Algorithm Analysis

Kumkum Saxena

Contents

- Introduction to advanced data structures:
- Introduction/Fundamentals of the analysis of algorithms
- Recurrences:
  - The substitution method
  - Recursive tree method
  - Masters method

Why performance analysis?

- There are many important things that should be taken care of, like user friendliness, modularity, security, maintainability, etc.
- Why to worry about performance? The answer to this is simple, we can have all the above things only if we have performance.
- So performance is like currency through which we can buy all the above things.

Given two algorithms for a task, how do we find out which one is better?

- One naive way of doing this is – implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time.
- There are many problems with this approach for analysis of algorithms.
  - It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.
  - It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.

Asymptotic Analysis

- It is the big idea that handles above issues in analysing algorithms.
- In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don’t measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the input size.

- For example, let us consider the search problem (searching a given item) in a sorted array. One way to search is Linear Search (order of growth is linear) and other way is Binary Search (order of growth is logarithmic).
- To understand how Asymptotic Analysis solves the above mentioned problems in analysing algorithms, let us say we run the Linear Search on a fast computer and Binary Search on a slow computer.
- For small values of input array size n, the fast computer may take less time. But, after certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine.
- The reason is the order of growth of Binary Search with respect to input size logarithmic while the order of growth of Linear Search is linear. So the machine dependent constants can always be ignored after certain values of input size.

Does Asymptotic Analysis always work?

- Asymptotic Analysis is not perfect, but that's the best way available for analysing algorithms.
- For example, say there are two sorting algorithms that take 1000nLogn and 2nLogn time respectively on a machine.
- So, With Asymptotic Analysis, we can't judge which one is better as we ignore constants in Asymptotic Analysis.
- Also, in Asymptotic analysis, we always talk about input sizes larger than a constant value.
- It might be possible that those large inputs are never given to your software and an algorithm which is asymptotically slower, always performs better for your particular situation.
- So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

- We can have three cases to analyze an algorithm:
  - Worst Case(upper bound)-In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed.
  - Average Case-In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases.

- Best Case(lower bound)-In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed.

- Most of the times, we do worst case analysis to analyse algorithms.
- In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.
- The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.
- The Best Case analysis is not required. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

Asymptotic Analysis

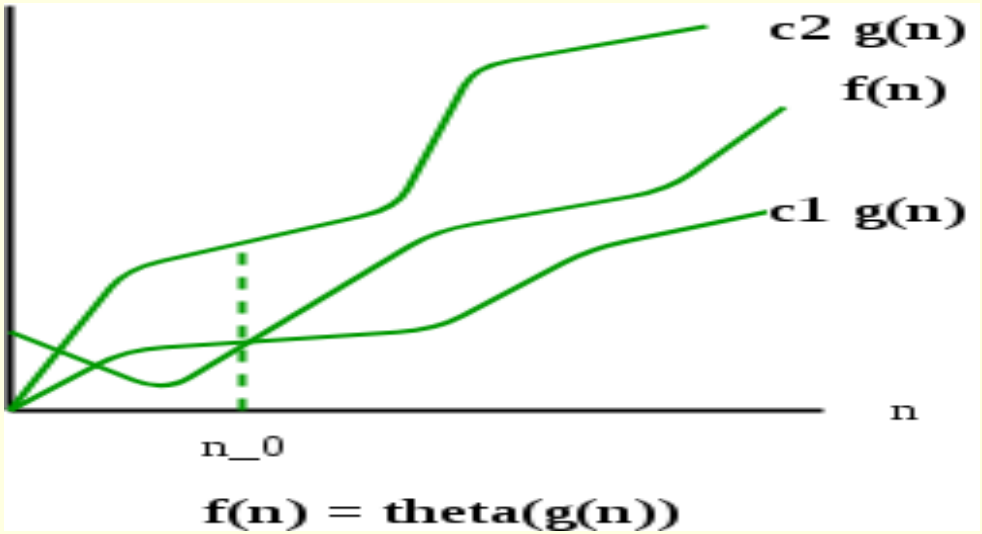
- The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared.
- Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.

Asymptotic notations

- **1) Θ Notation(Average case):** The theta notation bounds a functions from above and below, so it defines exact asymptotic behaviour.
- A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants.
- For example, consider the following expression. $3n^3 + 6n^2 + 6000 = \Theta(n^3)$
- Dropping lower order terms is always fine because there will always be a n0 after which $\Theta(n^3)$ has higher values than $\Theta n^2)$ irrespective of the constants involved.

- For a given function g(n), we denote Θ(g(n)) is following set of functions.

$$\Theta(g(n)) = \{f(n): \text{there exist positive constants } c1, c2 \text{ and } n0 \text{ such that } 0 <= c1*g(n) <= f(n) <= c2*g(n) \text{ for all } n >= n0\}$$

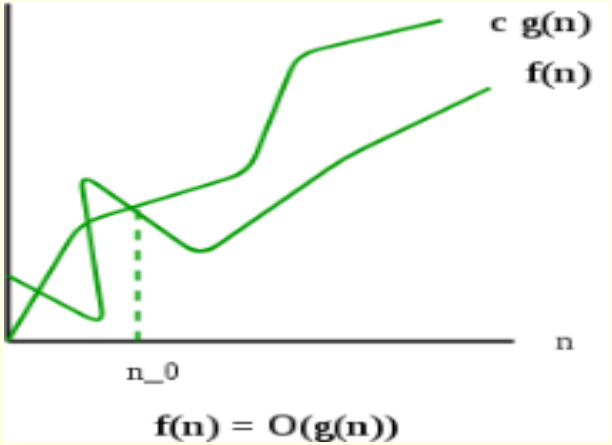


- The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0).
- The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.

- **2) Big O Notation(Worst Case):** The Big O notation defines an upper bound of an algorithm, it bounds a function only from above.
- For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case.
- We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.
- If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:
  - The worst case time complexity of Insertion Sort is Θ(n^2).
  - The best case time complexity of Insertion Sort is Θ(n).

- The Big O notation is useful when we only have upper bound on time complexity of an algorithm.

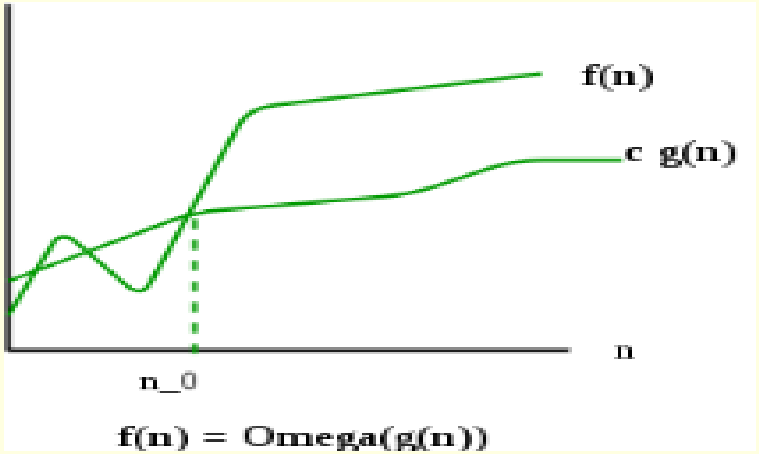
```
O(g(n)) = { f(n): there exist positive constants c and
                  n0 such that 0 <= f(n) <= c*g(n) for
                  all n >= n0}
```



- **3) Ω Notation(Best Case):** Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.
- Ω Notation can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previously, the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.

- For a given function g(n), we denote by Ω(g(n)) the set of functions.

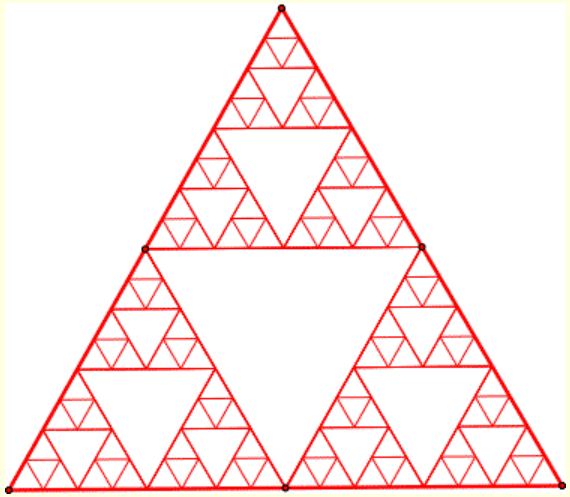
```
Ω (g(n)) = {f(n): there exist positive constants c and
                  n0 such that 0 <= c*g(n) <= f(n) for
                  all n >= n0}.
```



Some Functions

1. **O(1):** Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.
2. **O(n):** Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount.
3. **O(n^c)**: Time complexity of nested loops is equal to the number of times the innermost statement is executed.
4. **O(Logn)** Time Complexity of a loop is considered as O(Logn) if the loop variables is divided / multiplied by a constant amount.
5. **O(LogLogn)** Time Complexity of a loop is considered as O(LogLogn) if the loop variables is reduced / increased exponentially by a constant amount.

Recurrence Relation



Sierpinski Triangle

How to calculate time complexity of recursive functions?

- Time complexity of a recursive function can be written as a mathematical recurrence relation.
- To calculate time complexity, we must know how to solve recurrences.

Sequences

Consider the following two sequences:

$$S_1 : 3, 5, 7, 9, \ldots$$

$$S_2 : 3, 9, 27, 81, \ldots$$

We can find a formula for the nth term of sequences S_1 and S_2 by observing the pattern of the sequences.

$$S_1 : 2 \cdot 1 + 1, 2 \cdot 2 + 1, 2 \cdot 3 + 1, 2 \cdot 4 + 1, \ldots$$

$$S_2 : 3^1, 3^2, 3^3, 3^4, \ldots$$

For S_1, $a_n = 2n + 1$ for $n \geq 1$, and for S_2, $a_n = 3^n$ for $n \geq 1$. This type of formula is called an **explicit formula** for the sequence, because using this formula we can directly find any term of the sequence without using other terms of the sequence. For example, $a_3 = 2 \cdot 3 + 1 = 7$.

Recurrence

Let S denote the sequence

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots .$$

For this sequence, the explicit formula is not obvious. If we observe closely, however, we find that the pattern of the sequence is such that any term after the second term is the sum of the preceding two terms. Now

$$\begin{aligned}
&\text{3rd term} = 2 = 1 + 1 = \text{1st term} + \text{2nd term}\\
&\text{4th term} = 3 = 1 + 2 = \text{2nd term} + \text{3rd term}\\
&\text{5th term} = 5 = 2 + 3 = \text{3rd term} + \text{4th term}\\
&\text{6th term} = 8 = 3 + 5 = \text{4th term} + \text{5th term}\\
&\text{7th term} = 13 = 5 + 8 = \text{5th term} + \text{6th term}
\end{aligned}$$

Hence, the sequence S can be defined by the equation

$$f_n = f_{n-1} + f_{n-2} \tag{8.1}$$

for all $n \geq 3$ and

$$\begin{aligned} f_1 &= 1,\\ f_2 &= 1. \end{aligned} \tag{8.2}$$

Consider the function $f : \mathbb{N}^0 \to \mathbb{Z}^+$ defined by

$$f(0) = 1,$$
$$f(n) = nf(n-1) \quad \text{for all } n \geq 1.$$

Then

$$f(0) = 1 = 0!,$$
$$f(1) = 1 \cdot f(0) = 1 = 1!,$$
$$f(2) = 2 \cdot f(1) = 2 \cdot 1 = 2 = 2!,$$
$$f(3) = 3 \cdot f(2) = 3 \cdot 2 \cdot 1 = 6 = 3!,$$

and so on. Here $f(n) = nf(n-1)$ for all $n \geq 1$ is the recurrence relation, and $f(0) = 1$ is the initial condition for the function f. Notice that the function f is nothing but the factorial function, i.e., $f(n) = n!$ for all $n \geq 0$.

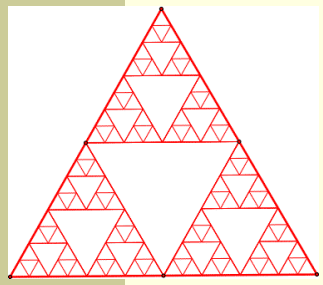
Definition

A **recurrence relation** for a sequence $a_0, a_1, a_2, \ldots, a_n, \ldots$ is an equation that relates a_n to some of the terms $a_0, a_1, a_2, \ldots, a_{n-2}, a_{n-1}$ for all integers n with $n \geq k$, where k is a nonnegative integer. The **initial conditions** for the recurrence relation are a set of values that explicitly define some of the members of $a_0, a_1, a_2, \ldots, a_{k-1}$.

The equation

$$a_n = 2a_{n-1} + a_{n-2} \quad \text{for all } n \geq 2,$$

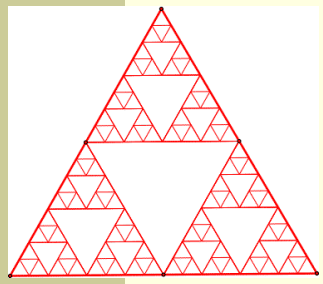
as defined above, relates a_n to a_{n-1} and a_{n-2}. Here $k = 2$. So this is a recurrence relation with initial conditions $a_0 = 5$ and $a_1 = 7$.



Recursive Definition

- Write the recursive definition the following sequences:
- 1,2,3,4,5……
  - – a1=1
  - – $a_n=a_{n-1}+1$, $n\geq 2$
- 2,4,8,16….
  - – a1=2
  - – $a_n=2a_{n-1}+1$, $n\geq 2$

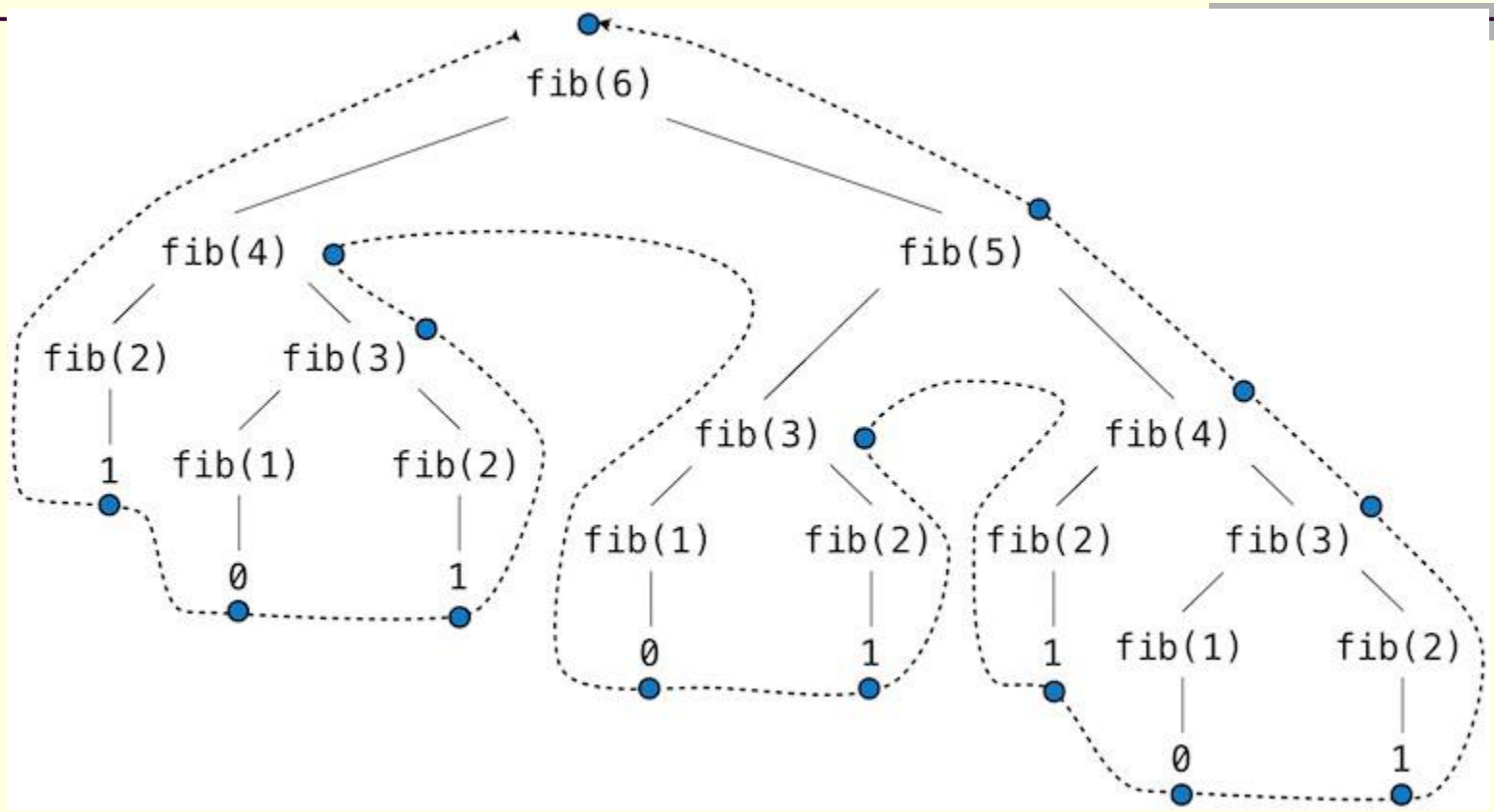
- Basic step: This step defines a primitive value or set of primitive values.
- Recursive step: This step defines the rule(s) to find a new element from existing element.

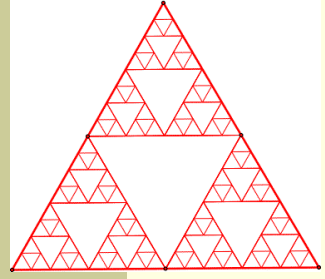


Fibonacci sequence

- Initial conditions:
  - $f_1 = 1, f_2 = 2$
- Recursive formula:
  - $f_{n+1} = f_{n-1} + f_n$ for $n \geq 3$
- First few terms:

n	1	2	3	4	5	6	7	8	9	10	11
f_n	1	2	3	5	8	13	21	34	55	89	144

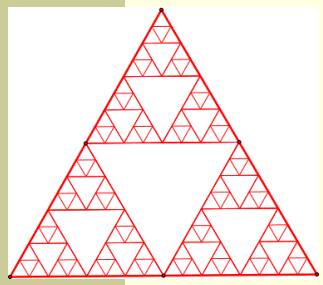

fib(6)
fib(4)
fib(5)
fib(2)
fib(3)
fib(3)
fib(4)
1
fib(1)
fib(2)
fib(1)
fib(2)
fib(2)
fib(3)
0
1
0
1
1
fib(1)
fib(2)
0
1



Recursive Algorithms

- Definition
  - An algorithm that calls itself
- Components of a recursive algorithm

1. Base cases
   - Computation with no recursion
2. Recursive cases
   - Recursive calls
   - Combining recursive results

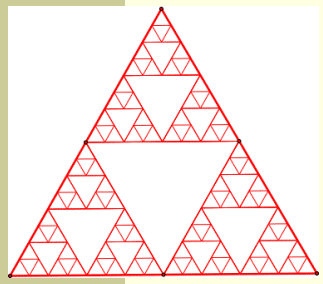


Example

- Code (for input size n)

```
1.DoWork (int n)
2.  if (n == 1)
3.        A
4.     else
5.        DoWork(n/2)
6.        DoWork(n/2)
```

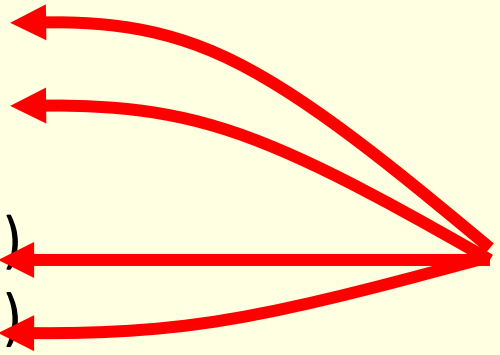
- Code execution
  - A ⇒
  - DoWork(n/2) ⇒
- Time(1) ⇒ Time(n) =



Example

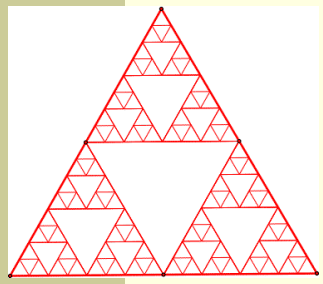
- Code (for input size n)

```
1.DoWork (int n)
2.  if (n == 1)
3.        A
4.     else
5.        DoWork(n/2)
6.        DoWork(n/2)
```



critical sections

- Code execution
  - A $\Rightarrow$ 1 times
  - DoWork(n/2) $\Rightarrow$ 2 times
- Time(1)=1 Time(n) = 2 $\times$ Time(n/2) + 1



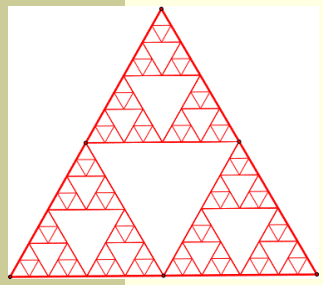
Recurrence Relations

A **recurrence relation** is an equation that describes a function in terms of itself by using smaller inputs

The expression:

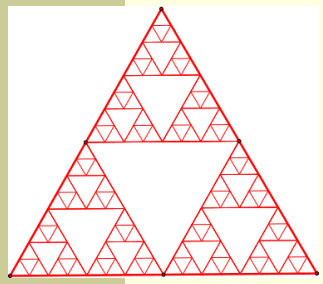
$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

describes the **running time** for a function contains recursion.



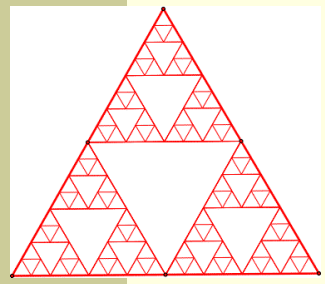
Recurrence Relations

- Definition
  - Value of a function at a point is given in terms of its value at other points
- Examples
  - $T(n) = T(n-1) + k$
  - $T(n) = T(n-1) + n$
  - $T(n) = T(n-1) + T(n-2)$
  - $T(n) = T(n/2) + k$
  - $T(n) = 2 \times T(n/2) + k$



Recurrence Relations

- Base case
  - Value of function at some specified points
  - Also called boundary values / boundary conditions
- Base case example
  - T(1) = 0
  - T(1) = 1
  - T(2) = 1
  - T(2) = k



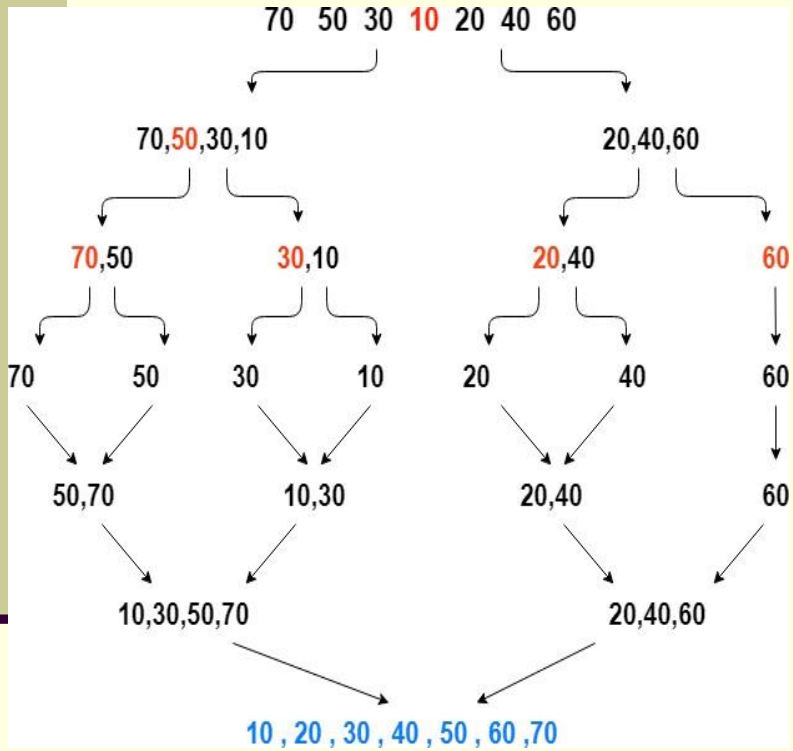
Recurrence Relations

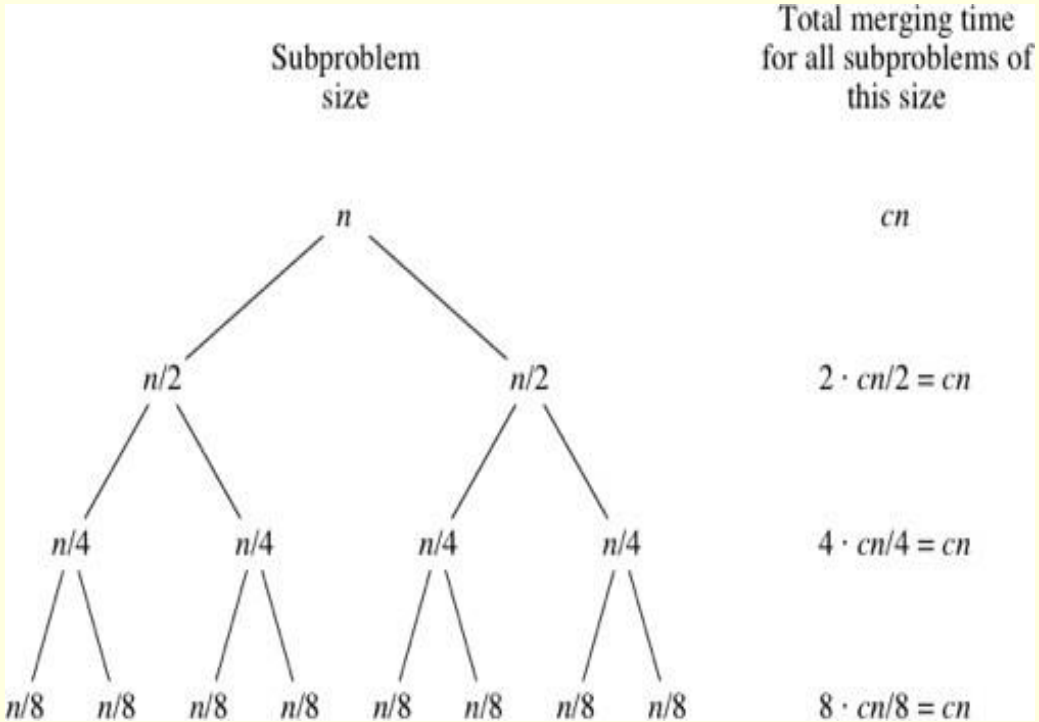
- Divide and conquer is a very common problem solving strategy in computer science
- Recursion, where you solve a simpler version of the same problem, is a common application of divide and conquer
  - the recursion "bottoms out" in a base case
  - Merge sort divides a problem in "half" and solves each half separately
  - The recurrence relation is

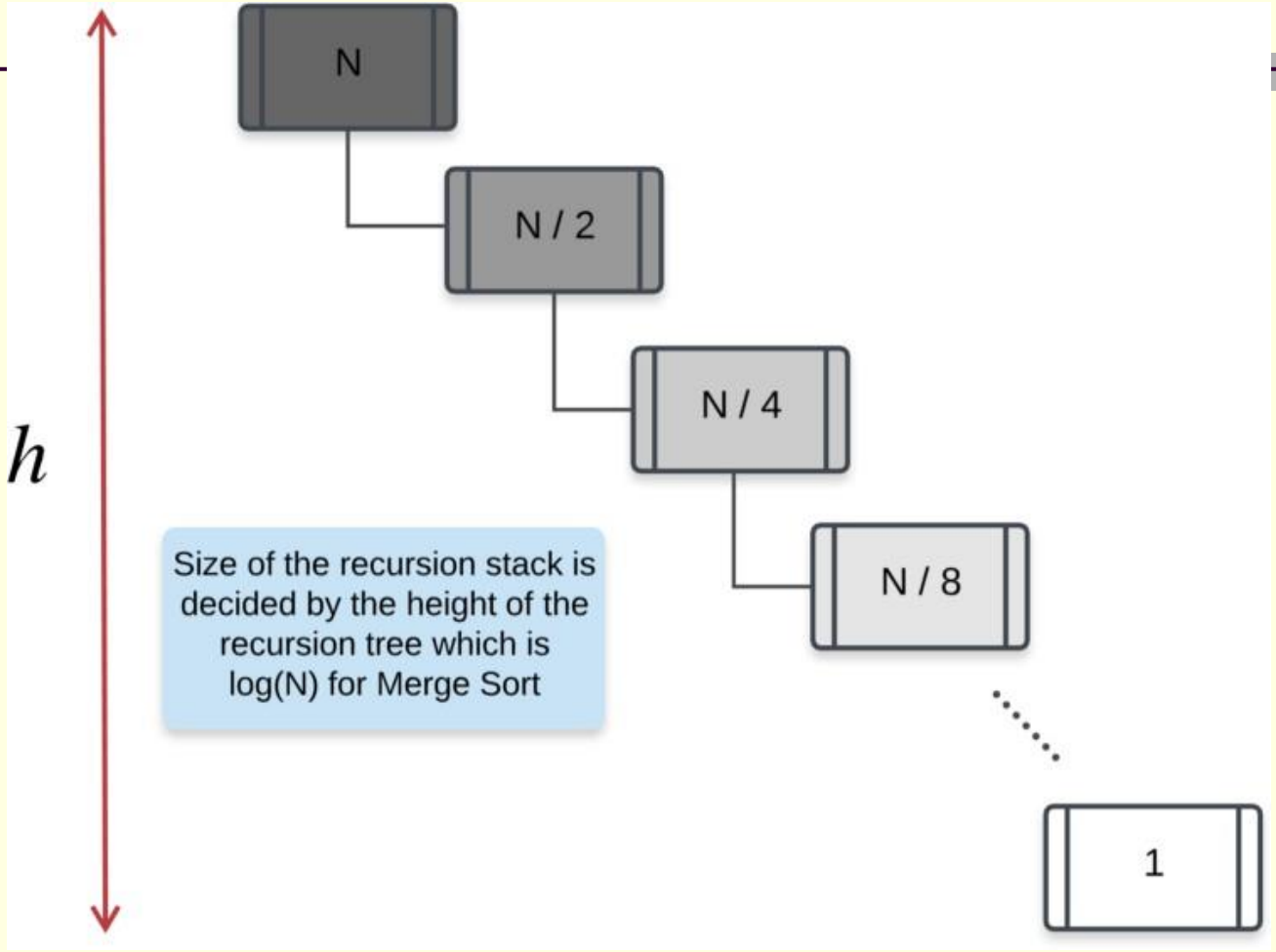
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1, \end{cases}$$

  - Binary search divides a problem in half but only has to solve the problem for one of the two halves

Example : Merge Sort






N
N / 2
N / 4
N / 8
1
h
Size of the recursion stack is decided by the height of the recursion tree which is log(N) for Merge Sort

There is constant amount of work being done at each level. Thus, the height of the tree defines the total amount of work done. As we've seen before, the height of this tree is O(logN) and so, the overall complexity is O(logN). **Note:** O(1) = C

$$T(n) = C + C + C + C + \cdots + C$$

$$T(n) = O(1) + O(1) + O(1) + \cdots + O(1)$$

$$T(n) = O(1) \times log_2 n = O(log_2 n)$$

Search for 47

0	4	7	10	14	23	45	47	53

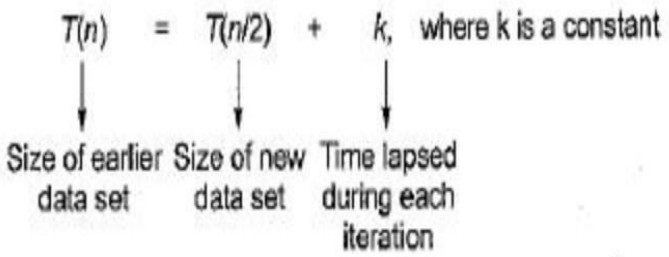
QUESTION: 8

The recurrence relation that arises in relation with the complexity of binary search is

- **A.** T{n) = T{n/2) + k, where k is a constant
- **B.** T(n) = 27(n/2) + k, where k is a constant
- **C.** T{n) = T(n/2) + log(n)
- **D.** T(n) = T(n/2) + n

Solution:

Since, with every iteration of Binary Search, data set to be searched is divided into half of the original/earlier dataset. Also, same constant time is lapsed during each iteration. So, recurrence relation for Binary Search would be:



Common Ways to Solve Recurrence Relations

- There are several common ways to solve a recurrence relation
  - The **substitution** method (guess and then check)
  - The **recurrence tree** method (sum up complexity at each level, then sum all the levels)
  - The **master method** (prove a general theorem once then apply the theorem where appropriate; covers many common cases but not all cases)

A *recurrence relation*

- is an infinite sequence $a_1, a_2, a_3, \ldots, a_n, \ldots$
- in which the formula for the nth term a_n depends on one or more preceding terms,
- with a finite set of start-up values or **initial conditions**

Fibonacci sequence

- Initial conditions:
  - $f_1 = 1, f_2 = 2$
- Recursive formula:
  - $f_{n+1} = f_{n-1} + f_n$ for $n \geq 3$
- First few terms:

n	1	2	3	4	5	6	7	8	9	10	11
f_n	1	2	3	5	8	13	21	34	55	89	144

Contents

- Introduction to advanced data structures:
- Introduction/Fundamentals of the analysis of algorithms
- Recurrences:
  - The substitution method
  - Recursive tree method
  - Masters method

Solving Recurrences

- Substitution method
- Recursive Tree
- Master method

Solving Recurrences

- The substitution method (CLR 4.1)
  - "Making a good guess" method
  - Guess the form of the answer, then use induction to find the constants and show that solution works
  - Examples:
    - $T(n) = 2T(n/2) + \Theta(n)$ □ $T(n) = \Theta(n \lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor) + n$ □ ???

Solving Recurrences

- The substitution method (CLR 4.1)
  - "Making a good guess" method
  - Guess the form of the answer, then use induction to find the constants and show that solution works
  - Examples:
    - $T(n) = 2T(n/2) + \Theta(n) \rightarrow T(n) = \Theta(n \lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor) + n \rightarrow T(n) = \Theta(n \lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor) + 17) + n \rightarrow$???

Substitution method

- *Guess the form of the solution .*
- *Use mathematical induction to find the constants and show that the solution works .*

The substitution method can be used to establish either upper or lower bounds on a recurrence.

Linear Search

Input: Array A, an element x

Question: Is $x \in A$

The idea behind linear search is to search the given element x linearly (sequentially) in the given array. A recursive approach to linear search first searches the given element in the first location, and if not found it recursively calls the linear search with the modified array without the first element i.e., the problem size reduces by one in the subsequent calls. Let $T(n)$ be the number of comparisons (time) required for linear search on an array of size n. Note when $n = 1$, $T(1) = 1$. Then, $T(n) = 1 + T(n-1) = 1 + \cdots + 1 + T(1)$ and $T(1) = 1$ Therefore, $T(n) = n - 1 + 1 = n$, i.e., $T(n) = \Theta(n)$.

Binary search

Input: Sorted array A of size n, an element x to be searched

Question: Is $x \in A$

Approach: Check whether $A[n/2] = x$. If $x > A[n/2]$, then prune the lower half of the array, $A[1,\ldots,n/2]$. Otherwise, prune the upper half of the array. Therefore, pruning happens at every iterations. After each iteration the problem size (array size under consideration) reduces by half. Recurrence relation is $T(n) = T(n/2) + O(1)$, where $T(n)$ is the time required for binary search in an array of size n. $T(n) = T(\frac{n}{2^k}) + 1 + \cdots + 1$

Since $T(1) = 1$, when $n = 2^k$, $T(n) = T(1) + k = 1 + \log_2(n)$.

$\log_2(n) \leq 1 + \log_2(n) \leq 2\log_2(n) \; \forall \; n \geq 2$.

$T(n) = \Theta(\log_2(n))$.

For example consider the recurrence T(n) = 2T(n/2) + n

We guess the solution as T(n) = O(nLogn). Now we use induction to prove our guess.

We need to prove that T(n) <= cnLogn. We can assume that it is true for values smaller than n.

T(n) = 2T(n/2) + n
<= cn/2Log(n/2) + n
= cnLogn - cnLog2 + n
= cnLogn - cn + n
<= cnLogn

An example (Substitution method)

- T(n) = 2T(floor(n/2)) +n

We guess that the solution is T(n)=0(n lg n).

i.e. to show that $T(n) \leq cn \lg n$, for some constant $c > 0$ and $n \geq m$.

Assume that this bound holds for [n/2]. So , we get

$T(n) \leq 2(c \text{ floor } (n/2) \lg(\text{floor}(n/2))) + n$

$\leq cn \lg(n/2) + n$

$= cn \lg n - cn \lg 2 + n$

$= cn \lg n - cn + n$

$\leq cn \lg n$

where , the last step holds as long as $c \geq 1$.

Similar to binary search, the ternary search compares x with $A[n/3]$ and $A[2n/3]$ and the problem size reduces to $n/3$ for the next iteration. Therefore, the recurrence relation is $T(n) = T(n/3) + 2$, and $T(2) = 2$. Note that there are two comparisons done at each iteration and due to which additive factor '2' appears in $T(n)$.

$T(n) = T(n/3) + 2; \Rightarrow T(n/3) = T(n/9) + 2$

$\Rightarrow T(n) = T(n/(3^k)) + 2 + 2 + ... + 2$ (2 appears k times)

When $n = 3^k$, $T(n) = T(1) + 2 \times \log_3(n) = \Theta(\log_3(n))$

Further, we highlight that for k-way search, $T(n) = T(\frac{n}{k}) + k - 1$ where $T(k-1) = k - 1$. Solving this, $T(n) = \Theta(\log_k(n))$.

It is important to highlight that in asymptotic sense, binary search, ternary search and k-way search (fixed k) are having same complexity as $\log_2 n = \log_2 3 \cdot \log_3 n$ and $\log_2 n = \log_2 k \cdot \log_k n$. So, $\theta(\log_2 n) = \theta(\log_3 n) = \theta(\log_k n)$, for fixed k.

Recurrence relation using change of variable technique

Problem: 1 $T(n) = 2 * T(\sqrt{n}) + 1$ and $T(1) = 1$

Introduce a change of variable by letting $n = 2^m$.
$\Rightarrow T(2^m) = 2 \times T(\sqrt{2^m}) + 1$

$\Rightarrow T(2^m) = 2 \times T(2^{m/2}) + 1$
Let us introduce another change by letting $S(m) = T(2^m)$
$\Rightarrow S(m) = 2 \times S(m/2) + 1$

$\Rightarrow S(m) = 2 \times (2 \times S(m/4) + 1) + 1$

$\Rightarrow S(m) = 2^2 \times S(m/2^2) + 2 + 1$

By substituting further,

$\Rightarrow S(m) = 2^k \times S(m/2^k) + 2^{k-1} + 2^{k-2} + ... + 2 + 1$

To simplify the expression, assume $m = 2^k$
$\Rightarrow S(m/2^k) = S(1) = T(2)$. Since $T(n)$ denote the number of comparisons, it has to be an integer always. Therefore, $T(2) = 2 \times T(\sqrt{2}) + 1$, which is approximately 3.

$\Rightarrow S(m) = 3 + 2^k - 1 \Rightarrow S(m) = m + 2.$
We now have, $S(m) = T(2^m) = m+2$. Thus, we get $T(n) = m+2$, Since $m = \log n$, $T(n) = \log n+2$
Therefore, $T(n) = \theta(\log n)$

Sorting

To sort an array of n elements using find-max (returns maximum) as a black box.

Approach: Repeatedly find the maximum element and remove it from the array. The order in which the maximum elements are extracted is the sorted sequence. The recurrence for the above algorithm is,

$$T(n) = T(n-1) + n - 1 = T(n-2) + n - 2 + n - 1 = T(1) + 1 + \cdots + n - 1 = \frac{(n-1)n}{2}$$

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

Merge Sort

Approach: Divide the array into two equal sub arrays and sort each sub array recursively. Do the sub-division operation recursively till the array size becomes one. Trivially, the problem size one is sorted and when the recursion bottoms out two sub problems of size one are combined to get a sorting sequence of size two, further, two sub problems of size two (each one is sorted) are combined to get a sorting sequence of size four, and so on. We shall see the detailed description of merge sort when we discuss divide and conquer paradigm. The recurrence for the merge sort is,

$$T(n) = 2T(\frac{n}{2}) + n - 1 = 2[2T(\frac{n}{2^2}) + \frac{n}{2} - 1] + n - 1$$

$$\Longrightarrow 2^k T(\frac{n}{2^k}) + n - 2^k + n - 2^{k-1} + \cdots + n - 1.$$

When $n = 2^k$, $T(n) = 2^k T(1) + n + \cdots + n - [2^{k-1} + \cdots + 2^0]$

Note that $2^{k-1} + \cdots + 2^0 = \dfrac{2^{k-1+1} - 1}{2 - 1} = 2^k - 1 = n - 1$

Also, $T(1) = 0$ as there is no comparison required if $n = 1$. Therefore, $T(n) = n\log_2(n) - n + 1 = \Theta(n\log_2(n))$

Heap sort

This sorting is based on the data structure *max-heap* which we shall discuss in detail at a later chapter. Creation of max-heap can be done in linear time. The approach is to delete the maximum element repeatedly and set right the heap to satisfy max-heap property. This property maintenance incur $O(\log n)$ time. The order in which the elements are deleted gives the sorted sequence. Number of comparisons needed for deleting an element is equal to at most the height of the max-heap, which is $\log_2(n)$. Therefore the recurrence for heap sort is,

$T(n) = T(n-1) + \log_2(n)$ where $T(1) = 0 \Rightarrow T(n) = (T(n-2) + \log_2(n-1)) + \log_2(n)$

By substituting further, $\Rightarrow T(n) = T(1) + \log 2 + \log 3 + \log 4 + ... + \log n$

$\Rightarrow T(n) = \log(2.3.4...n) \Rightarrow T(n) = \log(n!)$

$\Rightarrow \log(n!) \leq n \log n$ as $n! \leq n^n$ (Stirling's Approximation)

$\Rightarrow T(n) = \Theta(n \log_2(n))$

Problem: 2 $T(n) = 2 * T(\sqrt{n}) + n$ and $T(1) = 1$

$$\text{Let } n = 2^m \Rightarrow T(2^m) = 2 \times T(\sqrt{2^m}) + 2^m$$

$$\Rightarrow T(2^m) = 2 \times T(2^{m/2}) + 2^m$$

let $S(m) = T(2^m) \Rightarrow S(m) = 2 \times S(m/2) + 2^m$

$\Rightarrow S(m) = 2 \times (2 \times S(m/4) + 2^{m/2}) + 2^m$

$\Rightarrow S(m) = 2^2 \times S(m/2^2) + 2.2^{m/2} + 2^m$

By substituting further, we see that

$\Rightarrow S(m) = 2^k \times S(m/2^k).2^{m/2^k} + 2^{k-1}.2^{m/2^{k-1}} + 2^{k-2}.2^{m/2^{k-2}} + ... + 2.2^{m/2} + 1.2^m$

An easy upper bound for the above expression is;

$S(m) \leq 2^k \times S(m/2^k).2^m + 2^{k-1}.2^m + 2^{k-2}.2^m + ... + 2.2^m + 1.2^m$

$S(m) = 2^k \times S(m/2^k).2^m + 2^m[2^{k-1} + 2^{k-2} + ... + 2 + 1]$

To simplify the expression further, we assume $m = 2^k$

$S(m) \leq 2^k S(1).2^m + 2^m(2^k - 1)$

Since $S(1) = T(4)$, which is approximately, $T(4) = 4$.

$S(m) \leq 4m2^m + 2^m(m - 1)$

$S(m) = O(m.2^m), T(2^m) = O(m.2^m), T(n) = (n.\log n)$.

Is it true that $T(n) = \Omega(n.\log n)$?

From the first term of the above expression, it is clear that $S(m) \geq m.2^m$, therefore, $T(n) = \Omega(n.\log n)$

Contents

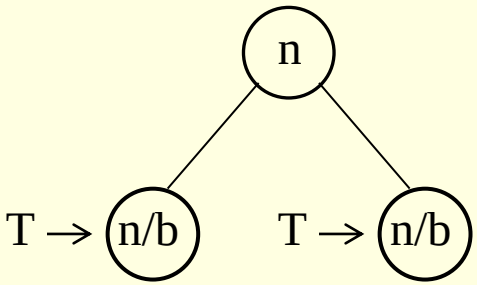
- Introduction to advanced data structures:
- Introduction/Fundamentals of the analysis of algorithms
- Recurrences:
  - The substitution method
  - Recursive tree method
  - Masters method
- Probabilistic analysis
- Amortized analysis
- Randomized algorithms
- Mathematical aspects and analysis of algorithms

Solving Recurrences using Recursion Tree Method

- Here while solving recurrences, we divide the problem into subproblems of equal size.

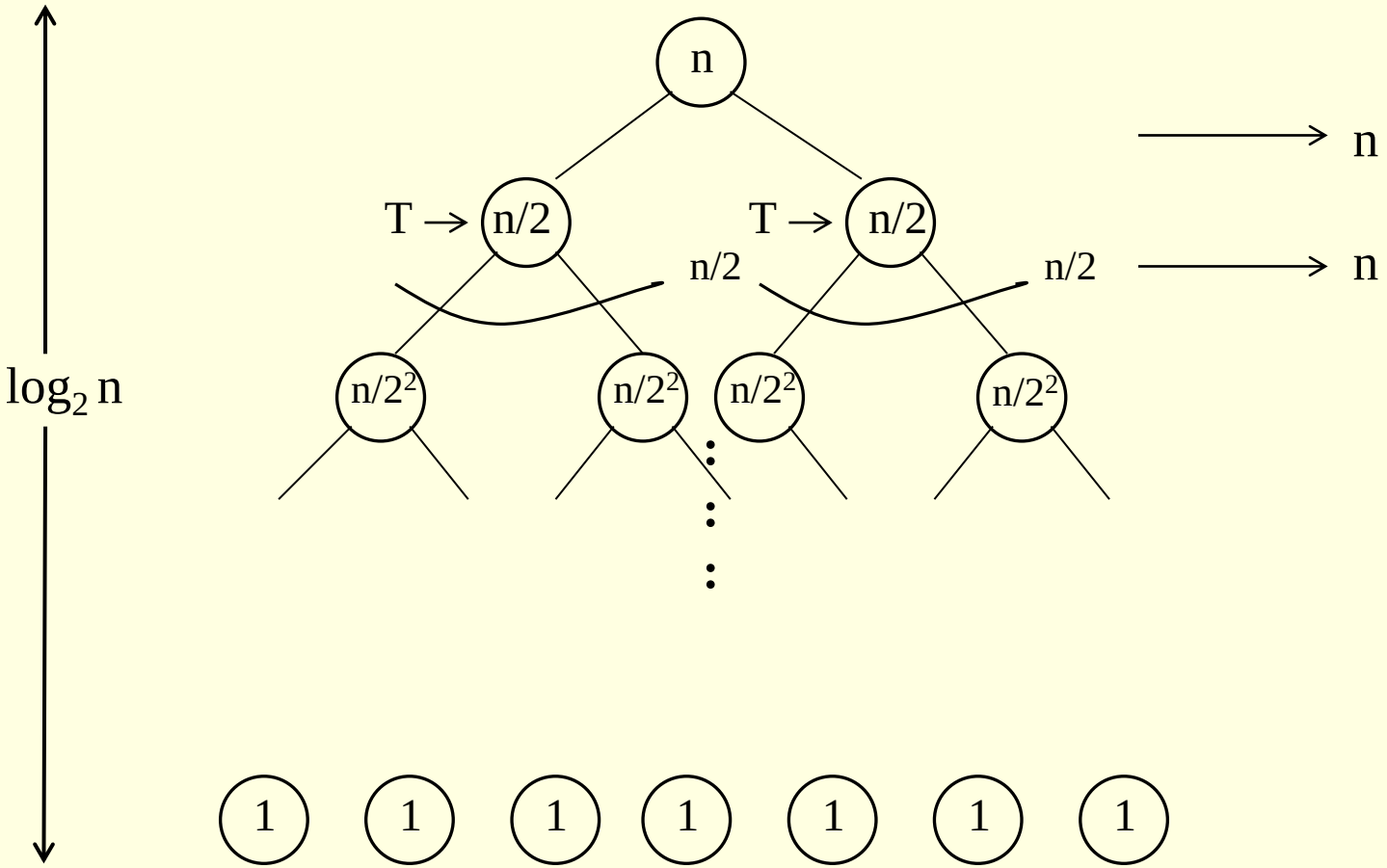
  For e.g., $T(n) = a\,T(n/b) + f(n)$ where $a \geq 1$,$b > 1$ and $f(n)$ is a given function .

  F(n) is the cost of splitting or combining the sub problems.



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

The recursion tree for this recurrence is :



When we add the values across the levels of the recursion tree, we get a value of n for every level.

We have $n + n + n + \ldots\ldots$ log n times

$= n\ (1 + 1 + 1 + \ldots\ldots$ log n times$)$

$= n\ (\log_2 n)$

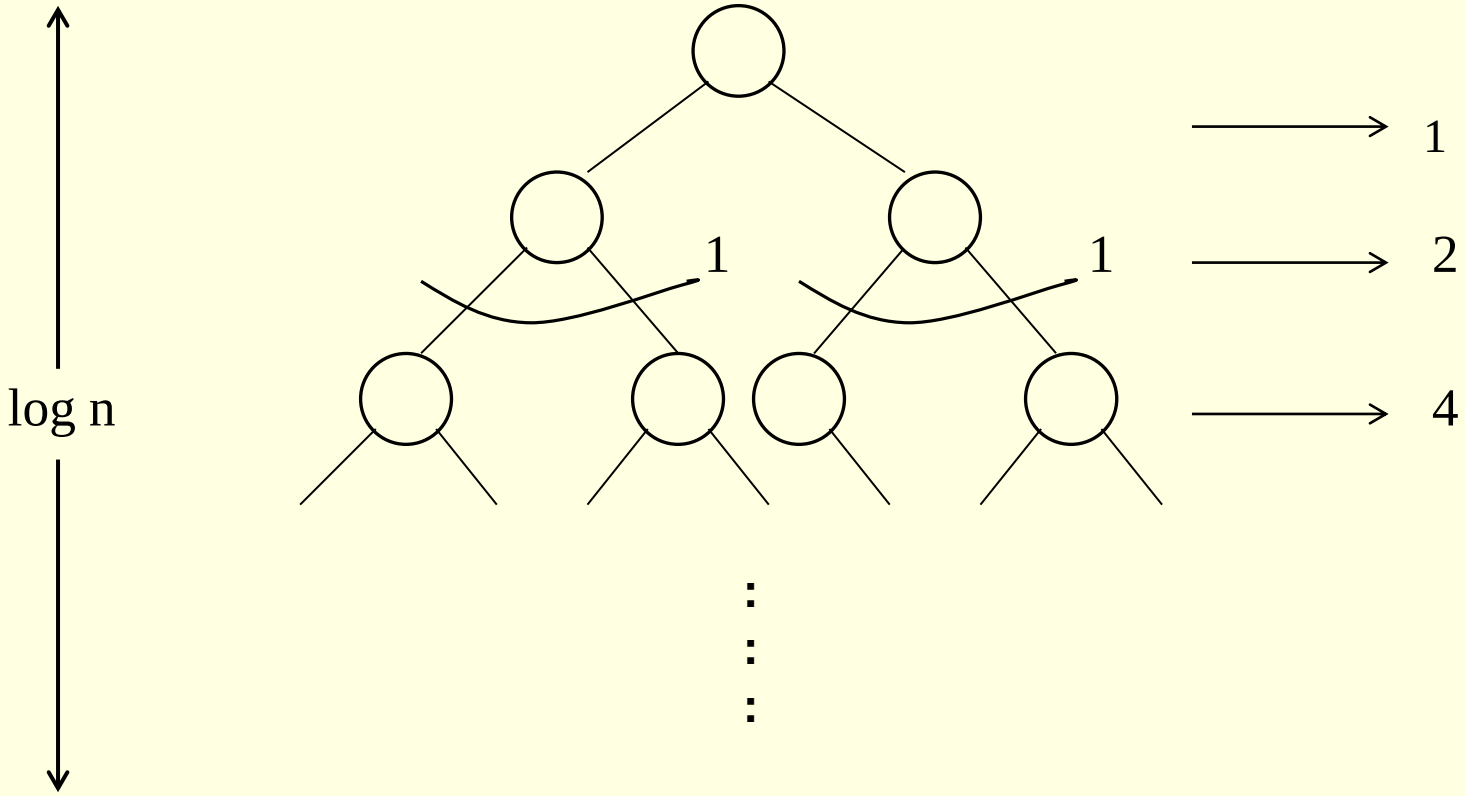
$= \Theta\ (n \log n)$

$T(n) = \Theta\ (n \log n)$

II.

Given : **T(n) = 2T(n/2) + 1**

Solution : The recursion tree for the above recurrence is



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

We get series like

$1 + 2 + 2^2 + 2^3 + \ldots\ldots$ log n times which is a G.P.

[So, using the formula for sum of terms in a G.P. :

$$a + ar + ar^2 + ar^3 + \ldots\ldots + ar^{n-1} = \frac{a(r^n - 1)}{r - 1} \quad]$$

$$= \frac{1\,(2^{\log n} - 1)}{2 - 1}$$

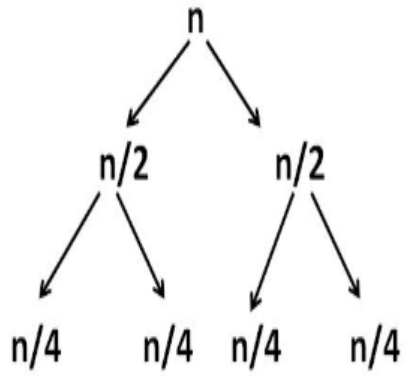
$$= n - 1$$

$$= \Theta\,(n - 1) \qquad \text{(neglecting the lower order terms)}$$

$$= \Theta\,(n)$$

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

Here the number of leaves $= 2^{\log n} = n$ and the sum of effort in each level except leaves is



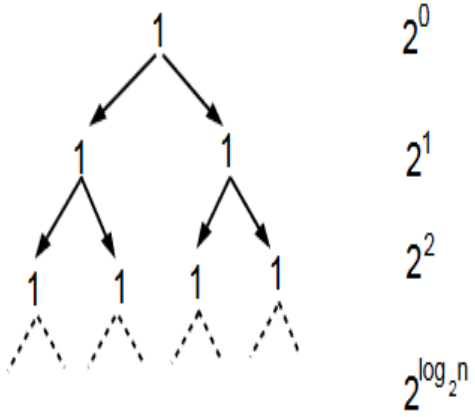


Fig. 1. *The Input size reduction tree* *The Corresponding Computation tree*

2. $T(n) = 2T(n/2) + 1$
$T(1) = \log n$

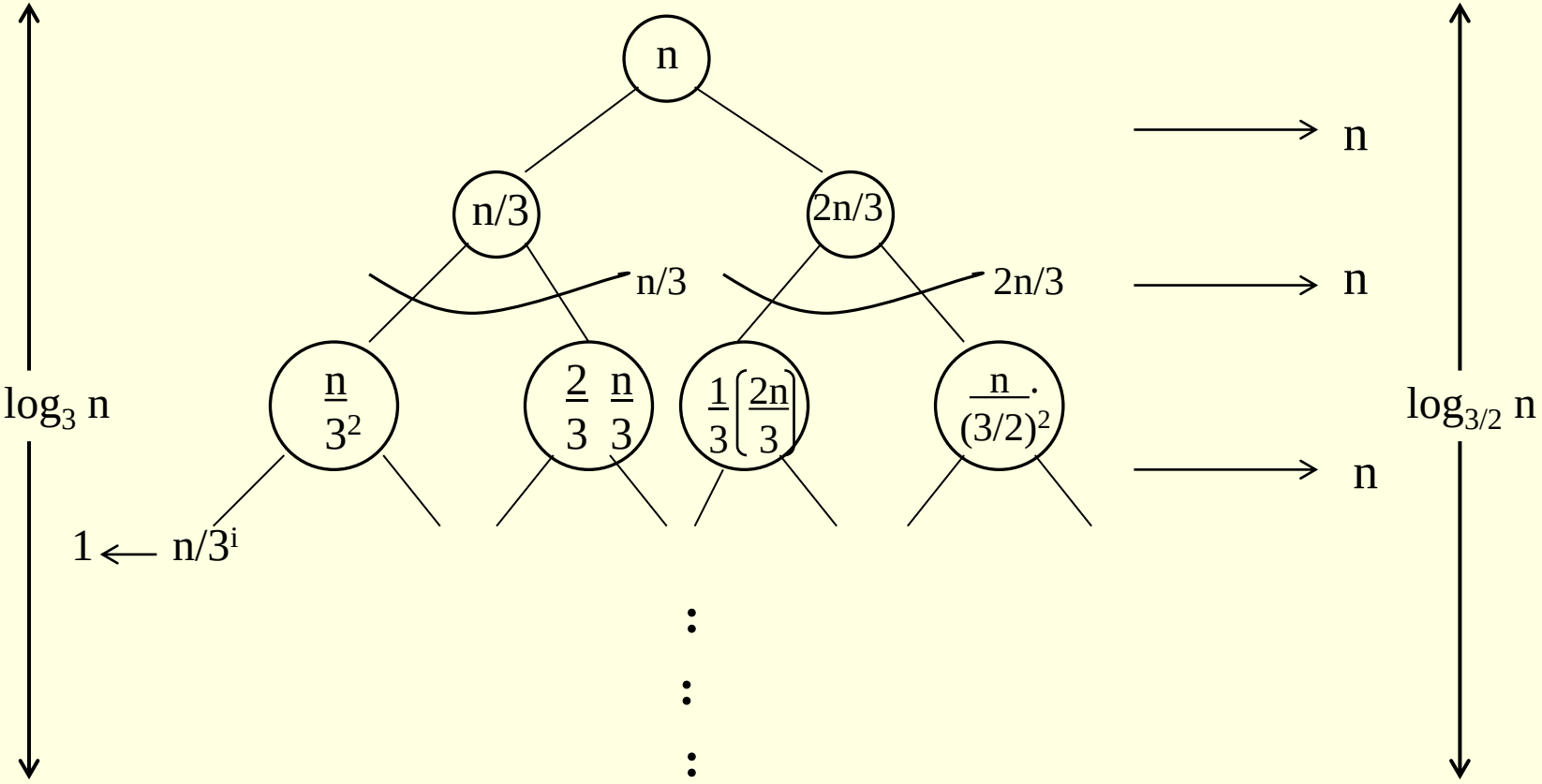
Solution: Note that $T(1)$ is $\log n$.
Given $T(n) = 2T(n/2) + 1$
$=2[2T(n/4) + 1] + 1 = 2^2T(n/2^2) + 2 + 1$
$= 2^2[2T(n/2^3) + 1] + 2 + 1 = 2^3T(n/2^3) + 2^2 + 2 + 1 = 2^kT(n/2^k) + (2^{k-1} + 2^{k-2} + ... + 2 + 1)$
We stop the recursion when $n/2^k = 1$.
Therefore, $T(n) = 2^kT(1) + 2^k - 1 = 2^k \log n + 2^k - 1$
$= 2^{\log_2 n} \log n + 2^{\log_2 n} - 1$

$= n \log n + n - 1$
Clearly, $n \log n$ is the significant term. $T(n) = \theta(n \log n)$.

III.

Given : **T(n) = T(n/3) + T(2n/3) + n**

Solution : The recursion tree for the above recurrence is



When we add the values across the levels of the recursion tree , we get a value of n for every level.

Since the shortest path from the root to the leaf is

$$n \rightarrow \frac{n}{3} \rightarrow \frac{n}{3^2} \rightarrow \frac{n}{3^3} \rightarrow \ldots 1$$

we have 1 when $\frac{n}{3^i} = 1$

$$\Rightarrow \quad n = 3^i$$

Taking $\log_3$ on both the sides

$$\Rightarrow \quad \log_3 n = i$$

Thus the height of the shorter tree is $\log_3 n$

$$T(n) \geq n \log_3 n \quad \ldots (A)$$

Similarly, the longest path from root to the leaf is

$$n \rightarrow \left(\frac{2}{3}\right) n \rightarrow \left(\frac{2}{3}\right)^2 n \rightarrow \ldots 1$$

So rightmost will be the longest

when $\left(\frac{2}{3}\right)^k n = 1$

or $\frac{n}{(3/2)^k} = 1$

$\Rightarrow \quad k = \log_{3/2} n$

$T(n) \leq n \log_{3/2} n \quad \ldots$ (B)

Since base does not matter in asymptotic notation , we guess from (A) and (B)

$$T(n) = \Theta (n \log_2 n)$$

3. $T(n) = 3T(n/4) + cn^2$

Note that the number of levels $= \log_4 n + 1$

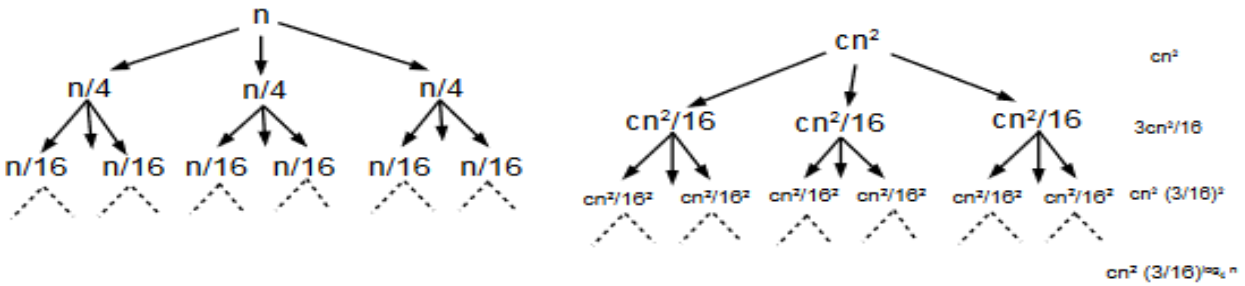


Fig. 2. *Input size reduction tree* *The Corresponding Computation tree*

Also the number of leaves $= 3^{\log_4 n} = n^{\log_4 3}$

The total cost taken is the sum of the cost spent at all leaves and the cost spent at each subdivision operation. Therefore, the total time taken is $T(n) = cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2cn^2 + \cdots + (\frac{3}{16})^{\log_4(n)-1}$ + number of leaves $\times\ T(1)$.

$$T(n) = \sum_{i=0}^{\log_4(n)-1} cn^2(\tfrac{3}{16})^i + n^{\log_4(3)} \times T(1)$$

$$= \frac{\frac{3}{16}^{\log_4(n)} - 1}{\frac{3}{16} - 1} cn^2 + n^{\log_4(3)} \times T(1)$$

$$= \frac{1 - \frac{3}{16}^{\log_4(n)}}{1 - \frac{3}{16}} cn^2 + n^{\log_4(3)} \times T(1)$$

$$= \frac{1 - n^{\log_4(\frac{3}{16})}}{1 - \frac{3}{16}} cn^2 + n^{\log_4(3)} \times T(1)$$

$= d'cn^2 + n^{\log_4(3)}T(1)$ where the constant $d' = \dfrac{1 - n^{\log_4(\frac{3}{16})}}{1 - \frac{3}{16}}$

$= d'cn^2 + n^{\log_4(3)}T(1)$. Therefore, $T(n) = O(n^2)$

Since the root of the computation tree contains cn^2, $T(n) = \Omega(n^2)$. Therefore, $T(n) = \theta(n^2)$

Steps to solve Recurrence relations using Recursion tree method-

Step-01:

Draw a recursion tree based on the given recurrence relation.

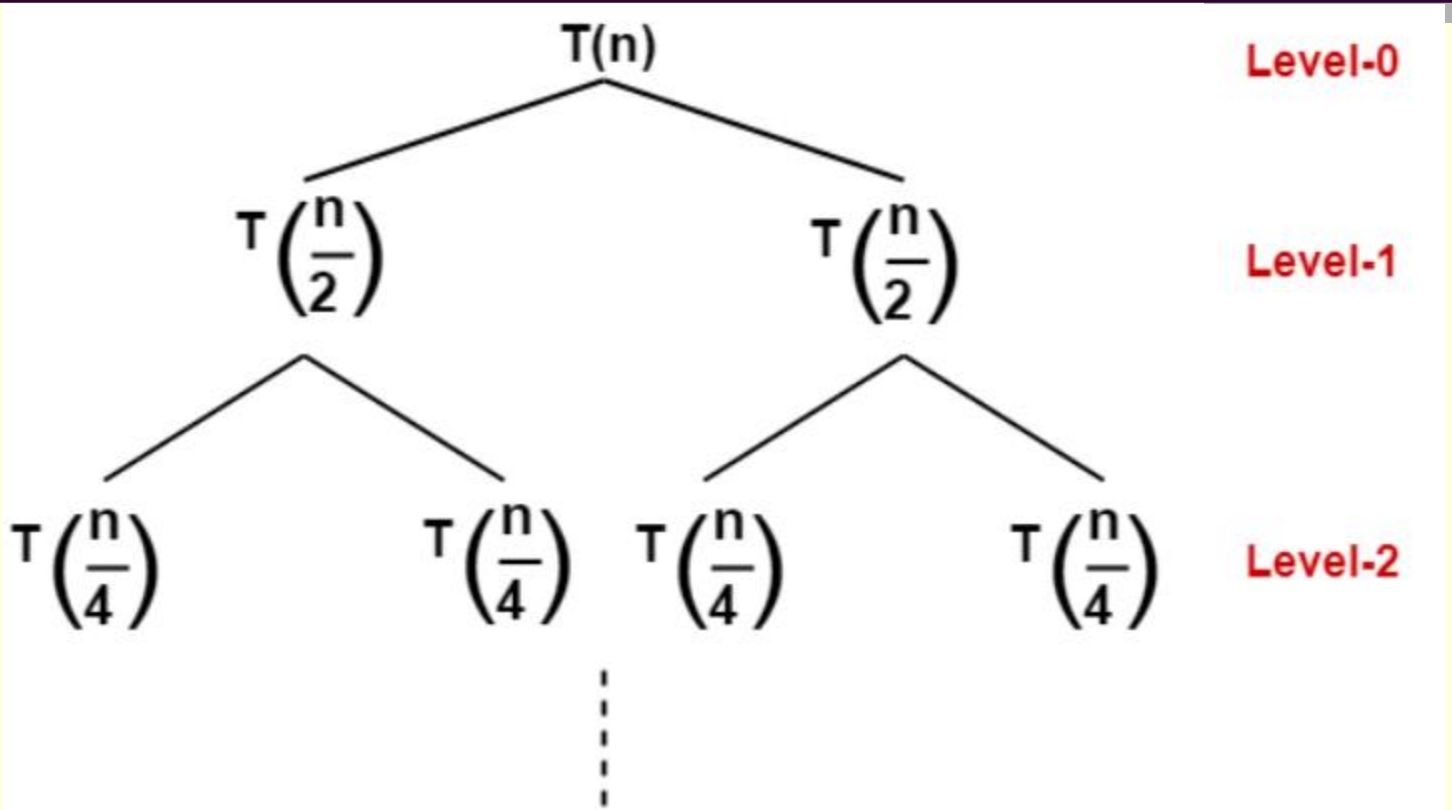
Step-02:

Determine-
Cost of each level
Total number of levels in the recursion tree
Number of nodes in the last level
Cost of the last level

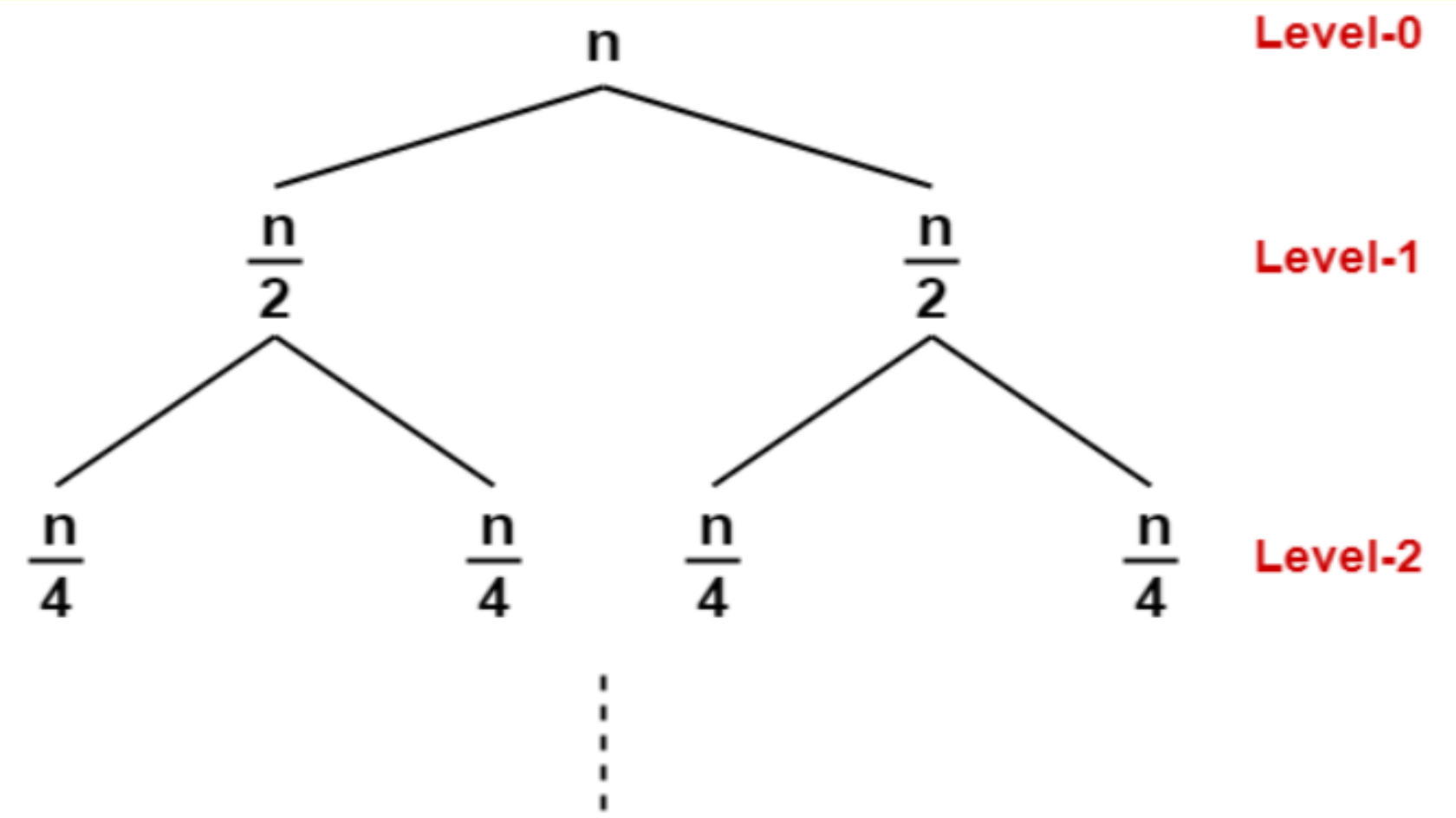
Step-03:

Add cost of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation.

- Solve the following recurrence relation using recursion tree method-
- $T(n) = 2T(n/2) + n$
- **Step-01:**
- Draw a recursion tree based on the given recurrence relation.


T(n)
Level-0
$T\left(\frac{n}{2}\right)$
$T\left(\frac{n}{2}\right)$
Level-1
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$
Level-2

- The given recurrence relation shows-
- The cost of dividing a problem of size n into its 2 sub-problems and then combining its solution is n.
- The cost of dividing a problem of size n/2 into its 2 sub-problems and then combining its solution is n/2 and so on.


n
Level-0
n/2
n/2
Level-1
n/4
n/4
n/4
n/4
Level-2

- **Step-02:**
  - Determine cost of each level-
    - Cost of level-0 = n
    - Cost of level-1 = n/2 + n/2 = n
    - Cost of level-2 = n/4 + n/4 + n/4 + n/4 = n and so on.
- **Step-03:**
  - Determine total number of levels in the recursion tree-
    - Size of sub-problem at level-0 = $n/2^0$
    - Size of sub-problem at level-1 = $n/2^1$
    - Size of sub-problem at level-2 = $n/2^2$
    - Continuing in similar manner, we have-
    - Size of sub-problem at level-i = $n/2^i$

- Suppose at level-x (last level), size of sub-problem becomes 1. Then-
- $n / 2^x = 1$
- $2^x = n$
- Taking log on both sides, we get-
- $x\log 2 = \log n$
- $x = \log_2 n$

$\therefore$ Total number of levels in the recursion tree $= \log_2 n + 1$

- **Step-04:**
-
- Determine number of nodes in the last level-
- Level-0 has 2^0 nodes i.e. 1 node
- Level-1 has 2^1 nodes i.e. 2 nodes
- Level-2 has 2^2 nodes i.e. 4 nodes
-
- Continuing in similar manner, we have-
- Level-$\log_2 n$ has $2^{\log_2 n}$ nodes i.e. n nodes

- **Step-05:**
-
- Determine cost of last level-
- Cost of last level = n x T(1) = θ(n)

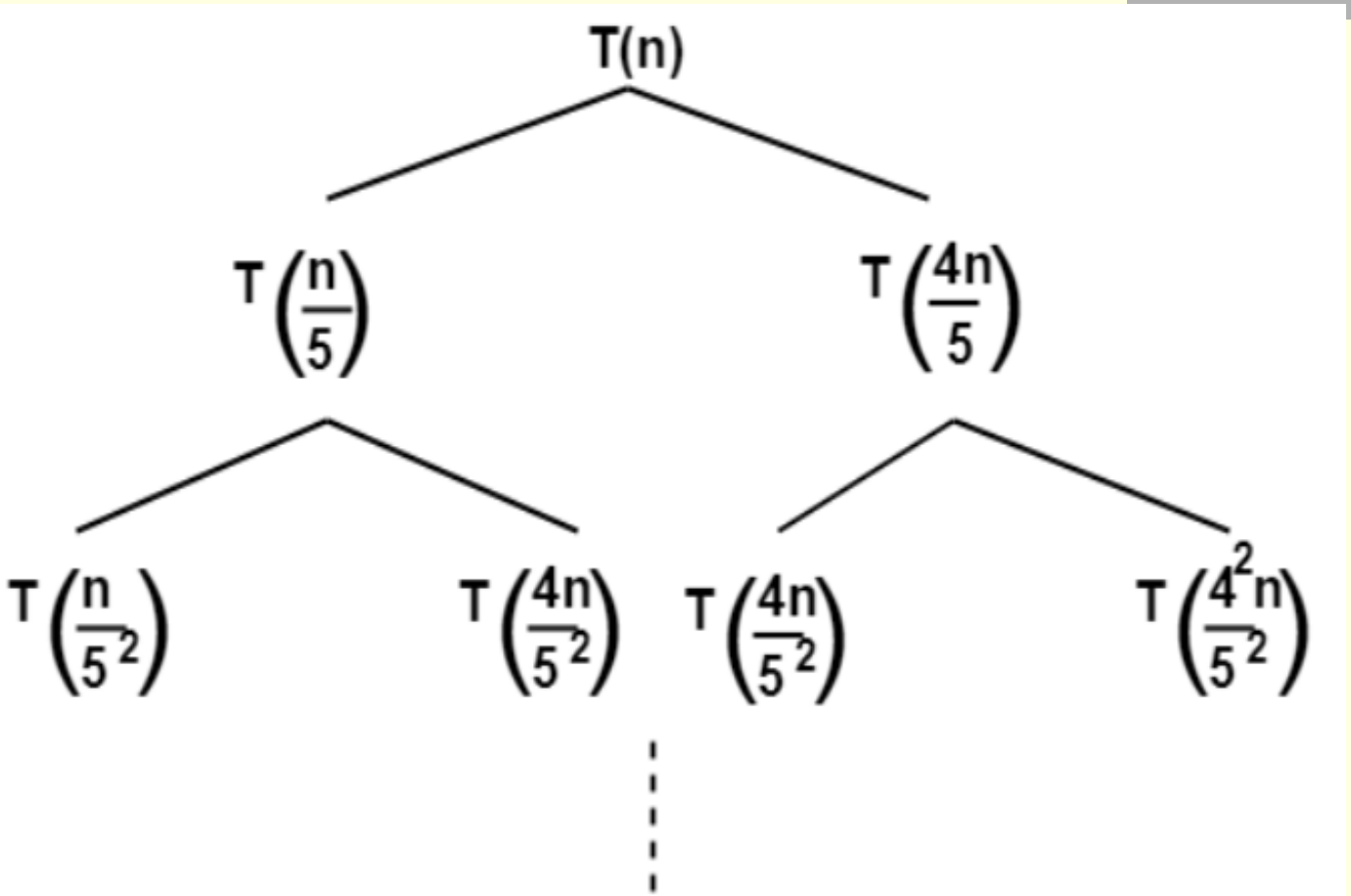
- **Step-06:**
- Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation-

$$T(n) = \underbrace{\{ n + n + n + ... \}}_{\text{For } \log_2 n \text{ levels}} + \theta (n)$$

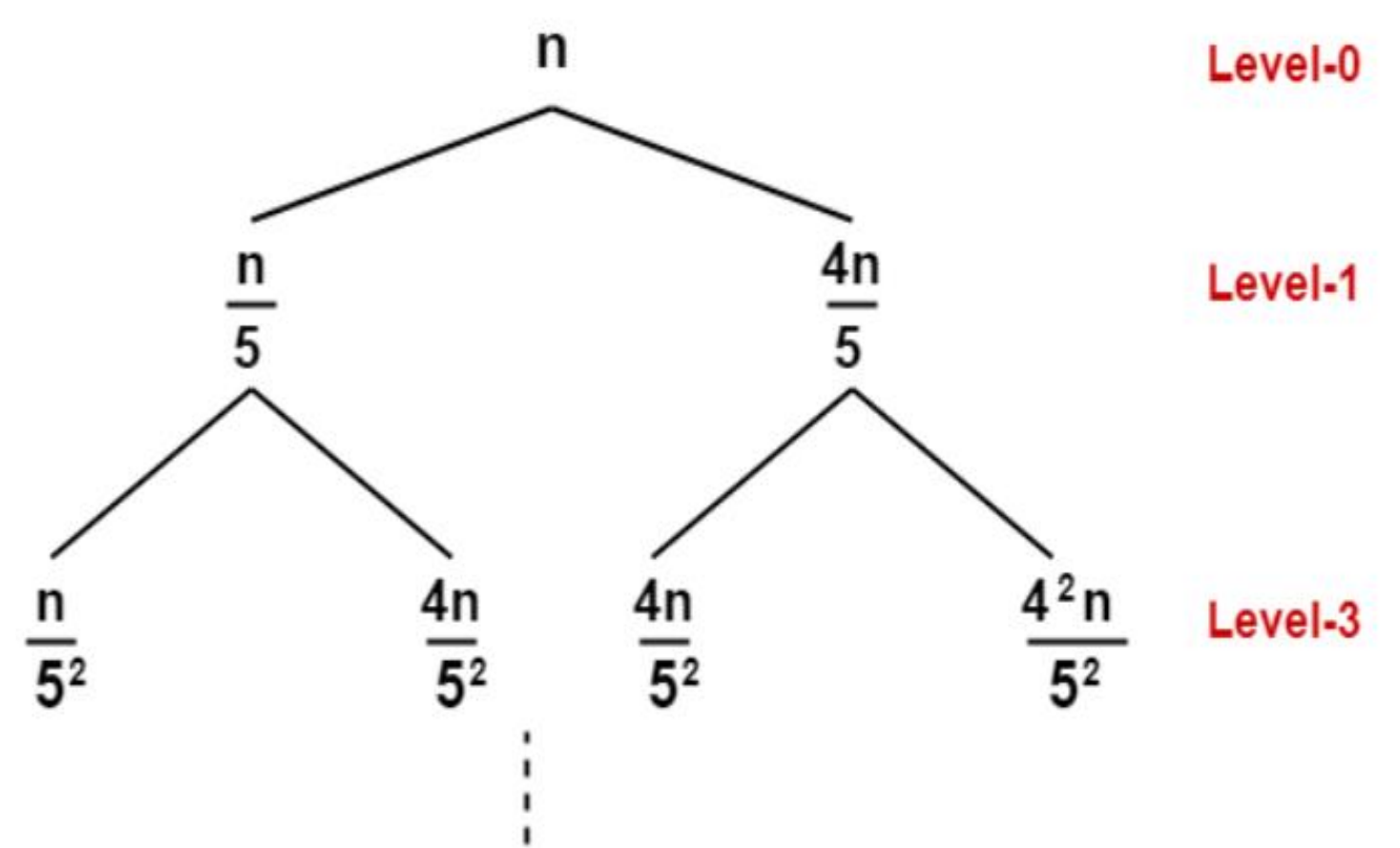
- $= n \times \log_2 n + \theta (n)$
- $= n\log_2 n + \theta (n)$
- $= \boldsymbol{\theta (n\log_2 n)}$

- **Problem-02:**
- Solve the following recurrence relation using recursion tree method-
- $T(n) = T(n/5) + T(4n/5) + n$

- **Step-01:**
- Draw a recursion tree based on the given recurrence relation.
- The given recurrence relation shows-
- A problem of size n will get divided into 2 sub-problems- one of size n/5 and another of size 4n/5.
- Then, sub-problem of size n/5 will get divided into 2 sub-problems- one of size $n/5^2$ and another of size $4n/5^2$.
- On the other side, sub-problem of size 4n/5 will get divided into 2 sub-problems- one of size $4n/5^2$ and another of size $4^2n/5^2$ and so on.
- At the bottom most layer, the size of sub-problems will reduce to 1.


T(n)
T(n/5)
T(4n/5)
T(n/5^2)
T(4n/5^2)
T(4n/5^2)
T(4^2 n/5^2)

- The given recurrence relation shows-
- The cost of dividing a problem of size n into its 2 sub-problems and then combining its solution is n.
- The cost of dividing a problem of size n/5 into its 2 sub-problems and then combining its solution is n/5.
- The cost of dividing a problem of size 4n/5 into its 2 sub-problems and then combining its solution is 4n/5 and so on.


n
Level-0
n/5
4n/5
Level-1
n/5^2
4n/5^2
4n/5^2
4^2n/5^2
Level-3

- **Step-02:**
- Determine cost of each level-
- Cost of level-0 = n
- Cost of level-1 = $n/5 + 4n/5 = n$
- Cost of level-2 = $n/5^2 + 4n/5^2 + 4n/5^2 + 4^2n/5^2 = n$

- **Step-03:**
- Determine total number of levels in the recursion tree. We will consider the rightmost sub tree as it goes down to the deepest level-
- Size of sub-problem at level-0 = $(4/5)^0 n$
- Size of sub-problem at level-1 =$(4/5)^1 n$
- Size of sub-problem at level-2 =$(4/5)^2 n$
-
- Continuing in similar manner, we have-
- Size of sub-problem at level-i = $(4/5)^i n$
- Suppose at level-x (last level), size of sub-problem becomes 1. Then-
- $(4/5)^x n = 1$
- $(4/5)^x = 1/n$
- Taking log on both sides, we get-
- $x \log(4/5) = \log(1/n)$
- $x = \log_{5/4} n$
-
- ∴ Total number of levels in the recursion tree = $\log_{5/4} n + 1$

- **Step-04:**
-
- Determine number of nodes in the last level-
- Level-0 has 2^0 nodes i.e. 1 node
- Level-1 has 2^1 nodes i.e. 2 nodes
- Level-2 has 2^2 nodes i.e. 4 nodes
-
- Continuing in similar manner, we have-
- Level-$\log_{5/4}n$ has $2^{\log_{5/4}n}$ nodes

- **Step-05:**
-
- Determine cost of last level-
- Cost of last level = $2^{\log_{5/4} n} \times T(1) = \theta(2^{\log_{5/4} n}) = \theta(n^{\log_{5/4} 2})$
-

- **Step-06:**
- Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation-

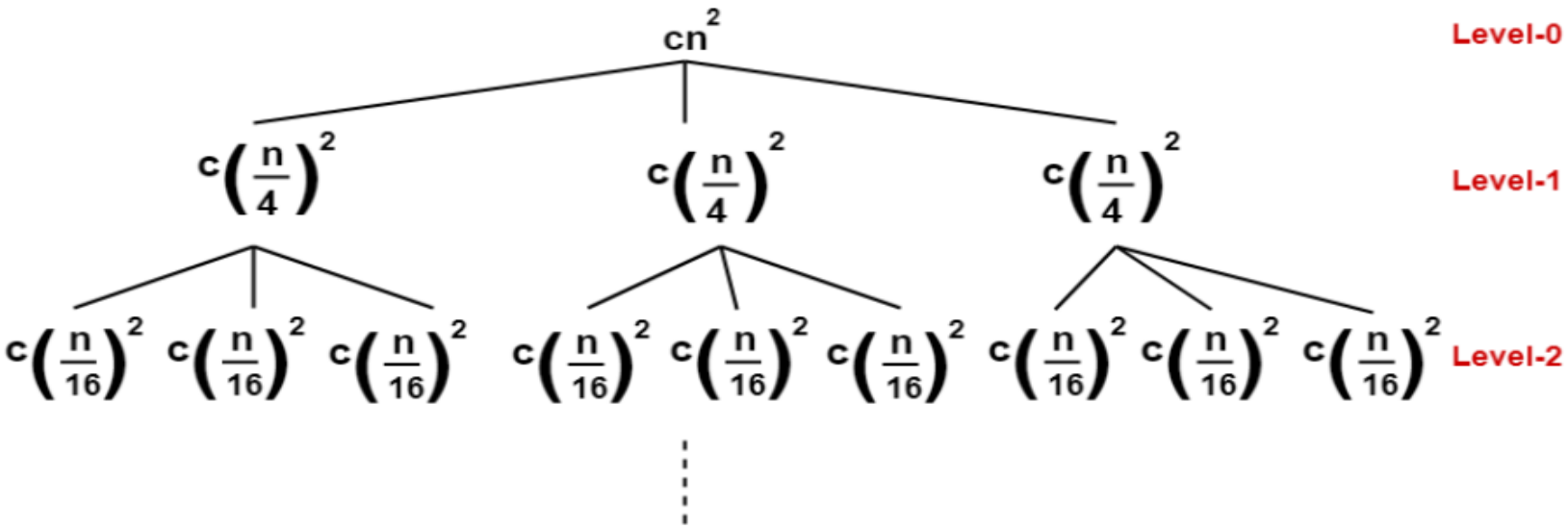
$$T(n) = \underbrace{\{ n + n + n + ... \}}_{\text{For } \log_{5/4} n \text{ levels}} + \theta(n^{\log_{5/4} 2})$$

- $= n\log_{5/4} n + \theta(n^{\log_{5/4} 2})$
- $= \theta(n\log_{5/4} n)$

- **Problem-03:**
- Solve the following recurrence relation using recursion tree method-
- $T(n) = 3T(n/4) + cn^2$

Step-01:

Draw a recursion tree based on the given recurrence relation-



(Here, we have directly drawn a recursion tree representing the cost of sub problems)

- **Step-02:**
-
- Determine cost of each level-
- Cost of level-0 = cn^2
- Cost of level-1 = $c(n/4)^2 + c(n/4)^2 + c(n/4)^2 = (3/16)cn^2$
- Cost of level-2 = $c(n/16)^2 \times 9 = (9/16^2)cn^2$

- **Step-03:**
- Determine total number of levels in the recursion tree-
- Size of sub-problem at level-0 = $n/4^0$
- Size of sub-problem at level-1 = $n/4^1$
- Size of sub-problem at level-2 = $n/4^2$
-
- Continuing in similar manner, we have-
- Size of sub-problem at level-i = $n/4^i$
- Suppose at level-x (last level), size of sub-problem becomes 1. Then-
- $n/4^x = 1$
- $4^x = n$
- Taking log on both sides, we get-
- $x\log 4 = \log n$
- $x = \log_4 n$
-
- $\therefore$ Total number of levels in the recursion tree = $\log_4 n + 1$

- **Step-04:**
-
- Determine number of nodes in the last level-
- Level-0 has 3^0 nodes i.e. 1 node
- Level-1 has 3^1 nodes i.e. 3 nodes
- Level-2 has 3^2 nodes i.e. 9 nodes
-
- Continuing in similar manner, we have-
- Level-$\log_4 n$ has $3^{\log_4 n}$ nodes i.e. $n^{\log_4 3}$ nodes

- **Step-05:**
-
- Determine cost of last level-
- Cost of last level = $n^{\log_4 3} \times T(1) = \theta(n^{\log_4 3})$
-

- **Step-06:**
-
- Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation-

$$T(n) = \underbrace{\left\{ cn^2 + \frac{3}{16} cn^2 + \frac{9}{(16)^2} cn^2 + \ldots\ldots\ldots \right\}}_{\text{For } \log_4 n \text{ levels}} + \theta\left(n^{\log_4 3}\right)$$

$= cn^2 \{ 1 + (3/16) + (3/16)^2 + \ldots\ldots\ldots \} + \theta(n^{\log_4 3})$

Now, $\{ 1 + (3/16) + (3/16)^2 + \ldots\ldots\ldots \}$ forms an infinite Geometric progression.

On solving, we get-

$= (16/13)cn^2 \{ 1 - (3/16)^{\log_4 n} \} + \theta(n^{\log_4 3})$

$= (16/13)cn^2 - (16/13)cn^2 (3/16)^{\log_4 n} + \theta(n^{\log_4 3})$

$= \mathbf{O(n^2)}$

Master Theorem

- In the analysis of algorithms, the master theorem for divide-and-conquer recurrences provides an asymptotic analysis (using Big O notation) for recurrence relations of types that occur in the analysis of many divide and conquer algorithms.

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Master Theorem

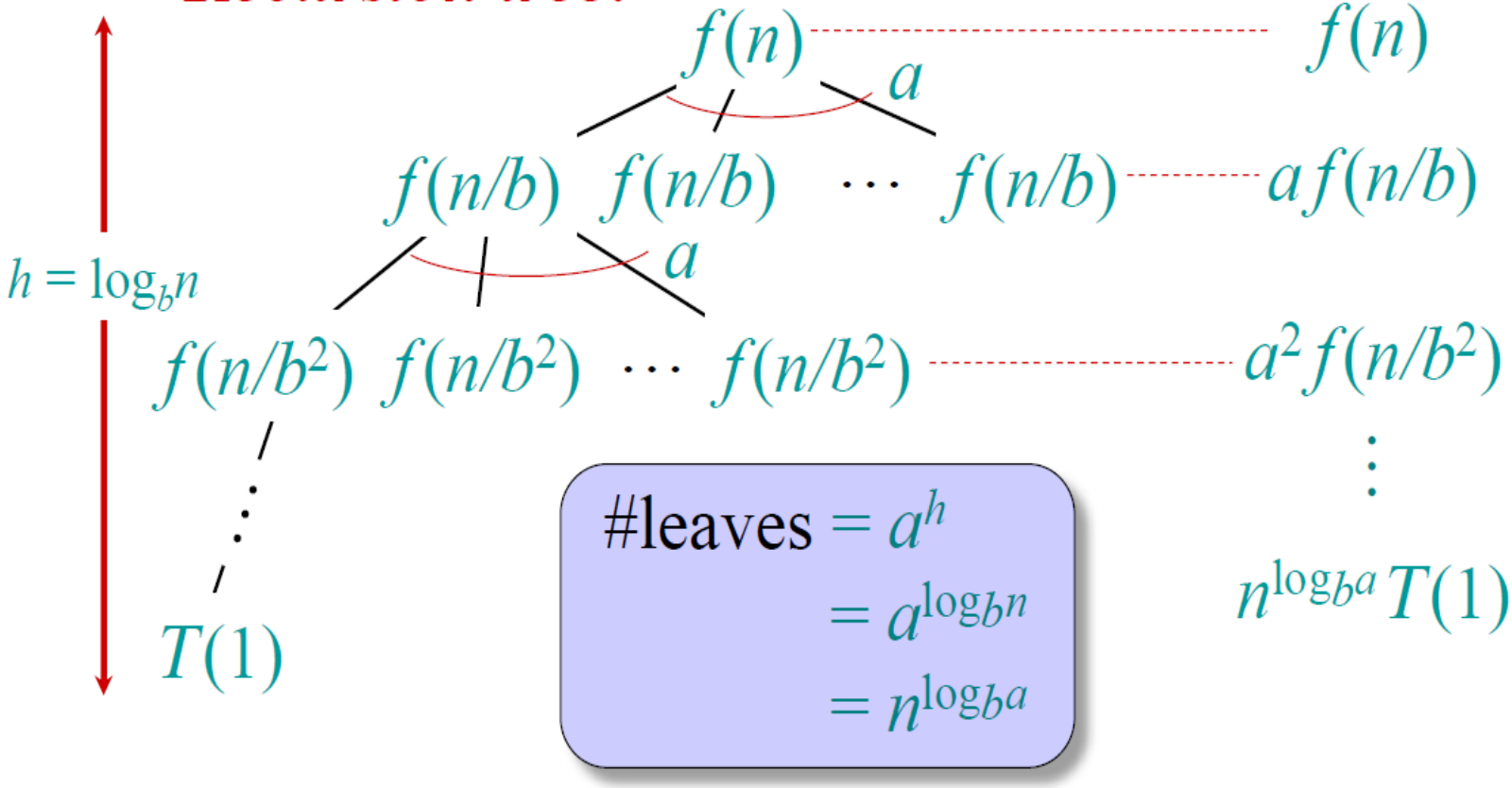
The master method applies to recurrences of the form

$$T(n) = a\,T(n/b) + f(n)\ ,$$

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

Idea of master theorem

Recursion tree:



Three common cases

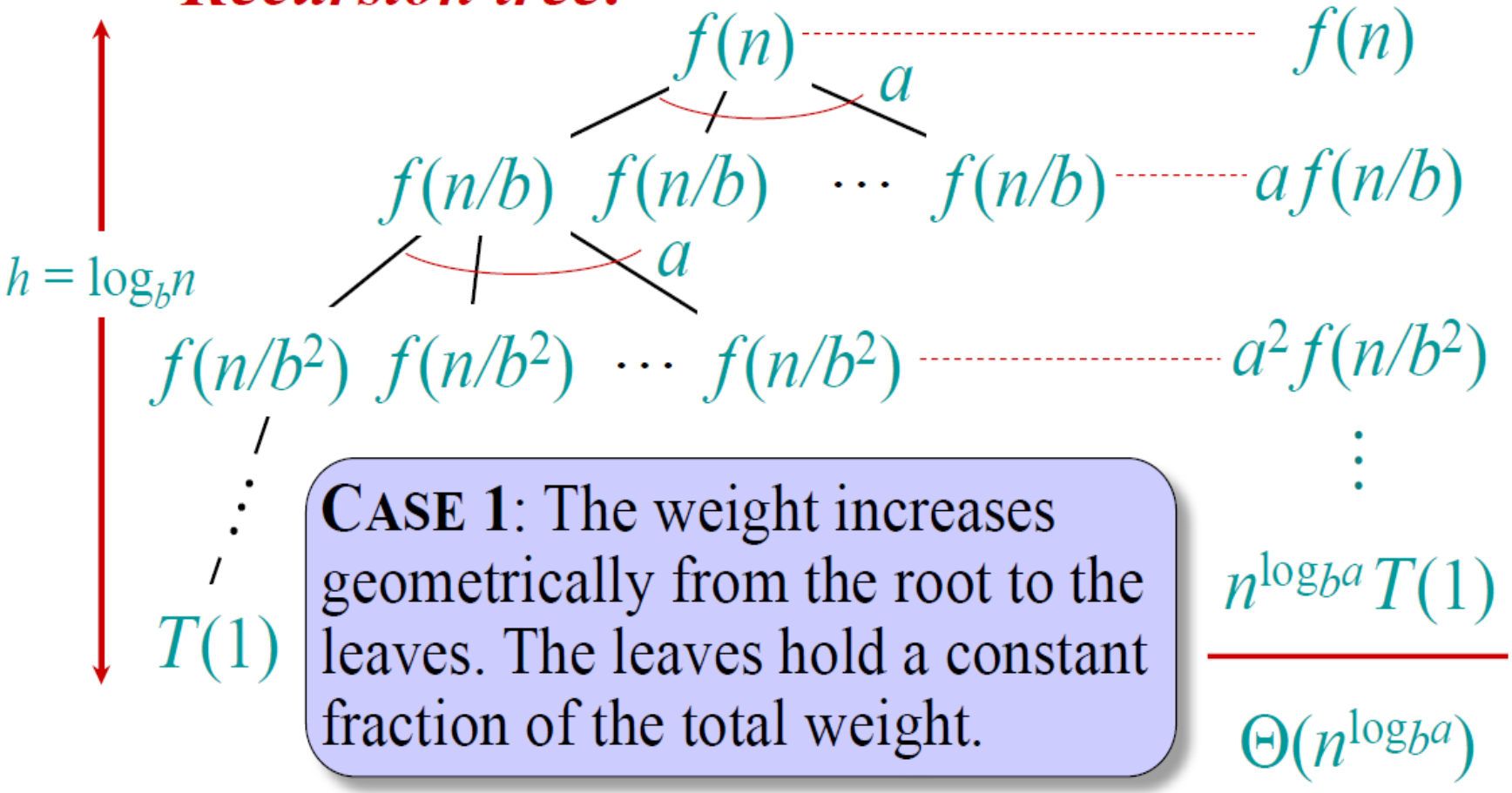
Compare $f(n)$ with $n^{\log_b a}$:

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor).

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

Idea of master theorem

Recursion tree:



CASE 1: The weight increases geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight.

Three common cases

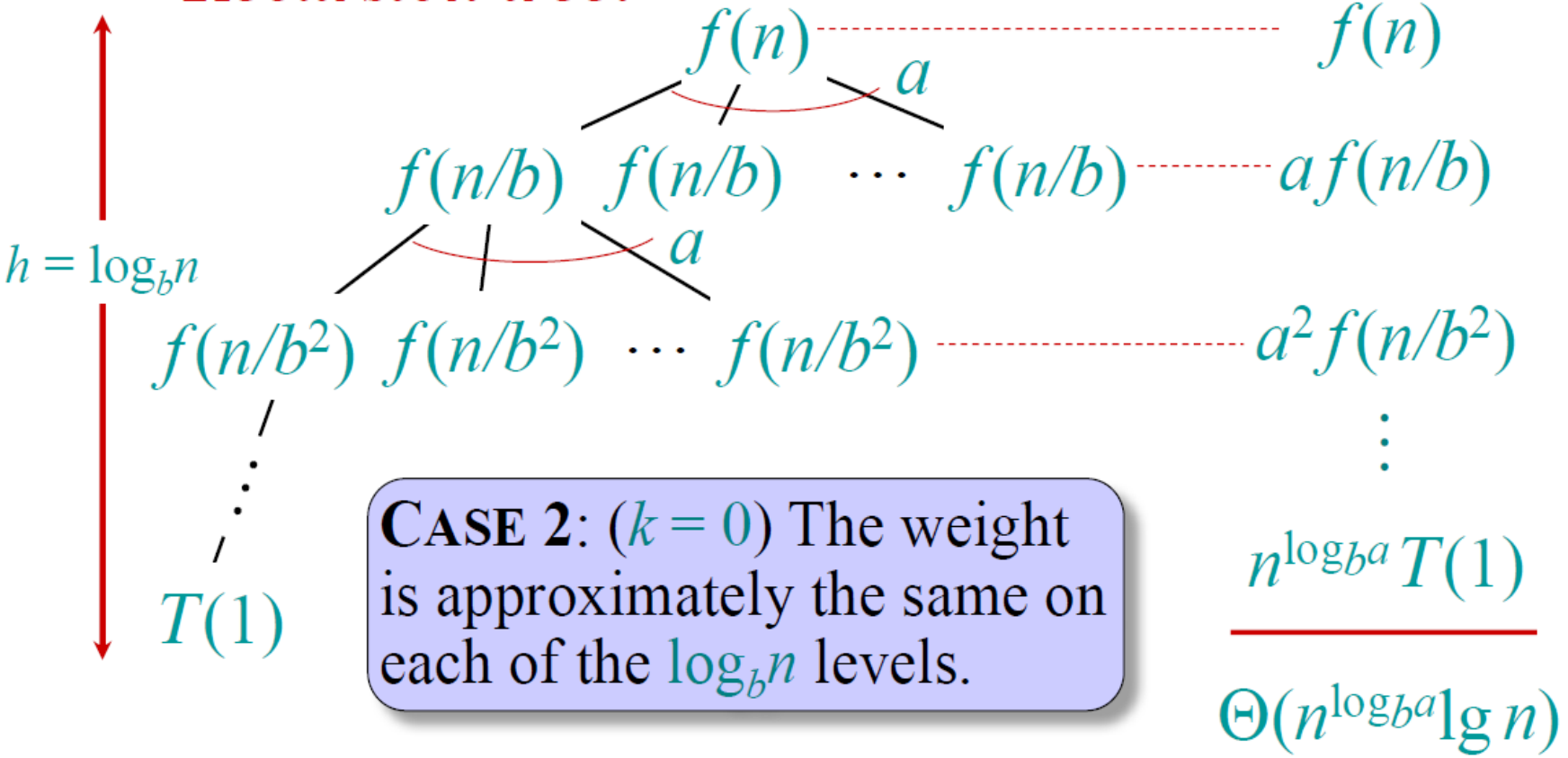
Compare $f(n)$ with $n^{\log_b a}$:

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.

   Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

Idea of master theorem

Recursion tree:



CASE 2: ($k = 0$) The weight is approximately the same on each of the $\log_b n$ levels.

Three common cases (cont.)

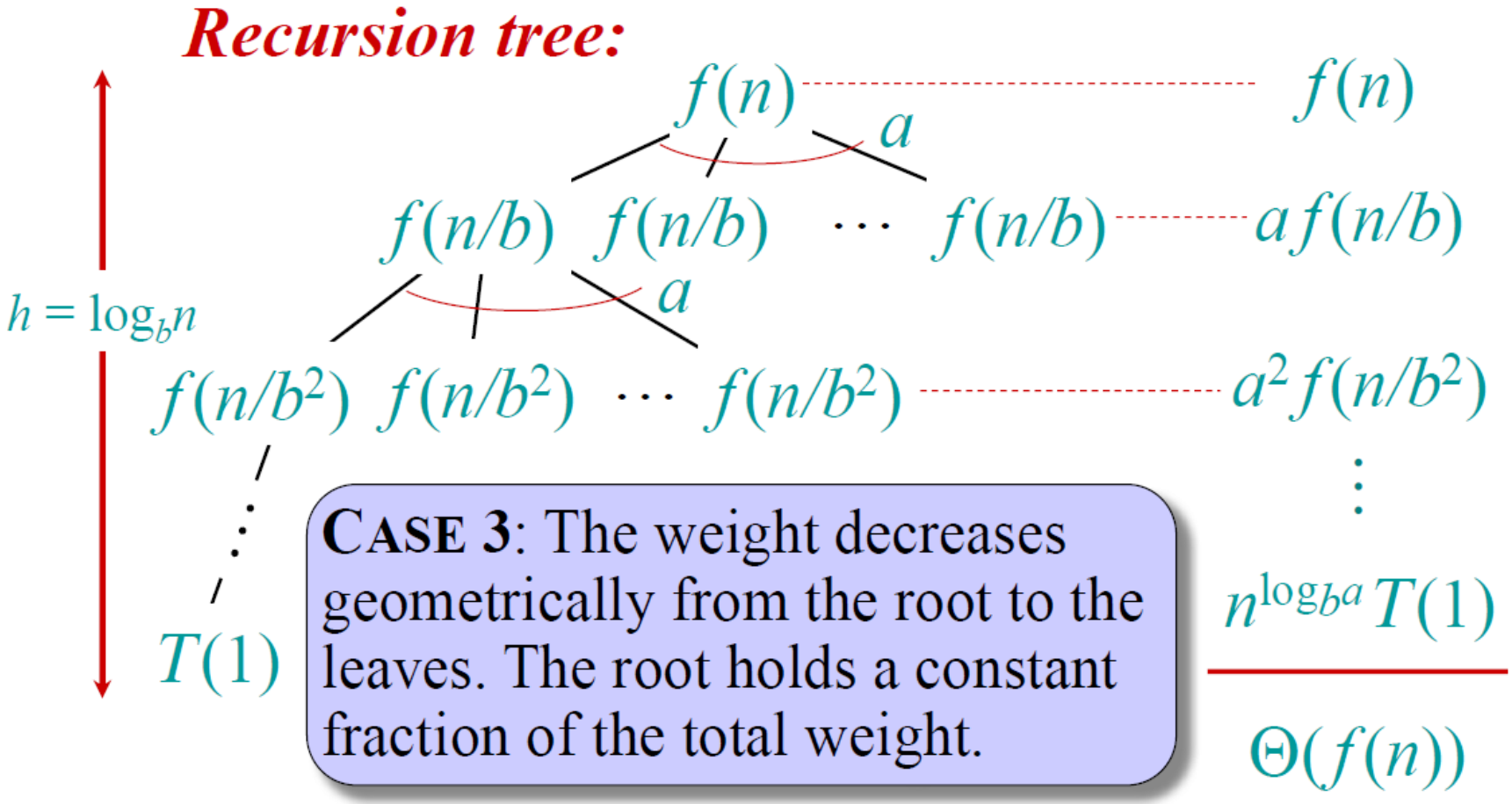
Compare $f(n)$ with $n^{\log_b a}$:

3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor),

   and $f(n)$ satisfies the ***regularity condition*** that $a\,f(n/b) \le c\,f(n)$ for some constant $c < 1$.

   Solution: $T(n) = \Theta(f(n))$.

Idea of master theorem



Applications

$k = \log_2 1 = 0; f(n) = 2^n$
$2^n = \Omega(n^{0+\log 2})$
$1 \cdot 2^{\frac{n}{2}} \leq \frac{1}{2} \cdot 2^n$

$k = \log_2 3; f(n) = n^2$
$n^2 = \Omega(n^{\log_2 3+(2-\log_2 3)})$
$3 \cdot \left(\frac{n}{2}\right)^2 \leq \frac{3}{4} \cdot n^2$

$T(n) = 3 * T(n/2) + n^2$
$\Rightarrow T(n) = \Theta(n^2)$ (case 3)

$T(n) = T(n/2) + 2^n$
$\Rightarrow T(n) = \Theta(2^n)$ (case 3)

* For sure question on this in final *

$T(n) = 16 * T(n/4) + n$
$\Rightarrow T(n) = \Theta(n^2)$ (case 1)

$T(n) = 2 * T(n/2) + n \log n$
$\Rightarrow T(n) = n \log^2 n$ (case 2)

$k = \log_4 16 = 2; f(n) = n$
$n = O(n^{2-1})$

$T(n) = 2^n * T(n/2) + n^n$
$\Rightarrow$ Does not apply!!
↳ ex. when master theorem cannot be applied!

$k = \log_2 2 = 1; f(n) = n \log n$
$n \log n = \Theta(n^1 log^1 n)$

Examples

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

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

CASE 1: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$.

$\therefore\ T(n) = \Theta(n^2)$.

Ex. $T(n) = 4T(n/2) + n^2$

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

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

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

Another way for Master's Theorem
General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Master Theorem: If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ.

Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$ Θ(n^2)

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$ Θ(n^2log n)

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$ Θ(n^3)