CSE 321 Introduction to Algorithm Design

Tuba TOPRAK- 161044116

Hw2

1. Solve the following recurrence relations and provide a Θ bound for each of them. You must use backward substitution, forward substitution or the Master's Theorem at least once to solve the following relations.

   a) $T(n) = 3 * T(n-1) - 2 * T(n-2)$
   b) $T(n) = T(n/2) + 1$
   c) $T(n) = 4T(n-1) - 4T(n-2) + 3n$
   d) $T(n) = 4T(n/2) + n^2$
   e) $T(n) = 2T(n/2) + O(n)$
   f) $T(n) = T(n/2) + T(n/4) + n$
   g) $T(n) = T(n/2) + n$
   h) $T(n) = 2T(\sqrt{n}) + 1$

h) $T(n) = 2T(\sqrt{n}) + 1$

$\Rightarrow$ Let $n = 2^k$ then, $T(2^k) = 2.T(2^{k/2}) + 1$

$\Rightarrow$ Now, let $T(2^k) = S(k)$, then $T(2^{k/2}) = S(k/2)$

$\Rightarrow$ So we have; $S(k) = 2.S(\frac{k}{2}) + 1$

master teorem:

$a = 2$
$b = 2$ $\rightarrow a > b^d \rightarrow 2 > 2^0$
$d = 0$

Case 3 $\Theta(k^{\log_2 2}) = \Theta(k)$

Now $n = 2^k \rightarrow$ So $\log_2 n = k$

using that we can get $\Theta(k) = \Theta(\log_2 n) = \Theta(\log n)$

a) $T(n) = 3T(n-1) - 2T(n-2)$

· Using Forword Subsititon

$T(1) = 1 = 2^{1-1}$

$T(2) = 2 = 2^{2-1}$

$T(3) = 3.T(2) - 2T(1) = 3.2 - 2.1 = 4 \rightarrow T(3) = 4 = 2^{3-1}$

$T(4) = 3.T(3) - 2T(2) = 12 - 4 = 8 \rightarrow T(4) = 8 = 2^{4-1}$

$T(5) = 3.T(4) - 2T(3) = 24 - 8 = 16 \rightarrow T(5) = 16 = 2^{5-1}$

$T(n) = 2^{n-1}$

$\Theta(2^{n-1}) = \Theta(2^n)$

$r^2 - 3r + 2 = 0 \Rightarrow r_1 = 2, r_2 = 1$

$T(n) = c_1 . 2^n + c_2 . 1^n$

$T(1) \rightarrow 1 = 2c_1 + c_2$ /-

$T(2) \rightarrow 2 = 4c_1 + c_2$

$2c_1 = 1 \quad c_1 = \frac{1}{2}, c_2 = 0$

$T(n) = \frac{1}{2} . 2^n \Rightarrow \Theta(2^n)$

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

$T(n) = a.T(\frac{n}{b}) + fn \quad T(1) = c$ where $a \geq 1, b \geq 2$

if $f(n) \in \Theta(n^d)$ where $d \geq 0$

$T(n) =$ case1 $\rightarrow \Theta(n^d)$, if $a < b^d$

case2 $\rightarrow \Theta(n^d \log n)$, if $a = b^d$

case3 $\rightarrow \Theta(n^{\log_b a})$, if $a > b^d$

$a = 1$
$b = 2 \quad 1 = 2^0$
$d = 0$

So the result is $\Theta(n^{\overset{0}{d}} \log n) = \Theta(\log n)$

c) $T(n) = 4T(n-1) - 4T(n-2) + 3n$

using characteristic equation method.

$r^2 - 4 + 4 = 0 \quad r_1 = r_2 = 2$

$T(n) = (A + Bn) . 2^n$

Particular Solution: $P(n) = Cn + D$

$Cn + D = 4[C(n-1) + D] - 4[C(n-2) + D] + 3n$

$C = \frac{1}{2}, D = 0$ Particular Sol: $\frac{1}{2}n$

$T(n) = (A + Bn) . 2^n + \frac{1}{2}n = \Theta(2^n)$

d) $T(n) = 4T(n/2) + n^2$

$a = 4$
$b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^2$

$f(n) = \Theta(n^2 \lg^0 n)$, that is, $k=0$

$T(n) = \Theta(n^2 \lg n)$

e) $T(n) = 2T(n/2) + O(n)$

$= 2(2T(n/4) + n/2) + n = 4T(n/4) + 2n$

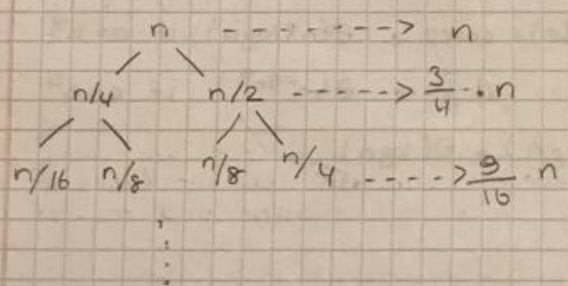
$= 4(2T(n/8) + n/4) + 2n = 8T(n/8) + 3n$

$= n^* T(1) + \log 2(n) ^* n$

$= O(n^* \log 2(n))$

$= \Theta(n \log n)$

f) $T(n) = T(n/2) + T(n/4) + n$



Totaly: $n\left(1 + \frac{3}{4} + \frac{9}{16} + \dots\right)$

$\Theta = (n)$

g) $T(n) = T(n/2) + n$ $\quad T(1) = 1 \quad T(2) = 3$

- Using Backward Subsition

$T(8) = T(4) + 8 \quad T(8) = 15 \rightarrow 2n - 1$

$T(4) = T(2) + 4 \quad T(4) = 7 \rightarrow 2n - 1$

$T(2) = T(1) + 2 \quad T(2) = 3 \rightarrow 2n - 1$

$T(1) = 1$

$\in \Theta(n)$

2. Provide pseudo code for the following operations on a given binary search tree (BST) with *n* nodes. Derive a recurrence relation for each of your algorithms. Calculate the average-case Θ() complexity of the derived recurrence relations.
   a) is_balanced(BST): This function checks whether the given binary search tree is balanced or not.
   b) height_of_tree(BST): This function returns the height of the given binary search tree.

2)

a) is_balanced (BST):

1. Define the base condition: if Bst is empty, return true becouse an empty tree is balanced.

2. Get the heigh of the left subtree (L) and right subtree (R).

3. If the absolute difference between the heights of L and R is less then or equal to 1, and both L and R are balanced, return True.

4. If Bst fails any of the conditions in steps 2-3, return False.

Recurrence Relation:

$T(n) = 2T(n/2) + O(1)$ (n is number of nodes).

we can use master teorem for average-case:

$a = 2$ here, $\log_b a = \log_2 2 = 1$
$b = 2$ $f(n) = O(1)$
$f(n) = O(1)$ $f(n) = O(n^c)$ where $c < \log_b a$, then
$T(n) = \Theta(n^{\log_b a})$

$T(n) = \Theta(n^{\log_b a}) = \Theta(n)$

b) heigh_of_tree ():

1. Define the base condition: if the BST is empty, return 0 becouse an empty tree has a heigh of 0.

2. Recursively calculate the heigh of the left subtree (L) and the right subtree (R)

3. Return the maximum heigh of L and R plus 1.

Recurrence Relation:

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

Average Case: $T(n) = \Theta(n^{\log_b a}) = \Theta(n)$

3. Suppose you are choosing between the following three algorithms:
   a) Algorithm A divides a problem with size *n* into five sub problems that are one-half the size, solves each one recursively, and then combines the results in cubic time.
   b) Algorithm B solves a problem with size *n* by resolving two sub problems of size *n*-2 recursively and then integrating the solutions in linear time.
   c) Algorithm C addresses issues of size *n* by dividing them into three subproblems, each half the size, solving each subproblem recursively, and then combining the solutions in $O(n^2)$ time.

   What is the running time of each algorithm (in terms of big -Oh notation), and which one would you choose? Provide a detailed explanation for your choice.

3-)

Algorithm A:

- Divides the problem into 5 subproblems of half the size.
- Solves each subproblem recursively.
- Combines the results in cubic time.

Recurrence Relation: $T(n) = 5T(\frac{n}{2}) + O(n^3)$

moster teorem:

$a = 5$
$b = 2$ $\quad 5 < 2^3$ $\quad$ if $a < b^d \rightarrow \Theta(n^d)$
$d = 3$ $\quad$ So runnig time : $\Theta(n^d) \rightarrow \Theta(n^3)$

Algorithm B:

- Solves a problem of size n by recursively solving two subproblems of size n-2
- Integrates the solution in linear time.

Recurrence Relation: $T(n) = 2T(n-2) + O(n)$

the recursive tree would have a depth of approximately n/2 levels because the problem size decreases by 2 with each recursive call. At each level, there is a linear cost of O(n).
The total time complexity is then the product of the cost per level and the number of levels:

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

Algorithm C:

- Divides the problem into 3 subproblems of half size.
- Solves each subproblem recursively.
- Combines the solutions in $O(n^2)$ time.

Recurrence Relation: $T(n): 3T(\frac{n}{2}) + O(n^2)$

moster teorem:

$a = 3$ $\quad f(n) = \Omega(n^{\log_b a + \epsilon})$
$b = 2$
$f(n) = O(n^2)$ $\quad$ here $\log_b a = \log_2 3$ and $f(n) = O(n^2) = O(n^{\log_2 3 + \epsilon})$
ϵ is a constant.
So
$T(n) = \Theta(f(n)) = \Theta(n^{\log_2 3})$

If I consider the worst-case time complexity alone, Algorithm C has the lowest asymptotic growth rate $O(n^{1.5})$. Therefore, if minimizing the running time is primary concern, Algorithm C would be the preferred choice.

4. The maximum cardinality matching problem in bipartite graphs involves finding the largest possible set of pairwise non-adjacent edges in a given bipartite graph. A bipartite graph is a graph in which the set of vertices can be divided into two disjoint sets, A and B, such that all edges connect a vertex from set A to a vertex in set B (and vice versa).

Provide a polynomial-time algorithm to compute a maximum cardinality matching in bipartite graphs and analyze the worst-case, best-case and average-case time complexity of the algorithm.

4-) The maximum cardinality matching problem in bipartite graphs is hopcroft-Karp algorithm

• Initialization:
  • Start with an empty matching.
  • While there exists an augmenting path, update the matching.

• Augmenting Path:
  • Use BFS to find an augmenting path in the bip graph.
  • If an augmenting path is found, update matching.

• Termination

The algorithm terminates when there are no more augmenting paths

→ number of vertex.
→ number of edges.

→ Worst Case: Time complexty: $O(\sqrt{V} \cdot E)$

• This occurs when the algorithm has to traverse the entire graph multiple times.

→ Best Case: $O(\sqrt{V} \cdot E)$

This is because, Algorithm may find an augmenting path quickly, but it still needs to iterate through th entire graph.

→ Average Case: $O(\sqrt{V} \cdot E)$

5. Write a recurrence relation to calculate the number of characters printed when the following function is called with input *n*.

```
foo(n):

   if n <= 1:
           return 1
else:
           for i in range(n):
             print("a")
 return foo (n / 2) + foo(n/2)
```

5-)

If n = 1, we have a base case that runs in constant time.
If n>1, we make two recursive calls and print "a" n times.

Recurrence Relation:

$$T(n) = \begin{cases} 1 & , \text{if } n <= 1 \\ 2T(n/2) + n & , \text{if } n > 1 \end{cases}$$

using master-teorem:

$a = 2$
$b = 2$ $\quad 2 = 2^1 \quad$ if $a = b^d$, $\Theta(n^c \log(n))$
$d = 1$

So time complexity: $\Theta(n \log(n))$

Notes:

- Your answer must be handwritten and submitted via the Course MS Teams page.
- Pseudocodes should be submitted as actual Python code and submitted as separate files together with your handwritten solutions.
- If you have any questions, you can send an email to b.koca@gtu.edu.tr
- Please complete your homework individually; group studies will be regarded as cheating.