

## CS 170 HW 13

Due on 2018-11-25, at 9:59 pm

### 1 (★) Study Group

List the names and SIDs of the members in your study group.

### 2 (★★★) Universal Hashing

Let  $[m]$  denote the set  $\{0, 1, \dots, m-1\}$ . For each of the following families of hash functions, determine whether or not it is universal. If it is universal, determine how many random bits are needed to choose a function from the family.

- (a)  $H = \{h_{a_1, a_2} : a_1, a_2 \in [m]\}$ , where  $m$  is a fixed prime and

$$h_{a_1, a_2}(x_1, x_2) = a_1x_1 + a_2x_2 \pmod{m}$$

Notice that each of these functions has signature  $h_{a_1, a_2} : [m]^2 \rightarrow [m]$ , that is, it maps a pair of integers in  $[m]$  to a single integer in  $[m]$ .

- (b)  $H$  is as before, except that now  $m = 2^k$  is some fixed power of 2.

- (c)  $H$  is the set of all functions  $f : [m] \rightarrow [m-1]$ .

#### Solution:

- (a) The hash function is universal. The universality proof is the same as the one in the textbook (only now we have a 2-universal family instead of 4-universal). To reiterate, assume we are given two distinct pairs of integers  $x = (x_1, x_2)$  and  $y = (y_1, y_2)$ . Without loss of generality, let's assume that  $x_1 \neq y_1$ . If we chose values  $a_1$  and  $a_2$  that hash  $x$  and  $y$  to the same value, then  $a_1x_1 + a_2x_2 \equiv a_1y_1 + a_2y_2 \pmod{m}$ . We can rewrite this as  $a_1(x_1 - y_1) \equiv a_2(y_2 - x_2) \pmod{m}$ . Let  $c \equiv a_2(y_2 - x_2) \pmod{m}$ . Since  $m$  is prime and  $x_1 \neq y_1$ ,  $(x_1 - y_1)$  must have a unique inverse. So  $a_1(x_1 - y_1) \equiv a_2(y_2 - x_2) \pmod{m}$  if and only if  $a_1 \equiv c(x_1 - y_1)^{-1} \pmod{m}$ , which will only happen with probability  $1/m$ .

We need to randomly pick two integers in the range  $[0, \dots, m-1]$ , so we need  $2 \log m$  random bits.

- (b) This family is not universal. Consider the following inputs:  $(x_1, x_2) = (0, 2^{k-1})$  and  $(y_1, y_2) = (2^{k-1}, 0)$ . We then have  $h_{\alpha_1, \alpha_2}(x_1, x_2) = 2^{k-1}\alpha_2 \pmod{2^k}$  and  $h_{\alpha_1, \alpha_2}(y_1, y_2) = 2^{k-1}\alpha_1 \pmod{2^k}$ . Now notice that if  $\alpha_2$  is even (i.e. with probability  $1/2$ ) then  $h_{\alpha_1, \alpha_2}(x_1, x_2) = 0 \pmod{2^k}$  otherwise (if  $\alpha_2$  is odd)  $h_{\alpha_1, \alpha_2}(x_1, x_2) = 2^{k-1} \pmod{2^k}$ ; likewise for  $\alpha_1$ . So we get that  $h_{\alpha_1, \alpha_2}(x_1, x_2) = h_{\alpha_1, \alpha_2}(y_1, y_2)$  with probability  $1/2 > 1/2^k$ , so the family is not universal.
- (c) This family is universal. To see that, fix  $x, y \in \{0, 1, \dots, m-1\}$  with  $x \neq y$ . Now we need to figure out the following: how many (out of the  $(m-1)^m$  in total) functions  $f : [m] \rightarrow [m-1]$  will collide on  $x$  and  $y$ , i.e.  $f(x) = f(y) = k$ , for some fixed  $k \in [m-1]$ .

Well, there are  $(m-1)^{m-2}$  different functions  $f: [m] \rightarrow [m-1]$  that have the property  $f(x) = f(y) = k$  (because I just fixed the output of 2 inputs to some fixed  $k \in [m-1]$  and allow the output of  $f$  for all other inputs to range over all  $m-1$  possible values). Finally, ranging over all  $m-1$  values of  $k$ , we get that there are  $(m-1)^{m-1}$  functions  $f: [m] \rightarrow [m-1]$  with the property  $f(x) = f(y)$ . So the probability of picking one such  $f$  is exactly  $\frac{(m-1)^{m-1}}{(m-1)^m} = \frac{1}{m-1}$ .

How many bits do we need in this case? Well, there is no succinct representation in this case, so we need to write down the whole family of functions explicitly and then pick one of the  $(m-1)^m$  functions of the family. To do that we can imagine indexing all functions with integers  $1, \dots, (m-1)^m$  and randomly picking one such integer  $k \in \{1, \dots, (m-1)^m\}$ ; this obviously requires  $\log(m-1)^m = m \log(m-1)$  bits.

### 3 (★★★) Preventing Conflicts

A group of  $n$  guests shows up to a house for a party, and any two guests are either friends or enemies. There are two rooms in the house, and the host wants to distribute guests among the rooms, breaking up as many pairs of enemies as possible. The guests are all waiting outside the house and are impatient to get in, so the host needs to assign them to the two rooms quickly, even if this means that it's not the best possible solution. In this problem, we aim to design an efficient algorithm that breaks up at least half the number of pairs of enemies as the best possible solution.

- (a) In homework 5 we constructed a greedy 1/2-approximation algorithm for “Preventing Conflict” that runs in time  $O(|V| + |E|)$ . Design a randomized algorithm that achieves (in expectation) a 1/2-approximation guarantee in time  $O(|V|)$ . Prove its correctness.
- (b) Your solution in part (a) probably had to flip  $O(|V|)$  random coins. Coins are expensive! Use hashing to give an algorithm that only flips  $O(\log |V|)$  random coins.

#### Solution:

- (a) **Main idea:** Assign each person to an independent, uniformly random room.

#### Pseudocode:

**procedure** PREVENTINGCONFLICT-RANODMIZED

$R_0, R_1 \leftarrow \emptyset.$

▷ Initialization:

Draw  $b_1, \dots, b_n \sim \{0, 1\}$  uniformly at random.

**for**  $v \in V$  **do**

▷ Main loop:

$R_{b_v} \leftarrow R_{b_v} \cup \{v\}.$

**Analysis:** For every pair of enemies  $u, v$ , we have that

$$\Pr[u \text{ and } v \text{ are in the same room}] = \Pr[(u \in R_1) \wedge (v \in R_1)] + \Pr[(u \in R_0) \wedge (v \in R_0)] = 1/2.$$

- (b) **Main idea:** Represent each guest  $v$  as a string of  $t = \lceil \log_2 |V| \rceil$  bits  $v = (v_1, \dots, v_t)$ . Instead of assigning guests to rooms independently at random, we use a universal family of hash functions  $h : V \rightarrow \{0, 1\}$ .

**Pseudocode:**

**procedure** PREVENTINGCONFLICT-HASHING

$R_0, R_1 \leftarrow \emptyset.$

▷ Initialization:

Draw  $b_1, \dots, b_t \sim \{0, 1\}$  uniformly at random.

**for**  $v \in V$  **do**

▷ Main loop:

$h(v) \leftarrow \sum_i b_i v_i \pmod{2}$

$R_{h(v)} \leftarrow R_{h(v)} \cup \{v\}.$

**Analysis:** By the property of universal hash functions:

$$\Pr[u \text{ and } v \text{ are in the same room}] = \Pr[h(u) = h(v)] = 1/2.$$

## 4 (★★★) Streaming for Voting

Consider the following scenario. Votes are being cast for a major election, but due to a lack of resources, only one computer is available to count the votes. Furthermore, this computer only has enough space to store one vote at a time, plus a single extra integer. Each vote is a single integer 0 or 1, denoting a vote for Candidate A and Candidate B respectively.

- (a) Come up with an algorithm to determine whether candidate A or B won, or if there was a tie.
- (b) Consider now an election with 3 candidates. Say there is a winner only if a candidate receives more than 50 percent of the vote, otherwise there is no winner. If we're given another integer's worth of storage, come up with an algorithm to determine the winner if there is one. For simplicity, your algorithm can output any of the candidates in the case that there is no winner (not necessarily the one with the most votes). Votes are now numbered 0, 1, 2.

**Solution:**

- (a) Initialize one of the integers  $i$  to 0. Use the other to store the incoming votes. For every vote for Candidate A, decrement  $i$  by one. For every vote to candidate B, increment  $i$  by 1.

If at the end  $i$  is negative, Candidate A won. If at the end  $i$  is positive, Candidate B won. If  $i$  is 0, there was a tie.

- (b) Let  $m$  be a variable which will hold either 0, 1, or 2. Let  $i$  be a counter similar to in the previous part. For each element in the stream, do the following. If  $i$  is 0, set  $m$  equal to the value of the current vote, and set  $i$  to 1. Else if  $i > 0$ , and  $m$  is equal to the current vote in the stream, increment  $i$ . Else decrement  $i$ .

At the end, if there is a majority vote,  $m$  will contain it. However, if there is no majority vote,  $m$  will still contain some element of the stream.

We can see that this is the case with the following short inductive proof:

Base case: Our algorithm will definitely detect the majority element of a 1 element stream.

Inductive Hypothesis: Assume our algorithm works for a stream of  $n$  or fewer elements.

We have two cases. Either the first candidate's count drops to zero at some point during the streaming, or it doesn't. If the first candidate's count never drops to zero, then the first candidate must be the majority candidate.

If the first candidate's count does drop to zero (let's say after the  $x$ th vote), then we can say the following. Suppose we have not yet encountered the true majority element. Then it must be the majority of the rest of the stream. By the inductive hypothesis we will find it by the end of the algorithm. If we have encountered the true majority element already, we can say the following. This element could have appeared only up to  $\frac{x}{2}$  times so far. Of the remaining  $n - x$  elements, at least  $(\frac{n}{2} + 1 - \frac{x}{2}) = \frac{n-x}{2} + 1$  must be the majority element. Thus the true majority element will also be the majority of the rest of the stream. We can then apply the inductive hypothesis again, completing the proof.