CS 170 Homework 2

Due **Friday 9/13/204, at 10:00 pm (grace period until 11:59pm)**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, explicitly write "none".

2 Quantiles

Let A be an array of length n. The boundaries for the k quantiles of A are

$$\{a^{(n/k)}, a^{(2n/k)}, \dots, a^{((k-1)n/k)}\}$$

where $a^{(\ell)}$ is the ℓ-th smallest element in A.

Devise an algorithm to compute the boundaries of the k quantiles in time $\mathcal{O}(n \log k)$. For convenience, you may assume that k is a power of 2.

Hint: With careful choice of pivot, SELECT*(A, ℓ) gives $a^{(\ell)}$ in guaranteed $\mathcal{O}(n)$ time. We will see this in the next question.*

Solution: The idea is to find the median of A, partition it into two pieces around this median. Then we recursively find the medians of the two partitions, partition those further, and so on. If we do this $\log k$ times, we will have found all of the k-quantiles.

Finding the median and partitioning A takes $\mathcal{O}(n)$ time. We also do this for two arrays of size $n/2$, four times for arrays of size $n/4$, etc. So the total time taken is

$$\mathcal{O}(n + 2 \cdot n/2 + 4 \cdot n/4 + \cdots + 2^{\log k} n/2^{\log k}) = \mathcal{O}(\underbrace{n + n + \cdots + n}_{\log k \text{ times}}) = \mathcal{O}(n \log k)$$

3 Median of Medians

The QuickSelect(A, k) algorithm for finding the kth smallest element in an unsorted array A picks an arbitrary pivot, then partitions the array into three pieces: the elements less than the pivot, the elements equal to the pivot, and the elements that are greater than the pivot. It is then recursively called on the piece of the array that still contains the kth smallest element.

(a) Consider the array $A = [1, 2, \ldots, n]$ shuffled into some arbitrary order. What is the worst-case runtime of QuickSelect(A, $n/2$) in terms of n? Construct a sequence of 'bad' pivot choices that achieves this worst-case runtime.

Solution: A single partition takes $\mathcal{O}(n)$ time on an array of size n. The worst case would be if the partition times were $n + (n-1) + \cdots + 2 + 1 = \mathcal{O}(n^2)$.

This would happen if the pivot choices were e.g. $1, 2, 3, \ldots, n/2-1, n, n-1, \ldots, n/2+1$. In each of these cases, the partition happens so that one piece only has one element, and the other piece has all the other elements. To get a better runtime, we would like the pieces to be move balanced.

The above 'worst case' has a chance of occurring even with randomly-chosen pivots, so the worst-case time for QuickSelect is $\mathcal{O}(n^2)$, even though it achieves $\Theta(n)$ on average.

Based on QuickSelect, let's define a new algorithm DeterministicSelect that deterministically picks a consistently good pivot every time. This pivot-selection strategy is called 'Median of Medians', so that the worst-case runtime of DeterministicSelect(A, k) is $\mathcal{O}(n)$.

Median of Medians

1. Group the array into $\lfloor n/5 \rfloor$ groups of 5 elements each (ignore any leftover elements)
2. Find the median of each group of 5 elements (as each group has a constant 5 elements, finding each individual median is $\mathcal{O}(1)$)
3. Create a new array with only the $\lfloor n/5 \rfloor$ medians, and find the true median of this array using DeterministicSelect.
4. Return this median as the chosen pivot

(b) Let p be the pivot chosen by DeterministicSelect on A. Show that at least $3n/10$ elements in A are less than or equal to p, and that at least $3n/10$ elements are greater than or equal p.

Solution: Let the choice of pivot be p. At least half of the groups ($n/10$) have a median m such that $m \leq p$. In each of these groups, 3 of the elements are at most the median m (including the median itself). Therefore, at least $3n/10$ elements are at most the size of the median.

The same logic follows for showing that $3n/10$ elements are at least the size of the median.

This content is protected and may not be shared, uploaded, or distributed.

(c) Show that the worst-case runtime of DETERMINISTICSELECT(A, k) using the 'Median of Medians' strategy is $\mathcal{O}(n)$.

Hint: Using the Master theorem will likely not work here. Find a recurrence relation for $T(n)$, then show that $T(n) \leq c \cdot n$ for some sufficiently large $c > 0$. You can assume $T(n) \leq c \cdot n$ for small values of n without proof.

Solution: We end up with the following recurrence:

$$T(n) \leq \underbrace{T(n/5)}_{\text{(A)}} + \underbrace{T(7n/10)}_{\text{(B)}} + \underbrace{d \cdot n}_{\text{(C)}}$$

Each term can be broken down as follows:

(A) Calling MEDIAN OF MEDIANS to find a suitable pivot.

(B) The recursive call to DETERMINISTICSELECT after performing the partition. The size of the partition piece is always at most $7n/10$ due to the property proved in the previous part.

(C) The time to construct the array of medians, and to partition the array after finding the pivot. This is $\mathcal{O}(n)$, but we explicitly write that it is $d \cdot n$ for convenience in the next part.

We cannot simply use the Master theorem to unwind this recurrence. Instead, we show by induction that $T(n) \leq c \cdot n$ for some $c > 0$. The base case happens when DETERMINISTICSELECT occurs on one element, which is constant time.

For the inductive case,

$$\begin{aligned} T(n) &\leq T(n/5) + T(7n/10) + d \cdot n \\ &\leq c(n/5) + c(7n/10) + d \cdot n \\ &\leq \left(\frac{9}{10}c + d\right) \cdot n \end{aligned}$$

We pick c large enough so that $\left(\frac{9}{10}c + d\right) \leq c$, i.e. $c \geq 10d$, and we are finished. Because $T(n) \leq c \cdot n$ for constant c, $T(n) = \mathcal{O}(n)$.

4 The Resistance

We are playing a variant of The Resistance, a board game where there are n players, s of which are spies. In this variant, in every round, we choose a subset of players to go on a mission. A mission succeeds if the subset of the players does not contain a spy, but fails if at least one spy goes on the mission. After a mission completes, we only know its outcome and not which of the players on the mission were spies.

Come up with a strategy that identifies all the spies in $O(s \log(n/s))$ missions. **Describe your strategy and analyze the number of missions needed.**

Hint 1: consider evenly splitting the n players into x disjoint groups (containing $\approx n/x$ players each), and send each group on a mission. At most how many of these x missions can fail? What should you set x to be to ensure that you can reduce your problem by a factor of at least $1/2$?

Hint 2: it may help to try a small example like $n = 8$ and $s = 2$ by hand.

Solution: Observe that if we partition the players into x disjoint groups and send each group on a mission, at least $x - s$ groups will succeed and we can rule out the players in those groups.

To discard at least half of the players in each iteration, partition them into $2s$ groups. For the missions that succeed, we remove those groups, and split the remaining groups in half to form a new set of groups. We repeat this procedure until each group has one person left, at which point we know they are a spy.

Runtime analysis: In the first iteration of this procedure we have $2s$ groups going on missions. After each group goes on a mission, since there are s spies, at most s of the missions fail, which means after splitting the groups that failed, we have at most $2s$ groups again.

In each iteration, at least half of the players are removed from groups. So we identify the spies after $O(\log(n/s))$ iterations. So the total number of missions needed is $O(s \log(n/s))$.

Alternate solution: Divide the players into two groups and send each group on a mission. If a group succeeds then discard those players and if a group fails then recursively use the same strategy on that group until a group of size 1 is reached.

Runtime Analysis: After drawing out the recursion tree you'll see that at level i, at most $\min(2^i, s)$ missions can fail and those are the nodes that will branch out to the next level. There are at most $\log n$ levels. Hence, the total number of missions is bounded by,

$$\begin{aligned}\sum_{i=i}^{\log n} \min(2^i, s) &\leq \sum_{i=1}^{\log s} 2^i + \sum_{i=\log s}^{\log n} s \\ &= O(s) + s(\log n - \log s) \\ &= O(s \log(n/s))\end{aligned}$$

5 Poker

You are playing poker with n other friends, who either always tell the truth or sometimes bluff (lie). You do not know who may bluff and who will always tell the truth, but all your friends do. All you know is that there are more people who always tell the truth than people who bluff.

Your goal is to identify one specific player who always tells the truth.

You are allowed to perform a 'query' operation as follows: you pick two players as partners. You ask each player if their partner bluffs or always tells the truth. When you do this, players who tell the truth will tell the truth about the identity of their partner, but a player who bluffs can either lie or tell the truth about their partner.

Your algorithm should work regardless of whether bluffing players lie or tell the truth.

(a) For a given player x, devise an algorithm that returns whether or not x is always telling the truth using $O(n)$ queries. Just an informal description of your test and a brief explanation of why it works is needed.

(b) Show how to find a player who always tells the truth in $O(n \log n)$ queries (where one query is taking two players x and y and asking x to identify y and y to identify x).

Hint: Split the players into two groups, recurse on each group, and use part (a). What invariant must hold for at least one of the two groups?

Give a 3-part solution.

(c) (**Optional, not for credit**) Can you give a $O(n)$ query algorithm?

Hint: Don't be afraid to sometimes 'throw away' a pair of players once you've asked them to identify their partners.

Give a 3-part solution.

Solution: For convenience, we will refer to players who always tell the truth as *truthers* and players who bluff as *bluffers*.

(a) To test if a player x is a truther, we ask the other $n-1$ players what $x's$ identity is. Claim: x is a truther if and only if at least half of the other players say x is a truther. To see this, notice that if x is a truther, at least half of the remaining players are also truthers, and so regardless of what the bluffers do at least half of the players will say x is a truther. On the other hand, if x is a bluffer, then strictly more than half of the remaining players are truthers, and so strictly less than half the players can falsely claim that x is a truther.

(b) **Main idea** The divide and conquer algorithm to find a truther proceeds by splitting the group of friends into two (roughly) equal sets A and B, and recursively calling the algorithm on A and B: $x = \mathit{truther}(A)$ and $y = \mathit{truther}(B)$, and checking x or y using the procedure in part (a) and returning one who is a truther. If there is only one player left, they are guaranteed to be a truther, so return that player.

Proof of correctness We will prove that the algorithm returns a truther if given a group of n players of which a majority are truthers. By strong induction on n:

Base Case If $n = 1$, there is only one player in the group who is a truther and the algorithm is trivially correct.

Induction Hypothesis The claim holds for $k < n$.

Induction Step After partitioning the group into two groups A and B, at least one of the two groups has more truthers than bluffers. By the induction hypothesis the algorithm correctly returns a truther from that group, and so when the procedure from part (a) is invoked on x and y at least one of the two is identified as a truther.

Runtime analysis Two calls to problems of size $n/2$, and then linear time to compare the two players returned to each of the friends in the input group: $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$ by Master Theorem.

(c) **Main idea** Split up the friends into pairs and for each pair, if either says the other is a bluffer, discard both friends; otherwise, discard any one and keep the other friend. If n was odd, use part (a) to test whether the odd man out is a truther. If yes, you are done, else recurse on the remaining at most $n/2$ friends.

Proof of correctness After each pass through the algorithm, truthers remain in the majority if they were in the majority before the pass. To see this, let n_1, n_2 and n_3 be the number of pairs of friends with both truthers, both bluffers and one of each respectively. Then the fact that truthers are in the majority means that $n_1 > n_2$. Note that all the n_3 pairs of the third kind get discarded, and one friend is retained from each of the n_1 pairs of the first kind. So we are left with at most $n_1 + n_2$ friends of whom a majority n_1 are truthers. It is straightforward to now turn this into a formal proof of correctness by strong induction on n.

Runtime analysis In a single run of the algorithm on an input set of size n, we do $O(n)$ work to check whether f_1 is a truther in the case that n is odd and $O(n)$ to pair up the remaining friends and prune the candidate set to at most $n/2$ players. Therefore, the runtime is given by the following recursion:

$$T(n) = T\left(\frac{n}{2}\right) + O(n) = O(n) \text{ by Master Theorem.}$$

Alternative solution: Pick a player at random and use part (a) to check whether they are truther, if not then repeat the process again. Since at least half the players are truthers, the **expected** number of players we will have to check until we find a truther is $O(1)$. Since the check requires $O(n)$ queries, the overall solution also uses $O(n)$ queries **in expectation**.

This content is protected and may not be shared, uploaded, or distributed.