CS5234: Combinatorial and Graph Algorithms

Problem Set 1

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

Ex. 1.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. 1.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 1. 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.)

Problem 2. 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 $(1-\epsilon)$: $n(x) \leq query(x) \leq n(x) + \delta N$**.** 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.)

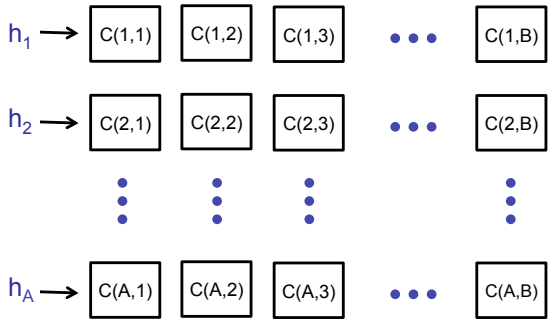

Figure 1: We use AB counters. Each of the A hash functions maps each of the incoming elements in the stream to one of the B counters in its row.

- 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 2.a. Explain why $query(x) \geq n(x)$.

Problem 2.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 $\Pr[error(x) \geq 2N/B] \leq 1/2$. (Hint: use Markov's Inequality.)

Problem 2.c. Explain how to choose the values A and B so that, for an integer x, with probability at least $(1 - \epsilon)$, we find: $n(x) \leq query(x) \leq n(x) + \delta N$. Prove that your choice of A and B gives the proper bounds.

Problem 2.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.

Problem 2.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.

Problem 2.f. (Just for fun) Compare the solution here to a (Counting) Bloom filter. How are they similar or different?