Introduction to Algorithm Design
Hamza Yoğurtcuoğlu
171044086 - Hw3

Question 1 :

We know that order of all boxes that the first n of them are black and the remaining n boxes are white. So we will do the same amount moves in **decrease by constant(1).** We move boxes one by one at the beginning and end of the line. If index is odd we will move. So , we do just n/2 moves in 2n boxes.

Best Case : We are moving same amount moves each 2n boxes standing in a row We can say f=n/2 moves so , $f \in \Omega(n)$

```
2n boxes standing in a row :

['black', 'black', 'black', 'black', 'white', 'white', 'white', 'white']


Designed array is:
['black', 'white', 'black', 'white', 'black', 'white', 'black', 'white']
black
white
black
white
black
white
black
white
```

Worst Case : The first n of them are black and the remaining n boxes are white. We are visiting all boxes but from end and beginning of boxes. We are doing same swapping n/2.

$f(x)=\sum_{i=0}^{n} 1$ **= 1+1+1…+1 = n -> visiting** $f \in O(n)$ **n/2 swappping**

Average Case :

Let f(*1*), f(*2*),…f(*n*) be the execution times for all possible inputs of size 2*n*, and let $p_1(1)$, $p_2(2)$,…$p_{2n}(2n)$ be the probabilities of these inputs.
$P_0 = 0$, $P_1 = 1$, $P_2 = 0$, $P_3 = 1$

$f(x)=\sum_{i=0}^{n} P_i T_i + P_{2n-i-1} T_{2n-i-1}$ = 1+1+1…= n $f \in \Theta(n)$

But we are saying that is interchanging any two boxes is considered to be one move. Again → n/2 moves.

Question 2:

We know the fake coin is heavier than the real ones. Put n/2 coins on each side of the balance. Discard the coins on the side that is lighter because we know the fake coin is on the heavier side. From the n/2 put n/4 coins on each side of the balance. Discard the coins on the side that is lighter. We now know that the fake coin is one of two. Put one coin on each side. The coin that is heavier is the fake coin.

Best Case : If n is odd number and fake coins is in end of list. We will find fake coins then we check balance 2,2 and 2,2. So, The coin outside of balance will prove to be fake because on both sides equal weight.

$$f \in \Theta(1)$$

```
--------------------------------------------------
[2, 2, 2, 2, 3]
Fake Coin Index -> 4
Fake Coin Weight -> 3
--------------------------------------------------
```

Worst Case : We divide n coins 2 part that i n/2, then continue to divided by 2 parts. When we divide 2 parts, can occur 1 extra coin if n is odd

T(n) = T(n/2) + 1 → We are using **master theorem.**

If a = 1 , b = 2, d = 0 → $a = b^d$ → $f \in \Theta(logn)$

Average Case : To simplify the calculation, let n be equal to $2^k - 1$.

$$f(x) = \sum_{i=0}^{logn} number\ of\ iteration\ at\ i * number\ of\ nodes\ at\ case\ i$$

$$f(x) = \sum_{i=0}^{logn} i * (\frac{n}{2^i})/n$$ → That is like binary search

= 0 + ½ + ½ + . . . + logn/2 = $f \in \Theta(logn)$

Question 3 :

[12, 11, 13, 5, 6]

```
QuickSort Swap Count ->  5
Sorted array is:
5
6
11
12
13
Insertion Swap Count ->  11
Sorted array is:
5
6
11
12
13
```

[1,2,3,4,5,6,7,8,9,10]

```
QuickSort Swap Count ->  54
Sorted array is:
1
2
3
4
5
6
7
8
9
10
Insertion Swap Count ->  9
Sorted array is:
1
2
3
4
5
6
7
8
9
10
```

Insertion Sort is not swapping when sorted list (9 means that not swap in insertion sort).Because sorted.

We see that if our array is mostly sorted insertion sort work fine. But Quick Sort takes more swaps. But mostly unsorted array is sorted better with Quick sort. Because Insertion sort n^2 time complexity. But Quick sort is nlogn depend on pivot.

Insertion Sort Average Case:

T_i i→ at basic operation at 0<=i<=n-1

$T(n) = T_1 + T_2 + \ldots + T_{n-1}$ → $E[T] = E[T_1 + T_2 + \ldots T_{n-1}]$ → $\sum_{i=0}^{n-1} E[Ti]$

for compute a $E[T_i]$ → $E[T_i] = \sum_{j=0}^{i} j.P(Ti=j)$ → P is j comparisons in step i

$P(T_i = j) = 1/(i+1)$ if $1<=j<=i-1$

$2/(i+2)$ if $j=i$

$E[T_i] = \sum_{j=0}^{i-1} (j.\frac{1}{(i+1)}) + 2\frac{i}{(i+1)} = (i^2 - i + 4i)/2(i+1) = i/2 + 1 - 1/(i+1)$

E[T] = $\sum_{i=0}^{n-1} i/2+1-1/(i+1)$ = n(n-1)/4 + n-1 $f \in \Theta(n^2)$

Quick Sort Average Case:

1) I select pivot element array[high] for each subarrays.

T = T_1 + T_2 T_1 → is operation in rearrange T_2 → operations in recursive calls

2) E[T] = E[T_1] + E[T_2]

3) E[T_2] = $\sum_{i=1}^{n} E[T2].P(x)$ each pivot same probabilities 1/n

so , E[T] = n + 1 + $\sum_{i=1}^{n} E[T2].P(x)$

n.E[n] = n.(n+1) + 2[E(0) + . . . E(n-1)

(n-1).E[n-1] = n.(n-1) + 2(E[0] + . . . E(n-2)

4) we subtract the equation below.

→ n.E[n] - (n-1).E(n-1) = 2n + 2E[n-1]

so , t(n) = E[n]/(n+1) = t(n-1) + 2/(n+1)

5) if t(1) → = t(0) + 1
t(2) → = t(1) + 2/3 = t(0) + 5/3

… That works like harmonic series.

6) t(n) = 2.H(n+1) - 2 $f \in \Theta(nlogn)$

For Quick Sort:
How often should we expect to see a split that's 3-to-1 or better? It depends on how we choose the pivot. Let's imagine that the pivot is equally likely to end up anywhere in an n-element subarray after partitioning. Then to get a split that is 3-to-1 or better, the pivot would have to be somewhere in the "middle half". So That depends on pivot.
For Insertion Sort :
Insertion sort has a fast best case running time and is a good sorting algorithm to use if the input list is already mostly sorted. For larger or more unordered lists, an algorithm with a faster worst and average-case running time, such as quick sort would be a better choice.

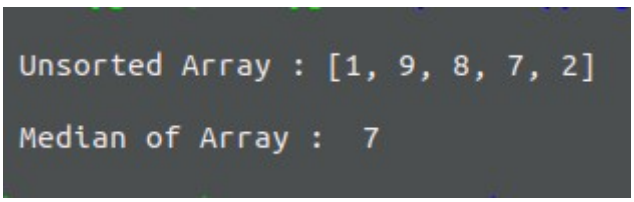
Question 4 :

Variable-Size-Decrease Problem:
As, in problem of finding median of unsorted array .We decrease neither by a constant nor by a constant factor. Below are example problems :
Algorithm
1) Divide arr[] into n/4 groups where size of each group.
2) Created n/4 groups and find median of all groups. Create an auxiliary array 'medianList[]' and store medians of all n/4 groups in this median array.
3) Find median of medians.
So, find median of all array.

Worst Case Complexity :
That takes linear time because we are divide (n/4 + n/4 + n/4 + n/4) four group (Variable size decrease) and finding median.

T(n) = c[n/4] + c(7n/8+6) + O(n)
= c.9n/8 + 6c + O(n)
= c.n

$$f \in O(n)$$

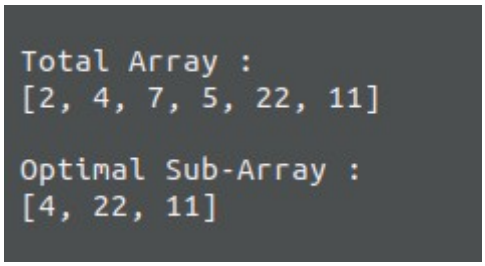


Sorted Array : 1 , 2 , 7 , 8 , 9

Question 5:

Algorithm :

Firstly, I found all possible subarrays of array using recursion. For every element in the array, there are two choices, either to include it in the subarrays or not include it. Apply this for every element in the array starting from index 0 until we reach the last index. Check all subarray is either optimal array or not. Once, the last index is reached and get back. When I found last subarray. Then If sum of subset is greater than or equal to **"SUM(B) >= (max(A) + min(A)) * n/4"** to satisfy the condition. I return checking the minimum number of multiplication of items in two arrays. Finally I can take optimal sub-array.



Worst Case :

If array → [1,2,3,4]

A subarray is a array that can be derived from another array by zero or more elements, without changing the order of the remaining elements.
For the same example, there are 15 subarray. They are (1), (2), (3), (4), (1,2), (1,3),(1,4), (2,3), (2,4), (3,4), (1,2,3), (1,2,4), (1,3,4), (2,3,4), (1,2,3,4). More generally, we can say that for a array of size n, we can have **$(2^n - 1)$** non-empty subarray in total.

$T(n) = T_1(n) + T_2(n)$

```
if index == len(A):
    if len(subarr) != 0:
        return subarr
else:
    optimalSubArray = recursiveExhaustiveSearchAlgorithm(A,rightSide ,index + 1, subarr)
    optimalSubArray2 = recursiveExhaustiveSearchAlgorithm(A,rightSide, index + 1, subarr + [A[index]])
```

This Part takes $T_1(n) = 2^{cn} - 1 \quad \in O(2^n)$

```python
def recursiveExhaustiveSearchAlgorithm(A,rightSide,index,subarr):

    optimalSubArray ,optimalSubArray2= [] , []

    if index == len(A):
        if len(subarr) != 0:
            return subarr
    else:
        optimalSubArray = recursiveExhaustiveSearchAlgorithm(A,rightSide ,index + 1, subarr)
        optimalSubArray2 = recursiveExhaustiveSearchAlgorithm(A,rightSide, index + 1, subarr + [A[index]])
    return findOptimal(optimalSubArray,optimalSubArray2,rightSide)
```

In $T_2(n)$: findOptimal function

We are checking 2 sub-array that are greater than **"SUM(B) >= (max(A) + min(A)) * n/4"** to satisfy the condition and returned minimum multiplication of sub-array.
Max subarray takes O(n) time complexity. But If we think our optimal array exists n-1 elements. This operations will be done $(2^n-1).(n-1)$ times.

$T(n) = 2^n \ n.2^n - 2^n \rightarrow f \in O(2^n.n)$ My algorithm takes this time complexity.