# CS170Hw2

Trustin Nguyen

September 13, 2024

## Study Group

Rohan Kudchadker: 3038065006

## Quantiles

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

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

where $a^l$ 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.

*Answer.* We can use Select algorithm to find the median of the array first, where the select algorithm is described more in q2 ($\mathcal{O}(n)$ runtime). Then we break the $n$ length array into two subarrays, one with elements less than the median found and one with elements greater than the median found. Then we find the median of those and repeat. Each median will be a $k$ quantile. Once we have found the $k-1$ medians, those will be our answer and we stop splitting the array.

Runtime: Each subarray of length $m$ gets $\mathcal{O}(m)$ work from the Select algorithm to find the median. At depth $d$, there are $2^d$ arrays. Each array at that depth does $\frac{n}{2^d}$ work, so it is $\mathcal{O}(n)$ at each level. The number of levels we have is $\log_2(k)$ because for each subarray we find their $k/2$ quantiles. The recursive formula ends up being $T(k) = 2T(k/2) + \mathcal{O}(n)$. We have runtime as $\mathcal{O}(n \cdot \log k)$

## Median of Medians

The $\mathsf{QuickSelect}(A, k)$ algorithm for finding the $k$th 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 $k$th smallest element.

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

*Answer.* The worst case is $\mathcal{O}(n^2)$. This happens if you pick the farthest element from the median as the pivot at every step. One example is

$$A = [1, 2, 3, 4, 5, 6, 7, 8] \text{ shuffled}$$

We can pick $1, 8, 2, 7, 6, 4$ as pivots in that order. We split $A$ into two subarrays, one is $[]$ and the other is $[2, 3, 4, 5, 6, 7, 8]$. When we split using these pivots, we will always get another subarray which is empty. This goes to depth $n$ and constructing each subarray takes order $d$ for the depth. So $\sum_{i=0}^{n} i = \mathcal{O}(n^2)$.

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

Based on $\mathsf{QuickSelect}$, let's define a new algorithm $\mathsf{DeterministicSelect}$ 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 $\mathsf{DeterministicSelect}(A, k)$ 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 $\mathsf{DeterministicSelect}$.
4. Return this median as the chosen pivot.

(b) Let $p$ be the pivot chosen by $\mathsf{DeterministicSelect}$ 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 $p$.

*Answer.* Suppose that $p$ is chosen by $\mathsf{DeterministicSelect}$. Then that means that there are $n/10$ medians chosen from step 2 that are less than or equal to $p$ and $n/10$ medians from step 2 that are greater than or equal to $p$. Let $L$ be the set of $n/10$ medians $\leqslant p$, $G$ be the set of $n/10$ medians $\geqslant p$. Each element of $L$ was a median from a set of 5 elements:

$$L = \{l_1, l_2, \dots, l_{n/10}\}$$
$$G = \{g_1, g_2, \dots, g_{n/10}\}$$

We know that each $l_i$ were chosen from an array of 5 elements. So each $l_i$ has a corresponding array $|l_{i_l}| = 3$ and $|l_{i_g}| = 3$ where $l_{i_l}$ are elements $\leqslant l_i$, $l_{i_g}$ are elements $\geqslant l_i$. This means that the number of elements that are less than or equal to our median from deterministic select is:

$$\sum_{i=0}^{n/10} |l_{i_l}| = \sum_{i=0}^{n/10} 3 = 3n/10$$

and the number of elements that are greater than or equal to our median is:

$$\sum_{i=0}^{n/10} |g_{i_g}| = \sum_{i=0}^{n/10} 3 = 3n/10$$

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

*Answer.* To get the pivot, we first split the $n$ length array into $n/5$ groups. For each group, getting the median gives a total of $\mathcal{O}(n/5)$. We have $n/5$ medians, and recurse with deterministic select. This means we get an added $T(n/5)$ runtime, where deterministic select is of time $T(n)$. When the recursive stack frames close and we make our way back up, deterministic select returns a pivot. On that same level, we want to find the median given the pivot from deterministic select which will be in a subarray of size $7n/10$. This will take at most $T(7n/10)$. So the recursive equation is:

$$T(n) = T(n/5) + T(7n/10) + \mathcal{O}(n/5)$$

Now we find the bound. We can start off our tree with a node of size $n$ that does $n$ work and splits into two nodes of sizes $n/5, 7n/10$. The nodes do $\mathcal{O}(n/5), \mathcal{O}(7n/10)$ work respectively and so on. At each level, we want to find the combined work. Using the binomial theorem, we can mimic the convolution product as $\mathcal{O}((n/5 + 7n/10)^d)$ combined work for depth $d$. Now to find the max height of our tree. We are looking for when $(\frac{7}{10})^d n = 1$ since the factor $7/10$ results in the least decay for our input $n$. Solving for $d$:

$$n = \left(\frac{10}{7}\right)^d$$
$$d = \log_{10/7} n$$

So the max depth is given above. Combine this for the total work:

$$\sum_{i=0}^{\log_{10/7} n} \left(\frac{1}{5} + \frac{7}{10}\right)^i n = n \sum_{i=0}^{\log_{10/7} n} \left(\frac{9}{10}\right)^i = n\left(\frac{1}{1 - \frac{9}{10}}\right) = 10n$$

So we have that $T(n) < \mathcal{O}(10n)$. Therefore, $T(n) = \mathcal{O}(n)$.

## 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.

*Answer.* The strategy will be:

1. Divide the players into $2s$ groups, each of size at least $\lfloor n/2s \rfloor$.
2. Run a mission for each group.
3. Remove the groups that ran a mission successfully.
4. For the unsuccessful groups, recombine the players and repeat from step 1 until there are $s$ players left.
5. The remaining players are spies.

We divide the players into $2s$ at each level because we know that there are $s$ spies. This means that at least $s$ groups do not have spies. So when we remove the players, we cut the number of players in half each time. The amount of missions at each level is $2s$.

Now we need to find the depth. The program terminates when the number of remaining players is equal to $s$. At depth $d$, the number of players is $n/2^d$. So we have:

$$\begin{aligned} \frac{n}{2^d} &= s \\ \frac{n}{s} &= 2^d \\ \log_2(n/s) &= d \end{aligned}$$

Since we go to depth $\log_2(n/s)$ and we have $2s$ missions per depth, the number of missions is $O(s\log(n/s))$

## 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.

*Answer.* There are $n-1$ other players. We just do a pairwise query operation between the chosen person and the $n-1$ other people. There are two cases:

(a) The person chosen always tells the truth. Then that means that when we query with the $n-1$ other people, There are at least $\lfloor \frac{n-1}{2} \rfloor$ queries in which the pair says that they are both telling the truth. This is because there are more truth tellers than there are people who bluff.

(b) The person chosen sometimes bluffs. When we query with the $n-1$ other people, there are less than $\lfloor \frac{n-1}{2} \rfloor$ queries in which the pair says that they are both telling the truth. This is because the only instance where they both report they are telling the truth is when a person who bluffs is paired with another who bluffs. There are less people who bluff than those who tell the truth.

So if there are at least $\lfloor \frac{n-1}{2} \rfloor$ queries where the pair says they are both telling the truth, then the chosen person always tells the truth. Otherwise, the chosen person bluffs.

(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 $x$).

*Answer.* Here is the pseudocode:

```
def query(p0, p1):
    p0_response <- response(p0, p1, {'T', 'F'})
    p1_response <- response(p1, p0, {'T', 'F'})
    return (p0_response, p1_response)

def is_truth(x, players):
    count <- 0
    for (p in players):
        if x != p and query(x, p) == ('T', 'T'):
            count += 1
    return count >= (len(players) - 1) // 2

def get_truth_p(players):
    if len(players) == 1:
        return players[0]

    num_players <- len(players)
```

```
truth_p0 <- get_truth_p(players[0:num_players])
truth_p1 <- get_truth_p(players[num_players:])

if is_truth(truth_p0, players):
    return truth_p0
else:
    return truth_p1
```

*Proof.* Because there are more truth tellers than bluffers, one subarray of size $n/2$ from the original array of size $n$ will have more truth tellers than bluffers, otherwise, we are at a contradiction.

This means that there exists a chain $C_0, C_1, C_2, \cdots, C_d$ such that $C_i$ is a subarray of half size of $C_{i-1}$, and that they all satisfy the condition that there are more truth tellers than bluffers, $|C_d| = 1$.

We will show that at every depth, a truth teller from chain $C_i$ will be returned from the function $get\_truth\_p(players)$. If $i = d$, then there is a 1 player group, therefore, the function returns the truth teller. Now for $i = k < d$, assume that the previous $i = k+1, k+2, \dots, d$ layers returned a truth teller. So either $truth\_p0$ or $truth\_p1$ is a truth teller. But since $C_k$ has more truth tellers than bluffers, the $is\_truth$ function accurately determines if the player always tells the truth or bluffs. Then a truth teller is guaranteed to be returned by the conditional statement at $is\_truth(truth\_p0, players)$.

This completes the induction proof. □

**Runtime Analysis**: We split the array in half at each layer of the recursion. Then the height of the recursion tree is $\log_2(n)$. Now to count the amount of work at each layer. At depth $d$, there are at most $2^d$ nodes of size $\leqslant n/2^d$. When we propagate up the recursion tree, each node will do $2|node|$ amount of work running the $O(n)$ queries on the two returned players from the previous subnodes. This means that each node at depth $d$ does at most $n/2^d$ work and there are at most $2^d$ nodes. So the total runtime is $2^d \cdot n/2^d = n$. With $O(n)$ time for each layer and $\log_2(n)$ layers, we get a total runtime complexity of $O(n \log n)$.

**Give a 3-part solution.**

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

**Give a 3-part solution.**