CS5234: Combinatorial and Graph Algorithms

Problem Set 1 (Solutions)

Due: August 18th, 6:30pm

Instructions. The problem set begins with a few exercises that you do not need to hand in. They are primarily for review (in this case of material covered in previous modules). The problems that follow are both related to processing streams of data (though the connect to streaming may not be immediately obvious for the first problem).

- Start each problem on a separate page.

- Make sure your name is on each sheet of paper (and legible).

- Staple the pages together, and hand it in before class starts, or submit it on IVLE in the workbin. Alternatively, if you submit it late, you can put it in the envelope next to my office door (and send me an e-mail).

Remember, that when a question asks for an algorithm, you should:

- First, give an overview of your answer. Think of this as the executive summary.

- Second, describe your algorithm in English, giving pseudocode if helpful.

- Third, give an example showing how your algorithm works. Draw a picture.

You may then give a proof of correctness, or explanation, of why your algorithm is correct, an analysis of the running time, and/or an analysis of the approximation ratio, depending on what the question is asking for.

Advice. Start the problem set early—some questions take time. Come talk to me about the questions. (Different students have different questions. Some have questions about how to write a good proof. Others need pointers of designing an algorithm. Still others want to understand the material from lecture more deeply before applying it to the problem sets.) I'm here for you to talk to.

Collaboration Policy. The submitted solution must be your own unique work. You may discuss your high-level approach and strategy with others, but you must then: (i) destroy any notes; (ii) spend 30 minutes on facebook or some other non-technical activity; (iii) write up the solution on your own; (iv) list all your collaborators. Similarly, you may use the internet to learn basic material, but do not search for answers to the problem set questions. You may not use any solutions that you find elsewhere, e.g. on the internet. Any similarity to other students' submissions will be treated as cheating.

Exercises (and Review) (*Do not submit.*)

Exercise 1.   Assume h is a hash function mapping $[1, n]$ to $[1, m]$. Assume that h is a perfectly random uniform map.

Ex. 0.a.   Let a, b, c be three different integers. What is the probability that $h(a) = h(b) = h(c)$?

Ex. 0.b.   Assume that you use h to build a hash table of size m with chaining. Now you add $17m$ items to the hash table. What is the expected number of items in each chain?

Ex. 0.c.   Assume that you use h to build a hash table of size m with chaining. Now you add m items to the hash table. Prove that all the chains have $O(\log m)$ elements (at most) with probability at least $1 - 1/m^2$.

Exercise 2.   In class this week we will solve the problem of graph connectivity where we are given a stream of edge additions and edge deletions. Imagine, instead, that we only have edge additions. That is, the stream $S = \langle e_1, e_2, e_3, \ldots \rangle$ consists of a collection of edges provided in an arbitrary order. (Each edge may appear more than once, but edges are never removed.) Once the stream is done, you must state how many connected components there are in the graph.

   Assume the graph G has m edges and n nodes. Give an efficient algorithm for answering this question that uses only $O(n \log n)$ space. Try to ensure that processing each edge in the stream and answering connectivity queries are fast, e.g., significantly sublinear time.

Standard Problems (to be submitted)

Problem 0.   Permuting an array.

Imagine you are given an array $A[1 \, .. \, n]$. Here is an algorithm for permuting the array in a random fashion:

```
for (i = 1 to n) do:
    Choose a random number j in [1..i].
    Swap A[i] and A[j] (leaving the array unchanged if i=j).
```

Prove that when this procedure completes, the array is a random permutation. For example, for a given item x, the probability that x ends up in slot i should be $1/n$ (for all slots i). (Hint: use induction from 1 to n.)

Solution:
Overview.   We will prove this by induction on i as it proceeds from 1 to n. Specifically, we will prove that after iteration i of the loop, the prefix $[1..i]$ of the array is a uniformly random permutation. To do this, we will show that, for each element $1 \leq j \leq i$, for each slot $1 \leq k \leq i$, the probability of element j being in a slot k is $1/i$. This property implies that the permutation is uniform.
Induction argument.   The base case where $i = 1$ is clearly true, as the prefix contains only one element. Assume that the claim holds after iteration $i - 1$.
Consider the ith iteration. First, look at the item that begins the iteration in slot i. This item is moved to a slot k chosen uniformly at random from the prefix $[1..i]$ and so it has a probability of $1/i$ of being in any slot in the prefix of the array. Thus for this one item, the property holds.
Now consider some other element j, e.g., an element that begins the iteration in the prefix $[1..i-1]$. By our induction hypothesis, when the iteration begins, this item is uniformly distributed in the prefix $[1..i-1]$, i.e., the probability that it is in some slot $k < i$ is $1/(i-1)$. During iteration i, this element x has a $1/i$ probability of being moved to slot i. Otherwise, with probability $(1 - 1/i)$, it remains in the same slot in which it began the iteration.
Let us calculate the probability that item j ends the iteration in a given slot $k \in [1..i]$. If $k = i$, it has a $1/i$ probability of being in that slot, as desired. Otherwise, it has a $(1/(i-1))$ probability of beginning the iteration in slot k and a $(1 - 1/i)$ probability of remaining slot k. Hence the total probability of being in slot k is $(1/(i-1))(1 - 1/i) = (1/(i-1))((i-1)/i) = 1/i$. Hence item j has a uniform probability of being at any position in the prefix $[1..i]$.
Notes.   An interesting note is that the same idea can be used to choose a random element from a stream. Imagine you are seeing a stream of elements $x_1, x_2, \ldots$, and you do not know how many elements are in the stream. Your goal is to choose a random element from the stream.
The basic algorithm is the same: When you see item x_1, you keep it. When you see item x_i, you keep item x_i with probability $1/i$ (throwing out your previous saved item). Otherwise, with probability $(1 - 1/i)$ you keep your old item.
This is known as "reservoir sampling" and by the same analysis above, it gives you a uniformly random sample of your stream.

Problem 1.   Counting the items in a stream.

Today you are given a stream $S = \langle s_1, s_2, \ldots, s_N \rangle$ in which each s_i is an integer (where every integer is less than some maximum value M). The goal is to count (approximately) how many times each integer appears in the stream. For example, if you observe stream:

$$S = \langle 5, 7, 5, 5, 9, 5, 4, 9, 5, 5, 7 \rangle$$

then once the stream is complete, you should be able to respond to the queries:

$$
\begin{aligned}
query(5) &= 6 \\
query(7) &= 2 \\
query(9) &= 2 \\
query(4) &= 1
\end{aligned}
$$

Assume the stream consists of N elements in total, and let $n(x)$ be the number of times that the integer x appears in the stream. Then we want to build a streaming algorithm that guarantees the following:

For every integer x, with probability at least $(1 - \epsilon)$: $n(x) \le query(x) \le n(x) + \delta N$.
That is, each query has at worst an additive error of δN (with probability at least $1 - \epsilon$).

To solve this problem, we are going to use a very similar idea as we saw in class for building the s-sparse sampler, except that we will replace each of the 1-samplers with a counter:

- We will use A hash functions: choose $h_1, h_2, \ldots, h_A$ to be hash functions mapping integers to $[1, B]$ uniformly at random. Assume each hash function is perfect, and that the hash functions are independent.

- We will also use AB counters: let $C(i, j)$ be a counter where $i \in [1, A]$ and $j \in [1, B]$. (See Figure 1.)

- When we see integer x in the stream, then for all $i \in [1, A]$, we will increment the counter $C(i, h_i(x))$. (When we increment a counter, we simply add one to the value of the counter.)

- When we perform a $query(x)$ operation, we return the minimum counter value for all the counters in the set $\{C(i, h_i(x)) | i \in [1, A]\}$.

Our goal in this problem is to show that using this information, when the stream is over we can derive a fairly good estimate for the number of times an element appeared in the stream.

Problem 1.a. Explain why $query(x) \ge n(x)$.

Solution: Every time the value x appears in the stream, we increment all the counters in the set $\{C(i, h_i(x)) | i \in [1, A]\}$. Thus all of these counters must be at least $n(x)$.

Problem 1.b. Fix some $i \in [1, A]$. Define $error(x) = C(i, h_i(x)) - n(x)$. Notice that $error(x)$ is the amount by which the counter deviates from the correct answer. Prove that $\Pr[error(x) \ge 2N/B] \le 1/2$. (Hint: use Markov's Inequality.)
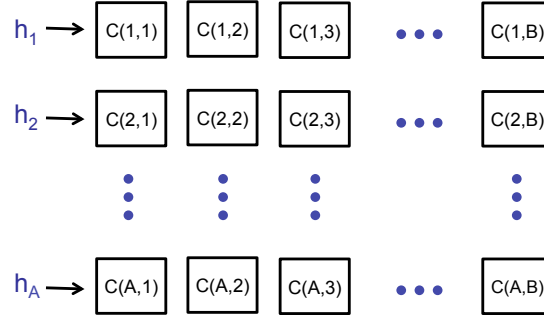
Figure 1: We use AB counters. Each of the A hash functions maps each of the incoming elements in the stream to one of the B counters in its row.

Solution: First, we calculate the expected error, i.e., $\mathrm{E}\left[error(x)\right] = \mathrm{E}\left[C(i, h_i(x)) - n(x)\right]$. (Recall, we have fixed i.)

Claim 1 $\mathrm{E}\left[error(x)\right] \le N/B$

Proof Let Y_j be an indicator random variable where $Y_j = 1$ if $h_i(s_j) = h_i(x)$. In this case, item j "collides" with item x in that they both hash to the same value. Observe that $\mathrm{E}\left[Y_j\right] = 1/B$, since the range of the hash functions is $[1, B]$.

Now, we see that $\mathrm{E}\left[C(i, h_i(x)) - n(x)\right] = \sum_{j:s_j \ne x} \mathrm{E}\left[Y_j\right]$, by linearity of expectation. (We have subtracted $n(x)$ by excluding the $n(x)$ variables Y_j that refer to the value x. If instead we had calculated $\sum_j \mathrm{E}\left[Y_j\right]$, we would have the expected value of the counter $C(i, h_i(x))$.)

Since $\mathrm{E}\left[Y_j\right] = 1/B$, and since there are $N - n(x)$ values of j where $s_j \ne x$, we conclude that $\mathrm{E}\left[error(x)\right] = (N - n(x))/B \le N/B$. $\qquad\square$

Now, we use Markov's Inequality, which states that for any random variable X, $\Pr\left[X > a\right] \le \mathrm{E}\left[X\right]/a$. Notable, in this case, $\Pr\left[error(x) > 2N/B\right] \le 1/2$, as needed.

Problem 1.c. Explain how to choose the values A and B so that, for an integer x, with probability at least $(1 - \epsilon)$, we find: $n(x) \le query(x) \le n(x) + \delta N$. Prove that your choice of A and B gives the proper bounds.

Solution:
Claim 2 *If $B = 2/\delta$ and $A = \log(1/\epsilon)$, then with probability at least $(1 - \epsilon)$: $n(x) \leq query(x) \leq n(x) + \delta N$.*

Proof First, we know that $query(x) \geq n(x)$ for all choices of A and B.
Next, we focus on the other side of the inequality. Fix $B = 2/\delta$. Now, as long as $error(x) \leq 2N/B$ for at least one of the hash functions, we will ensure that $query(x) \leq n(x) + 2N/B \leq n(x) + \delta N$. For each hash function, we have already shown that $\Pr[error(x) > 2N/B] \leq 1/2$. Since each of the hash functions is independent, the probability that $error(x) > 2N/B$ for all of them is at most $1/2^A$. Thus we choose $A = \log(1/\epsilon)$ and conclude that with probability at least $1 - 2^{\log(1/\epsilon)} = 1 - \epsilon$, for at least one of the hash functions, $error(x) \leq 2N/B$ as needed.
To conclude, with probability at least $1 - \epsilon$, $error(x) \leq 2N/B \leq \delta N$, and hence $n(x) \leq query(x) \leq n(x) + \delta N$. $\qquad\square$

Notice that the total space needed here is $2 \log 1/\epsilon/\delta$ counters, which is completely independent of N. The size of the stream does not matter at all! (Each counter needs log-bits to store the count, of course.)

Problem 1.d. Consider choosing the hash functions for use in the algorithm as follows: choose a random value $t \in [1, B]$ and define $h(x) = t$ for all x. (Obviously this is bad, since it maps every integer to the same counter.) Explain where your proof, above, goes wrong. Which step in the proof is not true for this hash function?
  Beware this may be more subtle than you expect: notice, for example, that the expected number of elements mapped to each counter is *still* N/B, even for this bad hash function.

Solution: This is quite subtle, because if you fix a counter C and ask for the expected value of that counter, you will find that $E[C] \leq N/B$. Thus, be Markov's Inequality, the probability that counter C is $\geq 2N/B$ is $\leq 1/2$. Notice this is still true, despite the fact that the hash function is bad!
And of course, if the value of the counter is $\leq 2N/B$, you might easily conclude that the error is also $\leq 2N/B$. But in doing so, you have made a mistake! The choice of the counter matters. By fixing C, you cannot later change your mind and decide to focus on counter $C(i, h_i(x))$. And when you choose the counter $C(i, h_i(x))$, you are already assuming something about the hash function, i.e., that item x hashs to this counter. Since we have a bad hash function, if even one item maps to this counter, it means that *every* item maps to this counter.
To be a little more technical, the problem occurs where you attempt to calculate the expected error. A given element in the stream s_j only contributes to the error if $h_i(s_j) = h(x)$. Therefore we defined the Y_j to capture this dependency. We stated that $\Pr[Y_j = 1] = 1/B$. However, for the bad hash function described in this part, that is not true! In this case, $\Pr[Y_j = 1] = 1$. Hence the expected error of counter $C(i, h_i(x)) = (N - n(x))$.
The moral of the story is that you have to be very, very careful with dependencies. In this case, the dependency was hidden by notation, i.e., by saying fix counter $C(i, h_i(x))$. Obscured by that notation was the implicit assumption that x mapped to this specific counter, and that changes the analysis significantly.

Problem 1.e.   (Optional) Instead of using a perfect hash function, assume that h is chosen at random from a universal family of hash functions that map elements to the range $[1, B]$. The only guarantee that we have on h is that for every pair of elements (x, y), $\Pr[h(x) = h(y)] \leq 1/B$. (Notice that we know how to find universal families where each hash function requires only $O(\log n)$ bits to specify. See CLRS.) Explain why your proof (above) still works, even when you only have this weaker property.

Solution: It is important that the hash functions are independent of each other. However, the only place we use the property of the hash function is in showing that the expected value of Y_j is $1/B$. This fact is *exactly* the property guaranteed by a universal family of hash functions.

In general, if you want to use a weaker hash function (and hence hash functions that are easier to store), look closely at exactly where in the analysis you are using the power of the hash function. Here, it is only in one place and hence it is easy to derive the exact property needed from our hash functions. The bad hash function in Part (d) does not satisfy this requirement, but a universal hash family (or a 2-independent hash family) will satisfy this requirement.

Problem 1.f.   (Just for fun) Compare the solution here to a (Counting) Bloom filter. How are they similar or different?