Answer Set 3

Kazi Samin Mubasshir
Email: kmubassh@purdue.edu
PUID: 34674350

Problem 1

Contributors: Farhanaz Farheen, Tania Chakraborty, Devin Atilla Ersoy
Derivation of a General Recurrence Relation: Median-of-Medians Quickselect's run-time is based on the three things it does:

1. Call itself recursively on the 'medians' list of size $ceil(\frac{n}{2k+1})$ to find the median-of-medians $\mathbf{M}$
2. $c \cdot n$ work to group items, partition the list around $\mathbf{M}$, and find the median of each small group-of-$2k+1$, and
3. Call itself recursively on either the partition with all elements less than $\mathbf{M}$, the partition with all elements greater than $\mathbf{M}$, or immediately return.

Ignoring the ceil and an additive $O(1)$ constant, this gives $T(n) = T(\frac{n}{2k+1}) + T(\textit{New Partition Size}) + cn$. What's the largest the New Partition Size can be? It's

$$max(\#\textit{elements less than } \mathbf{M},\ \#\ \textit{elements greater than } \mathbf{M})$$

Half of the $\frac{n}{2k+1}$ groups have medians less than M (so $\frac{(2k+1)-1}{2}$, plus 1 for the median, elements in that group are less than $\mathbf{M}$), and half had medians greater than $\mathbf{M}$ (again, giving $\frac{(2k+1)+1}{2}$ 'greater than $\mathbf{M}$' elements for each such group).

So, the general recurrence relation for an odd number would be

$$T(n) = T(\frac{n}{2k+1}) + T(1 - (\frac{n}{2 \cdot (2k+1)} \cdot \frac{(2k+1)+1}{2})) + cn$$

Time Complexity for different values of k: The median-of-medians divides a list into sublists of length five to get an optimal running time. Finding the median of small lists by brute force (sorting) takes a small amount of time, so the length of the sublists must be fairly small. However, adjusting the sublist size to three, for example, does change the running time for the worse.

If the algorithm divided the list into sublists of length three, p would be greater than approximately $\frac{n}{3}$ elements and it would be smaller than approximately $\frac{n}{3}$ elements. This would cause a worst case $\frac{2n}{3}$ recursions, yielding the recurrence $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + O(n)$, which by the master theorem is $O(n \log n)$, which is slower than linear time.

In fact, for any recurrence of the form $T(n) \leq T(an) + T(bn) + cn$, if $a + b < 1$, the recurrence will solve to $O(n)$, and if $a + b > 1$, the recurrence is usually equal to $\Omega(n \log n)$.

Proof:

$$T(n) = T(an) + T(bn) + O(n)$$

$$T(n) \leq T(an) + T(bn) + cn$$

$$T(n) \leq \alpha(an + bn) + cn$$

$$T(n) \leq \alpha n(a + b) + cn$$

$$T(n) \leq \alpha n + cn - \alpha n(1 - (a + b))$$

$T(n) \leq \alpha n$ if $cn - \alpha n(1 - (a + b)) \leq 0$.

If $(a + b) \geq 1$ then the term $-\alpha n(1 - (a + b))$ becomes positive.

If $a + b < 1$ then, $cn \leq \alpha n(1 - (a + b))$, or

$$\alpha \geq \frac{cn}{n - n(a + b)} \tag{1}$$

From Equation 1, we can see that for $a + b < 1$ we get a linear time solution. Otherwise it is not solvable in linear time. This concludes the proof.

The median-of-medians algorithm could use a sublist size greater than 5—for example, $7, 9, 11, \cdots$—and maintain a linear running time. However, we need to keep the sublist size as small as we can so that sorting the sublists can be done in what is effectively constant time.

Problem 2

Contributors: Farhanaz Farheen
An algorithm to optimize payment would be to *sort* the efforts in non-increasing order and designing the logos that take more efforts first and the logos with less efforts later. A sketch of the algorithm is given below.

Algorithm 1 An algorithm to optimize payment
```
1: $efforts \leftarrow e_0, e_1, e_2, \cdots, e_k$
2: $logos \leftarrow l_0, l_1, l_2, \cdots, l_k$
3: $payment \leftarrow 0$
4: $sorted\_index \leftarrow sort(efforts, decreasing = True)$
5: for $j = 0$ to $k - 1$ do
6:     design logos[sorted_index[j]]
7:     $payment \leftarrow payment + efforts[sorted\_index[j]] \cdot (k - j)$
8: end for
```

Correctness:
Two natural guesses for a good sequence would be to sort the e_i in decreasing order, or to sort them in increasing order. Faced with alternatives like this, it's perfectly reasonable to work out a small example and see if the example eliminates at least one of them. Here we could try $e_1 = 2$, $e_2 = 3$, and $e_3 = 4$ for $k = 3$. Designing the logos in increasing order results in a total payment of

$$2 \cdot (3 - 0) + 3 \cdot (3 - 1) + 4 \cdot (3 - 2) = 16$$

while designing them in decreasing order results in a total payment of

$$4 \cdot (3 - 0) + 3 \cdot (3 - 1) + 2 \cdot (3 - 2) = 20$$

This tells us that increasing order is not the way to go. (On the other hand, it doesn't tell us immediately that decreasing order is the right answer, but our goal was just to eliminate one of the two options.)

Let's suppose that there is an optimal solution O that differs from our solution S (In other words, S consists of the efforts sorted in decreasing order). So this optimal solution O must contain an inversion—that is, there must exist two neighboring days j and $j + 1$ such that the payment in day j for a logo with effort e_j is more than that in day $j + 1$.

We claim that by exchanging these two logo design choices, we can strictly improve our optimal solution, which contradicts the assumption that O was optimal. Therefore if we succeed in showing this, we will successfully show that our algorithm is indeed the correct one.

Notice that if we swap these two design choices, the rest of the designs are identically paid. In O, the amount paid during the two choices involved in the swap is $e_j \cdot (k - j) + e_{j+1} \cdot (k - j + 1)$. On the other hand, if we swapped these two choices, we would be paid $e_{j+1} \cdot (k - j) + e_j \cdot (k - j + 1)$. We want to show that the second one is more than the first one. So we want to show that

$$e_{j+1} \cdot (k - j) + e_j \cdot (k - j + 1) > e_j \cdot (k - j) + e_{j+1} \cdot (k - j + 1)$$

$$e_j \cdot (k - j + 1) - e_j \cdot (k - j) > e_{j+1} \cdot (k - j + 1) - e_{j+1} \cdot (k - j)$$

$$e_j \cdot (k - j + 1 - k + j) > e_{j+1} \cdot (k - j + 1 - k + j)$$

$$e_j > e_{j+1}$$

But this last inequality is true simply by construction as we sorted them in decreasing order. This concludes the proof of correctness.

Complexity:

Line 4 of the algorithm(sorting operation) takes $O(k \log k)$ time and the for loop from $line$ $5-8$ takes $O(k)$ time. So the overall time complexity of the algorithm is $O(k \log k)$.