1 Universal Hashing

Let \([m]\) denote the set \(\{0,1,\ldots,m-1\}\). Recall that a family of functions \(\mathcal{H}\) is universal if for any \(x \neq y\), \(\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq 1/m\). That is, the chance that \(h(x) = h(y)\) if we sample \(h\) uniformly at random from \(\mathcal{H}\) is at most \(1/m\).

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
\[
h_{a_1,a_2}(x_1,x_2) = a_1x_1 + a_2x_2 \mod m
\]
Notice that each of these functions has signature \(h_{a_1,a_2} : [m]^2 \to [m]\), that is, it maps a pair of integers in \([m]\) to a single integer in \([m]\).

(b) \(H\) is as before, except that now \(m = 2^k\) for \(k > 1\) is some fixed power of 2.

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

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
\[
a_1x_1 + a_2x_2 \equiv a_1y_1 + a_2y_2 \mod m.
\]
We can rewrite this as
\[
a_1(x_1 - y_1) \equiv a_2(y_2 - x_2) \mod m.
\]
Let \(c \equiv a_2(y_2 - x_2) \mod m\). Since \(m\) is prime and \(x_1 \neq y_1\), \((x_1 - y_1)\) must have a unique inverse. So
\[
a_1(x_1 - y_1) \equiv a_2(y_2 - x_2) \mod m
\]
if and only if \(a_1 \equiv c(x_1 - y_1)^{-1} \mod m\), which will only happen with probability \(1/m\).

We need to randomly pick two integers in the range \([0,\ldots,m-1]\), 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
\[
h_{a_1,a_2}(x_1,x_2) = 2^{k-1}a_2 \mod 2^k
\]
and
\[
h_{a_1,a_2}(y_1,y_2) = 2^{k-1}a_1 \mod 2^k
\]
Now notice that if \(a_2\) is even (i.e. with probability 1/2) then \(h_{a_1,a_2}(x_1,x_2) = 0 \mod 2^k\) otherwise (if \(a_2\) is odd) \(h_{a_1,a_2}(x_1,x_2) = 2^{k-1} \mod 2^k\); likewise for \(a_1\). So we get that
with probability $1/2 > 1/2^k$, so the family is not universal.

(c) This family is universal. To see that, fix $x, y \in \{0, 1, \ldots, m-1\}$ with $x \neq y$. Now we need to figure out the following: how many (out of the $(m-1)^m$ in total) functions $f : [m] \to [m-1]$ 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 $f : [m] \to [m-1]$ 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 $f : [m] \to [m-1]$ with the property $f(x) = f(y)$. So the probability of picking one such $f$ is exactly $(m-1)^{m-1} = \frac{1}{m-1}$.

There are $(m-1)^m$ functions in this family, so we need $\log(m-1)^m = m \log(m-1)$ bits to distinguish between them.

2 Monte Carlo Games

Let’s suppose we have a Monte Carlo algorithm (a randomized algorithm which has a deterministic bound on its runtime, but which only outputs the correct answer some of the time). Call this algorithm $A$; then $A(x, r)$ is the output of $A$ on input $x$ and random bits $r$. In this question, we will think of $A$ as a distribution over many deterministic algorithms. Convince yourself that this makes sense: after all, if we fix a setting to the random bits $r$, we get $A_r(x)$, which is a deterministic algorithm (which may be wrong on some inputs). Let’s fix a set of algorithms $S$ (say, polynomial-time algorithms). Note that $A$ has whatever property defines $S$ if and only if it is a distribution over only algorithms in $S$ (for example, we say a Monte Carlo algorithm is polynomial time if and only if it runs in polynomial time for all settings to the randomness, which is equivalent to all the deterministic algorithms in its distribution running in polynomial time).

We will define a function $c(a, x)$ which indicates whether the deterministic algorithm $a \in S$ is correct on input $x$; $c(a, x) = 1$ if $a$ is correct on input $x$, and 0 if it is incorrect.

Let’s use this function to define a zero-sum game; the row player will choose $a$ and the column player will choose $x$; then a payoff of $c(a, x)$ will go to the row player.

(a) Describe the action and goal of the row and column players. Interpret these in the setting of ‘correctness of the randomly chosen algorithm’ that we constructed the game from. Hint: Since $c(a, x)$ is an indicator, $\mathbb{E}[c(a, x)] = \text{Pr}[c(a, x) = 1]$.

Solution: The row player chooses a distribution over algorithms in order to maximize $\min_a \mathbb{E}[c(a, x)] = \min_a \text{Pr}[c(a, x) = 1]$; that is, maximize the probability the algorithm will be correct, for the worst-case input for that distribution. The column player chooses a distribution over inputs in order to minimize $\max_a \mathbb{E}[c(a, x)] = \max_a \text{Pr}[c(a, x) = 1]$; that is, minimize the probability that any single algorithm will be correct.

(b) Using zero-sum game duality in conjunction with your interpretation above, what can we say about a problem if we know there exists a polynomial-time randomized algorithm which is correct with probability $2/3$ on all inputs? What can we say if we know that there is a distribution of inputs on which no deterministic algorithm is correct with probability $2/3$? Hint: use the fact that a randomized algorithm induces a distribution over deterministic algorithms.

Solution: If there is a randomized algorithm which is correct with probability at least $2/3$ on all inputs, then there is a distribution $D_a$ for which $\min_r \text{Pr}_{x \sim D_a}[c(a, x) = 1] \geq 2/3$; by weak zero-sum game duality, this means that for all distributions $D_x$, $\max_a \text{Pr}_{x \sim D_x}[c(a, x) = 1] \geq 2/3$; in other words, for any distribution over the inputs, there is a deterministic algorithm which is
correct with probability at least $2/3$ on an input randomly selected from that distribution. The second question is the contrapositive of the first: if we have this average-case hardness result for deterministic algorithms, we know there cannot be a good randomized algorithm.

Here we used correctness, but this argument works just as well for any ‘cost’ of an algorithm (time, space, ...) and is known as Yao’s principle: The performance of the best deterministic algorithm on an average-case input is no better than the performance of the best randomized algorithm on a worst-case input with average-case randomness.

3 Streaming Integers

In this problem, we assume we are given an infinite stream of integers $x_1, x_2, \ldots$, and have to perform some computation after each new integer is given. Since we may see many integers, we want to limit the amount of memory we have to use in total. For all of the parts below, give a brief description of your algorithm and a brief justification of its correctness.

(a) Show that using only a single bit of memory, we can compute whether the sum of all integers seen so far is even or odd.

(b) Show that we can compute whether the sum of all integers seen so far is divisible by some fixed integer $N$ using $O(\log N)$ bits of memory.

(c) Assume $N$ is prime. Give an algorithm to check if $N$ divides the product of all integers seen so far, using as few bits of memory as possible.

(d) Now let $N$ be an arbitrary integer, and suppose we are given its prime factorization: $N = p_1^{k_1} p_2^{k_2} \cdots p_r^{k_r}$. Give an algorithm to check whether $N$ divides the product of all seen integers so far, using as few bits of memory as possible. Write down the number of bits your algorithm uses in terms of $k_1, \ldots, k_r$.

Solution:

(a) We set our single bit to 1 if and only if the sum of all integers seen so far is odd. This is sufficient since we don’t need to store any other information about the integers we’ve seen so far.

(b) Set $y_0 = 0$. After each new integer $x_i$, we set $y_i = y_{i-1} + x_i \mod N$. The sum of all seen integers at step $i$ is divisible by $N$ if and only if $y_i \equiv 0 \mod N$. Since each $y_i$ is between 0 and $N - 1$, it only takes $\log N$ bits to represent $y_i$.

(c) We can do this with a single bit $b$. Initially set $b = 0$. Since $N$ is prime, $N$ can only divide the product of all $x_i$s if there is a specific $i$ such that $N$ divides $x_i$. After each new $x_i$, check if $N$ divides $x_i$. If it does, set $b = 1$. $b$ will equal 1 if and only if $N$ divides the product of all seen integers.

(d) We can do this with $\lceil \log_2(k_1 + 1) \rceil + \lceil \log_2(k_2 + 1) \rceil + \cdots + \lceil \log_2(k_r + 1) \rceil$ bits. For each $i$ between 1 and $r$, we track the largest value $t_i \leq k_i$ such that $p_i^{t_i}$ divides the product of all seen numbers. We start with $t_i = 0$ for all $i$. When a new number $m$ is seen, we find the largest $t_i'$ such that $p_i^{t_i'}$ divides $m$ and set $t_i = \min\{t_i', t_i, k_i\}$. We stop once $t_i = k_i$ for all $i$ as this implies that $N$ divides the product of all seen numbers.

4 Lower Bounds for Streaming

(a) Consider the following simple ‘sketching’ problem. Preprocess a sequence of bits $b_1, \ldots, b_n$ so that, given an integer $i$, we can return $b_i$. How many bits of memory are required to solve this problem exactly?
(b) Given a stream of integers \(x_1, x_2, \ldots\), the *majority element* problem is to output the integer which appears most frequently of all of the integers seen so far. Prove that any algorithm which solves the majority element problem exactly must use \(\Omega(n)\) bits of memory, where \(n\) is the number of elements seen so far.

Solution:

(a) \(n\) bits. Intuitively, this is because at the end of the preprocessing, there are \(2^n\) different ‘states’ the algorithm has to be in, one for each bitstring. The number of states of a machine with \(\ell\) bits of memory is \(2^\ell\), so \(\ell \geq n\). A more detailed argument follows.

The preprocessing algorithm is a function \(f: \{0, 1\}^n \rightarrow \{0, 1\}^\ell\), where \(\ell\) is the number of bits of memory needed to answer queries. The query algorithm is a function \(q: [n] \times \{0, 1\}^\ell \rightarrow \{0, 1\}\). Observe that we can use the query algorithm to invert \(f\) on its image: if \(g(y) = (q(1, y), q(2, y), \ldots, q(n, y))\), then \(g(f(x)) = x\) for all \(x \in \{0, 1\}^n\). Hence \(f\) is injective, which means that \(\ell \geq n\).

(b) We can prove this by reduction from the previous problem. For any string of bits \(b_1, \ldots, b_\ell\), we define a stream of integers \(0, 0, (i, i)_{b_i = 1}\). Now we can query \(b_i\) by adding \(i\) to the stream and checking if it is the majority element. The length of the sequence is \(n \leq 2\ell + 1\), so the memory usage is at least \(\frac{n-1}{2}\) bits.

An alternative approach is for the stream to be \((-1)^{b_i} \cdot i, i \in [n]\). Then we can query \(b_i\) by adding \((i, -i)\) to the stream. If \(-i\) is the majority element, then \(b_i = 1\), otherwise \(b_i = 0\).