CSE 321 - Introduction to Algorithm Design

Ahmet Yusa Telli
151044092

Fall 2019 - 2020

1 Black - White Boxes

We have a list, which has black and white boxes. The list's length is 2n. There are n white and n black boxes. First n of them are black and the remaining n boxes are white. I use 'list' data structure. In the decrease-and-conquer (DAC) algorithm, we should split our list one by one element. My test list is:

```
WB = [1 , 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

My DAC algorithm is:

```
def DAC(row, mixed):
    if row:
        mixed.append(row[0])
        mixed.append(row[-1])
        DAC(row[1:-1], mixed)
        return mixed
    return row
```

In this algorithm, we take one element from the beginning and one element from the end. Then we append to each other. Then they are mixed.

Analyze

Analyzing the algorithm, we have same results for best and worst cases. We must do same thing every cases. Because we have one type input and one type output. There is no difference between best, worst and average cases.

We have 2n array and the recursive function calls n times. In the code; we have one recursive call, input array decreasing by 2 and run time without recursive line is 3.

$T(n) = a.T(n/b) + f(n)$

$a = 1 \qquad b = n - 2 \qquad f(n) = 3$

$T(n) = 1.T(n-2) + 3$

$Then \quad \Rightarrow T(n) = T(n) \in O(n)$

Every cases are $T(n) \in O(n)$. They are same complexity.

2 Fake Coin Problem

In this problem, We have a list with some coins and one fake coin. Original coins have same number (1) and fake coin is less number (0).
For solve this problem, we use a weighbridge. The weighbridge takes two lists and compares their summations. The input lists are usually same length. The weighbridge is like:

```
def weighbridge(liste1, liste2):
    if sum(liste1) > sum(liste2):
        return liste2
    elif sum(liste1) < sum(liste2):
        return liste1
    else:
        return liste1 if len(liste1) > len(liste2) else liste2
```

In the DAC algorithm, we take one list. We split it in the middle. Then We split again which has less summation. Until the list have one element, we split the input. At the end, we find the fake coin.

Analyze:

Best Case:
If the input has one element, we return the element. In this time, the element is real or not, it does not matter. The function returns in one step.
$T(n) \in O(1)$

Worst case:
If fake coin is in the last index. Every time we take half list, then we can find the fake coin in $n/2$ times call recursive. We have one recursive call and it takes half of input list.
$T(n) = 1.T(n/2) + O(1)$
$T(n/2) \in O(n/2)$
$a = 1 \quad b = 2 \quad d = 0$
$1 = 2^0 \Rightarrow a = b^d$
$\Rightarrow T(n) \in \theta(logn)$

Average Case:
Average Case is the same Worst case. But we need to add, if there is no fake coin in list. We add one more step.
$T(n) = 1.T(n/2) + O(1) + 1$
$T(n/2) \in O(n/2)$
$a = 1 \quad b = 2 \quad d = 0$
$1 = 2^0 \Rightarrow a = b^d$
$\Rightarrow T(n) \in \theta(logn)$

3 Insertion - Quick Sort

We compare the swap counters of two sorting algorithms. I test these algorithms with three different lists. First list is decreasing (>) list, second list is increasing (<) list, third list is mixed. These lists have 6 elements. The three test lists are:

```
arr1 = [10, 9, 8, 7, 6, 5]
arr2 = [5, 6, 7, 8, 9, 10]
arr3 = [5, 10, 6, 9, 7, 8]
```

For first array, the Insertion Algorithm has 15 times swap, the Quick Sort Algorithm has 6 times swap. We can see, the Quick Sort Algorithm is better than the Insertion Sort for the worst lists.

The second array, the Insertion Algorithm has 0 times swap, the Quick Sort Algorithm has 6 times swap. If the input array is already sorted, the Quick Sort Algorithm is the worst choice. Because the Quick Sort try to swap elements 6 times.

The third array, the mixed array, both algorithms have 6 times swap. They are the same result for this list.

Insertion Sort Average Case:

The worst and average case for Insertion Sort will occur when the input is in decreasing ordered list. To move the last element to the beginning, we need to do $length-1$ times swaps. For the second last element, we need to the $length-2$ times swaps.Therefore the number of swaps needed to do Insertion Sort is:
$2x(1+2+3+....+n-2+n-1)$

Calculate the recurrence relation for Insertions Sort:

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

$\frac{2(n-1)(n-1+1)}{2} = n(n-1)$

Then using master theorem we can find:

$T(n) \in O(n^2)$

Quick Sort Average Case:

For Quick Sort algorithm worst case is unbalanced lists. In unbalanced list, Quick Sort Algorithm involves $O(n)$ work + 2 recursive calls on lists of size 0 and n-1, the recurrence relations:

$T(n) = O(n) + T(0) + T(n-1) \Rightarrow O(n) + T(n-1)$

If balanced list, Quick Sort runs $O(n)$ work + two recursive calls on lists of size $n/2$, the recurrence relations:

$T(n) = O(n) + 2T(\frac{n}{2})$

Then the master theorem using:

$a = 2 \quad b = 2 \quad d = 0 \qquad a = b^d$

$\Rightarrow$ T(n) = O(n $\log n$)

4 Median

In this part, we have a unsorted list. For the list, We need to find the median of the list. Before the find, We have to sort the list. Then, We can find median of the list.

First, We call Insertion Sort algorithm for the list. After sorted list, if the size of list is odd number the median is the middle element. If the size of list is even number, the median is the average of the elements in the middle.

Worst Case:
We use Insertion Sort algorithm for this part. The worst case is the same Insertion sort worst case result. In Insertion Sort, worst case is decreasing order list. We find the worst case :

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

$\frac{2\times(n-1)\times(n-1+1)}{2} = n(n-1)$

Then using master theorem we can find: $T(n) \in O(n^2)$

5 Optimal Sub-Array

In this part, We need to find all sub-lists of input list. For sub-lists, we have a recursive call. The recursive runs with the list without the first element. Then, it runs n times. $T(n) \in O(n)$

We have a minimum total number. The sum of the number of elements of the sub-lists must be greater than this number. The sub-lists number of elements is 2^n. We eliminated some sub-lists.

In the "findMultiplications" function, we found the smallest multiplication of the sub-lists of the elements. Now, We have less than 2^n sub-lists, let say the number of removed sub-lists is x. We have $2^n - x$ sub-lists. In the sub-lists, the maximum number of elements of the sub-lists can be n. In the function, first loop runs form 0 to number of elements of sub-lists ($2^n - x$). The second loop runs from 0 to n. Let say $2^n - x = m$. The function runs mxn times.

We find the minimum total number using this formula:

$minimumtotal \geq (max(list) + min(list)) \times \frac{sizeoflist}{4}$

Worst Case : If minimum total number is too small, We have too many sub-lists. Then, the multiplication function runs $2^n \times n$ times. And it takes a lot of time. At the end of the result, this multiplication is the worst function in this part and we need to handle this function.

$T(n) \in \theta(2^n \times n)$

Ahmet Yuşa Telli
151044092