

CS 601 - Advanced Algorithms

HomeWork 5

Question 1) : Solve the following recurrence relations. For each one come up with a precise function of n in closed form (i.e., resolve all sigmas, recursive calls of function T , etc) using the substitution method studied in the class. Note: An asymptotic answer is not acceptable for this question.

Justify your solution and show all your work.

a) $T(n)=T(n-1)+n$, $T(0)=1$,

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

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

a) $T(n)=T(n-1) +n$, $T(0) =1$

Solution:

$$T(n) = T(n-1) + n \text{ for } n > 0 \text{ and}$$

$$T(n) = 1 \text{ for } n=0$$

Using Back substitution method,

$$T(n) = T(n-1) + n \dots\dots (i)$$

$$T(n) = [T(n-2) + n - 1] + n \dots\dots(ii)$$

.

.

.

Finally,

$$T(n) = T(n-k) + (n-(k-1)) + (n-(k-2)) + \dots + (n-1) + n$$

Assume $n=k$, and $n-k = 0$

$$T(n) = T(n-n) + (n-n+1) + (n-n+2) + \dots + (n-1) + n$$

$$T(n) = T(0) + 1+2+\dots+(n-1) + n$$

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

This can also be represented as **$O(n^2)$** .

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

Solution:

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

$$\text{For, } T(n/2) = 2T(n/2 \cdot 1/2) + n/2$$

$$T(n/2) = 2T(n/2^2) + n/2 \rightarrow (2)$$

Substituting (2) in (1), we get,

$$T(n) = 2[2T(n/2^2) + (n/2)] + n$$

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

$$T(n) = 2T(n/2^2) + 2n \rightarrow (3)$$

Now, continue the substitution for k times, we get,

$$T(n) = 2^k T(n/2^k) + k \cdot n$$

$$\text{If we assume that, } T(n/2^k) = T(1) \quad \text{then, } n/2^k = 1; \therefore n = 2^k$$

Taking log on both sides, we will get,

$$\log n = \log 2^k; \therefore \log n = k \log 2$$

$$k = \log n$$

Putting these values back in the T(n), we get,

$$T(n) = 2^k T(1) + k \cdot n \quad \because T(n/2^k) \text{ is assumed as } T(1)$$

$$T(n) = 2^k T(1) + k \cdot n$$

$$T(n) = n \cdot 1 + n \log n \quad \because n = 2^k \text{ and } k = \log n$$

Thus, the upper bound here will be **$O(n \log n)$**

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

$$T(n) = T(n/2) + 1 \text{ for } n > 1 \text{ and}$$

$$T(n) = 1 \text{ for } n = 1$$

Using back-substitution method,

$$T(n) = T(n/2) + 1 \dots (i)$$

$$= [T(n/2^2) + 1] + 1$$

$$T(n) = T(n/2^2) + 2 \dots (ii)$$

$$T(n) = T(n/2^3) + 3 \dots (iii)$$

.

.

.

$$T(n) = T(n/2^k) + k$$

Assuming $n/2^k = 1$ then $n=2^k$ and $k = \log n$

$$T(n) = T(1) + \log n$$

$$T(n) = 1 + \log n$$

This can also be represented as **$O(\log n)$** .

Question 2) : Consider Question 1 again. Apply Master Theorem if applicable for each case. Bound the recurrence relation in Big-O.

a) $T(n)=T(n-1) +n, T(0)=1$

Master's theorem cannot be applied for decreasing functions.

Master's theorem is expected to be of the form **$a \geq 1, b \geq 1, d \geq 0$** .

Since in the given equation, $a = 1$ so Master's theorem is not applicable.

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

By using master theorem for recursive functions, general form **$T(n) = a T(n/b) + f(n)$**

Here $a = 2, b=2$ thus $\log_a b = \log_2 2 = 1 \Rightarrow f(n) = n$ is written as n^1 , then $p=0$ and $k=1$

We have , $\log_a b = k$ [case 2 of the Master's theorem]

And since $p = 0$ which is > -1 results in first sub case

$$O(n \log(p+1)n) = O(n \log n)$$

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

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

By using master theorem for recursive functions, general form **$T(n) = a T(n/b) + f(n)$**

Here, $a = 2$, $b = 2$, thus $\log_a b = \log_2 2 = 1 \Rightarrow f(n) = n^0$ so $p=0$ and $k=0$ [Case 1 of Master's theorem].

$O(n^{\log_b(a)}) = O(n^1) = O(n)$

$T(n) = O(n)$

Question 3) A binary tree's "maximum depth" is the number of nodes along the longest path from the root node down to the farthest leaf node. Given the root of a binary tree, write a complete program in C++/Java that returns the tree's maximum depth. What is the time-complexity of your algorithm in the worst-case once you have n nodes in the tree. Analyze and clearly discuss your reasoning.

Time Complexity of maxDepth Function:

| Operation | Cost | Times |
|--|------|-------|
| if (root == null) | c1 | 1 |
| return 0; | c2 | 1 |
| int count = 0; | c3 | 1 |
| Queue<TreeNode> q = new LinkedList<TreeNode>(); | c4 | 1 |
| q.add(root); | c5 | 1 |
| while (!q.isEmpty()) { | c6 | n |
| int levelSize = q.size(); | c7 | n |
| int i = 0 | c8 | n |

| | | |
|--|-----|-----|
| i < levelSize | c9 | n+1 |
| i++ | c10 | n |
| if (q.peek().left != null) { q.add(q.peek().left); } | c11 | n |
| if (q.peek().right != null) { q.add(q.peek().right); } | c12 | n |
| q.poll(); | c13 | n |
| count++; | c14 | n |
| return count; | c15 | 1 |

$f(n)$ can be obtained by adding all costs time

$$f(n) = c1 + c2 + c3 + c4 + c5 + c6 + c7(n) + c8(n) + c9(n+1) + c10 + c11(n) + c12 + c13(n) + c14(n) + c15(1)$$

we consider only higher magnitude

$$f(n) = c(n) + c'$$

The time complexity in worst case scenario will be

$$O(n)$$

Reasoning:

In the maxDepth function we check whether root is null, if root is null will return 0. If not we store the root in the queue and iterate throughout the queue and store the left and right child of the dequeuing element in the queue levelwise. Once we finish dequeuing each level we increment the counter. So once we dequeue the last element from the queue we are in the last level. We just return the count.

Code:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Scanner;
import java.util.Stack;

public class MaximumDepth {
    static TreeNode root;
    static int goalElement; // Element to be searched

    static ArrayList<String> listOfInput = new ArrayList<>();

    public static class TreeNode {
        int data;
        TreeNode left, right;

        TreeNode(int data) {
            this.data = data;
            this.left = null;
            this.right = null;
        }
    }

    public static void constructTree() {
        for (int i = 0; i < listOfInput.size(); i++) {
            String characterElement = listOfInput.get(i); // Accessing each element
            if (!characterElement.equals("")) {
                String[] arrayOfcharacterElement = characterElement.split(",",
2); // Splitting input value into two
                int predecessorValue =
Integer.parseInt(arrayOfcharacterElement[0]);
                int successorValue =
Integer.parseInt(arrayOfcharacterElement[1]);
                if (root == null) {
                    root = new TreeNode(predecessorValue);
                    root.left = new TreeNode(successorValue);
                } else {
```

```

        insertIntoTree(predecessorValue, successorValue);
    }
} else {
    goalElement = Integer.parseInt(listOfInput.get(i + 1)); // Storing
    break;
}
}

}

}

public static void insertIntoTree(int predecessorValue, int successorValue) {
    Queue<TreeNode> queue = new LinkedList<TreeNode>();
    queue.add(root);
    while (!queue.isEmpty()) {
        TreeNode tempNode = queue.poll();
        if (tempNode.data == predecessorValue) {
            if (tempNode.left == null)
                tempNode.left = new TreeNode(successorValue);
            else
                tempNode.right = new TreeNode(successorValue);
        }
        if (tempNode.left != null) {
            queue.add(tempNode.left);
        }
        if (tempNode.right != null) {
            queue.add(tempNode.right);
        }
    }
}

public static void readTextFile() {
    try {
        File fileObject = new File(System.getProperty("user.dir") + "/input.txt"); //
        Scanner scanObject = new Scanner(fileObject); // file
        while (scanObject.hasNextLine()) {
            String characterElement = scanObject.nextLine();
            listOfInput.add(characterElement); // File input is stored in
        }
        scanObject.close();
    } catch (FileNotFoundException e) {
        System.out.println("Error occurred while accessing file");
        e.printStackTrace();
    }
}

```

goal element

Relative

arraylist

```

    }

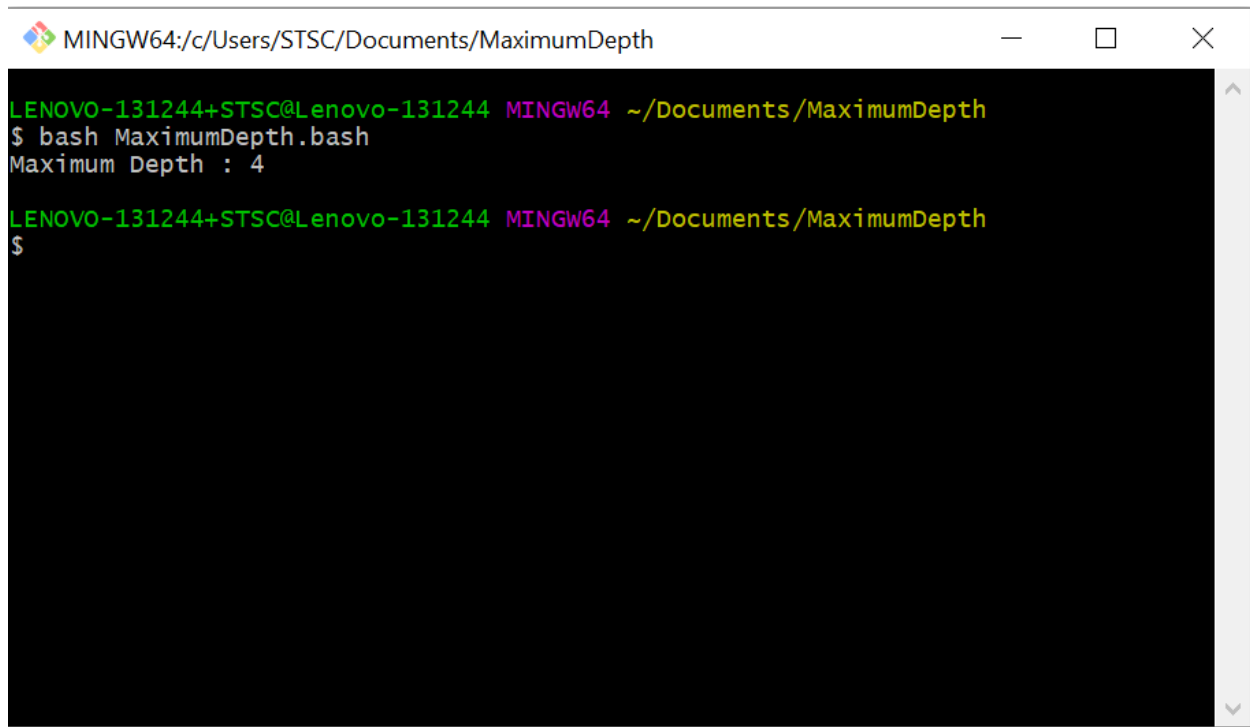
}

public static int maxDepth(TreeNode root) {
    if (root == null)
        return 0;
    int count = 0;
    Queue<TreeNode> q = new LinkedList<TreeNode>();
    q.add(root);
    while (!q.isEmpty()) {
        int levelSize = q.size();
        for (int i = 0; i < levelSize; i++) {
            if (q.peek().left != null) {
                q.add(q.peek().left);
            }
            if (q.peek().right != null) {
                q.add(q.peek().right);
            }
            q.poll();
        }
        count++;
    }
    return count;
}

public static void main(String args[]) {
    readTextFile(); // Function to read data
    constructTree(); // Function to construct tree
    System.out.println("Maximum Depth : " + maxDepth(root));
}
}

```

Output :

A screenshot of a MINGW64 terminal window. The title bar shows the path "MINGW64:/c/Users/STSC/Documents/MaximumDepth". The terminal output shows a prompt "LENOVO-131244+STSC@Lenovo-131244 MINGW64 ~/Documents/MaximumDepth" followed by the command "\$ bash MaximumDepth.bash". The output of the script is "Maximum Depth : 4". The prompt then changes to "\$" on the next line.

```
MINGW64:/c/Users/STSC/Documents/MaximumDepth
LENOVO-131244+STSC@Lenovo-131244 MINGW64 ~/Documents/MaximumDepth
$ bash MaximumDepth.bash
Maximum Depth : 4
LENOVO-131244+STSC@Lenovo-131244 MINGW64 ~/Documents/MaximumDepth
$
```

Question 4): Part (a) Write a linear time divide and conquer algorithm (i.e., $\Theta(n)$) to calculate x^n (x is raised to the power n). Assume a and n are ≥ 0 .

```
int calculateXN (int x, int n)
{
    if (n==0)
    {
        return 1;
    }
    else if (n%2==0) // if n is even
    {
        return calculateXN(x, n/2)*calculateXN(x, n/2);
    }
    else
    {
        return x*calculateXN(x, n/2)*calculateXN(x, n/2);
    }
}
```

Part (b) Analyze the time complexity of your algorithm in the worst-case by first writing its recurrence relation.

Time complexity of the function “calculateXN” is $T(n)$. We are calling this function two times for each call, each call of the size $n/2$ and other operations will take a constant time range.

$$T(n) = 2T(n/2) + c \dots\dots\dots (i)$$

Formulating for equation (i) with $n/2$

$$T(n/2) = 2T(n/2^2) + c \dots\dots\dots(ii)$$

By using Back Substitution , **Substituting (ii) in (i)**

$$T(n) = 2[2T(n/2^2) + c] + c$$

$$T(n) = 2^2T(n/2^2) + 2c + c, \quad [3c \Rightarrow (4-1)c \Rightarrow (2^2-1)c]$$

$$T(n) = 2^2T(n/2^2) + (2^2 - 1)c \dots\dots\dots (iii)$$

Formulating for equation (i) with $n/2^2$,

$$T(n/2^2) = 2T(n/2^3) + c \dots\dots\dots (iv)$$

By using Back Substitution , **Substituting (iv) in (iii)**

$$T(n) = 2^2[2T(n/2^3)] + (2^2 - 1)c, \quad [7c \Rightarrow 8c - 1c \Rightarrow 2^3c - 1c]$$

$$T(n) = 2^3T(n/2^3) + (2^3-1)c$$

.

.

$$T(n) = 2^kT(n/2^k) + (2^k-1)c, \text{ for } k\text{th iteration}$$

$$n/2^k = 1 \text{ as } T(1) = 1 \text{ then } n=2^k$$

Thus,

$$T(n) = nT(1) + (n-1)c$$

$$T(n) = n + (n-1)c$$

$$T(n) = O(n)$$

Part (c) Can you improve your algorithm to accomplish the end in $O(\log n)$ time complexity (we still look for a divide and conquer algorithm). If yes, write the corresponding algorithm, write the recurrence relation for its time complexity and analyze it. If not, justify your answer.

Yes, this is the Improved algorithm to accomplish $O(\log n)$ time complexity by calling the calculateXN function only once.

```
int calculateXN(int a,int b)
{
    int tempVar;
    if (b==0)
    {
        return 1;
    }
}
```

```

    }
    tempVar=calculateXN(a,b/2);
    else if (b%2==0) // if y is even
    {
        return tempVar*tempVar;
    }
    else
    {
        return a*tempVar*tempVar;
    }
}

```

Since we are calling the calculate function only once,

$$T(n) = T(n/2) + 1 \dots (i)$$

$$T(n/2) = T(n/2^2) + 1 \dots (ii)$$

Substituting (ii) in (i)

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

.

.

.

$$T(n) = T(n/2^k) + k$$

Since $T(1) = 1$ for base case,

$$n/2^k = 1$$

$$n = 2^k$$

$$\log n = \log 2^k \text{ gives } k = \log n$$

$$T(n) = T(1) + \log n$$

$$\mathbf{T(n) = O(\log n)}$$