Weekly Challenge 05 (Group): Divide and Conquer Recurrences

CS/CE 412/471 Algorithms: Design and Analysis

Spring 2025

Objective

In this WC, you will work in groups and:

- solve recurrence relations using recursion trees and any other method (unfolding, substitution, Master theorem, Akra-Bazzi)
- design efficient divide and conquer algorithms
- derive recurrence relations from recursive algorithms

Motivation

Understanding recurrences is crucial for analyzing the efficiency of recursive algorithms, which form the backbone of many algorithm design paradigms such as divide-and-conquer, dynamic programming, and parallel computing.

Many real-world problems, from sorting to pathfinding and optimization, rely on recursive formulations. This challenge will help you develop an intuition for how recurrences translate into computational complexity.

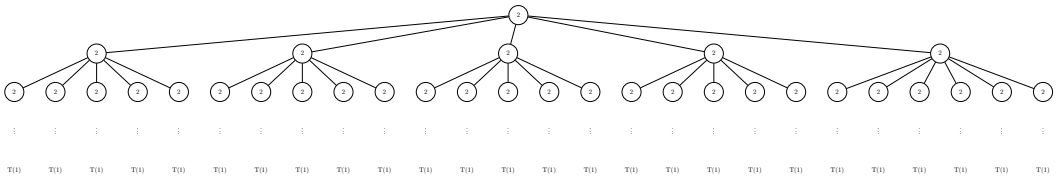
Problems

1. Consider the following recurrence relation for the Mystery divide-and-conquer algorithm:

$$T(n) = 5T\left(\frac{n}{3}\right) + 2$$

(a) Construct a recursion tree for the given recurrence.

Solution:



Explanation:

Since all levels could not be fully displayed in the diagram, imagine that the pattern remains, with each subproblem further subdivided into five smaller ones, each with size $\frac{n}{3}$ contributing a cost of 2, until it reaches base case $T(\frac{n}{3^k})$, where $\frac{n}{3^k} = 1$, indicating that the recursion level is $log_3 n$. Recursion ends at this stage.
The total number of leaf nodes at last level is $5^d = 5^{log_3 n}$. Using logarithmic properties we'll get $n^{log_3 5}$.

(b) Indicate the cost at each level.

Solution: The cost at level i is

$$5^i \cdot 2$$

where $0 \leq i \leq k$ (k is maximum depth of tree).

Detailed Breakdown:

- **Level 0:**

$$\text{Number Of Nodes} = 5^0 = 1, \quad \text{Cost Per Node} = 2$$

$$\text{Total Cost: } 1 \cdot 2 = 2$$

- **Level 1:**

$$\text{Number Of Nodes} = 5^1 = 5, \quad \text{Cost Per Node} = 2$$

$$\text{Total Cost: } 5 \cdot 2 = 10$$

- **Level 2:**

$$\text{Number Of Nodes} = 5^2 = 25, \quad \text{Cost Per Node} = 2$$

$$\text{Total Cost: } 5^2 \cdot 2 = 50$$

- **Level 3:**

$$\text{Number Of Nodes} = 5^3 = 125, \quad \text{Cost Per Node} = 2$$

$$\text{Total Cost: } 5^3 \cdot 2 = 250$$

$\vdots$

- **Level k:**

$$\text{Number Of Nodes} = 5^k, \quad \text{Cost Per Node} = 2$$

$$\text{Total Cost: } 5^k \cdot 2$$

where $k = log_3(n)$ because according to base condition of recursion tree $\frac{n}{3^k} = 1$,

$$\text{Total Cost: } 5^{log_3(n)} \cdot 2$$

(c) Add up the costs over all levels to determine the cost for the entire tree. Show your working.

Solution: Since the base condition of recursion tree is,

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

$$n = 3^k$$

$$k = log_3(n)$$

Now we add up the costs over all levels to determine the cost for the entire tree:

$$C = 5^0 \cdot 2 + 5^1 \cdot 2 + 5^2 \cdot 2 + \cdots + 5^k \cdot 2$$

$$= \sum_{i=0}^{k} 5^i \cdot 2$$

Replacing k with it's value,

$$= \sum_{i=0}^{\log_3(n)} 5^i \cdot 2$$

$$= 2 \cdot \sum_{i=0}^{\log_3(n)} 5^i$$

Using sum of geometric series,

$$= 2 \cdot \frac{5^{\log_3(n)+1} - 1}{4}$$

$$= \frac{1}{2} \cdot \left(5^{\log_3(n)+1} - 1\right)$$

Using the property of logarithms we get $5^{\log_3(n)} = n^{\log_3(5)}$, so now

$$= \frac{1}{2} \cdot \left(n^{\log_3(5)} \cdot 5 - 1\right)$$

The term $n^{\log_3(5)} \cdot 5$ dominates, so

$$C = \Theta(n^{\log_3(5)}).$$

So the total cost for entire tree is $\Theta(n^{\log_3(5)})$.

2. You are walking in a park with n trees arranged in a straight line. Each tree has a distinct height. A *dip* is defined as a tree that is shorter than both of its adjacent trees (or just one adjacent tree for the first and last tree).

   You want to find the height of the shortest dip.

   (a) Design `FindDip`, an efficient, divide-and-conquer procedure that takes as input an array of tree heights and returns the height of the shortest dip. (use pseudo-code notation given in CLRS)

Solution:

```
1:  procedure FindDip(A, low, high)
2:      if low == high then
3:          return A[low]
4:      if low + 1 == high then
5:          return min(A[low], A[high])
6:      mid ← ⌊(low + high)/2⌋
7:      if (mid == low or A[mid] < A[mid−1]) and (mid == high or A[mid] < A[mid+1])
    then
8:          return A[mid]
9:      left_dip ← FindDip(A, low, mid - 1)
10:     right_dip ← FindDip(A, mid + 1, high)
11:     return min(left_dip, right_dip)
```

   (b) Express the running time of your procedure as a recurrence relation.

Solution:
Divide: The middle index of the subarray is calculated in a constant amount of time by the division step. Thus,

$$D(n) = \Theta(1)$$

Conquer: We recursively solve two subproblems, since each subproblem is of size $\frac{n}{2}$. The running time is influenced by this in the amount $2T(\frac{n}{2})$.

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

Combine: Since the result is returned directly from the recursive calls, there is no need for an explicit combine step. So,

$$C(n) = \Theta(1)$$

The running time of the procedure can be expressed as,

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 2, \\ D(n) + 2T\left(\frac{n}{2}\right) + C(n) & \text{otherwise.} \end{cases}$$

Substituting $D(n) = \Theta(1)$ and $C(n) = \Theta(1)$, we have

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(1)$$

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Fourth Edition.*

(c) Find out the time complexity of your procedure by solving the recurrence relation using any method of your choice (iterative/substitution/recursion tree/Master/Akra-Bazzi). Show your working.

Solution: The given recurrence is in the form:

$$T(n) = aT(n/b) + f(n)$$

We'll use Master Theorem,

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(1)$$

Here, $a = 2$, $b = 2$, and $f(n) = \Theta(1)$.
We'll plug our values in watershed function $n^{\log_b a}$,

$$n^{\log_b a} = n^{\log_2 2} = n^1$$

Now we'll compare $f(n) = \Theta(1)$ with n^1. Since $f(n) = \Theta(1) = O(n^{log_2 2-\epsilon})$ for some $\epsilon = 1$, it follows that $f(n)$ grows asymptotically slower than $n^{\log_b a}$, which aligns with property of **Master Theorem Case 1**.

Master Theorem Case 1

If there exists a constant $\epsilon > 0$ such that:

$$f(n) = O(n^{\log_b a - \epsilon}),$$

then:

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

So by applying Case 1 of the Master Theorem, we conclude that

$$T(n) = \Theta(n^{\log_2(2)}) = \Theta(n)$$

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Fourth Edition.*