CS 170 Homework 13

Due **Friday 12/1/2023, at 10:00 pm (grace period until 11:59pm)**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write "none".

Solution: None in particular. Went to office hours for help on problems 2 and 3 mainly.

2 Local Search for Max Cut

Sometimes, local search algorithms can give good approximations to NP-hard problems. Recall that in the Max-Cut problem, we have an unweighted graph $G = (V, E)$ and we want to find a cut (S, T) that maximizes the number of edges "crossing" the cut (i.e. with one endpoint in each of S, T). Consider the following local search algorithm:

1. Start with any cut (e.g. $(S, T) = (V, \emptyset)$).
2. While there is some vertex $v \in S$ such that more edges cross $(S \setminus \{v\}, T \cup \{v\})$ than (S, T) (or some $v \in T$ such that more edges cross $(S \cup \{v\}, T \setminus \{v\})$ than (S, T)), move v to the other side of the cut.

Now, let us prove a couple of guarantees that this algorithm achieves.

(a) Give an upper bound on the number of iterations this algorithm can run for (i.e. the total number of times we move a vertex).

Solution: When we move a vertex over, we increase the number of edges that cross the cut. In the worst case, the max-cut cuts across all the edges in G meaning that it is possible for the algorithm to run for all the edges, or in total we move $|E|$ edges.

(b) Show that when this algorithm terminates, it finds a cut where at least half the edges in the graph cross the cut.

Hint: when we move v from S to T, v must have more neighbors in S than T. What does this observation suggest about the neighbors of each vertex once the algorithm terminates? Then, what can we say about the number of edges crossing the cut vs. the number of edges within each side of the cut?

Solution: We follow the hint, which says that whenever we have a vertex to move, it's because v has more neighbors in S than T. Once the algorithm terminates, this means that we can no longer find a vertex v that satisfies this condition. Therefore, it must be the case that for every vertex in S (and also in T), they each have more edges crossing the cut than edges that don't cross the cut. Since this is true for all vertices, then we can conclude that there are more edges that cross the cut than edges that don't cross the cut. This condition is only achieved when the number of edges that cross the cut is more than half the total number of edges, as desired.

3 Randomization for Approximation

Oftentimes, extremely simple randomized algorithms can achieve reasonably good approximation factors.

(a) Consider Max 3-SAT: given an instance with m clauses each containing exactly 3 distinct literals, find the assignment that satisfies as many of them as possible. Come up with a simple randomized algorithm that will achieve an approximation factor of $\frac{7}{8}$ in expectation. That is, if the optimal solution satisfies c clauses, your algorithm should produce an assignment that satisfies at least $\frac{7c}{8}$ clauses in expectation.

Hint: use linearity of expectation!

Solution: Our randomized algorithm is as follows: for every literal x_i, randomly assign its truth value to be either true or false (with probability $\frac{1}{2}$ for each).

The reason this achieves our desired factor of $\frac{7}{8}$ is because for any given clause of 3 literals, the only way that clause is not satisfied is if all three literals evaluates to false, which occurs $\frac{1}{8}$ of the time. This works for any clause, so in expectation we expect $\frac{7m}{8}$ of the total number of clauses to be satisfied. Further, since the optimal solution satisfies c clauses and $c \leq m$, this gives us that in expectation, we have at least $\frac{7c}{8}$ clauses satisfied.

(b) Given a Max 3-SAT instance I, let OPT_I denote the maximum fraction of clauses in I satisfied by any variable assignment. What is the smallest value of OPT_I over all instances I? In other words, what is $\min_I \mathsf{OPT}_I$?

Hint: use part (a), and note that a random variable must sometimes be at least its mean.

Solution: From part (a), we know that our randomized algorithm achieves in expectation at least $\frac{7m}{8}$ satisfied clauses. Now, let X be the random variable that counts the number of clauses that are satisfied using our randomized algorithm. We know from part (a) that in expectation we have $\frac{7m}{8}$ clauses satisfied, and since OPT_I performs better than our randomized algorithm (by definition of being the optimal), combined with the fact that X must sometimes be at least $\frac{7m}{8}$, then we get that the lower bound on the optimal number of satisfied clauses must be at least as good as our randomized algorithm, or $c \geq \frac{7m}{8}$.

(c) **(Extra Credit)** Derandomize your algorithm from part (a), i.e give a deterministic algorithm that achieves the same approximation factor. Justify the correctness of your algorithm.

Disclaimer: this subpart is particularly difficult, so please only attempt this after completing the rest of the homework and if you want extra practice.

4 Reservoir Variations

(a) Design an algorithm that takes in a stream $x_1, \dots, x_M$ of M integers in $[n] := \{1, \dots, n\}$ and at any time t can output a uniformly random element in $x_1, \dots, x_t$. Your algorithm may use at most polynomial in $\log n$ and $\log M$ space. Prove the correctness and analyze the space complexity of your algorithm. Your algorithm may only take a single pass of the stream.

Hint: for the proof of correctness, note that $\frac{1}{t} = 1 \cdot \frac{1}{2} \cdot \frac{2}{3} \cdot \frac{3}{4} \cdots \frac{t-1}{t}$.

Solution: Algorithm Description: Have a value in memory that stores one element in the stream. Then, for every element x_t in the stream we let the probability that the current value in memory gets replaced to be $\frac{1}{t}$. Then, once we've seen M integers, we output the stored value.

Proof of Correctness: To do this, we calculate the probability that our algorithm outputs element x_t. Suppose x_t is taken in at some time t. Then, at any time $T > t$, the probability that we output x_t is:

$$P(\text{output } x_t) = \frac{1}{t}\left(1 - \frac{1}{t+1}\right) \cdots \left(1 - \frac{1}{T}\right) = \frac{1}{t} \cdot \frac{t}{t+1} \cdots \frac{T-1}{T} = \frac{1}{T}$$

So this means that at any time t, the probability of outputting any element is given by $\frac{1}{t}$, as desired.

As for the space complexity, each number x_i takes at most $\log n$ space, and to get M integers, we can just keep a counter that ticks up to M and terminates the algorithm that that point, which takes $\log M$ bits in total. Therefore, the total space is:

$$\log n + \log M = \log nM$$

(b) For a stream $S = x_1, \dots, x_{2n}$ of $2n$ integers in $[n]$, we call $j \in [n]$ a *duplicate element* if it occurs more than once.

Prove that S must contain a duplicate element, and design an algorithm that takes in S as input and with probability at least $1 - \frac{1}{n}$ outputs a duplicate element. Your algorithm may use at most polynomial in $\log n$ space. Prove the correctness and analyze the space complexity of your algorithm. Your algorithm may only take a single pass of the stream.

Hint: Use $\log n$ *copies of the algorithm from part a to keep track of a random subset of the elements seen so far. For the proof of correctness, try to upper bound the probability that our algorithm fails to output a duplicate; also, note that there are at most* n *indices* t *such that element* x_t *never occurs after index* t.

Solution: By the pigeonhole principle, since $x_i \in \{1, \dots, n\}$ and we have $2n$ integers, we must have duplicates.

Algorithm Description: Have $\log n$ copies of the algorithm from part (a), and do the following at every time step:

- Initialize $\log n$ reservoirs of memory, initially set to empty.
- Read in the integer x_t, and compare it to the values currently stored in the reservoir. If x_t matches one of the values in the reservoir, output that value and terminate the algorithm.
- If x_t doesn't match any value in the reservoirs, then for every reservoir, replace that value with x_t with probability $\frac{1}{t}$.

- Keep going until all integers in S have passed.

Proof of Correctness: Now, to get the probability guarantee, let's analyze the probability of failure of our algorithm. Consider streaming through $2n$ elements using only a single reservoir. In this case, a failure occurs when at some time t, we take in x_t, and x_t remains in our reservoir but we never see x_t again. The probability that x_t remains in the reservoir is $\frac{1}{2n}$ by the same logic as in part (a), and since there are at most n indices where we never see x_t again, then the probability of failure (i.e. we never see x_t again) is $\frac{n}{2n} = \frac{1}{2}$.

Now, if we have $\log n$ reservoirs and update each element according to part (a), then each reservoir has a probability $\frac{1}{2}$ of failing. Since each reservoir is independent, then the total probability our algorithm fails is

$$\left(\frac{1}{2}\right)^{\log_2 n} = \frac{1}{n}$$

Since this is the probability of failure, then this means that our algorithm succeeds with probability $1 - \frac{1}{n}$, as desired.

As for the space complexity, we have $\log n$ values to store, each of which takes $\log n$ space to store. Therefore, in total we take $(\log n)^2$ space.

5 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.

Solution: We can do this by keeping track of the current sum using a single bit: 1 if the sum is currently odd and 0 if the sum is currently even. Then, for every x_i that comes in, we can determine its parity by looking at the last digit, and update our running sum accordingly:

- $x_i = 0$: Leave the running sum untouched.
- $x_i = 1$: Flip the running sum from $1 \to 0$ or $0 \to 1$.

The proof of correctness is pretty trivial: we just use the property that adding an even number to some number k, doesn't change the parity of k, and adding an odd number to k, does change its parity, hence the bit flip.

(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.

Solution: We employ the same idea as part (a), by our running sum can be expressed as some integer $\{0, \ldots, N-1\}$, and for every incoming x_i, we can compute its value mod N and add it to our running sum modulo N. This way, we are only storing one value at a time which has a size of at most $\log(N-1) = \log N$, which satisfies the space complexity requirement.

The proof of correctness is the same as the previous part: a number is divisible by N if and only if that number is $0 \mod N$, and by keeping a running sum we keep information about whether the current sum is divisible by N using $\log N$ bits.

(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.

Hint: suppose the first 3 integers of the stream are $7, 10, 6$. *How can you check whether* $N = 3$ *divides their product? Why is it not necessary to compute the entire product* $7 \times 10 \times 6 = 420$ *to determine divisibility?*

Solution: We can get away with using only a single bit b, which stores 1 if the product is divisible, and 0 if it isn't. The algorithm is as follows: begin with $b = 0$. Then, for every incoming integer, we check whether that number is divisible by N. If not, then we keep $b = 0$ and keep going. If yes, then we set $b = 1$, at which point any further computation is useless, since the product of the integers seen will always be divisible by N.

The proof of correctness is also fairly simple here: because N is prime, then the product of the integers seen so far is only divisible by N if one of the numbers has N as a factor. Then, once this condition is met, and b is set to 1 after some x_i, then we no longer need to keep checking, since the product of all integers seen will always include x_i, which has N as a factor.

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

Hint: keep track of r values (i.e. one for each prime factor of N) throughout your algorithm.

Solution: We use the hint, and keep track of one value for each prime factor of N, so in memory we have r bits $b_1, \dots, b_r$, where $b_i = k_i$ initially. Then, when the next x_j comes in, for every nonzero b_i, we find the largest power of p_i that divides x_j, and subtract that power off of b_i (while enforcing that $b_i \geq 0$). Then, the stream is going to be divisible by N then all $b_i = 0$. At this point, just like the previous part, we can stop here in terms of computation.

This is very similar to the approach in part (c) except we're tracking multiple primes, but it's the same thing in principle. Here, we have the added step of trying to compute the largest power of p_i that fits within each incoming x_j, and this is solely due to the fact that k_i could be larger than 1. Then, once all $b_i = 0$, then we know that N is now a factor of the product, since all prime factors are present. From this point forward, the product will always be divisible by N, as desired.

As for the space complexity, we know that it takes $\log(k_i + 1)$ bits to describe k_i, so the total space complexity is:

$$\log(k_1 + 1) + \log(k_2 + 1) + \cdots + \log(k_r + 1) = \log\left[\prod_i (k_i + 1)\right]$$