CS 670 (Fall 2022) Final Exam (Takehome) — Due 12/07/2022, noon, in David's office

This exam is takehome, but apart from having more time and getting to work on it wherever you want, you should treat it exactly like an open-book in-class exam. Specifically, here are the key rules. (If in doubt, check with the instructor.)

(a) **You may access the textbook, anything we handed out or posted for this class (assignments, sample solutions, class notes), and anything you wrote for this class (class notes, homework solutions). You may not access any other books, any online resources (not even Wikipedia), notes taken by classmates or friends, etc.**

(b) **You may not communicate about this exam with anyone except the instructor and TA. Not with classmates, labmates, friends from undergraduate, or anyone else. Not even basic questions — you should communicate exactly as if you were sitting in an in-class exam, i.e., not at all.**

(c) **You may ask the TA and instructor questions in person, by e-mail or on Piazza. If you use Piazza, ensure that your post is private and only visible to the instructor and TA.**

(d) **The exam is due by noon on Wednesday, December 07, in David's office. Late submissions will not be accepted. If you are traveling, you must make arrangements with the instructor ahead of time, and may submit by e-mail, but only before the deadline.**

There are 4 pages and 5 questions.

GOOD LUCK

(1) [0 points]
Describe at least one way in which what you have learned in this class helps you gain a new perspective on your own research area, or gives you new tools which you think will help improve your own research. You can talk about broad understanding, or about a specific problem for which you think you might now have better approaches available, or anything else relating to your own work. If you are not research-active, then talk about perspectives on another area of CS you are most interested in.

Your answer to this question should reflect some serious thought, not superficial comments that you come up with in 5 minutes.

(2) [6+6=12 points]
Imagine that you have a water tank which is a bit rusty. Every now and then, it develops a leak. For each leak, it loses water at a total value of 1 dollar per unit time. Time is continuous, so with 5 leaks, you will lose 1 dollar per 0.2 units of time (or 50 cents per 0.1 units of time, etc.). At any time, you can fix all current leaks for 1 dollar total, regardless of how many leaks there are. Of course, new leaks may start after you fix the old ones. You do not know in advance when new leaks will develop, and only find out about a new leak i at the time t_i when it starts.

(a) Give and analyze a 2-competitive algorithm for this problem.
[Hint: Inspiration may come from Ski Rental.]

(b) Show that no deterministic online algorithm can be better than $\frac{1+\sqrt{5}}{2}$-competitive for this problem.[1] You can get most credit by showing that no deterministic online algorithm can be better than 3/2-competitive.

(3) [7+8+5=20 points]
Consider the problem of Binary Search when you might get incorrect answers. Our model is as follows: you have a sorted "array" of n elements. There is some element x you are looking for, which we assume is definitely somewhere in the array. When you compare x with position j of the array, there are three possibilities:

(a) If x is in fact in position j, you definitely are told so.

(b) If x is to the left of position i, then with probability $p > 1/2$, you are correctly told that it is to the left. But with probability $1 - p$, you are instead told that it is to the right.

(c) If x is to the right of position j, then with the same probability p, you are correctly told that it is to the right. But with probability $1 - p$, you are instead told that it is to the left.

You would like to find the element x with probability at least $1-\epsilon$ (where ϵ is a given input parameter), using few queries. Notice that simply running standard binary search will not work: if the first query tells you to go left, in standard binary search, all future queries will be to the left half, so if the answer was wrong, Binary Search can never recover. In particular, if $p < 1 - \epsilon$ (which it will usually be), you cannot achieve the desired $1 - \epsilon$ guarantee.

(a) One way to "fix" Binary Search is to use the standard Binary Search algorithm, but increase your chance of getting the answer right by asking each query multiple times. So the algorithm is: ask each query k times, take a majority vote of the responses, and follow that. Choose an appropriate k and analyze how many queries total will be needed to find x with probability at least $1 - \epsilon$.
[Hint: You will likely not get $O(\log n)$. But it should not be a whole lot worse than that. You will probably have an $O(\log(1/\epsilon))$ term in there.]

(b) A different, better, way to fix Binary Search is the following. Initially, each array element starts with a weight of $w_{j,1} = 1$. In each iteration t, you query the position j_t which right now

[1] In fact, no deterministic online algorithm can be better than 2-competitive, but that is a little harder to show.

is a weighted median. That is, at least half the weight is to the left of the weighted median ($\sum_{i=1}^{j_t} w_{i,t} \geq \frac{1}{2} \cdot \sum_{i=1}^{n} w_{i,t}$) and half the weight is to the right ($\sum_{i=j_t}^{n} w_{i,t} \geq \frac{1}{2} \cdot \sum_{i=1}^{n} w_{i,t}$).
Consider the possible outcomes of querying position j_t in round t. First, of course, if you are told that j_t is correct, you are done. If not, then whichever direction you are pointed, you multiply the weights of all positions by p; in the other direction, you multiply the weights by $1-p$; and for position j_t, you set the weight to 0. For example, if you are told to go left, then you set $w_{j,t+1} := w_{j,t} \cdot p$ for all $j < j_t$, $w_{j,t+1} := w_{j,t} \cdot (1-p)$ for all $j > j_t$, and $w_{j_t,t+1} := 0$. (As a sanity check, notice that when $p = 1$, so all answers are always correct, this is exactly standard Binary Search.)

(a) Assume first that rather than by coin flips, the answers are decided by an adversary — however, the adversary can only give incorrect answers to at most a $1-p$ fraction of queries. Prove that against such an adversary, the algorithm always finds x within $O(\log n)$ queries. Here, the big-O is allowed to hide dependencies on p.

(b) Use the analysis with an adversary to show that in the probabilistic model, the algorithm finds x within $O(\log n + \log(1/\epsilon))$ queries with probability at least $1-\epsilon$. Again, the big-O is allowed to hide dependencies on p.

(4) [10 points]
Suppose that we generate a "social network" as follows. There are four parameters, k, p, q, n. Out of these, k is the number of "groups", a small absolute constant, and $0 \leq q < p \leq 1$ are edge probabilities, also absolute constants[2]. n is the number of people in each "group", and we will look at n getting large. There are k disjoint groups of n individuals each. If two individuals u, v are in the same group, they will have an edge between them with probability p. If they are in different groups, the probability of having an edge is q. All edges are generated independently of each other. (Obviously, this model is not completely realistic, as individuals will be friends with a constant fraction of all people in the world.)

Now suppose that someone shows you a social network (graph) G that has been generated in this way, and you want to infer the groups from this. Obviously, there will be lots of edges between people who are not actually in the same group. However, there is a simple heuristic for figuring out whether they are in the same group: count their common friends[3]. If that number is greater than some threshold value, then they are likely to be in the same group. Otherwise, they are not.

Prove that there is a choice of threshold (possibly depending on k, p, q, n) such that for large enough n (meaning n larger than some constant that may depend on k, p, q), this heuristic will reconstruct the original group memberships perfectly (i.e., not misclassify *any* individual), with probability at least $1 - 1/n^2$. (The probability is taken over the random process of generating the graph.)

(5) [10 points]
In class, we saw an $O(\log n / \log\log n)$ approximation for the problem of routing paths with minimum congestion in an undirected graph. Recall that the setup was that we had k pairs (s_i, t_i), and for each i had to select an s_i-t_i path. The goal was to minimize the maximum load (number of paths using it) of any edge.

Suppose now that the graph is actually an undirected cycle. (If it helps you, you may assume that the number n of nodes is odd.) To make things a bit more interesting, each pair i has a *demand* $d_i \geq 0$, which is the amount of traffic it will put on the selected path. The load of an edge e is now not the *number* of paths that use e, but rather the *total load* of all paths using e. We still want to minimize the maximum load of any edge.

Give and analyze a polynomial-time 2-approximation algorithm for this problem. If your algorithm is randomized, it needs to be a 2-approximation in expectation or with high probability (your choice).

[2]That means that these three numbers will not vary with n.
[3]This is what Facebook does for friend recommendations, for instance.

[Hint: There is a simple algorithm with a slightly more complicated analysis, and a slightly more complicated algorithm with a simple analysis.]