CSE321 – Introduction to Algorithm Design – Homework 3 Report

161044042 – Yasir Nacak

Q1.

In this part, we had an array of boxes with the size 2N. I designed my algorithm such that, to solve the problem with 2N case, we need to solve the subproblem for 2N-2 case to solve that, we need to solve the 2N-2-2 case and so forth. Each subproblem takes the array with its leftmost and rightmost elements trimmed with a parameter whether to swap or not. On the first run, I defaulted the swap value to False, so the first box always stays as black and the last box always stays as white, after that I alternated the swap parameter so at each step, we alternate between swapping and not-swapping. This way, I created an array of boxes alternating from my input. This solution performs N-2 interchange operations for even values of N and N-3 interchange operations for odd values of N. Average running time is: O(N) or O(n//2) where n is the total number of boxes and N is the count of black or white colored boxes.

Example Runs:

```
Question 1:
Test Case #1
Initial List:           ['B', 'W']
Expected List:          ['B', 'W']
Calculated List:        ['B', 'W']

Test Case #2
Initial List:           ['B', 'B', 'B', 'B', 'B', 'W', 'W', 'W', 'W', 'W']
Expected List:          ['B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W']
Calculated List:        ['B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W']

Test Case #3
Initial List:           ['B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W']
Expected List:          ['B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W']
Calculated List:        ['B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W']

Test Case #4
Initial List:           ['B', 'B', 'B', 'B', 'B', 'B', 'B', 'W', 'W', 'W', 'W', 'W', 'W', 'W']
Expected List:          ['B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W']
Calculated List:        ['B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W']

Test Case #5
Initial List:           ['B', 'B', 'B', 'W', 'W', 'W']
Expected List:          ['B', 'W', 'B', 'W', 'B', 'W']
Calculated List:        ['B', 'W', 'B', 'W', 'B', 'W']

Test Case #6
Initial List:           ['B', 'B', 'B', 'B', 'W', 'W', 'W', 'W']
Expected List:          ['B', 'W', 'B', 'W', 'B', 'W', 'B', 'W']
Calculated List:        ['B', 'W', 'B', 'W', 'B', 'W', 'B', 'W']

Test Case #7
Initial List:           ['B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W']
Expected List:          ['B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W']
Calculated List:        ['B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W', 'B', 'W']
```

Q2.

To solve this problem, I decreased my input size by a constant factor which is 2. This means that I split given coins to 2 groups containing n/2 coins each. Then I weighed them to pick the lighter pile. But there is a trick in this. If the pile has odd number of coins, I separate one coin from the pile, split the remaining into 2 piles and weigh them. If they weigh equally, then the separated coin is the fake one. Otherwise, I continue to solve the problem with one of the piles. This overall solution creates a binary decision tree. Best case occurs when we have odd number of coins where the last coin is the fake one. In that case the algorithm only runs once and exits after finding out two piles weigh the same and the separated coin is the fake one. Otherwise, we have a binary search tree where we need to reach leaf. Shortest path from root to leaf is O(log n) in every case.

Example Runs:

```
Question 2:
Test Case #1
Coins: [2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
Expected Fake Coin Index: 5
Calculated Fake Coin Index: 5

Test Case #2
Coins: [2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
Expected Fake Coin Index: 4
Calculated Fake Coin Index: 4

Test Case #3
Coins: [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
Expected Fake Coin Index: 12
Calculated Fake Coin Index: 12

Test Case #4
Coins: [2, 2, 2, 1]
Expected Fake Coin Index: 3
Calculated Fake Coin Index: 3

Test Case #5
Coins: [1, 2, 2, 2, 2]
Expected Fake Coin Index: 0
Calculated Fake Coin Index: 0

Test Case #6
Coins: [2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
Expected Fake Coin Index: 4
Calculated Fake Coin Index: 4

Test Case #7
Coins: [2, 2, 2, 2, 2, 2, 2, 1, 2, 2]
Expected Fake Coin Index: 7
Calculated Fake Coin Index: 7
```

Q3.

I implemented both insertion sort and quicksort. In my implementation Lomuto Partition Scheme like we learned in the class with using the *low* indexed element as my pivot. Average case running time of insertion sort depends on the inversions of the given array. Because we need to do as many swaps as the inversions in the array. In an inverse sorted array, number of inversions is the number of elements to the power of 2 since every element is inversed with every other element on the list so the complexity is O(n^2). This applies to both the average and worst case of the insertion sort. For the quicksort, at the worst case, we need to do N times work to partition + 2 recursive calls for each of the partitions. Which gives us T(n) = O(n) + T(0) + T(n-1) which shows the case where one partition has no elements and other partition has all other elements except for the pivot. This gives us a O(n^2) complexity. On average, quicksort works closer to its best case which is the balanced partition. Which gives us T(n) = O(n) + T(n/2) + T(n/2) which shows the case where each partition has the same number of elements. This gives us O(n logn) complexity.

Example Runs with Swap Counts:

List = [10, 16, 23, 52, 61, 77, 92, 126, 140, 181]

For the runtime test, I reverse sorted this array for my quicksort algorithm because I choose my pivot as the leftmost element.

Quicksort does: 35 swap operations

Insertion Sort does: 0 swap operations

This can also be confirmed when looking at the formal proof where it says quicksort does the worst when the array is already sorted and insertion does the best.

List = [23, 92, 140, 10, 52, 16, 181, 126, 77, 10]

This is a scramble of the previously given list.

Quicksort does: 16 swap operations

Insertion Sort does: 23 swap operations

This is expected since on average cases, quicksort works faster than insertion sort.

List = [181, 140, 126, 92, 77, 61, 52, 23, 16, 10]

This is the reverse sorted version of the previously given list. I also gave the reverse of this to my quicksort algorithm because of my pivot selection.

Quicksort does: 9 swap operations

Insertion Sort does: 45 swap operations

This is also expected because this is the worst case of insertion sort where it needs to do maximum number of swaps. This is also the best case of quicksort.

Q4.

To find out the median of a given array, I used the decrease and conquer method we discussed in our lectures. This method was to partition the array on a randomly selected pivot, checking if the pivot ended up at the middle index, and choosing to continue left or right of the pivot based on the pivot's position. Pseudocode is as follows:

```
procedure Median(L[l...r], k):
        pivot <- L[l]
        pivot_arranged = partition(L, l, r, pivot)

        if pivot_arranged is k - 1, return L[pivot_arranged]
        else if pivot_arranged > l + k - 1
                Median(A[l...pivot_arranged-1], k)
        else
                Median(A[pivot_arranged+1...r], k-1-pivot_arranged)
        end if
end procedure
```

I also utilized Lomuto Partition that was implemented in the previous part. Since we always split the list into two parts and keep looking for the answer in one of the parts, this creates a binary tree where each node represents its parent's left or right part after partition. I didn't choose the pivot at random. Instead I chose it as the current first element of the list. So, if we give this algorithm a list of numbers that are reversely sorted, we get O(n^2) complexity which is the worst case of this algorithm.

Example Runs:

```
QUESTION 4:
Test Case #1
List: [24, 31, 0, 18, 29, 69, 46, 7, 42, 24, 45, 19, 12, 49, 53, 91, 38, 86, 91, 56, 9, 5, 15, 64, 97, 66, 48, 13, 39, 70, 43, 36, 56, 38, 78]
Expected Median: 42
Calculated Median: 42

Test Case #2
List: [76, 6, 19, 98, 9, 30, 71, 40, 7, 53, 59]
Expected Median: 40
Calculated Median: 40

Test Case #3
List: [19, 95, 97, 46, 78, 41, 66, 16, 6, 26, 58, 72, 87, 53, 5, 22, 86, 2, 38, 25, 2, 40, 55, 60, 64, 7, 1, 8, 87, 35, 93, 53, 88, 14, 79, 64, 18]
Expected Median: 46
Calculated Median: 46

Test Case #4
List: [69, 42, 20]
Expected Median: 42
Calculated Median: 42

Test Case #5
List: [10, 63, 60, 24, 21, 19, 36, 74, 41, 51, 71, 0, 25, 95, 13, 76, 29, 43, 54, 30, 6, 48, 24, 67, 69, 53, 58, 83, 53, 13, 21, 50, 87]
Expected Median: 48
Calculated Median: 48
```

Q5.

For this question, we are asked to design an exhaustive search algorithm that needs to look for every subset of a set that satisfies a given condition. Independent of the condition, we can see that we need to exhaustively search for every subset of the given subset to find the answer. Since there are 2^n subsets of a subset, we need to do 2^n iterations to generate them all. After that, we need to look for each generated subset if it satisfies the condition. This means that we need to do 2^n * 2^n number of generating and checking the subsets. Also, since we need to find the multiplication of each sub array that also gives times n complexity. This problem gives the same running time for the best, worst and average cases because up until the final subset, we can't know if we have found the correct solution.

Example Runs:

```
QUESTION 5:
Test Case #1
List: [6, 65, 79, 50, 70, 53, 32, 7, 73, 79, 77, 72]
Optimal Sub-Array: [65, 79, 32, 79]

Test Case #2
List: [71, 62, 21, 53, 90, 16, 53, 15, 84, 81, 33, 11]
Optimal Sub-Array: [71, 62, 90, 81]

Test Case #3
List: [40, 68, 57, 57, 13, 65, 94, 41, 91, 59]
Optimal Sub-Array: [40, 68, 65, 94]

Test Case #4
List: [7, 36, 71, 51, 90, 51, 30, 41, 97, 73, 1, 40, 10]
Optimal Sub-Array: [71, 90, 97, 73]

Test Case #5
List: [13, 92, 90, 70, 2, 72, 52, 64, 51, 95]
Optimal Sub-Array: [92, 90, 64]

Test Case #6
List: [69, 61, 8, 29, 83, 19, 47, 89, 87, 44]
Optimal Sub-Array: [69, 89, 87]

Test Case #7
List: [93, 0, 22, 24, 9]
Optimal Sub-Array: [93, 0, 24]
```