



# **UNIT I**

# **INTRODUCTION TO ALGORITHM DESIGN**

# Master's theorem method

- Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

$T(n) = aT(n/b) + f(n)$  where  $a \geq 1$  and  $b > 1$  and  $f(n)$  is an asymptotically positive function.

- There are following three cases:
  1. If  $f(n) < O(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a})$ .
  2. If  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$ .
  3. If  $f(n) > \Omega(n^{\log_b a})$ , and  $f(n)$  satisfies the regularity condition, then  $T(n) = \Theta(f(n))$ .

# Solve the problems

- $T(n) = 3T(n/2) + n^2$
- $T(n) = 7T(n/2) + n^2$
- $T(n) = 4T(n/2) + n^2$
- $T(n) = 3T(n/4) + n \lg n$

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

**$T(n) = aT(n/b) + f(n)$  where  $a \geq 1$  and  $b > 1$**

$$a = 3 \quad b = 2 \quad f(n) = n^2$$

1. If  $f(n) < O(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a})$ .
2. If  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$ .
3. If  $f(n) > \Omega(n^{\log_b a})$ , and  $f(n)$  satisfies the regularity condition, then  $T(n) = \Theta(f(n))$ .

Step 1: Calculate  $n^{\log_b a} = n^{\log_2 3} = n^{1.58}$

Step 2: Compare with  $f(n)$

Since  $f(n) > n^{\log_b a}$   
i.e.  $n^2 > n^{1.58}$

Step 3: Case 3 is satisfied hence complexity is given as  
 $T(n) = \Theta(f(n)) = \Theta(n^2)$

$$T(n) = 7T(n/2) + n^2$$

**$T(n) = aT(n/b) + f(n)$  where  $a \geq 1$  and  $b > 1$**

$$a = 7 \quad b = 2 \quad f(n) = n^2$$

1. If  $f(n) < O(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a})$ .
2. If  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$ .
3. If  $f(n) > \Omega(n^{\log_b a})$ , and  $f(n)$  satisfies the regularity condition, then  $T(n) = \Theta(f(n))$ .

Step 1: Calculate  $n^{\log_b a} = n^{\log_2 7} = n^{2.80}$

Step 2: Compare with  $f(n)$

since  $f(n) < n^{\log_b a}$

Step 3: Case 1 is satisfied hence complexity is given as

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{2.8})$$

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

**T(n) = aT(n/b) + f(n) where a >= 1 and b > 1**

$$a = 4$$

$$b = 2$$

$$f(n) = n^2$$

1. If  $f(n) < O(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a})$ .
2. If  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$ .
3. If  $f(n) > \Omega(n^{\log_b a})$ , and  $f(n)$  satisfies the regularity condition, then  $T(n) = \Theta(f(n))$ .

Step 1: Calculate  $n^{\log_b a} = n^{\log_2 4} = n^2$

Step 2: Compare with  $f(n)$  // Since  $f(n) = n^{\log_b a} * \log^0 n$

Step 3: Case 2 is satisfied hence complexity is given as

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



# **UNIT I**

# **INTRODUCTION TO ALGORITHM DESIGN**

# Algorithms

- A finite set of instructions or logic, written in order, to accomplish a certain predefined task.
- Algorithm is not the complete code or program.
- It is just the core logic (solution) of a problem.
- Can be expressed either as an informal high level description as pseudo code or using a flowchart.

# Characteristics of an Algorithm

- **Input**  
An algorithm should have 0 or more well defined inputs.
- **Output**  
An algorithm should have 1 or more well defined outputs
- **Unambiguous**  
Algorithm should be clear and unambiguous.
- **Finiteness**  
Algorithms must terminate after a finite no. of steps.
- **Feasibility**  
Should be feasible with the available resources.
- **Independent**  
An algorithm should have step-by-step directions which should be independent of any programming code.

# Pseudo code

- It is one of the methods that could be used to represent an algorithm.
- It is not written in a specific syntax
- Cannot be executed
- Can be read and understood by programmers who are familiar with different programming languages.
- Transformation from pseudo code to the corresponding program code easier.
- Pseudo code allows to include control structures such as WHILE, IF-THEN-ELSE, REPEAT-UNTIL, FOR, and CASE, which are available in many high level languages.

# Difference between Algorithm and Pseudocode

Algorithm	Pseudo code
A finite set of instructions or logic, written in order, to accomplish a certain predefined task.	a generic way of describing an algorithm without using any specific programming language-related notations.
It is just the core logic (solution) of a problem	It is an outline of a program, written in a form which can easily be converted into real programming statements.
Easy to understand the logic of a problem	Can be read and understood by programmers who are familiar with different programming languages.
Can be expressed either as an informal high level description as pseudo code or using a flowchart.	It is not written in a specific syntax. It allows to include control structures such as WHILE, IF-THEN-ELSE, REPEAT-UNTIL, FOR, and CASE, which are present in many high level languages.



# **UNIT I**

# **INTRODUCTION TO ALGORITHM DESIGN**

# Why to design an algorithm?

- General approaches to the construction of efficient solutions to problems
  - They provide templates suited for solving a broad range of diverse problems.
  - They can be translated into common control and data structures provided by most high-level languages.
  - The temporal and spatial requirements of the algorithms which result can be precisely analyzed.

# Algorithm Design Approaches

Based on the architecture

- Top Down Approach
- Bottom up Approach

# Algorithm Design Techniques

## I. Brute Force

- To solve a problem based on the problem’s statement and definitions of the concepts involved.
  - Easiest approach to apply
  - Useful for solving small – size instances of a problem.
- Some examples of brute force algorithms are:
    - Computing  $a^n$  ( $a > 0$ ,  $n$  a nonnegative integer) by multiplying  $a*a*...*a$
    - Computing  $n!$
    - Selection sort, Bubble sort
    - Sequential search

## 2. Divide-and-Conquer & Decrease-and-Conquer

### Step 1

Split the given instance of the problem into several smaller sub-instances

### Step 2

Independently solve each of the sub-instances

### Step 3

Combine the sub-instance solutions.

- With the divide-and-conquer method the size of the **problem instance is reduced by a factor** (e.g. half the input size),
- With the decrease-and-conquer method the size is **reduced by a constant**.

Examples of divide-and-conquer algorithms:

- Computing  $a^n$  ( $a > 0$ ,  $n$  a nonnegative integer) by recursion
- Binary search in a sorted array (recursion)
- Mergesort algorithm, Quicksort algorithm recursion)
- The algorithm for solving the fake coin problem (recursion)

### **3. Greedy Algorithms "take what you can get now" strategy**

- **At each step the choice must be locally optimal**
- Works well on optimization problems
- **Characteristics**
  1. Greedy-choice property: A global optimum can be arrived at by selecting a local optimum.
  2. Optimal substructure: An optimal solution to the problem contains an optimal solution to sub problems.
- **Examples:**
  - Minimal spanning tree
  - Shortest distance in graphs
  - Greedy algorithm for the Knapsack problem
  - The coin exchange problem
  - Huffman trees for optimal encoding

## **4. Dynamic Programming**

- Finds solutions to subproblems and stores them in memory for later use.
- Characteristics

### **1. Optimal substructure:**

Optimal solution to problem consists of optimal solutions to subproblems

### **2. Overlapping subproblems:**

Few subproblems in total, many recurring instances of each

### **3. Bottom up approach:**

Solve bottom-up, building a table of solved subproblems that are used to solve larger ones.

- **Examples:**

- Fibonacci numbers computed by iteration.
- Warshall's algorithm implemented by iterations

## 5. Backtracking methods

- The method is used for state-space search problems.

What is State-space search problems

- State-space search problems are problems, where the problem representation consists of:

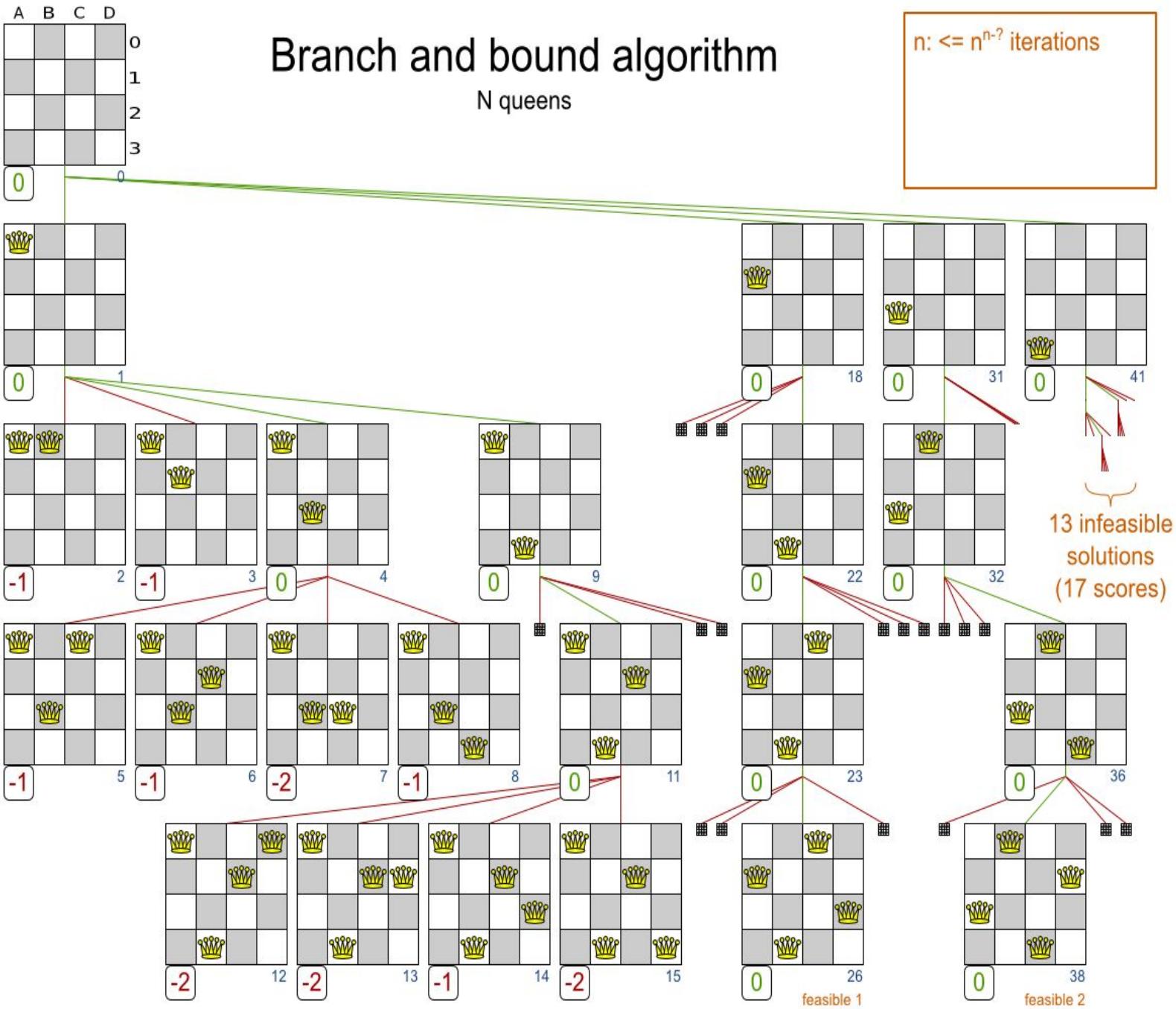
- initial state
- goal state(s)
- a set of intermediate states
- a set of operators that transform one state into another.
- a cost function – evaluates the cost of the operations (optional)
- a utility function – evaluates how close is a given state to the goal state (optional)

- The solving process solution is based on the construction of a state-space tree
- The solution is obtained by searching the tree ↴
- Examples:
  - DFS problem
  - Maze problems



## 6. Branch-and-bound

- Branch and bound is used when we can evaluate each node using the cost and utility functions.
- At each step we choose the best node to proceed further.
- Branch-and bound algorithms are implemented using a **priority queue**.
- The state-space tree is built in a **breadth-first manner**.
- **Example:**
  - 8-puzzle problem.
  - N queens problem



# Recollect

- **Different design Approaches/ Design Paradigms**

- **Brute force**

- **Divide and Conquer**

**Unit 2**

- **Greedy Algorithms**

- **Dynamic Programming**

} **Unit 3**

- **Backtracking**

**Unit 4**

- **Branch and Bound**

**Unit 5**



# **UNIT I**

# **INTRODUCTION TO ALGORITHM DESIGN**

# Algorithm Analysis

- An algorithm is said to be efficient and fast,
  - if it takes less time to execute
  - consumes less memory space.
- The performance of an algorithm is measured on the basis of
  - Time Complexity
  - Space Complexity

## ● Space Complexity

- The amount of memory space required by the algorithm in its life cycle.
- **A fixed part** For example simple variables & constant used and program size etc.
- **A variable part** For example dynamic memory allocation, recursion stacks space etc.
- Space complexity  $S(P)$  of any algorithm  $P$  is

$$S(P) = C + SP(I)$$

Where  $C$  is the fixed part

$S(I)$  is the variable part of the algorithm

## ● Time Complexity - $T(n)$

- The amount of time required by the algorithm to run to completion.
- $T(n)$  can be measured as the number of steps, provided each step consumes constant time.

# Algorithm analysis

- The **worst-case complexity** of the algorithm is the function defined by the maximum number of steps taken on any instance of size  $n$ .
- The **best-case complexity** of the algorithm is the function defined by the minimum number of steps taken on any instance of size  $n$ .
- Finally, the **average-case complexity** of the algorithm is the function defined by the average number of steps taken on any instance of size  $n$ .

# Mathematical Analysis

## ● For Non-recursive Algorithms

- There are four rules to count the operations:
  - **Rule 1: for loops - the size of the loop times the running time of the body**
    - Find the running time of statements when executed only once
    - Find how many times each statement is executed
  - **Rule 2 : Nested loops**
    - The product of the size of the loops times the running time of the body
  - **Rule 3: Consecutive program fragments**
    - The total running time is the maximum of the running time of the individual fragments
  - **Rule 4: If statement**
    - The running time is the maximum of the running times of if stmt and else stmt.

## Rule I: for loops

`for( i = 0; i < n; i++) // i = 0; executed only once: O(1)`

`// i < n; n + 1 times O(n)`

`// i++ n times O(n)`

`// total time of the loop heading:`

`// O(1) + O(n) + O(n) = O(n)`

`sum = sum + i; // executed n times, O(n)`

The loop heading plus the loop body will give:  $O(n) + O(n) = O(n)$ .

IF

- I. The size of the loop is  $n$  (loop variable runs from 0, or some fixed constant, to  $n$ )
2. The body has constant running time (no nested loops)

**Loop running time is:  $O(n)$**

$$C(n) = \sum_{i=0}^{n-1} 1 = (n-1) - 0 + 1 = n$$

# Rule 2 : Nested loops

```
sum = 0;  
for( i = 0; i < n; i++)  
    for( j = 0; j < n; j++)  
        sum++;
```

- Applying Rule 1 for the nested loop (the 'j' loop) we get  $O(n)$  for the body of the outer loop. The outer loop runs  $n$  times, therefore the total time for the nested loops will be  $O(n) * O(n) = O(n*n) = O(n^2)$

Mathematical analysis:

Inner loop:

$$S(i) = \sum_{j=0}^{n-1} 1 = (n-1) - 0 + 1 = n$$

Outer loop:

$$C(n) = \sum_{i=0}^{n-1} S(i) = \sum_{i=0}^{n-1} n = n * \sum_{i=0}^{n-1} 1 = n((n-1) - 0 + 1) = n^2$$

```
for( i = 0; i < n; i++)  
    for( j = i; j < n; j++)  
        sum++;
```

- Here, the number of the times the inner loop is executed depends on the value of i

i = 0, inner loop runs n times

i = 1, inner loop runs (n-1) times

i = 2, inner loop runs (n-2) times

...

i = n - 2, inner loop runs 2 times

i = n - 1, inner loop runs once.

Thus we get: ( 1 + 2 + ... + n) =  $n*(n+1)/2 = O(n^2)$

**Running time is the product of the size of the loops times the running time of the body.**

# Rule 3: Consecutive program fragment

```
sum = 0;  
for( i = 0; i < n; i++)  
    sum = sum + i;  
  
sum = 0;  
for( i = 0; i < n; i++)  
    for( j = 0; j < 2*n; j++)  
        sum++;
```

- The first loop runs in  $O(n)$  time, the second -  $O(n^2)$  time, the maximum is  $O(n^2)$

The total running time is the maximum of  
the running time of the individual  
fragments

## Rule 4: If statement

```
if C  
    S1;  
else  
    S2;
```

The running time is the maximum of the running times of S1 and S2.

The running time is the maximum of the running times of if stmt and else stmt

# Exercise

a.

```
sum = 0;  
for( i = 0; i < n; i++)  
    for( j = 0; j < n * n; j++)  
        sum++;
```

Ans :  $O(n^3)$

b.

```
sum = 0;  
for( i = 0; i < n; i++)  
    for( j = 0; j < i; j++)  
        sum++;
```

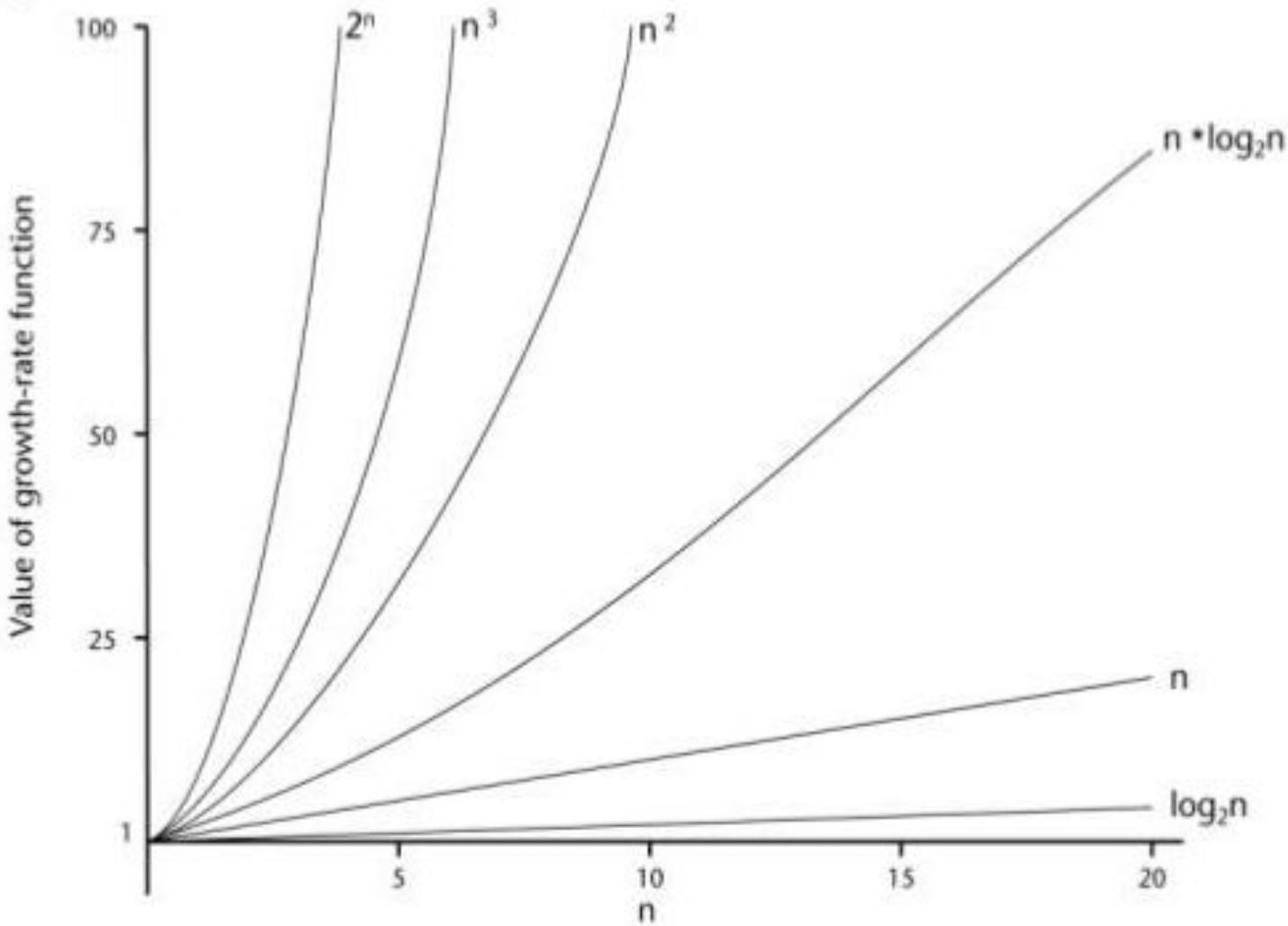
Ans :  $O(n^2)$

c.

# Order of Growth Function

	<i>constant</i>	<i>logarithmic</i>	<i>linear</i>	<i>N-log-N</i>	<i>quadratic</i>	<i>cubic</i>	<i>exponential</i>
<i>n</i>	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	$1.84 \times 10^{19}$

(b)



# Linear Search Analysis

```
linear(a[n], key)
    for( i = 0; i < n; i++)
        if (a[i] == key)
            return i;
        else return -1;
```

- Worst Case :  $O(n)$  // Rule 1 for loop explanation
- Average Case :

If the key is in the first array position: 1 comparison

If the key is in the second array position: 2 comparisons

...

If the key is in the  $i$ th position :  $i$  comparisons

...

So average all these possibilities:  $(1+2+3+\dots+n)/n = [n(n+1)/2]/n = (n+1)/2$  comparisons.

The average number of comparisons is  $(n+1)/2 = \Theta(n)$ .

- Best Case :  $O(1)$

# Binary Search Analysis

```
binarysearch(a[n], key, low, high)
while(low<high)
{
    mid = (low+high)/2;
    if(a[mid]=key)
        return mid;
    elseif (a[mid] > key)
        high=mid-1;
    else
        low=mid+1;
}
return -1;
```

## **Worst case analysis:** The key is not in the array

Let  $T(n)$  be the number of comparisons done in the worst case for an array of size  $n$ . For the purposes of analysis, assume  $n$  is a power of 2, ie  $n = 2^k$ .

$$\text{Then } T(n) = 2 + T(n/2)$$

$$= 2 + 2 + T\left(\frac{n}{2^2}\right) \quad // \text{ 2nd iteration}$$

$$= 2 + 2 + 2 + T(n/2^3) \quad // \text{ 3rd iteration}$$

...

$$= i * 2 + T(n/2^i) \quad // \text{i}^{\text{th}} \text{ iteration}$$

$$\dots = k * 2 + T(1)$$

Note that  $k = \log n$ , and that  $T(1) = 2$ .

So  $T(n) = 2\log n + 2 = O(\log n)$

# Bubble sort analysis

```
int i, j, temp;  
for(i=0; i<n; i++)  
{  
    for(j=0; j<n-i-1; j++)  
    {  
        if( a[j] > a[j+1] )  
        {  
            temp = a[j];  
            a[j] = a[j+1];  
            a[j+1] = temp;  
        }  
    }  
}
```

**Worst Case:** In Bubble Sort,  $n-1$  comparisons will be done in 1st pass,  $n-2$  in 2nd pass,  $n-3$  in 3rd pass and so on. So the total number of comparisons will be

$$(n-1)+(n-2)+(n-3)+\dots+3+2+1$$

$$\text{Sum} = n(n-1)/2$$

Hence the complexity of Bubble Sort is  $\mathcal{O}(n^2)$ .

**Best-case** Time Complexity will be  $\mathcal{O}(n)$ , it is when the list is already sorted.

# Insertion Sorting Analysis

```
int i, j, key;  
for(i=1; i<n; i++)  
{  
    key = a[i];  
    j = i-1;  
    while(j>=0 && key < a[j])  
    {  
        a[j+1] = a[j];  
        j--;  
    }  
    a[j+1] = key;  
}
```

- Worst Case Time Complexity :  $O(n^2)$
- Best Case Time Complexity :  $O(n)$
- Average Time Complexity :  $O(n^2)$



# **UNIT I**

# **INTRODUCTION TO ALGORITHM DESIGN**

# Asymptotic Notations

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

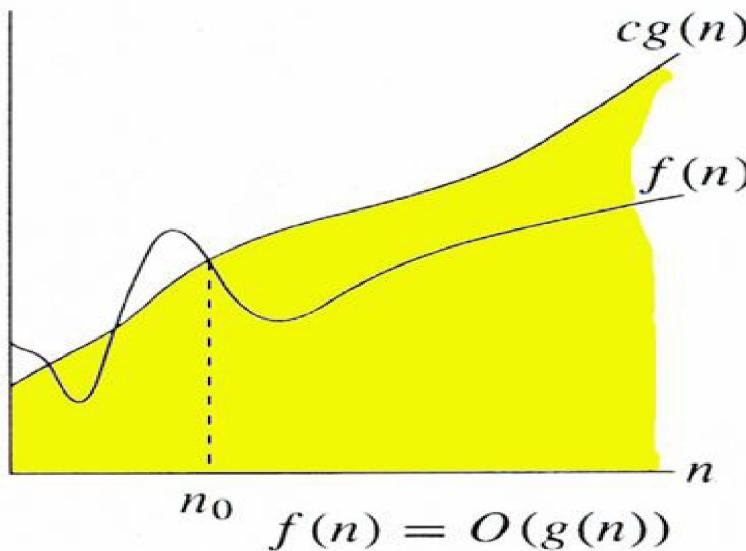
# Asymptotic Analysis

- Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation.
  - **O Notation**
  - **$\Omega$  Notation**
  - **$\Theta$  Notation**

## ● Big Oh Notation, O

- It measures the worst case time complexity or longest amount of time an algorithm can possibly take to complete.

$O(g(n)) = \{ f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$   
 $0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$



$0 \leq f(n) \leq cg(n)$  for all  $n \geq n_0\}$

If  $f(n) = 3n+2$   
 $g(n)=n$

Then

$$3n+2 \leq cn$$

For instance  $c=4$

$$3n+2 \leq 4n$$

$$n \geq 2$$

Hence  $f(n) = O(g(n))$

If  $f(n) = 3n+2$   $g(n)=n^2$

Then

$$3n+2 \leq cn^2$$

For instance  $c=1$

$$n=1,2,3,4,5,6\dots$$

$$3n+2 \leq n^2$$

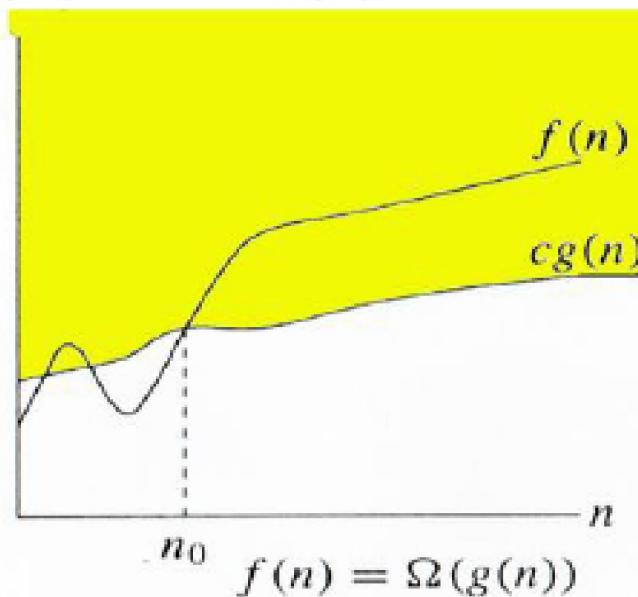
$$n \geq 5$$

Hence  $f(n) = O(g(n))$

## ● $\Omega$ Notation: (Best Case)

- $\Omega$  notation provides an asymptotic lower bound

$\Omega(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$   
 $0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}.$



$0 \leq cg(n) \leq f(n)$  for all  $n \geq n_0$ .

If  $f(n) = 3n+2$

$g(n)=n$

Then

$f(n) \geq cg(n)$

For instance  $c=1$

$3n+2 \geq n$

Hence  $f(n) = \Omega(g(n))$

If  $f(n) = 3n+2$   $g(n)=n^2$

Then

$3n+2 \geq cn^2$

For instance  $c=1$

$n=1, 2, 3, 4, 5, 6, \dots$

$3n+2 \geq n^2$

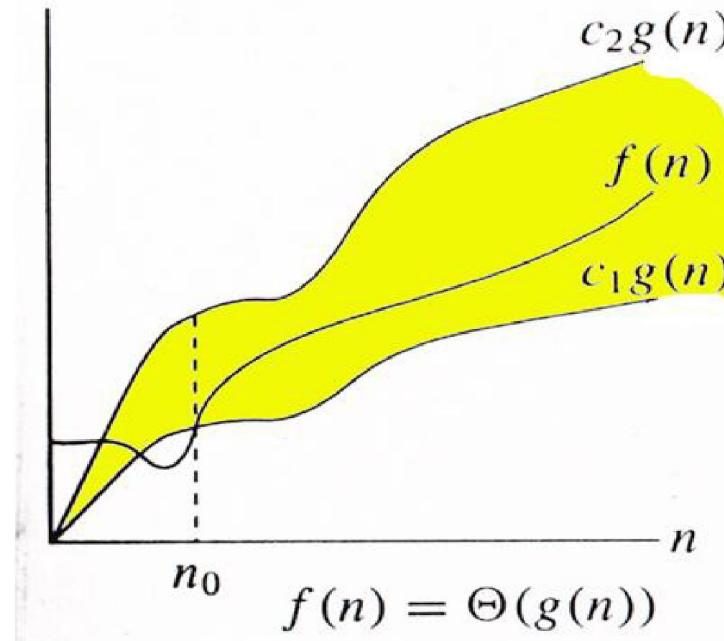
Hence  $f(n) = \Omega(n \log n)$

$\log(n)$

## ● $\Theta$ Notation:

$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that}$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$



$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$  for all  $n \geq n_0$

If  $f(n) = 3n+2$

$$g(n)=n$$

Then

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

$c_1, c_2 > 0$  and  $N > n_0$

For instance  $c_2=4$

$$f(n) \leq c_2 g(n)$$

$$3n+2 \leq 4n \quad n_{0=1}$$

$$f(n) \geq c_1 g(n)$$

For instance  $c_1=1$

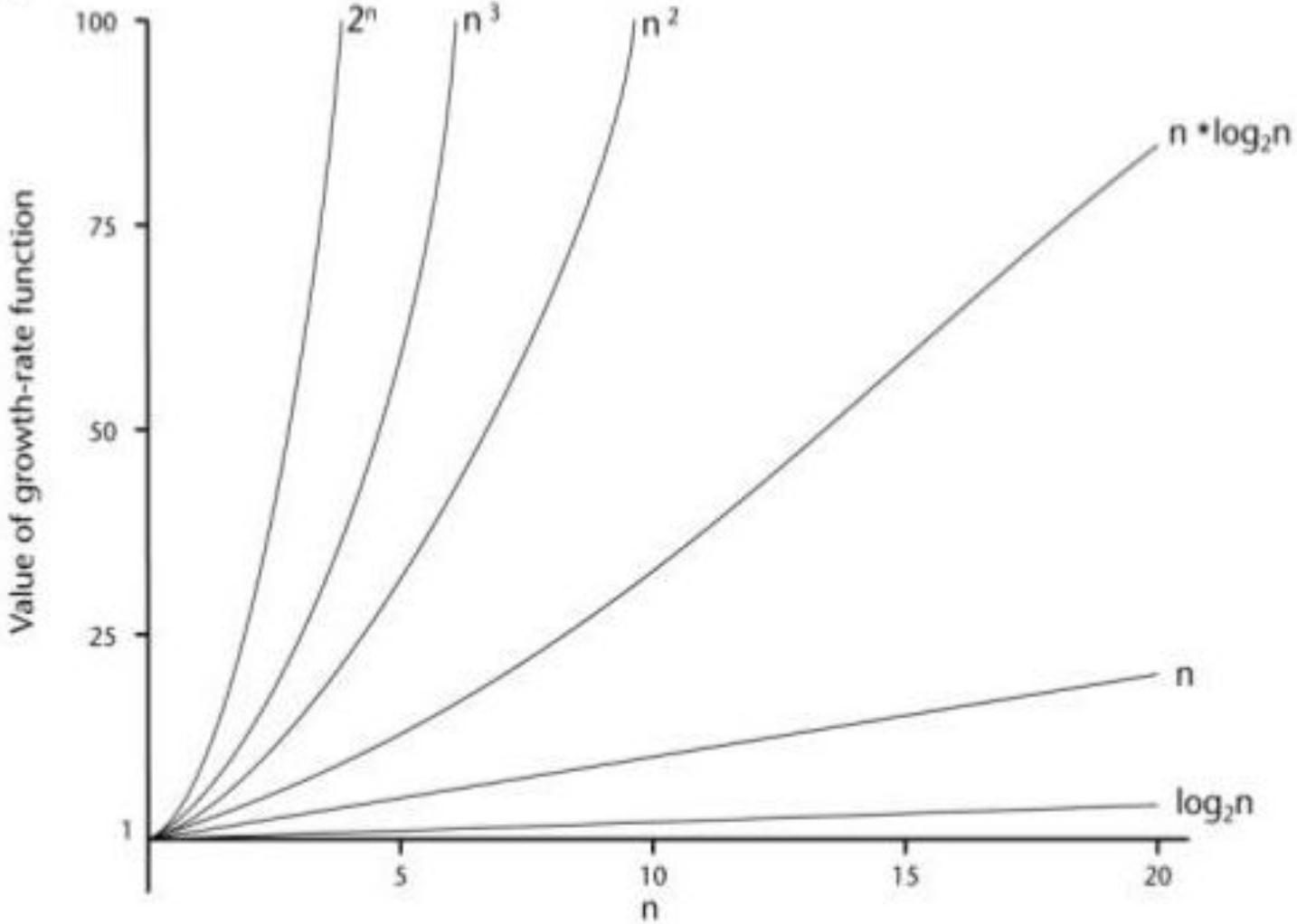
$$3n+2 \geq n$$

Hence  $f(n) = \Theta(n)$

# Order of Growth Function

	<i>constant</i>	<i>logarithmic</i>	<i>linear</i>	<i>N-log-N</i>	<i>quadratic</i>	<i>cubic</i>	<i>exponential</i>
<i>n</i>	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	$1.84 \times 10^{19}$

(b)





# **UNIT I**

# **INTRODUCTION TO ALGORITHM DESIGN**

# Recursion recall

- Functions calls itself
- consists of one or more base cases and one or more recursive cases.

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ f(n - 1) + 1 & \text{otherwise} \end{cases}$$

$$f(0) = 0 \quad f(n) = f(n - 1) + 1 \quad \text{for all } n > 0$$

# Mathematical Analysis - Induction

- Consider a recursive algorithm to compute the maximum element in an array of integers.
- You may assume the existence of a function “max(a,b)” that returns the maximum of two integers a and b .

*Function FIND – ARRAY – MAX (A , n )*

```
1: if (n = 1) then  
2:     return (A[1])  
3: else  
4:     return (max (A[n] , FIND – ARRAY – MAX (A , n – 1)))  
5: end if
```

# Solution

- Let  $p(n)$  stand for the proposition that Algorithm finds and returns the maximum integer in the locations  $A[1]$  through  $A[n]$ . Accordingly, we have to show that  $(\forall n) p(n)$  is true.
- BASIS:** When there is only one element in the array , i.e.,  $n=1$ , then this element is clearly the maximum element and it is returned on Line 2. We thus see that  $p(1)$  is true.
- INDUCTIVE STEP:** Assume that Algorithm finds and returns the maximum element, when there are exactly  $k$  elements in  $A$ .
- Now consider the case in which there are  $k + 1$  elements in  $A$ . Since  $(k + 1) > 1$ , Line 4 will be executed.
- From the inductive hypothesis, we know that the maximum elements in  $A[1]$  through  $A[k]$  is returned. Now the maximum element in  $A$  is either  $A[k+1]$  or the maximum element in  $A[1]$  through  $A[k]$  (say  $r$ ). Thus, returning the maximum of  $A[k+1]$  and  $r$  clearly gives the maximum element in  $A$ , thereby proving that  $p(k) \rightarrow p(k+1)$ .
- By applying the principle of mathematical induction, we can conclude that  $(\forall n) p(n)$  is true, i.e., Algorithm is correct.

- Find the exact solution to the recurrence relation using mathematical induction method

$$T(1) = 0$$

$$T(n) = 2 T\left(\frac{n}{2}\right) + n, \quad n \geq 2$$

- Solution is  $T(n)=n\log n$

# Solution

- **Basis:** At  $n = 1$ , both the closed form and the recurrence relation agree ( $0=0$ ) and so the basis is true.
- **Inductive step:** Assume that  $T(r) = r \log r$  for all  $1 \leq r \leq k$ . Now consider  $T(k+1)$ . As per the recurrence relation, we have,

$$\begin{aligned}T(k + 1) &= 2 T\left(\frac{k + 1}{2}\right) + (k + 1); \text{ since } (k + 1) \geq 2 \\&= 2 \left( \frac{(k + 1)}{2} \log \frac{k + 1}{2} \right) + (k + 1) \text{ as per the inductive hypothesis; since } \frac{k + 1}{2} < k \\&= (k + 1) [\log(k + 1) - \log 2] + (k + 1) \\&= (k + 1) \log(k + 1) - (k + 1) + (k + 1) \\&= (k + 1) \log(k + 1)\end{aligned}$$

- We can therefore apply the principle of mathematical induction to conclude that the exact solution to the given recurrence is  $n \log n$ .



# **UNIT I**

# **INTRODUCTION TO ALGORITHM DESIGN**

# Substitution method

- Guess the solution.
- Use induction to find the constants and show that the solution works.

# Example

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

1. Guess:  $T(n) = n \lg n + n$ .

[Here, we have a recurrence with function, rather than asymptotic notation, and the solution is also asymptotic. We'll have to check boundary conditions as well.]

2. Induction:

**Basis:**  $n = 1 \Rightarrow n \lg n + n = 1 = T(n)$

**Inductive step:** Inductive hypothesis is that  $T(k) = k \lg k + k$  for all  $k < n$ . We'll use this inductive hypothesis for  $T(n/2)$ .

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \\ &= 2\left(\frac{n}{2} \lg \frac{n}{2} + \frac{n}{2}\right) + n \quad (\text{by inductive hypothesis}) \\ &= n \lg \frac{n}{2} + n + n \\ &= n(\lg n - \lg 2) + n + n \\ &= n \lg n - n + n + n \\ &= n \lg n + n. \end{aligned}$$

■

Generally, we use asymptotic notation:

- We would write  $T(n) = 2T(n/2) + \Theta(n)$ .
- We assume  $T(n) = O(1)$  for sufficiently small  $n$ .
- We express the solution by asymptotic notation:  $T(n) = \Theta(n \lg n)$ .
- We don't worry about boundary cases, nor do we show base cases in the substitution proof.
  - $T(n)$  is always constant for any constant  $n$ .
  - Since we are ultimately interested in an asymptotic solution to a recurrence, it will always be possible to choose base cases that work.
  - When we want an asymptotic solution to a recurrence, we don't worry about the base cases in our proofs.
  - When we want an exact solution, then we have to deal with base cases.

For the substitution method:

- Name the constant in the additive term.
- Show the upper ( $O$ ) and lower ( $\Omega$ ) bounds separately. Might need to use different constants for each.

# Another Example

- How to prove by the substitution method that if  $T(n) = T(n-1) + \Theta(n)$  then  $T(n) = \Theta(n^2)$

We guess that  $T(n) \leq O(n^2)$ .

$$\begin{aligned}T(n) &\leq c_1(n-1)^2 + \Theta(n) \\&\leq c_1(n-1)^2 + c_0n \\&\leq c_1(n^2 - 2n + 1) + c_0n \\&\leq c_1n^2 - (2c_1 - c_0)n + c_1 \\&\leq c_1n^2 \text{ for } n_0 \geq 1 \text{ and } c_0 > c_1\end{aligned}$$

Thus  $T(n) \in O(n^2)$ . Similarly, we can prove that  $T(n) \in \Omega(n^2)$ . Consequently,  $T(n) \in \Theta(n^2)$ .

# Video Link

<https://www.youtube.com/watch?v=Zhh9qpAVN0>



# **UNIT I**

# **INTRODUCTION TO ALGORITHM DESIGN**

# Recurrence relation - recursion

- solving recurrences
  - expanding the recurrence into a tree
  - summing the cost at each level

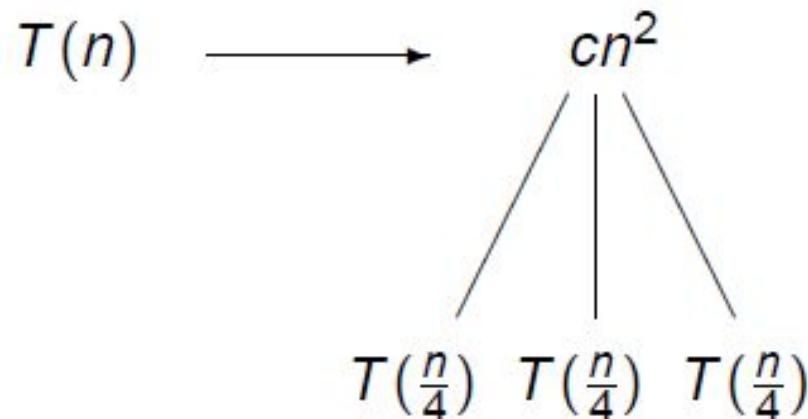
# Example

Consider the recurrence relation

$$T(n) = 3T(n/4) + cn^2 \quad \text{for some constant } c.$$

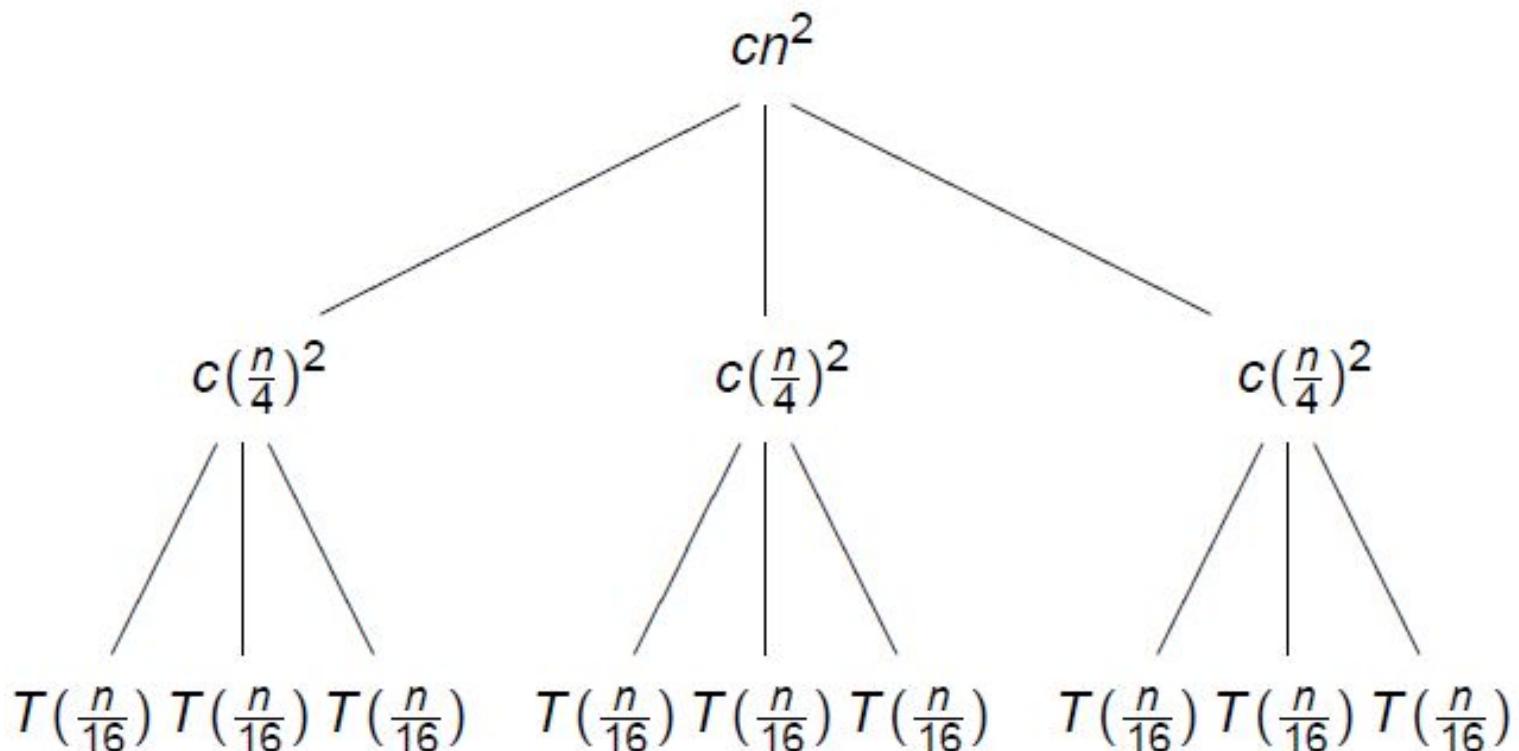
We assume that  $n$  is an exact power of 4.

In the recursion-tree method we expand  $T(n)$  into a tree:



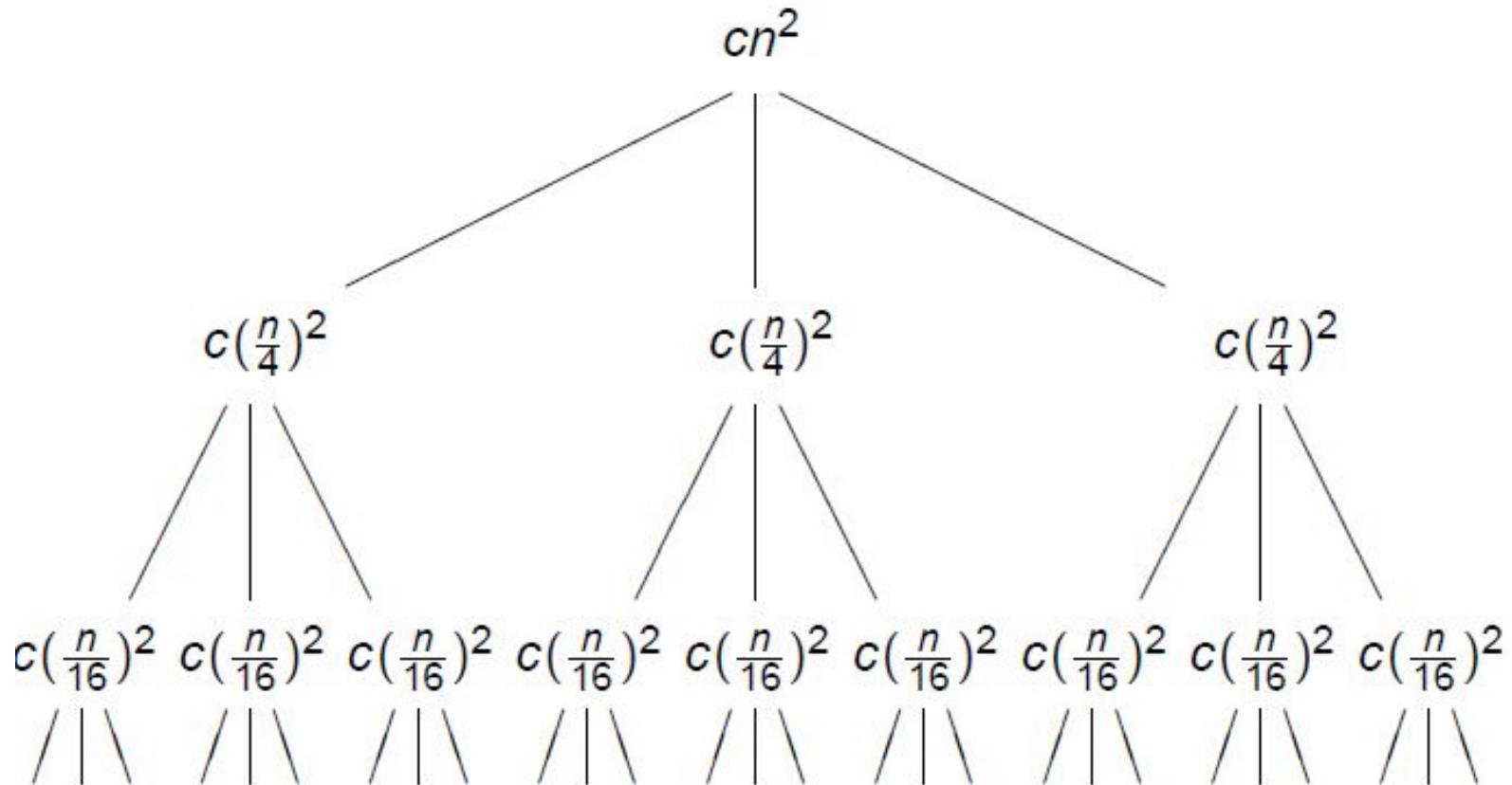
# Expand $T(n/4)$

Applying  $T(n) = 3T(n/4) + cn^2$  to  $T(n/4)$  leads to  $T(n/4) = 3T(n/16) + c(n/4)^2$ , expanding the leaves:



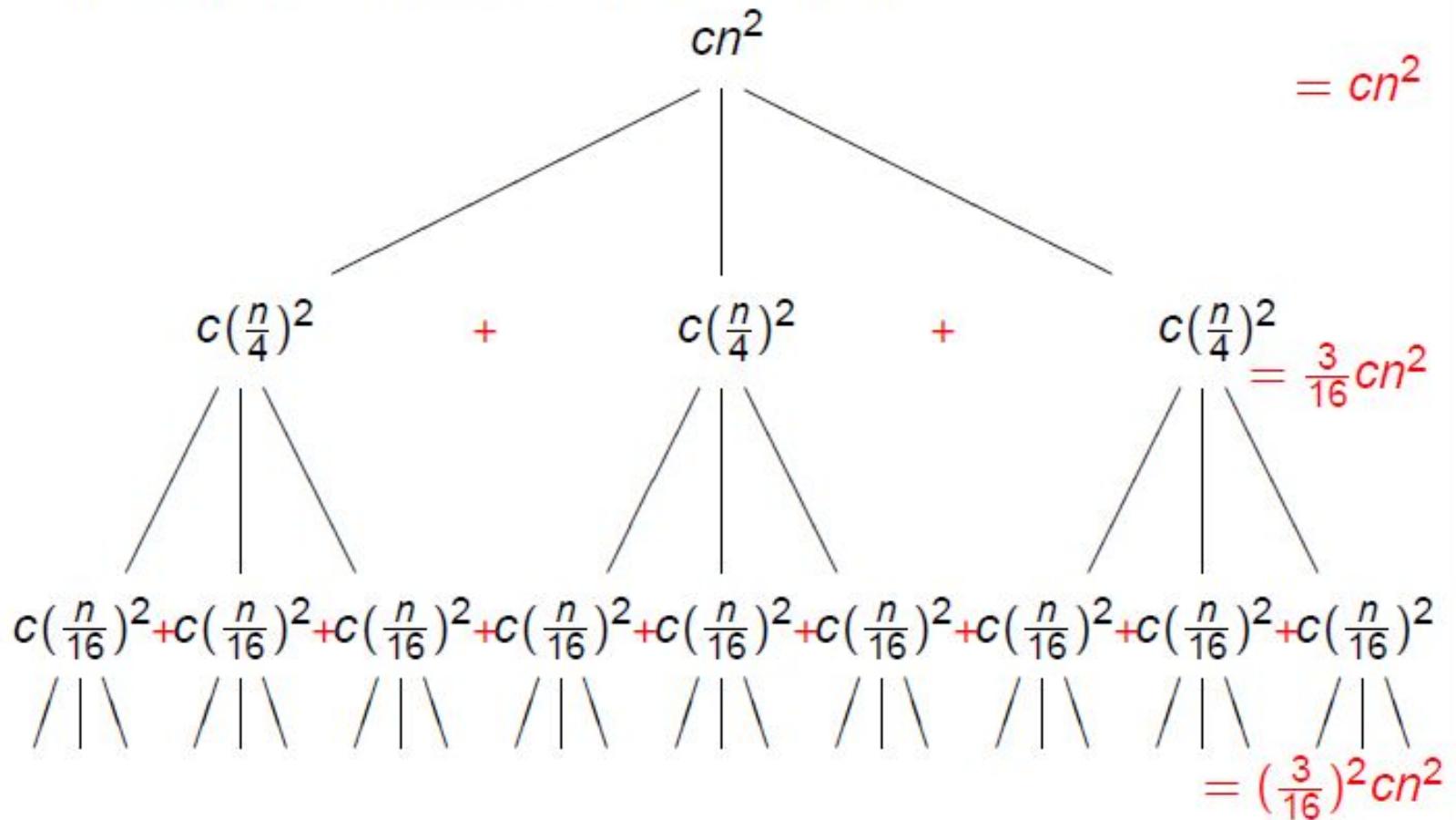
# Expand $T(n/16)$

Applying  $T(n) = 3T(n/4) + cn^2$  to  $T(n/16)$  leads to  $T(n/16) = 3T(n/64) + c(n/16)^2$ , expanding the leaves:



# Summing the cost at each level

We sum the cost at each level of the tree:



# Adding up the costs

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \dots \\ &= cn^2 \left(1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \dots\right) \end{aligned}$$

The  $\dots$  disappear if  $n = 16$ ,  
or the tree has depth at least 2 if  $n \geq 16 = 4^2$ .

For  $n = 4^k$ ,  $k = \log_4(n)$ , we have:

$$T(n) = cn^2 \sum_{i=0}^{\log_4(n)} \left(\frac{3}{16}\right)^i.$$

# Applying the geometric sum

Applying

$$S_n = \sum_{i=0}^n r^i = \frac{r^{n+1} - 1}{r - 1}$$

to

$$T(n) = cn^2 \sum_{i=0}^{\log_4(n)} \left(\frac{3}{16}\right)^i$$

with  $r = \frac{3}{16}$  leads to

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

# Polishing the result we get

Instead of  $T(n) \leq dn^2$  for some constant  $d$ , we have

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

Recall

$$T(n) = cn^2 \sum_{i=0}^{\log_4(n)} \left(\frac{3}{16}\right)^i.$$

To remove the  $\log_4(n)$  factor, we consider

$$\begin{aligned} T(n) &\leq cn^2 \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i \\ &= cn^2 \frac{-1}{\frac{3}{16} - 1} \leq dn^2, \text{ for some constant } d. \end{aligned}$$

## Verify the guess by applying substitution method

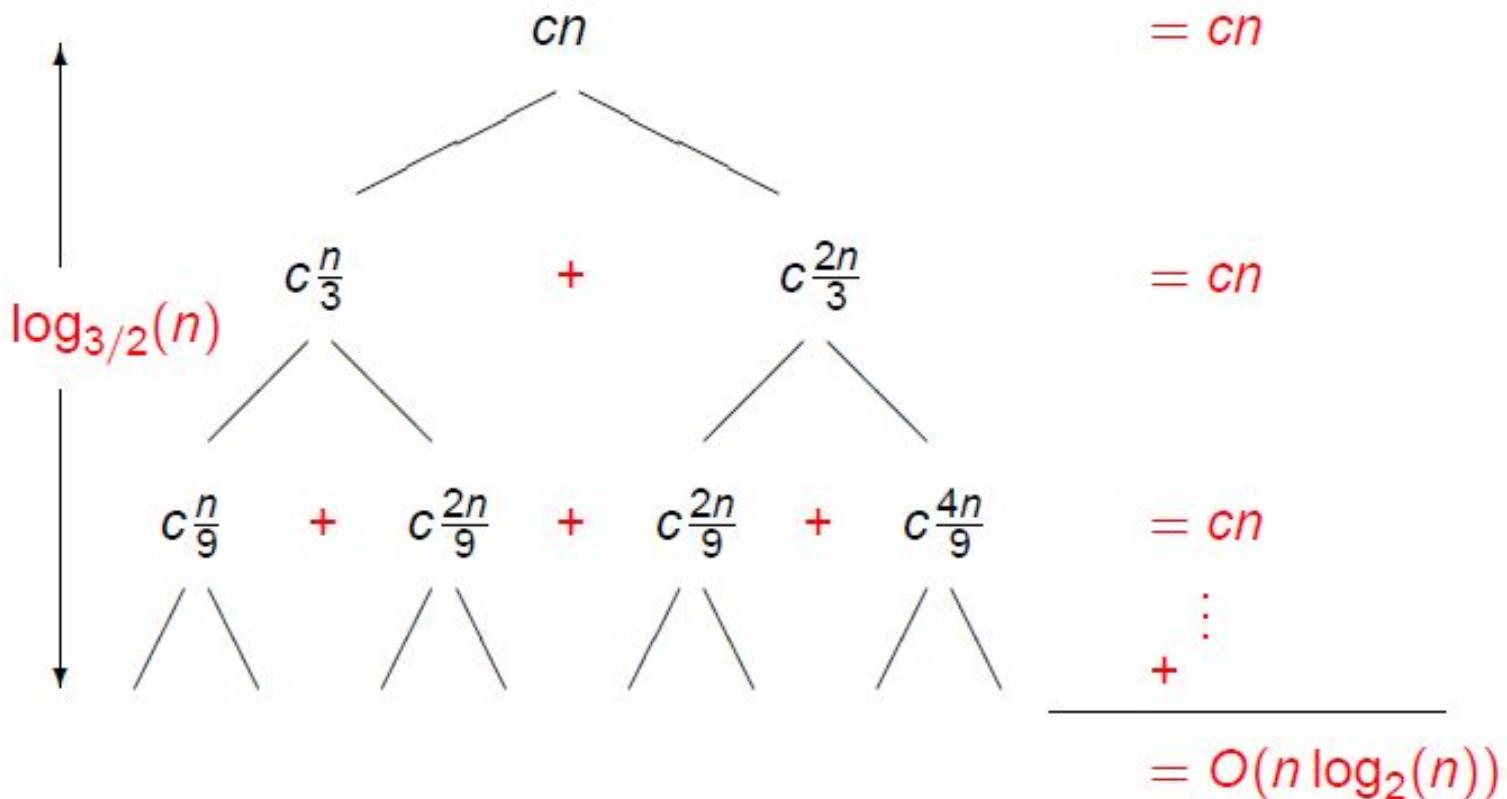
Let us see if  $T(n) \leq dn^2$  is good for  $T(n) = 3T(n/4) + cn^2$ .

Applying the substitution method:

$$\begin{aligned} T(n) &= 3T(n/4) + cn^2 \\ &\leq 3d \left(\frac{n}{4}\right)^2 + cn^2 \\ &= \left(\frac{3}{16}d + c\right)n^2 \\ &= \frac{3}{16} \left(d + \frac{16}{3}c\right)n^2 \\ &\leq \frac{3}{16} (2d)n^2, \quad \text{if } d \geq \frac{16}{3}c \\ &\leq dn^2 \end{aligned}$$

# Lets see another example

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



# Lets practice

- Consider  $T(n) = 3T(n/2) + n$ . Use a *recursion tree* to derive a guess for an asymptotic upper bound for  $T(n)$  and verify the guess with the substitution method.
- $T(n) = T(n/2) + n^2$ .
- $T(n) = 2T(n - 1) + l$ .