Problem Set 4

This problem set is all about randomness – randomized algorithms, randomized data structures, random variables, etc. By the time you're done with this problem set, we hope that you have a much more nuanced understanding of randomization and its role in algorithm design.

Please be sure to write your answers different problems on separate pieces of paper to make it easier for us to grade. Also, **please put your name on each page of your assignment**.

As always, please feel free to drop by office hours or send us emails if you have any questions. We'd be happy to help out.

This problem set has 36 possible points. It is weighted at 12% of your total grade.

Good luck, and have fun!

Due Monday, July 29 at 2:15 PM

Problem One: Insertion Sort Revisited (4 Points)

In our very first lecture, we saw that the runtime of insertion sort on an input array of length n was $\Theta(n + I)$, where I is the number of inversions in the array (recall that an *inversion* is a pair of elements $A[i]$ and $A[j]$ where $A[i] > A[j]$ but $i < j$). At the end of lecture, we sketched out a proof that if insertion sort is run over a uniformly-random permutation of 1, 2, …, n, its expected runtime is $\Theta(n^2)$ because $E[I] = \Theta(n^2)$.

Formally prove that $E[I] = \Theta(n^2)$ by writing I as the sum of appropriate indicator random variables and using linearity of expectation.

Problem Two: Weighted Min-Cut (8 Points)

This problem explores the *weighted min cut* problem. In this variation of the min cut problem, all edges in the graph are assigned a nonnegative weight w_e. The goal is to find a cut in the graph that minimizes the total weight of all edges crossing that cut.

Although it may seem like this problem is harder than finding a min cut, it turns out that Karger's algorithm (and the Karger-Stein algorithm) can easily be adjusted to solve the weighted min cut problem. In fact, these algorithms were originally designed to find weighted min cuts; the unweighted versions we discussed in lecture are just a special case where all weights are 1.

Here is the only modification necessary to Karger's algorithm to handle weighted graphs: when choosing an edge to contract, instead of choosing the edge uniformly at random, choose each edge e with probability

$$\frac{w_e}{\sum_{f \in E} w_f}$$

That is, the probability that we choose e is the fraction of the total weight associated with edge e.

As in our analysis of the unweighted version of Karger's algorithm, consider any min-weight cut C with total weight k. Let $\mathcal{E}$ be the event that the weighted version of Karger's algorithm never contracts an edge from C, and let the event $\mathcal{E}_i$ be the event that it does not contract any edge from C during iteration i. As before, we have that

$$P(\mathcal{E}) = P(\mathcal{E}_{n-2} \mid \mathcal{E}_{n-3}, \mathcal{E}_{n-4}, \ldots, \mathcal{E}_2, \mathcal{E}_1)\, P(\mathcal{E}_{n-3} \mid \mathcal{E}_{n-4}, \ldots, \mathcal{E}_1) \ldots P(\mathcal{E}_2 \mid \mathcal{E}_1)\, P(\mathcal{E}_1)$$

Your job is to determine this probability.

i. Prove that $P(\mathcal{E}_1) \geq (n - 2) / n$.

ii. Prove that $P(\mathcal{E}_i \mid \mathcal{E}_{i-1}, \mathcal{E}_{i-2}, \ldots, \mathcal{E}_2, \mathcal{E}_1) \geq (n - i - 1) / (n - i + 1)$

Because of your results from (i) and (ii), we can use the same analysis of Karger's algorithm to conclude that $P(\mathcal{E}) = 2 / n(n - 1)$, meaning that the weighted version of Karger's algorithm has the exact same success probability as the unweighted version. Since the probabilities work out the same way, this also means that the Karger-Stein algorithm can also find weighted min cuts with probability $\Omega(1 / \log n)$.

In other words, it's not much harder to find a weighted min cut than an unweighted min cut!

Problem Three: Set Partitioning (8 Points)

The *set partitioning problem* is an **NP**-hard problem. Given a set S of real numbers, the goal is to split S into two subsets T and $S - T$ such that the quantity

$$\sum_{x \in T} x - \sum_{x \in S - T} x$$

is as close to 0 as possible. We'll call the above quantity the *cost* of a partition.

i. Design a polynomial-time, randomized algorithm that returns a partition with expected cost 0. Then:
   - Describe your algorithm.
   - Prove your algorithm returns a partition that, on expectation, has cost 0.
   - Prove that your algorithm runs in polynomial time.

ii. Give an example of a set S has no partition of cost 0. Briefly explain how this is possible, given that you just designed an algorithm that returns a partition with expected cost 0.

Problem Four: The Count-Min Sketch (15 Points)

A *streaming algorithm* is a type of algorithm where the input to the algorithm is fed in one element at a time. As the algorithm is running, it can periodically be queried to learn properties of the data that have been received so far. At a high level, a streaming algorithm works as follows:

- The algorithm waits for data to arrive or queries about that data to be made.
- When data arrives, the algorithm performs work to process that data.
- When queries about the data are made, the algorithm performs work to answer that query.

One important operation on data streams is to report how many times a particular piece of data has appeared in the stream (this is called *frequency estimation*.) For example, Google gets billions of searches each day and periodically might want to know how many times a particular search has been made. One possible algorithm for this would be to store a hash table associating each search with its frequency. This is simple but space-inefficient; if there are n distinct searches (which, for Google, would be a very large number), this requires $\Omega(n)$ storage space.

The *Count-Min sketch* (or *CM sketch*) is a frequency estimation algorithm that uses $o(n)$ storage space. To save space, the CM sketch only gives probabilistic guarantees on the accuracy of its estimates: with high probability, the CM sketch will return a frequency estimate close to the true frequency. As you'll see, the CM sketch is simple, fast, space-efficient, and accurate. It's also fairly recent (developed in 2003) and uses universal hash functions in a clever way.

This problem is broken down into six parts:

- Part (i) asks you to trace through a simplified version of the algorithm.
- Parts (ii), (iii), (iv), and (v) ask you to determine quality of the estimates produced by a simplified version of the CM sketch.
- Part (vi) asks you to analyze the quality of the estimates given by the full CM sketch.

Note: If you get stuck proving any of the results, feel free to assume the result is true while working on the later parts of the problem. The problem starts on the next page.

Suppose that you have a data stream composed of elements drawn from the universe $U = \{x_1, x_2, \ldots, x_n\}$. Your goal is to be able to estimate, at any point in time, how many times some element x_k has appeared in the data stream so far. To do this, you create an array A of some length w, where each entry is an integer initialized to 0. (We'll assume the array is zero-indexed). You have available a universal family of hash functions $\mathscr{H}$ that hash from U to the set $\{0, 1, 2, \ldots, w - 1\}$. At the time you create array A, you select, uniformly at random, a hash function $h \in \mathscr{H}$.

On receiving data x, you compute $h(x)$ and increment the counter at that location. In pseudocode:

```
procedure onReceive(x):
    let bucket = h(x)
    A[bucket] = A[bucket] + 1
```

i. Suppose that $w = 5$ and queries are integers. If the hash function is $h(x) = x \bmod w$, what will the array values be after you receive these numbers from the data stream?

$$2, 7, 1, 8, 2, 8, 1, 8, 2, 8, 4, 5, 9, 0, 4, 5$$

Given this data structure, you can estimate the number of times that you've seen the value x by computing $h(x)$ and returning the value of the counter at that array location. In pseudocode:

```
procedure estimateFrequency(x):
    let bucket = h(x)
    return A[bucket]
```

This is not guaranteed to produce an accurate result. As you can see from your array from part (i), if you estimate the frequency of 1 or 8, you will get the right answer. However, estimating the frequency of 0, 2, 4, 5, 7, or 9 will produce the wrong answer. This happens because different values can hash to the same position, so each counter might count the frequencies of several elements.

However, because h is drawn from a family of universal hash functions, the value produced as the estimate for f_i will, with reasonable probability, be close to the true value. Suppose that at some point in time the values $x_1, x_2, \ldots, x_n$ have appeared with frequencies $f_1, f_2, \ldots, f_n$. Note that the f variables are not random variables; they're the true frequencies of the data. For any pair of distinct x_i, x_j, define C_{ij} to be the indicator random variable

$$C_{ij} = \begin{cases} 1 & \text{if } h(x_i) = h(x_j) \\ 0 & \text{otherwise} \end{cases}$$

In other words, C_{ij} is 1 if x_i and x_j hash to the same slot in the array and is 0 otherwise.

ii. For any x_i, define V_i to be a random variable equal to the estimated frequency of the element x_i. That is, $V_i = A[h(x_i)]$. Express V_i in terms of the x, f, and C variables.

iii. Prove that $V_i \geq f_i$. That is, the algorithm never underestimates the frequency of x_i.

Let N be the total number of data points received in the stream so far.

$$N = \sum_{i=1}^{n} f_i$$

iv. Show that $E[V_i] \leq f_i + N / w$. In other words, the expected value of the estimated frequency of x_i is at most f_i (the true frequency of x_i) plus the total number of data points received divided by the size of the array. *(Hint: Use the fact that h is drawn from a universal family of hash functions.)*

(Continued on the next page)

Your result from (iv) shows that we can adjust the quality of frequency estimates by changing the size of the array. As w gets larger, the expected value gets closer to f_i, the true frequency.

Next, you'll show that the probability of getting a very "noisy" answer is at most a constant:

v. Prove that $P(V_i \geq f_i + eN / w) \leq 1 / e$, where e is the base of the natural logarithm. *(Hint: Use Markov's inequality. You might find it easier to work with a new variable $V'_i = V_i - f_i$.)*

The full CM sketch is formed by having multiple copies of the above data structure running in parallel. Suppose that instead of creating just one array of length w, we create d different arrays $A_1, A_2, \ldots A_d$, each of length w. We associate with each array A_k a hash function $h_k \in \mathscr{H}$ chosen uniformly at random. Whenever a new data point x is received, we go to each array A_k, compute $h_k(x)$, then increment the counter at that position in A_k. In pseudocode:

```
procedure onReceive(x):
    for k = 1 to d:
        let bucket = hₖ(x)
        Aₖ[bucket] = Aₖ[bucket] + 1
```

Because we have multiple arrays, we will get back multiple different estimates for the frequency of an element x_i. To resolve this, we can just get the estimates from each of the arrays, then take the minimum. Intuitively, this picks the estimate with the lowest "noise." In pseudocode:

```
procedure estimateFrequency(x):
    let best = ∞
    for k = 1 to d:
        let bucket = hₖ(x)
        best = min(best, Aₖ[bucket])
    return best
```

For any element x_i and any array A_k, define V_{ik} to be the estimate of the frequency of x_i from array A_k. The final estimate U_i of the frequency of x_i is then given by

$$U_i = \min\{ V_{i1}, V_{i2}, \ldots, V_{id} \}$$

vi. Prove that $P(U_i \geq f_i + eN / w) \leq (1 / e)^d$. *(Hint: Under what circumstances will the minimum of $V_{i1}, V_{i2}, \ldots, V_{id}$ be greater than $f_i + eN / w$?)*

Let's suppose that we pick two parameters ε and δ, both of which are in the range (0, 1). We'll then choose $w = \lceil e / \varepsilon \rceil$ and $d = \lceil \ln (1 / \delta) \rceil$. Since you proved

$$P(U_i \geq f_i + eN / w) \leq (1 / e)^d$$

This means that

$$P(U_i \geq f_i + \varepsilon N) \leq \delta$$

You can think of ε as an "accuracy" parameter and δ as a "confidence" parameter. By lowering ε, we increase precision; by lowering δ, we increase the chance our answer is correct.

The CM sketch has excellent performance. Upon receiving new data, the CM sketch does only $\Theta(\ln (1 / \delta))$ work computing $\lceil \ln (1 / \delta) \rceil$ hash functions and incrementing $\lceil \ln (1 / \delta) \rceil$ counters. Performing a query similarly takes time $\Theta(\ln (1 / \delta))$. Moreover, the CM sketch does so without using much space; it stores $\lceil \ln (1 / \delta) \rceil$ arrays of size $\lceil e / \varepsilon \rceil$, so its space usage is $\Theta(\varepsilon^{-1} \ln (1 / \delta))$. This allows the performance to be tuned to optimize for precision, confidence, or space.

Problem Five: Course Feedback (1 Point)

We want this course to be as good as it can be, and we'd appreciate your feedback on how we're doing. For a free point, please answer the following questions. We'll give you full credit no matter what you write, as long as you write something.

i. How hard did you find this problem set? How long did it take you to finish? Does that seem unreasonably difficult or time-consuming for a five-unit course?

ii. Did you attend office hours? If so, did you find them useful?

iii. Did you read through the textbook? If so, did you find it useful?

iv. How is the pace of this course so far? Too slow? Too fast? Just right?

v. Is there anything in particular we could do better? Is there anything in particular that you think we're doing well?