Randomized Algorithms

CSCI-UA 480.73

Yao Xiao

Spring 2023

Contents

1/23 Lecture

Instructor. Richard Cole, cole@cs.nyu.edu.

Class time. Mondays & Wednesdays 9:30–10:45am, 317 WWH.

Office hours. Mondays 4:00–5:00pm; also by appointment, 417 WWH.

0 Introduction

Example 0.1. Some examples of randomized algorithms versus deterministic algorithms:

- Quicksort vs. Merge Sort or Heap Sort
- Hashing vs. Balanced Trees
- Bloom Filters vs. Bit Arrays
- Stochastic Gradient Descent vs. Deterministic Gradient Descent (not covered but expected in Machine Learning or Scientific Computing)

1 Repetition

1.1 Verifying Polynomial Identities

Consider the polynomial identity

$$\det \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{bmatrix} = \prod_{1 \le i \le j \le n} (x_i - x_j). \tag{1}$$

Is this true for $n = 100$?

Problem. Given polynomials F and G, check if F and G are identical. That is, check if $H = F - G$ is identically zero. Suppose H is a polynomial of degree d.

Solution. Choose $r \in [1, 100d]$ uniformly at random, then evaluate $H(r)$. Note that uniform random number generation is assumed to run constant time. If $H(r) \neq 0$, then we can answer H is not identically zero. However, if $H(r) = 0$, the answer H is identically zero may be wrong. The probability that the answer is wrong is $\le \frac{d}{100d} = \frac{1}{100}$, since a non-identically-zero polynomial of degree d can have at most d real roots.

However, what if we want a lower error probability? We can run the algorithm above multiple times. If we run the algorithm k times, and each time $H(r_i) = 0$, then the probability of giving a wrong answer is $\le \left(\frac{1}{100}\right)^k$.

Probability Background

Randomized algorithms typically have a finite number (discrete set) of outcomes (based on the random choices). Write $\Omega = \{O_1, \cdots, O_m\}$ as the set of outcomes, *e.g.*, when rolling a die, $\Omega = \{1, 2, 3, 4, 5, 6\}$. For each outcome, we have a probability $0 \leq \mathbb{P}[O_i] \leq 1$, and

$$\sum_{i=1}^{m} \mathbb{P}[O_i] = 1. \tag{2}$$

Group outcomes into events as $E = \{O_{i_1}, \cdots, O_{i_k}\} \subseteq \Omega$, *e.g.*, a die lands on an even value. We use the set theory notations for events.

Lemma 1.1. If E_1 and E_2 are disjoint, then $\mathbb{P}[E_1 \cup E_2] = \mathbb{P}[E_1] + \mathbb{P}[E_2]$.

Lemma 1.2. $\mathbb{P}[E_1 \cup E_2] = \mathbb{P}[E_1] + \mathbb{P}[E_2] - \mathbb{P}[E_1 \cap E_2]$.

Lemma 1.3 (Inclusion-Exclusion Principle)**.** Given events $E_1, \cdots, E_n$, we have that

$$\mathbb{P}\left[\bigcup_{i=1}^{n} E_i\right] = \sum_{i=1}^{n} \mathbb{P}[E_i] - \sum_{i<j} \mathbb{P}[E_i \cap E_j] + \sum_{i<j<k} \mathbb{P}[E_i \cap E_j \cap E_k] - \cdots. \tag{3}$$

Events E and F are independent if and only if $\mathbb{P}[E \cap F] = \mathbb{P}[E] \cdot \mathbb{P}[F]$. Events $E_1, \cdots, E_k$ are mutually independent if and only if for every subset $I \subseteq \{1, \cdots, k\}$,

$$\mathbb{P}\left[\bigcap_{i \in I} E_i\right] = \prod_{i \in I} \mathbb{P}[E_i]. \tag{4}$$

1/25 Lecture

1.2 Verifying Matrix Multiplication

Problem. Test if $A \times B = C$, where A, B, and C are $n \times n$ matrices.

Solution. Let $r = (r_1, \cdots, r_n)$ an n-bit vector be chosen uniformly at random. Note that choosing $r \in \{0, 1\}^n$ uniformly at random is equivalent to choosing $r_1, \cdots, r_n$ independently and uniformly at random (the **principle of deferred decisions**). Compute $s = Br$, $t = As$, and $w = Cr$. If $t = u$, we would answer true and we would answer false otherwise.

Now take $D = AB - C$ and suppose $D \neq 0$. Without loss of generality we assume that $d_{11} \neq 0$. Suppose $Dr = 0$, then we have that

$$d_{11} r_1 + \sum_{j=2}^{n} d_{1j} r_j = 0 \implies r_1 = -\frac{\sum_{j=2}^{n} d_{1j} r_j}{d_{11}}. \tag{5}$$

By the idea of the principle of deferred decisions, we can consider the right-hand side of (5) as already determined just before r_1 is chosen, leaving r_1 as random (or deferred). Since r_1 has two choices 0 or 1, the probability that $ABr = Cr$ is at most $1/2$.

Probability Background

The conditional probability that event E occurs given that event F occurs (*e.g.*, probability that a die roll of 2 occurs given that the die roll is an even number) is

$$\mathbb{P}[E|F] = \frac{\mathbb{P}[E \cap F]}{\mathbb{P}[F]}. \tag{6}$$

Lemma 1.4. If E and F are independent events, then $\mathbb{P}[E \cap F] = \mathbb{P}[E]$.

Theorem 1.5 (Law of Total Probability)**.** Let $E_1, \cdots, E_n$ be mutually disjoint events in the sample space Ω, and suppose that $\bigcup_{i=1}^{n} E_i = \Omega$. Then

$$\mathbb{P}[B] = \sum_{i=1}^{n} \mathbb{P}[B \cap E_i] = \sum_{i=1}^{n} \mathbb{P}[B|E_i]\mathbb{P}[E_i]. \tag{7}$$

Now back to the problem, if $AB \neq C$, we can compute that

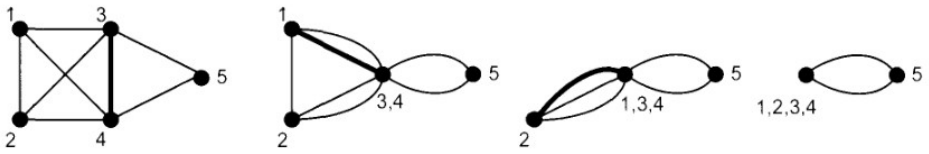
$$\begin{aligned}
\mathbb{P}[ABr = Cr] &= \sum_{(x_2,\cdots,x_n)\in\{0,1\}^{n-1}} \mathbb{P}[(ABr = Cr) \wedge ((r_1,\cdots,r_n) = (x_2,\cdots,x_n))] \\
&\leq \sum_{(x_2,\cdots,x_n)\in\{0,1\}^{n-1}} \mathbb{P}\left[\left(r_1 = -\frac{\sum_{j=2}^{n} d_{1j}r_j}{d_{11}}\right) \wedge ((r_1,\cdots,r_n) = (x_2,\cdots,x_n))\right] \\
&= \sum_{(x_2,\cdots,x_n)\in\{0,1\}^{n-1}} \mathbb{P}\left[r_1 = -\frac{\sum_{j=2}^{n} d_{1j}r_j}{d_{11}}\right] \cdot \mathbb{P}[(r_1,\cdots,r_n) = (x_2,\cdots,x_n)] \\
&\leq \sum_{(x_2,\cdots,x_n)\in\{0,1\}^{n-1}} \frac{1}{2} \cdot \mathbb{P}[(r_1,\cdots,r_n) = (x_2,\cdots,x_n)] = \frac{1}{2}.
\end{aligned} \tag{8}$$

Therefore, if $AB = C$ then the algorithm is always correct, and if $AB \neq C$ then the algorithm has $1/2$ probability of going wrong. By k repetitions, assuming independence, the probability of getting an error would be reduced to 2^{-k}.

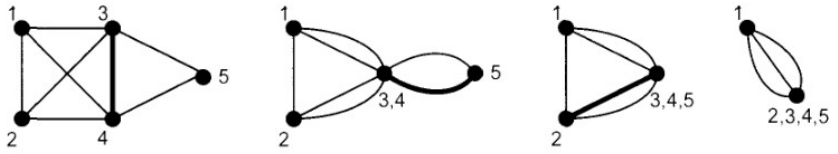
1.3 Randomized Min-Cut

Problem. A **cut-set** in a graph is a set of edges whose removal breaks the graph into two or more connected components. Given a graph G with n vertices, the min-cut problem is to find a minimum cardinality cut-set in G.

Solution. The main operation in the algorithm is **edge contraction**, as is shown in Figure 1. In contracting an edge (u, v), we merge the two vertices u and v into one vertex, eliminate all edges connecting u and v, and retain all other edges in the graph.



(a) A successful run of min-cut.



(b) An unsuccessful run of min-cut.

Figure 1: An example of two executions of min-cut in a graph with minimum cut-set of size 2.

The randomized algorithm consists of $n - 2$ iterations. In each iteration, it chooses an edge uniformly at random from the remaining edges and perform edge contraction. After $n - 2$ iterations, the graph will consist of only two vertices, and the algorithm outputs the set of edges connecting them.

1/30 Lecture

Theorem 1.6. The algorithm outputs a min-cut set with probability at least $\frac{2}{n(n-1)}$.

Proof. Let G be the graph with n vertices and m edges, and let C be a min-cut with $k = |C|$. Moreover, let E_i represent the event that the ith edge contracted is not in C, so $F_i = \bigcap_{h=1}^{i}$ will represent the event that none of the first i edges contracted are in C. Since the min-cut set C has k edges, each vertex must have degree at least k (otherwise we can separate a vertex of degree less than k from the rest of the vertices with fewer than k cuts). Therefore, the total number of edges in G is at least $nk/2$. Thus we have that

$$\mathbb{P}[F_1] = \mathbb{P}[E_1] \geq 1 - \frac{k}{nk/2} = 1 - \frac{2}{n}. \tag{9}$$

Similarly, suppose that F_1 happened, then

$$\mathbb{P}[E_2|F_1] \geq 1 - \frac{2}{n-1}. \tag{10}$$

Iterating this argument, we have that

$$\mathbb{P}[E_i|F_{i-1}] \geq 1 - \frac{2}{n-i+1}. \tag{11}$$

Since the randomized algorithm consists of $n-2$ iterations, we want to compute that

$$\begin{aligned}\mathbb{P}[F_{n-2}] &= \mathbb{P}[E_{n-2} \cap F_{n-3}] = \mathbb{P}[E_{n-2}|F_{n-3}] \cdot \mathbb{P}[F_{n-3}] = \cdots = \mathbb{P}[E_{n-2}|F_{n-3}] \cdots \mathbb{P}[E_2|F_1] \cdot \mathbb{P}[F_1] \\ &\geq \prod_{i=1}^{n-2} \left(1 - \frac{2}{n-i+1}\right) = \prod_{i=1}^{n-2} \frac{n-i-1}{n-i+1} = \frac{(n-2)(n-3)(n-4)\cdots 3 \cdot 2 \cdot 1}{n(n-1)(n-2)\cdots 5 \cdot 4 \cdot 3} = \frac{2}{n(n-1)}.\end{aligned} \tag{12}$$

Therefore, any min-cut of G can be found with probability at least $\frac{2}{n(n-1)}$. However, since we cannot know how many min-cut sets there are in G, so this bound is the best we can guarantee. □

Using l repetitions, the probability of never finding a min-cut is at most

$$\left(1 - \frac{2}{n(n-1)}\right)^l \leq \underbrace{\left(\exp\left(-\frac{2}{n(n-1)}\right)\right)^l}_{1-x \leq \exp(-x)} = \exp\left(-\frac{2l}{n(n-1)}\right). \tag{13}$$

By choosing $l = \frac{cn(n-1)\ln n}{2}$, this probability is at most $\exp(-c \ln n) = n^{-c}$.

Monty Hall Problem

Problem. Suppose you are on a game show, and you are given the choice of three doors: Behind one door is a car; behind the others, goats. You randomly pick a door, and the host will reveal another door with a goat inside. Then you are prompted to make a switch to the other door. Will this give you an advantage?

Solution. If one does not make the switch, he has only 1/3 probability of getting the car, since his first choice is independent of any proceeding activity. On the other hand, if one makes the switch, he will have 2/3 probability of getting the car. Consider a much more extreme case: Suppose there are a million doors with only one car, and the host reveals 999998 doors with goats after the first choice. The chance of picking a car in the first choice is only 1/1000000, and this probability will not raise because of revealing the other doors. Moreover, by opening the doors with goats, the host is almost telling the answer, so making a switch is indeed a wise choice.

2/1 Lecture

2 Discrete Random Variables

Probability Background

For example, X is the sum of the outcomes of the rolls of two dice, then for instance $X = 4$ corresponds to the outcomes $(1, 3)$, $(2, 2)$, and $(3, 1)$. This is an example of a discrete random variables, which takes a finite or countable number of values.

Definition 2.1. Random variables X and Y are **independent** if and only if

$$\mathbb{P}[(X = x) \wedge (Y = y)] = \mathbb{P}[X = x] \cdot \mathbb{P}[Y = y], \qquad \forall x, y. \tag{14}$$

Definition 2.2. Random variables $X_1, \cdots, X_k$ are **mutually independent** if and only if for any subset $I \subseteq \{1, \cdots, k\}$ and values x_i, $i \in I$,

$$\mathbb{P}\left[\bigwedge_{i \in I} (X_i = x_i)\right] = \prod_{i \in I} \mathbb{P}[X_i = x_i]. \tag{15}$$

Definition 2.3. The **expectation** (average value) of a random variable X is

$$\mathbb{E}[X] = \sum_i i \cdot \mathbb{P}[X = i]. \tag{16}$$

For instance, let X be the outcome of one roll of a die, then we have that

$$\mathbb{E}[X] = \frac{1}{6}(1 + 2 + 3 + 4 + 5 + 6) = \frac{7}{2}. \tag{17}$$

As another example, let Y be the sum of the rolls of two dice, then we have that

$$\mathbb{E}[Y] = \frac{1}{36} \cdot 2 + \frac{2}{36} \cdot 3 + \frac{3}{36} \cdot 4 + \cdots + \frac{1}{36} \cdot 12 = 7. \tag{18}$$

Theorem 2.4 (Linearity of expectations)**.** Let $X_1, \cdots, X_n$ be random variables, then

$$\mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i], \tag{19}$$

whether or not the random variables are (mutually) independent.

2.1 The Expected Runtime of Randomized QuickSort

Recall that the QuickSort algorithm can be represented as in Algorithm 1. For the sake of simplicity, we call the

Algorithm 1 QuickSort

Require: an array S of n distinct elements over a totally sorted universe.

1: choose an element x of S as the pivot;
2: compare every other element of S to x in order to divide the other elements into two subarrays, such that S_1 has all the elements of S that are less than x, and S_2 has all the elements of S that are greater than x;
3: run QuickSort (Algorithm 1) on S_1 and S_2, respectively;
4: **return** the concatenated array of S_1, $\{x\}$, and S_2;

elements of S as $z_1, \cdots, z_n$, where z_i represents the ith smallest element of S. In order to analyze the runtime, we want to count the number of comparisons. Since QuickSort will compare elements only to the pivots, and a pivot will be excluded from any further recursive calls, each pair of elements in S must be compared at most once. Therefore, *indicator random variables* can be used to count the number of comparisons. Let X be the total number of comparisons for sorting S and let X_{ij} represent the number of comparisons between z_i and z_j, then we have that

$$X = \sum_{i<j} X_{ij}. \tag{20}$$

Moreover, let E_{ij} be the event that z_i is compared with z_j. By the linearity of expectations, we can deduce that

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i<j} X_{ij}\right] = \sum_{i<j} \mathbb{E}[X_{ij}] = \sum_{i<j} (0 \cdot \mathbb{P}[\overline{E_{ij}}] + 1 \cdot \mathbb{P}[E_{ij}]) = \sum_{i<j} \mathbb{P}[E_{ij}]. \tag{21}$$

Therefore, it suffices to compute $\mathbb{P}[E_{ij}]$, *i.e.*, the probability that z_i and z_j are compared during the sorting. Consider the elements $z_i, z_{i+1}, \cdots, z_{j-1}, z_j$. If any of the elements $z_{i+1}, \cdots, z_{j-1}$ has been chosen as a pivot, then z_i and z_j must have been partitioned into different subarrays but QuickSort and will never be compared. Therefore, the only chance that z_i is compared with z_j is that either z_i or z_j is the first element to be chosen as a pivot among the elements $z_i, z_{i+1}, \cdots, z_{j-1}, z_j$ during the sorting. Therefore, we are choosing one of the two elements among a total of $j - i + 1$ elements, and thus

$$\mathbb{P}[E_{ij}] = \frac{2}{j-i+1}, \qquad \forall 1 \le i < j \le n. \tag{22}$$

Therefore, (21) can be written as

$$\mathbb{E}[X] = \sum_{i<j} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \underbrace{\sum_{l=1}^{n-i} \frac{2}{l+1}}_{l=j-i} < \sum_{i=1}^{n-1} \sum_{l=1}^{n-i} \frac{2}{l} = \sum_{i=1}^{n-1} O(\log n) = O(n \log n). \tag{23}$$

Since each comparison takes constant time, the runtime of making comparisons is $O(n \log n)$. Furthermore, this dominates the linear time required for partitioning the elements, thus the expected runtime of the randomized QuickSort is $O(n \log n)$.

2/6 Lecture

Skipped due to technical issues.

2/8 Lecture

Probability Background

Suppose that we run an experiment that succeeds with probability p and fails with probability $1 - p$. Let Y be a random variable such that $Y = 1$ if the experiment succeeds and $Y = 0$ otherwise. The variable Y is called a **Bernoulli random variable** or an **indicator random variable**. Note that

$$\mathbb{E}[Y] = p \cdot 1 + (1-p) \cdot 0 = p. \tag{24}$$

Now, if we perform the experiment above for n times, and let X be a random variable representing the number of successes, then

$$\mathbb{P}[X = j] = \binom{n}{j} p^j (1-p)^{n-j}, \tag{25}$$

which defines a **binomial random variable**. Note that $X = \sum_{j=1}^{n} Y_j$ if each Y_j is the Bernoulli random variable for the jth experiment, and it follows from the linearity of expectation that $\mathbb{E}[X] = np$.

Now suppose that we perform the experiment above until the first success, and denote by a random variable Z the number of experiments we perform. Then

$$\mathbb{P}[Z = n] = (1-p)^{n-1} p. \tag{26}$$

This is called a **geometric random variable**. Note that it satisfies the **memoryless** property, that is,

$$\mathbb{P}[Z = n + k | Z > k] = \frac{\mathbb{P}[(Z = n+k) \cap (Z > k)]}{\mathbb{P}[Z > k]} = \frac{(1-p)^{n+k-1} p}{\sum_{i=k+1}^{\infty} (1-p)^{i-1} p} = \frac{(1-p)^{n+k-1} p}{(1-p)^k} = (1-p)^n 0 = \mathbb{P}[Z = n]. \tag{27}$$

Moreover, we can compute the expectation of the geometric random variable Z as

$$\mathbb{E}[Z] = \sum_{j=1}^{\infty} j \mathbb{P}[Z = j] = \sum_{j=1}^{\infty} j (1-p)^{j-1} p = p \sum_{j=1}^{\infty} j (1-p)^{j-1} =: pI. \tag{28}$$

Note that

$$I = \sum_{j=1}^{\infty} j(1-p)^{j-1} = 1 + 2(1-p) + 3(1-p)^2 + 4(1-p)^3 + \cdots, \tag{29}$$

$$(1-p)I = \sum_{j=1}^{\infty} j(1-p)^{j} = (1-p) + 2(1-p)^2 + 3(1-p)^3 + \cdots, \tag{30}$$

so we can deduce that

$$\mathbb{E}[Z] = pI = I - (1-p)I = 1 + (1-p) + (1-p)^2 + (1-p)^3 + \cdots = \frac{1}{1-(1-p)} = \frac{1}{p}. \tag{31}$$

2.2 Coupon Collector's Problem

Problem. Suppose that each box of cereal contains one of n different coupons. Once you obtain one of every type of coupon, you can send in for a prize. Assuming that the coupon in each box is chosen independently and uniformly at random from the n possibilities, how many boxes of cereal must you buy before you can send in for the prize?

Solution. Let X be the number of boxes one must buy before obtaining all n types of coupons. Moreover, let X_i denote the number of purchases to obtain i types of coupons after having obtained $i-1$ types of coupons. Then we have that

$$X = \sum_{i=1}^{n} X_i. \tag{32}$$

Note that after having obtained $i-1$ types of coupons, if we want to further obtain one more type of coupon, each purchase will be of success probability $\frac{n-i+1}{n}$. Therefore, each X_i is a geometric random variable with success probability $\frac{n-i+1}{n}$, and thus by the linearity of expectation, we can obtain that

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} \frac{n}{n-i+1} = n \underbrace{\sum_{j=1}^{n} \frac{1}{j}}_{j=n-i+1} =: nH(n). \tag{33}$$

Note that $H(n) = \sum_{i=1}^{n} \frac{1}{i}$ is called the **harmonic number**, which we can approximate by

$$\sum_{i=1}^{n} \frac{1}{i} > \int_{1}^{n+1} \frac{1}{x} dx = \ln x \Big|_{x=1}^{n+1} = \ln(n+1), \tag{34}$$

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \sum_{i=2}^{n} \frac{1}{i} < 1 + \int_{1}^{n} \frac{1}{x} dx = 1 + \ln x \Big|_{x=1}^{n} = 1 + \ln n, \tag{35}$$

as is demonstrated in Figure 2.

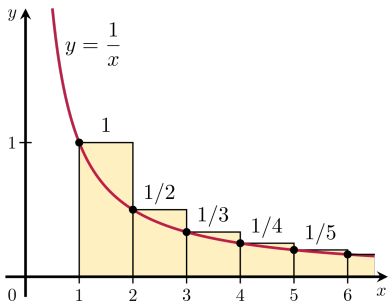


Figure 2: The lower bound for the harmonic number by integration. Note that this is obtained by taking the left boundary value for each bar. If we alternatively take the right boundary value for each bar, we will obtain an upper bound.

Back to the problem, we can now conclude that

$$n\ln(n+1) < \mathbb{E}[X] < n + n\ln n \implies \mathbb{E}[X] = \Theta(n\ln n). \tag{36}$$

2/13 Lecture

2.3 Stable Matching

Problem. Suppose there are n doctors and n hospitals. Each doctor has an ordering of preference for the hospitals, and each hospital also has an ordering of preference for the doctors. We say that (d, H) is a **blocking pair** if both d and H prefer each other to their current matches. A stable matching is a matching with no blocking pairs.

Solution. We propose the deferred acceptance algorithm as shown in Algorithm 2, which runs $O(n^2)$ time.

Algorithm 2 Deferred acceptance (doctor proposing)

```
while some doctor d is unmatched do
    d proposes to the next hospital H on her list;
    if H is unmatched or H prefers d to its current match d' then
        H is tentatively matched to d and d' is left unmatched;
    end if
end while
```

Lemma 2.5. Every doctor gets a match in Algorithm 2.

Proof. Once tentatively matched, a hospital remains matched. Therefore, if d is unmatched, at most $n-1$ hospitals have ever been matched. In the worst case, the nth hospital will be proposed to by d eventually, so that d will be matched. By arbitrariness of d we complete the proof. □

Lemma 2.6. The matching in Algorithm 2 is stable.

Proof. Each hospital's match only improves as the algorithm proceeds. In other words, each hospital is matched to the best proposal it received. Moreover, observe that each doctor gets the best match among those proposals that are not rejected. Now assume for contradiction that (d, H) forms a blocking pair, then d and H prefer each other to their current matches, say d to H' and H to d'. However, H' must be the best match for d among those that does not reject d. That is to say, if d prefers H over H', then H must have rejected d. Given that each hospital is matched to the best proposal it received, H must prefer d' over d. This leads to a contradiction, implying that their is no blocking pair produced by the algorithm. The matching is thus stable and the proof is done. □

Randomized input. Assume that the same problem statement as above, with each preference list a uniform draw over all permutations of the other side.

Theorem 2.7. In the uniform random preference model, the expected number of proposals using Algorithm 2 is $O(n\log n)$.

Proof. We use the principle of deferred decisions. In other words, we do not assume that the random preference lists are chosen in advance. Instead, we assume that each doctor does not know his preference list and each time he makes a proposal, he picks a hospital uniformly at random. This problem then immediately resembles the coupon collector's problem, observing that Algorithm 2 terminates once all the hospitals have received at least one proposal (since once a hospital receives a proposal, it will stay matched though its match may change, and once all the hospitals have received a proposal, this means all the hospitals are matched, thus all the doctors must have been matched as well). In other words, the doctors randomly proposes hospitals until all the n hospitals have received a proposal. We know that the expected number would then be $nH(n) \approx \Theta(n\ln n)$. However, this is only an upper bound. Doctors actually "memorize" which hospitals have rejected them so that they will not propose to the same hospitals again, instead of proposing completely randomly. Therefore, the total number of proposals could have been smaller, meaning that the expected number of proposals using Algorithm 2 is $O(n\ln n)$, and the proof is complete. □

2.4 String Matching (Fingerprinting)

Problem. Given a text $t = t_1 t_2 \cdots t_m$ and a pattern $p = p_1 p_2 \cdots p_n$ (assuming $m > n$), we want to find the occurrences of p in t. A basic example of string matching is when the pattern and the searched text are arrays of elements of an alphabet (finite set), which can be a human language alphabet (A through Z), a binary alphabet $\{0, 1\}$, or a DNA alphabet (A,C,G,T) in bioinformatics, etc. For a general notation, we denote the alphabet as $\Sigma = \{0, 1, \cdots, s-1\}$ so that each character is treated as a number.

Solution. The naive solution would be to create a window of length n and move through t, comparing the substring in the window to p in each iteration. This would be $O(m)$ iterations and $O(n)$ time for each iteration, thus requiring $O(mn)$ total runtime. To improve the efficiency, we propose the **Karp-Rabin algorithm**, which compares the "hash" values of the substrings and p, thus reducing each iteration to constant time. More specifically, calculating the initial hash values will take $O(n)$ time, but subsequent updating will be $O(1)$ for each iteration among the total $O(m)$ of them. This gives a better $O(n+m)$ runtime algorithm, but this can give error. To see this, we will represent p as a number, called $\texttt{val}(p)$, such that

$$\texttt{val}(p) = p_1 s^{n-1} + p_2 s^{n-2} + \cdots + p_{n-1} s + p_n \mod q, \tag{37}$$

where q is a suitable prime number that we will specify later. We also assume that each arithmetic modulus operation takes $O(1)$ time. We represent each of the substrings t^{i+1} (of length n), $0 \le i \le n - m$, using the same encoding as

$$\texttt{val}(t^{i+1}) = t_{i+1} s^{n-1} + t_{i+2} s^{n-2} + \cdots + t_{i+n-1} s + t_{i+n} \mod q. \tag{38}$$

Clearly if $t^{i+1} = p$, then $\texttt{val}(t^{i+1}) = \texttt{val}(p)$, but the reverse is not true. This can lead to a non-match being reported as a match, so the idea of the algorithm is to choose q so that the probability of this event is small. To do this, let us examine the situation when $t^{i+1} \neq p$ yet $\texttt{val}(t^{i+1}) = \texttt{val}(p)$. In this situation, we have that $\texttt{val}(p) - \texttt{val}(t^{i+1}) \equiv 0 \mod q$, implying that $\texttt{val}(p) - \texttt{val}(t^{i+1})$ is an integer multiple of q. Now note that $\texttt{val}(p), \texttt{val}(t^{i+1}) < s^n$, since they are at most

$$\sum_{i=1}^{n} (s-1) s^{n-i} = \sum_{i=1}^{n} s^{n-i+1} - \sum_{i=1}^{n} s^{n-i} = \sum_{i=0}^{n-1} s^{n-i} - \sum_{i=1}^{n} s^{n-i} = s^n - 1 < s^n. \tag{39}$$

As a result, $|\texttt{val}(p) - \texttt{val}(t^{i+1})| < s^n$. Suppose that $q \ge s$, then there are at most n distinct primes q for which $\texttt{val}(p) - \texttt{val}(t^{i+1})$ is an integer multiple of q. Otherwise, $\texttt{val}(p) - \texttt{val}(t^{i+1})$ can be expressed as an integer multiple of the product of more than n distinct primes that are $\ge s$, thus exceeding s^n and leading to a contradiction.

Given the previous observations, we will choose q uniformly at random from a large set of primes, so that there is only a small chance that q is one of the at most n primes that divide $\texttt{val}(p) - \texttt{val}(t^{i+1})$ exactly. In particular we choose q from a range $[v, 2v)$ to be specified shortly. This is based on a known fact that the number of primes in this range is at least $\frac{v}{2 \ln v}$ for $v \ge 61000$ (this is a loose bound). The algorithm to choose a prime selects an integer in this range uniformly at random and tests it for primality, and keeps going until a prime number is found. In expectation, $2 \ln v$ tests would suffice. As we will see later, each primality test can be done by a linear time randomized algorithm which has a small probability of reporting a composite as a prime (in which case the Karp-Rabin algorithm may report incorrect results). We will keep this probability to at most $\frac{1}{2n^c}$, where $c > 0$ is an arbitrary constant.

We would like to achieve a probability of at most $\frac{1}{2m^c}$ of having an incorrect answer, assuming that the q provided by the primality test is actually a prime number. We need to test $n - m + 1$ alignments of p with substrings of t, so it suffices to have a probability of an incorrect answer on any given test of $\frac{1}{2m^{c+1}}$. To this end, we need that the number of primes in the interval $[v, 2v)$ be at least $n \cdot 2m^{c+1}$, since there are at most n "bad" choices of q. Therefore, it suffices to have that

$$\frac{v}{2 \ln v} \ge 2nm^{c+1}. \tag{40}$$

Choosing $v = 8(c+2)nm^{c+1} \ln m$ would suffice (JUST TRUST THE PROFESSOR), and recall that we also want $v \ge s$ (though in practice this condition is satisfied automatically most of the times).

2/15 Lecture

Miller-Rabin Primality Test

We propose the Miller-Rabin primality test with probability of error at most 2^{-k} for some constant k, that is used in the previous algorithm. Suppose that m is the integer being tested (with $n = \log m$ bits).

Lemma 2.8. Consider the values $f(x) = x^{(n-1)/2} \mod n$ for numbers x coprime to n.

(1) If m is a prime number, then there are exactly $(n-1)/2$ numbers x for which $f(x) = -1$, and $(n-1)/2$ numbers x for which $f(x) = 1$.

(2) If m is an odd-valued composite number which has two distinct prime factors, then either $f(x) = 1$ for all of these x or $f(x) \neq \pm 1$ for at least half of these x.

Proof. This lemma can be shown by elementary arguments, but the proof is long which will be neglected here. □

In order to apply this lemma for primality testing, we need three algorithms: one for exponentiation in order to compute $f(x)$ as in the lemma, one for finding the greatest common divisor, and one to test if $m = x^i$ for some integers $x \geq 2$ and $i \geq 2$.

The exponentiation algorithm. To compute $x^y \mod n$ we proceed as follows.

- By repeated squaring, compute the following values modulus m: $x, x^2, x^4, \cdots, x^k$, where $2^k \leq y < 2^{k+1}$. As all results are computed modulus n, if we use long multiplication and division, then each squaring takes $O(n^2)$ time, and so $O(n^3)$ time is needed overall. WHY? WHAT IS THE RELATION BETWEEN k AND n?
- Suppose that $y = 2^{i_1} + 2^{i_2} + \cdots + 2^{i_j}$, where $0 \leq i_1 < i_2 < \cdots < i_j \leq k$, then we multiply the terms $x^{2^{i_1}}, x^{2^{i_2}}, \cdots, x^{2^{i_j}}$ together to obtain the result. This takes a further $O(n^3)$ time. WHY?

The greatest common divisor algorithm. We propose an algorithm to compute $\gcd(a, b)$ for $a > b$, as is described in Algorithm 3.

Algorithm 3 GCD

```
1: c ← a mod b;
2: if c ≠ 0 then
3:     return GCD(b, c);
4: else
5:     return b;
6: end if
```

Lemma 2.9. Algorithm 3 runs in $O(n^3)$ time on n-bit input integers.

Proof. We observe that $c < a/2$. Indeed, if $b > a/2$, then $c = a \mod b = a - b < a/2$. Otherwise if $b \leq a/2$, then $c < b \leq a/2$. If $c \geq 1$, then by a similar reasoning $\gcd(b, c) < b/2$. Therefore, we can see that in at most two iterations, the arguments passed to the algorithm halve in value, and thus the algorithm proceeds for at most $2n$ iterations. Each iteration entails a modulus operations which takes $O(n^2)$ time for long integers, so the total runtime would be $O(n^3)$, as desired. □

The power testing algorithm. This algorithm is intended to test if $m = x^i$ for some integers $x \geq 2$ and $i \geq 2$. It seeks, for each i in turn, to identify the value x such that $(x-1)^i < m \leq x^i$, by means of binary search over the possible values of x. Since m is an n-bit number, the values of i that need to be tested are $2 \leq i < n$. A test proceeds as follows. It will iteratively improve an estimate y of x such that $y = w \cdot 2^h$ for some integers w and h, and we will maintain the property that $y < x \leq y + 2^h - 1$. In other words, y consists of the leading bits of x. Each iteration of our algorithm will add one more bit to the estimate, so the algorithm for value i will use n/i iterations. The algorithm is described as in Algorithm 4.

Algorithm 4 IsPower

$y \leftarrow 1$, $h \leftarrow \lfloor n/i \rfloor$;
while $h > 0$ **do**
  $z \leftarrow (2y+1)2^{h-1}$, $w \leftarrow z^i$;
  if $w = m$ **then**
    return true;
  else if $w < m$ **then**
    $y \leftarrow z$, $h \leftarrow h - 1$;
  else
    $y \leftarrow z - 1$, $h \leftarrow h - 1$;
  end if
end while

Lemma 2.10. Algorithm 4 runs in time $O(n^3/i)$, and testing whether m is of the form x^i for some pair of integers $x, i \geq 2$, takes time $O(n^3 \log n)$.

Proof. If we use repeated squaring to compute w, assuming that we multiple the numbers in order of increasing (doubling) length, this will take $O(n^2)$ time. We have to repeat this n/i times, giving an overall runtime of $O(n^3/i)$. Since we need to repeat this algorithm for increasing values of $i \leq n$, the overall runtime would sum to $O(n^3 H(n)) = O(n^3 \log n)$, so the proof is complete. □

Now we can propose the **Miller-Rabin** primality test algorithm, as is described in Algorithm 5.

Algorithm 5 Miller-Rabin primality test

if m is even or $m = x^i$ for some integers $x, i \geq 2$ **then**
  return `composite`;
end if
select k integers $b_1, \cdots, b_k$ in the range $[1, n-1]$ uniformly at random;
if $\gcd(n, b_i) > 1$ for some $1 \leq i \leq k$ **then**
  return `composite`;
end if
compute $r_i = b_i^{(n-1)/2} \mod n$ for $1 \leq i \leq k$;
if $r_i \neq \pm 1$ for some $1 \leq i \leq k$ **then**
  return `composite`;
else if $r_i = 1$ for all $1 \leq i \leq k$ **then**
  return `composite`;
else
  return `prime`;
end if

Lemma 2.11. The Miller-Rabin algorithm has an error probability of at most 2^{-k}.

Proof. The first two tests identify composites that are even integers or integer powers of some smaller integer, so the numbers that remain to be tested are either prime or composites with at least two distinct prime factors. For such numbers, if they are prime, then $r_i = \pm 1$ for all $1 \leq i \leq k$, and each value have equal probability of occurring. Thus in this case, the probability that $r_i = 1$ for all $1 \leq i \leq k$ is 2^{-k}, which means that the probability of reporting a prime as a composite is 2^{-k}. Likewise, for such numbers, if they are composite, then either every r_i with b_i coprime to n equals 1, or at least half of these $r_i \neq \pm 1$. In the first case, the number is identified as composite for sure; in the second case, it is misidentified as a prime with probability at most 2^{-k}. □

Lemma 2.12. The Miller-Rabin algorithm runs $O(kn^3 + n^3 \log n)$ time.

Proof. The first testing condition runs the power testing algorithm (Algorithm 4), which takes $O(n^3 \log n)$ time. Then, selecting k integers uniformly at random takes $O(k)$ time. The second testing condition runs the greatest common divisor algorithm (Algorithm 3) for k times, which is overall $O(kn^3)$ runtime. Then, computing r_i requires running the exponentiation algorithm k times, which again runs overall $O(kn^3)$ time. The final judging can be done in the meantime when computing r_i. Therefore, the Miller-Rabin algorithm takes $O(kn^3 + n^3 \log n)$ time in total. □

2/22 Lecture

3 Randomized Binary Search Trees

We are going to consider some randomized alternatives to balanced binary search tree structures such as AVL trees, red-black trees, B-trees, or splay trees, which are arguably simpler than any of these deterministic structures.

3.1 Treaps

The term "treap" is a combination of "tree" and "heap". Recall that binary search trees require the values in the left subtree to be less than the value of the current node, and the values in the right subtree to be greater than the value of the current node. Also recall that (min-)heaps require the parents to have lower priorities than the children. An example is shown as in Figure 3.

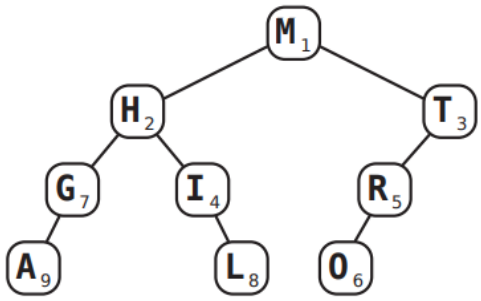


Figure 3: An example of a treap. Letters represent the search keys and numbers represent the priorities.

Insertion. We start by using the standard binary search tree insertion algorithm to insert a new node at the bottom of the tree. To this end, the search keys still form a search tree, but the priorities may no longer form a heap. For instance, if we want to insert $\mathbf{S}_{-1}$ into the treap in Figure 3. It will be inserted as the right child of $\mathbf{R}_5$. However, $5 > 0$ so the heap property is violated. In this case, recall the **rotation** operation in binary search trees, as is shown in Figure 4. If the priority of the yellow node is greater than the green node, then we perform the left rotation (from left to right in Figure 4) if necessary. Otherwise, we perform the right rotation (from right to left in Figure 4) if necessary.

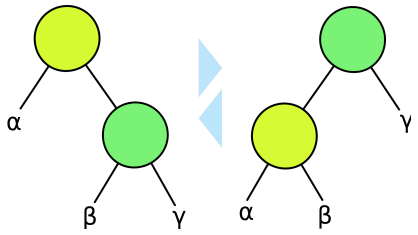


Figure 4: Rotations of binary search trees.

Note that the priority violation may not only happen between the new node and its parent. Even if we perform a left rotation between $\mathbf{R}_5$ and $\mathbf{S}_{-1}$, there will still be a violation between $\mathbf{T}_3$ and $\mathbf{S}_{-1}$. Therefore, we need to deal with all the possible priority violations between the new node and its ancestors (any other violation is not possible). An illustration is shown as in Figure 5.

Deletion. To delete a node, we just run the insertion algorithm backward in time. Suppose we want to delete node z. As long as z is not a leaf, perform a rotation at the child of z with smaller priority. This moves z down a level and its smaller-priority child up a level. The choice of which child to rotate preserves the heap property everywhere except at z. When z becomes a leaf, chop it off.

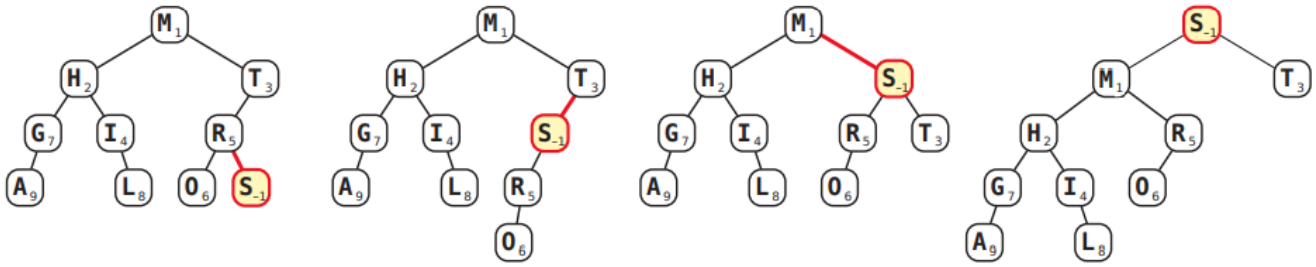


Figure 5: An example of inserting a new node in a treap.

Bound expected depth. We will use the following notations.

Notation	Interpretation
x_k	the node/item with the kth smallest key
$i \uparrow k$	i being a proper ancestor of k
[Property]	1 if Property is true and 0 otherwise
$\text{Depth}(x_k)$	$\sum_{i=1}^{n}[i \uparrow k]$
$X(i,k)$	either $\{x_i, x_{i+1}, \cdots, x_k\}$ or $\{x_k, x_{k+1}, \cdots, x_i\}$

Note that

$$\mathbb{E}[\text{Depth}(x_k)] = \sum_{i=1}^{n} \mathbb{E}[[i \uparrow k]] = \sum_{i=1}^{n} \mathbb{P}[i \uparrow k], \tag{41}$$

so it suffices to consider the probability that some node is a proper ancestor of some other node.

Lemma 3.1. For all $i \neq k$, we have $i \uparrow k$ if and only if x_i has the smallest priority among all nodes in $X(i,k)$.

Proof. There are four cases to consider.

- If x_i is the root, then $i \uparrow k$ for all k, so by definition it must have the smallest priority in $X(i,k)$.
- If x_k is the root, then $k \uparrow i$ for all i. Therefore, x_i does not have the smallest priority in $X(i,k)$ since x_k does.
- Suppose some other node x_j is the root, and x_i and x_k are in different subtrees, then either $i < j < k$ or $k < j < i$. In any of these two cases, $x_j \in X(i,k)$, so both $i \not\uparrow k$ and $k \not\uparrow i$. Moreover, x_i does not have the smallest priority in $X(i,k)$ since x_j does.
- Suppose some other node x_j is the root, and x_i and x_k are in the same subtree. Note that the subtree is just a smaller treap, so the lemma would follow by induction.

Combining the cases above, the proof is complete. □

Since each node in $X(i,k)$ is equally likely to have the smallest priority, it immediately follows that

$$\mathbb{P}[i \uparrow k] = \frac{[i \uparrow k]}{|k-i|+1} = \begin{cases} \dfrac{1}{k-i+1}, & \text{if } i < k, \\ 0, & \text{if } i = k, \\ \dfrac{1}{i-k+1}, & \text{if } i > k. \end{cases} \tag{42}$$

Plugging into the expression of the expected depth, we can thus deduce that

$$\begin{aligned} \mathbb{E}[\text{Depth}(x_k)] &= \sum_{i=1}^{n} \mathbb{P}[i \uparrow k] = \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1} = \sum_{j=2}^{k} \frac{1}{j} + \sum_{j=2}^{n-k+1} \frac{1}{j} \\ &= H(k) + H(n-k+1) - 2 < \ln k + \ln(n-k+1) - 2 < 2\ln n - 2 = O(\ln n). \end{aligned} \tag{43}$$

Now let us review the cost of each operation in the treap.

- **Search:** A successful search for key k takes $O(\text{Depth}(x_k))$ time. Let x_k^- and x_k^+ be the inorder predecessor and successor of x_k, respectively, then the time for an unsuccessful search is either $O(\text{Depth}(x_k^-))$ or $O(\text{Depth}(x_k^+))$.
- **Insertion/Deletion:** Inserting a new node with key k takes either $O(\text{Depth}(x_k^-))$ or $O(\text{Depth}(x_k^+))$ time. Deletion is just insertion in reverse.
- **Split/Join:** Splitting a treap at pivot value k takes either $O(\text{Depth}(x_k^-))$ or $O(\text{Depth}(x_k^+))$ time, since it costs the same as inserting a new dummy root with search key k and priority $-\infty$. Join is just split in reverse.

Therefore, given that $\mathbb{E}[\text{Depth}(x_k)] = O(\ln n)$, all the treap operations mentioned above in an n-node treap will take $O(\ln n)$ expected time.

Randomized Quicksort Revisited

Here we propose a new way to describe the randomized Quicksort algorithm as in Algorithm 6.

Algorithm 6 Randomized QuickSort (treap version)

Require: an array S of n distinct elements over a totally sorted universe.
1: $T \leftarrow$ an empty binary search tree;
2: insert the keys in S into T in random order;
3: **return** the inorder sequence of keys in T;

Since each key is inserted in $O(\ln n)$ expected time, the expected runtime of this randomized Quicksort algorithm will run in $O(n \ln n)$ expected time. This is a kind of Quicksort since roots are like pivots, and the two subtrees are like to two partitioned parts.

2/27 Lecture

3.2 Skip Lists

A skip list is just a sorted linked list with some random shortcuts. For each item in the linked list, we duplicate it with a 1/2 probability, string together all the duplicates into a second sorted linked list, and point each duplicate back to the original. We then repeat this step recursively, until there is no node left. An example of a skip list is shown as in Figure 6. When we insert an item into a skip list, we flip a coin and make a copy at each head, until we finally flip a tail. If the corresponding level does not exist, we create a new level. Deletion removes the key in every level that it exists. Searching is also simple: we start at the leftmost node in the highest level, scan through each level as far as we can without passing the target value x, and then proceed down to the next level. The search ends when we either reach a node with search key x or fail to find x on the lowest level.

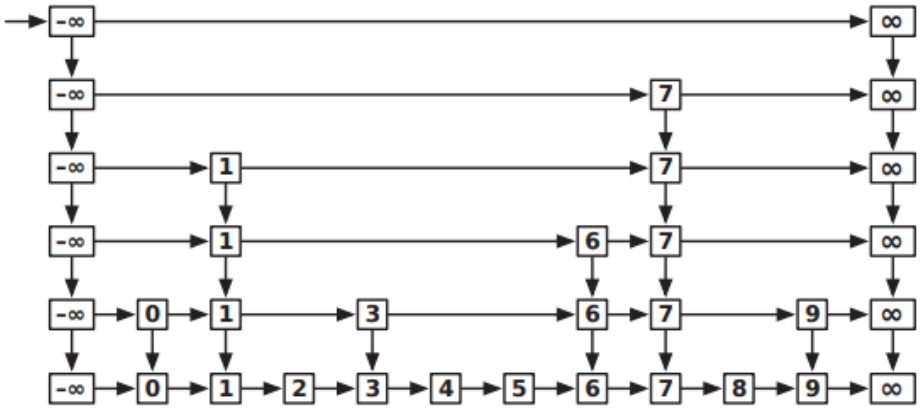


Figure 6: An example of the skip list data structure.

Number of levels. Assume that the keys in the skip list are the integers from 1 to n. Let $L(x)$ denote the number of levels of the skip list that contains the certain search key x, not counting the bottom level. Also assume that we duplicate x if a coin flip shows the head. Then we can compute the expectation of $L(x)$ recursively, such that

$$\mathbb{E}[L(x)] = \underbrace{\frac{1}{2}\cdot 0}_{\text{flip tail}} + \underbrace{\frac{1}{2}\cdot(1+\mathbb{E}[L(x)])}_{\text{flip head}} \implies \mathbb{E}[L(x)] = 1. \tag{44}$$

Moreover, we need a bound on the number of levels $L = \max_x L(x)$. Note that in order for a search key to appear on level l, it must have flipped l heads in a row when it was inserted, so that

$$\mathbb{P}[L(x) \geq l] = \frac{1}{2^l}, \qquad \forall x. \tag{45}$$

The skip list has at least l levels if and only if $L(x) \geq l$ for at least one of the n search keys. Therefore,

$$\mathbb{P}[L \geq l] = \mathbb{P}\left[\bigwedge_x (L(x) \geq l)\right] \leq \sum_x \mathbb{P}[L(x) \geq l] = \frac{n}{2^l}. \tag{46}$$

When $l \leq \log n$, this bound is trivial. However, for any constant $c > 1$, we have a strong upper bound

$$\mathbb{P}[L \geq c\log n] \leq \frac{n}{2^{c\log n}} = \frac{n}{n^c} = \frac{1}{n^{c-1}}, \tag{47}$$

which implies that with high probability, a skip list has $O(\log n)$ levels. Moreover, we can derive an upper bound for the expectation of L, such that

$$\mathbb{E}[L] = \sum_{l\geq 0} l\cdot\mathbb{P}[L=l] = \sum_{l\geq 1}\mathbb{P}[L\geq l] \leq \sum_{l=1}^{\lfloor\log n\rfloor} 1 + \sum_{l=\lfloor\log n\rfloor+1}^{\infty}\frac{n}{2^l} = \lfloor\log n\rfloor + \frac{\frac{n}{2^{\lfloor\log n\rfloor+1}}}{1-\frac{1}{2}} \leq \lfloor\log n\rfloor + 2. \tag{48}$$

Therefore, in expectation, a skip list has at most two more levels than an ideal version where each level contains exactly half the nodes of the next level below.

Logarithmic search time. We will perform a backward analysis. Suppose that we are searching for an item x in a skip list, and imaging that we are following the path backward from x to the start point of the search, *i.e.*, the $-\infty$ node in the topmost level of the skip list. By "imagine", we mean that a skip list does not have left or up pointers that really allow us to go backward: we are just doing this for analysis purpose. Now each edge in this path has one of the two possible types. Either it is an edge between two copies in adjacent levels (up), or it is an edge between a node and its predecessor in the same level (left). The second type of edge occurs only if the item does not have a copy at its next level up (*i.e.*, when there are edges both up and to the left, we will go left).

Now, we use deferred decision making to bound the expected length of the search path. Imagine we are building the skip list as the search proceeds in reverse as follows. When at an item w in level l, we toss a coin to determine whether there is a copy of w in level $l+1$. In addition, once the search path has moved up to level l, we perform coin tosses for all the remaining items on level l to determine whether they have copies at level $l+1$. Since heads (copy) and tails (no copy) are equally likely, so long as the $-\infty$ item has not been reached, the expected number of coin tosses to go up a level is 2 (since this is a geometric random variable with parameter 1/2). Once the $-\infty$ item is reached, no more coin tosses are needed: the rest of the reverse search path simply goes up level by level.

Let L be the topmost level with a copy of a real item in it (*i.e.*, the top level with only the $\pm\infty$ items is level $L+1$). Therefore, the expected number of coin tosses to reach level $\min\{1+\log n, L\}$ is at most $2(1+\log n)$. Now note that there are i items at level $1+\log n$ with probability at most

$$\binom{n}{i}\frac{1}{2^{i(1+\log n)}} \leq \frac{n^i}{i!}\cdot\frac{1}{2^i n^i} \leq \frac{1}{2^i\cdot i!}. \tag{49}$$

We will show inductively the following bound $C(n)$ on the expected number of coin tosses for a search in a skip list of n items, such that

$$C(n) = \begin{cases} 2, & \text{if } n = 1, \\ 2\log 2 + 4 & \text{if } n = 2, \\ 2\lceil\log n\rceil + 5, & \text{if } n \geq 3. \end{cases} \tag{50}$$

To verify this bound, for $n = 1$, the number of coin tosses is the number of levels that need to be climbed, which is 1 in expectation, and equals the number of heads. The number of tails is exactly 1. This gives a total of 2 coin tosses in expectation. As for $n \geq 2$, we start by looking at the situation after $1 + \lceil \log n \rceil$ heads are performed. At this point, there are i items present at level $1 + \lceil \log n \rceil$ with probability at most $\frac{1}{2^i \cdot i!}$. We can view the cost of the search starting at this level as being the same as the cost of a search on a skip list of i items. For $n = 2$, we first go up $1 + \log 2$ levels, with each level taking 2 tosses in expectation. Then there is $1/2$ probability that only 1 item is present on this level, and another $1/8$ probability that both 2 items are present on this level. This yields the bound

$$C(2) \leq 2(1 + \log 2) + \frac{C(1)}{2} + \frac{C(2)}{8} = 3 + 2\log 2 + \frac{C(2)}{8} \implies C(2) \leq \frac{8}{7}(3 + 2\log 2) \leq 2\log 2 + 4. \tag{51}$$

Similarly, for $n \geq 3$, we can deduce that

$$C(n) \leq 2(1 + \lceil \log n \rceil) + \frac{C(1)}{2} + \frac{C(2)}{8} + \sum_{i=3}^{n} \frac{C(i)}{2^i \cdot i!} \leq 2(1 + \lceil \log n \rceil) + 1 + \frac{3}{4} + \underbrace{\sum_{i=3}^{n} \frac{2\lceil \log n \rceil + 5}{2^i \cdot i!}}_{\text{induction hypothesis}}$$

$$\leq 2\lceil \log n \rceil + \frac{15}{4} + \underbrace{2 \cdot \frac{2\log 4 + 5}{2^3 \cdot 3!}}_{i=3,4} + \underbrace{4 \cdot \frac{2\log 8 + 5}{2^5 \cdot 5!}}_{i=5,6,7,8} + \cdots \leq 2\lceil \log n \rceil + \frac{15}{4} + \sum_{h=1}^{\infty} \left(2^h \cdot \frac{2(h+1) + 5}{2^{2^h+1}(2^h + 1)!} \right). \tag{52}$$

This is upper bounded by $2\lceil \log n \rceil + 5$. ANY RIGOROUS PROOF?

4 Moments and Deviations

4.1 Markov's and Chebyshev's Inequalities

Theorem 4.1 (Markov's inequality)**.** Let X be a random variable that assumes only nonnegative values. Then for all $a > 0$, we have that

$$\mathbb{P}[X \geq a] \leq \frac{\mathbb{E}[X]}{a}. \tag{53}$$

Proof. This is trivial by definition, since

$$\mathbb{E}[X] = \sum_x x\mathbb{P}[X = x] \geq \sum_{x \geq a} x\mathbb{P}[X = x] \geq a \sum_{x \geq a} \mathbb{P}[X = x] = a\mathbb{P}[X \geq a], \tag{54}$$

which by dividing both sides by a, we can conclude the desired result. □

The kth **moment** of a random variable X is defined as $\mathbb{E}[X^k]$. The **variance** of X is then defined as

$$\mathrm{Var}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - \mathbb{E}^2[X]. \tag{55}$$

The **covariance** of two random variables X and Y is defined by

$$\mathrm{Cov}[X, Y] = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])]. \tag{56}$$

Theorem 4.2. For any two random variables X and Y, we have that

$$\mathrm{Var}[X + Y] = \mathrm{Var}[X] + \mathrm{Var}[Y] + 2\mathrm{Cov}[X, Y]. \tag{57}$$

Proof. The proof is trivial by definition of variance and linearity of expectation, and will be ignored here. □

Theorem 4.3. If X and Y are two independent random variables, then

$$\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]. \tag{58}$$

Proof. By definition of expectation, we have that

$$\mathbb{E}[XY] = \sum_x \sum_y xy\mathbb{P}[(X=x)\wedge(Y=y)] = \sum_x \sum_y xy \underbrace{\mathbb{P}[X=x]\mathbb{P}[Y=y]}_{\text{by independence}}$$

$$= \sum_x x\mathbb{P}[X=x]\left(\sum_y y\mathbb{P}[Y=y]\right) = \mathbb{E}[Y]\sum_x x\mathbb{P}[X=x] = \mathbb{E}[X]\mathbb{E}[Y], \tag{59}$$

as desired, so the proof is complete. □

Note that if X and Y are independent random variables, we also have that

$$\mathrm{Cov}[X,Y] = \mathbb{E}[(X-\mathbb{E}[X])(Y-\mathbb{E}[Y])] = \mathbb{E}[X-\mathbb{E}[X]]\mathbb{E}[Y-\mathbb{E}[Y]] = (\mathbb{E}[X]-\mathbb{E}[X])(\mathbb{E}[Y]-\mathbb{E}[Y]) = 0, \tag{60}$$

and

$$\mathrm{Var}[X+Y] = \mathrm{Var}[X] + \mathrm{Var}[Y] + 2\mathrm{Cov}[X,Y] = \mathrm{Var}[X] + \mathrm{Var}[Y]. \tag{61}$$

The second observation can be generated, so that if $X_1, \cdots, X_n$ are mutually independent random variables, then

$$\mathrm{Var}\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n \mathrm{Var}[X_i]. \tag{62}$$

For instance, the variance of a Bernoulli random variable X with parameter p is

$$\mathrm{Var}[X] = \mathbb{E}[(X-\mathbb{E}[X])^2] = p\cdot(1-p)^2 + (1-p)\cdot(0-p)^2 = p(1-p)(1-p+p) = p(1-p), \tag{63}$$

and the variance of a binomial random variable Y as the sum of n such Bernoulli random variables is

$$\mathrm{Var}[Y] = np(1-p). \tag{64}$$

Theorem 4.4 (Chebyshev's inequality)**.** For any $a > 0$, we have that

$$\mathbb{P}[|X-\mathbb{E}[X]| \geq a] \leq \frac{\mathrm{Var}[X]}{a^2}. \tag{65}$$

Proof. By Markov's inequality, we can deduce that

$$\mathbb{P}[|X-\mathbb{E}[X]| \geq a] = \mathbb{P}[(X-\mathbb{E}[X])^2 \geq a^2] \leq \frac{\mathbb{E}[(X-\mathbb{E}[X])^2]}{a^2} = \frac{\mathrm{Var}[X]}{a^2}, \tag{66}$$

as desired, so the proof is complete. □

3/6 Lecture

4.2 Chernoff Bounds

The **moment generating function** of a random variable X is defined as

$$M_X(t) = \mathbb{E}[e^{tX}], \tag{67}$$

which captures all the moments of X. Indeed, assuming that exchanging the expectation and differential operands is legitimate, we have for all $n > 1$ that

$$M_X^{(n)}(t) = \mathbb{E}[X^n e^{tX}] \implies M_X^{(n)}(0) = \mathbb{E}[X^n]. \tag{68}$$

Next, we introduce the **Chernoff bounds**, such that

$$\mathbb{P}[X \geq a] \leq \min_{t>0} \frac{\mathbb{E}[e^{tX}]}{e^{ta}}, \tag{69}$$

$$\mathbb{P}[X \leq a] \leq \min_{t<0} \frac{\mathbb{E}[e^{tX}]}{e^{ta}}. \tag{70}$$

The proof is trivial. For any $t > 0$, we can deduce by Markov's inequality that

$$\mathbb{P}[X \geq a] = \mathbb{P}[e^{tX} \geq e^{ta}] \leq \frac{\mathbb{E}[e^{tX}]}{e^{ta}}, \tag{71}$$

so the first inequality holds. The deduction of the second inequality is analogous, which will be ignored here.

Chernoff bounds for the sum of Poisson trails. As a preliminary, Poisson trials **differ** from Poisson random variables. Let $X_1, \cdots, X_n$ be a sequence of independent Poisson trials with $\mathbb{P}[X_i = 1] = p_i$ (note that Bernoulli trials are just identically distributed Poisson trials). Now let $X = \sum_{i=1}^n X_i$ denote the total number of successes, and let

$$\mu = \mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n \mathbb{E}[X_i] = \sum_{i=1}^n p_i. \tag{72}$$

Theorem 4.5. In the above setting, the following Chernoff bounds hold:

(1) For any $\delta > 0$, we have that

$$\mathbb{P}[X \geq (1+\delta)\mu] < \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu. \tag{73}$$

(2) For $0 < \delta \leq 1$, we have that

$$\mathbb{P}[X \geq (1+\delta)\mu] \leq e^{-\mu\delta^2/3}. \tag{74}$$

(3) For $R \geq 6\mu$, we have that

$$\mathbb{P}[X \geq R] \leq 2^{-R}. \tag{75}$$

Remark 4.6. The first bound in this theorem is the strongest, and it is from this bound that we derive the second and the third bounds. The second and the third bounds have the advantage of being easier to state and compute with in many situations.

Proof. By Markov's inequality, we have for any $t > 0$ that

$$\begin{aligned}\mathbb{P}[X \geq (1+\delta)\mu] = \mathbb{P}[e^{tX} \geq e^{t(1+\delta)\mu}] &\leq \frac{\mathbb{E}[e^{tX}]}{e^{t(1+\delta)\mu}} = \frac{\prod_{i=1}^n \mathbb{E}[e^{tX_i}]}{e^{t(1+\delta)\mu}} = \frac{\prod_{i=1}^n (p_i e^t + (1-p_i))}{e^{t(1+\delta)\mu}} \\ &= \frac{\prod_{i=1}^n (1 + p_i(e^t - 1))}{e^{t(1+\delta)\mu}} \leq \frac{\prod_{i=1}^n e^{p_i(e^t-1)}}{e^{t(1+\delta)\mu}} = \frac{e^{\left(\sum_{i=1}^n p_i\right)(e^t-1)}}{e^{t(1+\delta)\mu}} = \frac{e^{\mu(e^t-1)}}{e^{t(1+\delta)\mu}}.\end{aligned} \tag{76}$$

By taking $t = \ln(1+\delta)$, we thus have that

$$\mathbb{P}[X \geq (1+\delta)\mu] \leq \frac{e^{\mu\delta}}{(1+\delta)^{(1+\delta)\mu}} = \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu, \qquad \delta > 0. \tag{77}$$

To show the second bound, it suffices to show for $0 < \delta \leq 1$ that

$$\left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu \leq e^{-\mu\delta^2/3} \iff \frac{e^\delta}{(1+\delta)^{1+\delta}} \leq e^{-\delta^2/3} \iff \delta - (1+\delta)\ln(1+\delta) + \frac{\delta^2}{3} \leq 0. \tag{78}$$

This can be shown by easily observing the first and second derivative of the left-hand side with respect to δ, showing that its maximum value in $\delta \in [0,1]$ is obtained at $\delta = 0$. Details will be ignored here. To see the third bound, let $R = (1+\delta)\mu$ with $\delta \geq 5$ (so that $R \geq 6\mu$). Then by using the first bound, we can deduce that

$$\mathbb{P}[X \geq R] \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu \leq \left(\frac{e}{1+\delta}\right)^{(1+\delta)\mu} \leq \left(\frac{e}{6}\right)^R \leq 2^{-R}. \tag{79}$$

Up till now, we have shown all three bounds as desired. □

Theorem 4.7. In the above setting, the following Chernoff bounds hold:

(1) For $0 < \delta < 1$, we have that

$$\mathbb{P}[X \leq (1-\delta)\mu] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^\mu. \tag{80}$$

(2) For $0 < \delta < 1$, we have that

$$\mathbb{P}[X \leq (1-\delta)\mu] \leq e^{-\mu\delta^2/2}. \tag{81}$$

Remark 4.8. Again, the first bound in this theorem is the strongest. The second bound is derived from it, and is generally easier to use and sufficient in most applications.

Proof. By Markov's inequality, we have for any $t < 0$ that

$$\begin{aligned}\mathbb{P}[X \leq (1-\delta)\mu] = \mathbb{P}[e^{tX} \geq e^{t(1-\delta)\mu}] &\leq \frac{\mathbb{E}[e^{tX}]}{e^{t(1-\delta)\mu}} = \frac{\prod_{i=1}^n \mathbb{E}[e^{tX_i}]}{e^{t(1-\delta)\mu}} = \frac{\prod_{i=1}^n (p_i e^t + (1-p_i))}{e^{t(1-\delta)\mu}} \\ &= \frac{\prod_{i=1}^n (1 + p_i(e^t - 1))}{e^{t(1-\delta)\mu}} \leq \frac{\prod_{i=1}^n e^{p_i(e^t-1)}}{e^{t(1-\delta)\mu}} = \frac{e^{\left(\sum_{i=1}^n p_i\right)(e^t-1)}}{e^{t(1-\delta)\mu}} = \frac{e^{\mu(e^t-1)}}{e^{t(1-\delta)\mu}}.\end{aligned} \tag{82}$$

By taking $t = \ln(1-\delta)$, we thus have that

$$\mathbb{P}[X \leq (1-\delta)\mu] \leq \frac{e^{-\mu\delta}}{(1-\delta)^{(1-\delta)\mu}} = \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\mu}, \qquad 0 < \delta < 1. \tag{83}$$

To show the second bound, it suffices to show for $0 < \delta < 1$ that

$$\left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\mu} \leq e^{-\mu\delta^2/2} \iff \frac{e^{-\delta}}{(1-\delta)^{1-\delta}} \leq e^{-\delta^2/2} \iff -\delta - (1-\delta)\ln(1-\delta) + \frac{\delta^2}{2} \leq 0. \tag{84}$$

This can be shown again by observing the first and second derivative of the left-hand side with respect to δ and concluding that it is nonincreasing in the interval $\delta \in [0, 1]$. Details will be ignored here. □

Corollary 4.9. In the above setting, we have for $0 < \delta < 1$ that

$$\mathbb{P}[|X - \mu| \geq \delta\mu] \leq 2e^{-\mu\delta^2/3}. \tag{85}$$

This corollary directly follows from the second bound in Theorem 4.5 and the second bound in Theorem 4.7. However in practice, we do not often have access to $\mathbb{E}[X]$. Therefore, we may use $\mu \geq \mathbb{E}[X]$ in Theorem 4.5 and $\mu \leq \mathbb{E}[X]$ in Theorem 4.7.

Example 4.10. Let X be the number of heads in a sequence of n independent flips of fair coins. Applying Chebyshev's inequality, we will be able to obtain that

$$\mathbb{P}[|X - n/2| \geq n/4] \leq \frac{4}{n}. \tag{86}$$

Using Chernoff bounds, we can further obtain stronger bounds, such that

$$\mathbb{P}[|X - n/2| \geq \sqrt{6n\ln n}/2] \leq 2\exp\left(-\frac{1}{3}\cdot\frac{n}{2}\cdot\frac{6\ln n}{n}\right) = \frac{2}{n}, \tag{87}$$

and

$$\mathbb{P}[|X - n/2| \geq n/4] \leq 2\exp\left(-\frac{1}{3}\cdot\frac{n}{2}\cdot\frac{1}{4}\right) = 2e^{-n/24}. \tag{88}$$

3/8 Lecture

4.3 Better Bounds for Some Special Cases

We can obtain some stronger bounds using a simpler proof technique for some special cases of symmetric random variables. For instance, let $X_1, \cdots, X_n$ be independent and identically distributed random variables such that

$$\mathbb{P}[X_i = 1] = \mathbb{P}[X_i = -1] = \frac{1}{2}, \qquad \forall i \in \{1, \cdots, n\}. \tag{89}$$

Moreover, let $X = \sum_{i=1}^n X_i$.

Theorem 4.11. In the above setting, we have for any $a > 0$ that

$$\mathbb{P}[X \geq a] \leq e^{-a^2/2n}. \tag{90}$$

Proof. For any $t > 0$, we can use the Taylor expansion of e^x to deduce that

$$\mathbb{E}[e^{tX_i}] = \frac{1}{2}(e^t + e^{-t}) = \frac{1}{2}\sum_{i=0}^{\infty} \frac{t^i + (-t)^i}{i!} = \sum_{j=0}^{\infty} \frac{t^{2j}}{(2j)!} \leq \sum_{j=0}^{\infty} \frac{t^{2j}}{2^j \cdot j!} = \sum_{j=0}^{\infty} \frac{(t^2/2)^j}{j!} = e^{t^2/2}. \tag{91}$$

Using this estimate, we thus have that

$$\mathbb{P}[X \geq a] = \mathbb{P}[e^{tX} \geq e^{ta}] \leq \frac{\mathbb{E}[e^{tX}]}{e^{ta}} = \frac{\prod_{i=1}^{n} \mathbb{E}[e^{tX_i}]}{e^{ta}} \leq \frac{e^{t^2 n/2}}{e^{ta}} = e^{t^2 n/2 - ta}. \tag{92}$$

By taking $t = a/n$, we can conclude that

$$\mathbb{P}[X \geq a] \leq e^{a^2/2n - a^2/n} = e^{-a^2/2n}, \tag{93}$$

as desired. □

Note that by symmetry, we also have that $\mathbb{P}[X \leq -a] \leq e^{-a^2/2n}$, so $\mathbb{P}[|X| \geq a] \leq 2e^{-a^2/2n}$. Now we move on to independent and identically distributed random variables $Y_1, \cdots, Y_n$ such that

$$\mathbb{P}[Y_i = 1] = \mathbb{P}[Y_i = 0] = \frac{1}{2}, \qquad \forall i \in \{1, \cdots, n\}. \tag{94}$$

Moreover, let $Y = \sum_{i=1}^{n} Y_i$ and $\mu = \mathbb{E}[Y] = n/2$. By applying the transformation $Y_i = (X_i + 1)/2$, we can easily use the previous theorem to deduce the following results.

Corollary 4.12. In the above setting, we have the following:

(1) For any $a > 0$, we have that

$$\mathbb{P}[Y \geq \mu + a] \leq e^{-2a^2/n}. \tag{95}$$

(2) For any $\delta > 0$, we have that

$$\mathbb{P}[Y \geq (1 + \delta)\mu] \leq e^{-\delta^2 \mu}. \tag{96}$$

Proof. For any $a > 0$, we have that

$$\mathbb{P}[Y \geq \mu + a] = \mathbb{P}\left[\frac{X}{2} + \frac{n}{2} \geq \mu + a\right] = \mathbb{P}[X \geq 2a] \leq e^{-2a^2/n}. \tag{97}$$

Now by setting $a = \delta\mu$, we have that

$$\mathbb{P}[Y \geq (1 + \delta)\mu] = \mathbb{P}[Y \geq \mu + a] \leq e^{-2\delta^2\mu^2/n} = e^{-\delta^2/\mu}, \tag{98}$$

using the fact that $\mu = \mathbb{E}[Y] = n/2$. □

We can deduce the following result analogously as above by symmetry of the random variables.

Corollary 4.13. In the above setting, we have the following:

(1) For any $0 < a < \mu$, we have that

$$\mathbb{P}[Y \leq \mu - a] \leq e^{-2a^2/n}. \tag{99}$$

(2) For any $0 < \delta < 1$, we have that

$$\mathbb{P}[Y \leq (1 - \delta)\mu] \leq e^{-\delta^2 \mu}. \tag{100}$$

4.4 High Probability Analysis of QuickSort

Let e be an item in the input, and consider the sequence of subproblems to which e belongs. This ends with e being chosen as a pivot, which occurs in a subproblem of size 1 or larger. Moreover, we classify the partitions involving e as good or bad. A **good partition** is one in which the subproblem containing e has size at most 3/4 the size of the subproblem being partitioned. Otherwise the partition is considered bad. The probability that a partition is good is at least 1/2, because a bad partition can occur only when the pivot lies in the first or last $\lfloor n/4 \rfloor$ items, where n denotes the size of the subproblem being partitioned.

It follows that the subproblems to which e belongs are obtained by some unknown number of bad partitions together with at most $\log_{4/3} n$ good partitions (because if all are good partitions, in the worst case each partition reduces the subproblem size to 3/4 the previous, that is, $n \cdot (3/4)^{\#} = 1$, which gives the upper bound on #). Since good partitions occur with probability at least 1/2, the expected number of partitions involving e would be at most $2 \log_{4/3} n$.

Now let Y_i denote whether the ith partition involving e is a good partition, *i.e.*, $Y_i = 1$ if the ith partition involving e is a good partition and $Y_i = 0$ otherwise. Suppose that there were $2(d+1)\log_{4/3} n$ partitions involving e and let Y be the sum of all Y_i. By the Chernoff bound on symmetric random variables as in Theorem 4.13, we then have that

$$\mathbb{P}[Y \leq (d+1)\log_{4/3} n - d \log_{4/3} n] \leq \exp\left(-\frac{2d^2(\log_{4/3} n)^2}{2(d+1)\log_{4/3} n}\right) = \exp\left(\frac{-d^2 \log_{4/3} n}{d+1}\right). \tag{101}$$

This implies that there are less than $\log_{4/3} n$ good partitions within a total of $2(d+1)\log_{4/3} n$ partitions involving e with probability at most $\exp\left(\frac{-d^2 \log_{4/3} n}{d+1}\right)$. Furthermore, since the subproblems to which e belongs contain no more than $\log_{4/3} n$ good partitions, the probability that there are more than $2(d+1)\log_{4/3} n$ partitions involving e in total is again at most $\exp\left(\frac{-d^2 \log_{4/3} n}{d+1}\right)$. Interpreting from the negative perspective, it follows that the probability that there are at most $2(d+1)\log_{4/3} n$ partitions involving e in total is at least

$$1 - \exp\left(\frac{-d^2 \log_{4/3} n}{d+1}\right) = 1 - \left(\exp\left(\frac{\ln n}{\ln(4/3)}\right)\right)^{-\frac{d^2}{d+1}} = 1 - \frac{1}{n^{d^2/((d+1)\ln(4/3))}}. \tag{102}$$

Now by the arbitrariness of item e, we can extend the previous statement to every item. In other words, the depth of the recursion is upper bounded by $2(d+1)\log_{4/3} n$ with probability at least $1 - 1/n^{d^2/((d+1)\ln(4/3))}$. Recall that each level of recursion of the QuickSort algorithm performs $O(n)$ operations. Therefore, the runtime is $O(cn \log n)$ with probability at least $1 - 1/n^c$ for some constant c. This can be considered as high probability.

4.5 Partition Sort

This is a family of randomized algorithms which are in-place like QuickSort and have an $O(n \log n)$ average case runtime, but achieve an $O(n \log^2 n)$ worst case runtime. Versions of this family of algorithms run either as fast or faster than the best versions of QuickSort. Here we will consider the version that executes as follows. Step 1: First, we recursively sort the first $n/2$ items. Step 2: Next, we partition the remaining items about the sorted $n/2$ items. Step 3: Finally, we recursively sort each partition.

Implementing the last two steps. We implement the second step recursively, and whenever we create a subproblem in which all items are unsorted, we immediately proceed to the third step for that subproblem. We begin by discussing the initial partition. Let p be the middle item among the sorted $n/2$ items. The intermediate goal is to end up with the following ordering of items.

Sorted items $< p$	Unsorted items $< p$	p	Sorted items $> p$	Unsorted items $> p$

Once this ordering is achieved, the items smaller than p and those larger than p can be partitioned recursively. We achieve this goal as follows. Step A: We first partition the unsorted items about p. Step B: Then, we swap the sorted items that have values at least p with the unsorted items that have values less than p.

In general (and in more detail), suppose that we are partitioning the subarray $A[i : l]$, where the portion $A[i : j]$ is sorted and the remaining portion $A[j+1 : l]$ is unsorted. Then we perform a standard partition of the unsorted

items about $A[\texttt{mid}]$, where $\texttt{mid} = \lfloor (i+j)/2 \rfloor$ is the middle item among $A[i:j]$. The partition procedure determines an index k such that $j \le k \le l$, and puts the items less than $A[\texttt{mid}]$ in $A[j+1:k]$, and the items greater than $A[\texttt{mid}]$ in $A[k+1:l]$. We then rearrange the items to put them in the order described as above via Step B. Following this we recursively rearrange the items smaller than p and the items larger than p. Whenever the recursive partitioning creates a subproblem with no sorted items, we call the partition sort algorithm to sort the items in this subproblem. The algorithm RecPartition for the second and the third steps is described as in Algorithm 7.

Algorithm 7 RecPartition(A, i, j, l)

Require: $A[i:l]$: the subarray to partition; l: the boundary such that $A[i:j]$ is sorted and $A[j+1:l]$ is unsorted.

```
if j < i then
  PartSort(A, i, l);   // all the items are unsorted
else
  m ← ⌊(i + j)/2⌋;
  k ← Partition(A, m, j + 1, l);    // make A[j+1:k] be < A[m] and A[k+1:l] be > A[m]
  for h = 0 to j − m do
    Swap(A[j − h], A[k − h]);    // complete the desired ordering, pivot at k-j+m
  end for
  if k > j then
    RecPartition(A, i, m − 1, k − j + m − 1);   // exists unsorted items < pivot
  end if
  if l > k then
    RecPartition(A, k − j + m + 1, k, l);   // exists unsorted items > pivot
  end if
end if
```

The runtime analysis. For simplicity, we perform an analysis when $n = 2^k$ for some integer k. We begin by bounding the number of comparisons. The number of comparisons needed for partitioning in the second step is at most $1 + \log(n/2) = \log n$ per item, since we are essentially performing a binary search to determine the partition to which it belongs. The remaining issue is to determine the number of comparisons needed to sort the items in each partition in the third step. We begin by considering the worst case, where all the items are in a single partition. What is "a single partition"? In this case, we will have that

$$C(n) = \begin{cases} 0, & \text{if } n = 1, \\ C\left(\frac{n}{2}\right) + \frac{n}{2}\log n + C\left(\frac{n}{2}\right), & \text{otherwise,} \end{cases} \tag{103}$$

because we sort the first $n/2$ items, partition the last $n/2$ items (with each item taking at most $\log n$ comparisons), and in the worst case sort all the last $n/2$ items. This recurrence relation has the solution

$$C(n) = \frac{1}{4} n(\log n + 1)\log n = O(n \log^2 n). \tag{104}$$

We now turn to the average case. To see the number of comparisons in total, it remains to bound the number of comparisons among items in each of the partitions. The probability that the rank-i and rank-k items end up in the same unsorted partition is $1/2^{k-i+1}$, which we will show shortly. In the recursive sort of a partition, these two items may or may not be compared. We overestimate by assuming that they are compared. Let X_{ik} be the Bernoulli random variable that indicates if the items of rank i and rank k are in the same partition, then the expected number of comparisons would be bounded by

$$\mathbb{E}\left[\sum_{1 \le i < k \le n} X_{ik}\right] = \sum_{1 \le i < k \le n} \mathbb{E}[X_{ik}] = \sum_{1 \le i < k \le n} \frac{1}{2^{k-i+1}} \le (n-1)\left(\frac{1}{4} + \frac{1}{8} + \cdots\right) \le \frac{n-1}{2}. \tag{105}$$

This yields the recurrence relation for the expected number of comparisons in the expected case, such that

$$C(n) \le \begin{cases} 0, & \text{if } n = 1, \\ C\left(\frac{n}{2}\right) + \frac{n}{2}\log n + \frac{n-1}{2}, & \text{otherwise.} \end{cases} \tag{106}$$

This has the solution $C(n) \le n \log n$. We now show the bound on $\mathbb{P}[X_{ik} = 1]$. For the rank-i and the rank-k items to be in the same partition, all the items from rank i to rank k must all remain in the unsorted portion of the array. WHY? Let E_j be the event that the rank-j item is in the unsorted portion of the array. Then we have that

$$\mathbb{P}\left[\bigwedge_{j=i}^{k} E_j\right] = \mathbb{P}\left[\bigwedge_{j=1}^{k-1} E_j\right] \cdot \mathbb{P}[E_k | E_i, \cdots, E_{k-1}] = \mathbb{P}\left[\bigwedge_{j=1}^{k-2} E_j\right] \cdot \mathbb{P}[E_{k-1} | E_i, \cdots, E_{k-2}] \cdot \mathbb{P}[E_k | E_i, \cdots, E_{k-1}]$$

$$= \cdots = \mathbb{P}[E_i] \cdot \mathbb{P}[E_{i+1} | E_i] \cdots \mathbb{P}[E_{k-1} | E_i, \cdots, E_{k-2}] \cdot \mathbb{P}[E_k | E_i, \cdots, E_{k-1}] = \left(\frac{1}{2}\right)^{k-i+1} = \frac{1}{2^{k-i+1}}. \quad (107)$$

WHY ARE THOSE CONDITIONAL PROBABILITIES 1/2? The number of data movements in the recursive partitioning is also $O(n \log n)$ WHY?, which implies analogous bounds on the running time, that is, $O(n \log^2 n)$ in the worst case, and $O(n \log n)$ in the average case.

In practice. The initial sort, rather than being applied to $n/2$ items, should be applied to a smaller fraction of the items. This would increase the number of comparisons, but would reduce the number of swaps. In practice, $n/128$ works well by experiments.

3/20 Lecture

4.6 Packet Routing on a Hypercube

Consider a parallel computer which has N processors, where N is fairly large. Each processor is essentially a standard computer. What changes in a parallel computer is that the individual processors need to exchange data, which they do by means of a communication network.

The simplest network would consist of $N(N-1)$ cables, each one carrying traffic in each direction between one pair of processors. However, this is not feasible when N is large; there would simply be too many cables. Instead, we use a sparser network. We describe the setup using graph terminology. Each processor is represented by a node in a graph, and the cables that are present are represented as directed edges. The network we will consider is called a **hypercube**. Here $N = 2^n$ for some integer n. Each processor is identified by a distinct n-bit integer, *e.g.*, $a = a_1 a_2 \cdots a_n$. Processors a and b are connected by edges in each direction if and only if a and b differ in exactly one bit, *i.e.*, $a = a_1 a_2 \cdots a_i \cdots a_n$ and $b = a_1 a_2 \cdots a_{i-1} \overline{a_i} a_{i+1} \cdots a_n$ for some $1 \le i \le n$. Then, the total number of directed edges can be computed as

$$n \cdot 2^{n-1} \cdot 2 = n \cdot 2^n = N \log N, \quad (108)$$

where the first n is because there are n possible bits to differ on, the second 2^{n-1} is because after fixing the differing bit, the rest $n-1$ bits have this many choices, and the third 2 is because for each pair they are connected by edges in each direction. Now note that traffic is sent in units called **packets**. We suppose it takes one unit of time to send a packet across an edge. Because multiple packets might seek to traverse a particular edge at the same time, we have a queue at the head of each edge. When a packet reaches a vertex v, if this is not its destination, it enters the queue for the next edge on its path. The packet at the front of the queue will be transmitted in the next step. Next, we will analyze the cost of routing a permutation, *i.e.*, each vertex sends a packet and each destination receives a packet. The bit-fixing algorithm is described as in Algorithm 8.

Algorithm 8 Bit-fixing

Require: $a = a_1 a_2 \cdots a_n$ is a start node for a packet p and $b = b_1 b_2 \cdots b_n$ is its destination.

```
for i = 1 to n do
    if a_i ≠ b_i then
        fix the ith bit, i.e., move from b_1 ··· b_{i-1} a_i a_{i+1} ··· a_n to b_1 ··· b_{i-1} b_i a_{i+1} ··· a_n;
    end if
end for
```

Consider the path followed by the bit-fixing algorithm in going from a to b. Note that if this path needs to change k bits, then it will have k edges. Necessarily, $k \le n$. Unfortunately, there are permutations that result in $\Omega(\sqrt{N})$

runtime (shown in some Homework) because $\sqrt{N}$ packets must go through a single edge. We give a randomized algorithm which, with high probability, has much better performance. It is called a **two-phase randomized routing protocol**, which proceeds as follows. Each node v chooses an intermediate destination $\phi(v)$ for its packet p_v uniformly at random from all the N nodes. PHASE 1: It first routes p_v from v to $\phi(v)$ using the bit-fixing algorithm. PHASE 2: It then routes p_v from $\phi(v)$ to its destination $\pi(v)$ using again the bit-fixing algorithm.

Analysis of Phase 1. Let γ_v denote the path from v to $\phi(v)$. Suppose that γ_v has k edges (because k bits need to be flipped), and that p_v reaches its destination in time T_v. The **delay** of p_v is defined to be $T_v - k$.

Lemma 4.14 (The delay lemma)**.** Let $S(v) = \{w;\ \gamma_v \text{ and } \gamma_w \text{ share at least one edge}\}$, then $\texttt{delay}(p_v) \le |S(v)|$.

3/27 Lecture

Proof. Suppose the path γ_v for p_v goes through vertices $u_0, u_1, \cdots, u_k$, or alternatively, through edges $e_1, e_2, \cdots, e_k$. Note that strictly speaking, all the notations should be indexed by v, but we will suppress this detail in the interest of readability. At each node u_j there is a queue Q_j of packets waiting to be routed on edge e_{j+1}, the next edge on the path γ_v. We seek to identify a distinct packet p_w "responsible" for each unit of delay that γ_v incurs. To this end, for a packet p_w in $S(v)$, define

$$\texttt{lag}_t(p_w) = \begin{cases} t - j, & \text{if } p_w \text{ is on } Q_j \text{ at time } t, \\ 0, & \text{otherwise.} \end{cases} \tag{109}$$

When the process starts, with p_v at node $v = v_0$, we have $t = 0$, *i.e.*, there is no lag initially. Now suppose that p_w was at the top of Q_j and traverses e_{j+1}, reaching u_{j+1} at time $t + 1$. There are generally three cases.

- **Case 1.** p_w joins Q_{j+1}, then $\texttt{lag}_{t+1}(p_w) = (t+1) - (j+1) = t - j = \texttt{lag}_t(p_w)$.
- **Case 2.** p_w does not join Q_{j+1} and is not at its destination. Thereafter p_w follows a different path. This path will never rejoin γ_v because the bit for p_v's destination corresponding to edge e_{j+2} differs from the same bit in p_w's destination. We say p_w leaves γ_v with lag equal to $\texttt{lag}_t(p_w)$.
- **Case 3.** p_w reaches u_{j+1} and this is its destination. Again we say p_w leaves γ_v with lag equal to $\texttt{lag}_t(p_w)$.

Consider the last time t when p_v has lag l. Then p_v must be on some queue Q_j but not at the top (since its lag increases). Therefore, some other packet p which is currently at the top of Q_j will move forward to u_{j+1}. Either p leaves γ_v at this point or it joins Q_{j+1}. In the latter case, there is some packet p' at the top of Q_{j+1} which has lag l (possibly $p = p'$). WHY? Iterating this process, there must be a final step at which a packet with lag l leaves γ_v. As each packet in $S(v)$ can leave γ_v once at most, and these are the only packets that can leave $S(v)$, we can conclude that the total lag of p_v is at most $|S(v)|$. □

Lemma 4.15. We have that $\mathbb{E}[|S(v)|] \le n/2$.

Proof. Let

$$H_{vw} = \begin{cases} 1, & \text{if } \gamma_v \text{ and } \gamma_w \text{ share at least one edge,} \\ 0, & \text{otherwise.} \end{cases} \tag{110}$$

By the delay lemma, $\texttt{delay}(p_v) \le |S(v)| = \sum_w H_{vw}$. The variables H_{vw} for $w = 0, 1, 2, \cdots$ are mutually independent as the destinations $\phi(w)$ are chosen independently. This suggests we could use a Chernoff bound to bound $\sum_w H_{vw}$. To this end, we first need to bound $\mathbb{E}\left[\sum_w H_{vw}\right]$.

Determining $\mathbb{E}[H_{vw}]$ is a bit complicated, so we bound $\mathbb{E}\left[\sum_w H_{vw}\right]$ directly. Let A denote the edge set and V the vertex set of the hypercube. For each path γ_v, let $\texttt{len}(\gamma_v)$ denote its edge length. For each edge $e \in A$, denote by $N(e)$ the number of paths γ_w that use edge e. Clearly by symmetry, $\mathbb{E}[N(e)]$ takes on the same value for every edge $e \in A$. Therefore, for each edge $e' \in A$, we have that

$$|A| \cdot \mathbb{E}[N(e')] = \sum_{e \in A} \mathbb{E}[N(e)] = \sum_{w \in V} \mathbb{E}[\texttt{len}(\gamma_w)]. \tag{111}$$

It is not hard to see that $\mathbb{E}[\texttt{len}(\gamma_w)] = n/2$, so that

$$\mathbb{E}[N(e')] = \frac{1}{|A|} \cdot |V| \cdot \frac{n}{2} = \frac{1}{nN} \cdot N \cdot \frac{n}{2} = \frac{1}{2}. \tag{112}$$

Define $N^-(e)$ to be the number of paths γ_w that use edge e, where $w \neq v$. Clearly, $N^-(e) \leq N(e)$, and thus

$$\mathbb{E}[N^-(e)] \leq \mathbb{E}[N(e)] = \frac{1}{2}. \tag{113}$$

It then follows that the expected number of nodes w such that γ_v and γ_w share an edge is at most $\mathbb{E}\left[\sum_{e \in \gamma_v} N^-(e)\right]$, since this is the expected number of paths γ_w that use some edge of γ_v but are not γ_v. Since $|\gamma_v| \leq n$, this number would be at most $n/2$. This implies that

$$\mathbb{E}[|S(v)|] = \mathbb{E}\left[\sum_w H_{vw}\right] \leq \frac{n}{2}. \tag{114}$$

□

Corollary 4.16. For $c \geq 6$, we have that

$$\mathbb{P}\left[\texttt{delay}(p_v) \geq \frac{cn}{2}\right] \leq \left(\frac{1}{2}\right)^{cn/2}. \tag{115}$$

Proof. For $c \geq 6$, we have that

$$\frac{cn}{2} \geq 6 \cdot \frac{n}{2} \geq 6 \cdot \mathbb{E}\left[\sum_w H_{vw}\right]. \tag{116}$$

By a Chernoff bound, as the Bernoulli random variables H_{vw} are independent, we thus have for $c \geq 6$ that

$$\mathbb{P}\left[\sum_w H_{vw} \geq \frac{cn}{2}\right] \leq 2^{-cn/2}. \tag{117}$$

By the delay lemma, we can therefore conclude that

$$\mathbb{P}\left[\texttt{delay}(p_v) \geq \frac{cn}{2}\right] \leq \mathbb{P}\left[|S(v)| \geq \frac{cn}{2}\right] \leq \mathbb{P}\left[\sum_w H_{vw} \geq \frac{cn}{2}\right] \leq 2^{-cn/2}, \tag{118}$$

as desired, so the proof is complete. □

Theorem 4.17. The delay for each packet is less than $\frac{c \log N}{2}$ with probability at least $1 - \left(\frac{1}{N}\right)^{(c-1)/2}$ for $c \geq 6$.

Proof. Note that $\frac{c \log N}{2} = \frac{cn}{2}$. Then this theorem follows directly from the previous corollary by observing that

$$1 - N\left(\frac{1}{2}\right)^{cn/2} = 1 - N\left(\frac{1}{N}\right)^{c/2} = 1 - \left(\frac{1}{N}\right)^{(c-2)/2}, \tag{119}$$

as desired, so the proof is complete. □

Analysis of Phase 2. An easy way to approach this is to keep Phase 1 and Phase 2 disjoint. The way to do this is to wait to start Phase 2 until a time at which Phase 1 will have finished with sufficiently high probability. It is straightforward to check that the lemmas, corollaries, and theorems proved for Phase 1 are all applicable in Phase 2 as well. This results in doubling the delay bound and doubling the failure probability bound.

A more natural algorithm is to allow each packet p_v to move to Phase 2 as soon as it completes Phase 1, *i.e.*, as soon as it reaches $\phi(v)$. Now packets in Phase 1 may have edges in common with packets in Phase 2, and we need to bound this effect.

3/29 Lecture

5 Hashing

5.1 Chain Hashing

The balls-and-bins model is useful for modeling hashing. For instance, consider the application of a password checker, which prevents people from using easily crackable passwords by keeping a dictionary of unacceptable passwords. When a user tries to set up a password, the application would like to check if the requested password is part of the unacceptable set. One possible approach would be to store the unacceptable passwords in alphabetical order and perform a binary search. This would require $O(\log m)$ time if there are m words.

Another possibility is to place the words into bins and then search the appropriate bin for the word. The words in a bin would be represented by a linked list. The placement of words into bins is accomplished via using a hash function. A hash function $f : U \to [0, n-1]$ maps the universe of items into n bins. In our example, the universe U would consist of all possible password strings. Moreover, the collection of bins is called a hash table. Using a hash table turns the dictionary problem into a balls-and-bins problem. If our dictionary of unacceptable passwords consists of m words, then we can model the distribution of words in bins with the same distribution as m balls randomly placed in n bins. Note that here we are making a strong assumption that the hash function maps words into bins in a fashion that appears random.

Let us consider the search time when there are n bins and m words. To search for an item, we first hash it to find the bin that it lies in and then search sequentially through the linked list for it. If we search for a word that is not in our dictionary, the expected number of words in the bin that the word hashes to is m/n. If we search for a word that is in our dictionary, the expected number of other words in that words' bin is $(m-1)/n$, and thus the expected number of words in that bin is $1 + (m-1)/n$. If we choose $n = m$ bins for our hash table, then the expected number of words we must search through in a bin would be constant. If the hashing also takes constant time, then the total expected search time would be constant.

The maximum time to search for a word, however, is proportional to the maximum number of words in a bin. We have shown that when $n = m$ this maximum load is $\Theta(\ln n / \ln \ln n)$ with probability close to 1 WHY?, and hence with high probability this is the maximum search time in such a hash table. Though better than binary search, this is much worse then the expected case. Moreover, having n bins for n items can leave several bins potentially empty, leading to a waste of space.

5.2 Bit Strings Hashing

If we want to save space instead of time, we can use hashing in another way. Consider the same problem as above. Assume that a password is restricted to be 8 ASCII characters (64 bits, 8 bytes). Suppose we use a hash function to map each word into a 32-bit string. The string will serve as a short fingerprint of the word. We keep the fingerprints of the unacceptable passwords in a sorted list, and to check if a proposed password is valid, we calculate its fingerprint and look for it on the list, for instance via binary search. If the fingerprint is on the list, we declare it to be unacceptable. However in this case, the password checker may give a wrong answer.

Suppose that we use b bits for a fingerprint, then the probability that an acceptable password has a fingerprint that is different from any specific allowable password is $1 - 1/2^b$. It follows that if the set of unacceptable passwords is of size m, the probability of a **false positive**, *i.e.*, misidentifying an acceptable password as unacceptable, is

$$1 - \left(1 - \frac{1}{2^b}\right)^m \geq 1 - \exp\left(-\frac{m}{2^b}\right). \tag{120}$$

If we want this probability of a false positive to be less than a constant c, we then require

$$\exp\left(-\frac{m}{2^b}\right) \geq 1 - c \implies b \geq \log_2\left(\frac{m}{-\ln(1-c)}\right) = \Omega(\log_2 m). \tag{121}$$

On the other hand, if we use $b = 2\log_2 m$ bits, then the probability of a false positive falls to

$$1 - \left(1 - \frac{1}{m^2}\right)^m < \frac{1}{m}. \tag{122}$$

In our example, if there are 2^{16} unacceptable passwords, then using 32 bits when hashing would yield a false positive probability of less than 2^{-16}.

5.3 Families of Universal Hash Functions

Up to this point, when studying hash functions, we modeled them as being completely random in the sense that, for any collection of items $x_1, \cdots, x_k$, the hash values $h(x_1), \cdots, h(x_k)$ were considered uniform and independent over the range of the hash function. This was the framework we used to analyze hashing as a balls-and-bins problem. However, completely random hash functions are too expensive to compute and store, so the model does not fully reflect reality. Instead, we trade away the strong statements one can make about completely random hash functions for weaker statements with hash functions that are efficient to store and compute.

Definition 5.1. Let U be a universe with $|U| \geq n$ and let $V = \{0, 1, \cdots, n-1\}$. A family of hash functions $\mathscr{H}$ from U to V is said to be k**-universal** if, for any elements $x_1, \cdots, x_k$ and for a hash function h chosen uniformly at random from $\mathscr{H}$, we have

$$\mathbb{P}[h(x_1) = h(x_2) = \cdots = h(x_k)] \leq \frac{1}{n^{k-1}}. \tag{123}$$

A family of hash functions $\mathscr{H}$ from U to V is said to be **strongly** k**-universal** if, for any elements $x_1, \cdots, x_k$, any values $y_1, \cdots, y_k \in V$, and a hash function h chosen uniformly at random from $\mathscr{H}$, we have

$$\mathbb{P}[(h(x_1) = y_1) \cap (h(x_2) = y_2) \cap \cdots \cap (h(x_k) = y_k)] = \frac{1}{n^k}. \tag{124}$$

We will primarily be interested in 2-universal and strongly 2-universal families of hash functions. When we choose a hash function from a family of 2-universal hash functions, the probability that any two elements x_1 and x_2 have the same hash value is at most $1/n$, just like a random hash function. However, it does not guarantee that the probability of any three elements x_1, x_2, and x_3 having the same hash value is at most $1/n^2$. On the other hand, if we choose a function from a family of strongly 2-universal hash functions, the values $h(x_1)$ and $h(x_2)$ are pairwise independent, since the probability that they take on any specific pair of values is $1/n^2$. Because of this, such functions are also known as pairwise independent hash functions.

Now, consider the case where there are m items and n bins. Let the items be labeled $x_1, \cdots, x_m$. For $1 \leq i < j \leq m$, let $X_{ij} = 1$ if items x_i and x_j land in the same bin. Let $X = \sum_{1 \leq i < j \leq m} X_{ij}$ be the total number of collisions between pairs of items. By the linearity of expectations, we have that

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{1 \leq i < j \leq m} X_{ij}\right] = \sum_{1 \leq i < j \leq m} \mathbb{E}[X_{ij}]. \tag{125}$$

Since our hash function is chosen from a 2-universal family, it follows that

$$\mathbb{E}[X_{ij}] = \mathbb{P}[h(x_i) = h(x_j)] \leq \frac{1}{n}. \tag{126}$$

Therefore, we can compute that

$$\mathbb{E}[X] \leq \frac{m(m-1)}{2} \cdot \frac{1}{n} = \frac{m^2 - m}{2n} < \frac{m^2}{2n}. \tag{127}$$

By Markov's inequality, we can see that

$$\mathbb{P}\left[X \geq \frac{m^2}{n}\right] \leq \mathbb{P}[X \geq 2\mathbb{E}[X]] \leq \frac{1}{2}. \tag{128}$$

Now, let Y denote the maximum number of items in a bin, then the number of collisions X must be at least $\binom{Y}{2}$. Therefore, we have that

$$\mathbb{P}\left[\frac{(Y-1)^2}{2} \geq \frac{m^2}{n}\right] \leq \mathbb{P}\left[\binom{Y}{2} \geq \frac{m^2}{n}\right] \leq \mathbb{P}\left[X \geq \frac{m^2}{n}\right] \leq \frac{1}{2} \implies \mathbb{P}\left[Y \geq 1 + m\sqrt{\frac{2}{n}}\right] \leq \frac{1}{2}. \tag{129}$$

In particular, if we take $m = n$, the maximum load is at most $1 + \sqrt{2n}$ with probability at least $1/2$. This result is much weaker, but it is extremely general in that it holds for any 2-universal family of hash functions. The result will prove useful for designing perfect hash functions as we will see later.

Constructing a 2-universal family of hash functions. Let the universe U be the set $\{0, 1, \cdots, m-1\}$ and let the range of our hash function be $V = \{0, 1, \cdots, n-1\}$, with $m \geq n$. Consider the family of hash functions obtained by choosing a prime $p \geq m$, letting

$$h_{a,b}(x) = ((ax - b) \mod p) \mod n, \tag{130}$$

and then taking the family

$$\mathscr{H} = \{h_{a,b};\ 1 \leq a \leq p-1, 0 \leq b \leq p-1\}. \tag{131}$$

Lemma 5.2. $\mathscr{H}$ is 2-universal.

Proof. We count the number of functions in $\mathscr{H}$ for which two distinct elements x_1 and x_2 from U collide. First we note that for any $x_1 \neq x_2$, we have that

$$ax_1 + b \not\equiv ax_2 + b \mod p, \tag{132}$$

since otherwise, $a(x_1 - x_2) \equiv 0 \mod p$, which is impossible since $1 \leq a \leq p-1$ and $1 \leq |x_1 - x_2| \leq m - 1 \leq p - 1$. In fact, for every pair of values (u, v) such that $u \neq v$ and $0 \leq u, v \leq p-1$, there exists exactly one pair of values (a, b), for which $ax_1 + b = u \mod p$ and $ax_2 + b = v \mod p$. This pair of equations has two unknowns, and its unique solution is given by

$$a = \frac{v-u}{x_2 - x_1} \mod p, \qquad b = (u - ax_1) \mod p. \tag{133}$$

Since there is exactly one hash function for each pair (a, b), it follows that there is exactly one hash function in $\mathscr{H}$ for which the previously mentioned pair of equations hold. Therefore, in order to bound the probability that $h_{a,b}(x_1) = h_{a,b}(x_2)$ when $h_{a,b}$ is chosen uniformly at random from $\mathscr{H}$, it suffices to count the number of pairs (u, v), where $0 \leq u, v \leq p-1$, $u \neq v$, and $u \equiv v \mod n$. For each choice of u, there are at most $\lceil p/n \rceil - 1$ possible appropriate values for v, giving at most $p(\lceil p/n \rceil - 1)$ pairs. Moreover, each pair corresponds to one of the $p(p-1)$ hash functions, so that

$$\mathbb{P}[h_{a,b}(x_1) = h_{a,b}(x_2)] \leq \frac{p(\lceil p/n \rceil - 1)}{p(p-1)} = \frac{\lceil \frac{p}{n} \rceil - 1}{p-1} \leq \frac{\frac{p}{n} + \frac{n-1}{n} - 1}{p-1} = \frac{1}{n}. \tag{134}$$

Therefore, we can conclude that $\mathscr{H}$ is 2-universal, thus completing the proof. □

4/3 Lecture

5.4 Perfect Hashing

Perfect hashing is an efficient data structure for storing a **static dictionary**. Items in a static dictionary are permanently stored in a table, and only the search operation is supported. A search of an item gives the location of the item in the table or reports that the items is not in the table. Now suppose that a set S of n items is hashed into a table of n bins, using a hash function from a 2-universal family and chain hashing. The number of operations for looking up an item x is then proportional to the number of items in x's bin. We have the following simple bound.

Lemma 5.3. Assume that m elements are hashed into an n-bin hash table by using a hash function h chosen uniformly at random from a 2-universal family. For an arbitrary element x, let X be the number of items at the bin $h(x)$, then we have that

$$\mathbb{E}[X] \leq \begin{cases} m/n, & \text{if } x \notin S, \\ 1 + (m-1)/n, & \text{if } x \in S. \end{cases} \tag{135}$$

Proof. Let X_i be the random variable that represents whether the ith element of S (under some arbitrary ordering) collides with x. In other words, $X_i = 1$ if it is in the same bin as x and $X_i = 0$ otherwise. Since the hash function is chosen from a 2-universal family, it follows that

$$\mathbb{P}[X_i = 1] \leq \frac{1}{n}. \tag{136}$$

Then, if $x \notin S$, we can easily see that

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{m} X_i\right] = \sum_{i=1}^{m} \mathbb{E}[X_i] \leq \sum_{i=1}^{m} \frac{1}{n} = \frac{m}{n}. \tag{137}$$

Now if $x \in S$, without loss of generality we let it be the first element of S. Therefore, similary to the previous case, we have that

$$\mathbb{P}[X_1 = 1] = 1, \qquad \mathbb{P}[X_i = 1] \leq \frac{1}{n}, \qquad \forall i \geq 2. \tag{138}$$

Therefore, we can obtain that

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{m} X_i\right] = 1 + \sum_{i=2}^{m} \mathbb{E}[X_i] \leq 1 + \sum_{i=2}^{m} \frac{1}{n} = 1 + \frac{m-1}{n}. \tag{139}$$

Combining the two cases, the proof is thus complete. □

The lemma above shows that the average performance of hashing when using a hash function from a 2-universal family is good, since the time to look through a bin of any item is bounded by a small number. Particularly, if $m = n$, then the average number of operations required to look up any item is 1. However, this does not guarantee the worst-case time of a lookup. This motivates the concept of **perfect hashing**. Given a set S, we would like to construct a hash table that gives excellent worst-case performance. Specifically, by perfect hashing, we mean that only a constant number of operations are required to find an item in a hash table (or to determine that it does not exist). We first show that perfect hashing is easy if we are given sufficient space for the hash table and a suitable 2-universal family of hash functions.

Lemma 5.4. If $h \in \mathscr{H}$ is chosen uniformly at random from a 2-universal family of hash functions mapping the universe U to $\{0, 1, \cdots, n-1\}$, then for any subset $S \subseteq U$ of size m, the probability of h being perfect is at least $1/2$ when $n \geq m^2$.

Proof. Let $s_1, \cdots, s_m$ be the m items of S. Let X_{ij} be 1 if $h(s_i) = h(s_j)$ and 0 otherwise. Let $X = \sum_{1 \leq i < j \leq m} X_{ij}$, which represents the total number of collisions, then by (128), we have that

$$\mathbb{P}\left[X \geq \frac{m^2}{n}\right] \leq \frac{1}{2}. \tag{140}$$

Therefore, when $n \geq m^2$, we can deduce that

$$\mathbb{P}[X < 1] = 1 - \mathbb{P}[X \geq 1] \geq 1 - \mathbb{P}\left[X \geq \frac{m^2}{n}\right] \geq \frac{1}{2}, \tag{141}$$

implying that a randomly chosen hash function in a 2-universal family is perfect with probability at least $1/2$. This completes the proof. □

Now to find a perfect hash function when $n \geq m^2$, we may simply try hash functions chosen uniformly at random from a 2-universal family until we find one with no collisions. This gives a Las Vegas algorithm, by which we need to try at most two hash function (in the expected sense). Now, we would like to have perfect hashing without requiring space for $\Omega(m^2)$ bins to store the set of m items. We can use a **two-level scheme** that accomplishes perfect hashing using only $O(m)$ bins. First, we hash the set into a hash table with m bins using a hash function from a 2-universal family. Some of these bins will have collisions. For each such bin, we provide a second hash function from an appropriate 2-universal family and an entirely separate hash table. If the bin has k ($k > 1$) items, then we use k^2 bins in the secondary hash table, since we have already shown in the previous lemma that with k^2 bins, we can find a hash function from a 2-universal family that will give no collisions. Therefore, it remains to show that, by carefully choosing the first hash function, we can guarantee that the total space used by this algorithm in $O(m)$.

Lemma 5.5. The two-level approach gives a perfect hashing scheme for m items using $O(m)$ bins.

Proof. Again by (128), the number of collisions X in the first stage satisfies that

$$\mathbb{P}\left[X \geq \frac{m^2}{n}\right] \leq \frac{1}{2}. \tag{142}$$

When $n = m$, this implies that the probability of having more than m collisions is at most $1/2$. Using the probabilistic method, there exists a choice of hash function from the 2-universal family in the first stage that gives at most m collisions. In fact, such a hash function can be found efficiently by trying hash functions chosen uniformly at random from the 2-universal family, giving a Las Vegas algorithm. We may therefore assume that we have found a hash function for the first stage that gives at most m collisions. Now, let c_i be the number of items in the ith bin. Clearly, there would be $\binom{c_i}{2}$ collisions between items in the ith bin, so we have that

$$\sum_{i=1}^{m} \binom{c_i}{2} \leq m. \tag{143}$$

For each bin with $c_i > 1$ items, we find a second hash function that gives no collisions using space c_i^2. Again, for each bin, this hash function can be found using a Las Vegas algorithm. Therefore, the total number of bins used can be bounded by

$$m + \sum_{i=1}^{m} c_i^2 \leq m + 2\sum_{i=1}^{m} \binom{c_i}{2} + \sum_{i=1}^{n} c_i \leq m + 2m + m = 4m, \tag{144}$$

so that we require only $O(m)$ bins in total. The proof is thus complete. □

4/5 Lecture

6 Random Graphs

6.1 Random Graph Models

There are many NP-hard computational problems defined on graphs: Hamiltonian cycles, independent sets, vertex covers, etc. One question worth asking is, whether these problems are hard for most inputs or just a relatively small fraction of all graphs. Random graph models provide a probabilistic setting for studying such questions.

One random graph model is $G_{n,p}$, where we consider all undirected graphs on n distinct vertices $v_1, \cdots, v_n$. A graph with a given set of m edges has probability $p^m(1-p)^{\binom{n}{2}-m}$. One way to generate a random graph in $G_{n,p}$ is to consider each of the $\binom{n}{2}$ possible edges in some order and then independently add each edge to the graph with probability p. The expected number of edges would then be $p\binom{n}{2}$, and each vertex would have an expected degree of $(n-1)p$.

Another random graph model is $G_{n,N}$, where we consider all undirected graphs on n vertices with exactly N edges. There are $\binom{\binom{n}{2}}{N}$ possible graphs, each selected with equal probability. One way to generate a graph uniformly from the graphs in $G_{n,N}$ is to start with a graph with no edges. Then, choose one of the $\binom{n}{2}$ edges uniformly at random, and add it to the edges in the graph. Now choose one of the remaining $\binom{n}{2} - 1$ possible edges independently and uniformly at random and add it to the graph. Iteratively repeat this process until there are N edges chosen.

6.2 Connectivity

We consider graphs in the $N_{n,p}$ model.

Lemma 6.1. Let $G = (V, E)$ be an undirected n-vertex graph drawn from $G_{n,p}$. Then, if $p = c \ln n/(n-1)$ and $c \geq 3$, the probability that no vertex is isolated is at least $1 - 1/n^{c-1}$.

Proof. The probability that a vertex v is isolated is equivalent to the probability that there is no edge incident on v. Since each possible edge is present with probability p, and there could be n possible edges incident on v, this probability would be $(1-p)^{n-1}$. Now substituting $p = c \ln n/(n-1)$ yields that

$$(1-p)^{n-1} = \left(1 - \frac{c \ln n}{n-1}\right)^{n-1} \leq \exp\left(-c \ln n\right) = \frac{1}{n^c}. \tag{145}$$

Therefore, the probability that no vertex is isolated can be obtained by the union bound, such that

$$\mathbb{P}[\text{no vertex is isolated}] \geq 1 - n \cdot \frac{1}{n^c} = 1 - \frac{1}{n^{c-1}}, \tag{146}$$

as desired, so the proof is complete. □

Lemma 6.2. Let $G = (V, E)$ be an undirected n-vertex graph drawn from $G_{n,p}$. Then, if $p = c \ln n/n$ and $c \geq 3$, the probability that G is connected is at least $1 - 2e/n^{(c-2)/2}$ for $n \geq 2e^2$.

Proof. It suffices to show that for every k-vertex subset S with $k < n/2$, there is at least one edge between a vertex in S and a vertex in $V \backslash S$. Note that the probability that S has no outgoing edge would be $(1-p)^{k(n-k)}$, since there is a total number of $k(n-k)$ edges between S and $V \backslash S$, and we want none of them to be drawn. Therefore, the probability that every k-vertex subset S with $k < n/2$ has some edge to $V \backslash S$ can be obtained by the union bound, and thus is at least

$$\begin{aligned} 1 - \sum_{k \leq n/2} \binom{n}{k} (1-p)^{k(n-k)} &\geq 1 - \sum_{k \leq n/2} \left(\frac{en}{k}\right)^k \left(1 - \frac{c \ln n}{n}\right)^{k \cdot n/2} \geq 1 - \sum_{k \leq n/2} \left(\frac{en}{k}\right)^k \exp\left(-\frac{c \ln n}{n} \cdot \frac{kn}{2}\right) \\ &= 1 - \sum_{k \leq n/2} \left(\frac{en}{k}\right)^k n^{-ck/2} \geq 1 - \sum_{k \leq n/2} \left(\frac{en^{1-c/2}}{k}\right)^k \geq 1 - \sum_{k \leq n/2} \left(en^{1-c/2}\right)^k. \end{aligned} \tag{147}$$

Note that since $n \geq 2e^2$, we have that $en^{1-c/2} \leq e \cdot (2e^2)^{1-c/2} = 2^{1-c/2} e^{3-c} < 1$, and thus

$$\sum_{k \leq n/2} (en^{1-c/2})^k \leq \sum_{k=1}^{\infty} (en^{1-c/2})^k = \frac{en^{1-c/2}}{1 - en^{1-c/2}} = \frac{e}{n^{c/2-1} - e}. \tag{148}$$

Moreover, given that $n \geq 2e^2$ and $c \geq 3$, we have that

$$n^{c/2-1} \geq n^{1/2} \geq 2e \implies e \leq \frac{n^{c/2-1}}{2}, \tag{149}$$

so that the bound on the sum above can be further simplified as

$$\sum_{k \leq n/2} (en^{1-c/2})^k \leq \frac{e}{n^{c/2-1} - e} \leq \frac{e}{n^{c/2-1} - \frac{n^{c/2-1}}{2}} = \frac{2e}{n^{c/2-1}}. \tag{150}$$

Therefore, the probability that every k-vertex subset S with $k < n/2$ has some edge to $V \backslash S$ is at least

$$1 - \sum_{k \leq n/2} \left(en^{1-c/2}\right)^k \geq 1 - \frac{2e}{n^{c/2-1}} = 1 - \frac{2e}{n^{(c-2)/2}}, \tag{151}$$

as desired, so the proof is complete. □

6.3 Cliques in $G_{n,p}$

A k-clique in an undirected graph is a set of k vertices, such that all the possible $\binom{k}{2}$ edges among these vertices are present. Testing whethere a graph has a k-clique for large $k$is an NP-complete problem, meaning that the only known algorithms have exponential running time. For a fixed k, running time $O(mn^k)$ is readily achieved, where m is the number of edges and n is the number of vertices. However, when the graphs are drawn from $G_{n,p}$ with probability $p = 1/2$, as we shall see, graphs with cliques of size $2\log n$ are rare, and on the other hand, there is a simple greedy algorithm that finds cliques of size $\log n - c$ with probability at least $1 - 1/2^c$ in linear time.

Lemma 6.3. Let $G = (V, E)$ be an undirected n-vertex graph drawn from $G_{n,p}$. Then, for each constant $c \geq 1$ and for large enough n, if $p = 1/2$, then the probability that G has a clique of size $k = 2\log n$ is at most $1/n^c$.

4/10 Lecture

Proof. We can compute that

$$\mathbb{P}[G \text{ has } k\text{-clique}] \leq \mathbb{E}[\#\ k\text{-cliques in } G] \leq \binom{n}{k} \cdot \left(\frac{1}{2}\right)^{k(k-1)/2} \leq \left(\frac{ne}{k}\right)^k \cdot \left(\frac{1}{2}\right)^{k(k-1)/2} = \left(\frac{ne}{k \cdot 2^{(k-1)/2}}\right)^k. \quad (152)$$

By substituting $k = 2\log n$, we thus have that

$$\begin{aligned}\mathbb{P}[G \text{ has a clique}] &\leq \left(\frac{ne}{\sqrt{2}n\log n}\right)^{2\log n} = \left(\frac{1}{\frac{\sqrt{2}\log n}{e}}\right)^{2\log n} = \left(\left(\frac{1}{2}\right)^{\log\left(\sqrt{2}\log n/e\right)}\right)^{2\log n} \\ &= \left(\left(\frac{1}{2}\right)^{2\log n}\right)^{\log\left(\sqrt{2}\log n/e\right)} = \left(\frac{1}{n}\right)^{\log\left(\sqrt{2}\log n/e\right)} \leq \frac{1}{n^c}, \end{aligned} \quad (153)$$

for c sufficiently large. Therefore, the proof is complete. □

A greedy algorithm to find a k-clique. We consider $k = \log n - c$. The algorithm initializes a clique C to the empty set. Then it tests vertices v one by one, and adds v to C if it has an edge to every vertex currently in C.

Lemma 6.4. Let (G, E) be an undirected graph drawn from $G_{n,p}$ uniformly at random, where $p = 1/2$. For $c \geq 1$, the greedy algorithm finds a clique of size $\log n - c$ with probability at least $1 - 1/2^c$.

Proof. Let X_i be the number of vertices tested while $|C| = i$. Note that the probability that a vertex v is connected to every vertex in C given $v \notin C$ and $|C| = i$ is $1/2^i$, since there is $1/2$ probability that each edge from v to an element of C is drawn, and there are i such edges in total. Therefore, X_i is a geometric random variable with success probability $1/2^i$, thus $\mathbb{E}[X_i] = 2^i$. Now let X be the number of vertices tested until C reaches size $k = \log n - c$. Suppose we change the algorithm to stop when $|C| = k$, and otherwise, we pretend to keep running using additional imaginary vertices. Then, we have that

$$\mathbb{E}[X] = 1 + 2 + 2^2 + \cdots + 2^{k-1} = 2^k - 1. \quad (154)$$

Therefore, by Markov's inequality, we can see that

$$\mathbb{P}[X > n] < \frac{\mathbb{E}[X]}{n} < \frac{2^k}{n}. \quad (155)$$

Substituting $k = \log n - c$, we thus have that

$$\mathbb{P}[X \leq n] = 1 - \mathbb{P}[X > n] > 1 - \frac{2^{\log n - c}}{n} = 1 - \frac{n2^{-c}}{n} = 1 - \frac{1}{2^c}. \quad (156)$$

Therefore, we can conclude that the greedy algorithm can find a clique of size $\log n - c$ with probability at least $1 - 1/2^c$, and the proof is complete. □

Note that this is indeed a moderately small probability, but is not of the $1/n^c$ type.

6.4 Small World Graphs

The small worlds setting has two features. The first is that there are short paths between every pair of vertices. We show that this property holds for $G_{n,p}$ when p is sufficiently large. The second property concerns whether these paths can be easily found. By this we mean being able to navigate based solely on local information and the destination. More precisely, if one is at a vertex u, then the endpoints of all edges incident on u are known, as is the destination t. The navigation rule is that one has to choose which edge to traverse based on this information alone. Clearly, without some additional structure of the graph, there is little to be done. We will explore a class of graphs due to Jon Kleinberg, in which one can identify shortest paths based solely on local information and the destination.

Shortest paths exist in $G_{n,p}$. Consider a graph $G = (V, E)$ drawn from $G_{n,p}$ with $p = a \ln n/n$, where $a > 0$ is a sufficiently large constant, and $a \ln n/n \leq 1$. Then, we will show that with probability at least $1 - 1/n^c$, for every pair of vertices s and t, there is a path of length at most $b \ln n$ from s to t. We first demonstrate an "expansion" property for these graphs. Suppose that $S \subseteq V$ and let $k = |S|$.

Lemma 6.5. Suppose that $S \subseteq V$ with $k = |S| \leq n/3$. Then, the probability that any such S has fewer than k neighbors in $V \backslash S$ is at most $\frac{2e^2}{n^{2a/3-2}}$ if $a \geq 12$ and $n \geq 2e^2$.

Proof. The probability P that any set S of size $k \leq n/3$ has at most k neighbors in $V \backslash S$ is given by the product of the ways of choosing sets S of size k, a neighboring set of size k (for if it is smaller than k, it is still contained in a set of size k), and the probability that there are no edges between S and the remaining $n - 2k$ vertices, namely

$$\begin{aligned} P &\leq \sum_{k=1}^{n/3} \binom{n}{k}\binom{n-k}{k} \cdot (1-p)^{k(n-2k)} \leq \sum_{k=1}^{n/3} \left(\frac{en}{k}\right)^k \left(\frac{e(n-k)}{k}\right)^k (1-p)^{k(n-2k)} \\ &< \sum_{k=1}^{n/3} \left(\frac{en}{k}\right)^{2k} (1-p)^{k(n-2k)} \leq \sum_{k=1}^{n/3} \left(\frac{en}{k}\right)^{2k} (1-p)^{kn/3}. \end{aligned} \tag{157}$$

Substituting $p = a \ln n/n$ into the expression, we then have that

$$\begin{aligned} P &\leq \sum_{k=1}^{n/3} \left(\frac{en}{k}\right)^{2k} \left(1 - \frac{a \ln n}{n}\right)^{kn/3} \leq \sum_{k=1}^{n/3} \left(\frac{en}{k}\right)^{2k} \exp\left(-\frac{ak \ln n}{3}\right) = \sum_{k=1}^{n/3} \left(\frac{en}{k}\right)^{2k} \left(\frac{1}{n}\right)^{ak/3} \\ &\leq \sum_{k=1}^{n/3} \left(\frac{e^2 n^2}{k^2} \cdot \frac{1}{n^{a/3}}\right)^k = \sum_{k=1}^{n/3} \left(\frac{e^2}{k^2 n^{a/3-2}}\right)^k \leq \sum_{k=1}^{n/3} \left(\frac{e^2}{n^{a/3-2}}\right)^k < \frac{\frac{e^2}{n^{a/3-2}}}{1 - \frac{e^2}{n^{a/3-2}}} \leq \frac{2e^2}{n^{a/3-2}}, \end{aligned} \tag{158}$$

if $a \geq 12$ and $n \geq 2e^2$ in order that $\frac{e^2}{n^{a/3-2}} \leq \frac{1}{2}$. Therefore the proof is complete. □

Now, suppose that $S \subseteq V$ with $k = |S| < n/3$ as before. We consider the event that every such S has at least k neighbors in $V \backslash S$. Let $S_0 = \{s\}$, and iteratively let S_{i+1} be the union of S_i and the neighbors of S_i. Then, clearly $|S_{i+1}| \geq 2|S_i|$ so long as $|S_i| \leq n/3$, since any S of size k would have at least k neighbors in $V \backslash S$. Therefore, iteratively we can see that $|S_i| \geq 2^i$. Now suppose that the set S_j has size $n/3 < |S_j| < 2n/3$, so we must have that $j \leq \lceil \log(n/3) \rceil$. *The reason is that, if $j \geq \lceil \log(n/3) \rceil + 1$, then $j \geq \log(n/3) + 1 = \log(2n/3)$, making $|S_j| \geq 2n/3$ and leading to a contradiction.* Then, S_j has a subset S' containing S_{j-1} of size exactly $n/3$, and S' has at least $n/3$ neighbors in $V \backslash S'$. Therefore, $|S_{j+1}| \geq 2n/3$. *The reason for using S' is that, we cannot make the doubling assumption on S since $|S| > n/3$.* Now, we can see that to reach the at least $2n/3$ vertices in S_{j+1}, which are at distances at most $j + 1$ from s, we need to use paths of length at most $j + 1 \leq \lceil \log(n/3) \rceil + 1 \leq \log(n/3) + 1 = \log(2n/3) \leq \lceil \log(2n/3) \rceil$. Except for the above case where $n/3 < |S_j| < 2n/3$, if $|S_{j-1}| \leq n/3$ we proceed further until there is a set such of size between $n/3$ and $2n/3$; if $|S_{j-1}| \geq 2n/3$ then we can retrieve backwards.

Similarly, paths from another vertex t of length at most $\lceil \log(2n/3) \rceil$ can reach at least $2n/3$ vertices. These two sets of vertices must intersect if $n \geq 3$, since $2 \cdot 2n/3 > n$. Therefore, there is a path of length at most $2 \lceil \log(2n/3) \rceil$ between s and t. This is true for every pair (s, t) of vertices in V, which can be concluded by arbitrariness. Now, note that the event we are looking at is the complement of the event investigated in the previous lemma. Therefore, we have probability at least $1 - \frac{2e^2}{n^{2a/3-2}}$ if $a \geq 12$ and $n \geq 2e^2$. Since we want to achieve high probability in the form $1 - 1/n^c$, it suffices to have that $2a/3 - 2 \geq c + 1 \implies a \geq 3(c + 3)/2$. We can conclude this result in the following theorem.

Theorem 6.6. Let G be a graph drawn from $G_{n,p}$, with $p = a \ln n/n$. If $n \geq 2e^2$, $a \geq 3(c+2)/2$, and $c \geq 5$, then with probability at least $1 - 1/n^c$, there are paths of length at most $2 \log n$ between every pair of vertices in G.

Short and findable paths exist in small worlds graphs. The **small worlds graph** is a directed graph, defined as follows. It has n^2 vertices, formed into an $n \times n$ grid. We identify each vertex using a pair of integers (i, j) for $1 \leq i, j \leq n$. Each vertex has directed edges to its immediate vertical and horizontal neighbors (so 4 edges per vertex, except for those on the sides of the grid). These are called **local edges**. We measure the distance from u to v, denoted by $d(u, v)$, as the minimum number of local edges on a path from u to v (this is also the distance from v to u). In addition, each vertex has one long-distance edge, chosen as follows. u has an edge to v with probability

$$\frac{\frac{1}{d(u,v)^2}}{\sum_{w:d(u,w)\geq 2} \frac{1}{d(u,w)^2}}. \tag{159}$$

The normalizing factor (the term in the denominator) ensures the total probability of a long-distance edge from u sums to 1. Before continuing, we deduce an upper bound on the normalization factor.

Lemma 6.7. We have that

$$\sum_{w:d(u,w)\geq 2} \frac{1}{d(u,w)^2} \leq 4 \ln 2n. \tag{160}$$

Proof. There are at most $4l$ vertices at distance exactly l from vertex u, since if the first coordinate of u is changed by i, the second coordinate must be changed by $\pm(l - i)$. When $i \neq 0, \pm l$, this gives two choices for the second coordinate. There are $2l - 2$ choices such that $0 < |i| < l$, so there are $2(2l - 2) = 4l - 4$ possible vertices in this case. When $i = 0$, the second coordinate can be $\pm l$. When $i = \pm l$, the second coordinate can only be 1. Therefore, overall we have $(4l - 4) + 2 + 2 = 4l$ possible vertices. Therefore, we can compute that

$$\sum_{w:d(u,w)\geq 2} \frac{1}{d(u,w)^2} \leq \sum_{l=2}^{2n-2} \frac{4l}{l^2} = 4 \sum_{l=2}^{2n-2} \frac{1}{l} < 4 \ln 2n, \tag{161}$$

as desired, so the proof is complete. □

4/12 Lecture

Note that having an edge (u, v) does not indicate whether there is also an edge (v, u). To find a path from s to t we use the following greedy algorithm. Suppose the path has reached some vertex x. The next vertex is the neighbor of x that is closest to t in distance, with ties broken arbitrarily. Note that the distance between a vertex (i, j) and the vertex $t = (k, l)$ is given by $|k - i| + |l - j|$, which is readily calculated at vertex x for each of its $O(1)$ neighbors (i, j) in $O(1)$ time. Note that at least one local edge from x reduces the distance to t by 1, and therefore every edge on the chosen path reduces the distance to t.

Let H_x be the set of vertices within distance $d(x, t)/2$ of t. We will show that with probability at least $\alpha/\log n$, x's long-distance neighbor (the far endpoint of its long-distance edge) lies in H_x, where $\alpha > 0$ is a suitable constant. We call call an edge with this property a **halving edge**. Since $d(s, t) \leq 2n - 2 < 2n$, the path from s to t would have at most $\log 2n$ halving edges. *This is because each halving edge at least halves the distance to t.* Therefore, the expected number of edges on the path would be at most $\log 2n \cdot \log n/\alpha$. *This is because with m edges, we will be able to hit at least $m\alpha/\log n$ halving edges, so that in order to hit $\log 2n$ halving edges, we would need at most $\log 2n \cdot \log n/\alpha$ edges.* For brevity going forward, we will denote $d = d(x, t)$.

Lemma 6.8. Let $d \leq 2n - 2$. Then, the number of vertices within distance $d/2$ of t is at least $d^2/16$.

Proof. There are at least $\lceil d/4 \rceil \geq d/4$ vertices on the horizontal line through t within distance $\lfloor d/4 \rfloor$ of t (note that this set includes vertex t), and similarly there are at least $d/4$ vertices on the vertical line through t within distance $\lfloor d/4 \rfloor$ of t. Note that this bound is tight if and only if t is a corner of the grid, but this is enough for our proof. Consequently, the square spanning these two sets of vertices contains vertices all within distance $2 \lfloor d/4 \rfloor \leq d/2$ of t, and note that this square contains at least $d^2/16$ vertices. Therefore, we can conclude that the number of vertices within distance $d/2$ of t is at least $d^2/16$, and the proof is complete. □

Lemma 6.9. The probability P that x's long-distance edge has its far endpoint in H_x is at least $\frac{\ln 2}{288 \log n}$.

Proof. Note that all the vertices in H_x are within distance $3d/2$ of x. *This is because all vertices in H_x are within distance $d/2$ of t, and thus we can get obtain this value by the triangle inequality.* Recall that by definition, x has an edge to some non-neighbor vertex v with probability

$$\frac{\frac{1}{d(x,v)^2}}{\sum_{w:d(x,w)\geq 2} \frac{1}{d(x,w)^2}}. \tag{162}$$

Therefore, we can compute using the union bound that

$$P \geq \frac{d^2}{16} \cdot \frac{\frac{1}{(3d/2)^2}}{4 \ln 2n} = \frac{1}{144 \ln 2n} \geq \frac{1}{288 \ln n} = \frac{\ln 2}{288 \log n}. \tag{163}$$

The proof is thus complete. □

Consequently, we can take $\alpha = \ln 2/288$, so that with probability at least $\alpha / \log n$, x's long-distance neighbor lies in H_x. Note however, that this choice of α is far from tight.

Similar graphs on which the greedy algorithm finds long paths. We use the same underlying grid graph, but we suppose that the probability of a long-distance edge from u to v is given by

$$\frac{\frac{1}{d(u,v)^r}}{\sum_{w:d(u,w)\geq 2} \frac{1}{d(u,w)^r}}, \tag{164}$$

where $r \neq 2$ is a constant. We will show that when $r < 2$, for each pair (s,t) of vertices with $d(s,t) \geq n/2$ and with probability at least $1/2$, employing the greedy algorithm as before leads to a long path from s to t. One can also show that when $r < 2$, with high probability there are short paths. In contrast, when $r > 2$, one can show that there is a substantial probability of no short path.

Lemma 6.10. Let $b = \frac{n^{(2-r)/3}}{2^{4/3}(2-r)^{1/3}}$. Then for $r < 2$, if $n/2 > b$, the probability that $d(s,t) \leq b$ given $d(s,t) \geq n/2$ is at most $1/2$.

Proof. Define U to be the set of vertices within distance b of t. Then, in order for a path from s to t to have edge length at most b, it must have one long-distance edge with its far endpoint in U. Let $\mathscr{E}_i$ be the event that the ith vertex on the greedy algorithm path from s to t has a long-distance edge into U, then

$$\mathbb{P}[\mathscr{E}_i] \leq \frac{|U|}{\sum_{l=2}^{n-1} l/l^r} = \frac{|U|}{\sum_{l=2}^{n-1} 1/l^{r-1}}, \tag{165}$$

where the numerator in the probability bound overestimates the term $1/d^r$ for each $u \in U$, and the denominator underestimates the normalization term. WHY? REALLY CONFUSED... Now, note that we can bound

$$\sum_{l=2}^{n-1} \frac{1}{l^{r-1}} \geq \int_2^n \frac{1}{l^{r-1}} dl = \frac{n^{2-r} - 2^{2-r}}{2-r} \geq \frac{n^{2-r}}{2(2-r)}, \tag{166}$$

as long as $n^{2-r} \geq 2 \cdot 2^{2-r} = 2^{3-r} \implies n \geq 2^{(3-r)/(2-r)}$. Therefore, by the union bound, the probability P that any of the first b edges reach a vertex with long-distance edge into U is at most

$$P \leq \frac{b|U| \cdot 2(2-r)}{n^{2-r}}. \tag{167}$$

Now we note that

$$|U| \leq 1 + 1 \cdot 4 + 2 \cdot 4 + \cdots + b \cdot 4 = 2b(b+1) + 1 \leq 4b^2, \tag{168}$$

as long as $b \geq 2$. Substituting into the upper bound of P, we can thus obtain that

$$P \leq \frac{8b^3(2-r)}{n^{2-r}} \leq \frac{1}{2}, \tag{169}$$

if we take

$$16b^3(2-r) \leq n^{2-r} \implies b^3 \leq \frac{n^{2-r}}{16(2-r)} \implies b \leq \frac{n^{(2-r)/3}}{2^{4/3}(2-r)^{1/3}}, \tag{170}$$

so the proof is complete. □

4/17 Lecture

Did not attend the lecture due to attending NSDI'23 in Boston. As follows are copied (perhaps reformulated) from the provided lecture materials.

7 Algorithmic Game Theory

7.1 No-Regret Dynamics

We discuss the **regret minimization problem**, which concerns a single decision maker playing a game against an adversary. We consider a set A of $n \geq 2$ actions, and the setup is as follows. At each time $t = 1, \cdots, T$, a decision maker picks a mixed strategy p^t (*i.e.*, a probability distribution) over its actions A. After that, an adversary picks a cost vector $c^t : A \to [0, 1]$. Then an action a^t is chosen according to the distribution p^t, and the decision maker incurs cost $c^t(a^t)$. Note that the decision maker learns the entire cost vector c^t rather than just the realized cost $c^t(a^t)$.

We seek a "good" algorithm for online decision-making of this type. However, the setup seems unfair since the adversary is allowed to choose a cost function after the decision maker has committed to a mixed strategy. Therefore, we seek some kind of guarantee that we could possibly hope for in such a model.

Impossiblity with respect to the best action sequence. There is no hope of comparing the cost of an online decision-making algorithm to the cost of the best action sequence in hindsight. The latter quantity $\sum_{t=1}^{T} \min_{a \in A} c^t(a)$ is too strong a benchmark. For instance, suppose $n = 2$ and fix an arbitrary online decision making algorithm. On each day t, the adversary chooses the cost vector c^t such that $c^t = (1, 0)$ if the algorithm plays the first strategy with probability at least $1/2$ and $c^t = (0, 1)$ otherwise. The adversary has forced the expected cost of the algorithm to be at least $T/2$ while ensuring that the cost of the best action sequence in hindsight is 0. This example motivates that rather than comparing with the best action sequence in hindsight, we compare with the best *fixed* action in hindsight.

Definition 7.1. The **(time-averaged) regret** of the action sequence $a^1, \cdots, a^T$ is

$$\frac{1}{T}\left(\sum_{t=1}^{T} c^t(a^t) - \min_{a \in A} \sum_{t=1}^{T} c^t(a)\right). \tag{171}$$

Definition 7.2. An online decision making algorithm is **no-regret** if for every $\epsilon > 0$, there exists a time $T(\epsilon) > 0$, such that the expectation of the regret is $\leq \epsilon$ for all $T \geq T(\epsilon)$.

Randomization is necessary for no-regret. A simple consequence of asymmetry between the decision maker and the adversary is that there does not exist a no-regret deterministic algorithm. To see this, suppose there are $n \geq 2$ actions and we fix a deterministic algorithm. At each time step t, the algorithm commits to a single action a^t. The obvious strategy for the adversary is to set the cost of the action a^t to be 1, and the cost of any other action to be 0. Then, the cost of the algorithm is T while the best single action has cost at most T/n (since otherwise the sum of the costs of all single actions will exceed T which is impossible). The regret would then be at least

$$\frac{1}{T}\left(T - \frac{T}{n}\right) = 1 - \frac{1}{n}, \tag{172}$$

which is constant. Therefore, no deterministic algorithm can be no-regret and randomization is unavoidable.

Lower bound on regret. Now we show that even with only $n = 2$ actions, no randomized algorithm has expected regret vanishing faster than the rate $\Theta(1/\sqrt{T})$. A similar argument will show that with n actions, the expected regret cannot vanish faster than $\Theta(\sqrt{\ln n/T})$. Consider an adversary that, on each day T, chooses independently and uniformly at random between the cost vectors $(0, 1)$ and $(1, 0)$. No matter how smart or dumb an online decision making algorithm is, its cumulative expected cost is exactly $T/2$. In hindsight, however, with constant probability, one of the two fixed actions has cumulative cost only $T/2 - \Theta(\sqrt{T})$. The reason is that, if you flip T fair coins, while the expected number of heads is $T/2$, the standard deviation is $\Theta(\sqrt{T})$. It then follows that for every algorithm, there exists an adversary for which the algorithm has expected regret $\Omega(1/\sqrt{T})$. What?

No-regret algorithms exists. The main result is that no-regret algorithms exist. We state the following theorem and propose the design guidelines of our solution algorithm.

Theorem 7.3. For every set A of n vertices with costs in $[-1, 1]$, there is an online decision making algorithm that, for every adversary, has expected regret at most $2\sqrt{\ln n/T}$.

Our solution is the **multiplicative weights algorithm** as we will introduce later. We keep a probability $p^t(a_i)$ of choosing the action a_i at time t. The algorithm simply updates p^t to p^{t+1} after the costs of step t are revealed. As follows are some design guidelines. First, past performance determines the probabilities. The higher the observed cost of an action, the lower its probability. Second, the probability of choosing an expensive action should decrease at an exponential rate, *i.e.*, by a fractional multiplier.

The multiplicative weights (MW) algorithm. Initially, we take $w^1(a) = 1$ for all $a \in A$. Then for $t = 1, \cdots, T$, *i.e.*, in each iteration, we let the probability of taking the action a_i be given by

$$p^t(a_i) = \frac{w^t(a_i)}{\Gamma^t}, \qquad \text{where } \Gamma^t = \sum_{i=1}^{n} w^t(a_i). \tag{173}$$

Then after the cost vector $c^t(a_i)$, $i = 1, \cdots, n$ is revealed in each iteration, we set

$$w^{t+1}(a_i) \leftarrow w^t(a_i)\left(1 - \eta c^t(a_i)\right), \qquad i = 1, \cdots, n, \tag{174}$$

where we choose $0 < \eta < 1/2$ as the "learning" rate. There is a tradeoff between exploration and exploitation. With smaller η, the probability distribution p^t will be closer to uniform and thus we can explore more actions. With larger η, we push the probability largely to high-performing actions, which exploits our observations.

The analysis. The adversaries can be categorizes into **adaptive adversaries** and **oblivious adversaries**. For an adaptive adversary, c^t may depend on the outcomes of the previous steps. For an oblivious adversary however, $c^1, \cdots, c^T$ are determined upfront (though not revealed to the agent in advance). It suffices to analyze oblivious adversaries, since the multiplicative weights algorithm depends only on the cost functions and not on the actions. Thus, there is no reason for a worst-case adversary to condition its cost vectors on previous realized actions. Similarly, a worst-case adversary does not need to explicitly condition on the distributions, since these are uniquely determined by the adversary's previous cost vectors.

Suppose that $T \geq 4 \ln n$. Fix $c^1, \cdots, c^T$, and let v^t denote the expectation of the cost of the multiplicative weights algorithm at time step t. Then we have that

$$v^t = \sum_{i=1}^{n} p^t(a_i) c^t(a_i) = \frac{1}{\Gamma^t} \sum_{i=1}^{n} w^t(a_i) c^t(a_i). \tag{175}$$

Lemma 7.4. We have that

$$\Gamma^{T+1} \leq n \exp\left(-\eta \sum_{t=1}^{T} v^t\right). \tag{176}$$

Proof. We can compute that

$$\begin{aligned} \Gamma^{t+1} &= \sum_{i=1}^{n} w^{t+1}(a_i) = \sum_{i=1}^{n} w^t(a_i)\left(1 - \eta c^t(a_i)\right) = \sum_{i=1}^{n} w^t(a_i) - \eta \sum_{i=1}^{n} w^t(a_i) c^t(a_i) \\ &= \Gamma^t - \eta \Gamma^t v^t = \Gamma^t\left(1 - \eta v^t\right) \leq \Gamma^t \exp\left(-\eta v^t\right). \end{aligned} \tag{177}$$

Therefore, iteratively applying this result, we can obtain that

$$\Gamma^{T+1} \leq \Gamma_1 \prod_{t=1}^{T} \exp\left(-\eta v^t\right) = n \exp\left(-\eta \sum_{t=1}^{T} v^t\right), \tag{178}$$

as desired, so the proof is complete. □

Lemma 7.5. Let a^* be the optimal single choice of action over the T time steps, and define $\texttt{Opt} = \sum_{t=1}^{T} c^t(a^*)$, *i.e.*, the optimal single choice cost. Then we have that

$$\Gamma^{T+1} \geq \exp\left(-\eta \texttt{Opt} - \eta^2 T\right). \tag{179}$$

Proof. We have that

$$\Gamma^{T+1} = \sum_{i=1}^{n} w^{T+1}(a_i) \geq w^{T+1}(a^*) = w^1(a^*) \prod_{t=1}^{T} \left(1 - \eta c^t(a^*)\right) = \prod_{t=1}^{T} \left(1 - \eta c^t(a^*)\right). \tag{180}$$

Note that for $|x| < 1/2$, from the Taylor expansion we can deduce that

$$\ln(1-x) = -\sum_{n=1}^{\infty} \frac{x^n}{n} = -x - \sum_{n=2}^{\infty} \frac{x^n}{n} = -x - \sum_{n=0}^{\infty} \frac{x^{n+2}}{n+2} \geq -x - \frac{x^2}{2} \sum_{n=0}^{\infty} x^n = -x - \frac{x^2}{2} \cdot \frac{1}{1-x} \geq -x - x^2. \tag{181}$$

Therefore, since $0 < \eta < 1/2$, we then have that

$$\begin{aligned} \Gamma^{T+1} &\geq \prod_{t=1}^{T} \left(1 - \eta c^t(a^*)\right) \geq \prod_{t=1}^{T} \exp\left(-\eta c^t(a^*) - \left(\eta c^t(a^*)\right)^2\right) = \exp\left(-\eta \sum_{t=1}^{T} c^t(a^*) - \eta^2 \sum_{t=1}^{T} \left(c^t(a^*)\right)^2\right) \\ &\geq \exp\left(-\eta \texttt{Opt} - \eta^2 \sum_{t=1}^{T} 1\right) = \exp\left(-\eta \texttt{Opt} - \eta^2 T\right), \end{aligned} \tag{182}$$

where the last inequality follows since we assume all costs are within the $[-1, 1]$ range. The proof is thus complete. □

Theorem 7.6. The expectation of cost of the multiplicative weights algorithm is at most $\texttt{Opt} + 2\sqrt{T \ln n}$.

Proof. By the previous two lemmas, we can see that

$$\exp\left(-\eta \texttt{Opt} - \eta^2 T\right) \leq \Gamma^{T+1} \leq n \exp\left(-\eta \sum_{t=1}^{T} v^t\right). \tag{183}$$

Taking natural logarithm on the two sides of this inequality, we can see that

$$-\eta \texttt{Opt} - \eta^2 T \leq \ln n - \eta \sum_{t=1}^{T} v^t = \ln n - \eta \cdot \mathbb{E}[\text{cost of MW}] \implies \mathbb{E}[\text{cost of MW}] \leq \texttt{Opt} + \eta T + \frac{\ln n}{\eta}. \tag{184}$$

By choosing the learning rate as $\eta = \sqrt{\ln n / T}$, we thus have that

$$\mathbb{E}[\text{cost of MW}] \leq \texttt{Opt} + T\sqrt{\frac{\ln n}{T}} + \frac{\ln n}{\sqrt{\frac{\ln n}{T}}} = \texttt{Opt} + 2\sqrt{T \ln n}, \tag{185}$$

as desired, so the proof is complete. □

From this theorem, we can see that there exists an algorithm (multiplicative weights algorithm with the specified learning rate), such that

$$\mathbb{E}[\text{regret}] \leq \frac{1}{T} \cdot 2\sqrt{T \ln n} = 2\sqrt{\frac{\ln n}{T}} \to 0, \qquad \text{as } T \to \infty, \tag{186}$$

so that it fulfills the requirement of Theorem 7.3, and thus is a no-regret algorithm.

Corollary 7.7. To achieve regret $\leq \epsilon$, one takes at most $4 \ln n / \epsilon^2$ steps.

Proof. Set $T = \epsilon^2 / (4 \ln n)$ in Theorem 7.3, and this lemma directly follows. □

4/19 Lecture

Did not attend the lecture due to attending NSDI'23 in Boston.

4/24 Lecture

8 Online Algorithms

An online problem is one where not all the input is known at the beginning. Rather, the input is presented in stages, and the algorithm needs to process this input as it is received. As the algorithm does not know the rest of the input, it may not be able to make optimum decisions.

8.1 Buying Skis

A simple example of an online algorithm is the ski-rental problem. A skier can either rent skis for a day or buy them once and for all. Buying a pair of skis costs k times as much as renting it. Also, skis once bought cannot be returned. Whenever the skier goes skiing, he does not know whether he will ski again or how many times. This makes the problem an online problem.

Consider the following scenario. A malicious *Weather God* lets him ski as long as he is renting skis, but does not let him go on any more skiing trips after he has bought them. Suppose the skier decides to rent skis on the first k' trips and buys them on the next trip. The total cost incurred by the skier is $k'+k$, but the optimum cost is $\min(k, k'+1)$. Therefore, the skier ends up paying a factor of $(k'+k)/\min(k, k'+1)$ times as much as the optimum. Choosing $k' = k-1$, this factor equals to $2 - 1/k$. This factor is called the **competitive ratio** of the algorithm. Note that when calculating this factor, we always use the worst possible input sequence. More formally, an online algorithm A is **α-competitive** if for all input sequences σ, we have that

$$C_A(\sigma) \leq \alpha \cdot C_{\texttt{Opt}}(\sigma) + \delta, \tag{187}$$

where C_A is the cost function of the algorithm A, $C_{\texttt{Opt}}$ is the cost function of the optimum, and δ is some cost function.

For the skier problem, if $k' < k-1$, then

$$\frac{k'+k}{\min(k, k'+1)} = \frac{k'+k}{k'+1} = 1 + \frac{k-1}{k'+1} > 1 + \frac{k-1}{k} = 2 - \frac{1}{k}, \tag{188}$$

and if $k' > k-1$, then

$$\frac{k'+k}{\min(k, k'+1)} = \frac{k'+k}{k} = 1 + \frac{k'}{k} > 1 + \frac{k-1}{k} = 2 - \frac{1}{k}. \tag{189}$$

Therefore, $2 - 1/k$ is both a lower and an upper bound on the best competitive ratio, attained only when $k' = k-1$.

8.2 Paging

Paging is a more realistic example of an online problem. We have some number of pages, and a cache of k pages. Each time a program requests a page, we need to ensure that it is in the cache. We incur zero cost if the page is already in the cache. Otherwise, we need to fetch the page from a secondary storage (*e.g.*, a disk) and put it in the cache, meanwhile kicking out one of the pages already in the cache. This operation costs one unit. The online decision we need to make is which page to kick out when a page fault occurs. The initial contents in the cache are assumed the same for all algorithms. Some online strategies include LIFO (Last In First Out), FIFO (First In First Out), LRU (Least Recently Used), and LFU (Least Frequently Used).

Theorem 8.1. LRU is k-competitive for the online paging problem.

Proof. Define a phase as a sequence of requests during which LRU faults exactly k times. All we need to show is that during each phase, the optimal algorithm `Opt` must fault at least once.

- **Case 1:** LRU faults at least twice on some page σ_i during the phase. In this case, σ_i must have been brought in once and then kicked out again during the phase. When σ_i was brought in, it becomes the most recently used page. When it is later kicked out, σ_i must have been the least recently used page because we are using

LRU. Therefore, each of the other $k-1$ pages in the cache at that time must have been requested at least once since σ_i was brought in. Counting σ_i and the request that causes σ_i to be evicted, at least $k+1$ distinct pages have been requested in this phase. Therefore, `Opt` must have faulted at least once.

- **Case 2:** LRU faults on k distinct pages. If this is the first phase, then `Opt` must fault at least once since `Opt` and LRU start off with the same pages in their cache (otherwise `Opt` does not even fault on the first page request, so the page must be in the cache, and LRU could not have faulted on it either). Else, let σ_i be the last page requested in the previous phase. This page has to be in both LRU's and `Opt`'s cache at the start of the current phase. Now if the k pages that LRU faults on in the current phase does not include σ_i, then `Opt` must also have received these k requests in the current phase. These k requests together with σ_i that exists in the cache at the start of the current phase are $k+1$ pages in total, so `Opt` must have faulted at least once. In the other case where the k pages that LRU faults on in the current phase includes σ_i, then similar to **Case 1**, σ_i has been brought in and then kicked out, so we will not repeat the reasoning.

Therefore, in each phase `Opt` faults at least once, and thus LRU is k-competitive, so the proof is complete. □

Oblivious adversary. For randomized online algorithms, we need to define our adversary carefully. An **oblivious adversary** is one that does not know the coin tosses of the online algorithm. That is, it knows the source code of the online algorithm but does not have access to its random source during execution. An online algorithm A is then **α-competitive against oblivious adversaries** if for all inputs σ, we have that

$$\mathbb{E}[C_A(\sigma)] \leq \alpha \cdot C_{\texttt{Opt}}(\sigma) + \delta, \tag{190}$$

where δ is a fixed constant, and the expectation is taken over the random coin tosses. We now present **MARK**, the marking algorithm for randomized paging and show that it is $O(\log k)$-competitive against oblivious adversaries. When a request for a page p comes in, MARK can be described as follows.

Algorithm 9 MARK (randomized paging)

Require: the current request: page p;
1: **if** p is not in the cache **then**
2: **if** all pages in the cache are marked **then**
3: unmark all pages in the cache;
4: **end if**
5: randomly choose an unmarked page for eviction;
6: **end if**
7: mark p;

Note that LRU is a deterministic version of MARK, where one uses a specific deterministic rule to choose which one of the unmarked pages to evict. Let us consider an intermediate version of MARK where we choose an unmarked page for eviction deterministically. We consider a phase to be the time between two unmarking resets by the deterministic MARK. During one phase, we bring in k new pages. To be a little more general, suppose `Opt` has a cache of size h. `Opt` can prepare $h-1$ slots (1 slot must be taken by the previous request) before the phase, and must fault on the remaining $k-(h-1)$ slots. Thus, the competitive ratio is $k/(k-h+1)$. For $k=2h$, thus is 2-competitive, but for $k=h$, this is k-competitive, so we need randomization to do better.

4/26 Lecture

Randomized MARK. Let a phase be a sequence during which k distinct pages are requested. At the beginning of each phase, all pages in MARK's cache are marked. Since the page requested during a phase is never evicted again during the same phase, we need to consider only the first time a page is requested within a phase. Let S_i be the set of pages in MARK's cache at the beginning of the ith phase. A request p during a phase is "old" if $p \in S_i$ and "new" otherwise. Let l_i be the number of "new" requests during the ith phase. Each "new" request must cost one unit, so we need to calculate the costs of requests for "old" pages during the phase, *i.e.*, the number of times we accidentally kick out an old request and have to bring it back in. Consider the point during the ith phase where we have served s requests for old pages, and we are serving the $(s+1)$th request for an old page. The s previously requested "old" pages are already in the cache, in addition to the (at most) l_i "new" pages, so at most l_i "old" pages

have been kicked out, and they have been chosen randomly from the $(k-s)$ "old" pages that have not yet been requested. The probability that p was one of these is at most $l_i/(k-s)$, and summing over s from 0 to $k-l_i-1$, the expected total cost of all the "old" requests in a phase is at most

$$l_i\left(\frac{1}{k}+\frac{1}{k-1}+\cdots+\frac{1}{l_i+1}\right)\leq l_i(H_k-1) \tag{191}$$

where H_k is the kth harmonic number. Adding the cost of all "new" requests, the expected total cost of MARK is then bounded by $\sum_i l_i H_k$.

Lemma 8.2. The cost of `Opt` on an input sequence is at least $\frac{1}{2}\sum_i l_i$.

Proof. The main idea is to show that if an offline algorithm does not fault on many "new" requests during the ith phase, then it must evict a lot of pages in the previous and following phases. Even though it is variable, we can take the performance of MARK as a baseline and compare `Opt` against it. Let ϕ_i be the number of pages in S_i (the cache of MARK) that are not in the cache of `Opt` at the beginning of the ith phase (denoted phase_i). Then the following two conditions hold.

- $C_{\texttt{Opt}}(\text{phase}_i)\geq l_i-\phi_i$. During the ith phase, l_i of the pages requested are not in S_i. But the cache of `Opt` differs from S_i in at most ϕ_i pages. Therefore, `Opt` must pay at least $l_i-\phi_i$.
- $C_{\texttt{Opt}}(\text{phase}_i)\geq \phi_{i+1}$. Each of the k pages requested in the ith phase is present in S_{i+1}, and each of these pages was present in the cache of `Opt` at some time during the ith phase. But the cache of `Opt` at the start of the next phase does not have ϕ_{i+1} of these pages, so there must have been at least ϕ_{i+1} evictions by `Opt` during the ith phase.

Adding these two together over all phases, we get

$$2C_{\texttt{Opt}}\geq\sum_i l_i+\phi_n-\phi_0\geq\sum_i l_i-\phi_0, \tag{192}$$

where n is the number of phases. Since $\phi_0=0$, $C_{\texttt{Opt}}\geq\frac{1}{2}\sum_i l_i$, so the proof is complete. □

We can now conclude that MARK is $2H_k$-competitive against oblivious adversaries. Since $H_k=O(\log k)$, the desired result follows, *i.e.*, MARK is $O(\log k)$-competitive against oblivious adversaries. Note however, that if the adversary were not oblivious, it could have chosen a freshly evicted page for all "old" requests.