CSE 321 HOMEWORK 3

FURKAN OZEV

161044036

1) Black – White Problem And Algorithm Analyze (Decrease and Conquer):

ALGORITHM:

1- Initialize i as 1.
2- Calculate half of length of array. (Calculate n)
3- If half of length of array is even, Initialize j as n else, initialize j as n + 1.
4- Until i is equal to n,
swap element i of the array with element j and increment i and j with 2.

ANALYZE COMPLEXITY:

1- Initialize i take O(1) time. T1(n) = O(1)
2- Calculate n take O(1) time. T2(n) = O(1)
3- For both n odd or even cases, only initialize j is performed. So, No matter what situation comes, this step take O(1) time. For n odd => T3(n) = O(1) For n even => T3(n) = O(1)
4- In this step, a swap(or move) operation is performed until i is equal n. We know that i starts as 1 for each case. In this loop, i is incremented by 2, so it returns n / 2 times. We assume that interchanging any two boxes is considered to be one move. So, this step take O(n/2) => O(n) time. T4(n) = O(n)

T(n) = T1(n) + T2(n) + T3(n) + T4(n)

Best Case:

T(n) = O(1) + O(1) + min(O(1), O(1)) + O(n)

T(n) = O(n) for best case.

Worst Case:

T(n) = O(1) + O(1) + max(O(1), O(1)) + O(n)

T(n) = O(n) for worst case.

Average Case:

T(n) = O(1) + O(1) + (O(1) * ½ + O(1) * ½) + O(n)

T(n) = O(n) for average case.

EXPLANATION ALGORITHM:

I have one array that has 2n length. The first n element of this array is blackbox and the other n element is white box.

The elements in this array are ordered to be black and white consecutively.

Think of the black and white part of this array as two different lists.

To bring black, white, If half of length of array is odd, we need to swap the odd-index elements of these lists. Else, we need to swap the odd-index elements of black list and even-index elements of white list.

This is done in order to prevent 2 boxes of the same color from side by side.

Thus, this problem is solved from the left side to the right side. So we can call this algorithm decrease-and-conquer.

Assuming that the length of the array is 2n, this problem is solved with n/2 moves.

EXAMPLE 1:

['Black', 'Black', 'Black', 'White', 'White', 'White']

2n = 6 so n is odd,

1- ['Black', 'White', 'Black', 'White', Black, 'White']

EXAMPLE 2:

['Black', 'Black', 'Black', 'Black', 'White','White', 'White', 'White']

2n = 8 so n is even,

1- ['Black','White','Black','Black','Black','White','White','White']

2- ['Black','White','Black',White,'Black','White','Black','White']

PSEUDOCODE:

```
function foo(array):
        i <- 1
        n <- half of the length of array
        if n is even:
                j <- n
        else:
                j <- n+1

        while i < n:
                blackbox <- row[i]
                whitebox <- row[i + n]
                row[j] <- blackbox
                row[i] <- whitebox
                i <- i + 2
                j <- j + 2
        end while
end
```

2) Fake Coin Problem And Algorithm Analyze (Decrease and Conquer):

ALGORITHM:

1- Calculate length of array. (Calculate n)
2- If n is even:
   a. The coins are divided into 2 equal amount parts and weighed.
   b. Choose lightest side.
3- Else:
   a. Take a 1 coin.
   b. (n-1) coins are divided into 2 equal amount parts and weighed.
   c. If both sides are equal weight left out is the coin is fake.
   d. Else Choose lightest side.
4- This process continues until the fake coin is found.

ANALYZE COMPLEXITY:

1- Calculate n take O(1) time. T1(n) = O(1)
2- If n is even: T2(n) = O(1) + O(1) => T2(n) = O(1)
    a. Dividing coins into 2 equal parts and weighing take O(1) time.
    b. Choosing lightest side take O(1) time.
3- Else: T3(n) = O(1) + O(1) + (O(1) or O(1)) => T3(n) = O(1)
    a. Taking a coin take O(1) time.
    b. Dividing coins into 2 equal parts and weighing take O(1) time.
    c. If both sides are equal:
        i. the coin will be found. This step take O(1) time.
    d. Else:
        i. Choosing lightest side take O(1) time.
4- Continuing this process until the fake coin is found take T(n/2) time. The algorithm will terminate when the fake coin is found. So it can take O(1) time. T4(n) = (T(n/2) or O(1))

T(n) = T1(n) + (T2(n) or T3(n)) + T4(n)

Best Case:

T(n) = O(1) + min(O(1), O(1)) + min(T(n/2) + O(1))

T(n) = O(1) + O(1)

T(n) = O(1) for best case.

This is the case where the number of case coins is odd and first coin taken is fake.

Thus, the first attempt is found fake coin.

Worst Case:

T(n) = O(1) + max(O(1), O(1)) + T(n/2)

T(n) = T(n/2) + O(1)

T(n) = O(logn) for worst case.

Average Case:

T(n) = (Teven(n) + Todd(n)) * ½

Teven(n) = O(1) + O(1) + (O(1) * ½ + O(1) * ½) + T(n/2)

Teven(n) = O(logn)

Todd(n) = O(1) + O(1) + (O(1) * ½ + O(1) * ½) + (T(n/2) *(n-1/n) + O(1) * 1/n)

(T(n/2) *(n-1/n) + O(1) * 1/n) => 1/n is probably the fake coin taken.

Todd(n) = O(logn)

T(n) = (O(logn) + O(logn)) * ½

T(n) = O(logn) for average case.

EXPLANATION ALGORITHM:

1- Calculate length of array. (Calculate n)
2- If n is even, then put half the coins on each side of the balance. The side which is lightest contains the fake coin.
3- If n is odd, then take the 1 coin and split the (n−1) coins in half and put each half on the balance. If both sides are equal weight, then we are found fake coin because the coin we left out is the fake. If the balance is not stable then choose the lightest half of balance. Because, The side which is lightest contains the fake coin.
4- We continue this process until we have found the fake coin. Reduce coins by selecting one on each side of the balance.

3) Quick Sort – Insertion Sort:

```
-------------- PART3 TEST --------------

Unsorted Array:
[15, 59, 8, 69, 25, 36, 24, 80, 20, 21, 76, 80, 51, 67, 39, 31, 60, 48, 4]

        --- AFTER QUICKSORT ALGORITHM ---
Sorted Array:
[4, 8, 15, 20, 21, 24, 25, 31, 36, 39, 48, 51, 59, 60, 67, 69, 76, 80, 80]
Number of Swap: 36

        --- AFTER INSERTIONSORT ALGORITHM ---
Sorted Array:
[4, 8, 15, 20, 21, 24, 25, 31, 36, 39, 48, 51, 59, 60, 67, 69, 76, 80, 80]
Number of Swap: 79

----------------------------------------
```

In Quick sort algorithm, there is less swap operation than insertion sort algorithm.

QuickSort Algorithm:

```
quickSort(arr[], low, high):
  if (low < high):
    pi = partition(arr, low, high);
    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
  endif
end
```

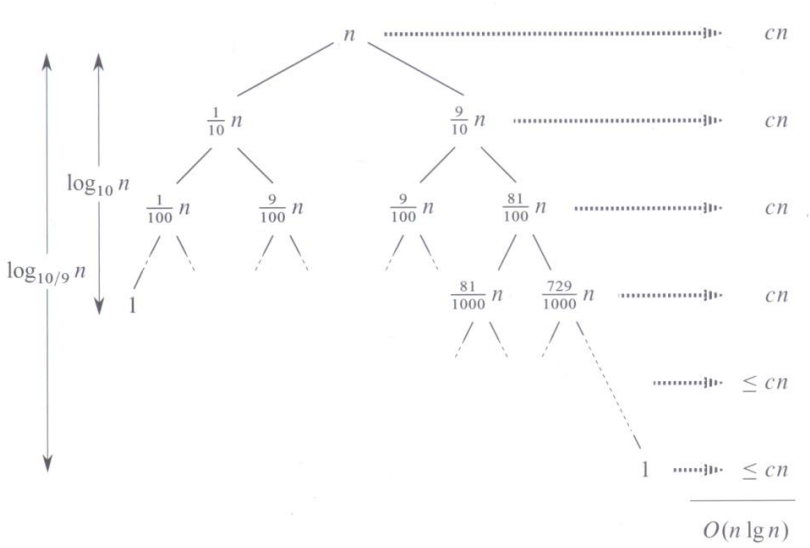
Time taken by QuickSort in general can be written as following.

$T(n) = T(k) + T(n-k-1) + (n)$

There are two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.

The time taken by QuickSort depends upon the input array and partition strategy. We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Because pivot is last element. Following is recurrence for this case:

$T(n) = T(n/9) + T(9n/10) + O(n)$



In this case, the shortest depth of a leaf is log10(n), while the largest depth is log10/9(n). In this bad splitting, the cost is still Θ(n log n).
Solution of above recurrence is also O(nLogn)
So average case of quicksort is O(nLogn)

InsertionSort Algorithm:

```
insertionSort(A)
    for j = 2 to n
        key ← A [j]
        j ← i – 1
        while i > 0 and A[i] > key
            A[i+1] ← A[i]
            i ← i – 1

        end loop

        A[j+1] ← key

    end loop

end
```

$$\sum_{i=1}^{N-1} \frac{i+1}{2} = \frac{(N-1)N}{4} + \frac{N-1}{2} = \frac{(N-1)(N+2)}{4}$$

So average case of InsertionSort is O(n^2)

Theoretically, we see that the quicksort algorithm is better than the insertionsort algorithm. Because, Quicksort has O(nlogn) complexity and InsertionSort take O(n^2) complexity.
When we look at the number of swaps, we can see that the calculations are theoretically and experimentally equivalent.

4) Median Problem (Decrease and Conquer):

ALGORITHM:

1- Divide array into groups of 5 elements. Last group may have less than 5 elements.
2- Sort the above created n/5 groups and find median of all groups.
3- Create an array like median [ ] and store medians of all ⌈n/5⌉ groups in this median array.
4- Medianres <- Recursively call this method to find median of median array
5- position <- partition(array, n, Medianres)
6- If position equal k, result <- Medianres
7- If postion > k, result <- Recursively call this method to find left part of array
8- If postion < k, result <- Recursively call this method to find right part of array
9- Return result

ANALYZE WORST CASE COMPLEXITY:

1- Dividing array into groups of 5 elements take O(1) time.
2- Sorting then finding median of these groups take O(1) time. (Assume that sorting 5 element take constant time)
3- Creating array and storing these 5 median value take O(1) time.
4- Recursive calling for median array take T(n/5) time.
5- This step standart partition so take O(n) time.
6- If position equal k, just initiliaze result take O(1) time.
7- Else, recursive calling for one part of array take maximum T(n-1) time.

$T(n) = O(1) + O(1) + O(1) + T(n/5) + O(n) + \max(O(1), T(n-1))$

$T(n) = T(n/5) + T(n-1) + O(n)$

$T(n) = cn/5 + c(n-1) + O(n)$

$T(n) = 6cn/5 + O(n)$

$T(n) <= cn$

$T(n) = O(n)$

5) Sub-array Problem (Exhaustive Search):

ALGORITHM:

1- Calculate length of array. (Calculate n)
2- If index equal length of array:
    a. Determine maximum and minumum element of array
    b. Calculate condition value: (min+max) * n / 4.0
    c. Calculate sum of subarray
    d. If subarray bigger than this condition value:
        i. The multiplication of elements of the last result subarray satisfying this condition is calculated and the multiplication of elements of the new subarray.
        ii. If the product of the new sub array is smaller, it is stored as a result. If not, result will not change and remains as the previous subarray.
3- Else:
    a. Subarray includes this element.
    Result1 <- Recursive calling with this subarray

b. Subarray does not include this element.
   Result2 <- Recursive calling with this subarray
c. Select the one that best fits the condition for these 2 results.

ANALYZE WORST CASE COMPLEXITY:

1- In this algorithm, for every element in the array, there are two choices, either to include it in the subsequence or not include it. Applied this for every element. So we can say that:
T(n) = 2T(n-1) + k

$$
\begin{aligned}
T(n) &= 2T(n-1) + k \\
&= 2\big(2T(n-2) + k\big) + k \\
&= 2\big(2\big(2T(n-3) + k\big) + k\big) + k \\
&= \cdots \\
&= 2^r T(n-r) + k\sum_{i=0}^{r-1} 2^i \\
&= 2^r T(n-r) + k(2^r - 1)\,.
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= 2^n T(0) + k(2^n - 1) \\
&= 2^n (T(0) + k) - k \\
&= O(2^n)
\end{aligned}
$$

T(n) = O(2^n)*f(n) => f(n) is other operation for each subarray

2- For each subarray, operations such as multiplication addition comparison and assignments are performed. These operations require linear complexity for each subarray. Because multiplication or summation elements of subarray takes o(n) time.

T(n) = O(2^n)*O(n)

T(n) = O(n*2^n)

EXPLANATION ALGORITHM:

1- For every element in the array, there are two choices, either to include it in the subarray or not include it. Apply this for every element in the array starting from index 0 until it reach the last index.
2- If it reach last index, check whether each subarray meets the requirement.
3- If it satisfies the condition, this subarray is compared in terms of element multiplication with the last result subarray satisfying the condition.
4- The subarray with the smallest result is stored as a result.
5- The result is return.

NOTE: All part algorithms are implemented as python code.