CS5234: Combinatorial and Graph Algorithms

Problem Set 2 (Solutions)

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 $O(\log n)$.)

Solution: **[Note: when solutions for exercises are included, they tend to be sketches at a lower level of detail than is expected for a regular problem.]** Let x be the largest item in T_1. Delete x from tree T_1. Assume w.l.o.g. that $height(T_2) > height(T_1)$. If $height(T_2) < height(T_1)$, the situation is similar, but in reverse.
Find a node v on the left spine of T_2 with height $T_2 + 1$. (This takes time $T_2 - T_1$ to walk down tree T_1.) Insert item x into node v, splitting and propagating up the tree as needed. Set the leftmost child of the node containing x to point to the root of T_1. Notice that when done, every node has the proper number of children (i.e., 2, 3, or 4), and every node in the tree has the same depth (i.e., either $height(h_2)$ or $height(h_2) + 1$, depending on whether the root split on insertion).
If $height(h_2) = height(h_1)$, then merge the two roots and x into a single new root. If it is overfull, then split it.

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.

Solution: **[Sketch]** Begin with two empty trees T_1 and T_2. Imagine a tree walk from the root to item v. At each step along the way, if the tree walk traverses right at node v, then merge the left sub-trees into T_1; if the tree walk traverses left, merge the right sub-trees into T_2. When v is reached, merge the children left of v into T_1 and the children right of v into T_2, inserting v into T_1. A naive analysis would yield a running time of $O(\log^2 n)$, but a more careful analysis—noting that the height of the merged trees sum to $O(\log n)$ and hence the $(height(T_2) - height(T_1))$ terms all sum to the height of the tree—yields an $O(\log n)$ time algorithm.

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.)

Solution: [**Sketch**] Asssume not, i.e., there is a cut $(S, V \setminus S)$ that does not have k edges crossing the cut. Let u be a node in S and let v be a node in $V \setminus S$. Let $P_1, \ldots, P_k$ be the k edge-disjoint paths connecting u and v. For each such path, there must be at least one edge on the path that crosses the cut. Thus, there are least k edges across the cut, which is a contradiction.
Conversely, assume the graph is k-connected, i.e., the minimum cut is at least k. Fix two nodes u and v, and set a capacity of 1 on every edge. By the MaxFlow-MinCut theorem, this implies that there is a flow connecting u and v of at least k. Since each edge only supports a flow of 1, we conclude (by flow decomposition) that there are at least k edge-disjoint flows connecting u and v, i.e., at least k edge-disjoint paths.

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 $0 \leq \delta \leq 1$:

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

Solution: Let $z_i = y_i/s$ and $Z = \sum(z_i)$. Let $\mu' = \mathrm{E}[Z]$. Then $\Pr[X \geq (1+\delta)\mu] = \Pr[Y \geq (1+\delta)\mu'] \leq e^{-\mu'\delta^2/3} \leq e^{-\mu\delta^2/(3s)}$. The same trick works for the other tail bound.

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.

Solution: There are two cases to consider: when A is all zeros, and when A is far from all zeros. (The third case, when of the other two cases hold, is trivial because either answer is acceptable.)
Claim 1 *If the array A is all zero, it always returns true.*

This is immediate, since if there are no ones in the array, then the algorithm will never find a one and hence will never return false.

Claim 2 *Assume the array contains at least ϵn 1's. Then with probability at least $2/3$, the algorithm returns false.*

Proof For each sample, the probability of finding a 1 is at least $\epsilon n/n = \epsilon$. Thus the probibility of *not* finding a 1 for $2/\epsilon$ samples is at most $(1-\epsilon)^{2/\epsilon} \leq e^{-\epsilon(2/\epsilon)} \leq e^{-2} \leq 1/3$. □

The key inequality here is that $e^{-2} \leq (1-1/n)^n \leq e^{-1}$ for all $n \geq 2$. You can prove this yourself by looking at the Taylor expansion of $e^x = 1 + x + x^2/2 + \ldots$.
Notice that, if you want a probability of success better than $2/3$, you can of course repeat more times.

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?)

Solution:
Summary. Assume that the array contains z non-zero entries, i.e., the correct answer is z/n. The goal is to show that $(z/n - \epsilon) \leq \mathit{count}/s \leq (z/n + \epsilon)$. Equivalently, we want to show that $(sz/n - \epsilon s) \leq \mathit{count} \leq (sz/n + \epsilon s)$. We do this using a Hoeffding Bound.

Analysis. For each element in the sample, we define $x_i = 1$ if the ith sample finds a 1, and $x_i = 0$ otherwise. Notice that when the algorithm completes, $\mathit{count} = \sum(x_i)$ and it returns $\sum x_i/s$.
We know that $\Pr[x_i = 1] = z/n$, and hence $\mu = \mathrm{E}[\sum x_i] = s(z/n)$. That is, the expected outcome of our sampler, divided by s, matches the desired answer. We now need to show that the actual outcome is close to the expectation:
Claim 3 $Pr[|\sum x_i - \mu| \geq \epsilon s]$.

Notice that if this claim is true, then we get the desired result. If $|\sum x_i - \mu| < \epsilon s$, then we know that $s(z/n) - \epsilon s < \sum x_i < s(z/n) + \epsilon s$. Dividing everything by s, we see that:

$$z/n - \epsilon < \sum x_i/s < z/n + \epsilon \ .$$

That is, we have shown that the algorithm returns a close approximation of the percentage of zeros.
Proof We can calculate the probability of this event using Hoeffding's Inequality, where $t = \epsilon s$, we see that:

$$\begin{aligned}
\Pr[|\textstyle\sum x_i - \mu| \geq \epsilon s] &\leq 2e^{-2\epsilon^2 s^2/s} \\
&\leq 2e^{-2\epsilon^2 s} \\
&\leq 2e^{-2\epsilon^2/\epsilon^2} \\
&\leq 2e^{-2} \\
&\leq 1/3
\end{aligned}$$

□

This implies that the algorithm returns the right answer with probability at least 2/3.

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 if the list is almost sorted:

```
1 repeat k times
2     Choose an element i ∈ [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.

Solution:
Summary. Our goal is to find an example array A such that the algorithm returns an incorrect answer with probability greater than $1/3$ as long as $k = o(n)$. That is, the array A should not be $(3/4)$-sorted, and yet the algorithm has probability $< 2/3$ of returning false.

Defining an array. Fix n to be even, and consider the following array $A = [n/2 + 1, n/2 + 2, \ldots, n, 1, 2, 3, \ldots, n/2$. First, we argue that this array is not $(3/4)$-sorted:
Claim 4 *The array A is not $(3/4)$-sorted.*

Proof Assume for the sake of contradiction that array A is $(3/4)$-sorted, meaning that there is some set B where $|B| \leq n/4$ and if we remove B from A, then A is correctly sorted. Consider the right half of the array A, i.e., $[1, 2, 3, \ldots, n]$. Since B contains at most $n/4$ elements and the right half of A contains $n/2$ elements, we know that the right half of A contains at least one element $< n/2$. Now consider the left half of the array A, i.e., $[n/2 + 1, n/2 + 2, \ldots, n$. Again, since B contains at most $n/4$ elements, we know that the left half of A contains at most one element $> n/2$. Thus we conclude that A is not sorted correctly even after the elements in B have been removed. □

Analysis. Next, we calculate the probability that the algorithm detect that A is not sorted after k repetitions:

Claim 5 *The probability of detecting that A is not sorted is at least $e^{-4k/n}$.*

Proof In each iteration, the algorithm only detects a problem if it chooses either n or 1. That is, the probability of detecting the problem in one iteration is $2/n$. Thus, the probability of not detecting the problem for k iterations is:

$$\begin{aligned}(1 - 2/n)^k &= (1 - 2/n)^{(n/2)(2k/n} \\ &\geq (e^{-2})^{2k/n} \\ &\geq e^{-4k/n}\end{aligned}$$

□

Conclusion. If $k < n/10$, then $4k/n < 2/5$, and hence the probability of not detecting the problem is at least $e^{-2/5} \geq 2/3$. Hence the probability of returning a correct answer is $< 1/3 < 2/3$.

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.

Solution: The probability that a given ball is not blue is p. The probability that none of the balls are blue is p^k. Thus we need to calculate the value of k such that $p^k < 1/3$. That is, $k = \log(1/3)/log(p)$

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.

Solution:

Summary. At some point during the search, the binary search must diverge, and from this we can conclude that one key must go one way and the other key must go the other way. A simple picture illustrates this quite well.

Details. Fix k_1 and k_2 where $k_1 \geq k_2$. We need to show that $s_1 \geq s_2$. Assume, for the sake of contradiction, that slot $s_1 < s_2$.

Consider the search taken be key k_1: let $i_1, i_2, i_3, \ldots, i_\ell$ be the slots examined by search s_1, where i_1 is the middle element of the array and $i_\ell = s_1$ is the final slot found by the search.

Now consider the search taken by key k_2. Notice that it also begins at i_1. At some point along the way, however, the search for k_2 diverges from the search for k_1, since the search for k_2 ends at slot $s_2 \neq s_1$. Let i be the last element in the sequence $i_1, i_2, \ldots$ where the search for k_1 and k_2 follow the same sequence. (That is, the search sequence for both k_1 and k_2 is identical up until i, but diverges after i.)

Since $k_1 > k_2$, there are three possibilities for what happens to the two search paths after slot i. Either k_1 and k_2 both search to the left of i, or both search to the right of i, or k_2 searches to the left of i and k_1 searches to the right of i. Since i is the last slot on the search sequence that the two share, we know that k_2 searches the left half and k_1 searches the right half.

Notice that by the property of binary search, the entire remaining search for k_2 will be $\leq i$ and the entire remaining search for k_1 will be $\geq i$. Hence we know that k_2 will find a slot $s_2 \leq s_1$, which is a contradiction.

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.*

Solution:
Summary. The majority of the proof focuses on the case where the array is not p-sorted. (The other case is easy.) In each iteration of the algorithm, it chooses an index and does a binary search. We want to show that there are a large number of elements which, if chosen by the algorithm, will result in it returning false. In particular, we want to show that there are more than pn possible bad choices, and hence the algorithm will have a probability of more than p of detecting that A is not sorted. We show this by arguing that, if it were not the case, then we could simply remove the "bad" elements and the array would be sorted. Hence if the array is not p-sorted, there must be a large number of bad choices.
Throughout the analysis, we assume that all the elements in the array are distinct. We leave it as an exercise to think about what happens if the array has duplicate elements.
We now proceed to give more details.

Two cases. There are two cases we need to examine: when A is perfectly sorted, and when A is far from sorted. One of these claims is easier:
Claim 6 *If A is sorted, then the algorithm returns true.*

This follows immediately from the fact that binary search works.

The hard case. We now consider the case where the array is not p-sorted. For each element in the array, we define the following terminology:

- We say that i is good if `BinarySearch(A, A[i], i, A.length)` returns i.
- We say that i is bad otherwise.

Notice that if the algorithm always finds good items, then it returns true. If it finds any bad items, then it returns false. We need to argue that when the array is not p-sorted, there are a lot of bad items:

Claim 7 *If A is not p-sorted, then there are more than pn bad items.*

Proof Assume for the sake of contradiction that there are $\leq pn$ bad items.
Let k_1 and k_2 be two good items in A, where k_1 is in slot s_1 and k_2 is in slot s_2. Since k_1 and k_2 are good, we know that a search for k_1 finds slot s_1 and a search for slot s_2 finds k_2.
By the previous part, we know that if $s_1 < s_2$, then $k_1 < k_2$; if $s_2 < s_1$, then $k_2 < k_1$. That is, items k_1 and k_2 are sorted properly with respect to each other in the array A.
In general, this means that all the good items in the array are sorted properly with respect to each other. Hence if we remove all the bad items from the array, then the array is sorted. That is, set B equal to the set of bad items. Then $A \setminus B$ is sorted.
However, by assumption, $|B| \leq pn$. That means that the array is at least p-sorted, which is a contradiction. Thus we conclude that there must be more than pn bad items. □

.
Now, we can fix a value of k. Assume $k = \log(1/3)/log(1-p)$, as defined in the previous part.

Claim 8 *If A is not p-sorted, then with probability at least $2/3$, the algorithm returns false.*

Proof By our previous claim, we know that there are at least pn bad items. This means that when we choose an item uniformly at random, we have a probability of $> p$ of finding a bad item. If we repeat k times, then the probability of not finding a bad item is $< (1-p)^{\log(1/3)/\log(1-p)} = 1/3$. From this we conclude that with probability at least $2/3$, the algorithm chooses a bad item and correctly returns false. □