Jake Tantorski
Assignment 3

Problem 1

i) The idea behind my algorithm was to use two different functions in order to find the first index in which x occurs and also find the last index that x occurs. From there, I am able to quickly calculate the number of times it occurs by taking the last index and subtracting the first index and adding 1.

ii)

```
count(double[] A, double x, double n)
        Double first = firstIndex(A[],0, int n-1,x,n)                   //1
        if(first = -1)                                                  //1
                Return first
        End if
        Double last = lastIndex(A[],0, int n-1,x,n)                     //1
        Double total = last-first+1;                                    //1
        Return total

firstIndex(A[],low,high,x,n)
        if(high >= low)                                                 //logn
                Int mid = (low+high)/2                                  //1
                if((mid = 0 OR x > A[mid])AND(A[mid] = x)               //3
                        Return mid
                Else if(x>A[mid])                                       //1
                        Return firstIndex(A, mid+1,high,x,n)
                Else
                        Return firstIndex(A, low,mid-1,x,n)             //1
                End if
           Return -1
        End if

lastIndex(A[],low,high,x,n)
        if(high >= low)                                                 //logn
                Int mid = (low+high)/2                                  //1
                if((mid = n-1 OR x < A[mid])AND(A[mid] = x)             //3
                        Return mid
                Else if(x > A[mid])                                     //1
                        Return firstIndex(A, mid+1,high,x,n)            //1
                Else
                        Return firstIndex(A, low,mid-1,x,n)             //1
                End if
```

```
    Return -1
End if
```

iii) The run time of the algorithm is O(logn) due to the fact that the two functions to locate the first and last index is using binary search and the rest and the program runs at constant time, therefore, the highest order would be O(logn)

Problem 2

i) The idea behind this algorithm is that if you take the pile of coins and split it up into two different groups and compare the two you will be able to see which pile is heavier considering the rest of the coins would weigh the same. From there you can recursively call the compare function to again split the heavier pile in half due to the fact that you know the coin would be in that pile. Keep dividing the piles until the heaviest pile has one coin remaining which would be the coin you are looking for.

ii) The running time for this algorithm would be O(logn) due to the fact that the compare function uses a recursive binary in order to find the coin. Using the formula n/2^k-1 you can see that k = log(n)

Problem 3

The run time of this binary search would take O(n) time. This is because it takes n/2 moves, therefore, $n(½+(½)^2+(½)^3+...+)$ resulting in $n((½)^{n+1}-1)/(½-1)-1) = n$ moves therefore it would be in O(n)

Problem 4

```
squareroot(x)
Input X          //2
Lower = 1        //2
Upper = x        //2
Ans = 0          //2
while(lower<=upper)    //logn
        Mid = (lower+upper)/2
        if(mid*mid == x)          //2
                Return mid
        Else if(mid*mid < x)     //2
                lower = mid + 1
                ans = mid
        Else
                Upper = mid - 1
        End if
```

End while
Return ans

Time: O(log(n)) because it only uses binary search

The idea behind this algorithm is to use a binary search to find the square root. In the beginning, the while loop makes it so the loop can continue. From there the x value is compared and depending on where it is the search end or start changes to make the search more precise.