CS5234: Combinatorial and Graph Algorithms

Problem Set 2

Due: August 25th, 6:30pm

Instructions. The problem set begins with a few exercises that you do not need to hand in. The problems this week introduce the idea of sublinear time approximation algorithms: they show how to solve large-scale problems while looking at only a very small amount of the data! These types of algorithms rely heavily on sampling techniques (and Chernoff-style bounds) to determine large-scale properties from small samples.

- 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. In this exercise, we will design a mergable set using a $(2, 3, 4)$-tree.

Ex. 1.a. First, we want to maintain in each node v in the $(2, 3, 4)$-tree a variable $height(v)$ that represents the height of v. Explain how to modify the $(2, 3, 4)$-tree algorithm to maintain the height properly. (Note: assume that leaf nodes have height 1, and every node has height exactly one more than its children.)

Ex. 1.b. Next, assume we have two $(2, 3, 4)$-trees T_1 and T_2 where every item in T_1 is less than every item in T_2. Show how to merge the two trees into a new $(2, 3, 4)$-tree in time $O(1 + |height(T_2) - height(T_1)|)$. (Notice that if the two trees are approximately the same size, this results in an $O(1)$ merge operation. In the worst-case, this costs $O(\log n)$.)

Ex. 1.c. Given a $(2, 3, 4)$-tree T and an element v, give an algorithm for splitting the tree T into two parts T_1 and T_2 where every element in T_1 is $\leq v$, and every element in T_2 is $> v$. Your algorithm should run in $O(\log n)$ time, and the resulting trees should be valid $(2, 3, 4)$-trees. You may use the merge routine from the previous part.

Exercise 2. Recall, a graph $G = (V, E)$ is said to be k-edge-connected if, for every cut in the graph $(S, V \setminus S)$ where $S \subseteq V$, there are at least k edges across the cut. Show that for a graph $G = (V, E)$: if, for every pair of nodes u, v, there are at least k edge-disjoint paths connecting u and v, then graph G is k-edge-connected. (You can prove the converse as well, but that requires more powerful tools.)

Exercise 3. Recall the following two Chernoff Bounds: Given independent random variables $x_1, x_2, \ldots, x_n$ where each $x_i \in [0, 1]$, let $X = \sum(x_i)$ and $\mu = \mathrm{E}[X]$. Choose $0 \leq \delta \leq 1$.

$$\Pr[X \geq (1 + \delta)\mu] \leq e^{-\mu\delta^2/3} \Pr[X \leq (1 - \delta)\mu] \leq e^{-\mu\delta^2/2}$$

Assume you have independent random variables $y_1, y_2, \ldots, y_n$ where each $y_i \in [0, s]$ for some fixed constant s. Let $Y = \sum(y_i)$ and $\mu = \mathrm{E}[Y]$. Prove that for all $0 \leq \delta \leq 1$:

$$\Pr[Y \geq (1 + \delta)\mu] \leq e^{-\mu\delta^2/(3s)} \Pr[Y \leq (1 - \delta)\mu] \leq e^{-\mu\delta^2/(2s)}$$

Standard Problems (to be submitted)

Problem 1. All Zero Arrays

Assume you have a very large array $A[1\,..\,n]$ where each $A[i]$ is either 0 or 1. The array large and stored on disk (or perhaps, on a remote server) and so we want to minimize the number of cells in the array that we access.[1]

For this problem, you may want to use the following version of a Chernoff Bound, known as Hoeffding's inequality. Given independent random variables $x_1, x_2, \ldots, x_n$ where each $x_i \in [0,1]$, let $X = \sum(x_i)$ and $\mu = \mathrm{E}[X]$. Choose $t > 0$:

$$\Pr[|X - \mu| \geq t] \leq 2e^{-2t^2/n}$$

Notice in this version, we bound the deviation from the mean by t. In the more traditional Chernoff bound, we set $t = \epsilon\mu$.

Problem 1.a. Our first goal is to decide whether A is all-zeros, or A has many 1's. Since we want to minimize access to the array, we are not going to demand a perfect answer. Instead we just want to know if it is all zeros, or far from all zeros. More specifically, here are the requirements for the procedure, for a fixed parameter $\epsilon < 1/2$:

- If $A[i] = 0$ for all $i \in [1, n]$, return TRUE. (The array is all zero.)
- If the array A contains at least ϵn slots i where $A[i] = 1$, then return FALSE. (The array if far from all zero.)
- Otherwise, if the array contains > 0 and $< \epsilon n$ slots where $A[i] = 1$, then the algorithm may return TRUE or FALSE. (The array is in an intermediate state.)

Consider the following algorithm:

Algorithm 1: AllZeros(A, ϵ)

```
repeat 2/ε times
    Choose a random i ∈ [1, n].
    if A[i] = 1 then return false.
return true.
```

Prove that with probability at least 2/3, this algorithm satisfies the above requirement.

[1] An alternative motivation is that the array represents a set of experiments you might choose to run. If you run experiment i, then $A[i] = 0$ if experiment i succeeds and $A[i] = 1$ if experiment i fails. We want to minimize the number of experiments that we run.

Problem 1.b. Now our goal is to estimate the number of 1's in the array. More precisely, let z be the number of ones in the array A, and let $\epsilon < 1/2$ be a fixed error probability. Our algorithm should return a value p such that $(z/n) - \epsilon \leq p \leq (z/n) + \epsilon$. That is, p is within ϵ of being the precise fraction of ones in the array A.

Consider the following algorithm, which samples $s = 1/\epsilon^2$ positions in the array and uses this sample to estimate the overall fraction.

Algorithm 2: SampleOnes(A, ϵ)

```
1 count = 0
2 repeat s = 1/ε² times
3     Choose a random i ∈ [1, n].
4     if A[i] = 1 then count = count + 1.
5 return (count/s).
```

Prove that with probability at least 2/3, this algorithm satisfies the requirements. (Hint: How far can *count* be from its expected value in order to meet the requirement?)

Problem 2. Is it sorted?

How do you test whether an array is sorted? What if there is a bug in your sorting code? Or, even worse, what if the sorting service (that you *paid* to sort your array) is cheating and not performing as promised? Unfortunately, the dataset is very large (and perhaps stored on a remote server), and hence it is expensive to verify whether or not it is correctly sorted. Your goal in this question is to develop an efficient routine that tests whether your data is sorted.

In order to improve efficiency, we are going to give up on a 100% guarantee. Instead, we are going to ask for the following: we want to know, with probability at least 2/3, the data is at least 99% sorted.

What does it mean for a list to be 99% sorted? We say that a list of n elements is p-sorted (for $0 < p \leq 1$) if we can create a sorted list simply by deleting some set of $(1-p)n$ elements from the list. That is, the list is close to sorted in the sense that only a small number of elements are out of place.

For example, the list $[1, 4, 2, 6, 8, 7, 10, 12, 14, 20]$ is 80%-sorted since we can delete $\{2, 7\}$ to produce a sorted list $[1, 4, 6, 8, 10, 12, 14, 20]$.

Our goal is to find an $O(\log n)$ time procedure for ensuring that a list is 99% sorted with 99% accuracy. That is, we want our test to have the following guarantees:

- If the list is properly sorted, then 100% of the time, it returns the correct answer: true.
- If the list is not properly sorted, then 99% of the time, it returns the correct answer: false.

We first begin with a simple probability calculation. We then proceed to develop and analyze the algorithm.

Problem 2.a. Consider the following algorithm for deciding of the list is almost sorted:

---

1 **repeat** k times
2 Choose an element $i \in [1, n]$.
3 Check if $A[i] < A[i+1]$. If not, return **false**.
4 Check if $A[i] > A[i-1]$. If not, return **false**.
5 **return true**.

Explain why this is not a good solution. Give an example array where the array is not (3/4)-sorted (i.e., you need to remove more than 1/4 of the elements for the list to be sorted), but yet this algorithm fails to detect the problem whenever $k = o(n)$. (Prove that your example really is a bad example.)

Problem 2.b. Imagine you are given a bag of n balls. Fix a probability $0 < p < 1$. At least $(1-p)n$ of the balls are blue, and no more than pn of the balls are red. You randomly choose k balls from the bag, one at a time, replacing each ball in the bag after you look at it. For what value of k is it true that you will see a blue ball with probability at least 2/3? Give your answer as a function of p.

Problem 2.c. Assume you are given an unsorted array A, and you perform a "binary search" on this array. (Notice that it is very strange to perform a binary search on an unsorted list. There is no reason to believe it will find anything useful.) That is, you execute the following algorithm:

```
BinarySearch(A, key, left, right)
    if (left == right) then return left;
    else {
        mid = ceiling((left+right)/2)
        if (key < A[mid]) then return BinarySearch(A, key, left, mid-1);
        else return BinarySearch(A, key, mid, right);
    }
```

Assume that you have two keys k_1 and k_2. And assume:

- Binary search for k_1 returns slot s_1 (even though the array is not sorted).
- Binary search for k_2 returns slot s_2 (even though the array is not sorted).

Explain why the following is true: if $s_1 < s_2$, then $k_1 < k_2$.

Hint: Draw a picture. Think about what happens during a search. Think about why this is obviously true if the array A were sorted.

Problem 2.d. Now consider the following procedure for determining if your array is sorted:

```
IsSorted(A, k)
    for (r = 1 to k) do
        i = Random(1, A.length);
        j = BinarySearch(A, A[i], 1, A.length);
        if (i != j) then return false;
    return true;
```

In this procedure, we randomly choose k items in the array A. For each item, we perform a binary search for that item. If the binary search ever fails, then we return false, i.e., the test has failed. Otherwise we return true. What is the right value of k? Explain why for the proper value of k, this procedure guarantees the desired accuracy:

- If the list is properly (and perfectly) sorted in ascending order, then 100% of the time, it returns true.
- If the list is not p-sorted, then with probability at least $2/3$, it returns false.

(If neither case holds, then it can return any answer.) *Hint: think about what it would imply if the BinarySearch succeeded for more than p fraction of the elements in the array, even if it were not p-sorted. Use the result from Part (b) above.*