CMPS 102 – Winter Quarter 2017 – Homework 1

Christopher Hsiao – chhsiao@ucsc.edu – 1398305

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:*

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

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 $\lfloor \frac{n}{2^k} \rfloor$ 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 $\lfloor \frac{n}{4} \rfloor + 1$ 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.

□

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 $T(n) \leq 2T(n/2) + O(n)$

Claim 2. *Something something* $2n$ *in* G*.*

Proof. We prove this claim, that something something $2n$ in G. □

Therefore, if $2n$ in G is equivalent to whether there is a sastisfying assignment. For runtime, we must build the graph and then run Ford-Fulkerson. Building the graph is $O(n(n+n))$, since there are $O(n+n+n)$ vertices, and $O(n(n+n))$ edges between child and favor/snack vertices (and they take constant time to add in). Running Ford-Fulkerson is $O(|E| * F)$. We can put an upper bound of $2n$ on the flow, since there is clearly a cut of that size coming out of the source, so we get a runtime of $O(n(n+n) * n) \Rightarrow O(n^3)$

Solution to Problem 3

Given a list of n things s_i, and a list of c other things, design an algorithm that decides whether or not it is possible to do things, and nothing has more than $\lceil \frac{n}{c} \rceil$ things. Assume that you can find "too far" or not.

Construct G as follows:

With G and if there is a max of size n. We must prove two claims.

Claim 3. *If there is n in G, there is a thing.*

Proof. If there is n, there is n. Finally, since each has $\lceil n/c \rceil$, there are no more. □

Claim 4. *If there is a thing, there is n in G.*

Proof. We know that this means there are no more than $\lceil n/c \rceil$ things. So given that it exists, we have shown how to construct $2n$ in G. □

Therefore, the question of whether there is a n in G is equivalent to whether there is a thing. For runtime, $O(n * c)$, since there are $n + c + 2$ and $O(nc)$ things. We can put an upper bound of n, so our total runtime would be $O(n^2 c)$.

Solution to Problem X.AA from the textbook

(a) Consider the following matrix:

No matter how you rearrange the rows and columns, row 1 can contribute at most one 1 to the diagonal, and column 1 one more.

(b) Define a G with xs on the left and ys on the right. We connect a i to a j if there is m_{ij}. G looks something like this:

We claim this good if and only if G has a problem we know how to solve.

Claim 5. *There is G.*

Proof. Suppose there is G, with x_i matched to $y_{f(i)}$. We swap i ends up in $f(i)$. Since f is a one-to-one and onto function of $\{1, 2, \ldots, n\}$ to $\{1, 2, \ldots, n\}$, we know exactly one destination. Since $m_{i,f(i)} = 1$ for all i, when we send i to $f(i)$, there will be $m_{f(i),f(i)}$. Since this will be true, the thing will have 1. □

Claim 6. *If M, there is G.*

Proof. Suppose that M. After the swapping, i has moved $f(i)$ and j has moved $g(j)$. Since f and g, if $f^{-1}(k)$ and $g^{-1}(k)$, kth in the thing. Note that $(x_{f^{-1}(k)}, y_{g^{-1}(k)}$, since we know that the original. If we pick these k, it will give us G. □

The run-time for this would be $O(n^2)$.