CS256: Algorithm Design and Analysis **Spring 2021**

Assignment 9 (due 05/12/2021)

Instructor: Shikha Singh *Solution template:* Overleaf

Randomized Algorithms and Hashing

Problem 1. In this problem, we will design an algorithm that maximizes the probability of selecting the maximum element from a randomly-ordered input sequence. This problem is known as the **secretary problem**, because of the following motivating formulation.

Imagine that you need to hire a new secretary and you have n people to interview. You want to hire the best person for the position. Unfortunately you cannot tell how good a candidate is without interviewing them. Furthermore, during your interview with a candidate, you must immediately decide whether to make them an offer or to reject them. In particular, your decisions are irrevocable—you have to decide about the kth person, without observing any of the $k+1, \ldots, n$ candidates, and if you don't make an offer you forever lose the chance to hire that candidate.

We suppose the candidates are interviewed in a random order, chosen uniformly at random from all $n!$ possible orderings. Also, suppose that when you interview a candidate, you give them a score, with the highest score being the best and no ties being possible.

Consider the following strategy. First, interview the first m candidates but reject them all (too bad for them!); these candidates give you an idea of how strong the field is. After the mth candidate, hire the first candidate you interview who is better than all of the previous candidates you have interviewed.

Let A be the event that we hire the best candidate. Show that

$$\Pr[A] = \frac{m}{n} \sum_{j=m+1}^{n} \frac{1}{j-1}.$$

More about the secretary problem. We can approximate the sum $\sum_{j=m+1}^{n} \frac{1}{j-1} \approx \ln \frac{n}{m}$.

This gives us that, $\Pr(A) = \frac{m}{n} \ln \frac{n}{m}$. This probability is maximized by setting $m = n/e$.

Thus, the best strategy rejects the first $1/e$ fraction of the candidates, and uses them as a benchmark to hire among the remaining. In doing so, the algorithm hires the best candidate with probability exactly $1/e$! This is sometimes called the $1/e$ or 37% rule. In online decision-making problems (problems where the input is arriving *online* or one by one), this is sometimes referred to as the *optimal stopping rule*.

This problem first appeared in Martin Gardner's *Mathematical Games* column in the 1960 of the *Scientific American*. Since then, the problem and its variants have been studied extensively and have proved extremely useful in designing and analyzing online algorithms.

Problem 2 (Erickson handout)**.** Your boss wants you to find a perfect hash function for mapping a known set of n items into a table of size m. A hash function is perfect if there are no collisions; each of the n items is mapped to a different slot in the hash table. Of course, a perfect hash function is only possible if $m \geq n$.

After cursing your algorithms instructor for not teaching you about (this kind of) perfect hashing, you decide to try something simple: repeatedly pick random hash functions until you find one that happens to be perfect.

(a) Suppose you pick an ideal random hash function h. What is the exact expected number of collisions, as a function of n (the number of items) and m (the size of the table)?

Don't worry about how to resolve collisions; just count them.

(b) What is the exact probability that a random hash function is perfect?

(c) What is the expected number of different random hash functions you have to test before you find a perfect hash function? Give an exact answer (using your answer to part (b)), then simplify your answer using the techniques we've seen in class. (The simplified answer should be a function of m and n, rather than a product of many terms.)

Problem 3. (*Hashing with open addressing* from Mitzenmacher and Upfal.) In hashing with open addressing, the hash table is implemented as an array and there are no linked lists or chaining. Each entry in the array either contains one hashed item or is empty. On open addressing, we resolve collisions by looking at a different slot in the table. Specifically, we have a sequence of hash functions $(h_0, h_1, \ldots, h_{m-1})$, such that for any element x, the *probe sequence* $(h_0(x), h_1(x), \ldots, h_{m-1}(x))$ is a permutation of slots $(0, 1, \ldots, m-1)$.

To insert an element x, we first examine the sequence of table locations in the order defined by the element's probe sequence until we find an empty location; then we insert the item at that position. When searching for an item in the hash table, we examine the sequence of table locations in the order defined by the item's probe sequence until either the item is found or we have found an empty location in the sequence. If an empty location is found, this means the item is not present in the table.

Consider an open-address hash table with $m = 2n$ slots which is used to store n items. Assume that each $h_k(x)$ is uniform over the $2n$ slots, and each probe in $h_1(x), h_2(x), \ldots, h_{2n-1}(x)$ is independent of each other, and the probe sequence of other elements.

Our goal is to show that the cost of inserts in a hash table with open addressing is $O(\log n)$ with high probability.[1] We do this in parts.

(a) Suppose the ith probe is successful if the corresponding slot in the hash table is free. Show that on any insertion, the probability that the first t probes fail is at most 2^{-t}.

(b) Use (a) to show that the probability that a given insertion requires more than $3 \log n$ probes is at most $1/n^3$.

(c) Use (b) to show that when we insert n elements to the hash table, the cost of any insert is $O(\log n)$ with high probability. *Hint. Show that the probability that any insertion (out of the n) requires more than $3 \log n$ probes is at most $1/n^2$.*

[1] Recall that cost of an algorithm is $O(C)$ with high probability iff the probability that the cost is a constant factor greater than C is polynomially small in n, that is, is at most $1/n^d$ for some constant $d \geq 2$.

Approximation Algorithms for NP hard Problems

Problem 4. (Max Cut.) Given an undirected graph $G = (V, E)$, the max cut problem is to find a cut (A, B) of maximum cardinality. In other words, the problem is to find a cut in G with the maximum number of edges crossing the cut. Surprisingly, even though there are many ways to solve the minimum-cardinality cut, the max cut problem is NP hard. In this question, we will design a simple approximation algorithm for this problem using randomization.

Consider the following simple strategy:

> For each vertex, toss a fair coin
>
> > If it lands heads, place the node into A
> > If it lands tails, place the node into B

Let c_{AB} denote the size of the cut (A, B) returned by this algorithm, and let OPT be the size of the largest cut in G. Show that $E(c_{AB}) \geq 1/2 \cdot \text{OPT}$. Thus, in expectation this algorithm is a 0.5 approximation to the max-cut problem. *Hint. Show that* $\text{E}(c_{AB}) = m/2$*, where* m *is the number of edges.*

Problem 5. (Vertex Cover.) In this question, we design a simple deterministic approximation algorithm for the NP hard problem, Vertex Cover. Given a graph G, the optimization version of the problem is to find the smallest vertex cover.

Consider the following simple strategy:

> Start with vertex cover $S \leftarrow \emptyset$
> While there is an uncovered edge $e = (u, v)$:
>
> - Add both endpoints to S, that is, $S \leftarrow S \cup \{u, v\}$
> - Delete all edges that are incident on u and v

Show that the above algorithm is a 2-approximation. In particular, show that S is a vertex cover and that $|S| \leq 2 \cdot |S^*|$, where S^* is a optimal (minimum-size) vertex cover.