CMPS 102 – Winter Quarter 2017 – Homework 1

Christopher Hsiao – chhsiao@ucsc.edu – 1398305
Homework Heavy

Solution to Problem 1 - Is it a Knock-out?

Q: You are given a pile of envelopes, each containing a vote for a candidate, and a machine that can tell you if two envelopes contain votes for the same candidate or not without opening them. Your task is to design an algorithm which by using the machine $O(nlogn)$ times, decides if a second round is needed or not. You are not allowed to open any envelopes. If the answer is that no second round is needed, your algorithm should also offer an envelope containing a vote for the winner.

Claim 1. *Given a pile of votes, I will be able to recursively find and return the envelope of a majority candidate in that pile recursively (assuming one exists). Note that this not guarantee anything with regards to the winning candidate condition of:*

$$\textit{number of votes for any candidate} \geq \left\lfloor \frac{n}{2} \right\rfloor + 1$$

Proof. To prove **Claim 1**, we need to first observe out problem as a recursive tree.

Divide: For our function $f(n)$, we divide our input n by 2 per recursive level, halving the size of the input and creating two smaller problems each time. This makes the recursive relations properties:

$$a = 2, b = 2, \text{and } k \leq (log_2 n) - 1$$

This yields, at the lowest level, $n/2$ problems, each of size 2, $a.k.a$ each problem is at least a pair of votes.

Conquer: Now, we have set the stage to "conquer" the problem, by running an algorithm on the smaller problems and returning (in this case) an envelope containing a vote. We can view each sub problem as a mini-election. The role of each "minion", in this case, is to attempt to return an envelope containing the majority vote for its respective mini-election. Each minion only ever looks at two envelopes, which, aside from the lowest level, will be two envelopes handed to it by its own minions.

More formally, minions on each level (aside from the $k = (log_2 n) - 1$ level) return an envelope if any candidate has $\left\lfloor \frac{n}{2^k} \right\rfloor$ votes.
The combination algorithm is as follows:

Algorithm: From our perspective at the top of the recursive tree, we will potentially be handed up an envelope from the left sub-tree (represented as L), and an envelope from the right sub-tree (represented as R). If no envelope is handed up from either half, L or R = null.
Using the machine, first observe there are four outcomes of the recursion. We will detail how to handle these cases more generally later.

1. $L = null, R = null$ - no envelope passed up
   - If no envelope is passed up, then no candidate won either half of the tree, meaning they received less than $\left\lfloor \frac{n}{4} \right\rfloor + 1$ votes in their respective halves, and no candidate could possibly win. Second round is necessary.

2. $L = 1, R = null$ - one envelope is passed up. note this represents one envelope passed up. This scenario is handled the same way as $L = 0, R = 1$.
   - If one envelope is passed up, then that is the only candidate who has a potential to win the election. Cross check that envelope against the entire array of envelopes in $O(n)$, and count the number of $true's$ output by the machine. If that count $\geq \lfloor \frac{n}{2} \rfloor + 1$, then that candidate has won. Otherwise, no candidate has won, and a second round is necessary.

3. $L = 1, R = 2$ - an envelope is passed up by both sub-trees, but for different candidates.
   - Do the same cross checking method above in $O(n)$ for each envelope, and if the count for either envelope is $\geq \lfloor \frac{n}{2} \rfloor + 1$, then that candidate won. If neither candidate has at least $\lfloor \frac{n}{2} \rfloor + 1$ votes, then second round is necessary.

4. $L = 1, R = 1$ - two envelopes for the same candidate are passed up by both sub-trees.
   - The candidate then has at least $\lfloor \frac{n}{4} \rfloor + 1$ votes, since they must have strictly more than half of their respective trees, so the candidate is guaranteed to win.

Now, when these cases are encountered in our recursion, here is how they are handled.

1. $L = null, R = null$ - no envelope passed up
   - return null.

2. $L = 1, R = null$ - one envelope is passed up. note this represents one envelope passed up. This scenario is handled the same way as $L = 0, R = 1$.
   - return that one envelope, since that candidate still has a chance of being the majority winner.

3. $L = 1, R = 2$ - an envelope is passed up by both sub-trees, but for different candidates.
   - you "cancel" these votes out, and return null.

4. $L = 1, R = 1$ - two envelopes for the same candidate are passed up by both sub-trees.
   - you return either envelope.

Correctness: The meat of this problem lies in the third case, where the machine returns false and returns neither envelope. The point of this algorithm isn't to figure out how many votes a winning candidate has (indeed, if one even exists), but to propagate a single vote up the recursive tree, and that envelope will only survive the trip to the top if that candidate has a majority in that sub-tree.

Consider the following case: there is a candidate A with $\lfloor \frac{n}{4} \rfloor + 1$ votes in the L sub-tree, and the remainder of the votes are distributed to an arbitrary number of candidates. At the k^{th} level, distribute the votes such that every single pair has at least one vote for candidate A. By the pigeonhole principle, candidate A is guaranteed at least one pair of envelopes containing votes for him, since there are $\frac{n}{4}$ pairs available in the L sub-tree at the lowest level. Since he is guaranteed at least one pair, this also means that every single other pair effectively cancels out, and no other envelopes are propagated upwards. The only envelope that gets propagated is an envelope in the pair with two votes for candidate A.

Now, let us consider the other cases. Say for example, instead of envelopes for candidate A being distributed equally amongst the available pairs, with only one containing two A votes, consider the case where

there are now more A envelope pairs. The trade off, now, is that we have opened up room for other candidates to potentially make pairs. However, we have already established that in the worst case, candidate A is guaranteed at least one pair, so for any extra pair that candidate A makes, no other candidate is able to make more pairs than candidate A. Candidate A will always at least have one more pair than any other candidate (assuming A has a majority).

This means, that no matter what, some candidate A with a majority of his subtree will at least have 1 pair more than any other candidate, and therefore will be propagated to the top of the subtree and eventually returned by the subtrees root node. Also, to handle for cases with an odd number of envelopes, observe that the condition we must meet to satisfy the subtrees is the floor of some number proportional to n. This means that we can simply remove the odd vote for the entirety of the recursive calls, and re-include it at the end, presuming that any envelopes make it that far. The reason we handle an odd number of envelopes so precisely, is because when there is a lone envelope, it disrupts the pigeonhole principle, by creating another "seat" for an envelope since at the lowest level envelopes come in pairs to be compared. □

As you can see, we can now successfully propagate an envelope containing a vote for a majority candidate through the recursive tree to the top of the sub-tree, and to us. Then, I provided an algorithm to handle all four possible cases at the top of the tree, which can successfully determine whether or not a second round is necessary.

The recurrence relation for this process is $T(n) \leq 2T(n/2) + O(n)$, which by the Master Theorem is yields a runtime of $O(nlogn)$.

Solution to Problem 2 - Coastal California

The amazing algorithmic skills you developed in 201 made you very rich, so that you are in the market for a waterfront lot on which to build a house. All such lots have three straight sides and one irregular side, due to the coastline. The irregular side consists of segments parallel to the road, each segment specified by its width and its distance from the road. Both numbers for each segment are **integers**.

Your dream is to build (i) a rectangular house (ii) with one side that touches the road, that is (iii) as big as possible (money can't buy taste). Design an algorithm which given a lot returns the area of your dream house in $O(nlogn)$ steps, where n is the integer equal to the lot's entire length along the road, i.e., the sum of the digths of the segments.

Algorithm: Divide and Conquer First we will rotate the image so that the road is on the bottom, and the land is facing upwards.

1. Scan the array and search for the smallest height.
2. Split the array at this point, and recursively find the maximum area of each side respectively.
3. Find the $\max$ between a.) the maximum rectangle in the left half of the array, b.) the maximum rectangle in the right half of the array, or c.) the smallest height $\times$ length of the array.

Proof. The reason we initially scan the array for the smallest height is so we can guarantee that when we search for a rectangle that spans both sides of the graph, one can exist. The lowest height also guarantees that we can at least create a rectangle that spans the entire graph.

The idea is to increment upon what the algorithm knows as the existing largest rectangle (in terms of the data its seen), and then compare it with the other pertinent areas at the end of the recursive calls.

At first glance, it may seem like the recursive relation is dependent on index of the smallest height, but this isn't actually the case, since we split by bars, every segment of the graph will need to be recursed on. This means that though the number of levels in the recursive tree may fluctuate, the number of sub-problems created will stay the same. As such, we can treat the problem as a traditional $T(n) \leq 2T(n/2) + O(n)$, which by the master theorem is a $O(nlogn)$ algorithm. □

Solution to Problem 3 - Save the Environment

You are working as a climate change analyst for the Cooperative Institute for Climate Science. Your main task is to study the variation in Chicago's temperature. Let C be an array containing the temperatures of the last n days. Your goal is to find the pair of days with the greatest temperature **increase**, i.e.,

$$\max_{1 \leq j < k \leq n} C[k] - C[j]. \tag{1}$$

If the temperature never increased, then report that. Give a $O(nlogn)$ algorithm for this task.

Claim 2. *We can find the maximum temperature increase (assuming the temperature did increase) across the mid point of the array where the left index j_c is found in the left half of the array and the right index k_c is found in the right half in O(n) time.*

Proof. The case of finding a max temperature pair across the center is straightforward. Observe that the max temperature value can be found by simply scanning for the smallest element of the left half, and pairing that up with the largest value of the right half, sequential property is guaranteed to be preserved. This way, we can state the largest pair across the center to be (j_c, k_c) such that

$$j_c = \min(C[1], C[2], ..., C[\lfloor \frac{n}{2} \rfloor]), k_c = \max(C[\lfloor \frac{n}{2} \rfloor + 1], ..., C[n]) \tag{2}$$

Now, if $C[j_c] > C[k_c]$, then we can set the values of j_c and k_c to 1, so that the max temperature is reported as 0, implying there is no temperature increase.

To scroll through and find $min(C[1], C[2], ..., C[\lfloor \frac{n}{2} \rfloor])$ and $max(C[\lfloor \frac{n}{2} \rfloor + 1], ..., C[n])$ are both $O(\lfloor \frac{n}{2} \rfloor)$ operations, so the runtime for locating the indices of the max temperature if one exists across the midpoint is $O(n)$. it is important to note this can be done in a modified MergeSort, where in the $Merge(L, R)$, we don't sort the values, but use that time to scroll through L and R and find the indices of the values we need. □

Claim 3. *Given some array C, we can find and return a pair of indices (j_l, k_l) and (j_r, k_r), each representing the maximum temperature increase (if one exists) in their respective halves of the array. Otherwise, their values will equal 1, which implies a maximum temperature increase of 0.*

Proof. At the highest level, our goal is to solve the problem by breaking it into two subproblems of size $\frac{n}{2}$. This yields a recurrence relation of:

$$T(n) \leq 2T(\frac{n}{2}) + O(n) \tag{3}$$

We will be using a modified MergeSort for this proof. For the array C, we will find a pair of indices from both the left $L[1...\frac{n}{2}]$ half and the right $R[\frac{n}{2} + 1...n]$ as (j_l, k_l) and (j_r, k_r). Our goal is to find the indices which correspond to the greatest profit across the left half, the right half, and across the midpoint of C.

After splitting up our original problem, at the lowest level we're left with potential pairings of dates on either side of the array. We only want to split to $n/2$ levels down, such that we can form pairs of indices at the lowest level. The only time where $n = 1$ is when there is an odd number of vertices on the half of the problem we're solving, such that one index ends up as a leaf node.

Algorithm: Modified Merge Sort

1. At some node, get the left and right arrays from the two subproblems of said node.

2. Get the max temperature increase from the left and right nodes.

3. Using our claim above to get the max temperature increase across both halves using the modified $Merge(L, R)$.

4. Compare $(j_l, k_l), (j_r, k_r)$, and (j_c, k_c) values.

5. Push up the pair of indices representing the greatest increase in temperature along with a new array the output of the modified $Merge(L, R)$.

□

As you can see, $(j_l, k_l), (j_r, k_r)$, are the values being passed up from the two subtrees of any non-leaf node in the recursive subtree, and with that information we can compare it with (j_c, k_c), which was determined in the merge step.

Since we're using a modified $MergeSort$, our recurrence relation as stated above is $T(n) \leq 2T(n/2) + O(n)$, which by the Master Theorem is yields a runtime of $O(nlogn)$.

Solution to Problem 4 - Dive Bar

1. Give a simple $O(n^2)$ algorithm for the slot machine problem given a $O(n)$ time algorithm for l-LMONEY.
Algorithm: $O(n^2)$

1. Call l-Money on all indices. Note that this is already an $O(n^2)$ process, since we're calling an $O(n)$ algorithm on n indices.

2. Loop over the indices again, this time to calculate and determine which actually yielded the greatest return. Note that this adds $O(n^2)$ to our runtime, which gets absorbed into the existing $O(n^2)$ runtime anyway.

Proof. On our first pass through, we first determine the optimal stopping index for each index in the array. Then, it is simply a matter of actually calculating the greatest return based on those previous calculations, which clearly works. □