)}{m} \\
&\qquad \leq 2 \cdot \sum_{1\leq j<m} \frac{\log(m/j)}{m} \\
&\qquad \leq \frac{2}{m} \cdot \sum_{1\leq j<m} \lceil \log(m/j) \rceil.
\end{aligned}$$

For any $1 \leq r \leq \log m$, if $m/2^r \leq j < m/2^{r-1}$, then $2^{r-1} < m/j \leq 2^r$ and thus $\lceil \log(m/j) \rceil = r$. Hence, there are at most $m/2^r$ values of $j \in \{1, \ldots, m-1\}$ such that $\lceil \log(m/j) \rceil = r$. It follows that the sum above is bounded by

$$\frac{2}{m} \cdot \sum_{1\leq r\leq \log m} r \cdot \frac{m}{2^r} = 2 \cdot \sum_{1\leq r\leq \log m} \frac{r}{2^r} \leq 2 \cdot \sum_{r=1}^{\infty} \frac{r}{2^r} = O(1).$$

(For the last step we used the well-known fact that $\sum_{i=0}^{\infty} (i^a/b^i) = O(1)$ for any $a, b \in \mathbb{R}$, $b > 1$.) Now, from (7.2) it follows that $E[\log(m/m')] = O(1)$. □

7.5 A Probabilistic Recurrence

Our next goal is to find an upper bound on the number of recursive calls. For this we look at a different technique.

Observation 7.4. *Let $g : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, and X and Y be two discrete random variables. Further, assume that*

$$E[g(X,Y) \,|\, Y = y] \leq f(y)$$

for some function f. Then

$$E[g(X,Y)] \leq E[f(Y)].$$

(Note that $E[g(X,y)]$ depends only on the random variable X, while $E[g(X,Y)]$ depends on two random variables.)

Proof:

$$\begin{aligned}
E[g(X,Y)] &= \sum_{x,y} g(x,y) \cdot \text{Prob}(X = x \wedge Y = y) \\
&= \sum_{y} \sum_{x} g(x,y) \cdot \text{Prob}(X = x | Y = y) \cdot \text{Prob}(Y = y) \\
&= \sum_{y} \text{Prob}(Y = y) \sum_{x} g(x,y) \cdot \text{Prob}(X = x | Y = y) \\
&= \sum_{y} \text{Prob}(Y = y) E\big[g(X,Y) | Y = y\big] \\
&\leq \sum_{y} \text{Prob}(Y = y) f(y) \\
&= E[f(Y)].
\end{aligned}$$

Now consider the board game from the beginning of this section. For convenience we now start with the token at position $n-1$ and move it towards the 0. Assume that the token is in position m. We pick a random number $X_m \in \{1, \ldots, m-1\}$ according to some distribution, and then we move the token to position $m - X_m$.

Let T_n be the time it takes the token to reach space 1, when started in position n. What is an upper bound for T_n?

Of course we need some knowledge about the distribution used to pick the random numbers X_m. Assume that if the token is in position m, then the step size X_m has expectation $E(X_m) \geq g(m)$ for some non-decreasing function $g : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$

Theorem 7.5. $E[T_n] \leq \displaystyle\int_1^n \frac{dx}{g(x)}$.

Proof: Let $f(m) = \int_1^m \frac{dx}{g(x)}$.

We prove by induction on n that $E[T_n] \leq f(n)$. For $n = 1$, the $T_n = f(n) = 0$, so there is nothing to show.

Now assume that $n > 1$. In the first step, the token moves from position n to position $n - X_n$, where for the step size X_n it holds $E[X_n] \geq g(n)$. Thus,

$$E[T_n] = 1 + E[T_{n-X_n}] \tag{7.3}$$

Note that here the expectation is taken over the random variable "T_{n-X_n}", whose value depends on the outcome of two random experiments: The chosen step size, X_n, and the time it takes to go from space $n - X_n$ to space 1. Even once X_n is fixed to some value i, T_{n-i} is still a random variable whose value depends on future events.

But we know from the induction hypothesis that $E[T_{n-i}] \leq f(n-i)$ for all i. Hence, $E[T_n | X_n = i] \leq 1 + f(n-i)$. From Observation 7.4 it follows that

$$E[T_n| \leq E[1 + f(n - X_n)] = 1 + E[f(n - X_n)].$$

Hence,

$$\begin{aligned}
E[T_n] &\leq 1 + E[f(n - X_n)] \\
&= 1 + E\left[\int_1^{n-X_n} \frac{dx}{g(x)}\right] \\
&= 1 + E\left[\int_1^{n} \frac{dx}{g(x)} - \int_{n-X_n}^{n} \frac{dx}{g(x)}\right] \\
&= 1 + E\left[f(n) - \int_{n-X_n}^{n} \frac{dx}{g(x)}\right] \\
&\overset{\text{lin. of exp.}}{=} 1 + f(n) - E\left[\int_{n-X_n}^{n} \frac{dx}{g(x)}\right] \\
&\overset{g \text{ non-dec.}}{\leq} 1 + f(n) - E\left[\int_{n-X_n}^{n} \frac{dx}{g(n)}\right] \\
&= 1 + f(n) - \frac{E[n - (n - X_n)]}{g(n)} \\
&= 1 + f(n) - \frac{E[X_n]}{g(n)} \\
&\overset{E[X_n] \geq g(n)}{\leq} f(n).
\end{aligned}$$

■

Theorem 7.6. *The expected number of recursive calls of* `Find` *is at most* $4 \ln n$.

Proof. Consider a set S of size m obtained during a call of `Find`. Let S' be the set for which we call Find(S') recursively (if we don't call Find recursively, then $S' = \emptyset$). Thus, in one step the process replaces S with a set S', and the progress is $X_m = |S| - |S'| = m - |S'|$ for some random variable X_m. We show that $E[X_m] \geq m/4$. The theorem then follows from Theorem 7.5 for $g(m) = m/4$, because for the expected number of recursive calls, T_n,

$$E[T_n] \leq \int_1^n \frac{dx}{x/4} = 4 \cdot \int_1^n \frac{dx}{x} = 4 \cdot \big(\ln(n) - \ln(1)\big) = 4 \cdot \ln n.$$

Hence, it remains to show that $E[X_m] \geq m/4$. When does X_m have a value of i, $i \in \{1, \ldots, m-1\}$? Assume that the pivot y has rank r, $1 \leq r \leq m$. Then $|S_1| = r - 1$ and $|S_2| = m - r - 1$. If $r = k$, then $S' = \{y\}$, and thus $X_m = m - 1$. If $r < k$, then $S' = S_2$ and thus $X_m = m - |S_2| = r + 1$. If $r > k$, then $S' = S_1$ and thus $X_m = m - |S_1| = m - r + 1$.

For any $r \in \{1, \ldots, m\}$, the pivot y has rank r with probability $1/m$ Hence, we obtain

$$\begin{aligned} E[X_m] &= \frac{m-1}{m} + \sum_{r=1}^{k-1} \frac{r}{m} + \sum_{r=k+1}^{m} \frac{m-r+1}{m} = \frac{m - 1 + \sum_{r=1}^{k-1} r + \sum_{i=1}^{m-k} i}{m} \\ &\geq \frac{m - 1 + (k-1)k/2 + (m-k)(m-k+1)/2}{m} \geq \frac{k^2 + (m-k)^2}{2m}. \quad (7.4) \end{aligned}$$

Note that $a^2 + b^2$ with $a + b = c$ is minimal if $a = b = c/2$. Thus, $a^2 + b^2 \geq 2((a+b)/2)^2 = (a+b)^2/2$. We obtain from (7.4)

$$E[X_m] \geq \frac{m^2/2}{2m} = \frac{m}{4}.$$

□

7.6 Another Token Process and the Trick of Logarithms

In the following we give another example of a token process. To analyze it we use the techniques discussed in the previous sections.

We maintain a population of objects, and in subsequent rounds we decrease the size of the population. In the beginning we have a population of n objects. If at the beginning

of some round we have m objects, during the round we decide independently for each object in the population whether we throw it out or not. More precisely, each object remains in the population only with probability $1/\sqrt{m}$. Hence, given that a round starts with a population of size m, the expected size of the population after the round is $m/\sqrt{m} = \sqrt{m}$.

We could simulate this with a token process on a board with n spaces, where from a space m the token moves to a new space $m - X_m$, where $E[X_m] = m - \sqrt{m}$. However, this is a bit difficult to analyze. Instead, we use a logarithmic scale, and keep track of the logarithm of the size of the population.

Thus, the token process is played on the set of real numbers in $\log 1, \log 2, \ldots, \log n$. We start at position $\log n$. A position ℓ means that we have a population of size $m = 2^\ell$. Thus, we move to position $\ell' = \log(2^\ell - X_{2^\ell})$. Note that

$$E[X_{2^\ell}] = 2^\ell - \sqrt{2^\ell} = 2^\ell - 2^{\ell/2}. \tag{7.5}$$

We are interested in the expected step size of this new token process. Hence, if we are in position ℓ and move to ℓ', what is $E[\ell - \ell']$? Let $Z_\ell = \ell - \ell'$ denote that progress, given that we start in position ℓ. (I.e., ℓ is fixed and ℓ' is a random variable.) Then $E[Z_\ell] = E[\ell - \ell'] = \ell - E[\ell']$. Recall that $\ell' = \log(2^\ell - X_{2^\ell})$. Hence, $E[Z_\ell] = \ell - E[\log(2^\ell - X_{2^\ell})]$. Since the mapping $x \mapsto \log x$ is concave we have by Jensen's Inequality:

$$E[\log(2^\ell - X_{2^\ell})] \leq \log\left(E[2^\ell - X_{2^\ell}]\right) = \log\left(2^\ell - E[X_{2^\ell}]\right) \overset{(7.5)}{=} \log(2^\ell - (2^\ell - 2^{\ell/2})) = \ell/2.$$

Hence, $E[Z_\ell] \geq \ell/2$. We use Theorem 7.5 with $g(x) = \ell/2$, which is non-decreasing. Then the expected time until the token reaches position 1 is at most

$$\int_1^{\log n} \frac{dx}{g(x)} \leq \int_1^{\log n} \frac{dx}{x/2} = \Big[2 \ln x\Big]_{x=1}^{\log n} = O(\log \log n).$$

8 Tail Bounds

8.1 Markov's Inequality

Theorem 8.1 (Markov Inequality)**.** *Let $X : \Omega \to M$ be a random variable over the probability space Ω, where $M \subseteq \mathbb{R}_{\geq 0}$ is a countable set. If $E[X]$ exists, then for all $t \in \mathbb{R}_{>0}$,*

$$\text{Prob}(X \geq t) \leq \frac{E[X]}{t}.$$

Proof. In the following we sum only over values of i in M. Since $E[X]$ exists, these sums converge absolutely.

$$\begin{aligned}\text{Prob}(X \geq t) = \sum_{i \geq t} \text{Prob}(X = i) \leq \sum_{i \geq t} \frac{i}{t} \cdot \text{Prob}(X = i) &\leq \sum_{i \in M} \frac{i}{t} \cdot \text{Prob}(X = i) \\ &= \frac{1}{t} \cdot \sum_{i \in M} i \cdot \text{Prob}(X = i) = \frac{E[X]}{t}.\end{aligned}$$

□

Consider for example the algorithm `Find()` from Section 7.4, which finds in a set S of size n the element of a given rank, k. We have proved that the expected running time of that algorithm is $\Theta(\log n)$. Let T be the random variable that denotes the time it takes to execute `Find()` for some input of size n. Hence, $E[T] = O(\log n)$. By Markov Inequality we know that

$$\text{Prob}\big(T \geq (\log n)^2\big) \leq \frac{E[T]}{(\log n)^2} = O\left(\frac{1}{\log n}\right).$$

Hence, the probability that the `Find()` algorithm does not terminate after $\log^2 n$ steps is bounded by $O(1/\log n)$.

Often, Markov Inequality does not give tight bounds. For example, it is not hard to see that the `Find()` algorithm never requires more than n recursive calls. Hence,

$\text{Prob}(T \geq n^2) = 0$ (for large enough n). On the other hand, if we apply Markov Inequality, the best bound we get is

$$\text{Prob}\big(T \geq n^2\big) \leq \frac{E[T]}{n^2} = O\bigg(\frac{\log n}{n^2}\bigg).$$

8.2 Chernoff Bounds

Definition 8.2. A random variable X is *binomially distributed* with parameters $n \in \mathbb{N}$ and $q \in [0,1]$, if

$$\text{Prob}(X = k) = \binom{n}{k} q^k (1-q)^{n-k}. \tag{8.1}$$

Example 8.3. Let $0 \leq q \leq 1$ and $n \in \mathbb{N}$. Consider the set $[n] := \{1, \ldots, n\}$. We can sample a set $S \subseteq [n]$ at random as follows: For each element $i \in [n]$ we independently decide whether i is added to S or not, and the probability that i is added to S is q.

Let $X = |S|$. Then X is binomially distributed with parameters n and q: For each fixed set $R \subseteq [n]$ with $|R| = k$, the probability that $S = R$ is exactly $q^k(1-q)^{n-k}$ (the k elements in R have been chosen for S and the $n-k$ elements not in R have not been chosen for S). Hence, the number of all $\binom{n}{k}$ subsets of $[n]$ with size k is the term in (8.1).

Often we are only interested in upper bounds on the probability that a binomially distributed random variables deviates from its expectation. I.e., if X is binomially distributed, we would like to obtain upper bounds on the probability that $X < (1-\delta)E[X]$ or $X > (1+\delta)E[X]$ for small values of δ. Chernoff Bounds allow us to obtain such upper bounds.

A sequence of random variables $X_1, \ldots, X_n$ is called *Poisson trials*, if X_i takes only the values 0 or 1. Poisson trials $X_1, \ldots, X_n$ are called *Bernoulli trials*, if there is a p such that $\text{Prob}(X_i = 1) = p$ for all $1 \leq i \leq n$.

Theorem 8.4. *Let $X_1, \ldots, X_n$ be independent Poisson trials, $X = \sum_{1 \leq i \leq n} X_i$, and $\mu = E[X]$. Then for $\delta > 0$, $0 < \epsilon \leq 1$, and for $D \geq 6\mu$ it holds*

$$\text{Prob}(X \geq (1+\delta)\mu) \leq \frac{e^{\delta \cdot \mu}}{(1+\delta)^{(1+\delta)\mu}} \tag{8.2}$$

$$\text{Prob}(X \geq (1+\epsilon)\mu) \leq e^{-\mu\epsilon^2/3} \tag{8.3}$$

$$\text{Prob}(X \geq D) \leq 2^{-D}. \tag{8.4}$$

Proof: Let $p_i = E[X_i] = \text{Prob}(X_i = 1)$. Further, let $t > 0$ – we pick the "best t" later. Finally, let $Y_i = e^{tX_i}$ for $1 \leq i \leq n$, and $Y = e^{tX}$. Then

$$E[Y_i] = e^{t\cdot 1}\text{Prob}(X_i = 1) + e^{t\cdot 0}\text{Prob}(X_i = 0) = p_i e^t + 1 - p_i = 1 - p_i(1 - e^t)$$

Note that $1 + x \leq e^x$ for all x. We apply this inequality for $x = -p_i(1 - e^t)$ and obtain

$$E[Y_i] = 1 - p_i \cdot (1 - e^t) \leq \exp\big(-p_i \cdot (1 - e^t)\big) = \exp\big(p_i \cdot (e^t - 1)\big).$$

Since $X_1, \ldots, X_n$ are independent, so are $Y_1, \ldots, Y_n$. Hence, for $Y = Y_1 \cdot \cdots \cdot Y_n$ we have

$$\begin{aligned} E[Y] = e^{t(X_1+\cdots+X_n)} = E\left[\prod_{1\leq i\leq n} Y_i\right] = \prod_{1\leq i\leq n} E[Y_i] \leq \prod_{1\leq i\leq n} \exp\big(p_i \cdot (e^t - 1)\big) \\ = \exp\big((e^t - 1)(p_1 + \cdots + p_n)\big) = \exp\big((e^t - 1)\mu\big). \quad (8.5) \end{aligned}$$

By applying Markov's inequality (MI):

$$\text{Prob}(X \geq (1+\delta)\mu) = \text{Prob}\big(Y \geq e^{t(1+\delta)\mu}\big) \overset{\text{(MI)}}{\leq} \frac{E[Y]}{e^{t(1+\delta)\mu}} \leq \frac{e^{(e^t-1)\mu}}{e^{t(1+\delta)\mu}}.$$

We can now choose t, depending on δ. Let $t = \ln(1+\delta)$ (> 0), and recall that $1 + x < e^x$ for all $x \neq 0$. Then

$$\text{Prob}(X \geq (1+\delta)\mu) \leq \frac{e^{(1+\delta-1)\cdot\mu}}{(1+\delta)^{(1+\delta)\mu}} = \frac{e^{\delta\cdot\mu}}{(1+\delta)^{(1+\delta)\mu}}.$$

This is (8.2).

Using basic calculus, it is not hard to see that for $0 < \delta \leq 1$

$$\frac{e^\delta}{(1+\delta)^{1+\delta}} \leq e^{-\delta^2/3}.$$

This implies (8.3).

In order to show (8.4) let $D \geq 6\mu$. Let $\delta = D/\mu - 1 \geq 5$, thus $D = (1+\delta)\mu$. Hence, by (8.2)

$$\text{Prob}(X \geq D) = \text{Prob}(X \geq (1+\delta)\mu) \leq \frac{e^{\delta\mu}}{(1+\delta)^{(1+\delta)\mu}} \leq \left(\frac{e}{1+\delta}\right)^{(1+\delta)\mu} \leq \left(\frac{e}{6}\right)^D \leq 2^{-D}.$$

□

Theorem 8.5. *Let $X_1, \ldots, X_n$ be independent Poisson trials, $X = \sum_{1 \le i \le n} X_i$, and $\mu = E[X]$. Then for $0 < \delta < 1$*

$$\text{Prob}(X \le (1-\delta)\mu) \le \frac{e^{-\delta \cdot \mu}}{(1-\delta)^{(1-\delta)\mu}} \tag{8.6}$$

$$\text{Prob}(X \le (1-\delta)\mu) \le e^{-\mu\delta^2/2}. \tag{8.7}$$

The proof is very similar to the one of Theorem 8.4.

Example: Balls Into Bins. Suppose we throw m balls into n bins, and for each ball we choose its bin uniformly at random, and independent from all other ball/bin pairings. Clearly, the expected load of each bin will be m/n. What is the probability that some bin has a load that deviates significantly from this? Using Chernoff Bounds, upper bounds for such events are relatively easy to obtain.

For now consider the first bin only. Let X_i be the indicator variable, where $X_i = 1$ if and only if the i-th ball is thrown into the first bin. Then $X = \sum_{1 \le i \le n} X_i$ is the number of balls that end up in the first bin. Moreover, $X_1, \ldots, X_m$ are independent Poisson trials, and $\text{Prob}(X_i = 1) = 1/n$ for $1 \le i \le n$. Hence, $\mu := E[X] = m/n$.

Now let for example $m = n$ and c be an arbitrary constant such that $c \cdot \log n \ge 6$. Using (8.4)

$$\text{Prob}(X \ge c \cdot \log n \cdot \mu) \le 2^{-c \log n} = n^{-c}.$$

The probability that there is a bin which has more than $c \log n \cdot m/n$ balls is at most n^{-c+1} (by simply summing up the probabilities n^{-c} for each bin).

Chernoff Bounds can sometimes be weaker than good ad-hoc analyses. For example, let $m = c \cdot \ln n$. What is the probability that the first bin is empty?

$$\text{Prob}(X = 0) = \text{Prob}(X \le (1-1)\mu) \le e^{-m/(2n)} = e^{-(c/2)\ln n} = n^{-c/2}.$$

Hence, the probability that there exists an empty bin is at most $n^{-c/2-1}$. Later, we use an "ad-hoc" analysis to prove a bound of n^{-c-1}.

9 Randomized Complexity Classes

9.1 Las Vegas and Monte Carlo Algorithms

A *Las Vegas algorithm* is a randomized algorithm which always outputs the correct result, but the running time may be a random variable. An example of a Las Vegas algorithm is Randomized Quicksort.

A *Monte Carlo algorithm* is a randomized algorithm which may err, depending on its random choices. An example is the Max2Cut algorithm from the previous section.

For decision problems (which output either 0 or 1), there are three kinds of Monte Carlo algorithms:

1. **Two-sided error:** May err on any input.
2. **One-sided error, false negatives:** May err only on 1-inputs. I.e., the algorithm may output 0 (negative), if the true result is 1 (positive). This means, if the algorithm outputs 1, then we know the answer is correct.
3. **One-sided error, false positives:** May err only on 0-inputs.

For example, every decision problem in NP has a randomized Monte Carlo algorithm with one-sided error: A nondeterministic algorithm can be turned into a polynomial time "randomized" algorithm with one-sided error, false negatives. Consider for example the Hamiltonian Path problem, in which we want to find out whether a given graph G (with n vertices) has a Hamiltonian path. A randomized algorithm could just randomly generate one of the $n!$ permutations, π, and then verify (in polynomial time) whether π corresponds to a Hamiltonian path in G. If yes, the algorithm accepts, otherwise it rejects.

If the algorithm accepts, then it must have guessed one of the right permutations, and its output is correct. On the other hand, if the algorithm rejects, this only proves that the permutation the algorithm picked does not correspond to a Hamiltonian path. Therefore, the algorithm may err, but only with false negatives. Moreover, the error probability is bounded: If the graph has a Hamiltonian path, the probability that the algorithm finds one is at least $1/n!$.

In general, NP is exactly the class of decision problems for which there is a randomized Monte Carlo algorithm that solves that problem in polynomial time with one-sided (false-negatives) error, and error probability less than 1.

9.2 Probabilistic Turing Machines

See Section 1.5.1. in the textbook.

9.3 Complexity Classes

See Section 1.5.2 in the textbook.

Denote by $\mathrm{RP}(\epsilon(n))$ and $\mathrm{BPP}(\epsilon(n))$ the class of languages L that have a polynomial time algorithm A, such that for any input of length n the error probability is at most $\epsilon(n)$. In case of $\mathrm{RP}(\epsilon(n))$ the algorithm has a one-sided (false-negatives) error, in the case of $\mathrm{BPP}(\epsilon(n))$, the error can be two-sided. Similarly, let $\mathrm{ZPP}(\epsilon(n))$ be the class of languages that have a polynomial time algorithm that either accepts, rejects, or returns "?". Unless the algorithm returns "?", the answer is always correct, and the probability that A returns "?" is at most $\epsilon(n)$.

Then $\mathrm{RP} = \mathrm{RP}(1/2)$, $\mathrm{BPP} = \mathrm{BPP}(1/4)$, and $\mathrm{ZPP} = \mathrm{ZPP}(1/2)$. Moreover,

$$\mathrm{NP} = \bigcup_{\epsilon(n)<1} \mathrm{RP}\big(\epsilon(n)\big) \quad \text{and} \quad \mathrm{PP} = \bigcup_{\epsilon(n)<1/2} \mathrm{BPP}\big(\epsilon(n)\big)$$

9.4 Amplifying Failure and Error Probabilities

Theorem 9.1. *For any two polynomials p and q,*

$$\mathrm{RP} = \mathrm{RP}(1 - 1/p(n)) = \mathrm{RP}(1/2^{q(n)}).$$

Proof. It suffices to prove that $\mathrm{RP}(1 - 1/p(n)) \subseteq \mathrm{RP}(1/2^{q(n)})$. Let $L \in \mathrm{RP}(1 - 1/p(n))$ and let A be an algorithm with one-sided error probability at most $1 - 1/p(n)$. For an input x of length n, we can simulate A $t(n)$ times using independent coin flips. As soon as A accepts, we accept. If A rejects the input every time, we reject as well.

Let $t(n) = \lceil (\ln 2) \cdot p(n) \cdot q(n) \rceil$. If $x \notin L$, then A rejects every time, so we will reject in the end. If $x \in L$, then the probability that A rejects every time is

$$(1 - 1/p(n))^{\lceil (\ln 2) \cdot p(n) \cdot q(n) \rceil} \leq e^{-(\ln 2) q(n)} = 2^{-q(n)}.$$

□

Theorem 9.2. *For any two polynomials p and q,*

$$\mathrm{ZPP} = \mathrm{ZPP}\big(1 - 1/p(n)\big) = \mathrm{ZPP}\big(2^{-q(n)}\big).$$

We leave the proof as an exercise.

We can use Chernoff Bounds to show that the error probability of polynomial time algorithms with two-sided error can be amplified from "polynomially close to 1/2" to "exponentially small".

Theorem 9.3. *For any two constants $c, d > 0$,*

$$\mathrm{BPP}\big(1/2 - 1/n^c\big) = \mathrm{BPP}\Big(1/2^{n^d}\Big) = \mathrm{BPP}.$$

Proof. It suffices to show $\mathrm{BPP}\big(1/2 - n^c\big) \subseteq \mathrm{BPP}\Big(1/2^{n^d}\Big)$. Let $L \in \mathrm{BPP}\big(\epsilon(n)\big)$ and let A be a polynomial time algorithm for L with error probability at most $\epsilon(n) \leq 1/2 - 1/n^c$.

Now consider an algorithm B that works on input x as follows: B simulates $t(n)$ runs of A on the input x, using independent random bits for each simulation. Then B makes a majority decision, i.e., it returns the same answer as the majority of the outcomes of A. (Ties can be broken arbitrarily.)

Consider an input x of length n and let $t := t(n)$ and let $s := s(x)$ be the probability that A gives the correct answer for input x. Then

$$s \geq 1 - \epsilon(n) \geq \frac{1}{2} + \frac{1}{n^c}.$$

Let X_i be the indicator random variable, where $X_i = 1$ if and only if the i-th run of A returns the correct result, and let $X = X_1 + \cdots + X_t$. Then the random variables $X_1, \ldots, X_t$ are independent and $E[X_i] = \mathrm{Prob}(X_i = 1) = s$ for all $1 \leq i \leq t$. For $\mu = E[X] = s \cdot t$, we will apply the Chernoff Bound from Theorem 8.5,

$$\mathrm{Prob}(X \leq (1 - \delta)\mu) \leq e^{-\mu\delta^2/2}. \tag{9.1}$$

We need to bring "Prob$(X \leq t/2)$" in the required form:

$$\text{Prob}(X \leq t/2) = \text{Prob}(X \leq \mu \cdot 1/(2s)) = \text{Prob}\big(X \leq \mu(1 - (1 - 1/(2s)))\big)$$

Thus, for $\delta = (1 - 1/(2s))$, inequality (9.1) yields

$$\text{Prob}(X \leq t/2) \leq e^{-\mu\delta^2/2} = e^{-t \cdot s \cdot \delta^2/2}$$

By the definition of δ,

$$s \cdot \delta^2/2 = s(1 - 1/(2s))^2/2.$$

Since this term increases with s and $s \geq 1/2 + 1/n^c$, the value of $s \cdot \delta^2/2$ is minimal for $s = 1/2 + 1/n^c$. For this value of s we obtain

$$\delta = 1 - 1/(2s) = 1 - \frac{1}{1 + 2/n^c} = \frac{2/n^c}{1 + 2/n^c} = \frac{2}{n^c + 2}$$

and thus

$$s \cdot \delta^2/2 \geq \left(\frac{1}{2} + \frac{1}{n^c}\right)\frac{2}{(n^c + 2)^2} \geq \frac{1}{2n^{2c}}.$$

Hence, for $t = t(n) = \left\lceil 2(\ln 2) \cdot n^{2c+d} \right\rceil$, we obtain

$$\text{Prob}(X \leq t/2) \leq \exp\left(\frac{-t}{2n^{2c}}\right) \leq \exp\left(-(\ln 2) \cdot n^d\right) = 2^{-n^d}.$$

Thus, for this choice of $t(n)$, the error probability of our algorithm B is at most $1/2^{n^d}$. Moreover, since $t(n)$ is a polynomial, the running time of algorithm B is polynomial. □

9.5 Relations Between Complexity Classes for Decision Problems

Problems in PP are not in general efficiently computable, or at least we don't know that all problems in PP have efficient randomized algorithms with acceptable error probabilities. In fact, NP is a subset of PP, so PP contains NP-complete problems.

Theorem 9.4. NP $\subseteq$ PP

Proof. Let $L \in$ NP. Then there exists a polynomial time randomized Turing machine M, such that if $x \in L$, then M accepts with probability larger than 0, and if $x \notin L$, then M rejects with probability 1. Recall that in each step, a Turing machine can either apply transition function δ_0 or δ_1, so it can use exactly one random bit per step. Let $t_M(n)$ be the worst-case the running time of M, i.e., $t_M(n) = n^{O(1)}$.

We construct a PP-algorithm B for L. Let x be some input of length n, and $t = t_M(n)$. First, B chooses an integer $z \in \{0, \ldots, 2^{t+1} - 1\}$ uniformly at random, by sampling $t+1$ random bits. If $0 \leq z \leq 2^t$, then B simulates M and returns M's answer. Otherwise, B accepts.

We prove that B has an error probability of less than $1/2$. First consider the case $x \notin L$. If B simulates M, then M rejects and so B rejects. If B does not simulate M, then B accepts. So B is correct if and only if it simulates M, i.e., if $z \leq 2^t$. This happens with probability

$$\text{Prob}\big(z \leq 2^t\big) = \frac{2^t + 1}{2^{t+1}} = \frac{1}{2} + \frac{1}{2^{t+1}} > 1/2.$$

Now suppose $x \in L$. Then B accepts if $z > 2^t$, or if $z \leq 2^t$ and M accepts. Recall that M uses at most t random bits. For at least one combination of these random bits, M has to accept. There are at most 2^t possible bit combinations, so Prob(M accepts) $\geq 1/2^t$. We obtain

$$\begin{aligned}
\text{Prob}(B \text{ accepts}) &= \text{Prob}\big(z > 2^t\big) + \text{Prob}\big(z \leq 2^t \wedge M \text{ accepts}\big) \\
&\geq \frac{1}{2} - \frac{1}{2^{t+1}} + \left(\frac{1}{2} + \frac{1}{2^{t+1}}\right)\frac{1}{2^t} \\
&= \frac{1}{2} - \frac{1}{2^{t+1}} + \frac{1}{2^{t+1}} + \frac{1}{2^{2t+1}} \\
&> \frac{1}{2}.
\end{aligned}$$

□

Recall that ZPP algorithms cannot err, but they can fail, and return no answer. We will now show that problems in ZPP have efficient Las Vegas algorithms.

Theorem 9.5. *If $L \in$ ZPP, then and only then there exists a randomized algorithm A for L that has expected polynomial running time and always outputs the correct answer.*

Proof. "$\Leftarrow$": Suppose A is an algorithm that solves L without failure or error in expected running time $t_A(n)$, where $t_A(n) \leq p(n)$ for some polynomial p. Let A' be an algorithm that for any input x simulates A on x until A halts or until $2 \cdot p(n)$ steps have been simulated. If A halts within $2p(n)$ steps, A' returns the same result as A; otherwise A' returns "?". Now consider an arbitrary input x of length n. If the simulation of A on x halts within $2p(n)$ steps, then A' returns the correct answer, and otherwise A' fails. Thus, we have to prove that the probability that A needs more than $2p(n)$ steps is at most $1/2$. Let Y be the random variable that is the running time of A for the input x. We know that $E[Y] \leq p(n)$. By Markov Inequality (Theorem 8.1),

$$\text{Prob}\big(Y \geq 2p(n)\big) \leq \frac{E[Y]}{2p(n)} \leq \frac{p(n)}{2p(n)} = \frac{1}{2}.$$

Hence, the probability that A' fails is at most $1/2$.

"$\Rightarrow$": Let $L \in$ ZPP and B an algorithm that solves L in worst-case running time $t_B(n) \leq r(n)$ for some polynomial r, and that fails with probability at most $1/2$. Let B' be the algorithm which, on the input x, runs B on x repeatedly until B provides an answer "yes" or "no". If B returns an answer, then B' returns the same answer. Clearly, B' always outputs the correct answer, so it suffices to show that $E[t_{B'}(x)] = n^{O(1)}$, where n is the length of the input x. For each run of B, the probability that B does not fail is $q \geq 1/2$. Hence, the number of repetitions, X, is a geometrically distributed random variable with parameter q. Thus, $E[X] = 1/q \leq 2$. Hence, the expected running time of B' on input x is

$$E[t'_B(x)] = E[X \cdot t_{B'}(n)] \leq E[X \cdot r(n)] = E[X] \cdot r(n) \leq 2 \cdot r(n).$$

□

Theorem 9.6. ZPP $\subseteq$ RP

Proof. Let $L \subseteq \Sigma^*$ be a language in ZPP and let A be a ZPP-algorithm for L. Then A halts on any input in polynomial time. With probability at most $1/2$ it fails (i.e., outputs "?"), but if it does not fail, then its answer (accept or reject) is correct.

Prob. accept	$x \in L$	$x \notin L$
A	$\geq 1/2$	0
A'	1	$\leq 1/2$

Figure 9.1: Error modes for an RP algorithm A and a co-RP algorithm A'.

We construct an RP-algorithm A'. A' simulates A on the input x, and if A outputs "?", then A' rejects. Then clearly, A' also has polynomial running time.

If $x \notin L$, then A either rejects or fails. In either case, A' rejects, so with probability 1 the result is correct. If $x \in L$, then A accepts with probability at least $1/2$ and otherwise fails. If A accepts, then A' gives the correct answer, otherwise it errs. Hence, A' errs with probability at most $1/2$. Moreover, if it errs, it rejects, so it can only give false negatives. It follows that A' is an RP-algorithm for L. □

Theorem 9.7. $\mathrm{RP} \cap \text{co-RP} \subseteq \mathrm{ZPP}$.

Proof. Let $L \subseteq \Sigma^*$ be a language in $\mathrm{RP} \cap \text{co-RP}$. Let A and A' be RP and co-RP-algorithms for L, respectively. I.e., both algorithms have polynomial worst-case running times, both have a one-sided error probability of at most $1/2$, and all errors are false-negatives in the case of A and false-positives in the case of A'.

Here is a ZPP-algorithm B that decides L for the input x: Run first A and then A' on x. If both, A and A', give the same answer, return that answer. Otherwise, output "?".

Clearly, B has polynomial running time. Hence, it suffices to show that B makes no error, and that the probability that B outputs "?" is at most $1/2$.

If B accepts, then this means that A and A' accepted x and thus the answer is correct, as A cannot give a false-positive answer. Similarly, if B rejects, then A and A' rejected x, which is correct because A' cannot give a false-negative answer. This implies that if B accepts or rejects, its answer is correct.

It remains to bound the probability that B outputs "?". Let $A(x)$, $A'(x)$, and $B(x)$ denote the (random) outputs of a run of A, A', and B, respectively. First note that it is not possible that A accepts and A' rejects, as this would imply that either A gives a false-positive or A' a false-negative answer, both of which are impossible. Hence, B outputs "?" if and only if A rejects and A' accepts.

Note that if an event E implies some other event, D (ie., $E \subseteq D$), then $\mathrm{Prob}(E) \leq \mathrm{Prob}(D)$. Since the event "$B(x) =$ "?"" implies the events "$A(x) = 0$" and "$A'(x) = 1$"

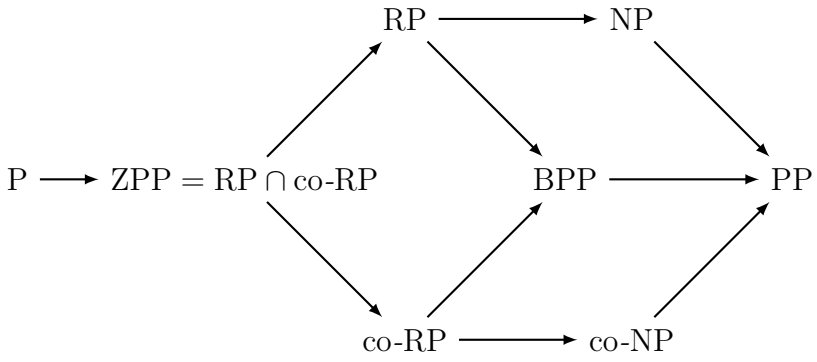
accepts, we have

$$\text{Prob}(B(x) = \text{“?”}) \leq \text{Prob}(A(x) = 0) \quad \text{and} \quad \text{Prob}(B(x) = \text{“?”}) \leq \text{Prob}(A(x) = 1)$$

If $x \in L$, then $\text{Prob}(A(x) = 0) \leq 1/2$ and $\text{Prob}(A'(x) = 1) = 1$ (as A' cannot give a false-negative answer). Similarly, if $x \notin L$, then $\text{Prob}(A(x) = 0) = 1$ and $\text{Prob}(A'(x) = 1) \leq 1/2$. Hence, in either case $\text{Prob}(B(x) = \text{“?”}) \leq 1/2$. □

Corollary 9.8. $\text{ZPP} = \text{RP} \cap \text{co-RP}$.

Corollary 9.9. *The following inclusions hold, where $A \to B$ means “$A \subseteq B$”:*



10 Chebyshev's Inequality and Two-Point Sampling

Recall that we can use Chernoff Bounds to bound the probability that a random variable X deviates from its expectation, if X is the sum of independent Bernoulli trials. In many cases, we do not have complete independence among Bernoulli trials. On the other hand, Markov Inequality makes no such assumptions on the distribution of X. However, the probability bounds we can achieve with Markov Inequality are often too weak.

10.1 Chebyshev's Inequality

We now present another tail inequality that is also useful for the sum of possion trials, but doesn't require that *all* trials are independent. In fact, we only need to be able to bound the *variance* of X, which we can often do effectively, if X is the sum of *pairwise independent* Bernoulli trials.

Definition 10.1. Let X be a random variable, and $\mu := E[X]$. Then the *variance* of X is

$$\mathrm{Var}\,[X] := E[(X-\mu)^2] \quad (= E[X^2] - \mu^2).$$

For example, let $X : \Omega \to \{0,1\}$ be an indicator random variable, where $p = \mathrm{Prob}(X = 1)$. Then $X^2 = X$, so $E[X^2] = E[X] = p$, and thus

$$\mathrm{Var}\,[X] = E[X^2] - E[X]^2 = E[X] - E[X]^2 = E[X](1 - E[X]) = p(1-p).$$

Theorem 10.2 (Chebyshev's Inequality)**.** *Let X be a random variable. Then for any $t \in \mathbb{R}_{\geq 0}$*

$$\mathrm{Prob}(|X - E[X]| \geq t) \leq \frac{\mathrm{Var}\,[X]}{t^2}.$$

Proof. Let $Y := (X - E[X])^2$. Then $E[Y] = \mathrm{Var}\,[X]$. Thus, by Markov inequality

$$\begin{aligned}\mathrm{Prob}(|X - E[X]| \geq t) &= \mathrm{Prob}\big((X - E[X])^2 \geq t^2\big) \\ &= \mathrm{Prob}\big(Y \geq t^2\big) \leq \frac{E[Y]}{t^2} = \frac{\mathrm{Var}\,[X]}{t^2}.\end{aligned}$$

□

One of the reasons why Chebyshev's Inequality is useful for algorithmic application is that Variance is linear for pairwise independent random variables.

Definition 10.3. The random variables $X_1, \ldots, X_k$ are *pairwise independent*, if for any $1 \leq i < j \leq k$ the random variables X_i and X_j are independent.

Say, $X_1, \ldots, X_k$ are indicator random variables, indicating whether the run of the i-th run of an algorithm was successful or not. If we want that all random variables $X_1, \ldots, X_k$ are independent, then we need to use independent random bits for each run of the algorithm. This may require a lot of randomness. In the next section we will see that with very little randomness we can ensure that $X_1, \ldots, X_k$ are at least pairwise independent. In order to apply Chebyshev's Inequality, we may need to bound the variance of the sum $X_1 + \cdots + X_k$. The next lemma implies that this is easy for pairwise independent random variables.

Lemma 10.4. *Let $X_1, \ldots, X_k$ be pairwise independent random variables and $X = \sum_{i=1}^{k} X_i$. Then* $\operatorname{Var}[X] = \sum_{i=1}^{k} \operatorname{Var}[X_i]$. □

Proof. By induction on k. For $k = 1$ there is nothing to show. Now assume $k \geq 2$.

Let $Y = X_1 + \cdots + X_{k-1}$. Then by the inductive hypothesis, $\operatorname{Var}[Y] = \sum_{1 \leq i < k} \operatorname{Var}[X_i]$.

Then

$$\sum_{1 \leq i \leq k} \operatorname{Var}[X_i] = \operatorname{Var}[Y] + \operatorname{Var}[X_k] = E[Y^2] + E[{X_k}^2] - E[Y]^2 - E[X_k]^2. \tag{10.1}$$

Moreover,

$$\begin{aligned}
\operatorname{Var}[X] &= \operatorname{Var}[Y + X_k] \\
&= E[(Y + X_k)^2] - (E[Y + X_k])^2 \\
&= E[Y^2 + 2YX_k + {X_k}^2] - E[Y]^2 - 2E[Y]E[X_k] - E[X_k]^2 \\
&= E[Y^2] + E[{X_k}^2] - E[Y]^2 - E[X_k]^2 - 2E[Y]E[X_k] + 2E[YX_k] \\
&\overset{(10.1)}{=} \sum_{1 \leq i \leq k} \operatorname{Var}[X_i] + 2(E[YX_k] - E[Y]E[X_k]).
\end{aligned}$$

So it suffices to prove that $E[Y] \cdot E[X_k] = E[YX_k]$.

By linearity of expectation

$$\begin{aligned}
E[Y] \cdot E[X_k] &= E[X_k] \cdot \sum_{1 \le i < k} E[X_i] \\
&= \left(\sum_{1 \le i < k} E[X_k] \cdot E[X_i] \right) \\
&\overset{\text{pairw. indep.}}{=} \left(\sum_{1 \le i < k} E[X_k \cdot X_i] \right) \\
&= E\left[\sum_{1 \le i < k} X_k \cdot X_i \right] \\
&= E\left[X_k \cdot \sum_{1 \le i < k} X_i \right] \\
&= E[X_k \cdot Y].
\end{aligned}$$

□

10.2 Error-Amplification with Low Randomness

How can we generate pairwise independent random variables with as few random bits as possible?

Theorem 10.5. *Let p be a prime. Then for randomly chosen $a, b \in \{0, \ldots, p-1\}$, the random variables $X_i := (a \cdot i + b) \bmod p$, $0 \le i < p$, are uniformly distributed and pairwise independent.*

Proof: Recall that $\mathbb{Z}_p = \{0, \ldots, p-1\}$, equipped with addition and multiplication modulo p, is a finite field. I.e., it is a group with respect to addition and with respect to multiplication. In the following, let $\equiv$ denote the equivalence relation "modulo p" on

$\mathbb{Z}$. Thus,

$$\begin{aligned}
\text{Prob}(X_i = y_i) &= \sum_{A \in \mathbb{Z}_p} \text{Prob}(X_i = y_i \mid a = A)] \cdot \text{Prob}(a = A) \\
&= \sum_{A \in \mathbb{Z}_p} \text{Prob}(Ai + b \equiv y_i) \frac{1}{p} \\
&= \frac{1}{p} \sum_{A \in \mathbb{Z}_p} \text{Prob}(b \equiv y_i - Ai) \\
&= \frac{1}{p} \sum_{A \in \mathbb{Z}_p} \frac{1}{p} \\
&= \frac{1}{p}.
\end{aligned}$$

This proves that X_i is uniformly distributed.

It remains to show for any $0 \leq i < j \leq p-1$ that the random variables X_i and X_j are independent. To do so, it suffice to show for any two values y_i, y_j that

$$\text{Prob}(X_i = y_i \mid X_j = y_j) = \text{Prob}(X_i = y_i).$$

We have already proved that the right side of the equation is $1/p$. Hence, it suffices to show that

$$\text{Prob}(X_i = y_i \mid X_j = y_j) = \frac{1}{p}.$$

Since

$$\text{Prob}(X_i = y_i \mid X_j = y_j) = \frac{\text{Prob}(X_i = y_i \wedge X_j = y_j)}{\text{Prob}(X_j = y_j)} = p \cdot \text{Prob}(X_i = y_i \wedge X_j = y_j),$$

it suffices to show that $\text{Prob}(X_i = y_i \wedge X_j = y_j) = 1/p^2$.

Consider the polynomials $f(x) = (ax + b) \bmod p$, for $a, b \in \mathbb{Z}_p$. These polynomials have degree one. Hence, for any $i \neq j$, if we fix $y_0 \in \mathbb{Z}_p$, and $y_1 \in \mathbb{Z}_p$, then there is exactly one such polynomial that satisfies $f(i) = y_0$ and $f(j) = y_1$. There are p^2 such polynomials, and so the probability that a randomly chosen polynomial f satisfies this constraint is $1/p^2$. □

We now want to amplify the error probability of an RP-algorithm, without using too many random bits.

Algorithm 7: A^*

Input: $x \subseteq \Sigma^*$, and an integer $1 \leq t < p$.

1 Randomly pick $a, b \in \mathbb{Z}_p$.
2 **for** $0 \leq i < t$ **do**
3 Let $r_i := a \cdot i + b$
4 Compute $A(x, r_i)$ by simulating A
5 **if** $A(x, r_i) = 1$ **then** accept
6 **end**
7 reject

Figure 10.1: Error-amplification with two-point sampling.

Consider an algorithm A for some language L, that has a one-sided (false-negatives) error probability of at most $1/2$. For simplicity reasons, assume that A uses exactly one random word $r \in \{0, \ldots, p-1\}$ for a run, where p is a prime. For an input x, we define $A(x, r) = 1$, if A accepts the input for the random word r, and $A(x, r) = 0$ if A rejects. Hence, we know that A has polynomial running time, and

$$x \in L \Rightarrow \text{Prob}_r(A(x, r) = 1) \geq \frac{1}{2}, \quad \text{and}$$
$$x \notin L \Rightarrow \text{Prob}_r(A(x, r) = 0) = 1.$$

If we repeat the algorithm t times and use *independent* random choices for every repetition, we can decrease the error probability to $1/2^t$. However, then we need at least t random words.

Alternatively, consider the algorithm in Figure 10.1. Clearly, this algorithm needs only two random words. If $x \notin L$, then A will reject in each run, and thus A^* rejects, too. It therefore remains to consider the error probability for the case $x \in L$.

If the numbers r_i were chosen *independently*, then this probability would be at most $1/2^t$. In our case, these numbers are only *pairwise* independent. Let X_i be the indicator variable, where $X_i = 1$ if and only if $A(x, r_i)$ accepts. Since the integers r_i are pairwise independent, so are the random variables X_i. Moreover, all random variables X_i have the same distribution, i.e.,

$$\text{Prob}(X_i = 1) = \text{Prob}(A(x, r_i) \text{ accepts}) = q$$

for some parameter $q \geq 1/2$. Thus, $E[X_i] = q \geq 1/2$, and

$$\text{Var}[X_i] = q(1-q) \leq 1/4. \tag{10.2}$$

We let $X = X_0 + \cdots + X_{t-1}$ be the number of runs in which A accepts. Obviously, our new algorithm A^* accepts (correctly), iff $X \geq 1$. Hence, we have to bound the probability that $X = 0$. We know that $E[X] = t \cdot E[X_i] \geq t/2$, and by pairwise independence of the random variables X_i and (10.2)

$$\text{Var}[X] = \sum_{1 \leq i \leq t} \text{Var}[X_i] \leq t/4.$$

Applying Chebyshev's inequality, we obtain

$$\text{Prob}(X = 0) \leq \text{Prob}(|X - E[X]| \geq E[X]) \leq \frac{\text{Var}[X]}{E[X]^2} \leq \frac{t/4}{(t/2)^2} = \frac{1}{t}.$$

11 Occupancy Problems

Suppose we throw m balls into n bins, and for each ball we choose one bin uniformly at random, and all random choises are independent. We can ask several natural questions, e.g.,

1. How large should m be, so that no bin remains empty?
2. What is the maximum load of any bin?
3. What is the probability that one particular bin has a high load?

All these questions are interesting for hashing: Suppose you want to store m keys in a hash table of size n. Each hash table cell contains a linked list, in which the elements that are hashed to that cell will be stored. One often assumes that the hash function maps all keys uniformly at random to the hash table cells. For example, the maximum load of a bin then corresponds to the maximum length of a linked list and thus to the maximum lookup time.

11.1 How many balls are needed to cover all bins?

This is also known as the "Coupon's Collector" problem. Suppose there are n types of coupons. The coupon's collector buys coupons until it has at least one coupon of each type. We assume that each time the collector buys a coupon, its type is determined uniformly at random and independently from all other coupons. What is the expected number of coupons the collector buys? This is the same as the expected number of balls we need to throw in the bins until each bin is covered.

Recall that for two events A, B it holds $\text{Prob}(A \cup B) \leq \text{Prob}(A) + \text{Prob}(B)$. Hence, for n events $A_1, \ldots, A_n$ we have

$$\text{Prob}\left(\bigcup_{i=1}^{n} A_i\right) \leq \sum_{i=1}^{n} \text{Prob}(A_i).$$

Let X be the number of balls needed, until all bins are covered. Consider some point in time when exactly $i-1$ bins are covered. Let X_i be the number of balls we need to still throw into random bins until i bins are covered. Clearly, then $X = X_1 + \cdots + X_n$.

The probability that a ball is thrown into an empty bin, when $i-1$ bins have a ball, is $(n-i+1)/n$. Thus, X_i is geometrically distributed with parameter $q = (n-i+1)/n$ and $E[X_i] = n/(n-i+1)$. We get

$$E[X] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{n}{n-i+1} = n \cdot \sum_{i=1}^{n} \frac{1}{i} = n \cdot H_n.$$

This yields the following theorem.

Theorem 11.1. *Let X be the number of balls we need to throw into n bins (chosen uniformly and independently at random) until all bins have at least one ball. Then*

$$n \ln n \leq E[X] = n \cdot H_n \leq n \ln n + n.$$

So after $n \ln n + n$ balls have been thrown into the n bins, we can "expect" that each bin has a ball. But what if for example we throw in twice that many balls? What can we say about the probability that each bin has a ball? We could apply Markov Inequality, but this would give us only a weak bound. Here is a much stronger one.

Theorem 11.2. *Let $\epsilon \geq 0$ be an arbitrary constant. If $(1+\epsilon) \cdot n \ln n$ balls are distributed uniformly at random over n bins, then the probability that some bin remains empty is at most $n^{-\epsilon}$.*

Proof: Consider bin i. The probability that bin i remains empty is at most

$$\left(1 - \frac{1}{n}\right)^{(1+\epsilon)\cdot n \ln n} \leq e^{-(1+\epsilon)\cdot \ln n} = n^{-(1+\epsilon)}.$$

The probability that either bin 1, or bin 2, or ..., or bin n remains empty is at most $n \cdot n^{-(1+\epsilon)} = n^{-\epsilon}$. □

11.2 Bounding the Probability for Large Maximum Loads

We now want to determine the load of the fullest bin if the number of balls is equal to the number of bins. For that (and many other purposes), the following inequalities are helpful:

$$\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{e \cdot n}{k}\right)^k \quad \text{and} \quad \binom{n}{k} \leq \frac{n^k}{k!}. \tag{11.1}$$

We consider the case that there are n balls and $m = n$ bins. Let Y_i be the number of balls in the i-th bin. Hence, $E[Y_i] = 1$ for all $1 \leq i \leq n$. Let $Y = \max\{Y_i \mid 1 \leq i \leq n\}$. What is the probability, that $Y \geq k$?

If $Y_i \geq k$, then there are k balls that all end up in bin i. There are $\binom{n}{k}$ possibilities to choose exactly k balls. The probability that all k balls are thrown in bin i is $(1/n)^k$. Hence,

$$\text{Prob}(Y_i \geq k) \leq \binom{n}{k} \cdot \left(\frac{1}{n}\right)^k \leq \left(\frac{e \cdot n}{k}\right)^k \cdot \left(\frac{1}{n}\right)^k = \left(\frac{e}{k}\right)^k.$$

For example, for $n \geq 16$

$$\begin{aligned}
\text{Prob}(Y_i \geq e \ln n / \ln\ln n) &\leq \left(\frac{e \ln\ln n}{e \ln n}\right)^{e \ln n / \ln\ln n} \\
&= \exp\left((\ln\ln\ln n - \ln\ln n)\frac{e \ln n}{\ln\ln n}\right) \\
&= \exp\left(-e \ln n + \ln\ln\ln n \frac{e \ln n}{\ln\ln n}\right) \\
&\leq \exp\left(-e \ln n + .7 \ln n\right) \leq \exp\left(-2 \ln n\right) = \frac{1}{n^2}.
\end{aligned}$$

We used that for $n > e^e$, i.e., $n \geq 16$

$$\frac{e \ln n}{\ln\ln n} \ln\ln\ln n \leq .7 \ln n$$

Thus,

$$\text{Prob}(Y \geq e \ln n / \ln\ln n) \leq \sum_{1 \leq i \leq n} \text{Prob}(Y_i \geq e \ln n / \ln\ln n) \leq 1/n.$$

Theorem 11.3. *If n balls are thrown uniformly at random into n bins, then with probability at least $1 - 1/n$ no bin has $e \ln n / \ln\ln n$ or more balls.*

12 The Principle of Deferred Decisions

(See also p. 55-56 in the textbook.)

12.1 Example: Solitaire

Consider the following solitaire-game played with a deck of 52 playing cards. The 4 suits are diamonds (♢), hearts (♡), spades (♠), and clubs (♣). Shuffle the deck of cards, and then divide it into 4 piles. Each pile is associated with one of the suits (but contains random cards).

Now we pick a card from the ♢-pile and throw it away. The suit of the card picked, determines a pile from which we pick the next card, and so on. The game ends, once we pick a card and the pile associated with that card's suit is empty. We win the game, if we can throw away all cards. What is the probability that this happens?

Observation 12.1. *The card we hold in hand when the game ends is always a ♢-card.*

Proof. For the purpose of a contradiction, assume that the suit of the last card we hold in hand is not ♢, for example it is ♠. Then the ♠-pile is empty, which means that the card we hold in hand is the 14th ♠-card. This is a contradiction, because there are only 13 ♠-cards in the deck of 52 cards.

(Note that once 12 ♢ cards have shown, the ♢-pile is empty, because we pick the first card from that pile. Thus, when we pick the 13-th ♢ card, the game ends.) □

By the *principle of deferred decision*, we can in advance fix the order in which the cards appear: Fix an arbitrary permutation of cards and then put the first card on top of the ♢-pile, the second card on top of the pile that corresponds to the suit of the first card, and so on. Thus, for every permutation π of the 52 cards, we can distribute the cards on the pile in such a way that the cards appear in the order of π (until we cannot pick up a card anymore). By Observation 12.1, the game ends with empty piles, if and only if the last card in this permutation is ♢. The probability that we can finish the game is the same as the probability that the suit of the last card in π is ♢. This probability is 1/4.

12.2 Example: Stable Marriages

(See also Section 3.5 in the textbook.)

Consider a monogamous, heterosexual society of n men numbered $1, \ldots, n$ and n women numbered $1, \ldots, n$. A *marriage* is a permutation π, and we say that man i is *married* to woman $\pi(i)$. The pairs $(i, \pi(i))$ are called *couples*. Each man and each woman has a *preference list* of all possible partners of the opposite sex. Two distinct couples (i, i') and (j, j') are *dissatisfied*, if i prefers j' over i', and j' prefers i over i'. (I.e., i and j' would leave their mates because they found "each other".) A marriage is *stable*, if there are are no dissatisfied pairs. The goal is to find a stable marriage for given preference lists.

The naive approach to solve this problem is to start with an arbitrary matching and then pick an unsatisfied pair, remarry it, and so on. However, this process may *cycle* and never stop. Now consider the *proposal-algorithm* in Figure 12.1.

Algorithm 8: Proposal Algorithm

```
1 while there is a man i that is not engaged to any woman do
2 |   Let S_i be the set of women that have not yet rejected man i or split up with i
3 |   Man i proposes to the most desirable woman j in S_i.
4 |   If j is currently engaged to a man i' that she prefers over i, then she rejects i
5 |   Otherwise, she gets engaged with i (after splitting up with the man she is
  |   currently engaged with, if any).
6 end
7 Every engaged pair marries.
```

Figure 12.1: Computing a Stable Matching.

Proposition 12.2. *The resulting matching is stable.*

Proof: For the purpose of a contradiction, assume that in the end there are two couples (A, c) and (B, d) such that A prefers d over c and d prefers A over B. Since A prefers d over c, he proposes to d before he proposes to c. After that, d will not accept a proposal from a man worse than A, in particular not a proposal from B. □

The worst-case running time of the algorithm is at most quadratic.

Proposition 12.3. *The algorithm terminates after at most $n^2 - n + 1$ iterations.*

Proof: If a set S_i is empty, man i has proposed to all women hence all women are married (a woman gets married when someone proposes to her for the first time, and from then on she is never unmarried). In an iterations in which we consider man i, the cardinality of set S_i decreases by one. After at most $n^2 - (n-1)$ iterations at least one set S_i is empty, and the algorithm stops because all women (and thus all men) are married. □

Note that the running time can be $\Omega(n^2)$: Assume that every man has the same preference list $(1, \ldots, n)$, and every woman has the preference list $(1, \ldots, n)$, and the men are considered in the order $1, \ldots, n$. When it's man i's turn to propose, all men $1, \ldots, i-1$ are engaged with women $1, \ldots, i-1$. Thus, women $1, \ldots, i-1$ will all reject man i's proposal and man i gets engaged with woman i. Hence, the total number of iterations is $1 + 2 + \cdots + n = n(n+1)/2$.

Average Case Analysis. We now analyze the running time, when the men's preference lists are chosen uniformly at random. The women's preference lists can be arbitrary (but must be fixed before the men make up their mind).

We apply the principle of deferred decisions in the following way: We do not pick the men's preferences lists in advance, but whenever its man i's turn to propose, he picks one woman randomly from S_i. Clearly, this leads to exactly the same probability distribution.

Let T_i be the number of proposals man i makes.

Now assume that the men are amnesiac, so they forget to whom they have already proposed to, and they pick the women to propose to uniformly at random from all women. Let T_i' be the number of proposals man i makes in this situation.

Since proposing to the same woman twice will always mean that the second proposal will be rejected, it follows $\text{Prob}(T_i' > k) \geq \text{Prob}(T_i > k)$ for all k. Hence,

$$E[T_i] = \sum_{k \geq 0} \text{Prob}(T_i > k) \leq \sum_{k \geq 0} \text{Prob}(T_i' > k) = E[T_i'].$$

Whenever a man proposes to an unmarried woman, that woman will accept, and she will never be single again. Thus, T' is bounded by the number of balls we need to throw into n bins, until all bins are covered. (Throwing ball i in bin j means that man i proposes to woman j.) We know that the probability that we need more than $(1+\epsilon)n \ln n$ balls is bounded by $n^{-\epsilon}$, and that the exepcted number of balls we need is $O(n \ln n)$.

Theorem 12.4. *With probability at least* $1 - n^{-\epsilon}$*, the number of iterations is bounded by* $(1 + \epsilon)n \ln n$*, and the expected number of iterations is* $O(n \ln n)$*.*

13 Dictionaries

A *dictionary* maintains a set S of n *keys* from some universe U, and possibly with each key $x \in S$ and additional data item D_x. We distinguish *static* and *dynamic* dictionaries.

A static dictionary receives the entire set S and builds the dictionary for set S in a preprocessing phase, so that the search for a key x and the data item D_x can be performed efficiently. Usually, it supports two operations:

1. `Preprocess`(S): builds the data structure for set S and the data items D_x, $x \in S$.
2. `Find`(x): Returns D_x, if $x \in S$, and returns $\perp$ if $x \notin S$.

A *dynamic dictionary* is used when frequent updates are necessary. It supports the operations

1. `Insert`(x, D_x): insert the data item D_x and its key x in the dictionary.
2. `Delete`(x): removed the key x as well as the data item D_x from the dictionary (and return $\perp$ if $x \notin S$).
3. `Find`(x): Returns D_x, if $x \in S$, and returns $\perp$ if $x \notin S$.

In the following we discuss randomized data structures for dynamic and static dictionaries. We focus on storing the *key* from S. It is easy to modify the data structure so that data items are stored together with the keys.

13.1 Skip Lists

Skip Lists can be used to implement a dynamic dictionary. We assume that the universe, U, is totally ordered.

A *pointer-tower of height* h, $h \in \mathbb{N}$, is an object that stores an element from U together with h pointers. We say the i-th pointer in a pointer-tower, is the pointer at *level* i. A skip list for a set $S = \{x_1, \ldots, x_n\} \subseteq U$ of size n is an ordered sequence of $n+1$ pointer-towers $T_0, \ldots, T_n$ of heights $H_0, H_1, \ldots, H_n$, respectively, where $H_0 = \max\{H_1, \ldots, H_n\}$. The value H_0 is denoted the *height* of the skip-list. The *head tower*, T_0, does not store any element, and if $x_1 < x_2 < \cdots < x_n$, then the x_i is stored in T_i. The pointer on level ℓ of tower T_i points to the next pointer-tower in the sequence which has a height of at least H_i, and to NIL, if no next pointer-tower of height at least H_i exists.

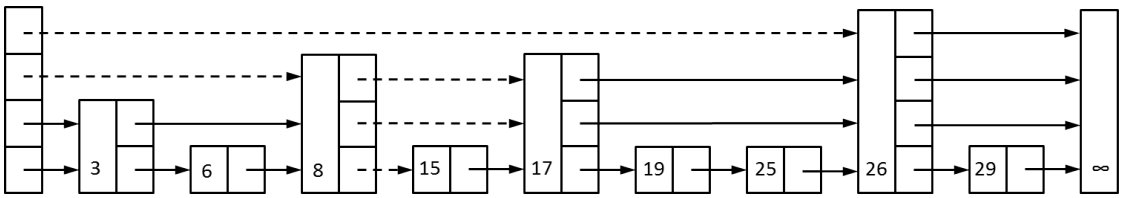


Figure 13.1: A skip list. The dotted lines indicate pointers that are traversed in a search for any key x, where $8 < x \leq 15$.

13.1.1 Algorithm

To search for a key x, we first follow the pointers on the highest level, H_0, until we find the pointer-tower T of height H_0 that stores the largest value $y \leq x$. Then, starting at pointer-tower T, we continue the traversing the list at height $H_0 - 1$, until we find the pointer-tower storing the largest element with a value at most x, and so on. Once we find the largest value $x^* \leq x$ in the bottom list (i.e., in all pointer towers), we either have $x^* = x$ or x is not stored in the skip list. Note that without any extra "work", the search allows us to determine for every height i, $1 \leq i \leq H_0$, the unique pointer-tower of height at least i which stores the largest value that is no larger than x.

To insert a new key x, we create a new pointer-tower T^* of a random height $h \geq 1$, where h is geometrically distributed with parameter $1/2$, and store x in it. (I.e., the height is determined by repeatedly flipping an unbiased coin, and setting h to the number of flips until head appears for the first time.) Then we search for x in the skip list, and if x is not already present we just have to insert T^* at the right positions in each of the lists (on every level) i: Let T be the pointer-tower of height at least i which stores the largest value smaller than x. (Recall that this pointer-tower is obtained during the search for every height i.) If T's pointer at height i points to T', then we simply redirect that pointer to point to T^*, and set the pointer of T^* at height i to T'.

Deletions can be performed in a similar straight-forward manner.

13.1.2 Analysis

Consider a skip list for n keys.

We first analyse the space requirements. Let H_i denote the height of the i-th element stored in the skip list. Then H_i is geometrically distributed with parameter $1-q$. Recall that $H_0 = \max\{H_1, \ldots, H_n\}$ is the height of the skip list.

Since H_i, $1 \leq i \leq n$, is geometrically distributed (see Example 4.7), we have for any integer $h \in \mathbb{N}$ that $\text{Prob}(H_i > h) = 2^{-h}$. Hence, by the union bound the probability that there exists a tower T_i, $1 \leq i \leq n$, of height $H_i > h$ is at most $n \cdot 2^{-h}$. Let $\ell = \lceil \log n \rceil$. Then

$$\text{Prob}(H_0 > \ell + c) \leq n \cdot 2^{-\ell - c} \leq n \cdot 2^{-\log n - c} = 2^{-c}. \tag{13.1}$$

Hence, we obtain the following upper bound on the expectation of H_0:

$$E[H_0] \overset{\text{Theorem 4.6}}{=} \sum_{h \geq 0} \text{Prob}(H_0 > h) = \sum_{0 \leq h \leq \ell - 1} \text{Prob}(H_0 > h) + \sum_{i \geq 0} \text{Prob}(H_0 > \ell + i)$$
$$\overset{(13.1)}{\leq} \sum_{0 \leq h \leq \ell - 1} 1 + \sum_{i \geq 0} 2^{-i} = \lceil \log n \rceil + 2 \tag{13.2}$$

Theorem 13.1. *The expected number of pointers needed to store in a skip list of n elements is*

$$2n + \lceil \log n \rceil + O(1).$$

Proof. We need to store H_i pointers in the tower T_i, $0 \leq i \leq n$. For $1 \leq i \leq n$, H_i is geometrically distributed with parameter $1/2$, so we have $E[H_i] = 2$. Thus, the theorem follows from (13.2). □

We now analyze the time it takes to search in a skip list. The time for an insertion and a deletion is asymptotically the same.

Suppose we search for an element x. Let x^* be the largest value stored in the skip list that is not larger than x. (I.e., if x is stored in the skip list, then $x^* = x$.) The time to search for x^* is asymptotically the same as the time to search for x, as the search for x will at the end find the tower that stores x^* and reach the bottom of that tower.

Note that each tower that gets visited during the search for element x^* is reached for the first time on the highest level of that tower.

We visit all towers found during the search in a reverse order, starting with the tower that stores x^*. By the principle of deferred decisions (see Chapter 12), we can determine the random height of a tower when we visit it for the first time during this backward search.

Suppose during the backward search we reach a tower T_i at level h. Then we know that $h_i \geq h$. Given that $h_i \geq h$, the actual height h_i of T_i can be determined by starting at height h, and then repeatedly flipping an unbiased coin at random until it shows heads, and for each tails flip adding one to the height. In other words, with probability $1/2$,

our backwards search goes "up" one level at the tower T_i, and with probability 1/2 it goes to a previous tower T_j, $j < i$, reaching that tower on the same level.

We continue the backwards search until we either reach the "head" tower, T_0, or until we reach height $\ell = \lceil \log n \rceil$, whatever happens first. Let X denote the number of steps until this happens. Then X is bounded by the number of coin-flips we need until we see ℓ times tails (as a tails flip corresponds to going "up" one level in the backwards search). I.e., $X = X_1 + \cdots + X_\ell$, where X_i is the number of coin-flips after $i-1$ tails have been flipped until we see another tails flip. Since $E[X_i] = 2$ for $1 \leq i \leq \ell$ we have $E[X] = 2\ell = 2\lceil \log n \rceil$.

Once we have reached level ℓ (or tower T_0) on our backwards search, the remaining number of steps until we reach tower T_0 is bounded by the number of towers of height at least ℓ. The probability that a fixed tower T_i, $1 \leq i \leq n$, has height at least ℓ is $1/2^\ell \leq 1/n$. Hence, the expected number of towers of height at least ℓ is at most 1. It follows that once we have reached level ℓ, the expected remaining number of towers we visit on the backwards search is at most 1. In addition, the expected remaining number of steps to reach the highest level of tower T_0, i.e., level H_0, is at most H_0. It follows that the total number of steps of the backwards search has expectation $E[X] + E[H_0] + 1$. Since $E[X] \leq 2\lceil \log n \rceil$ and by (13.2) $E[H_0] \leq \lceil \log n \rceil + 2$, we obtain that the expected search time is $O(\log n)$.

Theorem 13.2. *The expected time needed to perform a search, insert, or delete operation in a skip list with n elements is $O(\log n)$.*

13.2 Randomized Hashing Techniques

Throughout this section we assume that $|U| = n^{O(1)}$, i.e., each key can be identified by an $O(\log n)$-bit string. In order to construct random hash functions, we pick hash functions randomly from a family of hash functions, called *hash family*. One may wonder why we don't pick a random function from the set of all functions $h : U \to R$. The reason is that we have to store such a function which would require at least $|U| \cdot \lfloor \log |R| + 1 \rfloor$ bits. Hence, we restrict ourselves to "small" hash families with good randomness properties.

Definition 13.3. Let $\mathcal{H}$ be a *hash family* with hash functions $h : U \to R$ for two finite sets U and R. Then $\mathcal{H}$ is called *c-universal*, if for all $x, x' \in U$, $x \neq x'$, it holds

$$\text{Prob}_{h \in \mathcal{H}}\big(h(x) = h(x')\big) \leq \frac{c}{|R|}.$$

A 1-universal hash family is simply called *universal*.

The probability that two keys have the same hash function value is called *collision probability*.

13.3 Randomized Hashing with Chaining

Assume that we have a c-universal hash family $\mathcal{H}$ for a constant c. Then it is easy to implement hashing with chaining in such a way that a `Find`-operation requires only constant time in expectation.

Function Preprocess(S, D)

Input: A sequence S of pairs $(x_1, D_1), (x_2, D_2), \ldots$, where $x_i \in U$ and D_i is a data item.

`/* Initializes a static dictionary` $\mathcal{D} = (h, L_0, \ldots, L_{n-1})$. `*/`

Let $n := |S|$
Initialize n empty linked lists $L_0, \ldots, L_{n-1}$.
Let $\mathcal{H}$ be a c-universal hash family $U \to \{0, \ldots, n-1\}$
Pick $h \in \mathcal{H}$ uniformly at random
for $i = 1, \ldots, n$ **do**
  Insert the pair (x_i, D_i) into the list $L_{h(x_i)}$ (at the front)
end
Let $\mathcal{D} = (h, L_0, \ldots, L_{n-1})$

Function Find(x)

Input: A key $x \in U$.
Output: D_i if $x = x_i$ and $(x_i, D_i) \in S$; otherwise $\perp$.

Search for a pair $(x, \cdot)$ in the list $L_{h(x)}$
if (x_i, D_i) *with* $x = x_i$ *found* **then**
  return D_i
else
  return $\perp$
end

Figure 13.2: Implementation of a static dictionary based on Hashing with Chaining

Assuming that a hash function value, i.e., $h(x)$, can be computed in constant time, the

total time for preprocessing is $O(n)$. What is the search time?

Consider an element x^*. Then x^* is in list L_i, where $i = h(x^*)$. In order to find x^* we have to compute $h(x^*)$ and search (at least partly) through list L_i. Hence, the time is bounded by the number of elements in that list, $|L_i|$. Assuming that the hash function can be evaluated in constant time, the time for `Find(`x^*`)` is $O(|L_i|)$.

This is a random variable, whose expectation we can bound as follows. For $x \in S - \{x^*\}$ let $Y_x = 1$ if $h(x) = i$, and $Y_x = 0$, otherwise. Hence,

$$E[Y_x] = \text{Prob}(Y_x = 1) = \text{Prob}(h(x) = h(x^*)) \leq \frac{c}{n}.$$

It follows

$$E[|L_i|] = \sum_{x \in S} E[Y_x] \leq c.$$

Thus, the expected time for `Find(`x^*`)` is $O(1 + c)$ in addition to the time to evaluate h. Hence, if h can be evaluated in constant time, and if $c = O(1)$, then we obtain a constant upper bound for the expected `Find`-time.

A 2-Universal Hash Family. Let $U = \{0, \ldots, u-1\}$ and $R = \{0, \ldots, r-1\}$ for $r \leq u$. Fix a prime p, $p \geq u$, and let $h_a : U \to R$, $h_a(x) = \big((a \cdot x) \bmod p\big) \bmod r$, and $\mathcal{H}_{p,r} = \{h_a \mid a \in \{0, \ldots, p-1\}\}$.

Then for $x, x' \in U$, $x \neq x'$, and $d = x - x' \neq 0$, it holds

$$\begin{aligned} h_a(x) = h_a(x') &\Leftrightarrow \big((ax) \bmod p\big) \bmod r = \big((ax') \bmod p\big) \bmod r \\ &\Leftrightarrow \big((ax) \bmod p\big) - \big((ax') \bmod p\big) \equiv 0 \pmod r \\ &\Leftrightarrow (a \cdot d) \bmod p \equiv 0 \pmod r. \end{aligned}$$

Let $k = \lceil p/r \rceil$. Thus, there are at most k multiples of r in $\{0, \ldots, p-1\}$. Since $(a \cdot d) \bmod p$ is uniformly distributed over $\{0, \ldots, p-1\}$ for a randomly chosen $a \in \{0, \ldots, p-1\}$, it follows that

$$\text{Prob}(h_a(x) = h_a(x')) \leq \frac{k}{p} = \frac{\lceil p/r \rceil}{p} \leq \frac{1}{r} + \frac{1}{p} \leq \frac{2}{r}.$$

We have just proved the following theorem.

Theorem 13.4. *The hash family $\mathcal{H}_{p,r}$ is 2-universal.*

Note that by our assumption that $|U| = n^{O(1)}$ we need only $O(\log n)$ bits to store a hash function $h_a \in \mathcal{H}_{p,r}$. (We only have to store the integer $a \in \{0, \ldots, p-1\}$.) Assuming that arithmetic over integers in U is also possible in constant time with constant expected time for the `Find` operation. If pointers require $O(\log n)$ bits and data items need d bits, we obtain a dynamic dictionary with $O(n \log n + dn)$ space, linear preprocessing time and constant expected time for a search operation.

13.4 Perfect Hashing

Often, the efficiency of `Find` is much more important than the preprocessing time. In this section we discuss a static dictionary data structure, where the time for `Find` is constant in the *worst-case*. For that we are willing to sacrifice the linear worst-case time for preprocessing. However, we can ensure that preprocessing has *expected* linear time, while the space bound remains $O(n \log n + nd)$.

Definition 13.5. A function $h : U \to \{0, \ldots, m-1\}$ is a *perfect hash function* (PHF) for an n-element set $S \subseteq U$, if h is injective on S. A perfect hash function is a *minimal perfect hash function* (MPHF), if $m = O(n)$.

Remark 13.6. In literature, sometimes the definition of *minimality* requires $m = n$ instead of $m = O(n)$.

Assume we know how to construct a minimal perfect hash function $h : U \to R$ for an n-element set $S \subseteq U$, such that h can be evaluated in constant time. Then we can store each pair (x, D_x), $x \in S$, in a table $T[0, \ldots, m-1]$ at position $h(x)$. This way, we obtain a dictionary with constant lookup-time, and space $O(n \log n + nd + \log |\mathcal{H}|)$, where $\mathcal{H}$ is the family from which we have chosen h. Our goal is to have $|\mathcal{H}| = 2^{O(n \log n)}$, so that the space for storing a hash function is not asymptotically larger than the space for storing the n keys.

In the following we assume that $\mathcal{H}_m$ is a c-universal hash family $U \to \{0, \ldots, m-1\}$ for some constant c, and $m \geq n$ is chosen appropriately. Let $D = \{(d_1, \ldots, d_m) \mid d_i \in \{0, \ldots, m-1\}\} = \{0, \ldots, m-1\}^m$. For $f, g \in \mathcal{H}_m$ and $d \in D$, let

$$h_{f,g,d} : U \to \{0, \ldots, m-1\}, \quad x \mapsto \big(f(x) + d_{g(x)}\big) \bmod m.$$

The hash family $\mathcal{F}_{m,n}$ consists of the hash functions $h_{f,g,d}$ with $f, g \in \mathcal{H}_m$ and $d \in D = \{0, \ldots, m-1\}^m$.

Algorithm 9: PerfectHash

Input: An n-element set $S \subseteq U$
Output: A function $h_{f,g,d} \in \mathcal{F}_{m,n}$ for $m = O(n)$ that is perfect for S.

1 Let $m = 5 \cdot n$.
2 **repeat**
3 Pick $f \in \mathcal{H}_m$ and $g \in \mathcal{H}_m$ uniformly at random.
4 Let $B_i = \{x \in S \mid g(x) = i\}$, for $0 \leq i < m$, $b_i = |B_i|$, and $b = \sum_{\substack{0 \leq i < m \\ b_i > 1}} b_i^2$
5 **until** $b \leq n$ and f is injective on each B_i, $0 \leq i < m$
6 Sort the sets B_i according to their size, i.e., find π s.t. $b_{\pi(0)} \geq \cdots \geq b_{\pi(m-1)}$.
7 Let $d_0 = \cdots = d_{m-1} = 0$.
8 **for** $0 < i < m$ **do**
9 **repeat**
10 Pick d_i uniformly at random from $\{0, \ldots, m-1\}$.
11 **until** $h_{f,g,d}$ is injective on $B_{\pi(0)} \cup \cdots \cup B_{\pi(i)}$
12 **end**
13 **return** $h_{f,g,d}$

Figure 13.3: An Algorithm to compute a perfect hash function.

How much space do we need to describe a hash function $h \in \mathcal{F}_{m,n}$? Clearly, $O(\log n)$ bits suffice for f and g, if $|U| = n^{O(1)}$. The vector d consists of m entries with $O(\log m)$ bits each. So if $m = O(n)$, then the space for d is $O(n \log n)$. In order to simplify calculations, we assume in the following that $c = 2$, i.e., $\mathcal{H}_m$ is a 2-universal hash family.

Figure 13.3 (on p. 72) depicts an algorithm that for a given set $S \subseteq U$ of cardinality n computes a perfect hash function $h_{f,g,d} \in \mathcal{F}m, n$. If the algorithm terminates, then the second repeat-until loop guarantees that $h_{f,g,d}$ is injective on $B_{\pi(0)} \cup \cdots \cup B_{\pi(m-1)}$, i.e., on the entire set S of keys. Hence, the algorithm returns a minimal perfect hash function.

So we have to find out whether it terminates at all, and if so, how long it takes.

First note that the sorting in Line 6 takes only $O(m) = O(n)$ time if we use for example Bucket sort. In the following lemmas we show that each of the two repeat-until loops has a "success" probability that is constant and larger than 0. Hence, each loop will be

repeated a constant number of times in expectation.

Lemma 13.7. *For $m = 5n$ and $f, g \in \mathcal{H}_m$ chosen at random, let B_i, b_i, and b as in the algorithm in Figure 13.3. Consider the following events.*

A: $\sum_{\substack{0 \le i < m \\ b_i > 1}} b_i^2 \le n$.

B: f is injective on all B_i.

Then $\text{Prob}(A \wedge B) \ge 3/25$.

Proof: We first bound the probability of the event $\bar{A}$. If $b_i > 1$, then $b_i^2 \le 2b_i(b_i - 1)$. Clearly, two keys x and x' collide under the hash function g, if and only if there exists a value i, $0 \le i < m$, such that $x, x' \in B_i$. Thus,

$$b_i(b_i - 1) = |\{(x, x') \in S^2 \mid x \neq x' \wedge g(x) = g(x') = i\}.$$

For two keys $x, x' \in S$ let $Y_{x,x'}$ be the indicator variable with value 1 iff $x \neq x'$ and $g(x) = g(x')$. Since $g : U \to \{0, \ldots, m-1\}$ is chosen at random from a c-universal hash family, the event "$g(x) = g(x')$" occurs with probability at most c/m. Hence, $E[Y_{x,x'}] \le c/m$. It follows that

$$\sum_{0 \le i < m} b_i(b_i - 1) = \sum_{x,x' \in S} Y_{x,x'}. \le \sum_{x,x' \in S} \frac{c}{m} = \frac{c \cdot n(n-1)}{m} \le \frac{2 \cdot n}{5}.$$

By Markov Inequality,

$$\text{Prob}\left(\overline{A}\right) = \text{Prob}\left(\sum_{0 \le i < m} b_i(b_i - 1) > n/2\right) < \frac{2 \cdot n}{5} \cdot \frac{1}{n/2} = \frac{4}{5}.$$

Now we prove a lower bound for the conditional probability of event B, given that A occurs. Consider B_i for some value $0 \le i < m$. The probability that f is *not injective* on B_i is

$$\begin{aligned} \text{Prob}(\exists x, x' \in B_i, x \neq x' : f(x) = f(x')) &\le \sum_{x \neq x' \in B_i} \text{Prob}(f(x) = f(x')) \\ &\le \begin{cases} b_i^2 \cdot \frac{c}{m} & \text{if } b_i > 1 \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

Hence,

$$\text{Prob}\big(\overline{B}\big) \leq \frac{c}{m} \sum_{\substack{0 \leq i < m \\ b_i > 1}} b_i^2$$

Given A, we obtain

$$\text{Prob}\big(\overline{B}|A\big) \leq \frac{c}{m} \cdot n \leq \frac{2}{5}.$$

Thus,

$$\begin{aligned} \text{Prob}(A \wedge B) &= \text{Prob}(A) \cdot \text{Prob}(B|A) \\ &= \big(1 - \text{Prob}\big(\bar{A}\big)\big)\big(1 - \text{Prob}\big(\bar{B}|A\big)\big) \\ &\geq \frac{3}{5} \cdot \frac{1}{5} = \frac{3}{25}. \end{aligned}$$

□

Lemma 13.8. *Assume that f is injective on all B_i, $0 \leq i < m$, and that $\sum b_i^2 < n$, where the sum is taken over $0 \leq i < m$, where $b_i > 1$. If $h_{f,g,d}$ is injective on $B_{\pi(0)} \cup \cdots \cup B_{\pi(i-1)}$, then for a randomly chosen $d_{\pi(i)}$, the probability that $h_{f,g,d}$ is injective on $B_{\pi(0)} \cup \cdots \cup B_{\pi(i)}$ is at least $4/5$.*

Proof: Assume w.l.o.g. that $\pi = \text{id}$, i.e., $\pi(j) = j$ for $0 \leq j < m$. Hence, $b_0 \geq \cdots \geq b_{m-1}$. Further, let $h_{f,g,d}$ be injective on $B_0 \cup \cdots \cup B_{i-1}$. We consider B_i.

First consider the case that $b_i = 1$. In this case there is only one element $x \in B_i$. Thus, if we pick a random value d_i, only the hash function value $h_{f,g,d}(x)$ gets changed. The probability that $h_{f,g,d}(x)$ is different from all $n-1$ other hash function values $h_{f,g,d}(x')$, $x' \in S - \{x\}$ is

$$1 - \frac{n-1}{m} \geq 1 - \frac{n}{5n} = \frac{4}{5}.$$

Now assume that $b_i > 1$. We say that a set B_j, $j \neq i$, gets "invalidated" if an element from B_i collides with an element in B_j, i.e., if there exist $x \in B_i$ and $x' \in B_j$ such that $h_{f,g,d}(x) = h_{f,g,d}(x')$, where d_i is chosen at random. Consider each element $x \in B_i$. The probability that one particular such element $x \in B_i$ collides with one of the elements in B_j for a randomly chosen d_i is $|B_j|/m$. Hence, the total expected number of elements that get invalidated in sets B_j, $j \neq i$, $|B_j| > 1$, is at most

$$\sum_{x \in B_i} \sum_{\substack{0 \leq j < i \\ b_j > 1}} \frac{|B_j|^2}{m} < |B_i| \frac{n}{m} \leq \frac{|B_i|}{5}.$$

Hence, the probability that at least $|B_i|$ elements get invalidated is at most $1/5$ by Markov Inequality. But if less than $|B_i|$ elements get invalidated, then none of the sets $B_0, \ldots, B_{i-1}$ can get invalidated, because each of them has cardinality at least $|B_i|$. And in this case $h_{f,g,d}$ is injective on $B_0 \cup \cdots \cup B_i$. □

By Lemma 13.7, the first repeat-until loop of algorithm PerfectHash has a probability of at least $3/25$ to terminate after each iteration. Hence, the expected number of iterations of that loop is at most $25/3$. Similarly, by Lemma 13.8, the second repeat-until loop will be repeated at most $5/4$ times on average. It is not hard to see that every other step of the algorithm can be implemented in such a way that it has running time $O(n)$. Hence, we have derived the following theorem.

Theorem 13.9. *The algorithm PerfectHash computes an MPHF in expected time $O(n)$.*

Remark 13.10. The algorithm can be optimized in several ways. In fact, if n is a prime power, then a very similar algorithm can be used to construct a perfect hash function $h_{f,g,d}$ with range $\{0, \ldots, n-1\}$ (i.e., $n = m$), and which can be stored using $(2+\epsilon)n \log n$ bits for an arbitrary small constant $\epsilon > 0$. Moreover, the algorithm can be modified such that also insertions and deletions are possible in expected constant time.

13.5 Cuckoo Hashing

Cuckoo Hashing is a dynamic dictionary data structure that allows look-ups of keys in constant worst-case time, and insertion and deletion in expected constant time. The space requirements are linear in the number of keys.

The main idea of Cuckoo Hashing is to use *two* hash tables and *two* hash functions instead of one, and to store each element in one of the hash tables, in a position identified by the corresponding hash function. Similar to the 2-choices balls-into-bins algorithm, multiple choices lead to good performance.

Let $m = (1 + \varepsilon)n$, for some constant $\varepsilon > 0$. We use two hash tables $T_0[0 \ldots m-1]$ and $T_1[0 \ldots m-1]$. Then we choose two random hash functions $h_0, h_1 : U \to \{0, \ldots, m-1\}$.

We assume for now that h_0 and h_1 map all keys uniformly at random to the range, i.e., the random variables $h_0(x_1), \ldots, h_0(x_n), h_1(x_1), \ldots, h_1(x_n)$ are uniformly distributed and independent, for any set $S = \{x_1, \ldots, x_n\}$ of keys stored in the dictionary. Note that this is not a realistic assumption, but there are randomized techniques to achieve this (that we cannot discuss in this lecture).

Function RecIns(x, i, ctr)

Input: $x \in U$, $i \in \{0, 1\}$, $ctr \in \mathbb{N}$.
1 **if** $ctr > threshold$ **then return** `False`
2 $y := T_i[h_i(x)]$
3 $T_i[h_i(x)] := x$
4 **if** $y \neq \perp$ **then**
5 | `RecIns`$(y, (i+1) \bmod 2, ctr+1)$
6 **end**
7

Function Insert(x)

Input: $x \in U$.
1 **if** $\neg$`RecIns`$(x, 0, 0)$ **then**
2 | `Rehash()`
3 **end**

Figure 13.4: Cuckoo Hashing — The Insertion Procedure

In order to insert a key in the dictionary, we first insert it in the first hash table at $T_0[h_0(x)]$. If that hash table cell occupies another key, say x', remove that key before inserting x. We then re-insert the removed key, x', in the second hash table at $T_1[h_1(x')]$. This may require to remove a key x'' from that hash table cell that then is re-inserted in the first hash table, and so on.

It is not hard to see that for "bad" hash functions h_0, h_1, this procedure may cycle and never come to an end. Therefore, we keep track of the number of insertion attempts. Once too many insertion (i.e., more than *threshold*) attempts have been made, we choose a new hash function pair (h_0, h_1) and *rehash* everything, i.e., we collect all keys stored in the hash tables, and reinsert them into empty hash tables using the new hash functions. See Figure 13.4 for the insertion procedure; the implementation of the `Rehash()` procedure is obvious and omitted from the code.

In order to find a key x, we simply have to check both hash table cells $T_i[h_i(x)]$ for $i = 0, 1$.

The Cuckoo-Graph

Let S be a set of keys and V_0, V_1 be sets of m vertices, each. Fix 2 hash functions $h_0 : U \to V_0$, $h_1 : U \to V_1$. For $x \in S$, let $e(x)$ be the edge $\{h_0(x), h_1(x)\}$. The cuckoo-graph induced by S, h_0, h_1 is a multi-graph (i.e., a graph that may have multiple edges between the same vertex pairs) $G(S; h_0, h_1) = (V_0 \cup V_1, E)$, where E is the multi-set $\{e(x) \mid x \in S\}$.

In the following let $m = c \cdot n$ for $c = 1 + \varepsilon$, $\varepsilon > 0$. A path between two vertices u and

v is *edge-distinct*, if it contains each edge at most once. (Note that the length of a path is the number of its edges.)

Lemma 13.11. *Let $G = G(S; h_0, h_1)$ be the cuckoo-graph for two random functions h_0, h_1. Then for all $u, v \in V_0, V_1$ and all $\ell \in \mathbb{N}$,*

$$\text{Prob}(\textit{there is an edge-distinct path of length } \ell \textit{ between } u \textit{ and } v) \leq \frac{1}{m \cdot c^\ell}.$$

Proof. W.l.o.g. let $u \in V_1$. For a path $(u = u_1, \ldots, u_{\ell+1} = v)$ the vertex u_i is in V_1 if and only if i is odd.

For fixed $u_1, \ldots, u_{\ell+1}$ and fixed $x_1, \ldots, x_\ell$, the event "$e(x_i) = \{u_i, u_{i+1}\}$" occurs if and only if $h_0(x_i) = u_i$ and $h_1(x_i) = u_{i+1}$ in the case that i is even, and $h_1(x_i) = u_i$ and $h_0(x_i) = u_{i+1}$ in the case i is odd. Hence, the event has probability $1/m^2$. If the path is edge-distinct, all events "$e(x_i) = \{u_i, u_{i+1}\}$" are independent, and thus in this case

$$\text{Prob}(\forall 1 \leq i < \ell : e(x_i) = \{u_i, u_{i+1}\}) = \frac{1}{m^{2\ell}}.$$

There are at most $m^{\ell-1}$ possibilities of choosing the vertices $u_2, \ldots, u_\ell$, $u_i \in V_0$ if i is even and $u_i \in V_1$ if i is odd. Moreover, there are at most n^ℓ possibilities of choosing the keys $x_1, \ldots, x_\ell \in S$. Hence,

$$\begin{aligned}\text{Prob}(\exists u_2, \ldots, u_\ell, x_1, \ldots, x_\ell : \forall 1 \leq i \leq \ell : e(x_i) = \{u_i, u_{i+1}\}) \\ \leq \frac{n^\ell \cdot m^{\ell-1}}{m^{2\ell}} = \frac{n^\ell}{m^{\ell+1}} \leq \frac{1}{m \cdot c^\ell}\end{aligned}$$

□

Now let X_u be the length of the longest edge-distinct path leaving u, i.e.,

$$X_u = \max_v \{\ell \mid \text{there is an edge-distinct path of length } \ell \text{ between } u \text{ and } v\}.$$

Lemma 13.12. $E[X_u] = O(1)$.

Proof. By Lemma 13.11, for any ℓ, the probability that there exists a vertex v and an edge-distinct path of length exactly ℓ between u and v is at most $m/(m \cdot c^\ell) = 1/c^\ell$. (Note that given u and ℓ, there are m choices for v: If ℓ is even, then v is from the same set among V_0 and V_1 as u, otherwise it is from the other set.) Hence,

$$E[X_u] = \sum_{\ell \geq 1} \text{Prob}(X_u \geq \ell) \leq \sum_{\ell=1}^{\infty} \frac{1}{c^\ell} = -1 + \sum_{\ell=0}^{\infty} \frac{1}{c^\ell} = -1 + \frac{1}{1 - 1/c} = \frac{1}{c-1} = \frac{1}{\varepsilon}$$

□

Now suppose the keys in S are inserted in an arbitrary order. For each key, the expected length of the longest edge-distinct path it belongs to is $O(1)$. Hence, the expected insertion time for a fixed key is $O(1 + E[X_u])$ plus the time for rehashes that may occur during the insertion of that key. In the following we bound the probability that such rehashes occur.

A Simple Rehash Analysis

Let x be a key and $h_0(x) = u$ and $h_1(x) = v$. If inserting x causes a rehash, then either the edge $\{u, v\}$ is part of a cycle, or it is on an edge-distinct path of length at least $threshold$. If $threshold = d \cdot \log n$, then by Lemma 13.11, the probability that x is on an edge-distinct path of that length is at most (recall that there are m possible endpoints of such a path)

$$\frac{1}{c^{d \log n}} = \frac{1}{n^d}. \tag{13.3}$$

If the edge $\{u, v\}$ is part of a cycle, then there is an edge-distinct path from v to u, that uses at least one edge, and $\{u, v\}$ is not one of the edges on the path. The probability that such a path exists is bounded by

$$\sum_{\ell \geq 1} \frac{1}{m \cdot c^\ell} = \frac{1}{c \cdot m} \cdot \sum_{\ell \geq 0} \frac{1}{c^\ell} = \frac{1}{c \cdot m} \cdot \frac{1}{1 - 1/c} = \frac{1}{c \cdot m} \cdot \frac{c}{c-1} = \frac{1}{m(c-1)}. \tag{13.4}$$

Summing up the probabilities in (13.3) and (13.4), the probability that some fixed key $x \in S$ causes a rehash when `Insert`(x) is called is at most

$$\frac{1}{n^d} + \frac{1}{m(c-1)}. \tag{13.5}$$

Thus, the probability that a rehash occurs during n insert operations is bounded by

$$\frac{1}{n^{d-1}} + \frac{1}{c(c-1)}. \tag{13.6}$$

Now choose $c > (1 + \sqrt{5})/2 = 1.6180\ldots$. Then $c(c-1) > 1$. Hence, with constant positive probability no rehashes occur during the insertion of all keys of S.

Now consider an arbitrary single `Insert()` operation *op* during a sequence of n `Insert()` operations that starts with an empty dictionary. According to (13.5), the probability that *op* causes a rehash is $O(1/n)$. On the other hand, if it does incur a rehash, then by

(13.6) and our choice of c, only a constant number of rehashes will occur in expectation, until all elements are stored. Hence, the expected time for those rehashes is $O(n)$. Since $O(n)$ expected time for rehashes is only needed with probability $O(1/n)$, and with probability $1 - O(1/n)$ no rehashes are needed, the total expected time for operation *op* is $O(1)$. This yields the following result.

Theorem 13.13. *Let threshold* $\geq 2 \log n$ *and* $c > (1 + \sqrt{5})/2$. *If* n *keys are inserted into two hash tables, each of size* $m = \lceil c \cdot n \rceil$, *then every single* `Insert` *operation has an expected running time of* $O(1)$.

14 A Technique for Proving Lower Bounds

14.1 Two-Person Zero-Sum Games

See Section 2.2.1 in the textbook

Consider a *payoff* matrix M with entries in $\mathbb{R}$. Two players, Rachel and Charlie, try to maximize their payoff as follows: First Rachel announces a row i, then Charlie picks a column j. Finally, Charlie has to pay $M_{i,j}$ to Rachel. Hence, Charlie tries to minimize $M_{i,j}$ and Rachel tries to maximize $M_{i,j}$. An example is the rock-paper-scissors game. In this game, two players count aloud "Rock Paper Scissors" and on the fourth count they both simultaneously change their hands into one of three gestures, indicating either rock, paper, or scissors. Each gesture defeats exactly one other gesture (rock defeats scissors, scissors defeat paper, and paper defeats rock), and each player's objective is to defeat the other player's gesture. The game has the following payoff matrix:

	Scissors	Paper	Rock
Scissors	0	1	-1
Paper	-1	0	1
Rock	1	-1	1

Such a game is called a *zero-sum* game, because the total gain of all players is equal to the total loss of all players. A *pure* strategy of Rachel is a fixed row i. Assume that Rachel has the pure strategy i. Then the best strategy of Charlie is to pick the column which minimizes $M_{i,j}$. Thus, the amount Charlie has to pay is $P(i) = \min_j M_{i,j}$. Hence, Rachel should choose the row that maximizes $P(i)$. It follows that the best pure strategy of Rachel guarantees her a payoff of

$$V_R^{\text{pure}} = \max_i \min_j M_{i,j}$$

But what happens if Charlie starts the game, i.e., he has to announce a column first? Then his best pure strategy guarantees that Charlie does not have to pay more than

$$V_C^{\text{pure}} = \min_j \max_i M_{i,j}$$

In the rock-paper-scissor game, it holds $V_R^{\text{pure}} = -1$ and $V_C^{\text{pure}} = 1$. The same is true for any payoff matrix:

Lemma 14.1. *For any payoff matrix M,*

$$\max_i \min_j M_{i,j} \leq \min_j \max_i M_{i,j}.$$

Proof: Let $a_i = \min_k M_{i,k}$ be the minimal entry in row i, and $b_\ell = \max_i M_{\ell,j}$ be the maximal entry in column j. Since row i and column j have one entry in common, we have $a_i \leq b_j$. That is,

$$\forall i, j : \min_k M_{i,k} \leq \max_\ell M_{\ell,j}.$$

This immediately implies the claim. □

We say that a two-person zero-sum game has the *solution* (r, c), if r maximizes $\min_j M_{r,j}$ and c minimizes $\max_i M_{i,c}$ and

$$\min_j M_{i,r} = \max_i M_{i,c}.$$

This term is called the *value* of the game.

The rock-paper-scissors game has no solution. But in the following game, if Rachel chooses row i, and the Charlie chooses column j, then Charlie has to pay $j - i$ to Rachel.

	1	**2**	**3**
1	0	1	2
2	-1	0	1
3	-2	-1	0

Therefore, both players will choose the maximal row (column), no matter who starts the game. Thus, the game has a solution $(3, 3)$ with value 0.

Now suppose the players can make random choices. A *mixed* strategy is a probability distribution on the set of possible (pure) stratgies. Rachel picks a $p = (p_1, \ldots, p_n)$ such that $\sum_i p_i = 1$, and Charlie picks $q = (q_1, \ldots, q_m)$ such that $\sum_j q_j = 1$. Then row i is chosen with probability p_i and column j with probability p_j. Again, Charlie pays $M_{i,j}$ to Rachel.

Now the payoff is a random variable P, and

$$E[P] = \sum_{1 \le i \le n} \sum_{1 \le j \le m} p_i M_{i,j} q_j = p^T M q.$$

Assume that Charlie knows Rachel's mixed strategy (i.e., he knows p), and he can fix his probability distribution according to this knowledge. For Charlies' best choice, he has to pay

$$V_R = \min_q p^T M q.$$

Thus, Rachel will choose a strategy p that maximizes this term. It follows that if both choose the best strategies, but Rachel has to announce her strategy first, the expected payoff is

$$V_R = \max_p \min_q p^T M q.$$

On the other hand, if Charlie has to announce his strategy first, and then Rachel can "optimize" her strategy, then Rachel gains

$$V_C = \min_q \max_p p^T M q.$$

The following result says that it does not matter who has to announce the strategy first.

Theorem 14.2 (von Neumann's Minmax Theorem)**.** *For any two-person zero-sum gam specified by a matrix M,*

$$\max_p \min_q p^T M q = \min_q \max_p p^T M q.$$

□

Once Rachel's mixed strategy p is fixed, then $p^T M q = a_1 q_1 + \cdots + a_m q_m$ for some fixed values $a_1, \ldots, a_m$, where $a_k = a_1 M_{1,k} + \cdots + a_n M_{n,k}$ for $1 \le k \le m$. Hence, Charlie's best strategy is to pick the column j with probability 1 which minimizes a_j. Thus, some pure strategy for Charlie is optimal, if Rachel has to announce her strategy first. Similarly, for a fixed mixed strategy announced by Charlie, Rachel has an optimal pure strategy.

Let $e_k = (0, \ldots, 0, 1, 0, \ldots, 0)$, where the 1 is at the k-th position.

Theorem 14.3 (Loomis' Theorem)**.** *For any two-person zero-sum game specified by matrix M,*

$$\max_p \min_j p^T M e_j = \min_q \max_i {e_i}^T M q.$$

14.2 Yao's Technique

See Section 2.2.2 in the textbook.

Let $\mathcal{A}$ be a *finite* set of deterministic algorithms, and $\mathcal{I}$ be a *finite* set of inputs. For an input $x \in \mathcal{I}$ and an algorithm $A \in \mathcal{A}$, let $C(x, A)$ denote the cost of A on input x (the cost can be the running time, space complexity, or any other complexity measure).

Consider a game, where first Charlie announces a random distribution p_A over the algorithms, and, given p_A, Rachel decides for an input x. Then we pick A according to the distribution p_A, and run A on the input x. Charlie has to pay an amount proportional to the running time of A on x. Hence, the expected payoff is

$$\sum_{A \in \mathcal{A}} p_A \cdot C(x, A).$$

The expected running time of the algorithm described by p_A on the worst input is thus

$$\max_{x \in \mathcal{I}} \sum_{A \in \mathcal{A}} p_A \cdot C(x, A).$$

Our goal is to prove a *lower bound* on this term.

We consider a two-person zero-sum game, where Charlie plays the *algorithm designer* — he designs a randomized algorithm (i.e., he picks a random distribution over $\mathcal{A}$). Rachel is the *adversary* — she finds the worst input for a given randomized algorithm. Loomi's Theorem tells us that if first Charlie designs the optimal randomized algorithm p_A and then Rachel picks the worst case input x for that algorithm, the expected cost is the same as when first Rachel picks the "worst" input distribution q_x and then Charlie designs the optimal (deterministic) algorithm A for that distribution. I.e.,

$$\max_{q_x} \min_{A \in \mathcal{A}} \sum_{x \in \mathcal{I}} q_x \cdot C(x, A) = \min_{p_A} \max_{x \in \mathcal{I}} \sum_{A \in \mathcal{A}} p_A \cdot C(x, A) \tag{14.1}$$

We can conlude that

$$\forall q_x, p_A : \quad \min_{A \in \mathcal{A}} \sum_{x \in \mathcal{I}} q_x \cdot C(x, A) \leq \max_{x \in \mathcal{I}} \sum_{A \in \mathcal{A}} p_A \cdot C(x, A).$$

This means that if the adversary picks an input distribution q first, and then the algorithm designer designs the best possible deterministic algorithm for that distribution, then the expected cost is at most as large, as if first the algorithm designer has to find a randomized algorithm and then the adversary can choose the worst input for that algorithm.

Theorem 14.4 (Yao's Minmax Principle)**.** *Let $\mathcal{I}$ be a finite set of inputs and $\mathcal{A}$ be a finite set of algorithms. For all distributions p over $\mathcal{I}$ and q over $\mathcal{A}$,*

$$\min_{A\in\mathcal{A}} \mathop{E}_{x\in_p\mathcal{I}}[C(x,A)] \leq \max_{x\in\mathcal{I}} \mathop{E}_{A\in_q\mathcal{A}}[C(x,A)]$$

Proof. Let $\alpha = \min_{A\in\mathcal{A}} E_{x\in_p\mathcal{I}}[C(x,A)]$ and $\beta = \max_{x\in\mathcal{I}} E_{A\in_q\mathcal{A}}[C(x,A)]$. Then

$$\forall A_0 \in \mathcal{A} : \mathop{E}_{x\in_p\mathcal{I}}[C(x,A_0)] \geq \alpha \quad \text{and}$$
$$\forall x_0 \in \mathcal{I} : \mathop{E}_{A\in_q\mathcal{A}}[C(x_0,A)] \leq \beta.$$

Note that

$$\begin{aligned}
\mathop{E}_{x\in_p\mathcal{I},A\in_q\mathcal{A}}[C(x,A)] &= \sum_{x_0\in\mathcal{I}}\sum_{A_0\in\mathcal{A}} C(x_0,A_0)\cdot \mathrm{Prob}_{x\in_p\mathcal{I}}(x=x_0)\cdot \mathrm{Prob}_{A\in_q\mathcal{A}}(A=A_0) \\
&= \sum_{A_0\in\mathcal{A}} \mathop{E}_{x\in_p\mathcal{I}}[C(x,A_0)]\cdot \mathrm{Prob}_{A\in_q\mathcal{A}}(A=A_0) \qquad (14.2)\\
&= \sum_{x_0\in\mathcal{I}} \mathop{E}_{A\in_q\mathcal{A}}[C(x_0,A)]\cdot \mathrm{Prob}_{x\in_p\mathcal{I}}(x=x_0). \qquad (14.3)
\end{aligned}$$

From (14.2) we obtain

$$\begin{aligned}
\mathop{E}_{x\in_p\mathcal{I},A\in_q\mathcal{A}}[C(x,A)] &= \sum_{A_0\in\mathcal{A}} \mathop{E}_{x\in_p\mathcal{I}}[C(x,A_0)]\cdot \mathrm{Prob}_{A\in_q\mathcal{A}}(A=A_0) \\
&\geq \sum_{A_0\in\mathcal{A}} \alpha\cdot \mathrm{Prob}_{A\in_q\mathcal{A}}(A=A_0) \\
&= \alpha. \qquad (14.4)
\end{aligned}$$

Similarly, (14.3) implies

$$\begin{aligned}
\mathop{E}_{x\in_p\mathcal{I},A\in_q\mathcal{A}}[C(x,A)] &= \sum_{x_0\in\mathcal{I}} \mathop{E}_{A\in_q\mathcal{A}}[C(x_0,A)]\cdot \mathrm{Prob}_{x\in_p\mathcal{I}}(x=x_0) \\
&\leq \sum_{x_0\in\mathcal{I}} \beta\cdot \mathrm{Prob}_{x\in_p\mathcal{I}}(x=x_0) \\
&= \beta. \qquad (14.5)
\end{aligned}$$

It follows that $\alpha \leq \beta$, which proves the claim. □

14.3 Black-Box Optimization

Black-Box optimization is the problem of finding the optimum of a function, if the only way of gaining information about the function is to query the function value of a point.

We consider a finite *search space*, S_n (often $S_n = \{0,1\}^n$) and a class F_n of functions $f : S_n \to \mathbb{R}$. For example, F_n can be the set of all quadratic functions on S_n, or the class of all monotone functions, etc.

The problem is to find for some "unknown" function $f \in F_n$, the point x such that $f(x)$ is optimal. Throughout this lecture we assume that optimal means *maximal*, but similarly we could try find the minimum. The only operation that reveals information about f is the operation `Query(`x`)`, which returns $f(x)$.

A typical black-box algorithm queries a random point x_0, and obtains $f(x_0)$. For a query history $\big(x_0, f(x_0)\big), \big(x_1, f(x_1)\big), \ldots, \big(x_k, f(x_k)\big)$, the algorithm then decides (according to some probability distribution), which point to query next.

Usually we cannot assume that we know whether a function value is optimal or not.' Therefore, we allow a black-box algorithm to run forever, but we only measure the number of queries until the a point x is queried that maximizes $f(x)$.

Definition 14.5. The *black-box complexity* of a family $\mathcal{F}$ of functions $f : S \to \mathbb{R}$ is

$$\min_{B \in \mathcal{B}} \max_{f \in \mathcal{F}} E[Q(f, B)],$$

where $\mathcal{B}$ is the class of randomized black-box algorithms and $Q(f, B)$ is the random variable that measures the number of queries it takes during a run of B until a point $x^* \in S$ has been queried, such that $f(x^*)$ is maximal.

Definition 14.6. A *Needle in the Haystack Function* over S_n is a function $f : S_n \to \mathbb{R}$, with exactly one point $x_0 \in S_n$ such that $f(x_0) = 1$ and where $f(x) = 0$ for all $x \in S_n - \{x_0\}$.

Let $S_n = \{0,1\}^n$, and N_a be the Needle in the Haystack function over S_n which has its maximum function value in $a \in \{0,1\}^n$.

Theorem 14.7. *Any black-box algorithm for the class $\mathcal{N}_n$ of functions N_a, $a \in \{0,1\}^n$, has black-box complexity at least $2^{n-1} + \frac{1}{2}$.*

Proof: Let n be fixed. We want to prove a lower bound, so we have to choose a probability distribution over inputs, i.e., functions in $\mathcal{N}_n$, and then show a lower bound for the black-box complexity of the best deterministic algorithm, if the input is chosen according to the selected probability distribution. We use the uniform distribution, i.e., each function N_a, $a \in \{0,1\}^n$, is picked with the same probability 2^{-n}.

Now let $\mathcal{A}$ be the class of all deterministic algorithms that do not query the same point twice. Then $\mathcal{A}$ is finite, so we can apply Yao's Min-Max principle. Consider an arbitrary algorithm $A \in \mathcal{A}$. Our goal is to prove a lower bound for the black-box complexity of the algorithm A on a function N_a that is chosen uniformly at random from $\mathcal{N}_n$.

Let T be the random variable that denotes the number of queries A performs for the function N_a. What is the expectation of T?

Whenever A queries a point $x \neq a$, it obviously gains no information, other that that x is not the maximum. Thus, there is a unique sequence $x_1, x_2, \ldots, x_{2^n}$, of the points that A queries in this order, if it doesn't discover the optimum until the last step. The algorithm queries point a in the i-th step, if and only if $a = x_i$. Since a is uniformly distributed over $\{0,1\}^n$, the probability that $a = x_i$ is exactly $1/2^n$ for each $1 \leq i \leq 2^n$. Thus,

$$E[T] = \sum_{1 \leq i \leq 2^n} i \cdot \frac{1}{2^n} = \frac{1}{2^n}(1 + \cdots + 2^n) = \frac{1}{2^n} \cdot \frac{2^n(2^n+1)}{2} = 2^{n-1} + \frac{1}{2}.$$

Hence, there exists a probability distribution over the inputs (i.e., the uniform distribution over $\mathcal{N}_n$), such that every deterministic algorithm needs at least and expected number of $2^{n-1} + \frac{1}{2}$ queries. As a consequence of Yao's Min-Max principle, for every randomized algorithm there is a function in $\mathcal{N}_n$, such that the expected number of queries is at least $2^{n-1} + \frac{1}{2}$. □

How good is the lower bound proved in Theorem 14.7? Consider an algorithm that first fixes a random order $x_1, \ldots, x_{2^n}$, and then queries the points in this order. Then again, the probability that the optimum is found in the i-th step is exactly $1/2^n$. Hence, the expected running time is also

$$\frac{1}{2^n}(1 + \cdots + 2^n) = 2^{n-1} + \frac{1}{2}.$$

So this algorithm is optimal.

14.4 Game Tree Evaluation

See Sections 2.1 and 2.2.3 in the textbook.

A *game tree* is a rooted tree, where each leaf u is labeled with a value $val(u)$. The *level* of a node v, $\ell(v)$ is the length of the path from the root to v. Hence, the root has level 0, its children have level 1, and so on. Each inner node u has a value that is determined recursively as

$$val(u) = \begin{cases} \min\{val(v) \mid v \text{ is a child of } u\} & \text{if } \ell(v) \text{ is even} \\ \max\{val(v) \mid v \text{ is a child of } u\} & \text{if } \ell(v) \text{ is odd.} \end{cases}$$

The game tree corresponds to a game between two players. When the first player makes the first move, this corresponds to choosing one of the root's children, say u_1. Then the second player chooses a child of u_1, and so on. When a leaf is reached, the game is over, and the first player has to pay the amount corresponding to the value of the leaf to the second player. Thus, the value of a node in the game tree corresponds to the optimal payoff for each player, if both players play optimally.

Our goal is to compute the value $val(r)$ of the root r. This tells us how much the first player hast to pay at most, if that player plays an optimal strategy. Here we only consider games, where the payoff is 0 or 1, i.e., it is only about winning or losing the game. (The first player wins, if the outcome is 0, and otherwise the second player wins.) Moreover, we will only consider game trees with an even number of levels and where each non-leaf node has the same number of children.

Definition 14.8. Let $\mathcal{T}_{d,k}$ be the family of trees with the following three properties:

1. Every inner node has exactly d children.
2. All leafs are on level $2k$.
3. For all leafs u, $val(u) \in \{0, 1\}$.

Every tree in $\mathcal{T}_{d,k}$ has $N = d^{2k}$ leaves. Moreover, we can replace min by AND ($\wedge$) and max by OR ($\vee$).

It is easy to see that for every deterministic algorithm, the leaves can be assigned values, such that the algorithm has to read all leaf values. Hence, the worst-case time complexity is N.

14.4.1 A Randomized Algorithm for Game Tree Evaluation

In the following we consider only the case $d = 2$ ($d > 2$ works with essentially the same idea). Let u be an $\wedge$-vertex with children u_0, u_1. We would like to find out, whether either $val(u_0) = 0$ or $val(u_1) = 0$.

We can do this as follows: First, we choose $i \in \{0, 1\}$ at random and recursively compute $val(u_i)$. If $val(u_i) = 0$, we are done, as we know that $\text{val}(u) = \text{val}(u_i) \wedge \text{val}(u_{1-i}) = 0$. Otherwise, if $val(u_i) = 1$, we have to compute $val(u_{1-i})$ and return that value.

Algorithm 10: GameTreeEval

Input: A node v in a tree $T_{2,k}$
Output: $val(v)$

```
1 if v is a leaf then return val(v)
2 Let v_0 and v_1 be the children of v
3 Pick i ∈ {0, 1}
4 t :=GameTreeEval(v_i)
5 if v is a ∨-node and t = 1 then return 1
6 if v is a ∧-node and t = 0 then return 0
7 return GameTreeEval(v_{i−1})
```

The idea for the analysis is as follows: Suppose $val(u) = 0$. Then at most one child has value 0, and so with probability at least $1/2$ we are done after computing the value of one child. Now consider the case $val(u) = 1$. Let v be he parent of u, and let u' be the sibling of u. Then v is an $\vee$-node. In order to evaluate v, we chose a node from u and u' at random to evaluate that node first. We had a probability of $1/2$ of choosing u for evaluation first. If we chose u first, then we don't have to evaluate u'.

Analysis. For a tree $T \in \mathcal{T}_{2,k}$, let $X(T)$ be the number of leaves the algorithm GameTreeEval queries. Let $X_k = \max\{E[X(T)] \mid T \in \mathcal{T}_{2,k}\}$. We prove by induction that $E[X_k] \leq 3^k$.

For $k = 0$, each tree in $\mathcal{T}_{2,k}$ consists only of one leaf. Hence, $E[X_k] = 1 = 3^k$.

Now let $k \geq 1$. Consider a tree $T \in \mathcal{T}_{2,k}$ whose root is an $\wedge$-vertex w with $\vee$-children v_1 and v_2. For $i = 1, 2$, let the two children of v_i be the roots of trees $T_{i,1}, T_{i,2} \in \mathcal{T}_{2,k-1}$. Thus,

$$val(w) = val(v_1) \wedge val(v_2) = \big(val(T_{1,1}) \vee val(T_{1,2})\big) \wedge \big(val(T_{2,1}) \vee val(T_{2,2})\big).$$

For $i, j \in \{1, 2\}$ let $Y_{i,j}$ be the indicator random variable indicating that tree $T_{i,j}$ is evaluated and $Y = Y_{1,1} + Y_{1,2} + Y_{2,1} + Y_{2,2}$.

Claim 14.9. $E[Y] \leq 3$.

Proof. We consider the cases $val(w) = 1$ and $val(w) = 0$:

Case 1: $val(w) = 1$. Then $val(v_1) = val(v_2) = 1$. The algorithm has to call GameTreeEval(v_i) for $i = 1$ and $i = 2$. Fix $i \in \{1, 2\}$. Since $val(v_i) = 1$ and v_i is an $\vee$-node, $val(T_{i,j})$ is 1 for at least one index $j \in \{1, 2\}$. Thus, with probability $1/2$ the tree $T_{i,j}$ gets evaluated first, and then the other tree, $T_{i,1-j}$ won't get evaluated at all. With probability at most $1/2$, both trees get evaluated. Therefore, $E[Y_{i,1} + Y_{i,2}] \leq 2 \cdot 1/2 + 1/2 = 3/2$. Since this is true for $i = 1$ and for $i = 2$, we have $E[Y] \leq 2 \cdot 3/2 = 3$.

Case 2: $val(w) = 0$. Then there is an index $i \in \{1, 2\}$ such that $val(v_i) = 0$. If we choose v_i for evaluation first, then we will not evaluate $val(v_{1-i})$. Hence, with probability at least $1/2$ we will only evaluate one of the two nodes v_1 and v_2. If that happens, we evaluate only two of the sub-trees $T_{i,j}$. With probability at most $1/2$ we may have to evaluate all four sub-trees.1 Thus, $E[Y] \leq (1/2) \cdot 2 + (1/2) \cdot 4 = 3$. □

Now let Z_ℓ be the number of leafs we evaluate in the ℓ-th tree among $T_{i,j}$, $1 \leq i, j \leq 2$ that gets evaluated. By the induction hypothesis, $E[X(T_{i,j})] \leq 3^{k-1}$ for $1 \leq i, j \leq 2$, and thus $E[Z_\ell \,|\, \ell \leq Y] \leq 3^{k-1}$.

We apply Wald's Theorem (Theorem 7.1) to obtain

$$E[X(T)] = E[Z_1 + \cdots + Z_Y] \leq E[Y] \cdot E[Z_\ell \,|\, \ell \leq Y] \leq 3 \cdot 3^{k-1} = 3^k.$$

This proves the induction hypothesis and we get the following result.

Theorem 14.10. *For a tree from $\mathcal{T}_{2,k}$, the expected number of leaves queried by algorithm GameTreeEval is less than $N^{0.793}$.*

Proof: Note that $N = 2^{2k} = 4^k$. We have

$$E[X(T)] \leq 3^k = 2^{k \log 3} = 4^{(k/2) \log 3} = N^{(1/2) \log 3}.$$

And $(\log 3)/2 = 0.79248\ldots$. □

14.4.2 A Lower Bound for Game Tree Evaluation

In the following we prove a lower bound on the running time for randomized algorithms that evaluate game trees. In particular, we prove a lower bound on $E[X]$, where X is the number of of leaves that the optimal randomized algorithm queries.

First note that we can replace all gates in a game tree with an even number of levels with NOR gates:

$$x \barwedge y := \mathrm{NOR}(x, y) := \overline{x \vee y}.$$

Recall the De Morgan's laws:

$$\overline{x \vee y} = \overline{x} \wedge \overline{y} \quad \text{and} \quad \overline{x \wedge y} = \overline{x} \vee \overline{y}$$

Thus,

$$(x_1 \vee x_2) \wedge (x_3 \vee x_4) = \overline{\overline{(x_1 \vee x_2) \wedge (x_3 \vee x_4)}} = \overline{\overline{(x_1 \vee x_2)} \vee \overline{(x_3 \vee x_4)}}$$
$$= (x_1 \barwedge x_2) \barwedge (x_3 \barwedge x_4)$$

If we replace one $\wedge$- and one $\vee$-level at once by two $\barwedge$-levels, the function represented at the root does not change.

W.l.o.g. we can assume that an algorithm never queries the same leaf twice. Then each deterministic algorithm for game tree evaluation corresponds to a sequence of functions: When leaves $\ell_1, \ldots, \ell_k$ have been queried by the algorithm, the next leaf to query is determined by the values $val(\ell_1), \ldots, val(\ell_k)$. Thus, each deterministic algorithm can be described by a sequence of functions $f_0, \ldots, f_{N-1}$, where $f_i : \{0,1\}^i \to \{1, \ldots, N\}$ takes as input the values of the first leaves that were queried and outputs the next leaf to be queried. Since there is only a finite number of such function sequences, the family $\mathcal{A}$ of deterministic algorithms for game tree evaluation is finite, too.

The set of inputs is $\mathcal{I} = \{0,1\}^N$. We have to fix a distribution μ over $\mathcal{I}$ and prove a lower bound for

$$\min_{A \in \mathcal{A}} \underset{x \in_\mu \mathcal{I}}{E}[C(x, A)].$$

We choose an input distribution, where each node of the tree has value 1 with the same probability p. We then show that an optimal algorithm has a special form: The algorithm DFP (*Depth-First-Pruning*) queries leaves from left to right, and skips a leaf, when that one has an ancestor whose value is already determined.

We prove below that a depth-first-pruning algorithm is optimal for all balanced NOR game trees, where each node has value 1 with the same probability p. Hence, we can restrict ourselves to the class of algorithms $\mathcal{A}'$ of depth-first-pruning algorithms if we choose each leaf with a probability p as 0 and 1, so that **every node** will have value 0 and 1 with the same probability p.

If $x = 1$ with probability p and $y = 1$ with probability p, then

$$\text{Prob}(x \barwedge y = 1) = \text{Prob}(x \vee y = 0) = \text{Prob}(x = 0 \wedge y = 0) = (1-p)^2.$$

Thus, we solve for p:

$$\begin{aligned} & p = (1-p)^2 \\ \Leftrightarrow \quad & 0 = p^2 - 3p + 1 \\ \Leftrightarrow \quad & p = \frac{-(-3) \pm \sqrt{(-3)^2 - 4 \cdot 1 \cdot 1}}{2 \cdot 1} \\ \Leftarrow \quad & p = \frac{3 - \sqrt{5}}{2}. \end{aligned}$$

Hence, if we choose $p = (3 - \sqrt{5})/2$, and let each leaf have value 1 with probability p, then every node in the tree has value 1 with probability p. It follows that the DFP algorithm has the lowest expected running time for that tree.

Let $Q(h)$ be the number of leaves that the DFP algorithm queries in such a tree of height h. If $h = 0$, then $Q(h) = 1$, so $E[Q(h)] = 1$. Now suppose $h \geq 1$, and let the root of such a tree be r, and let u_0 and u_1 be the left and right child of r, respectively. The DFP algorithm first evaluates leaves in the sub-tree rooted at u_0. The expected number of leaves to evaluate in this sub-tree is $E[Q(h-1)]$. With probability p, $val(u_0) = 1$, and in this case $val(r) = 0$ is determined, so u_1 needs not be evaluated. With probability $1-p$, $val(u_0) = 0$, and in this case u_1 also needs to be evaluated. In this case the expected number of leaf queries in the sub-tree rooted at u_1 is again $E[Q(h-1)]$. Thus, we obtain the following recurrence:

$$\begin{aligned} E[Q(h)] = E[Q(h-1)] + (1-p) \cdot E[Q(h-1)] = (2-p) \cdot E[Q(h-1)] = (2-p)^2 \cdot E[Q(h-2)] \\ = \cdots = (2-p)^h \cdot E[Q(0)] = (2-p)^h. \end{aligned}$$

If the tree has N leaves, then N is a power of two and $h = \log N$. It follows that

$$E[Q(\log N)] = (2-p)^{\log N} = N^{\log(2-p)} \geq N^{0.6942}.$$

It remains to prove that a DFP algorithm is optimal.

Lemma 14.11. *Let T be a balanced NOR game tree, where each node has value 1 with probability exactly p (for some constant $0 \leq p \leq 1$). If there is a game tree algorithm for T that needs $Q(T)$ queries, then the expected number of queries by the depth-first-pruning algorithm is at most $E[Q(T)]$.*

Proof. A *depth-d straight* algorithm is an algorithm that behaves as a depth-first-pruning algorithm for all subtrees that have depth d. More precisely, if a depth-d straight algorithm queries a leaf in a subtree T_v with root v, where T_v has depth d, then it determines the value of v before it queries any leaf outside of T_v. Clearly, it suffices to show that if T is a tree of depth $2k$, then there is an optimal depth-$2k$ straight algorithm that evaluates T.

By induction, we prove that there is an optimal depth-d straight algorithm for all $d \geq 0$. For $d = 0$, all subtrees of depth d consist only of a leaf, so the claim is trivially true.

Now let A be an optimal depth-$(d-1)$ straight algorithm. Consider a run of A and assume that A queries a leaf in a subtree of depth d rooted at some node w. Let a, b be the children of w. Suppose A queries some leaves in the subtree T_a rooted at a, but then queries some leaves outside of T_a before $\text{val}(w)$ is determined. Since A is depth-$(d-1)$ straight, it determines $\text{val}(a)$ completely, but does not query a single leaf in T_b before querying a leaf outside of T_w. Note that $\text{val}(a) = 0$ if $\text{val}(w)$ is not determined after evaluating a.

In the following we break up A into several parts, and then re-assemble these parts in order to obtain two new algorithms, B and C. Algorithm B does not query a in the beginning, but instead "assumes" $\text{val}(a) = 0$, and acts like A does in this case. Only when algorithm A evaluates b and thus w later on, algorithm B has to evaluate both a and b one after the other. Algorithm C evaluates a and b at the very beginning, and then behaves like A does once it knows these values. We then prove that either B or C is optimal. Clearly, B and C are depth-d-straight on the subtree T_w. Therefore, if we apply the same technique for every subtree of depth d, we obtain an optimal algorithm that is depth-d-straight.

First we describe how to break up A in several parts. Suppose $\text{val}(a) = 1$, and let A_1 be the queries A performs after it has determined $\text{val}(a) = 1$ (and thus $val(w) = 0$), until it has evaluated the root of the game-tree. (More precisely, A_1 is a probability distribution over all possible leaf query sequences, that A performs if $\text{val}(a) = 1$.) Now suppose $\text{val}(a) = 0$. Then with some probability $1 - q$, A will be able to finish the algorithm without evaluating b at all. Let A_2 be the queries A executes in order to finish the algorithm in this case. (To be precise, now A_2 is a conditional probability distribution over all leaf query sequences, given that the input is such that A does not need to go

back and evaluate b.) With probability q, A will go back to evaluate b. Let A_3 be the query sequence of A until it evaluates b, and let A_4 and A_5 be the remaining leaf query sequences by A, given that $\text{val}(b) = 1$ and $\text{val}(b) = 0$, respectively.

Given this algorithm A, we can define the new algorithms, B and C as described by the flowcharts on p. 96. Algorithm B skips evaluating a in the beginning, and acts as A after A has discovered that $\text{val}(a) = 0$. I.e., it either executes the query sequence A_2, if the game tree is such that A would not go back to evaluate b, or the query sequence A_3 in case A would go back. In the former case, B finishes the algorithm. If B executes A_3 instead, then it will eventually have to evaluate w, so it first evaluates a. If $\text{val}(a) = 1$, then B continues as A would in the case A found $\text{val}(w) = 0$ after executing A_3 and then evaluating b. Hence, B executes A_4. IF $\text{val}(a) = 0$, then B still has to evaluate b. After that it is exactly in the same situation as A after it has evaluated b, and thus can proceed as A by either executing A_4 or A_5.

Algorithm C first evaluates a and if $\text{val}(a) = 1$ it continues with A_1 (as A). But if $\text{val}(a) = 0$, then C evaluates b right away. If it determines $\text{val}(b) = 1$, then it is in the same situation as when $\text{val}(a) = 1$, so it can proceed with A_1. Otherwise, it proceeds as A does for $\text{val}(a) = 0$, i.e., it either executes the query sequence A_2 or A_3, depending on whether the input is such that A goes back to evaluate b or not. If C executes A_2, then it is done. In the case of A_3, algorithm C now already "knows" that $\text{val}(b) = 0$ and thus can proceed as A in this case, namely with A_5.

The probabilities for each query sequence to be executed can be read off the flow chart on page 96. (The label at the top left of a node marks the probability that this node is being reached.) In the table below, we list for each algorithm and each query sequence part, the probability that this query sequence is executed. Moreover, we list the differences of these probabilities between algorithms B and A and algorithms C and A.

	A	B	C	$B - A$	$C - A$
eval. a	1	q	1	$-(1-q)$	0
eval. b	$q(1-p)$	$q(1-p)$	$1-p$	0	$(1-p)(1-q)$
A_1	p	0	$p+p(1-p)$	$-p$	$p(1-p)$
A_2	$(1-p)(1-q)$	$1-q$	$(1-q)(1-p)^2$	$p(1-q)$	$-p(1-p)(1-q)$
A_3	$q(1-p)$	q	$q(1-p)^2$	pq	$-pq(1-p)$
A_4	$pq(1-p)$	$pq+pq(1-p)$	0	pq	$-pq(1-p)$
A_5	$q(1-p)^2$	$q(1-p)^2$	$q(1-p)^2$	0	0

Now let Q_i be the random variable that describes the number of queries in A_i, and let γ be the expected number of queries to evaluate the root of a subtree at depth $d-1$

(e.g., a or b). If X_A, X_B, and X_C are the number of queries that algorithm A, B, and C perform, respectively, then their expectations can be determined as the weighted sum of the values in the corresponding columns, where the weights are chosen according to the expected number of queries performed in the query sequence corresponding to the rows. For example, we have

$$E[X_A] = 1 \cdot \gamma + q(1-p)\gamma + pE[Q_1] + (1-p)(1-q)E[Q_2] + q(1-p)E[Q_3] \\ + pq(1-p)E[Q_4] + q(1-p)^2E[Q_5].$$

Similarly, we can read off $E[X_B]$, $E[X_C]$, and

$$E[X_B - X_A] = -(1-q)\gamma - pE[Q_1] + p(1-q)E[Q_2] + pqE[Q_3] + pqE[Q_4],$$

and

$$E[X_C - X_A] = (1-p)(1-q)\gamma + p(1-p)E[Q_1] - p(1-p)(1-q)E[Q_2] \\ - pq(1-p)E[Q_3] - pq(1-p)E[Q_4].$$

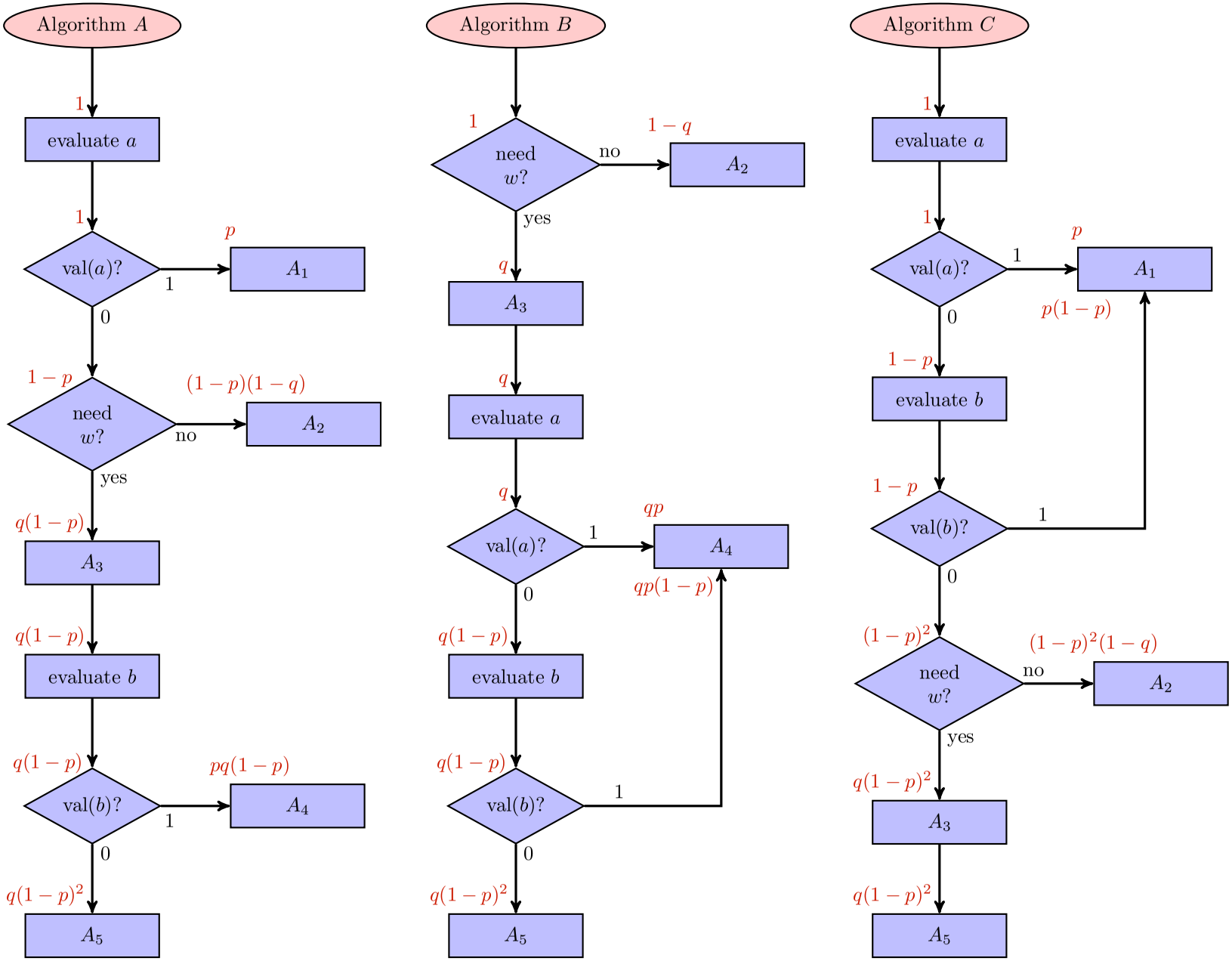
Now we multiply $E[X_B - X_A]$ with $1-p$ and add it to $E[X_C - X_A]$ to obtain

$$E[(1-p)(X_B - X_A) + X_C - X_A] = 0.$$

Hence,

$$(1-p)E[X_B] + E[X_C] = (1-p)E[X_A] + E[X_A].$$

But this means either $E[X_B] \leq E[X_A]$ or $E[X_C] \leq E[X_A]$. Therefore, since A is assumed to be optimal, at least one of B and C is optimal, too. Since B and C are both depth-d straight, the induction hypothesis is proven. □


Algorithm A
1
evaluate a
1
val(a)?
p
A_1
1
0
$1-p$
need w?
$(1-p)(1-q)$
A_2
no
yes
$q(1-p)$
A_3
$q(1-p)$
evaluate b
$q(1-p)$
val(b)?
$pq(1-p)$
A_4
1
0
$q(1-p)^2$
A_5
Algorithm B
1
need w?
$1-q$
A_2
no
yes
q
A_3
q
evaluate a
q
val(a)?
qp
A_4
1
$qp(1-p)$
0
$q(1-p)$
evaluate b
$q(1-p)$
val(b)?
1
0
$q(1-p)^2$
A_5
Algorithm C
1
evaluate a
1
val(a)?
1
p
A_1
$p(1-p)$
0
$1-p$
evaluate b
$1-p$
val(b)?
1
0
$(1-p)^2$
need w?
$(1-p)^2(1-q)$
A_2
no
yes
$q(1-p)^2$
A_3
$q(1-p)^2$
A_5

15 Maximum Satisfiability and Derandomization

A Randomized Approximation Algorithm for MaxSAT. See Section 5.2 in the textbook.

Derandomization - The Method of Conditional Probabilities The method of conditional probabilities is a method that allows us to derandomize randomized algorithms if they have certain properties. We illustrate this with an example; Additional, more general explanations can be found in Section 5.6 in the texbook.

Theorem 15.1. *There is a polynomial time deterministic algorithm that finds for any* Max-3-*SAT input with m clauses an assignment that satisfies at least $(7/8)m$ clauses.*

Proof. Let the input be a collection of m clauses $C_1, \ldots, C_m$ over n variables $x_1, \ldots, x_n$. Consider the randomized algorithm that assigns each literal x_i with probability $1/2$ the value 0 and with probability $1/2$ the value 1. Then for each $1 \leq i \leq m$, clause C_i is satisfied with probability $7/8$. Thus, the expected number of clauses that are satisfied is $(7/8)m$.

Now we *derandomize* this algorithm. We assign Boolean values to the variables $x_1, \ldots, x_n$ (in this order) in such a way, that when we have assigned the values $a_1, a_2, \ldots, a_i$ to the first i variables, then if we would pick random values for the remaining variables, the expected number of satisfied clauses is still at least $(7/8)m$.

More precisely, let X denote the number of clauses that are satisfied by a random assignment. We have seen above that

$$E[X] = (7/8)m.$$

Our goal is to find an assignment $a_1 \in \{0, 1\}$ for the variable x_1, so that if we let $x_1 = a_1$, then the expected number of clauses satisfied by a random assignment to the remaining clauses is still $(7/8)m$. More precisely, we would like to determine a_1 so that

$$E[X|x_1 = a_1] \geq (7/8)m.$$

Such a value $a_1 \in \{0, 1\}$ exists, because

$$\frac{7}{8} \cdot m \leq E[X] = E[X|x_1 = 0] \cdot \frac{1}{2} + E[X|x_1 = 1] \cdot \frac{1}{2},$$

so either $E[X|x_1 = 0] \geq (7/8)m$ or $E[X|x_1 = 1] \geq (7/8)m$.

But how do we know which one is the right one? It turns out that it is easy to compute $E[X|x_1 = a_1]$ efficiently: The probability p_j that a clause C_j is satisfied by a random assignment to the remaining variables $x_2, \ldots, x_n$, given $x_1 = a_1$ is 1 if the clause contains the literal x_1, and it is $1-1/2^k$, if the clause contains exactly k literals that are neither x_1 or $\overline{x_1}$. We can easily compute p_j for $j = 1, \ldots, m$, and then $E[X|x_1 = a_1] = p_1 + \cdots + p_m$.

Hence, in polynomial time we can find an assignment a_1 for variable x_1, such that $E[X|x_1 = a_1] \geq (7/8)m$. In the same way we can then determine $a_2 \in \{0, 1\}$ such $E[X|x_1x_2 = a_1a_2] \geq (7/8)m$, and so on until we have found an assignment $a_1, a_2, \ldots, a_n$ such that $E[X|x_1x_2 \ldots x_n = a_1a_2 \ldots a_n] \geq (7/8)m$. Then clearly, the number of clauses satisfied by this assignment is at least $(7/8)m$. It is obvious that the algorithm has polynomial running time. □

16 Bloom Filters and Negative Associated Random Variables

16.1 Chernoff Bounds for Negative Dependence

Sometimes we need strong tail bounds for the sums of random variables that are not completely independent, but almost independent. Suppose we throw m balls into n bins and we would like to determine the number of covered bins, i.e., the number of bins that receive at least one ball. Let X_i be the indicator random variable with value 1 if and only if bin i receives a ball and $X = \sum_{1 \leq i \leq n} X_i$ the number of covered bins. Then $E[X_i] = (1 - 1/n)^m$ and thus $E[X] = n \cdot (1 - 1/n)^m$.

We would like to apply tail bounds to obtain that with high probability X does not deviate much from its expectation. However, we cannot apply Chernoff Bounds immediately because the random variables X_i are not independent (if $X_i = 1$, then the probability that $X_j = 1$, $j \neq i$, is smaller than if $X_i = 0$, because the event $X_i = 1$ implies that one of the n balls falls in bin i).

But the dependence of these random variables is of a special type: If one of them (or a subset of them) has a "high value" (in this case 1), then the probability other random variables have a high value is smaller. This kind of dependence is "negative". For such a dependence we can still use strong concentration bounds, as we will show in this section.

Definition 16.1 (Negative Association)**.** Random variables $X_1, \ldots, X_n$ are *negatively associated* if for any two disjoint sets $I, J \subseteq \{1, \ldots, n\}$ and all functions f and g that are both non-increasing

$$E\big[f(X_i, i \in I) \cdot g(X_j, j \in J)\big] \leq E[f(X_i, i \in I)] \cdot E[g(X_j, j \in J)]. \tag{16.1}$$

The main observation is that Chernoff Bounds still hold if the underlying indicator variables are negatively independent

Theorem 16.2 (Chernoff Bounds with Negative Association)**.** *Let $X_1, \ldots, X_n \in \{0,1\}$ be negatively associated random variables and let $X = \sum_{1 \le i \le n} X_i$ and $\mu = E[X]$. Then the Chernoff Bounds from Theorems 8.4 and 8.5 hold. In particular, for all $0 < \varepsilon \le 1$*

$$\text{Prob}(X > (1+\varepsilon)\mu) \le e^{-\mu\varepsilon^2/3}, \quad \textit{and} \tag{16.2}$$

$$\text{Prob}(X < (1+\varepsilon)\mu) \le e^{-\mu\varepsilon^2/2}. \tag{16.3}$$

In these notes we will only prove (16.2. We use the following simple observation.

Lemma 16.3. *Let $X_1, \ldots, X_n$ be negatively associated. Then for any non-decreasing and non-negative functions f_i, $i \in \{1, \ldots, n\}$*

$$E\left[\prod_{1 \le i \le n} f_i(X_i)\right] \le \prod_{1 \le i \le n} E[f_i(X_i)].$$

Proof. We prove the lemma by induction on n. For $n = 1$ the claim states that $E[f_1(X_1)] \le E[f_1(X_1)]$, which is obviously true.

Now suppose $n > 1$ and we have proved the induction hypothesis for $n-1$ variables, i.e.,

$$E\left[\prod_{1 \le i \le n-1} f_i(X_i)\right] \le \prod_{1 \le i \le n-1} E[f_i(X_i)]. \tag{16.4}$$

Define $I := \{1, \ldots, n-1\}$ and $J = \{n\}$ and

$$f(X_i, i \in I) := \prod_{1 \le i \le n-1} f_i(X_i) \quad \text{and} \quad g(X_j, j \in J) := f_n(X_n).$$

Then f and g are non-decreasing, since the functions f_i, $1 \le i \le n$, are non-decreasing. Hence, by the definition of negative dependence we have

$$\begin{aligned} E\left[\prod_{1 \le i \le n} f_i(X_i)\right] &= E[f(X_i, i \in I) \cdot g(X_n)] \le E[f(X_i, i \in I] \cdot E[g(X_n)] \\ &= E\left[\prod_{1 \le i \le n-1} f_i(X_i)\right] \cdot E[f_n(X_n)] \overset{(16.4)}{\le} \prod_{1 \le i \le n-1} E[f_i(X_i)] \cdot E[f_n(X_n)] \\ &= \prod_{1 \le i \le n} E[f_i(X_i)]. \end{aligned}$$

□

Proof of Theorem 16.2. Consider the proof of Theorem 8.4. We defined $Y_i = e^{t \cdot X_i}$ for $1 \leq i \leq n$ and some positive value t. To prove an inequality in (8.5), we used that the random variables X_i and thus the variables Y_i are all independent, and thus

$$E[\prod_{1 \leq i \leq n} Y_i] = \prod_{1 \leq i \leq n} E[Y_i].$$

Nowhere else in the proof is the independence of the random variables X_i required. But if the random variables X_i are negatively associated, then applying Lemma 16.3 for the functions $f_i(X_i) = Y_i = e^{t \cdot X_i}$ yields

$$E[\prod_{1 \leq i \leq n} Y_i] \leq \prod_{1 \leq i \leq n} E[Y_i].$$

Hence, the inequality stated in (8.5) remains true. □

16.2 Properties of Negative Dependence

In the following we present some simple properties of negatively associated random variables. The proofs follow easily from the definition of (conditional) expectation and linearity of expectation and are omitted in these notes.

Lemma 16.4 (Closure under Products)**.** *Let $X_1, \ldots, X_n$ and $X_{n+1}, \ldots, X_m$ be two independent families of random variables that are each negatively associated. Then the family $X_1, \ldots, X_m$ is also negatively associated.*

Lemma 16.5 (Disjoint Monotone Aggregation)**.** *Let $X_1, \ldots, X_n$ be negatively associated random variables and $J_1, \ldots, J_k$ be disjoint subsets of $\{1, \ldots, n\}$, and $f_1, \ldots, f_k$ be non-decreasing functions. Then the random variables*

$$Y_j = f_j(X_i, i \in J_i), \quad 1 \leq j \leq k$$

are negatively associated.

Remark 16.6. Consider two real valued functions f and g. Let $f'(x) = a \cdot f(x) + b$ and $g'(y) = b \cdot g(y) + c$ for some fixed values $a, b, c, d \in \mathbb{R}$, where a and c have the same sign (i.e., $a \cdot c \geq 0$). A simple calculation shows the following implication:

$$E[f(x) \cdot g(y)] \leq E[f(x)] \cdot E[g(y)] \;\Leftrightarrow\; E[f'(x) \cdot g'(y)] \leq E[f'(x)] \cdot E[g'(y)]. \qquad (16.5)$$

Hence, in order to show that the sequence $X_1, \ldots, X_n$ of random variables is negatively associated it suffices to prove inequality (16.1) for all functions f and g which have the following properties:

1. $f(0\ldots0) = g(0\ldots0) = 0$, and
2. f and g are both non-decreasing.

This can be seen as follows: Suppose we need to prove statement (16.1) for functions f and g which don't satisfy both of these properties. We let $a = c = 1$ if f and g are both non-decreasing and $a = b = -1$ if f and g are both non-increasing. Then we define $b = -a \cdot f(0,\ldots,0)$ and $d = -c \cdot g(0,\ldots,0)$. Defining the functions $f' = a \cdot f + b$ and $g' = c \cdot g + d$ we have $f'(0,\ldots,0) = a \cdot f(0,\ldots,0) - a \cdot f(0,\ldots,0) = 0$, and similarly $g'(0,\ldots,0) = 0$. Moreover, f' and g' are both non-decreasing. Hence, if we prove that f' and g' satisfy (16.1), it follows from (16.5) that f and g satisfy the property as well.

16.3 Example I: Balls Into Bins

Consider the example of throwing m balls into n bins. Let $B_{i,k}$, $1 \le i \le n$, $1 \le k \le m$, be the indicator random variable which is 1 if and only if ball k falls in bin i.

Proposition 16.7. *For each $k \in \{1,\ldots,m\}$ the random variables $B_{1,k},\ldots,B_{n,k}$ are negatively associated.*

Proof. Let $I, J \subseteq \{1,\ldots,n\}$ be disjoint and let f, g be either both non-decreasing or both non-increasing functions. By Remark 16.6 we can assume w.l.o.g. that f and g are both non-decreasing and $f(0,\ldots,0) = g(0,\ldots,0) = 0$. In particular, f and g are non-negative. Note that $B_{i,k} = 1$ implies $B_{j,k} = 0$ for all $j \neq i$: if ball k falls in bin i then it cannot fall in bin $j \neq i$. Hence, if one of the random variables $B_{i,k}$, $i \in I$, has value 1 then all random variables $B_{j,k}$, $j \in J$, must have value 0. It follows that if $f(B_{i,k}, i \in I) > 0$ then $g(B_{j,k}, j \in J) = 0$. In particular, since f and g are non-negative, $f(B_{i,k}, i \in I) \cdot g(B_{j,k}, j \in J) = 0$. We conclude

$$E[f(B_{i,k}, i \in I) \cdot g(B_{j,k}, j \in J)] \le f(B_{i,k}, i \in I)] \cdot E[g(B_{j,k}, j \in J)]$$

□

Now let B_i, $1 \le i \le n$, be the number of balls that fall in bin i, and let T_i, $1 \le i \le n$, be the indicator variable which is 1 if and only if at least one ball falls in bin i.

Proposition 16.8. *The random variables $B_1,\ldots,B_n$ are negatively associated, and so are the random variables $T_1,\ldots,T_n$.*

Proof. From Proposition 16.7 we have that for each $1 \leq k \leq m$ the random variables $B_{i,k}$, $1 \leq i \leq n$, are negatively associated. By closure under products, all random variables $B_{i,k}$, $1 \leq i \leq n, 1 \leq k \leq m$ are negatively associated. Now the claim follows immediately from disjoint monotone aggregation using

$$B_i = B_{1,1} + \cdots + B_{1,k}$$

and

$$T_i = \max\{B_{1,1}, \cdots + B_{1,k}\},$$

and noting that the mappings $(x_1, \ldots, x_k) \mapsto x_1 + \cdots + x_k$ and $(x_1, \ldots, x_k) \mapsto \max\{x_1, \ldots, x_k\}$ are non-decreasing. □

16.4 Example II: Bloom Filters

[TO DO]

17 Routing on a Hypercube

See the textbook, Section 4.2. These notes are partly based on a draft provided by Lisa Higham.

In a parallel computer, where processing elements are connected through communication links, often communication delay overwhelms local computation time. Bottlenecks in the communication make this delay even greater. Hence, we are motivated to design schemes that decrease the likelihood that packets, moving between different source and destination pairs, try to traverse the same communication link at the same time.

Model. We model a parallel computer as a directed graph where nodes represent processes and there is a directed edge from node u to node v, if there is a communication link from the process corresponding to u to that corresponding to v. Communication proceeds in synchronous rounds: in each round each process can send at most one packet of communication on each of its outgoing links. To this end, every directed edge contains a queue. Once a packet reaches node v, the packet is inserted in the queue Q_e of an edge e leaving v, and in each subsequent round exactly one packet from Q_e is routed over that edge.

One common topology for modeling aparallel computer is a hypercube. The *n-dimensional hypercube* is the undirected graph $H = (V, E)$ where

$$V = \Big\{ b_1 b_2 \dots b_n \;\Big|\; b_i \in \{0,1\} \Big\}, \quad \text{and}$$
$$E = \Big\{ \big(b_1 \dots b_{i-1} b_i b_{i+1} \dots b_n,\ b_1 \dots b_{i-1} \overline{b_i} b_{i+1} \dots b_n\big) \;\Big|\; 1 \le i \le n \Big\}.$$

I.e., each node is represented by a binary string of length n, and two nodes with binary representations $b_1 \dots b_n$ and $c_1 \dots c_n$ are adjacent, if their binary representations differ in exactly one bit.

The n-dimensional hypercube has $N = 2^n$ nodes and $n \cdot N/2$ undirected edges, and its diameter is $n = \log N$. To model our parallel computer, we consider the directed graph obtained by replacing each undirected edge in the hypercube with two directed edges in opposite directions, thus accounting for $n \cdot N$ directed edges.

Function HypercubeRouting($b_1 \dots b_n,\ d_1 \dots d_n$)

Let $c_1 \dots c_n := b_1 \dots b_n$.
while $c_1 \dots c_n \neq d_1 \dots d_n$ **do**
  Let j be the smallest index such that $c_j \neq d_j$.
  Send pkt_v over the edge $(c_1 \dots c_{j-1} c_j c_{j+1} \dots c_n,\ c_1 \dots c_{j-1} \overline{c_j} c_{j+1} \dots c_n)$.
end

Figure 17.1: The Bit Fixing Algorithm

The Permutation Routing Problem. Consider a packet pkt_v originating at process v, and let $D(v)$ denote its destination. A *route* for pkt_v is a directed path in the graph from v to $D(v)$. A routing algorithm must specify a route for each packet. Furthermore, when the routes of two or more packets use the same edge e in the same round, all but one of these packets must be delayed. So an algorithm must also specify a queuing discipline for each queue Q_e, which determines how to schedule the traversal of the packets across the common edge e.

A routing algorithm is *oblivious* if, for every process v, the route specified for a packet originating at v is independent of $D(u)$ for all $u \neq v$. We consider *permutation routing*, where the function $D : V \mapsto V$ is a bijection.

A Deterministic Permutation Routing Scheme. A simple oblivious deterministic hypercube routing algorithm uses a "bit-fixing" strategy (see Figure 17.1): To route a packet pkt_v from $v = b_1 \dots b_n$ to $D(v) = d_1 \dots d_n$ the algorithm considers each bit in order from 1 through n, and routes the packet along the edge that "fixes" the bit.

Clearly, any packet will reach its destination under the algorithm in Figure 17.1 after travelling along at most n edges. Nevertheless, this simple scheme can have very bad performance for the n-dimensional hypercube because of congestion at nodes. Define the *transpose* mapping given by

$$D(b_1 \dots b_n) = b_{\lfloor n/2 \rfloor + 1} \dots b_n b_1 \dots b_{\lfloor n/2 \rfloor}.$$

Notice that D is a permutation.

It is not hard to see that for the permutation D and the algorithm `HypercubeRouting()` in Figure 17.1 (using any queuing discipline) some packets require $\Omega(\sqrt{N})$ steps: W.l.o.g. let n be even. Consider the set U of the nodes in $1\{0,1\}^{n/2-1}0^{n/2}$, i.e, the nodes

Function RandomRouting(v, D)

```
  /* Use FIFO (first-in first-out) for the queuing discipline, with
     ties broken arbitrarily                                        */
1 Pick independently, uniformly and at random a destination, σ, from {1,...,N}
2 HypercubeRouting(v, σ)                                  /* Phase 1 */
3 wait until at least 4n rounds in total have elapsed
4 HypercubeRouting(σ, D(v))                               /* Phase 2 */
```

Figure 17.2: A randomized routing algorithm.

$1b_2 \dots b_{n/2}0 \dots 0$ for $b_2, \dots, b_{n/2} \in \{0,1\}$. The packets pkt_v, $v \in U$, have destinations $D(v) = 0 \dots 01b_2 \dots b_{n/2}$ for some bits $b_2, \dots, b_{n/2} \in \{0,1\}$. Thus, each of them reaches the node $0^n = 0 \dots 0$ after "fixing" the first $n/2$ bits, and leaves that node via the edge $e^* = (0^n, 0^{n/2}10^{n/2-1})$. Hence, all $2^{n/2-1}$ packets pkt_u, $u \in U$, have to get routed through the same edge e^*. Since in every round at most one packet can traverse edge e^*, the total routing time must be at least $2^{n/2-1} = \sqrt{N}/2$.

Note that this lower bound is independent of the queuing discipline used. It is known that no deterministic permutation routing algorithm can be fast. More precisely, for any oblivious deterministic permutation routing algorithm for the an n-dimensional hypercube (with $N = 2^n$ nodes) there is a permutation that requires $\Omega\sqrt{N/n}$ steps.

A Randomized Permutation Routing Scheme. A very simple randomization of the deterministic routing scheme can lead to an exponential speedup: We use the deterministic bit-fixing strategy to route each packet to a randomly chosen intermediate node, and then to route it from the intermediate node to the final destination. See Figure 17.2 for the algorithm `RandomRouting()`. As a queuing discipline we use a fist-in first-out (FIFO) strategy, where the packets leave each queue Q_e in the order in which they were inserted. Packets that reach the start point v of edge e in the same round are inserted in the queue in arbitrary order. (Several other queuing disciplines are known work, too.)

In the remainder of this section we prove that with high probability the algorithm `RandomRouting()` routes every packet to its destination in fewer than $8n = 8 \log N$ rounds.

First, we analyze Phase 1 in which every packet is routed to its intermediate target. For any packet v, let $\sigma(v)$ denote the randomly chosen intermediate node for pkt_v. Let $\rho(v)$

denote the route of pkt_v in Phase 1 of algorithm `RandomRouting()`, expressed as the sequence of edges traversed.

The following lemma says that if two routes meet at some node and later diverge, they will not meet again in Phase 1.

Lemma 17.1. *Suppose the routes for two packets, say pkt_a and pkt_b have a node, $w \notin \{\sigma(a), \sigma(b)\}$, in common, but the node v_a that immediately succeeds w in a's route is not the node v_b that immediately succeeds w in b's route. (I.e., there is an edge $\{w, v_a\} \in \rho(a)$ and an edge $\{w, v_b\} \in \rho(b)$ s.t. $a \neq b$.) Then there is no node v that pkt_a and pkt_b both traverse after w.*

Proof. For a node v let $\text{bit}_i(v)$ denote the i-th bit in the binary representation of v. More precisely, if $v = v_1 \dots v_n$, then $\text{bit}_i(v) = v_i$. There is exactly one bit i such that $\text{bit}_i(w) \neq \text{bit}_i(v_a)$. Similarly, there is exactly one bit j such that $\text{bit}_j(w) \neq \text{bit}_j(v_b)$. Since $v_a \neq v_b$, obviously $i \neq j$.

W.l.o.g. assume that $i < j$ and $\text{bit}_i(w) = 0$. Then a and w agree in the first $i-1$ bits, i.e., $\text{bit}_\ell(w) = \text{bit}_\ell(a)$ for $1 \leq \ell < i$ and b and w agree in the first $j-1 \geq i$ bits, i.e., $\text{bit}_k(w) = \text{bit}_k(b)$ for $1 \leq k \leq i$. Moreover, by the bit-fixing strategy all edges that are used on path $\rho(a)$ after node v_a will not change the value of any of the first $i-1$ bits anymore. Hence, every node on path $\rho(a)$ after node w has the prefix $a'_1 \dots a'_{i-1} a'_i = w_1 \dots w_{i-1} 1$. Similarly, no edge that is used on path $\rho(b)$ after node w will change the value of any of the first i bits. Hence, every node on the path $\rho(b)$ after node w has the prefix $b'_1 \dots b'_i = w_1 \dots w_{i-1} 0$. Since all nodes after w on path $\rho(a)$ have a different prefix of length i than the nodes after w on $\rho(b)$, the lemma is true. □

The following lemma states that the total number of times a packet pkt_v has to wait in a queue is bounded by the number of packets that "cross" its path, i.e., the number routes of other packets that have at least one edge with $\rho(v)$ in common.

Lemma 17.2. *Fix a vertex v and let*

$$S = \{pkt_u \mid \rho(u) \cap \rho(v) \neq \emptyset\}.$$

Then pkt_v reaches its destination in at most $k + |S|$ rounds, where k is the length of $\rho(v)$.

Proof. Let $\rho(v) = (e_1, \dots, e_k)$. Consider the beginning of some round t in which packet v is located in queue $Q_{e_{R_t}}$, $1 \leq R_t \leq k$. For every edge e_i, $i \in \{1, \dots, k\}$, we define a value $\text{val}_t(e_i)$ as follows:

- if $i \in \{1, \ldots, R_t - 1\}$ then $\text{val}_t(e_i) = 0$;
- if $i = R_t$ then $\text{val}_t(e_i)$ is the number of packets in queue Q_{e_i} which are not behind pkt_v;
- if $i \in \{R_t + 1, \ldots, k\}$ then $\text{val}_t(e_i) = \max\{1, |Q_{e_i}|\}$.

Let S_t denote the set of packets in S which have not been entered in any queue Q_{e_i}, $1 \leq i \leq k$, prior to round t. We define $\Phi_t = |S_t| + \sum_{1 \leq i \leq k} \text{val}_t(e_i)$.

We prove for all t the following claim:

$$\text{By the end of round } t + \Phi_t - 1 \text{ packet } \text{pkt}_v \text{ has reached its target.} \qquad (*)$$

At the beginning of round 1 every queue Q_{e_i}, $1 \leq i \leq k$, contains at most 1 element and thus has value 1. Moreover, $S_1 \subseteq S$, so we have $\Phi_1 \leq |S| + k$. Hence, the lemma follows from $(*)$ for $t = 1$. Thus, it suffices to prove Claim $(*)$.

First consider the case $\Phi_t = 1$. Note that $\text{val}_t(e_i) \geq 1$ for all $i \in \{R_t, \ldots, k\}$. Then Φ_t is at least $R_t - k + 1$, i.e., the number of edges that pkt_v still has to traverse. Hence, if $\Phi_t = 1$ then $R_t = k$. Moreover, $\text{val}_t(e_k)$ can be at most 1, so pkt_v must be the first element in queue Q_{e_k}, and thus it will traverse the last edge on its path in round t.

Now suppose $\Phi_t > 1$. Consider some queue Q_{e_i}, $i > R_t$, and suppose $\text{val}_{t+1}(e_i) = \text{val}_t(e_i) + \delta$ for some value $\delta > 0$. We argue that δ packets from S_t get added to Q_{e_i} in round t: If Q_{e_i} is not empty at the beginning of round t, then in round t queue Q_{e_i} loses one packet (the first packet in the queue gets routed over edge e_i), and it may receive at most one packet from another queue Q_{e_j}, $j < R_t$, namely from queue $Q_{e_{i-1}}$. If Q_{e_i} is empty at the beginning of round t, then it does not lose a packet, but one packet that may be entered from some the queue $Q_{e_{i-1}}$ does not affect the value of the queue, as even with 0 packets its value is 1. Hence, in either case there are δ packets in S which are added to Q_{e_i} in round t, and which were not in one of the queues Q_{e_j}, $j \neq i$, at the beginning of the round. By Lemma 17.1, these δ packets cannot have been in any queue Q_{e_ℓ}, $1 \leq \ell \leq k$, in any previous round, so the δ packets are in S_t (but not in S_{t+1}). To conclude, for every increase of the value of a queue Q_{e_i}, $i > R_t$, there is a unique packet in $S_t - S_{t+1}$. Hence, we have

$$\sum_{R_t \leq i \leq k} \big(\text{val}_{t+1}(e_i) - \text{val}_t(e_i)\big) \leq |S_t| - |S_{t+1}|,$$

which implies

$$|S_{t+1}| + \sum_{R_t \leq i \leq k} \text{val}_{t+1}(e_i) \leq |S_t| + \sum_{R_t \leq i \leq k} \text{val}_t(e_i). \qquad (17.1)$$

Now consider queue $Q_{e_{R_t}}$. In round t packet pkt_v either leaves that queue and traverses the edge e_{R_t}, which decreases the value of the queue from 1 to 0, or it moves one position up in the queue, decreasing its value by 1. In either case, $\text{val}_{t+1}(e_{R_t}) = \text{val}_t(e_{R_t}) - 1$. Since the value of all queues Q_{e_j}, $j < R_t$, remains 0 in round t, we obtain from (17.1)

$$|S_{t+1}| + \sum_{1 \le i \le k} \text{val}_{t+1}(e_i) \le |S_t| - 1 + \sum_{1 \le i \le k} \text{val}_t(e_i).$$

Hence, $\Phi_{t+1} \le \Phi_t - 1$. □

Theorem 17.3. *With probability at least $1 - 1/N$ every packet reaches its final destination in at most $8n$ rounds.*

Proof. Consider an arbitrary packet pkt_v with route $\rho(v) = (e_1, \ldots, e_k)$, $k \le n$, from the source v to the intermediate destination $\sigma(v)$. As before, let $S = \{u \ne v \mid \rho(u) \cap \rho(v) \ne \emptyset\}$ be the set of packets which cross v's path in Phase 1, and let $X = |S|$. First we determine an upper bound on $E[X]$:

Consider some edge e, $1 \le i \le k$, which goes from $b_1 \ldots b_n$ to $b_1 \ldots b_{i-1}\overline{b_i}b_{i+1} \ldots b_n$. Then the packets that use edge e must be routed from a source $\{0,1\}^{i-1}b_i b_{i+1} \ldots b_n$ to a destination $b_1 \ldots b_{i-1}\overline{b_i}\{0,1\}^{n-i}$. Therefore, there are exactly 2^{i-1} packets (those whose last $n - i + 1$ source bits are $b_i \ldots b_n$) that could potentially use edge e. Each of these packets uses edge e with probability $1/2^i$, because its destination must agree in the first i coordinates with $b_1 \ldots b_{i-1}\overline{b_i}$. Therefore, the expected number of packets that use edge e is

$$\frac{2^{i-1}}{2^i} = 1/2.$$

It follows that the expected number of packets that cross v's path, i.e., use one of the edges $e_1, \ldots, e_k$ is

$$E[X] \le \frac{k}{2} \le \frac{n}{2}.$$

Now we prove that the random variable X is highly concentrated around its expectation, i.e., the probability that X is a constant factor larger than its expectation is very small. We would like to apply Chernoff Bounds, but in order to do that we have to show that X is the sum of independent indicator random variables.

For each node $u \ne v$ let X_u be the indicator random variable which has value 1 if and only if $\rho(u)$ contains an edge in $\{e_1, \ldots, e_k\}$. Then all random variables X_u, $u \ne v$, are independent (because each route $\rho(u)$ is uniquely determined by u and the intermediate destination $\sigma(u)$, which is chosen independently for each packet). Hence, $X = \sum_{u \ne v} X_u$,

is the sum of $N-1$ independent indicator random variables, and we can apply Chernoff Bounds: Using Inequality (8.4) of the Chernoff Bounds from Theorem 8.4 for $D = 3n \geq 6E[X]$, we get

$$\text{Prob}(X \geq D) < 2^{-D},$$

and thus

$$\text{Prob}(X > 3n) < 2^{-3n}.$$

If $X \leq 3n$, then at most $3n$ packets cross pkt_v's path, so by Lemma 17.2 pkt_v reaches its intermediate destination $\sigma(v)$ after at most $3n+k \leq 4n$ rounds. Hence, with probability at least $1-2^{-3n}$ packet v reaches its destination after at most $4n$ rounds. By the union bound, all N packets reach their destination after at most $4n$ rounds with probability at least

$$1 - N \cdot 2^{-3n} = 1 - 2^{-2n}.$$

The probability that some packet fails to reach its final destination within $8n$ rounds is at most the probability that there is a packet that fails to complete Phase 1 in $4n$ rounds plus the probability that there is a packet that fails to complete Phase 2 in $4n$ rounds. Notice, however, that Phase 2 behaves like Phase 1 run backwards. Therefore the analysis of analysis of Phase 1 applies to the analysis of Phase 2. So the probability that all packets reach their final destination in $8n$ rounds is at least $1-2\cdot 2^{-2n} \geq 1-1/N$ for $N \geq 1$. □