CS 170 HW 12

Due on 2018-04-23, at 11:59 pm

1 (★) Study Group

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

2 (★★★★) One-Sided Error and Las Vegas Algorithms

An RP algorithm is a randomized algorithm that runs in polynomial time and always gives the correct answer when the correct answer is 'NO', but only gives the correct answer with probability greater than $1/2$ when the correct answer is 'YES'.

(a) Prove that every problem in RP is in NP (i.e., show that $RP \subseteq NP$).

(b) In the last homework, we saw an example of a *Las Vegas Algorithm*: a random algorithm which always gives the right solution, but whose runtime is random. A ZPP algorithm is a Las Vegas algorithm which runs in expected polynomial time (ZPP stands for Zero-Error Probabilistic Polytime). Prove that if a problem has a ZPP algorithm, then it has an RP algorithm.

Solution:

(a) Assume we have some problem that is in RP and let A be an algorithm that solves this problem. If A is given an input x such that the correct answer given input x is 'YES', then A will return 'YES' will probability greater than $1/2$. We want to use A to construct an NP algorithm for x.

Randomized algorithms run like regular algorithms, but will occasionally use random coin-flips to make decisions. For any randomized algorithm B, we can build a deterministic algorithm B' that takes the outcome of those coin-flips beforehand and runs the same computation as B. Let A' be that deterministic algorithm for A.

Since A is an RP algorithm, there must be a poly-size sequence of coin-flip outcomes $\vec{b} = b_1, \ldots, b_k \in \{0,1\}^k$ such that A returns 'YES' given input x and outcomes $\vec{b}$. If we treat $\vec{b}$ as the 'solution' to x, we can see that A' is an NP algorithm for the given problem. Thus every problem in RP is in NP.

(b) Let A be any ZPP algorithm and assume A runs in expected time at most n^k for some constant k. We construct an RP algorithm A' that given input x of size n, does the following:

- Run A for $2n^k$ steps, then stop.
- If A returns 'YES' in that time, return 'YES'
- If A returns 'NO' in that time, return 'NO'
- Otherwise, return 'NO'

How do we know A' is an RP algorithm? First of all, if the correct answer on input x is 'NO', then either A will stop and A' will say 'NO' or A will not stop and A' will still say 'NO'. So A' will always give the right answer when the right answer is 'NO'.

No assume the correct answer is 'YES'. If A stops in time, A' will give the correct answer. But if A does not stop in time, then A' will give the wrong answer. So we want to bound the probability that A does not stop in time.

Let X be a random variable representing the number of steps A will take on input x. We know that $E[X] \leq n^k$. By Markov's inequality this gives us

$$Pr[X \geq a] \leq \frac{n^k}{a}$$

If we set $a = 2n^k$, this gives us

$$Pr[X \geq 2n^k] \leq \frac{n^k}{2n^k} = \frac{1}{2}$$

Thus if the correct answer on input x is 'YES', A' will give the correct answer with probability greater than $1/2$.

So A' is an RP algorithm.

3 (★★★★★) Polynomial Identity Testing

Suppose we are given a polynomial $p(x_1, \ldots, x_k)$ over the reals such that p is not in any normal form. For example, we might be given the polynomial:

$$p(x_1, x_2, x_3) = (3x_1 + 2x_2)(2x_3 - 2_x1)(x_1 + x_2 + 3x_3) + x_2$$

We want to test if p is equal to 0, meaning that $p(a_1, a_2, \ldots, a_k) = 0$ on all real inputs $a_1, a_2, \ldots a_k$. This suggests a simple algorithm to check if p equals 0: test p on some $a_1, \ldots, a_k$ and say p equals 0 if and only if $p(a_1, \ldots, a_k) = 0$. However, this algorithm will fail if $p(a_1, \ldots, a_k) = 0$ and p does not equal 0. We will show through the following questions that if the degree of p is d, then for any finite set of reals S, by randomly choosing $a_1, \ldots, a_k \in S$, we get $Pr[p(a_1, \ldots, a_k) = 0] \leq \frac{d}{|S|}$. So by choosing a large enough subset, S, we are very likely to correctly determine whether p is equal to 0 or not. Note that the degree of, say, $p(x, y) = x^2y^2 + x^3$ is 4, since the monomial x^2y^2 has degree $4 = 2 + 2$. You may assume here that all basic operations on the reals take constant time.

(a) First let's see why a probabilistic algorithm might be necessary. Give a naive deterministic algorithm to test if a given polynomial p is equal to 0. What is the worst-case runtime of your algorithm?

(b) Show that if p is univariate and degree d, then $Pr[p(a_1) = 0] \leq \frac{d}{|S|}$

(c) Show by induction on k that for all polynomials p on k variables:

$$Pr[p(a_1, \ldots, a_k) = 0] \leq \frac{d}{|S|}$$

Hint: Write $p(x_1, \ldots, x_n) = \sum_{i=0}^{d} p_i(x_1, \ldots, x_{n-1}) x_n^i$. Consider the largest i such that the polynomial p_i is not equal to 0, and check $Pr[p_i(x_1, \ldots, x_{n-1}) = 0]$. Then use the fact that $Pr[A] = Pr[A \cap B] + Pr[A \cap \bar{B}] = Pr[A|B]Pr[B] + Pr[A|\bar{B}]Pr[\bar{B}] \leq Pr[B] + Pr[A|\bar{B}]$ for events A and B (and complement event $\bar{B}$).

(d) Use the bound you found in the last part to show a way to check if two polynomials p and q are identical (i.e., yield the same outputs on all inputs).

Solution:

(a) We can multiply out all parentheses and determine if there is any monomial with a non-zero coefficient. This takes exponential time in the worst case (e.g., it takes 2^k time if $p(x_1, x2) = (x_1 + x_2)(2x_1 + 2x_2) \ldots (kx_1 + kx_2)$)

(b) A degree d univariate polynomial can have at most d zeros. In the worst case, all d zeros are in S, yielding the probability $Pr[p(a_1) = 0] = \frac{d}{|S|}$.

(c) Let i be the largest value such that p_i is not equal to 0. Let B be the event that $Pr[p_i(x_1, \ldots, x_{n-1}) = 0]$. Since p is degree d, p_i is degree at most $d - i$. So $Pr[B] \leq \frac{d-i}{|S|}$.

Now assume we randomly chose some $a_1, \ldots, a_{n-1}$ such that $p_i(a_1, \ldots, a_{n-1}) \neq 0$. When we apply these values $a_1, \ldots, a_{n-1}$ to p we get a degree i polynomial of a single variable (call it $p'(x_n)$) By the induction hypothesis, $Pr[p'(x_n) = 0] \leq \frac{i}{|S|}$. If we let A be the event that $p(x_1, \ldots, x_n) = 0$, this is equivalent to saying that $Pr[A|\bar{B}] \leq \frac{i}{|S|}$.

Using the identity from the hint, we get $Pr[A] \leq Pr[B] + Pr[A|\bar{B}] \leq \frac{d-i}{|S|} + \frac{i}{|S|} = \frac{d}{|S|}$, and we're done.

(d) Given $p(x_1, \ldots, x_k)$ and $q(x_1, \ldots, x_k)$, we create a new polynomial $r(x_1, \ldots, x_k) = p(x_1, \ldots, x_k) - q(x_1, \ldots, x_k)$. We know r is equal to zero if and only if p and q are equivalent, so we can just apply our algorithm from the last part to r.

4 (★★★) Universal Hashing

Let $[m]$ denote the set $\{0, 1, ..., m-1\}$. For each of the following families of hash functions, say whether or not it is universal, and 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

$$h_{a_1,a_2}(x_1, x_2) = a_1 x_1 + a_2 x_2 \mod m$$

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 $m = 2^k$ 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 $h_{\alpha_1,\alpha_2}(x_1, x_2) = 2^{k-1}\alpha_2 \mod 2^k$ and $h_{\alpha_1,\alpha_2}(y_1, y_2) = 2^{k-1}\alpha_1 \mod 2^k$. Now notice that if α_2 is even (i.e. with probability $1/2$) then $h_{\alpha_1,\alpha_2}(x_1, x_2) = 0 \mod 2^k$ otherwise (if α_2 is odd) $h_{\alpha_1,\alpha_2}(x_1, x_2) = 2^{k-1} \mod 2^k$; likewise for α_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 $x, y \in \{0, 1, \ldots, m-1\}$ 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 $\frac{(m-1)^{m-1}}{(m-1)^m} = \frac{1}{m-1}$.

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 $1, \ldots, (m-1)^m$ and randomly picking one such integer $k \in \{1, \ldots, (m-1^m)\}$; this obviously requires $\log(m-1)^m = m \log(m-1)$ bits.

5 (★★★) Streaming Algorithms

In this problem, we assume we are given a stream of integers x_1,x_2, ..., 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 store after each new integer.

(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 using $log(N)$ bits of memory, we can compute whether the sum of all integers seen so far is divisible by some fixed number N.

(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) Assume N is not prime, and we are given the prime factorization of N as $p_1^{k_1} p_2^{k_2} \dots p_r^{k_r}$. Give an algorithm to check if N divides the product of all integers seen so far, using as few bits of memory as possible. Represent the number of bits you use in terms of the prime factorization of N.

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_is 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 $\lceil log_2(k_1) \rceil + \lceil log_2(k_2) \rceil + \dots + \lceil log_2(k_r) \rceil$ bits. For each i between 1 and r, we track the largest value $t_i \leq k_i$ such that p^{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 $p^{t_i'}$ 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.