CS5234: Combinatorial and Graph Algorithms

Problem Set 4

Due: September 13th, 6:29pm

Instructions.

- 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. Consider the following discrete "alternative" implementation of the Flajolet-Martin algorithm for counting the number of distinct elements in a stream. Assume the stream consists of integers in the range $[1, M]$. And assume you have $L = \log M$ hash functions $h_1, h_2, \ldots, h_L$ where each $h_i : [1, M] \to \{0, 1\}$ maps elements from the stream to 0 or 1. For each hash function h_i, for each element $\ell \in [1, M]$, assume that:

$$\Pr[h_i(\ell) = 1] = 1/2^i \ .$$

The stream processing algorithm proceeds as follows:

1. $maxHash = 0$
2. for each element x in the stream:

   for each $i \in [1, L]$ do:

   if $h_i(x) = 1$ then $maxHash = \max(maxHash, i)$.
3. Return $2^{maxHash}$.

Assume there are D distinct elements in the stream. Prove that, for some constant c, the value returned by the algorithm is at least D/c and at most cD with probability at least $2/3$. (How is this algorithm similar to FM? How is it different?).

Exercise 2. Carefully complete the missing details of the analysis for FM+ and FM++.

Exercise 3. In tutorial, we looked at the Morris Counter and showed that the expectation is n. See the Tutorial Notes for details.

Ex. 3.a. Calculate the variance of the Morris Counter.

Ex. 3.b. Let Morris+ be the algorithm wherein you run a copies of Morris and return the average value. Analyze Morris+, and show that for a properly chosen value of a, this ensures that

the counter gives you a $(1 \pm \epsilon)$ approximation with constant probability.

Ex. 3.c. Let Morris++ be the algorithm wherein you run b copies of Morris+ and return the median value. Analyze Morris++, and show that for a property shown value of b, this ensures that the counter gives you a $(1 \pm \epsilon)$ approximation with probability at least $1 - \delta$.

Exercise 4. In tutorial, we looked at reservoir sampling. It turns out that reservoir sampling is closely related to permuting an array. Imagine you are given an array $A[1 \,..\, n]$. Here is an algorithm for permuting the array in a random fashion that is, essentially, the same as reservoir sampling:

```
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 random in the following sense: the probability that x ends up in slot i should be $1/n$ (for all slots i). (Hint: use induction from 1 to n.). (Note: this is not exactly equivalent to proving that the array ends up as a random permutation, however that is true too!)

Standard Problems (to be submitted)

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

That is, each query has at worst an additive error of δN (with probability at least $1 - \epsilon$). We will use a two-dimensional array of counters to keep track of the elements:

- Assume we have 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 acts as a uniform random function (except that it always maps the same item to the same value), 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 $\{C(i, h_i(x)) | i \in [1, A]\}$. (That is, for each of the rows, look at the counter that x hashes to and take the minimum of the rows.)

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.

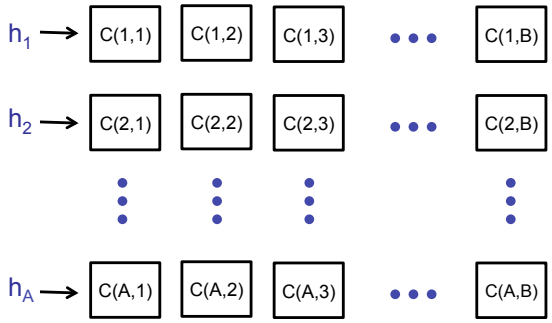

Figure 1: We use AB counters. Each of the A hash functions maps each of the incoming elements in the stream to one of the B counters in its row.

Problem 1.a. Explain why $query(x) \geq n(x)$. (That is, the result of the query is at least as big as the number of times x has appeared in the stream.)

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

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: $n(x) \leq query(x) \leq n(x) + \delta N$. Prove that your choice of A and B gives the proper bounds.

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.

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