CSE 4810: Algorithm Engineering Lab
Lab - 2

Md Farhan Ishmam (180041120)

Group - CSE 1A

January 26, 2023

Task 1

We generalize the given functions in the following form:

(a) $f_1(n) = n^{1.01}log(n^{2019}) = O(n^{1.01}log(n))$

(b) $f_2(n) = 200000n^{1.5} = O(n^{1.5})$

(c) $f_3(n) = 1.00001^n = O(a^n)$

(d) $f_4(n) = n^2log(n^2) = O(n^2log(n))$

(e) $f_5(n) = \binom{n}{1} = O(n)$

The sorted functions in increasing order of asymptotic (big-O) complexity are given below:

1. $f_5(n) = O(n)$
2. $f_1(n) = O(n^{1.01}log(n))$
3. $f_2(n) = O(n^{1.5})$
4. $f_4(n) = O(n^2log(n))$
5. $f_3(n) = O(a^n)$

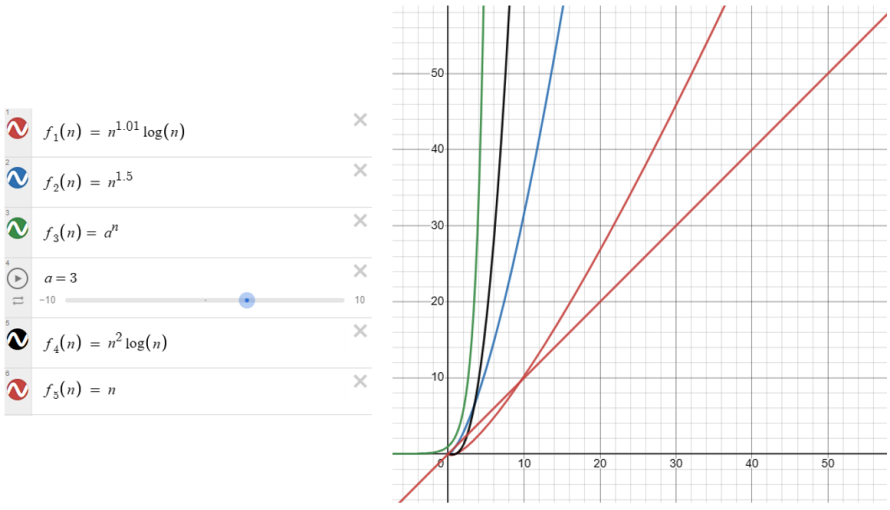

Figure 1: Comparison of the asymptotic complexities of the given functions

Task 2

The whole algorithm contains two nested loops. In the first nested loop, the outer loop runs for n iterations and the inner loop runs from at best 3 iterations for the values of $j = 0, 1, 2$. Hence, asymptotic complexity is

$$O(3n) = O(n)$$

Similarly, the second nested loop has an outer loop that runs for at best 3 iterations for values of $i = 0, 1, 2$. The inner loop will run for n iterations thus also making the asymptotic complexity

$$O(3n) = O(n)$$

Overall, the asymptotic complexity of the two nested loops is

$$O(3n + 3n) = O(n)$$

i.e. it takes linear time for the function to run.
The asymptotic complexity of the function is $O(n)$.

Task 3

The function is analogous to a binary search tree. The middle index of the array is calculated as y using the bit-shift operator and the base case is defined as when the value of the middle index is equal to the search value, then the middle index is returned. Afterward, the function calls itself for its two halves - from the start to before the middle index and after the middle index to the end. In each recursion call, the array gets divided into two halves until the array is of a single element. Let's assume the depth of the tree to be d. Hence,

$$\frac{n}{2^d} = 1$$
$$=> d = log_2(n)$$

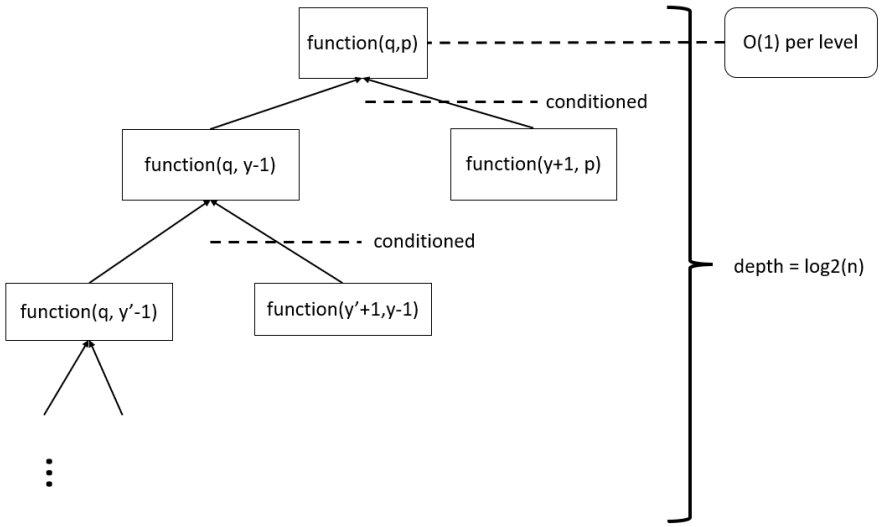

Figure 2: Recursion Tree of the given function

Each recursion call takes $O(1)$ to compute. However, the function makes only one recursion call as there is an if condition. According to the Master Theorem of Recursion, the top-level complexity and bottom-level complexity are the same and hence, the overall complexity is the complexity in each depth multiplied by the depth of the tree i.e.

$$O(1 * d) = O(1 * log_2(n)) = O(log_2(n))$$

. **The asymptotic complexity of the function is** $O(log_2(n))$.

Task 4

Complexity of function1

The complexity of function1 is similar to Task 3. However, in each function call, there is a nested loop. The outer loop runs for n iterations but the inner loop breaks after a single iteration making the overall complexity of the nested loop $O(n)$. Similar to Task 3, the array is split into two halves, and instead of an if condition, both halves are passed for the recursion calls. At the root node of the tree and at the bottom level of the tree, the complexity is $O(n)$. The depth of the tree is $log_2(n)$.
So, overall asymptotic complexity is $O(nlog_2(n))$

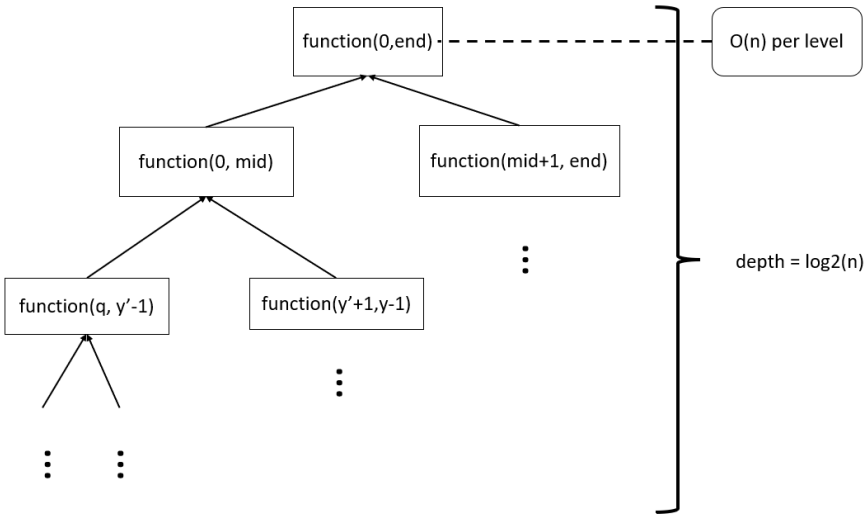

Figure 3: Recursion Tree of the given function

Complexity of function2

The function has a nested loop, where both the inner and outer loop runs on n iterations, making the complexity of each recursion call $O(n^2)$. But the array is divided into three halves, and each half needs to be called making the depth of the tree

$$\frac{n}{3^d} = 1$$

$$=> d = log_3(n)$$

So, overall asymptotic complexity is $O(n^2log_3(n))$

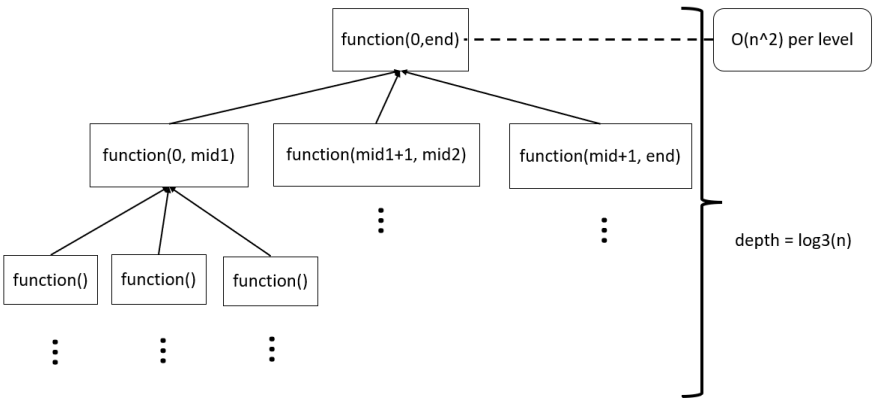

Figure 4: Recursion Tree of the given function

Difference between the Asymptotic Complexities

Generalizing the two equations

(a) $function1 = O(nlog_2(n)) = O(nlog_a(n))$

(b) $function2 = O(n^2log_3(n)) = O(n^2log_a(n))$

Clearly, function2 has a power of 2 and thus has the higher big-O complexity.
The asymptotic complexity of function2 is more than that of function1.

Task 5

Problem Statement

Parentheses have to be matched in a mathematical expression. Example - () is valid but (} is invalid.

Idea of Solution

1. We use a stack data structure and push the input symbols in the stack.
2. If the top of the stack is an opening parenthesis and the input symbol is a closing parenthesis, then we pop the symbol from the stack. Otherwise, the symbol is pushed.
3. At the end of input, if the stack is empty, we return true. Otherwise, we return false.

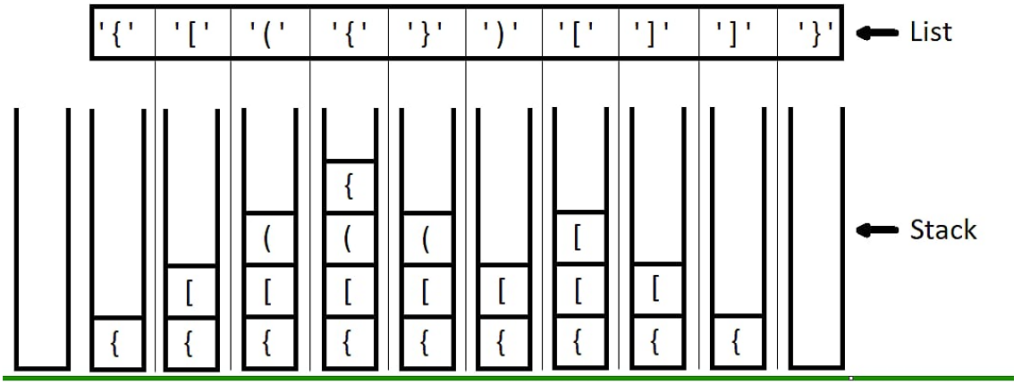

Figure 5: Stack used for Parenthesis Matching

Pseudocode

```
check_paranthesis(str):
    for symbol in str:
        if symbol in opening_symbols:
            stack.push(symbol)
        else if symbol matches closing_symbols:
            stack.pop()
        else if stack is empty:
            return false
    if stack is empty:
        return true
    else:
        return false
```