CSE321 Homework 4

Ahmet Tuğkan AYHAN

1901042692

Question 1

* This algorithm uses recursive calls. First if condition is the base condition for algorithm. I selected `n==0` and `n==1` because when wire length becomes 1 or 0 this means we can't cut them anymore

* Second part (else part) calls the function with `alg((n+1)//2)`. In this part there is two condition;
  → Length can be an even number
  → Length can be an odd number

* If length is an even number there is no difference between calling for `n/2` or `n+1/2`. Because Python rounds int division to floor. For example if `n=4`, then `4//2 = 2` and `5//2 = 2`

* If length is an odd number, well, there is a difference between `n//2` and `n+1//2`. We need to call it with `n+1//2` because the some reason as even number. For example if `n=3` then `n//2` returns 1. This is base condition and algorithm stops. But answer is not 1 cut, It is 2. Because of that, to round odd number to ceiling, we use `n+1//2`. The last +1 corresponds to a cut.

Recurrence Relation

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

a : 1
b : 2
f(n) : 1

* $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$

* $1 \Leftrightarrow f(n) \rightarrow 1 = 1$, thus we use 2nd case of Master's theorem

2nd Case : $T(n) = \underbrace{n^{\log_b a}}_{1} \cdot \log n \rightarrow \boxed{T(n) \in \Theta(\log n)}$

Time Complexity = $\Theta(1)$, because there is no extra memory allocation inside the algorithm that increases with n

Question 2

* I used merge sort algorithm but I removed every swapping, copying operation etc. Because I don't need to make any sort operation. Instead of that I constantly divided the array into two subarrays until array length becomes 1. Then I compared worstRate and bestRate with that array. I initially gave bestRate → 0 and worstRate → 999. If bestRate is lower than that array element, I put that element to bestRate. And if worstRate is higher than that array element I put that element to worstRate. Finally I returned the max bestRate between left and right subarrays. and returned the min worstRate between left and right subarrays.

Recurrence Relation

$$T(n) = 2\,T(n/2) + 1$$ → because there is no loop in the algorithm

a : 2
b : 2
f(n) : 1

* $n^{\log_b a} = n^{\log_2 2} = n$
* $n \Leftrightarrow f(n)$, $n > 1$ thus we need to use 1st case in master's theorem

1st Case : $T(n) = n^{\log_b a} = n \Rightarrow \boxed{T(n) \in \Theta(n)}$

Space Complexity = $\Theta(n)$, because I allocated new arrays for every subarrays

Question 3

* This algorithm is actually pretty easy and straightforward. What it does in the first place is getting max and min value of the list. After that it finds mean of these two values. Since we don't want first k-1 element we increment countLess value everytime we encounter with a value less than mean value. We repeat this until we check all elements in the array. Since equal values are acceptable but not counted as less than mean we increment them seperately

* If we find k-1 element that is less than k_{th} element then we break the inner loop. If countLess value is less than k (it must be k-1) and sum of countLess and countEqual values are bigger than or equal to k then that means mean value is actually the k_{th} smallest value in the list. If not then we recalculate the mean value by decrementing bestRate by 1 and incrementing worstRate by 1 until bestRate is bigger than or equal to worstRate. It is very similar to binarySearch

Time Complexity

* There are two loops. First loop $\log k$ because it acts like binary search

* $\sum_{m=0}^{\log k} \sum_{i=0}^{n} 1 = \sum_{m=0}^{\log k} (n+1) = (n+1).\log k + 1 \Rightarrow n.\log k$

* $T(n) \in \Theta(n.\log k)$

Space Complexity : $\Theta(1)$, because there is no allocation which increases with n

Question 4

* I used merge sort algorithm. But difference is I'm returning reverse pair count from each subarray. And before changing (sorting) the array I check if $i < j$ and $A[i] > A[j]$. if so then I increment pair count and return it when swapping is done.

Recurrence Relation

$T(n) = 2T(\frac{n}{2}) + n$ → because "conquer" function contains a loop which iterate n/2 times

a : 2
b : 2
f(n) : n

* $n^{\log_b a} = n^{\log_2 2} = n$
* $n \Leftrightarrow f(n)$ → we need to use case 2 in master's theorem

Case 2 : $T(n) = n^{\log_b a} . \log n \rightarrow$ $T(n) \in \Theta(n \log n)$

Space complexity : $\Theta(n)$, because in conquer function I copy array elements to a new array

Question 5

Brute Force

Inside brute force function I created a loop which iterates n times and multiplies "a" with itself. Then, it returns the result

Time Complexity : $T(n) = \sum_{i=0}^{n} 1 = n+1 \rightarrow T(n) \in \Theta(n)$

Space Complexity : $\Theta(1)$, because there is no allocation which increases with n

Divide and Conquer

Inside divide and conquer algorithm there is 2 base condition :

1) n can be 0 ($a^n = a^0 \rightarrow 1$)
2) n can be 1 ($a^n = a^1 \rightarrow a$)

Other than base conditions, a constantly divided n value to two parts :

Left part : $(n/2)$
Right part : $n - (n/2)$
} when we sum these two we get n

Until I get one of the base conditions I divided the problem into half and multiplied the return value from left side and right side. And returned the result.

Recurrence Relation

$T(n) = 2\,T(\frac{n}{2}) + 1$ → there is no loop etc.
- 2 → There is two recurrence calls
- $T(\frac{n}{2})$ → each recurrence call divides the problem into half size subproblems

a : 2 * $n^{\log_b a} = n^{\log_2 2} = n$
b : 2
f(n) : 1 * $n \iff f(n)$, $n > 1$, thus we need to use first case of the master's theorem

Case 1 : $T(n) = n^{\log_b a} = n^{\log_2 2} = n$, $T(n) \in \Theta(n)$

Space Complexity : $\Theta(1)$, because there is no allocation which is related to size of n