Homework One

Adam Paz

January 25th

1. (a) **Algorithm**:For this function assume that the pile of ballots can be accessed as a global element such as a global array that can be accessed from any point in the function, with the $INDEX$ 0 being a sentinel. The function is called FindMajority(Z). This function will have an input of n votes. If there is one vote, return it's index. If not we split the votes into two even groups. The group containing every element up to the middle element will be group L. The rest of the votes will be group R. We compare FindMajority(L) with FindMajority(R). There are several cases to take into consideration.

   i. If they both return a non 0 $INDEX$ we compare the ballots at the two idices. If they are different we return a 0. If they are the same return either one of their indices.

   ii. If both return a 0 we return a 0.

   iii. If one passes a non-zero and one passes a 0 we return the $INDEX$ of the non-zero.

   At the end of the Algorithm you will either be passed an index, in which case the last two items compared are the same, or a 0 in which the last two items passed are different. If a non-zero $INDEX$ is passed take the ballot at this $INDEX$ and compare it with the original array. Keep a counter until there are n/2+1 comparisons that returned true, if this is the case there is no need for a runoff. If the counter does not reach n/2+1 a run off is needed. If a 0 is returned take the last two ballots that were compared, and compare each with the entire original array. Keep a counter for how many return true for each. If a counter ends up to n/2+1 then there is no run off needed, if not then a runoff is needed.

   Proof of correctness For this function there is a pile of n ballots that will be brought down to individual piles of 1 through recursion, in which no comparison is needed. From there they are brought into groups of two, the two are compared and if there is a pair of the same vote then the $INDEX$ of one is passed on representing a pair. The reason this algorithm works is because to win and not need a run off you must have n/2+1 of the votes. In every case of there being n/2+1 votes for one candidate there will be a pair of votes for this candidate and for every other pair that this candidate is not present there is another pair for this candidate. It then follows that the $INDEX$ for one of these ballots will be passed on. If the 0 is returned at the end the last two ballots that were compared will be compared with all other votes each counted, or it is the only ballot passed on in which it will be compared with all other ballots. Since it is compared with the entire array at the end the ballot that won will surely reach a counter of n/2+1. In the case that there is no winner and a runoff is needed in the last comparison done with the entire array the counter will never reach n/2+1.

Proof of run-time The input runs in time nlogn, where n is how many voting ballots are in the pile. (2T(n/2) + d) is the recurrence relation for findMajority and by the master theorem is in O(logn) then the final comparison of the ballot or ballots recieved at the end with the rest of the array is O(n). Therefore the total run time is O(nlogn).

(b) **Arbitrary N** For an arbitrary n the only difference that needs to be maid is that the two groups that are split instead of being even can vary by one when there are three to be compared. Splitting the first two ballots and comparing the second ballot with the third. Throughout different points in the recursion there will be times where there are some pairs and some groups of one. In the cases that there is a group of one return it's $INDEX$ representing a pair and the algorithm will work.

Proof of correctness Similar to above proof with the addition that since there is not an exact even value of 2^k there are occasions in which ballots stand alone, in which case they are treated as pairs. This raises the question if these are treated as pairs and some of these are not the candidate that should win how will the winning ballots trump this candidate that should not win but instead has an advantage due to stand alone votes. In this case the returned number at the end of the recursion will be a 0. For the candidate with n/2+1 of the votes will trump half of the votes and both ballots will be compared with the entire array. Therefore, if no runoff is needed this will be the conclusion.

Proof of run-time Same as proof of runtime for $n = 2^k$. Varries only slightly and will be picked up by the constant term in the definition O(nlogn), which is a constant C * (nlogn).

2. (a) **Algorithm**: Lets say our array of values for win or loss amounts is called A[ ]. Take the number at the $INDEX$ L, for given L. Then let's use the variables MAX and $INDEX$ and set $INDEX$ to L and MAX to A[L]. Then take the number A[L] and add it to A[L+1] compare this number with the current MAX. Do this with all subsequent numbers until the last number in the array. If at any point the number compared with MAX is larger, set MAX to this number and set $INDEX$ to the $INDEX$ that this larger number was created at.at the end of the array return INDEX. For Rgain it is similar but starts at the largest $INDEX$ and moves downward toward an $INDEX$ of 0.

Proof of correctness Since the algorithm must start at given L or R it then moves from this point and compares every possible addition in the rest of the array to whatever the highest addition has gotten to until that point. It takes the highest from the entire possible additions from this point all the way to the end of the array and returns that index.

proof of run time Since it compares each addition to the MAX addition it is therefore n-1 comparisons plus the 1 comparison for the original MAX established as the first number in the array making it n comparisons. It is therefore O(n).

(b) **Algorithm**: Given the O(n) time complexity for lgain and rgain, the simple algorithm that runs in O(n^2) is to run lgain from the right most $INDEX$ store the number returned from that as $TMAX$, and then do the same thing for every other $INDEX$ in the array and compare each to $TMAX$, setting $TMAX$ to the highest returned number and from which $INDEX$ the highest returned number was started on and ended on. Such as the L for which lgain starts and whatever $INDEX$ in the array that lead to the highest returned value. Similar for rgain, but starting from the smallest $INDEX$ and moving upwards towards the last index.

Proof of correctness Since it has been proven that lgain and rgain return the largest possible streak from given L or R, trying it at every $INDEX$ will surely give you the largest possible profit to start from and stop at.

Proof of runtime an O(n) algorithm is run on n items, it is therefore O(n^2). It takes n * n comparisons which is n^2 comparisons.

(c) I intentionally left this blank, I made progress but could never figure out a full solution.

3. (a) **Proof of correctness** Assuming that this algorithm will work for an input of size n<= 3, for if there is one element it is always sorted, if there are two elements they will be compared and sorted. If there are three elements the first two will be put in order so that the 1st element could not be the largest since there is one larger than it. So one of the 2nd two must be the largest element of the array. once these are sorted the largest will be in the last index, and in the final sort the first two will be sorted, and therefore all 3 indices of the array will be in correct order. For input of size n:

In the first sort, the first 2/3 of the array is sorted so that the smaller half of the elements from the first 2/3 is put into the first 1/3 of the total array, and the larger 1/2 of the elements in this part of the array are shifted into the middle 1/3 of the total array. It follows that there is no possibility for the elements in the first 1/3 of the array to be needed in the last 1/3 of the array because they are already smaller than the middle 1/3 of the array.

In the second sort, the last 2/3 of the array is sorted, which had the largest 1/3 of the elements. when the sort finishes that largest 1/3 of the elements of the total array will be put into the last 1/3 of the array. Once the largest 1/3 of the elements is in the top 1/3 of the array, it follows that the first 2/3 of the array will then be sorted and therefore the whole array will be sorted.

(b) **Master Theorem** $T(n) = 3T(2N/3)$ + O(1). It is big O(1) because at the smallest recursive call n<=3 is just comparing two elements. Through the master theorem and the formula $n^{log_b(a)}$ I found that $log_{3/2}3$ = to roughly 2.71. Therefore it is in O($n^{(}2.71$))

4. (a) **Algorithm for** Log_2n: Our algorithm will be called findLighter(Z). Z is a pile of n coins, take the n coins if Z is an odd amount of coins take one coin out. Split the n or N-1 coins into two piles of the same size lets call them A and B. Compare their weights. If they are the same weight, then the coin that was taken out from the pile for Z being an odd amount of coins is the lighter coin, return it. If A weighs less than B do findLighter(A), if B is lighter than A do findLight(B).

Proof of Correctness There is one coin that is lighter than the rest, so any two groups of the same number of coins can only weigh the same amount if the lighter coin is not present. It follows that there are two cases.

i. In the case where there is an even amount of coins, when the two piles are measured one will be lighter because the lighter coin is present, no coins have been removed from the larger pile that contained the lighter coin. When the lighter pile is picked it now takes the place of the larger pile.

ii. In the case where there are an odd amount of coins, one coin is taken out and the rest of the coins are split into two piles and measured. In the case that they are the same weight the single outlying coin must be the lighter one because there

is only one lighter coin, and if it were in one of the piles the piles could not be an even weight. If the piles are split and they weigh different amounts, the coin taken out could not be the lighter coin, and the lighter pile must have the lighter coin and is therefore picked to go through the recursive process again.

Proof of Timing The algorithm runs in $log_2 n$ time, where n is how many coins there are. Since each run of the algorithm cuts the amount of coins measured in the next recursion by at least half(half -1 in cases where there is an odd amount of coins). The size of the input of each sub problem gets continuously smaller as the amount of sub problems remain equal. So the number of times the input must be halved to get from n to 2 is $log_2 n$. And we need the +1 to do the last comparison of the final two elements. T(n/2)+d= C($log_2 n$) where C = 1 and d =1.

(b) **Algorithm for** $Log_3 n$: Our algorithm will be called findLighter(Z). Z is a pile of n coins, take the n coins and split them into three piles. Two of which are the same size (A and B)and one of which can be a different size by a factor of one(C). Take the two piles of the same size and compare their weights. If they weigh the same amount call findLighter(C). If A is lighter than B call findLighter(A) if B is lighter than A call findLigher(B).

Proof of Correctness There is only one coin that is lighter than the rest, so any two groups of the same number of coins will weigh the same if the lighter coin is not present, or if the lighter coin is present they must weigh a different amount. When the larger pile is split into three piles there is always a way to make two weigh the same amount and the third pile only differ by one. Either it is a multiple of three (3n+0, in which case it splits perfectly), or it 3n+1 in which, the extra one can go into the third non-even pile. Or it is 3n+2, in which case the third pile could have one less coin than the two even piles.There are two cases in this algorithm when measuring the two even piles.

i. In the first case the two piles with an even amount of coins have the same weight. This means that the third, outlying pile must have the lighter coin, because no coins have been removed and the other two piles cannot have the coin. This third pile is then treated as the larger pile and goes through the recursive process.

ii. In the second case the two piles with the same amount of coins are weighed and have different weights. This means one of the two piles definitely has the lighter coin. Take this pile and repeat the process until the lighter coin can be isolated.

Proof of Timing The algorithm runs in $log_3 n$ time, where n is how many coins there are. Since each run of the algorithm cuts the amount of coins measured in the next recursion by 1/3 plus or minus 1. The size of the input of each sub problem gets continuously smaller as the amount of sub problems remain equal. So the number of times the input must be cut into a third to get from n to 2 or 3 is $log_3 n$. And we need the +1 to do the last comparison of the final two or three elements. T(n/3)+d = C($log_3 n$) where C = 1 and d = 1.