CSE 208A Homework 1
Randomized algorithms
Due Tuesday, May 7

For each of the algorithm problems, design as asymptotically efficient an algorithm as possible. Give a correctness argument (explanation, if it is relatively simple, or proof if not) and time analysis. You may use any well-known algorithm or data structure, or algorithm from the text or from class, as a sub-routine without needing to provide details.

Plentiful pigeons for finding quardratic non-residues: Exercise 14.8 of MR, page 405.

Actually, exercise 14.8 has a small problem: it should read, for any *odd* prime, exactly half the elements of Z_p^* are quardratic residues. (It is easy to see that for $p = 2$ the one member 1 of Z_2^* is a quardratic residue).

Let $p \geq 3$ be a prime. Note that the map f from Z_p^* to Z_p^* defined by $f(x) = x^2 \ mod \ p$ is exactly 2-to-1. First, $f(p-x) = (p-x)^2 \ modp = x^2 \ modp = f(x)$. Secondly, if $f(x) = f(y)$, p divides $x^2 - y^2 = (x-y)(x+y)$ and either $x = y$ or $x = p - y$, so the above is all of the possible collisions. Since f is 2-1, its range (quardratic residues mod p) is exactly $1/2$ the size of its domain Z_p^*. So the quardratic residues are exactly half the elements of Z_p^*. (Where did we use p is odd?)

The probabilistic Las Vegas algorithm is immediate from the above and Euler's Criterion (page 404 of MR). Pick $q \in Z_p^*$ at random. Compute $q^{(p-1)/2} \ modp$ using the method of repeated squaring, using $O(|p|)$ arithmetic operations $modp$. If it is -1 output q, otherwise repeat the whole process. Since half the elements are non-quardratic residues, we expect to do the above twice before finding an output, so the total expected time is $O(|p|)$ times the cost of arithmetic operations.

The Monte Carlo version is to repeat the above $\log 1/\delta$ times, outputting an error message if all fail to produce non-residues.

Plentiful pigeons for irreducible polynomials: A polynomial is *irreducible* if it cannot be factored into a product of smaller degree polynomials. Give (and analyze) a probabilistic algorithm that given input a prime number p, generates a uniformly chosen random irreducible degree 2 polynomial over Z_p, i.e., a polynomial $ax^2 + bx + c$ that cannot be written as $(ex + f)(gx + h)$ over Z_p. You will probably need to be able to test quadratic residousity; see page 404 of MR. You need to give a bound on the expected running time of your algorithm.

This is similar to the previous problem. Let p be an odd prime. Remember that multiplication is invertible mod p: if $a \not\equiv_p 0$, there is an element a^{-1}

so that $aa^{-1} \equiv_p 1$. Because of this, we'll see that it is sufficient to generate *unitary* irreducible polynomials, i.e., where $a = 1$.

Moreover, if x^2+bx+c is factorable as $(ex+f)(gx+h)$, $g = e^{-1}$ and we can also factor it as $(x+gf)(x+eh)$. So we can reduce the problem to finding unitary polynomials that cannot be written as $(x+r)(x+s)$. Note that there are p^2 polynomials of the form x^2+bx+c, since b, c can be chosen independently. However, there are only $p + (p)(p-1)/2$ polynomials of the form $(x+r)(x+s)$ since for $r \neq s$, $(x+r)(x+s) = (x+s)(x+r)$. Thus, if we pick b, c at random, x^2+bx+c is factorable with probability at most $1/2(1+1/p) \leq 2/3$. Also, if $x^2+bx+c = (x+r)(x+s)$, then $x = -r$ is a root to x^2+bx+c. Since $x^2+bx+c = (x+b2^{-1})^2+c-b^2 4^{-1}$, then $0 = r^2+br+c = (r+b2^{-1})^2 - (b^2 4^{-1} - c)$, or $(2r+b)^2 = b^2-4c$. So b^2-4c is a quardratic residue if the polynomial is factorable. Conversely, if $q^2 = b^2-4ac$, $-(q-b)2^{-1}$ is a root of the polynomial, and $x+(q+b)2^{-1}$ is a factor. So to tell whether a unitary polynomial x^2+bx+c is irreducible, we can test whether b^2-4c is a quardratic residue.

Then an algorithm to generate random irreducible unitary polynomials is: pick $b, c \in Z_p$ uniformly. Compute the Legendre symbol for b^2-4c. If it is -1, return x^2+bx+c.

To get a random irreducible polynomial, we generate a random irreducible unitary polynomial x^2+bx+c and multiply through by a a random $a \in Z_p^*$, to get $ax^2+abx+ca$. This is factorable if and only if x^2+bx+c is.

Achieving the Average for Max Bisection: Consider the Max Bisection problem: Given a weighted, undirected graph $G = (V, E)$ with an even number of nodes, find a partition of V into two sets S, T with $|S| = |T|$ that maximizes the total weights of edges in $Cut(S, T)$. Give a *deterministic* algorithm that is guaranteed to find solutions to this problem within a factor of 2 of optimal. (Hint: this is very close to the Max Cut problem. The algorithms might also be very close.)

We first give a probability distribution on bisections, whose expected total weight is at least 1/2 the total weight. Arbitrarily label the nodes $a_1, b_1, ..a_{n/2}, b_{n/2}$. The probability distribution is: Independently for each $1 \leq i \leq n/2$, with probability 1/2 put $a_i \in S, b_i \in T$ and with probability 1/2 put $a_i \in T, b_i \in S$. Note that each edge between a_i and b_i is always in the cut, and each edge of any other form is in the cut with probability 1/2. So the expected cut weight is at least 1/2 the total weights.

We'll use the method of conditional expectations to make a deterministic algorithm with the same guarantee. Let S_i, T_i be the two sets after we have made the first i decisions above. Let E_i be the set of edges with at most one endpoint in $a_1, b_1, ..a_i, b_i$. Let M_i be the set of all edges between a_j and b_j for some $j > i$. and $a_{i+1}, b_{i+1}...a_n, b_n$. Then

$E[Cut(S,T)|S_i,T_i] = \sum_{e \in Cut(S_i,T_i) \cup M_i} w(e) + 1/2 \sum_{e \in E_i - M_i} w(e)$. Since E_i and M_i depend only on i, not on S_i, T_i, the only part of this formula that depends on our decisions is $\sum_{e \in Cut(S_i,T_i)} w(e)$. So by maximizing the partial cut, we also maximize the expected total cut. Fix S_{i-1}, T_{i-1}, and let $SA_i = S_{i-1} \cup \{a_i\}, TA_i = T_{i-1} \cup \{b_i\}, SB_i = S_{i-1} \cup \{b_i\}, TB_i = T_{i-1} \cup \{a_i\}$. Then $E[Cut(S,T)|S_{i-1},T_{i-1}] = 1/2 E[Cut(S,T)|SA_i,TA_i] + 1/2 E[Cut(S,T)|SB_i,TB_i]$, and $E[Cut(S,T)|SA_i,TA_i] - E[Cut(S,T)|SB_i,TB_i] = Cut(SA_i,TA_i) - Cut(SB_i,TB_i) = \sum_{t \in T_{i-1}} w(\{a_i,t\}) - w(\{b_i,t\}) - \sum_{s \in S_{i-1}} w(\{a_i,s\}) - w(\{b_i,s\})$. So one of the two placements does not decrease this expectation, and we can determine which placement has the larger expectation is $O(n)$ time.

Thus, if we compute each S_i, T_i from S_{i-1}, T_{i-1} by placing a_i, b_i in the way that maximizes the partial cut $Cut(S_i,T_i)$, as above, our algorithm takes $O(n^2)$ time total to compute a cut that achieves the initial expectation, which is at least 1/2 the total weights of all edges. Thus, this algorithm produces a cut that is at least half the weight of the maximum cut in the graph.

Sampling for Set Union: Give a probabilistic algorithm that is $O(n)$ expected time and uses as little as possible external memory to solve (with high probability) the following problem: Given two sets of exactly n elements each, S_1 and S_2, and a number $1 > \epsilon > 1/\log n$, approximate the size of $|S_1 \cup S_2|$ to within a factor of $1+\epsilon$, i.e., with probability say at least 90%, your algorithm should output a value s with $|S_1 \cup S_2| \leq s \leq (1+\epsilon)|S_1 \cup S_2|$. (Note: S_1 and S_2 are given as lists of elements with no repetitions in each list, but elements in S_1 may be equal to elements in S_2. S_1 and S_2 are not given in sorted order. Assume elements of S_1, S_2 are represented as k bit strings, and that comparisons, arithmetic operations and inner products mod 2 on k bit inputs are all constant time. In other words, k is the "word size" for the machine model. Elements of S_1, S_2 are representable as single words, and operations on single words are constant time. It follows that $k \geq logn$.)

Two observations to start: First, since $|S_1| = |S_2| = n$, $n \geq |S_1 \cup S_2| \leq 2n$, so being off by an *additive* factor of ϵn is roughly equivalent to being off by a *multiplicative* factor of $1+\epsilon$. Secondly, $|S_1 \cup S_2| = |S_1| + |S_2| - |S_1 \cap S_2|$, so to estimate the union, we can estimate the intersection.

Let $I = S_1 \cap S_2$. We could compute I exactly in $O(n \log n)$ time and memory by sorting and merging S_1 and S_2, and seeing which elements are duplicated. However, this would be more time and memory than we want to use. (One application might be two streams giving logs of requests for routers; we want to see roughly how many people are requesting both routers simultaneously, but we don't want to store all requests. Or we want to know how much information we'd get from consulting two databases of address lists vs. one, but we don't want to pay for the full lists until we see how redundant they are.) What we can do is pick random subsets $T_1 \subseteq S_1, T_2 \subseteq S_2$ and compute $I' = T_1 \cap T_2 \subseteq S_1 \cap S_2 = I$. How big is I' likely to be? Say we put each element in T_1, resp. T_2 with probability p independently. Then each element x in I is placed in T_1 with probability p and placed in T_2 with independent probability p, so is in I' with prob p^2. So the expected size of I' is $E[|I'|] = p^2|I|$. So using Baysian reasoning, we guess that the size of I is about $|I'|/p^2$. It takes linear time to construct T_1 and T_2 (with the added advantage that it can be done on-line), and then the total expected time taken to compute I' by merging and sorting T_1 and T_2 is on the order of $E[|T_1|+|T_2|\log(|T_1|+|T_2|)]) \leq E[|T_1|+|T_2|\log 2n] \leq \log 2nE[|T_1|+|T_2|] = pn\log 2n$. We also just need $O(|T_1|+|T_2|)$ additional memory.

So the smaller we can make p, the more efficient the algorithm will be. But if we make it too small, chances are that I'/p^2 will not reflect I. For example, if $|I| = n$ and $p < 1/\sqrt(n)$, there is a good chance that I' is empty, and so our guess will be 0. $|I'|$ is just the number of heads when $|I|$ coins of bias p^2 are flipped, so we can use standard techniques to bound the probability of error. Our answer is off by ϵn if and only if $||I'| - p^2|I|| = |I' - E[I']| \geq p^2\epsilon n$.

We could use Chernoff bounds from class and Chapter 4 of MR or Chebyschev bounds from Chapter 3 to bound this probability. I'm going to use the Chebyschev bounds, because later I'll show a different approach where Chernov bounds don't apply. A quick review from Chapter 3. The Variance of a random variable X is given by $V[X] = E[(X-E[X])^2] = E[X^2] - (E[X])^2$. Chebyschev's inequality says: $Prob[|X - E[X]| \geq t] \leq V[X]/t^2$. This is because, if $|X - E[X]| \geq t$, $(X - E[X])^2 \geq t^2 = (t^2/V[X])V[X] = (t^2/V[X])E[(X-E[X])^2]$, and the probability that a non-negative variable can be D times its expectation is at most $1/D$.

To compute $V[X]$ is easy when $X = \sum_i X_i$ where X_i are pairwise independent $0,1$ variables with probability p_i of being 1. Then $V[X] = E[(\sum X_i)^2] - (E[\sum_i X_i])^2 = E[\sum_{i,j} X_iX_j] - (\sum p_i)^2 = E[\sum_i X_i^2 + 2\sum_{i\neq j} X_iX_j] - (\sum_i p_i^2 + 2\sum_{i\neq j} p_ip_j) = \sum_i E[X_i^2] + 2\sum_{i\neq j} E[X_iX_j] - -(\sum_i p_i^2 + 2\sum_{i\neq j} p_ip_j) == \sum_i [p_i] + 2\sum_{i\neq j} p_ip_j - -(\sum_i p_i^2 + 2\sum_{i\neq j} p_ip_j) == \sum_i p_i(1-p_i) \leq \sum_i p_i = E[X]$. Note that what we used is that $X_i^2 = X_i$ because X_i is 0 or 1, and that $X_iX_j = 1$ if both its factors are 1, 0 otherwise, which happens with probability p_ip_j since the variables are independent.

So from our case, letting $I_x = 1$ if $x \in I'$, 0 otherwise, $|I'| = \sum_{x\in I} I_x$, and I_x are independent random variables of probability p^2. Thus, $V[|I'|] \leq E[|I'|] \leq p^2 n$. So probability $||I'| - E[I']| \geq \epsilon p^2 n \leq (p^2 n)/(\epsilon p^2 n)^2 = 1/(\epsilon^2 p^2 n)$. This is less than say, $1/25$, for $p = 5/(\epsilon n^{1/2})$. For this choice of p, our algorithm runs in time $O(n + pn\log n = n + n^{1/2}\log n/\epsilon) = O(n)$

for $\epsilon > \log n/n^{1/2}$. It uses expected external memory $O(pn) = O(n^{1/2}/\epsilon)$.

Note that this algorithm has an essential bottleneck if we have less than $\sqrt(n)$ external memory available, which might make it hard to scale for network applications. This is an artifact of the independence between x being put in T_1 and x being put in T_2. We can make these events instead identical while keeping x being in T_1 independent from y being in T_1. The way we do this is to base our random choice on a hash function of the value. We need our hash function to be randomly chosen, to preserve independence. The idea is: pick a random hash function h mapping k bit strings to $1/p$ buckets. Pick a bucket at random. Find the set of elements of S_1 and S_2 that hash to the same bucket; call them B_1, B_2. Note that elements that hash to other buckets can be forgotten. Compute $J = B_1 \cap B_2$, and estimate $|I| = |J|/p$.

What needs to be filled in is how we pick a hash function h. We'll use the following: Let $t = \log 1/p$. Pick random k bit strings $r_1, ...r_t$ and let $h(x) =< r_1, x >, < r_2, x > ... < r_t, x >$. We then pick t bits $b_1, ..b_t$. We let B_1 be all $x \in S_1$ so that $h(x) = b_1...b_t$. Note that, although it takes t time to compute $h(x)$, we can compute B_1 in expected linear time. That is because, once $< r_i, x > \neq b_i$, we know $x \notin B_1$. Since $b_1, ..b_t$ are chosen randomly, for any fixed x, the expected number of inner products before we conclude $x \notin B_1$ is 2. So by additivity of expectations, we expect $2n$ inner products total to compute B_1 and similarly for B_2. Also, note that $h(x) = h(y) = \vec{b}$ if and only if $h(x \oplus y) = 0$ and $\vec{b} = h(x)$. As we saw in class $< r, z >= 0$ with probability exactly $1/2$ for any non-zero vector z. So the probability that $h(x \oplus y) = 0$ for any $x \neq y$ is $1/2^t = p$, and then $\vec{b}$ is independently chosen, so the combined probability is p^2. So letting $J_x = 1$ if $x \in I$ has $h(x) = \vec{b}$, we see that the J_x are pairwise independent Boolean variables of probability p, and $|J| = \sum_x J_x$. Thus, $V[|J|] \leq E[|J|] \leq pn$, and $Prob[||J| - p|I| \geq p\epsilon n] \leq pn/(p\epsilon n)^2 = 1/(p\epsilon^2 n)$. Thus, to make this at most .1, we can pick $p = 10/(\epsilon^2 n)$. The expected time is $O(n + pn \log n) = O(n + \log n/\epsilon^2) = O(n)$ for $\epsilon > 1/\sqrt{n}$, and the external memory is $O(\log n + pn) = O(\log n + 1/\epsilon^2) = O(\log^2 n)$ for the specified range for ϵ.