CSE 321
INTRODUCTION TO ALGORITHM DESIGN

HOMEWORK 03 REPORT

Ömer ÇEVİK

161044004

1 Question's Solution

In this part, I keep an array which contains 'BLACK' boxes in first half and last half contains 'WHITE' boxes. In my test I used 4 BLACK and 4 WHITE boxes in that array.

In *question1()* function, I create an empty B array and sending it to *decrease_and_conquerQ1()* two half of this array and B array.

decrease_and_conquerQ1() function checks the index of array is smaller than length of this array and if it is true then B array inserts indexed array to B's correct index. Then returns the recursive call increasing indexes.

Time Complexity :

Worst Case : The worst case scenario is about all elements of array completely searched and it is already happened! The purpose of this part is moving on each element at least once. In my solution, half of array 'n' is recursively run and other half of array 'n' is recursively run either.

So $T_W(n) \in O(2n)$ and $T_W(n) \in \underline{O(n)}$

Best Case : As same scenario with worst case. Because there is only one condition in function. That checks the length of array. So it always make a recursive call until reach to length of array.

So $T_B(n) \in O(2n)$ and $T_B(n) \in \underline{O(n)}$

Average Case : That scenario must not change as other way. Because condition is same.

So $T_A(n) \in O(2n)$ and $T_A(n) \in \underline{O(n)}$

2 Question's Solution

In this part, I keep an array which contains '1' values in array and in test I gave the input size as 10. In $test2()$ function I used random library just to set fake coin in random index and the fake coin has been set as '0'.

In my solution, $findFakeCoin()$ function gets coins as array and its start and last indexes. As last parameter length of array. Checking firstly the length of array if it is 1 then first index must be the fake coin. If the length of array is 2 and last index is not length of array and first element of array is greater than last element of array then we know the last element is fake coin. If the length of array is 2 and last index is not length of array and first element of array is smaller than last element then we know that first element is the fake coin.

What if the last index is not array's length then if first element of array is greater than 'last-1' indexed element of array then 'last-1' is our fake coin's index. If the last index is not array's length and first element of array is smaller than 'last-1' indexed element of array then first element is our fake coin's index.

We see that two conditions are our base conditions. For the recursive call we have other situations like if the length of array is greater than 2.

In that way we calculate the 'weighbridge' and using weighbridge calculating A's weight and B's weight.

If A's weight equals to B's weight then our result is recursive call giving new first index as first index and 2 of weighbridge and length of array size must be last minus first index and 2 of weighbridge.

If A's weight greater than B's weight then our result is recursive call giving new first index as first index plus weighbridge and last index must be first index and 2 of weighbridge and length of array is weighbridge.

If A's weight smaller than B's weight then our result is recursive call giving new last index must be first index plus weighbridge and length of array is weighbridge.

Then returns the index of fake coin.

Time Complexity :

Worst Case : findFakeCoin be summarized as T(n) = T(n/3) + $\theta(1)$ where $\theta(1)$ represents a constant number of weighings. Using Master Theorem we see the $log_3 n$.

So $T_W(n) \epsilon O(logn)$

Best Case : The time complexity of our algorithm for finding a fake coin is analyzed in terms of the number of times at which the comparing of weights between two coinsets occurs for a problem of size n coins. The best case occurs when n is a multiple of 3 and no counterfeit coin exists. When this happens, only 2 weightings occur and thus findFakeCoin has a best-case performance of $\theta(1)$.

So $T_B(n) \epsilon O(1)$

Average Case : In that scenario average complexity of findFakeCoin function must be include the best case O(1). Also there are three recursive function call which represents O(logn). Also the fake coin couldn't be in the array. This possibility is 3logn.

So $T_A(n) \epsilon O(logn)$

3 Question's Solution

In this part, there is an array in test as [10, 7, 8, 9, 1, 5]. And we are trying to sort it using Quick Sort and Insertion Sort. While sorting, we are counting the swap operations. In my solution I counted 5 for Quick Sort and 16 for Insertion Sort. For practically I see that Quick Sort algorithm is better choice to sort.

Time Complexity :

Average Case For Insertion Sort: To insert the last element, we need at most n-1 comparisons and at most n-1 swaps. To insert the second to last element, we need at most n-2 comparisons and at most n-2 swaps, and so on. The number of operations needed to perform insertion sort is therefore:

$2 \times (1 + 2 + \ldots +$ n-2 + n-1). To calculate the recurrence relation for this algorithm, use the following summation:

$$\sum_{q=1}^{p} q = \frac{p(p+1)}{2}$$

It follows that: $\frac{2n(n-1)(n-1+1)}{2} = n(n-1)$.

Use the master theorem to solve this recurrence for the running time. As expected, the algorithm's complexity is $O(n^2)$. When analyzing algorithms, the average case often has the same complexity as the worst case. So insertion sort, on average, takes $O(n^2)$ time.

∇ So $T_A(n) \epsilon O(n^2)$

Average Case For Quick Sort: To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy. 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. Following is recurrence for this case.

T(n) = T(n/9) + T(9n/10) + θ(n) and using Master Theorem we see find the complexity.

∇ So $T_A(n) \epsilon O(nlogn)$

In theoretically we can see that Quick Sort algorithm is better than Insertion Sort algorithm in average time complexity way ($O(nlogn) \leq O(n^2)$).

4 Question's Solution

In this part, I created an array as [1, 3, 4, 2, 7, 5, 8, 6] in test function and used Insertion Sort decrease and conquer algorithm to sort the array. After that operation the middle one is our median element if the length of array is odd. If length of array is even then middle two elements' of array average will give the median.

Time Complexity :

Worst Case : If we check the Insertion Sort algorithm to analyze, we see the nested loop which searches in 'n' times. For the worst case scenario we can assume that the loops will run until the nth elements. So the worst case scenario will apply n × n basically and to find the median operations only $\theta(1)$.

So $T_W(n) \in O(n^2)$

5 Question's Solution

In this part, I created the array in test function which is given in homework. Then called a function named *question*5() and passed the argument the array. In that function, calculated the condition which is given in homework and called the *exhaustiveSearch*() function.

That *exhaustiveSearch*() function evaluates all sub-arrays using recursive calls. The base case of that function is condition of index's length of array equal. If the base case happens then we can check the conditions to select the right sub-array. I give a parameter *result* which keeps 2D array. It keeps the necessary sub-array in index 0.

One of the wanted condition is multiplication of sub-array's each elements. For to evaluate that, created a function *arrayMultiply*() for each sub-array.

Under the base case there are two recursive calls. One of this recursive call runs until n. Other one is to n/2. This recursive calls can be represented as combinational formula: $\binom{n}{\frac{n}{2}}$

Time Complexity :

Worst Case : Using the combinational formula we can assume that in mathematically $\binom{n}{\frac{n}{2}}$ as approximately 2^n. So the worst case scenario becomes 2^n.

So $T_W(n) \in O(2^n)$