

**Subject Name: Design and Analysis of  
Algorithms**

**Module No: 1**

**Module Name: Introduction**

# Course Objectives

- 1 To provide mathematical approach for Analysis of Algorithms.
- 2 To understand and solve problems using various algorithmic approaches.
- 3 To analyze algorithms using various methods.



# What we learn in this course?

Objective 1 : To provide mathematical approach for Analysis of Algorithms.

## Types of Algorithm

- **Recursive**
- **Non-Recursive**
- **Divide and Conquer Approach**
- **Greedy Method Approach**
- **Dynamic Programming Approach**
- **Backtracking and Branch and Bound**
- **String Matching Algorithms**
- Asymptotic analysis using Big –Oh, Omega, Theta notations
  - **The substitution method**
  - **Recursion tree method**
  - **Master method**

## Methods of Analysis



# **What we learn in this course?**

Objective 2 : To understand and solve problems using various algorithmic approaches.

## Types of Algorithm

- **Divide and Conquer Approach**
- **Greedy Method Approach**
- **Dynamic Programming Approach**
- **Backtracking and Branch and Bound**
- **String Matching Algorithms**

# What we learn in this course?

Objective 3 : To analyze algorithms using various methods.

## Methods of Analysis

- Asymptotic analysis using Big –Oh, Omega, Theta notations
  - The substitution method
  - Recursion tree method
  - Master method

# Course Outcome

---

- CO1 : Analyze the running time and space complexity of algorithms.
- CO2 : Describe, apply and analyze the complexity of divide and conquer strategy.
- CO3 : Describe, apply and analyze the complexity of greedy strategy.
- CO4 : Describe, apply and analyze the complexity of dynamic programming strategy.
- CO5 : Explain and apply backtracking, branch and bound.
- CO6 : Explain and apply string matching techniques.

# Outline-Module 1

---

S.No	Topic
1	Performance analysis , space and time complexity Mathematical background for algorithm analysis
2	Growth of function – Big –Oh ,Omega , Theta notation
3	Analysis of Selection Sort
4	Analysis of Insertion Sort.
5	The substitution method and Recursion tree method



Module No 1: Introduction

---

# Lecture No: 1

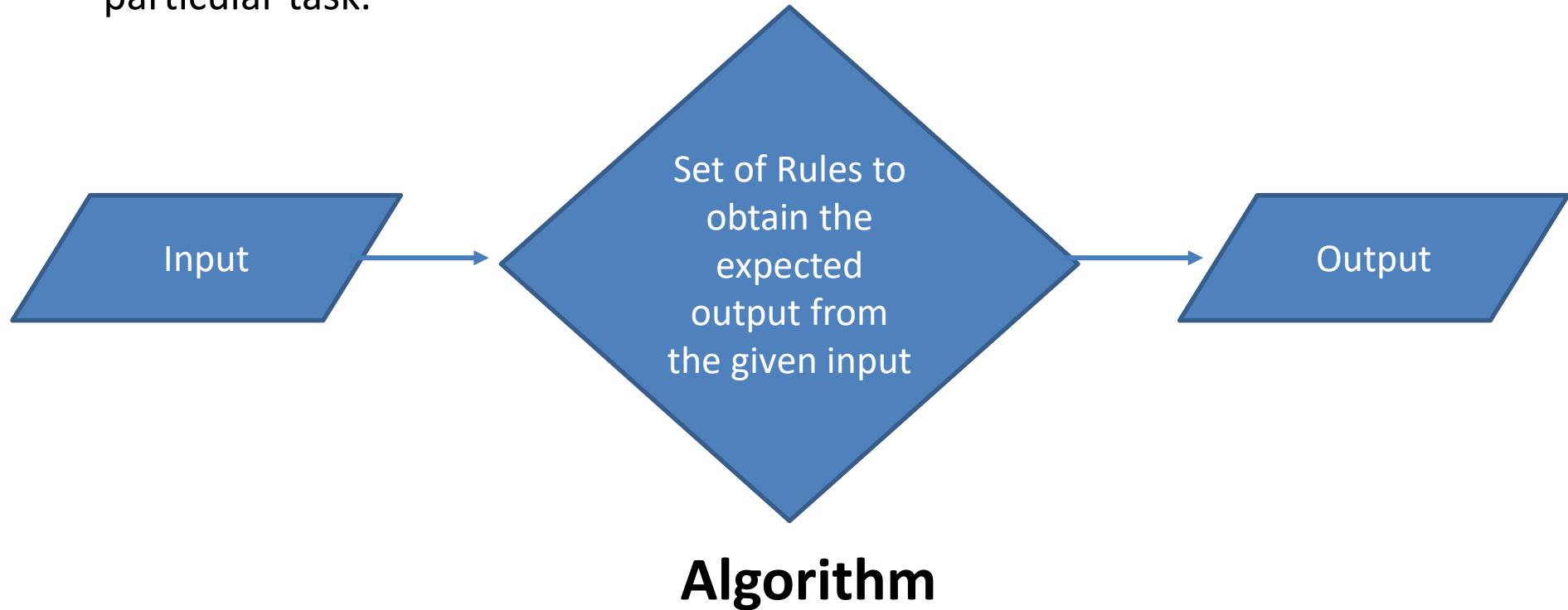
## Performance analysis , space and time complexity Mathematical background for algorithm analysis



## Algorithm

---

- An algorithm is a finite set of instruction that if followed accomplishes a particular task.



## Google Hangout Video Conference call

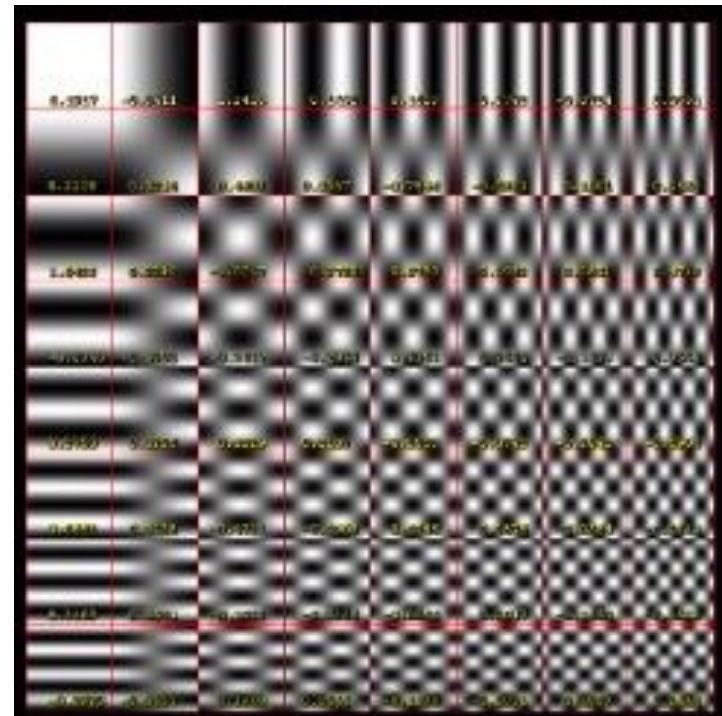
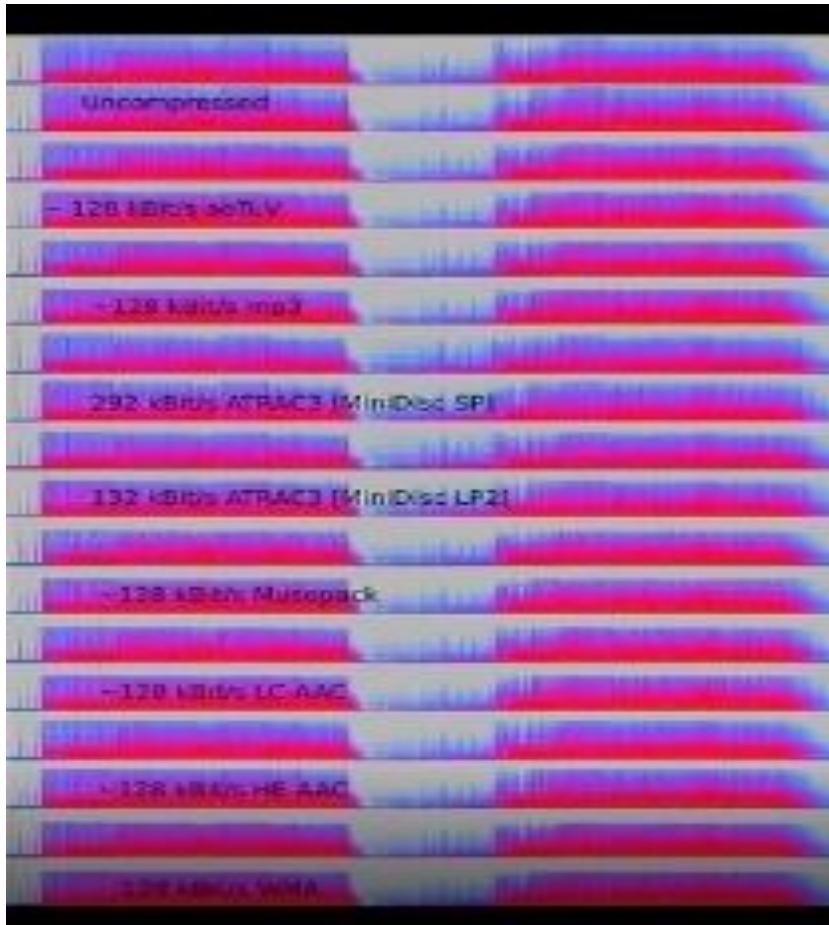
---



•How does Google Hangouts transmit live video across the Internet so quickly?

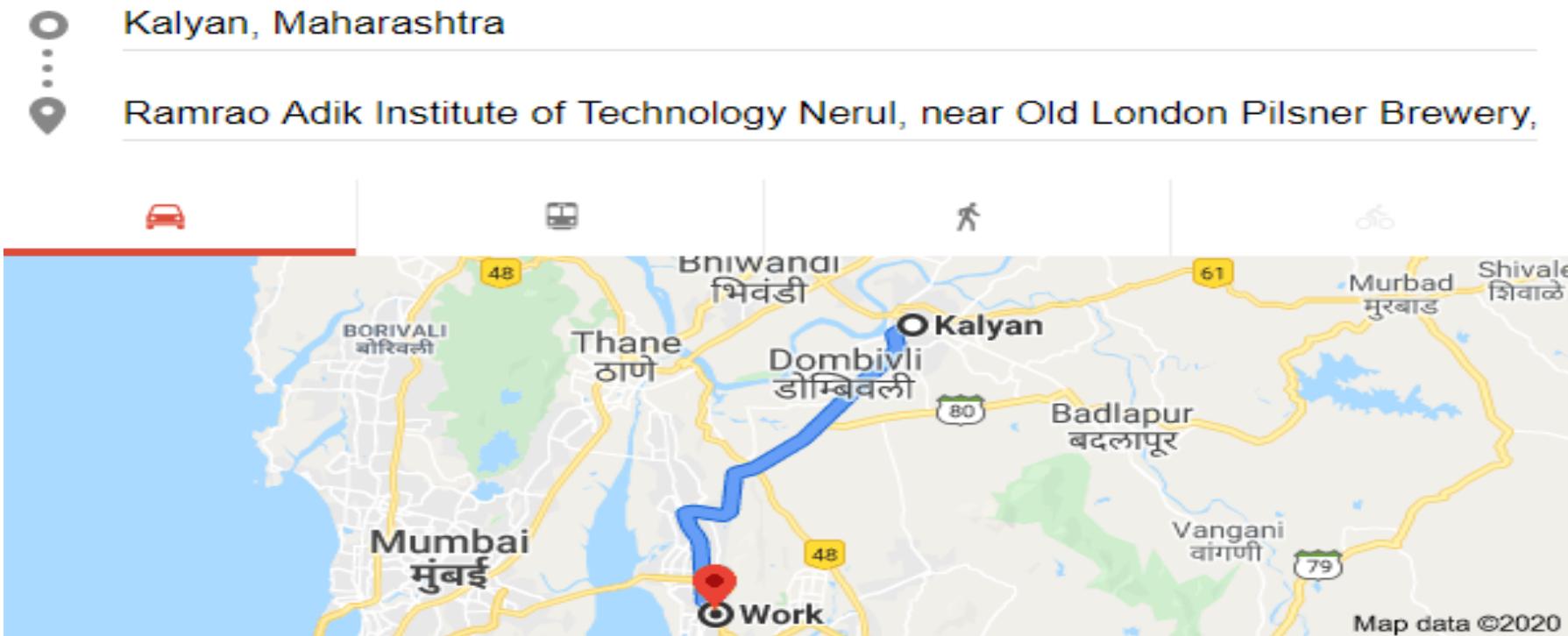
## Google Hangout Video Conference call

# Audio and Video Compression Algorithm



## Google Map

How does Google Maps figure out how to get from one place to another?



59 min (30.6 km) via Kalyan - Shilphata Rd

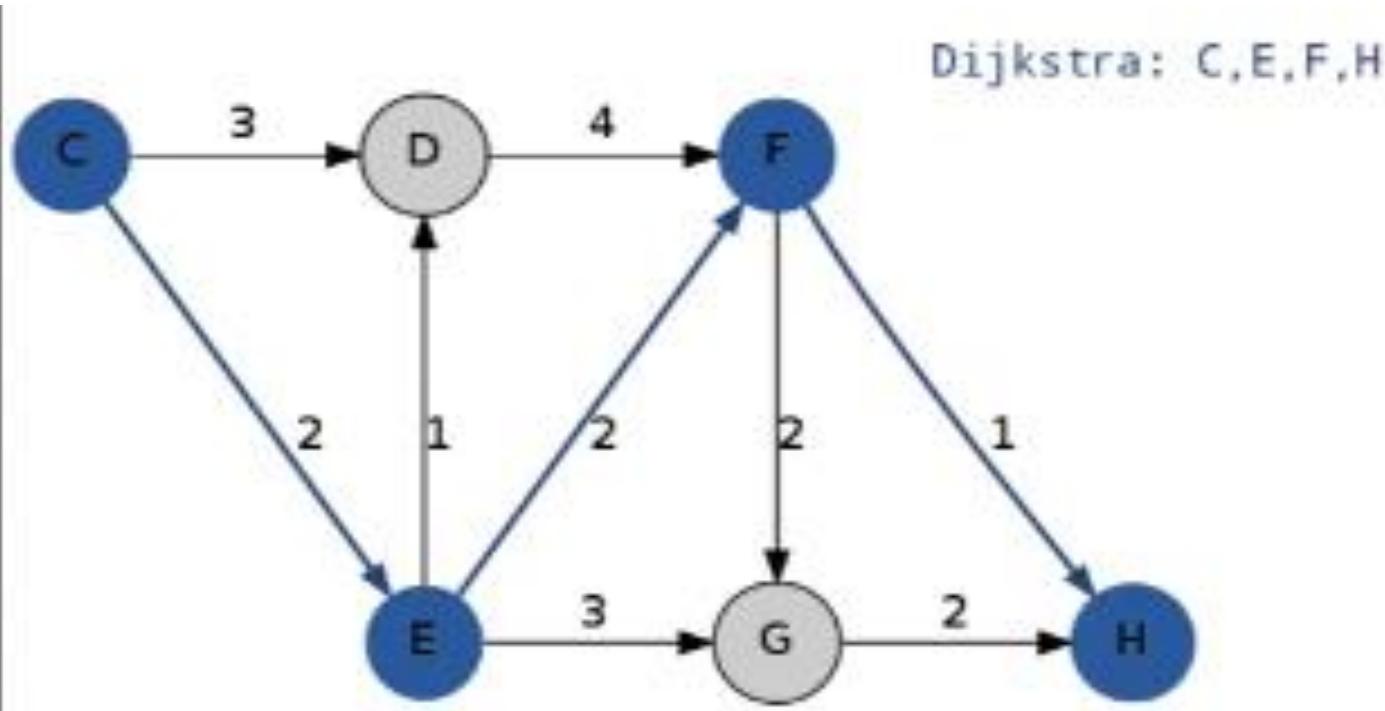
1 h 12 min (39.9 km) via Thane - Belapur Rd

Algorithm??



UNIVERSITY  
RAMRAO ADIK  
INSTITUTE OF TECHNOLOGY  
NAVI MUMBAI

# Route finding Algorithm



## Activity

---

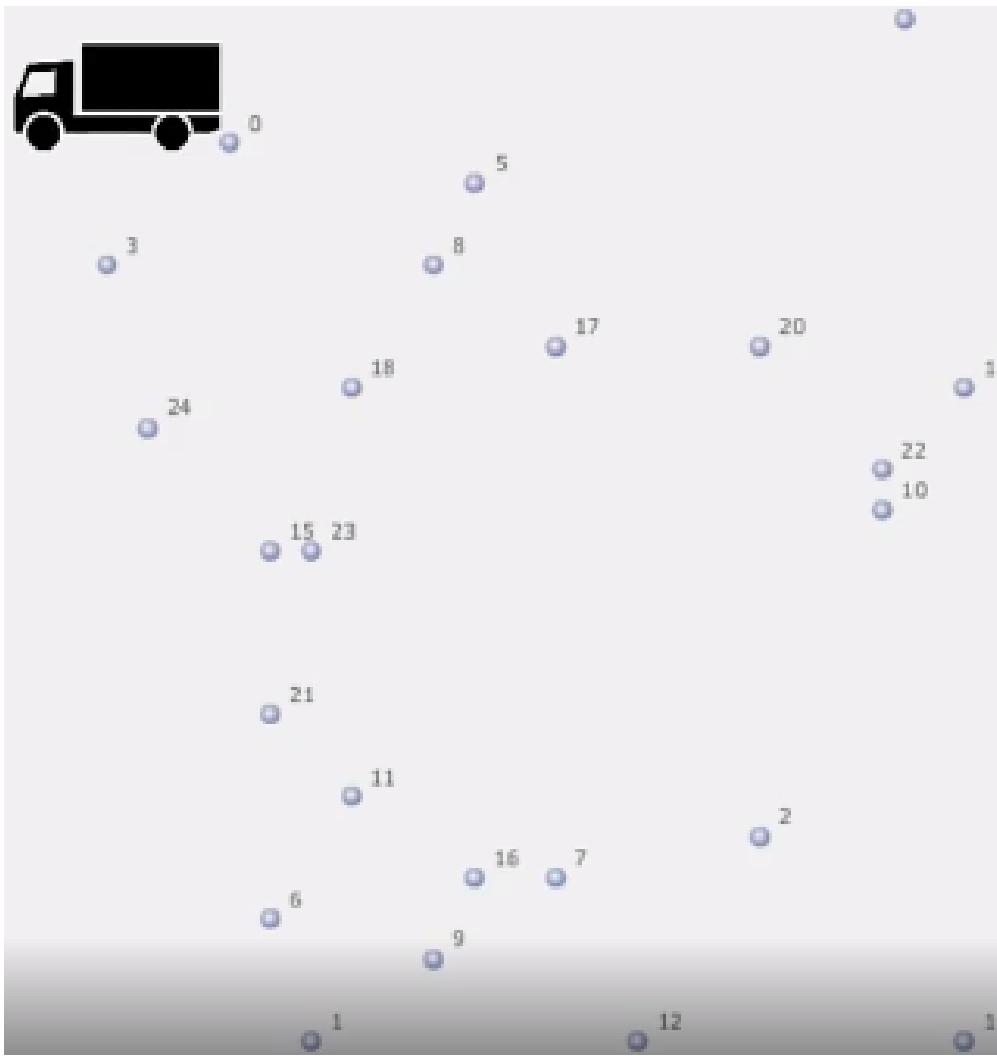
# What makes a good algorithms?

- A. Correctness
- B. Efficiency



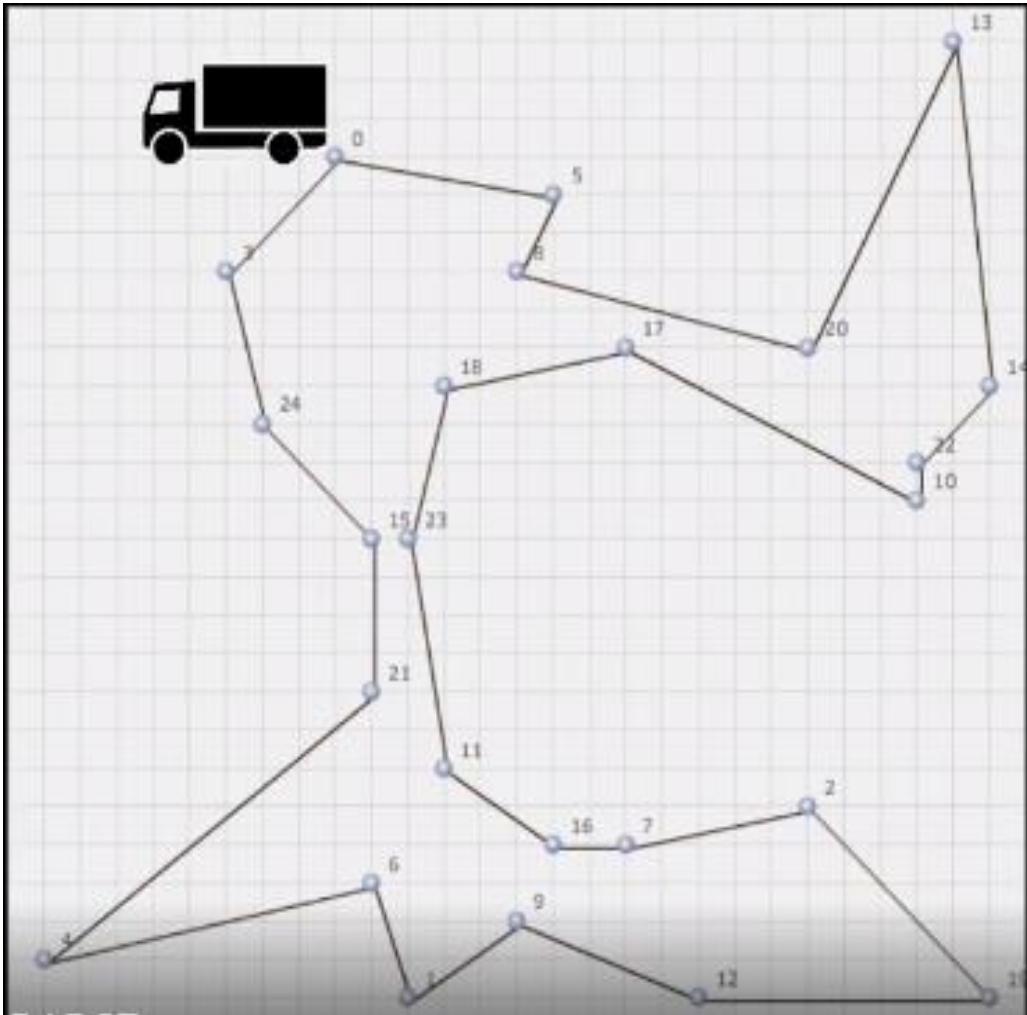
## Example: Delivery Truck

---



1 Truck  
25 location  
=  $24!$  Routes  
=  $620,448,401,733,239,439,360,000$  routes

## Example: Delivery Truck



1Truck  
25 location  
.solved using  
“Nearest  
insertion”  
algorithm

## Characteristics of an Algorithm

---

Well Defined Inputs

Clear and Unambiguous

Feasible

Finiteness

Language Independent

Well Defined Outputs

## Activity

---

# Why do we need to analyze algorithms?



## Need of Analyzing Algorithm

---

- To measure the performance
- Comparison of different algorithms
- Can we find a better one? Is this the best solution?
- Running time analysis
- Memory usage



# Introduction to analysis of Algorithms

---

## Analysis of Algorithms

To analyze an algorithm means determining the amount of resources (such as time and storage) needed to execute it

- *Efficiency* of an algorithm can be measured in terms of:

Execution time (*time complexity*)

*Time complexity*: A measure of the amount of time required to execute an *algorithm*

The amount of memory required (*space complexity*)

Calculated by considering data and their size



# How we measure Space Complexity?



## Space Complexity

---

- Fixed Space Requirements ( $C$ )  
**Independent of the characteristics of the inputs and outputs**
  - instruction space
  - space for simple variables, fixed-size structured variable, constants
- Variable Space Requirements ( $S_P(I)$ )  
**depend on the instance characteristic  $I$** 
  - number, size, values of inputs and outputs associated with  $I$
  - recursive stack space, formal parameters, local variables, return address

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



## Example

---

Algorithm ADD(x,n)

{

//Problem Description: The algorithm performs addition of all the elements in an array. Array is of floating type

//Input: An array x and n is total number of element in array

// output: return sum which is of data type float.

Sum=0.0                   -----1 unit of space by Sum

for i=1 to n do           -----1 unit of space by i   -----1 unit of space by n

    Sum=Sum + x[i]       -----n unit of space by x[]

return Sum

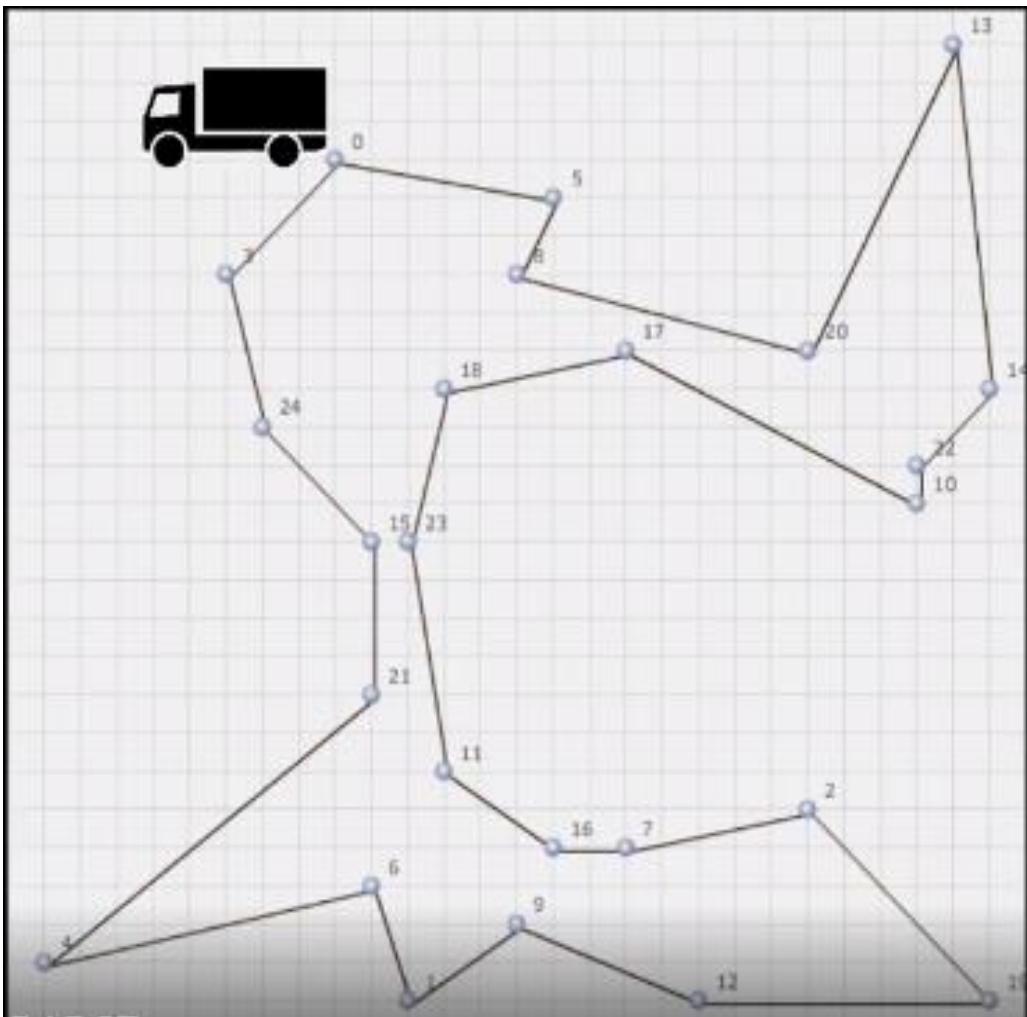
}

$$S(p) \geq (n+3)$$

# How we measure Time Complexity?



## Example: Delivery Truck



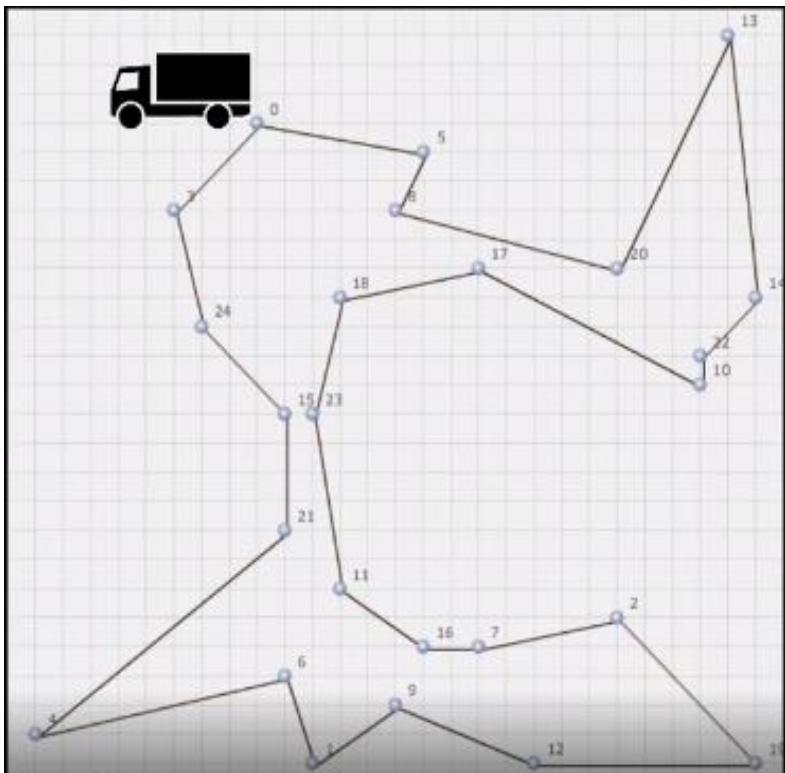
Let Suppose

1. Python  
Intel Dual core  
6GB RAM  
**TIME=20MIN**

2. Python  
Intel Quad Core  
16GB RAM  
**TIME=7MIN**



## Example: Delivery Truck



Number of nodes	Nearest Insertion	Brute force
1	1	1
2	4	1
3	9	2
4	16	6
...		
25	625	620,448,401, 733,239,439, 360,000
n	$n^2$	$n!$

## Time Complexity $T(P)=C+TP(I)$

---

- Compile time (C)  
independent of instance characteristics
- run (execution) time  $T_P$

Definition

A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

$$T_P(n)=c_aADD(n)+c_sSUB(n)+c_mMUL(n)+c_dDIV(n)\dots$$

Where,

n is the instance characteristics

$c_a, c_m, c_s, c_d$  are the time for addition, multiplication, subtraction, division so on..



## Example

Lets suppose system take

- 1 unit time for arithmetic and logical operations
- 1 unit time for assignment and return statements

1. Sum of 2 numbers :

```
Sum(a,b)
```

```
{
```

```
return a+b
```

//Takes 2 unit of time(constant) one for arithmetic operation and one for  
return. cost=2 no of times=1

```
}
```

2. Sum of all elements of a list :

```
list_Sum(A,n)
```

{ //A->array and n->number of elements in the array

```
total =0
```

// cost=1 no of times=1

```
for i=0 to n-1
```

// cost=2 no of times=n+1 (+1 for the end false condition)

```
sum = sum + A[i]
```

// cost=2 no of times=n

```
return sum
```

// cost=1 no of times=1

```
}
```

$$T_{\text{sum}} = 2 \text{ (proportional to constant)}$$

$$T_{\text{sum}} = 1 + 2 * (n+1) + 2 * n + 1 = 4n + 1 = C_1 * n + C_2$$

(proportional to n)



Module No 1: Introduction

---

# Lecture No:2

## Growth of function – Big Oh ,Omega , Theta notation



## Asymptotic Notation( Growth of function)

---

- The efficiency can be measured by computing time complexity of each algorithm
- Asymptotic notation is a shorthand way to represent the time complexity.
- It is also defines in terms of function
- Various notation such as  $\Omega$ ,  $\Theta$ , and  $O$  used as asymptotic notation.

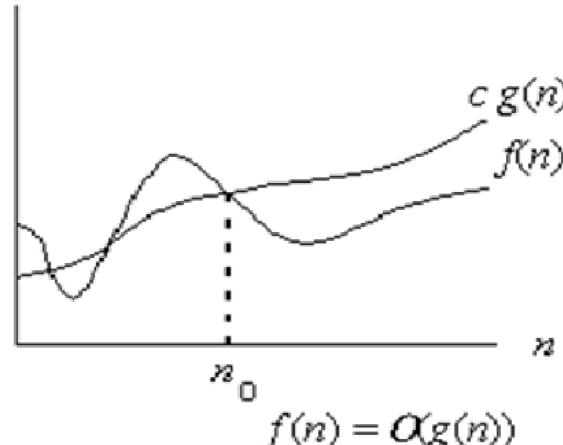


## Big-oh(O) Notation [study material]

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

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

- We use O-notation to give an asymptotic upper bound of a function, to within a constant factor.
- $f(n)=O(g(n))$  means that there exists some constant  $c$  s.t.  $f(n)$  is always  $\leq cg(n)$  for large enough  $n$ .
- It represent the upper bound of algorithm running time.



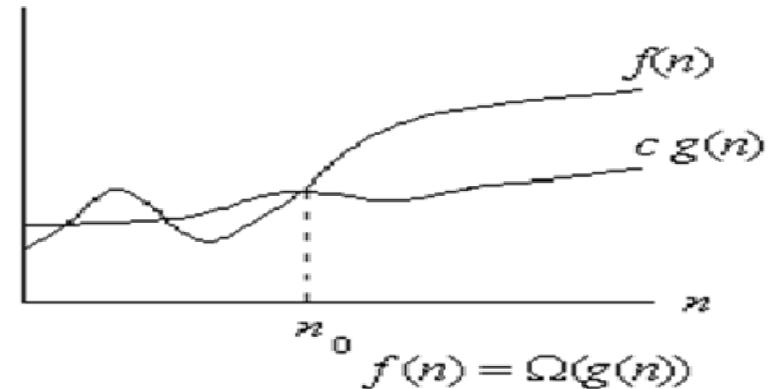
## Omega Notation ( $\Omega$ ) [study material]

---

- For a given function  $g(n)$ , we denote by  $\Omega(g(n))$  the set of functions

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

- We use  $\Omega$ -notation to give an asymptotic lower bound on a function, to within a constant factor.
- $f(n) = \Omega(g(n))$  means that there exists some constant  $c$  s.t.  $f(n)$  is always  $\geq cg(n)$  for large enough  $n$ .
- It represent the lower bound of algorithm running time

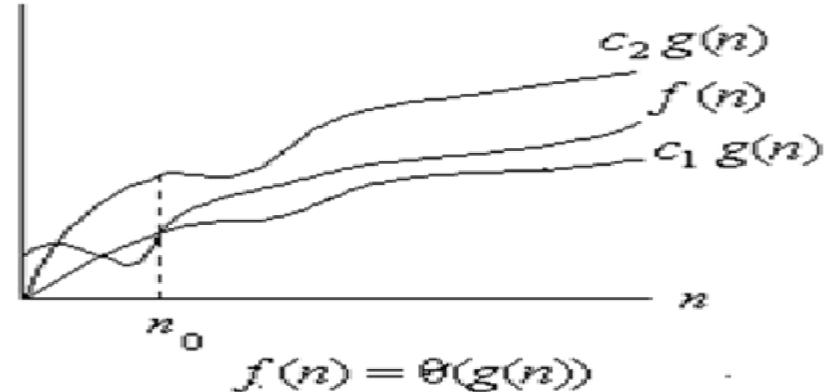


## Theta Notation ( $\Theta$ ) [study material]

- For a given function  $g(n)$ , we denote by  $\Theta(g(n))$  the set of functions

$$\Theta(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ s.t.} \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

- A function  $f(n)$  belongs to the set  $\Theta(g(n))$  if there exist positive constants  $c_1$  and  $c_2$  such that it can be “sandwiched” between  $c_1 g(n)$  and  $c_2 g(n)$  or sufficiently large  $n$ .
- $f(n) = \Theta(g(n))$  means that there exists some constant  $c_1$  and  $c_2$  s.t  $c_1 g(n) \leq f(n) \leq c_2 g(n)$  for large enough  $n$ .
- It represents running time between upper bound and lower bound of algorithm.



## Example

Lets suppose system take

- 1 unit time for arithmetic and logical operations
- 1 unit time for assignment and return statements

1. Sum of 2 numbers :

```
Sum(a,b){
```

```
return a+b //Takes 2 unit of time(constant) one for arithmetic operation and one for  
return. cost=2 no of times=1
```

```
}
```

Tsum= 2 = C =**O(1)**

2. Sum of all elements of a list :

```
list_Sum(A,n)
```

```
{ //A->array and n->number of elements in the array
```

```
total =0 // cost=1 no of times=1
```

```
for i=0 to n-1 // cost=2 no of times=n+1 (+1 for the end false condition)
```

```
sum = sum + A[i] // cost=2 no of times=n
```

```
return sum // cost=1 no of times=1
```

```
}
```

Tsum=1 + 2 \* (n+1) + 2 \* n + 1 = 4n + 1 =C1 \* n + C2 = **O(n)**

## Best case, Worst case and Average case Analyses

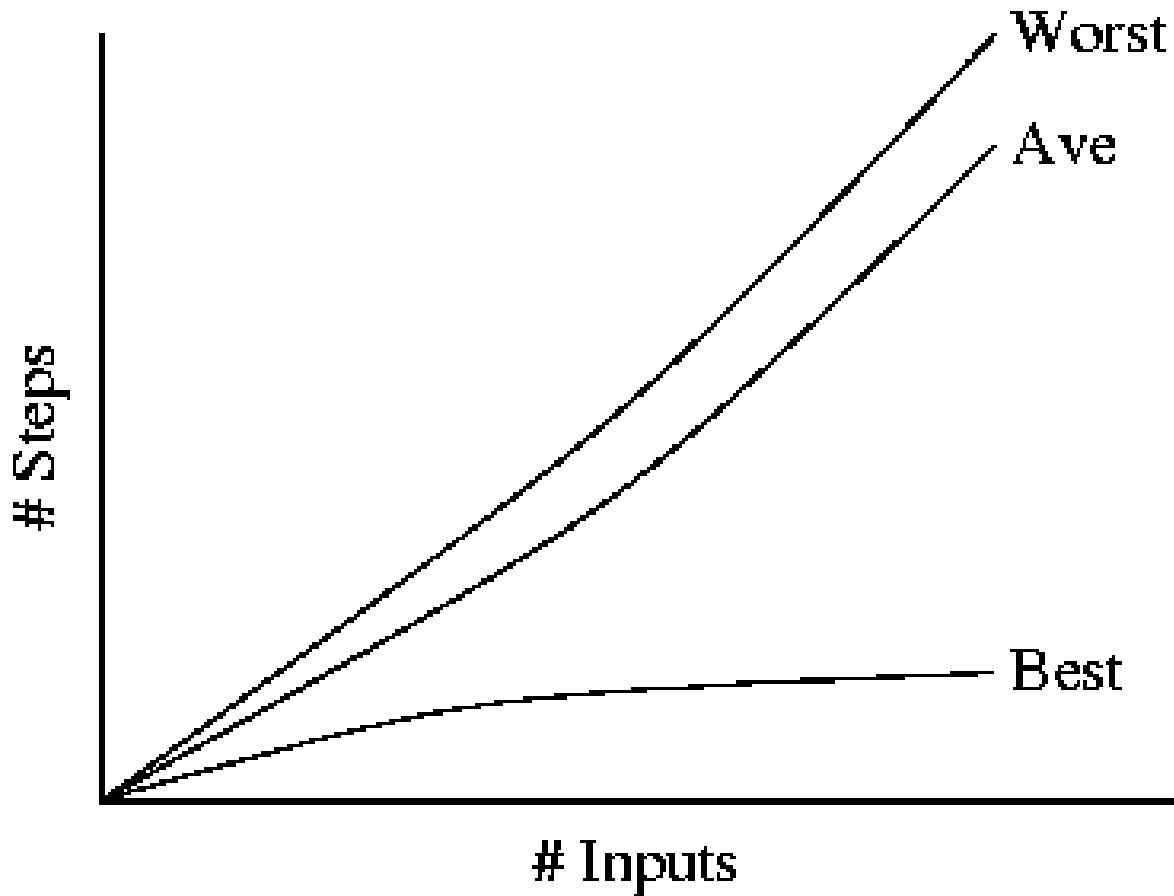
---

- The worst case
  - running time  $T(n)$  of an algorithm is the function defined by the maximum number of steps taken on any instance of size  $n$ .
- The best case
  - running time  $T(n)$  of an algorithm is the function defined by the minimum number of steps taken on any instance of size  $n$ .
- The average-case
  - running time  $T(n)$  of an algorithm is the function defined by an average number of steps taken on any instance of size  $n$ .



## Best, Worst, and Average Case

---



## Order of growth

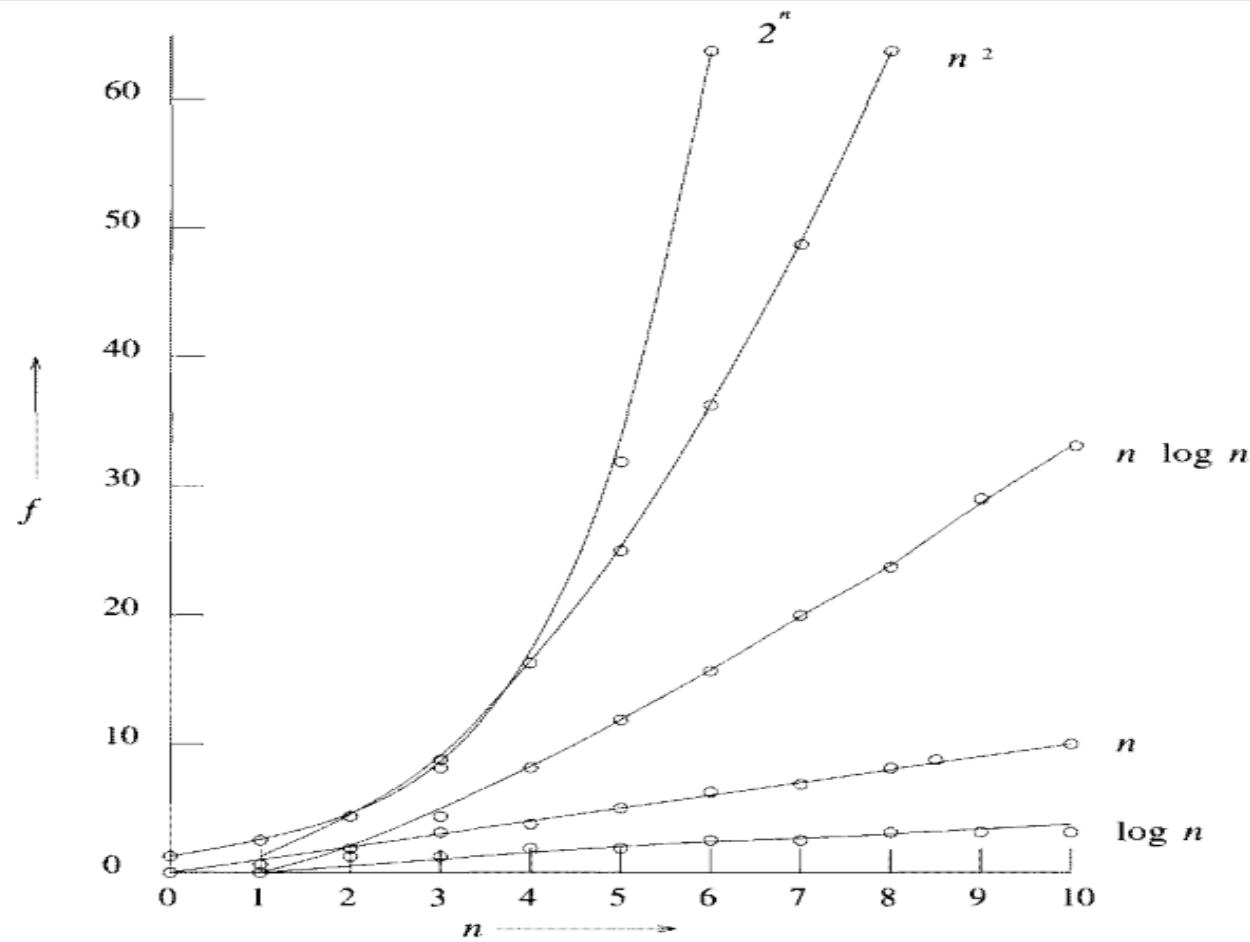
---

- Measuring the performance of an algorithm in relation with input size  $n$  is called order of growth.

$\log n$	$n$	$n \log n$	$n^2$	$n^3$	$2^n$
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4,096	65,536
5	32	160	1,024	32,768	4,294,967,296



## Order of growth



## Activity

---

# Find the Complexity Quiz



Open Link [www.kahoot.it](http://wwwkahootit)



## Answer

---

1.  $3n+2=O(n)$
2.  $5n^5 + 100n + 6=O(n^5)$
3.  $10n^2+4n+2=O(n^2)$
4.  $2n^2\log(n)+ 5n\log^2(n) =O(n^2\log(n))$
5.  $\log(\log(n))+\log(n) =O(\log(n))$

**Order to follow:  $\log(\log(n)), \log(n), (\log(n))^2,$**

**$n^{1/3}, n^{1/2}, n, n\log(n), n^2/\log(n), n^2, n^3, 2^n$**



## Example

---

Consider the following algorithm

Algorithm Seq\_search ( x[0,1,...n-1] ,key)

{

//problem description: This algorithm is for searching the key element from an  
//array x[0,1,...n-1] sequentially

//Input: an array X[0...n-1] and search key

//output: return the index of x where key value is present.

for i=0 to n-1 do

  if(x[i]==key) then

    return i

}

## Activity

---

# What is the Best Case Complexity?



## Best case

---

if key element is present at first location in the list  $x[0,1,\dots,n-1]$  the algorithm run for a very short time and there by we will get the best case time complexity.

$$C_{\text{best}} = 1 = O(1)$$



## Activity

---

# What is the Worst Case Complexity?



## Worst case

---

if key is not present in the array  
 $C_{\text{worst}} = n+1 = O(n)$

## Activity

---

# What is the Average Case Complexity?



## Average Case

---

Let the algorithm is for sequential search and P be a probability of getting successful search n is the total number of elements in the list.

- The first match of the element will occur at ith location. Hence probability of occurring first match is  $P/n$  for every ith element.
- The probability of getting unsuccessful search is  $(1-P)$

Now we can find average case

**Cavg(n)=Probability of successful search + Probability of unsuccessful search**

$$\begin{aligned}\text{Cavg} &= [1.P/n + 2.P/n + 3.P/n + \dots + i.P/n] + n(1-P) \\ &= P/n [1+2+3+\dots+i+\dots+n] + n(1-P) \\ &= P(n+1)/2 + n(1-P)\end{aligned}$$

If  $P=0$  means no successful search

$\text{Cavg}=n \rightarrow$  worst case

If  $P=1$  we get a successful search

$\text{Cavg}=(n+1)/2$

Module No 1: Introduction

---

# Lecture No:3

# Analysis of Selection Sort



# Selection Sort

---

Idea:

Find the smallest element in the array

Exchange it with the element in the first position

Find the second smallest element and exchange it with the element in the second position

Continue until the array is sorted

Disadvantage:

Running time depends only slightly on the amount of order in the file



# Example

---

8	4	6	9	2	3	1
---	---	---	---	---	---	---

1	4	6	9	2	3	8
---	---	---	---	---	---	---

1	2	6	9	4	3	8
---	---	---	---	---	---	---

1	2	3	9	4	6	8
---	---	---	---	---	---	---

1	2	3	4	9	6	8
---	---	---	---	---	---	---

1	2	3	4	6	9	8
---	---	---	---	---	---	---

1	2	3	4	6	8	9
---	---	---	---	---	---	---

1	2	3	4	6	8	9
---	---	---	---	---	---	---



# Selection Sort

---

Algorithm SELECTION-SORT( $A[0,1\dots n-1]$ )

{//problem description: This algorithm sort the element using selection sort

//Input: an array of element  $a[0,1\dots n-1]$  that is to be stored

//output: the sorted array  $A[0,1,\dots,n-1]$

```
for i ← 0 to n – 2
do smallest ← i
    for j ← i + 1 to n-1
        do if A[j] < A[smallest]
            then smallest ← j
    //swap A [i] and A[smallest]
    temp=A[i]
    A[i]=A[smallest]
    A[smallest]=temp
}
```



# Analysis of Selection Sort

SELECTION-SORT( $A[0,1,\dots,n-1]$ )

	cost	times
<b>for</b> $i \leftarrow 0$ <b>to</b> $n - 2$	$c_1$	$n$
<b>do</b> $\text{smallest} \leftarrow i$	$c_2$	$n-1$
<b>for</b> $j \leftarrow i + 1$ <b>to</b> $n-1$	$c_3$	$\sum_{i=0}^{n-2} (n-i)$
<b>do if</b> $A[j] < A[\text{smallest}]$	$c_4$	$\sum_{i=0}^{n-2} (n-i-1)$
<b>then</b> $\text{smallest} \leftarrow j$	$c_5$	$\sum_{i=0}^{n-2} (n-i-1)$
//swap $A[i]$ and $A[\text{smallest}]$		
$\text{temp} = A[i]$	$c_6$	$n-1$
$A[i] = A[\text{smallest}]$	$c_7$	$n-1$
$A[\text{smallest}] = \text{temp}$	$c_8$	$n-1$

$$T(n) = c_1 n + c_2 (n-1) + c_3 \sum_{i=0}^{n-2} (n-i) + (c_4 + c_5) \sum_{i=0}^{n-2} (n-i-1) + (c_6 + c_7 + c_8) n - 1$$
$$= O(n^2)$$

# Analysis of Selection Sort

---

Step 1: the Input size is n

Step 2: Basic Operation : if  $A[j] < A[\text{smallest}]$

Step 3: The comparison is executed on each repletion of the loop

$$c(n) = \sum_{i=0}^{n-2} (n - i - 1)$$

Step 4: Calculation of complexity

$$\begin{aligned} c(n) &= \sum_{i=0}^{n-2} (n - i - 1) \\ &= (n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i \\ &= (n-1)(n-2-0+1) - ((n-2)(n-2+1)/2) \\ &= n(n-1)/2 \\ &= O(n^2) \end{aligned}$$

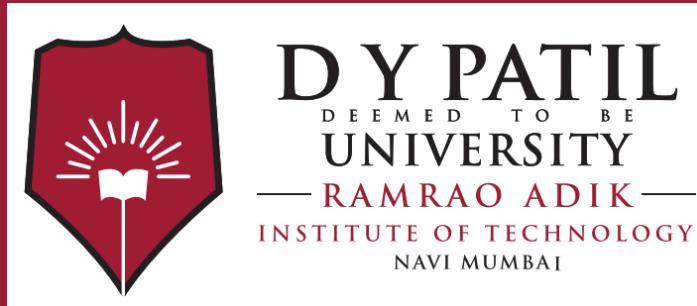
## Activity

---

# What is the Best and Worst Case Complexity?



# Thank You



**Subject Name: Design and Analysis of  
Algorithms**

**Module No: 1**

**Module Name: Introduction**

# Outline-Module 1

---

Sr. No	Topic
1	Performance analysis , space and time complexity Mathematical background for algorithm analysis
2	Growth of function – Big –Oh ,Omega, Theta notation
3	Analysis of Selection Sort
4	Analysis of Insertion Sort
5	The substitution method and Recursion tree method
6	Master Theorem

# **Lecture No: 4**

## **Analysis of Insertion Sort**



# Insertion Sort

---

Idea:

Like sorting a hand of playing cards

Start with an empty left hand and the cards facing down on the table.

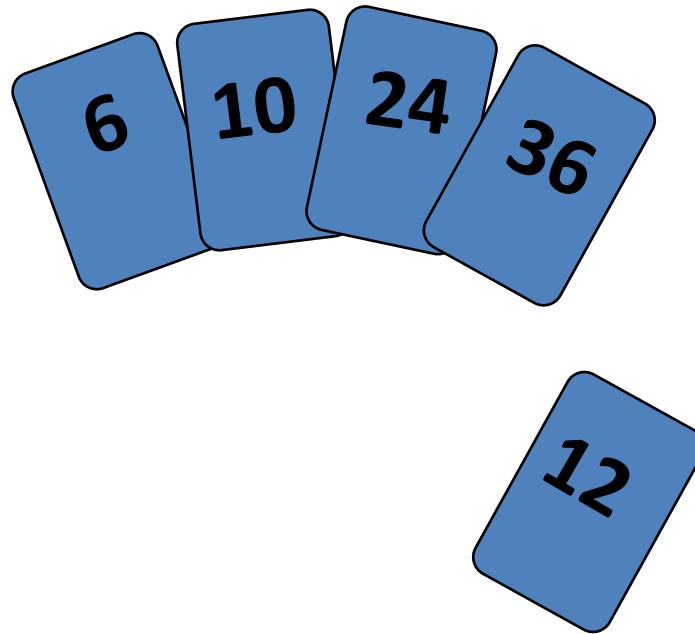
Remove one card at a time from the table, and insert it into the correct position in the left hand.

Compare it with each of the cards already in the hand, from right to left

The cards held in the left hand are sorted these cards were originally the top cards of the pile on the table

# Insertion Sort

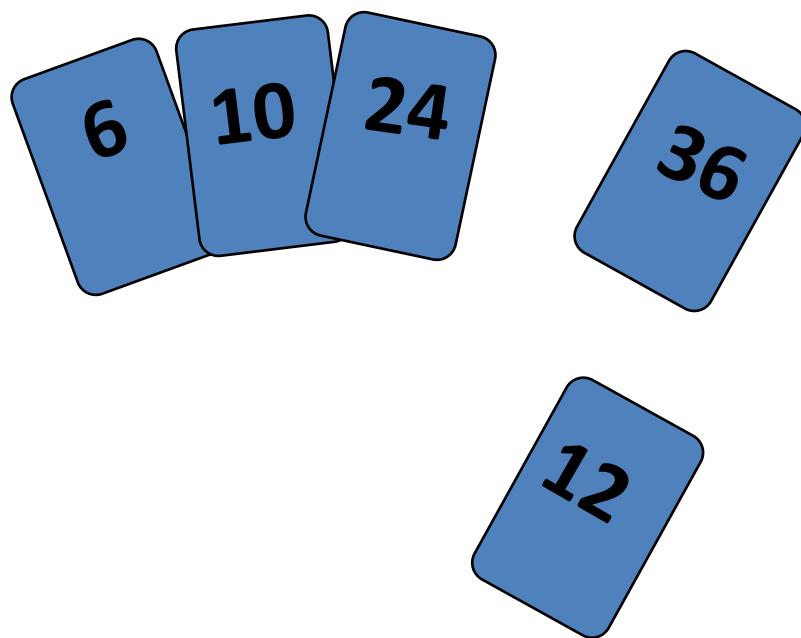
---



To insert 12, we need to make room for it by moving first 36 and then 24.

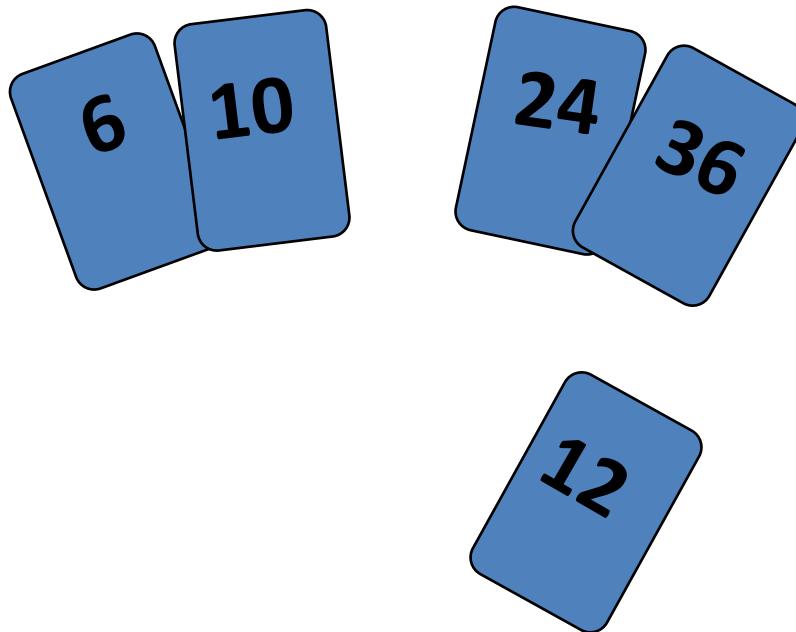
# Insertion Sort

---



# Insertion Sort

---

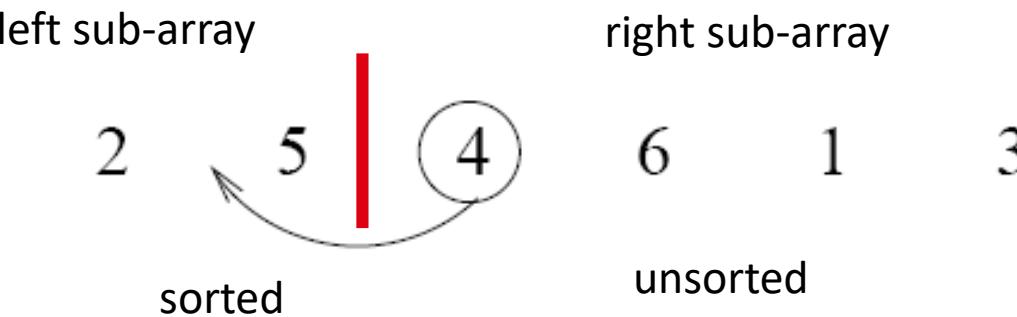


# Insertion Sort

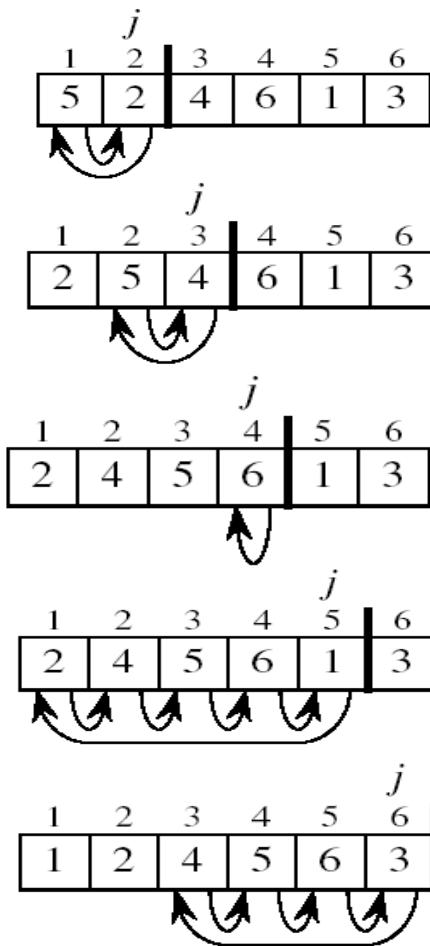
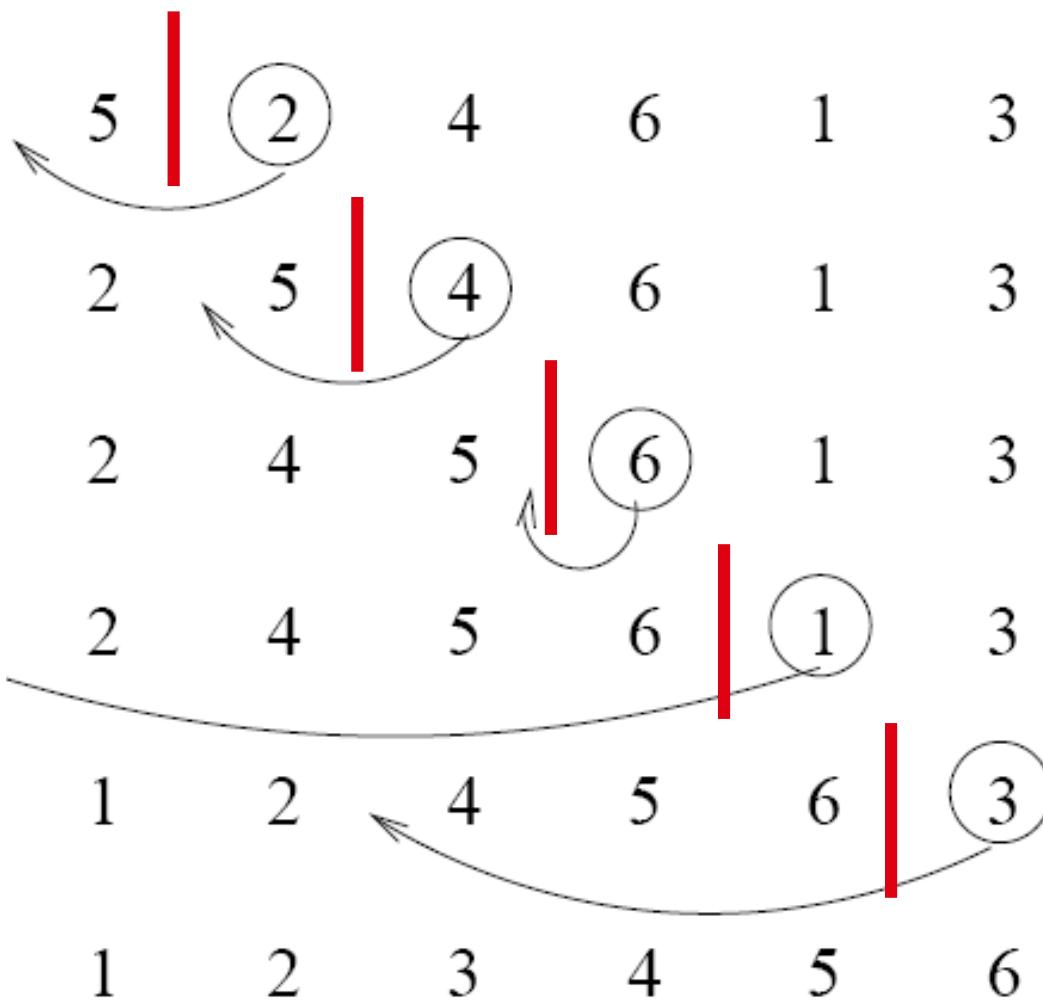
---

input array  
5    2    4    6    1    3

at each iteration, the array is divided in two sub-arrays:



# Insertion Sort



# INSERTION-SORT

Algorithm INSERTION-SORT( $A$ )

//problem description: This algorithm sort the element using Insertion sort

//Input: an array of element  $a[0,1\dots n-1]$  that is to be stored

//output: the sorted array  $A[0,1,\dots,n-1]$

**for**  $j \leftarrow 2$  **to**  $n$

**do**  $key \leftarrow A[j]$

// Insert  $A[j]$  into the sorted sequence  $A[1 \dots j - 1]$

$i \leftarrow j - 1$

**while**  $i > 0$  and  $A[i] > key$

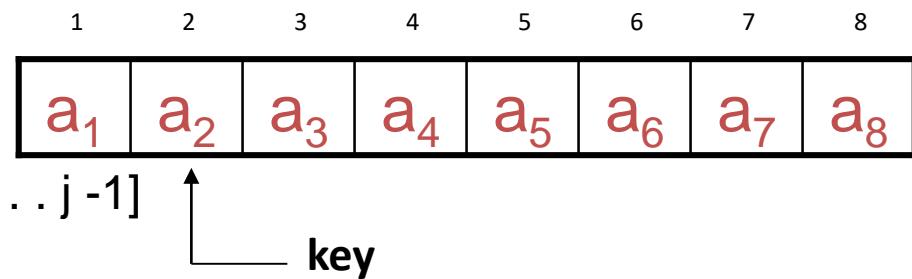
**do**  $A[i + 1] \leftarrow A[i]$

$i \leftarrow i - 1$

$A[i + 1] \leftarrow key$

}

Insertion sort – sorts the elements in place



# Analysis of Insertion Sort

INSERTION-SORT( $A$ )

**for**  $j \leftarrow 2$  **to**  $n$

**do**  $\text{key} \leftarrow A[j]$

$i \leftarrow j - 1$

**while**  $i > 0$  and  $A[i] > \text{key}$

**do**  $A[i + 1] \leftarrow A[i]$

$i \leftarrow i - 1$

$A[i + 1] \leftarrow \text{key}$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_7(n-1)$$



D Y PATIL  
DEEMED TO BE  
UNIVERSITY  
— RAMRAO ADIK —  
INSTITUTE OF TECHNOLOGY  
NAVI MUMBAI

# Analysis of Insertion Sort

---

Step 1: the Input size is n

Step 2: Basic Operation : **while**  $i > 0$  and  $A[i] > \text{key}$

Step 3: Basic operation complexity depends on input and worst case

$$c(n) = \sum_{j=2}^n \sum_{i=0}^{j-1} 1$$

Step 4: Calculation of complexity

$$c(n) = \sum_{j=2}^n \sum_{i=0}^{j-1} 1$$

$$= \sum_{j=2}^n j$$

$$= (n-1)(n)/2$$

$$= O(n^2)$$

# Lecture No: 5

## The substitution method and Recursion tree method



## Recurrence relation

---

# What is Recurrence?

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

A recurrence relation is an equation that defines a sequence based on a rule that gives the next term as a function of the previous term(s). It is also defined as a function by means of an expression that includes one or more smaller instances of itself.

Example for the recursive definition is factorial function , Fibonacci series.



D Y PATIL  
DEEMED TO BE  
UNIVERSITY  
— RAMRAO ADIK —  
INSTITUTE OF TECHNOLOGY  
NAVI MUMBAI

# Recurrence

---

<https://www.coursera.org/learn/simulation-algorithm-analysis-pointers/lecture/oyhDs/recursion>

## Example of Recursive Program

---

Recursive program to compute  $n!$

$$n! = \begin{cases} 1 & \text{if } n=0 \text{ or } n=1 \\ n \times (n-1)! & n>1 \end{cases}$$

Factorial(int n)

{ if( $n==0$  ||  $n==1$ )

    return 1;

    else

        return  $n * \text{Factorial}(n-1)$

}

$$T(n) = \begin{cases} 1 & \text{if } n=0 \text{ or } n=1 \\ T(n-1)+1 & n>1 \end{cases}$$

$$T(1) = 1$$

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

$$T(3) = T(2) + 1 = 3$$

.

.

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

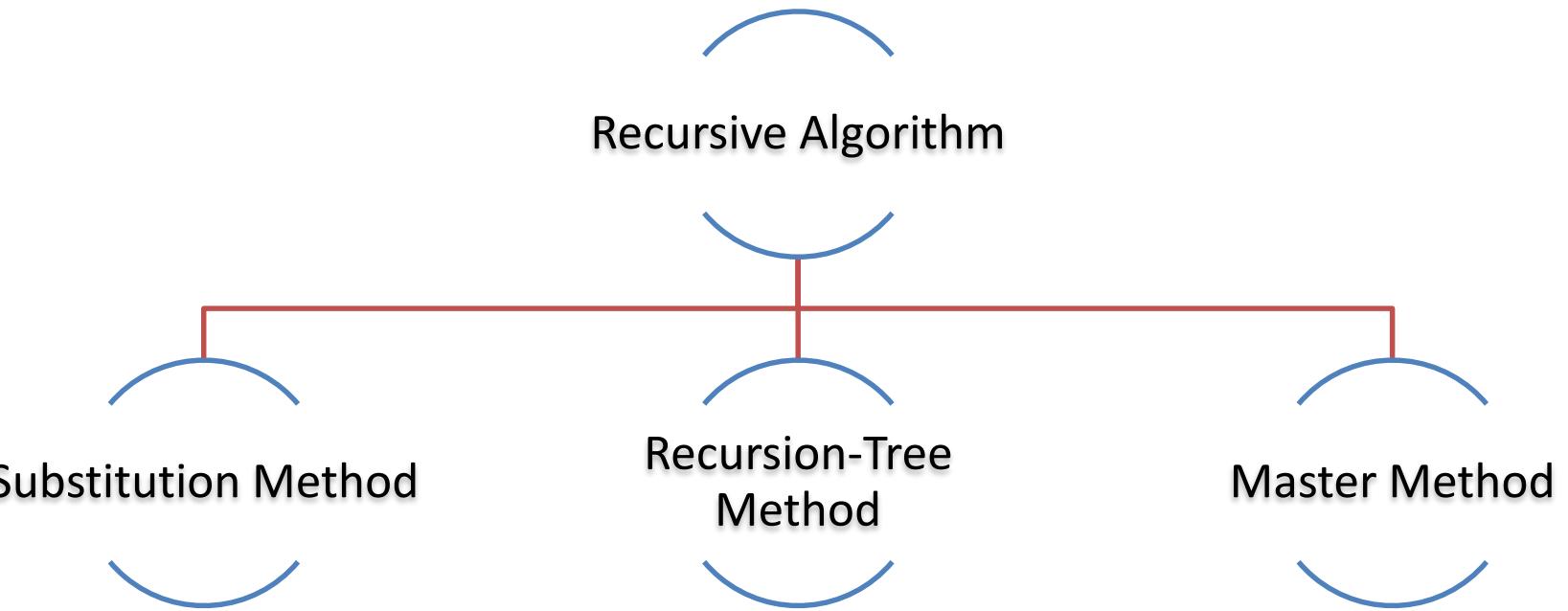
---

# FINDING COMPLEXITY OF RECURSIVE ALGORITHMS



# Finding Complexity of recursive algorithms

---



# Solving Recurrence Relations

Special Cases of Geometric Series and Arithmetic series:

$$\sum_{k=1}^n k = 1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

$$\sum_{k=1}^n \lg k \approx n \lg n$$

$$\sum_{k=0}^n x^k = 1 + x + x^2 + \dots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

$$\sum_{k=0}^{n-1} c = cn.$$

$$\sum_{k=0}^{n-1} x^k = \frac{x^n - 1}{x - 1} (x \neq 1)$$

$$\sum_{k=0}^{n-1} \frac{1}{2^k} = 2 - \frac{1}{2^{n-1}}$$

$$1 + x + x^2 + \dots = \frac{1}{1-x} \quad \text{for } |x| < 1$$

$$\sum_{k=0}^{n-1} 2^k = 2^n - 1$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

if  $x < 1$

L



## Substitution Method

---

Merge Sort  $T(n) = T(n/2) + T(n/2) + Cn$  for  $n > 1$  and  $T(1) = 1$

$$T(n) = 2T(n/2) + n \quad \text{----- Equation 1}$$

For  $n=n/2$

$$T(n/2) = 2 [T((n/2)/2) + (n/2)] \quad \text{----- Put this in Equation 1}$$

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

$$T(n) = 4T(n/4) + 2n \quad \text{----- Equation 2}$$

For  $n=n/4$

$$T(n/4) = 2 [T((n/4)/2) + (n/4)] \quad \text{----- Put this in Equation 2}$$

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

$$T(n) = 8T(n/8) + 3n \quad \text{----- Equation 3}$$

General Equation at  $n=k$   $T(n) = 2^k T(n/2^k) + kn$

## Substitution Method

---

General Equation at  $n=k$   $T(n)=2^kT(n/2^k)+kn$

The base case is reached when  $n / 2^k = 1 \rightarrow n = 2^k \rightarrow k = \log_2 n$ , we then have:

$$T(n)=2^kT(n/2^k)+kn \Rightarrow T(n)=nT(1)+\log_2 n * n \Rightarrow T(n)=n+\log_2 n * n$$

Complexity =  $O(n\log_2 n)$

## Activity

---

# What is the Complexity of Equation?

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

With initial condition  $T(0) = 0$



## Example

Consider a recurrence relation

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

With initial condition  $T(0) = 0$

Let,

$$T(n) = T(n-1) + 1 \quad \text{----- equation 1}$$

$$T(n-1) = T(n-2) + 1 \quad \text{----- putting this in equation 1 we get}$$

$$T(n) = T(n-2) + 1 + 1 = T(n-2) + 2 \quad \text{----- equation 2}$$

$$T(n-2) = T(n-3) + 1 \quad \text{----- putting this in equation 2 we get}$$

$$T(n) = T(n-3) + 1 + 2 = T(n-3) + 3 \quad \text{----- equation 3}$$

repeat up to k steps

$$T(n) = T(n-k) + k \quad \text{----- equation k}$$

The base case is reached when  $n - k = 0 \rightarrow k = n$ , we then have :

$$T(n) = T(0) + n$$

$$T(n) = 0 + n = n$$

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

## Activity

---

# What is the Complexity of Equation?

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

With initial condition  $T(0) = 0$



## Example

Consider a recurrence relation

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

With initial condition  $T(0) = 0$

Let,

$$T(n) = T(n-1) + n \quad \text{----- equation 1}$$

$T(n-1) = T(n-2) + n-1$  ----- putting this in equation 1 we get

$$T(n) = T(n-2) + (n-1) + n \quad \text{----- equation 2}$$

$T(n-2) = T(n-3) + n-2$  ----- putting this in equation 2 we get

$$T(n) = T(n-3) + (n-2) + (n-1) + n \quad \text{----- equation 3}$$

repeat up to k steps

$$T(n) = T(n-k) + \dots + (n-2) + (n-1) + n \quad \text{----- equation k}$$

The base case is reached when  $n - k = 0 \rightarrow k = n$ , we then have :

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

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

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

## Home Activity

---

# What is the Complexity of Equation?

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



## Activity

---

# What is the Complexity of Equation?

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



## Example

## Consider a recurrence relation

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

If initial condition is not given consider  $T(1)=1$

Let,

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

----- equation 1

$$T(n/2) = T(n/4) + 1$$

-----putting this in equation 1 we get

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

----- equation 2

$$T(n/4) = T(n/8) + 1$$

-----putting this in equation 2

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

----- equation 3

- repeat up to k steps

■

$$T(n) = T(n/2^k) + k \quad \text{----- equation k}$$

The base case is reached when  $n / 2^k = 1 \rightarrow n = 2^k \rightarrow k = \log_2 n$ , we then have:

Put the value of k in general equation

$$T(n) = T(1) + \log_2 n$$

$$T(n) = 1 + \log_2 n$$

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

## Activity

---

# What is the Complexity of Equation?

$$T(n) = 2T(n/2 + 17) + n$$



## Example

Consider a recurrence relation

$$T(n)=2T(n/2 +17)+n$$

If initial condition is not given consider  $T(1)=1$

Let,

$$T(n)=2T(n/2 +17)+n$$

----- equation 1

$$T(n/2 +17)=2T((n/2 +17)/2 +17)+(n/2 +17)$$

----- putting this in equation 1 we get

$$=2T(n/4 +25.5) +n/2 +17$$

$$T(n)=2[2T(n/4 +25.5) +n/2 +17 ]+n$$

----- equation 2

$$= 2^2 T(n/4 +25.5)+3n+34$$

$$T(n/4 +25.5)=2T((n/4 +25.5)/2 +17)+(n/2 +25.5)$$

----- putting this in equation 2 we get

$$= 2T(n/8+29.75)+ n/2 +25.5$$

$$T(n)= 2^2 [2T(n/8+29.75)+ n/2 +25.5 ]+3n+34$$

----- equation 3

$$= 2^3 T(n/8+29.75)+5n+136$$

----- equation k

$$\text{T}(n)= 2^k T(n/2^k +c1)+(2k+1)n+c2$$

$T(1)=1$  is compare the term  $T(n/2^k)=T(1)$

If we compare the term  $T(n/2^k)=T(1)$

$n/2^k=1 \Rightarrow n=2^k \Rightarrow$  then  $k= \log_2 n$

Put the value of k in general equation

$$T(n)= nT(1)+(2\log_2 n +1)n+c2 \quad T(n)=2n+ 2n\log_2 n+c2$$

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

## Activity

---

# What is the Complexity of Equation?

$$T(n) = 2T(\sqrt{n}/2) + \log_2 n$$



## Example

$$T(n) = 2T(\sqrt{n}/2) + \log_2 n$$

Let,  $m = \log_2 n \rightarrow n = 2^m$

$$T(2^m) = 2T(2^{m/2}) + m \quad \text{----- equation 1}$$

Let,  $T(2^m) = S(m)$

$$S(m) = 2S(m/2)$$

-----putting this in equation 1 we get ----- equation 2

$$S(m/2) = 2S(m/4) + m/2$$

$$S(m) = 2[2S(m/4) + m/2] + m$$

-----putting this in equation 2 we get

$$S(m) = 2^2 S(m/2^2) + 2m$$

----- equation 3

$$S(m/4) = 2S(m/8) + m/4$$

-----putting this in equation 2 we get

$$S(m) = 2^2 [2S(m/8) + m/4] + 2m$$

$$S(m) = 2^3 S(m/2^3) + 3m \quad \text{----- equation 3}$$

$$S(m) = 2^k S(m/2^k) + km \quad \text{----- equation k}$$

$S(1) = 1$  is compare the term  $S(m/2^k) = S(1)$

$m/2^k = 1 \Rightarrow m = 2^k \Rightarrow$  then  $k = \log_2 m$

Put the value of k in general equation

$$S(m) = mS(1) + m\log_2 m = m + m\log_2 m$$

now replace  $S(m)$  with  $T(2^m)$

$$T(2^m) = m + m\log_2 m$$

now replace  $m = \log_2 n$  and  $2^m = n$

$$T(n) = \log_2 n + \log_2 n * \log_2 \log_2 n = O(\log_2 n * \log_2 \log_2 n)$$

## Activity

---

# What is the Complexity of Equation?

$$T(n) = T(n-1) + n^4$$



## Example

Consider a recurrence relation

$$T(n) = T(n-1) + n^4$$

If initial condition is not given consider  $T(1)=1$

Let,

$$T(n) = T(n-1) + n^4 \quad \text{----- equation 1}$$

$$T(n-1) = T(n-2) + (n-1)^4 \quad \text{----- putting this in equation 1 we get}$$

$$T(n) = T(n-2) + (n-1)^4 + n^4 \quad \text{----- equation 2}$$

$$T(n-2) = T(n-3) + (n-2)^4 \quad \text{----- putting this in equation 2 we get}$$

$$T(n) = T(n-3) + (n-2)^4 + (n-1)^4 + n^4 \quad \text{----- equation 3}$$

$$T(n-k) = T(n-k-1) + (n-k)^4 \quad \text{----- putting this in equation k we get}$$

$$T(n) = T(n-k-1) + (n-k)^4 + \dots + (n-2)^4 + (n-1)^4 + n^4 \quad \text{----- equation k}$$

$$= 1^4 + 2^4 + 3^4 + \dots + (n-2)^4 + (n-1)^4 + n^4$$

$$= \sum_{n=1}^n n^4 = n^{4+1}/(4+1)$$

$$= n^5/5$$

$$= \mathbf{O(n^5)}$$

$$\sum_{n=1}^n n^k = n^{k+1}/(k+1)$$

## Activity

---

# What is the Complexity of Equation?

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



## Example

Consider a recurrence relation

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

If initial condition is not given consider  $T(1)=1$

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

----- equation 1

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

----- putting this in equation 1 we get

$$\begin{aligned} T(n) &= 4[4T(n/3^2) + (n/3)^2] + n^2 \\ &= 4^2 T(n/3^2) + 4(n/3)^2 + n^2 \end{aligned}$$

----- equation 2

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

----- putting this in equation 2 we get

$$\begin{aligned} T(n) &= 4^2 [4T(n/3^3) + (n/3^2)^2] + (n/3)^2 + n^2 \\ &= 4^3 T(n/3^3) + 4^2 (n/3^2)^2 + 4(n/3)^2 + n^2 \end{aligned}$$

----- equation 3

$$T(n/3^k) = 4^{k-1} T(n/3^k) + (n/3^k)^2$$

----- putting this in equation k we get

$$\begin{aligned} T(n) &= 4^k T(n/3^k) + 4^{k-1} (n/3^k)^2 \dots \dots + 4^2 (n/3^2)^2 + 4(n/3)^2 + n^2 \quad \text{----- equation k} \\ &= 4^k T(n/3^k) + n^2 [4^{k-1} (1/3^k)^2 \dots \dots + 4^2 (1/3^2)^2 + 4(1/3)^2 + 1^2] \end{aligned}$$



### Constant C

$$= 4^k T(n/3^k) + Cn^2$$

$T(1)=1$  is compare the term  $T(n/3^k)=S(1)$

$$n/3^k=1 \Rightarrow n=3^k \Rightarrow \text{then } k=\log_3 n$$

$$= 4^{\log_3 n} T(1) + Cn^2 = n^{\log_3 4} + Cn^2 = n^{1.26} + Cn^2 = O(n^2)$$

# Recurrence Relations to Remember

---

## Recurrence

## Algorithm

## Big-Oh Solution

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

Binary Search

$$O(\log n)$$

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

Sequential Search

$$O(n)$$

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

tree traversal

$$O(n)$$

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

Selection Sort (other  $n^2$  sorts)

$$O(n^2)$$

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

Mergesort (average case Quicksort)

$$O(n \log n)$$

# Solving Recurrence Relations

---

Special Cases of Geometric Series and Arithmetic series:

$$\sum_{k=1}^n k = 1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

$$\sum_{k=1}^n \lg k \approx n \lg n$$

$$\sum_{k=0}^n x^k = 1 + x + x^2 + \dots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

$$\sum_{k=0}^{n-1} c = cn.$$

$$\sum_{k=0}^{n-1} x^k = \frac{x^n - 1}{x - 1} (x \neq 1)$$

$$\sum_{k=0}^{n-1} \frac{1}{2^k} = 2 - \frac{1}{2^{n-1}}$$

$$1 + x + x^2 + \dots = \frac{1}{1-x} \quad \text{for } |x| < 1$$

$$\sum_{k=0}^{n-1} 2^k = 2^n - 1$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \quad \text{if } x < 1$$

## The recursion-tree method

---

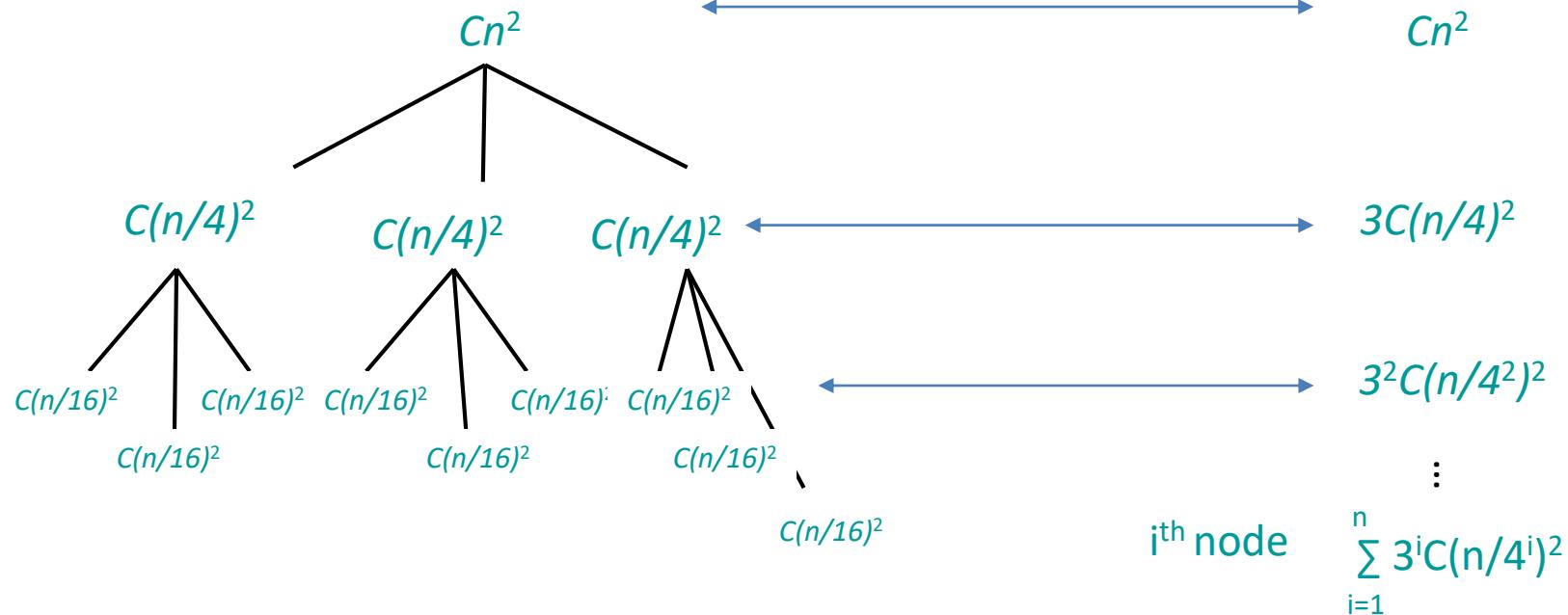
- ❖ Convert the recurrence into a tree:
  - Each node represents the cost incurred at various levels of recursion
  - Sum up the costs of all levels
- ❖ Steps
  1. Cost of each level
  2. Find the depth of tree
  3. Find the number of leaves



## Example 1

Solve  $T(n) = 3T(n/4) + \Theta(n^2)$ :

*Cost Calculation*



## Example 1

---

Solve  $T(n) = 3T(n/4) + \Theta(n^2)$ :

*Depth Calculation*

$$i^{\text{th}} \text{ node} \quad \sum_{i=1}^n 3^i C(n/4^i)^2$$

$$= Cn^2 \sum_{i=1}^n (3/16)^i$$

$$= Cn^2(1/(1-(3/16)))$$

$$= 16/13 Cn^2$$

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

*Using Formula*

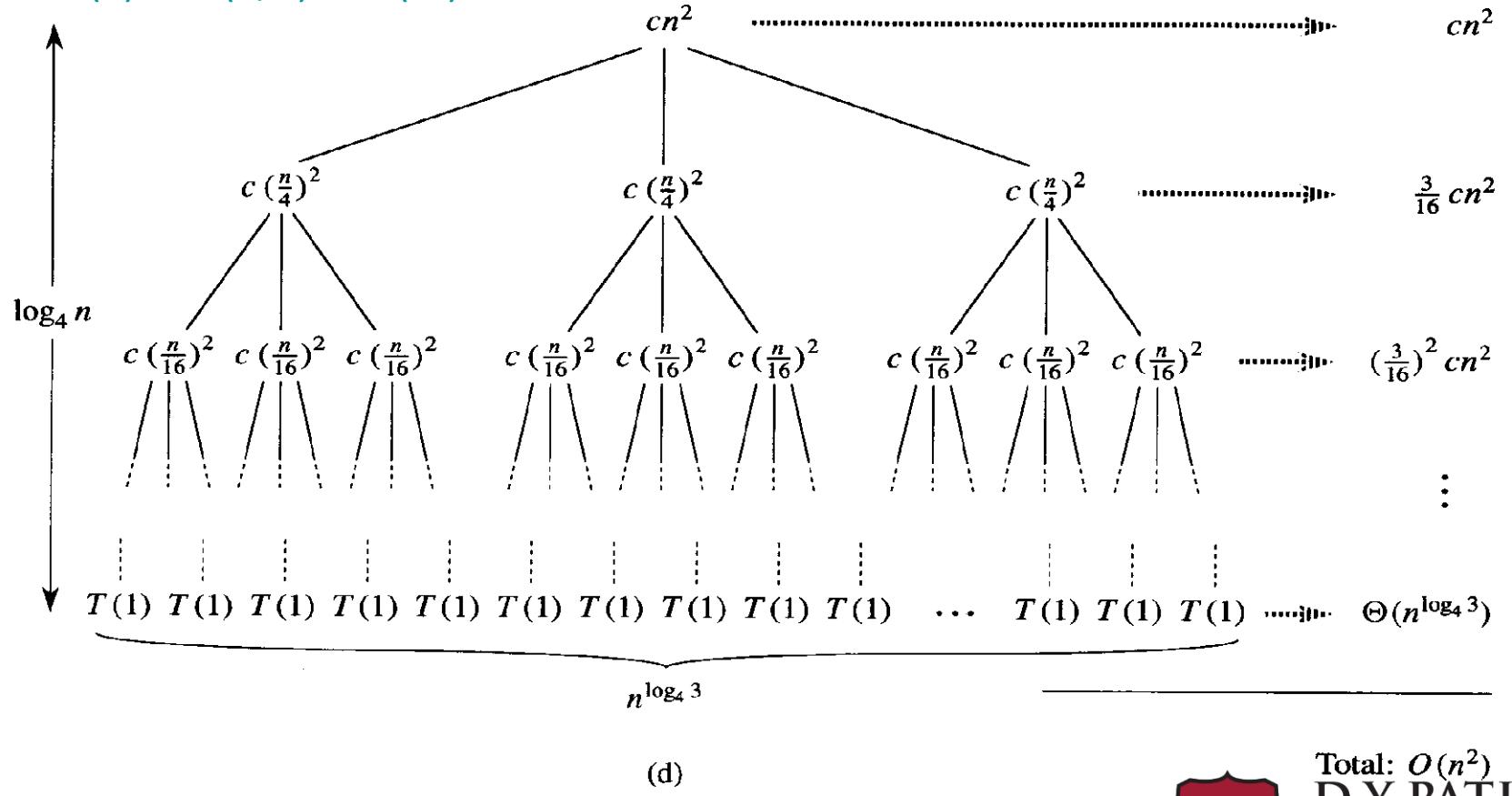
$$1 + x + x^2 + \dots = \frac{1}{1-x} \quad \text{for } |x| < 1$$

- At  $\Theta(1)=T(n/4^i)$   $n/4^i=1$   $i=\log_4 n$
- So tree has  $i=\log_4 n+1$  levels
- No of nodes at depth  $i \Rightarrow 3^i \Rightarrow 3^{\log_4 n} \Rightarrow n^{\log_4 3}$

## Example 1

---

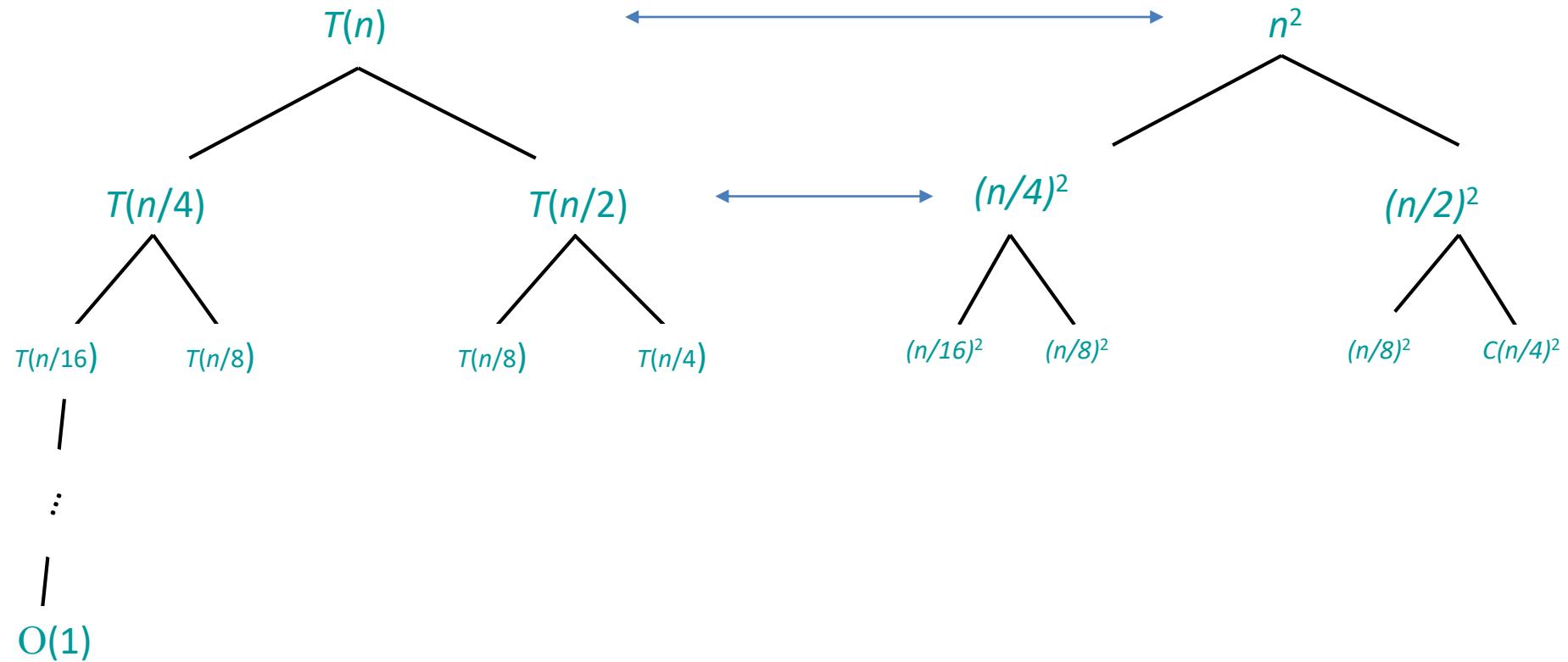
Solve  $T(n) = 3T(n/4) + \Theta(n^2)$ :



## Example 2

---

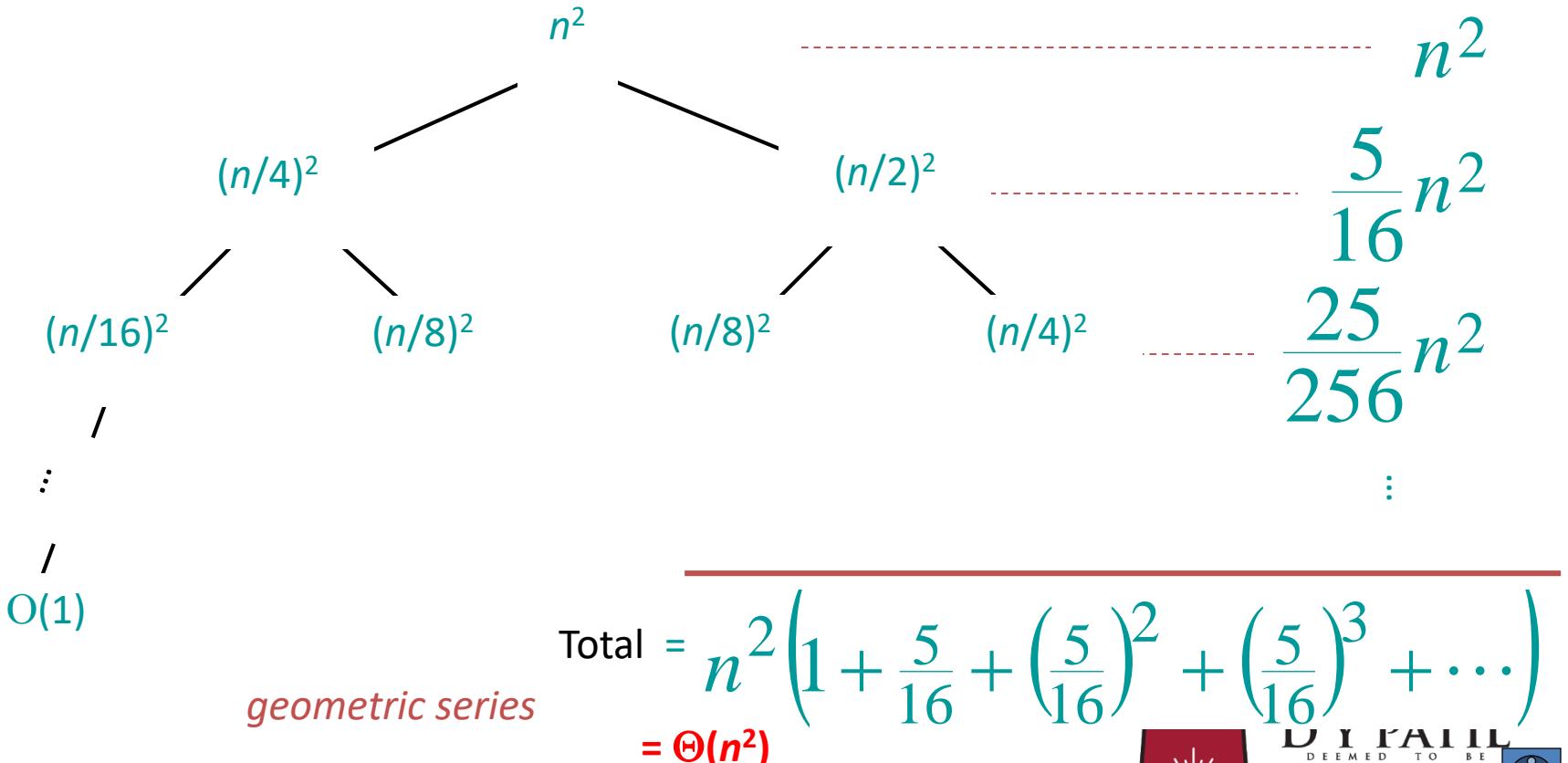
Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :



## Example 2

---

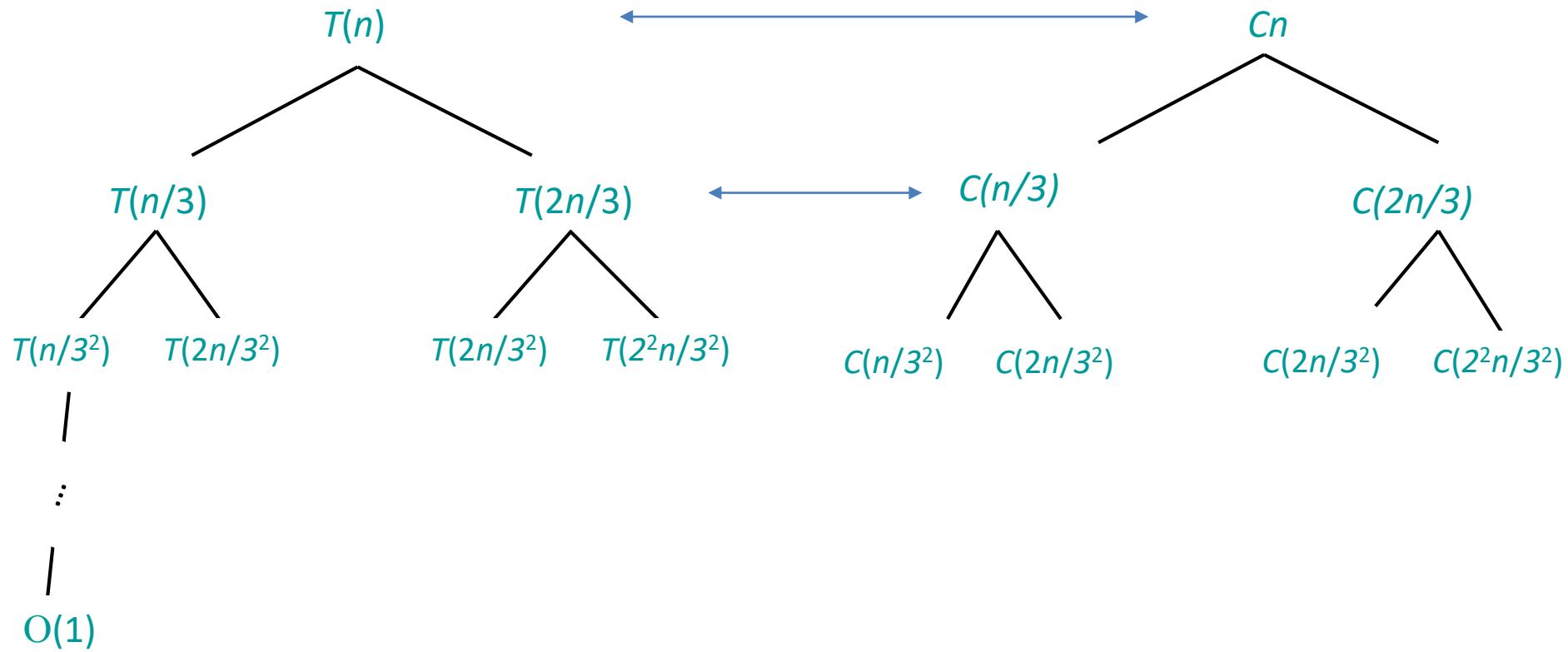
Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :



## Example 3

---

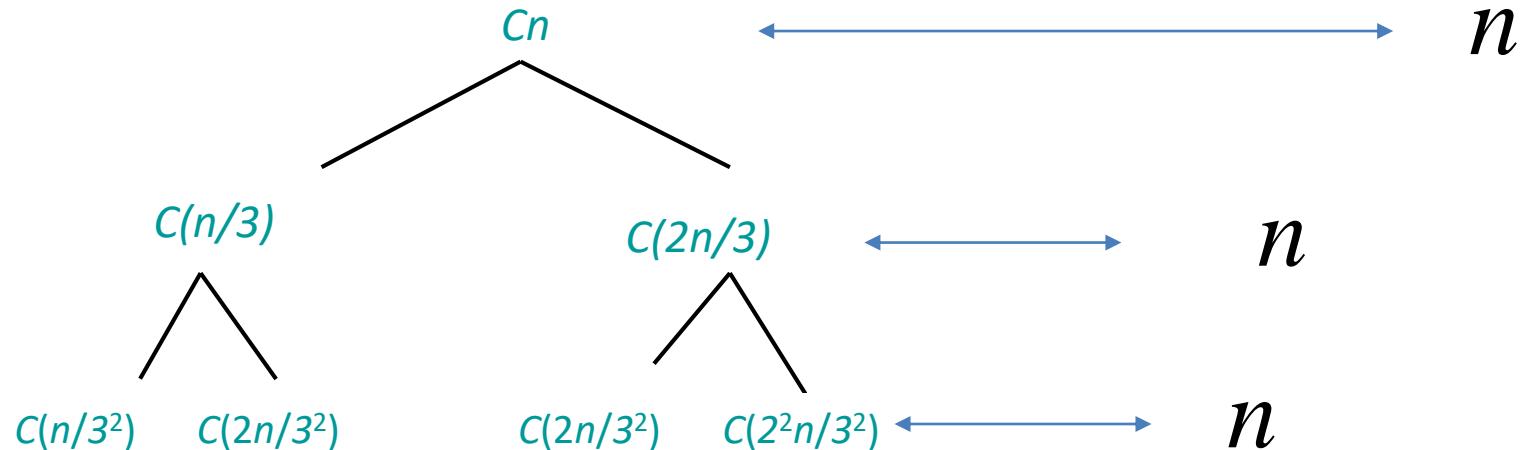
Solve  $T(n) = T(n/3) + T(2n/3) + O(n)$



## Example 3

---

Solve  $T(n) = T(n/3) + T(2n/3) + O(n)$



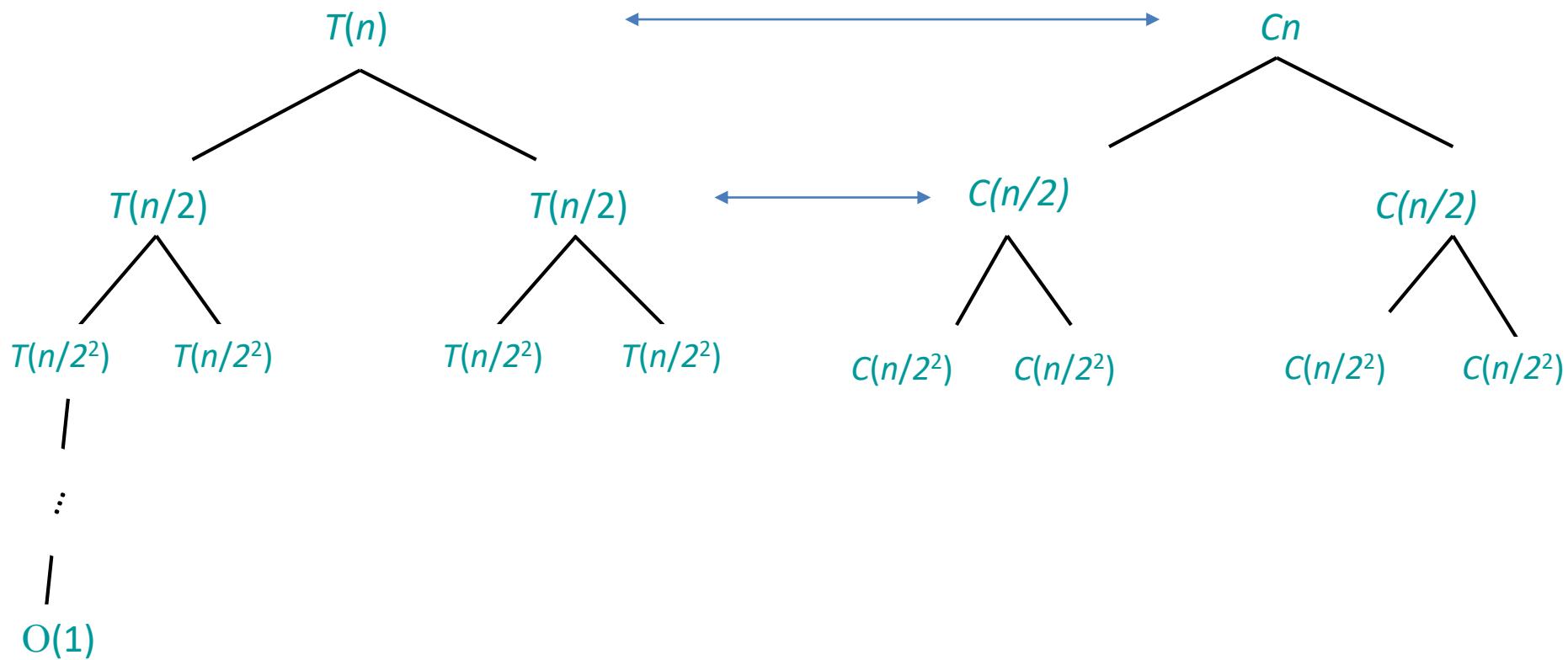
- Cost =  $n * \text{depth}$
- At  $O(1) = T(n(2/3)^i)$     $n(2/3)^i = 1$        $n = (3/2)^i$
- $i = \log_{3/2} n$
- Cost =  $n \log_{3/2} n = O(n \log_{3/2} n)$



## Example 4

---

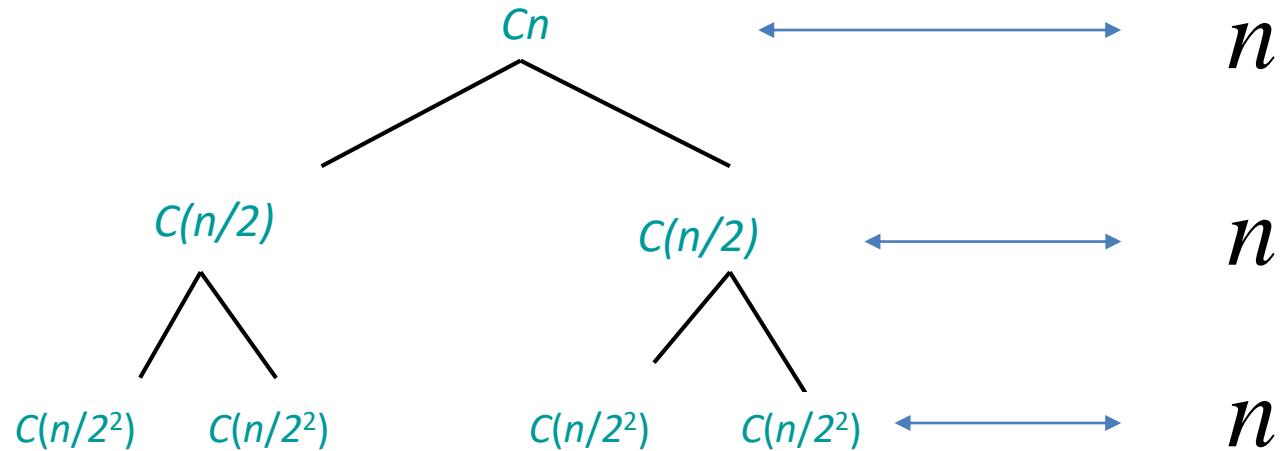
Solve  $T(n) = 2T(n/2) + O(n)$



## Example 4

---

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



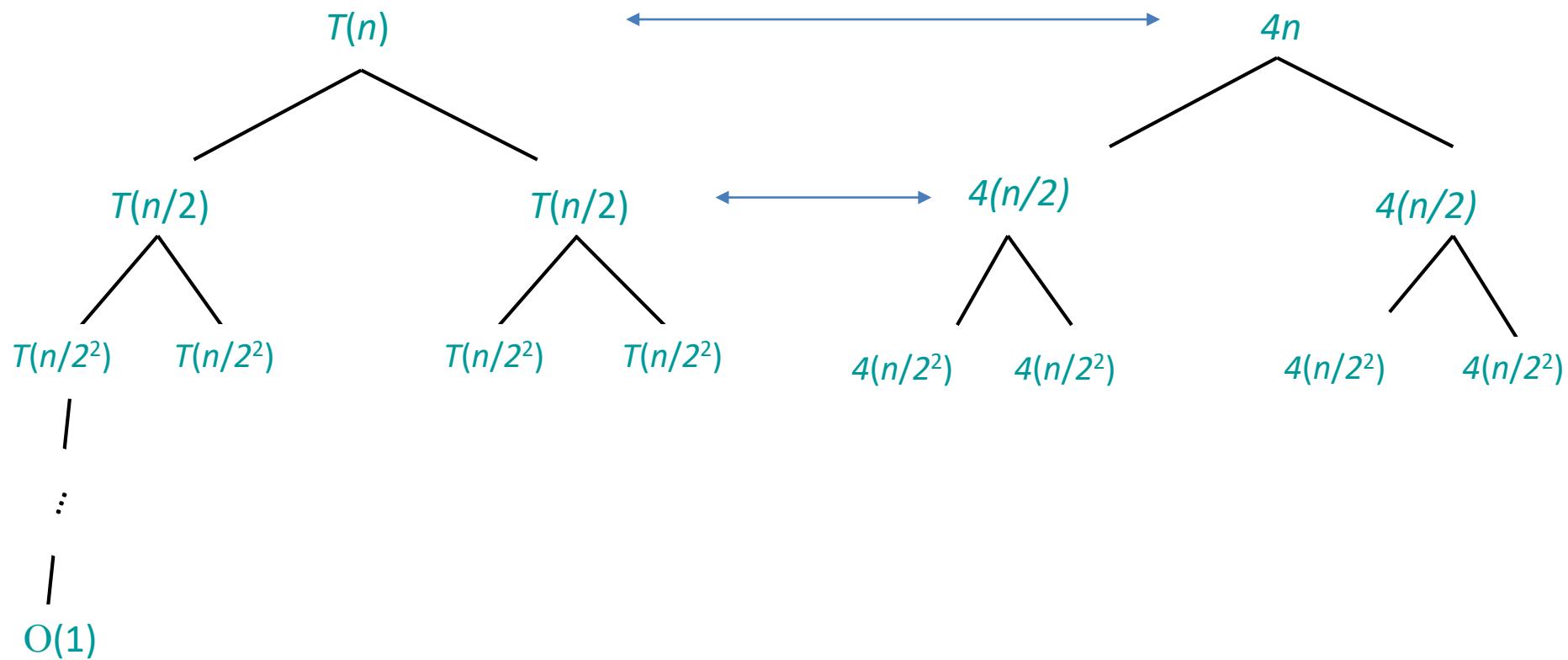
- Cost =  $n * \text{depth}$
- At  $O(1)=T(n/2^i)$   $n/2^i=1$
- $i=\log_2 n$
- Cost =  $n \log_2 n = O(n \log_2 n)$



## Example 5

---

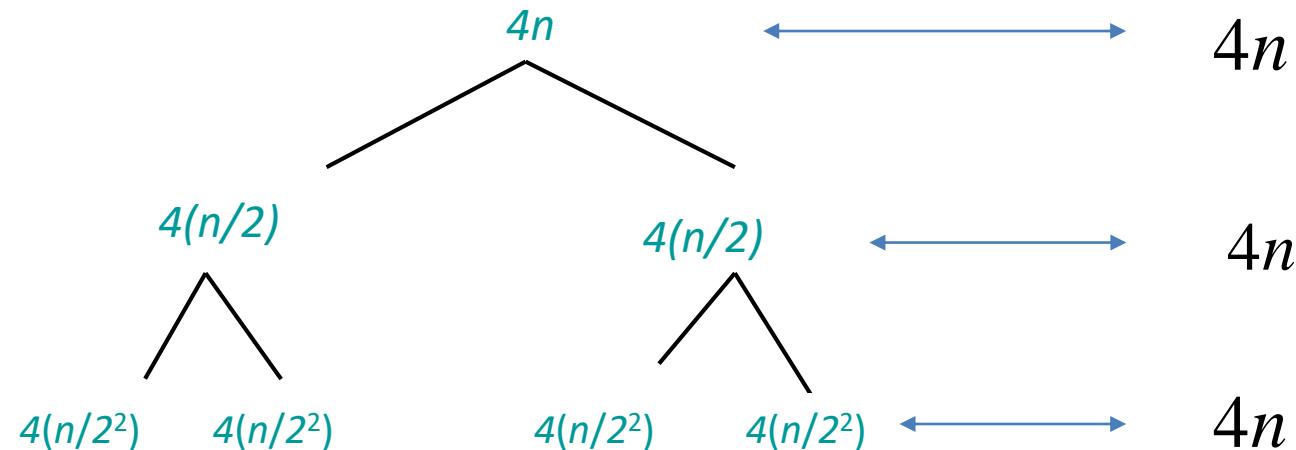
Solve  $T(n) = 2T(n/2) + 4n$



## Example 5

---

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



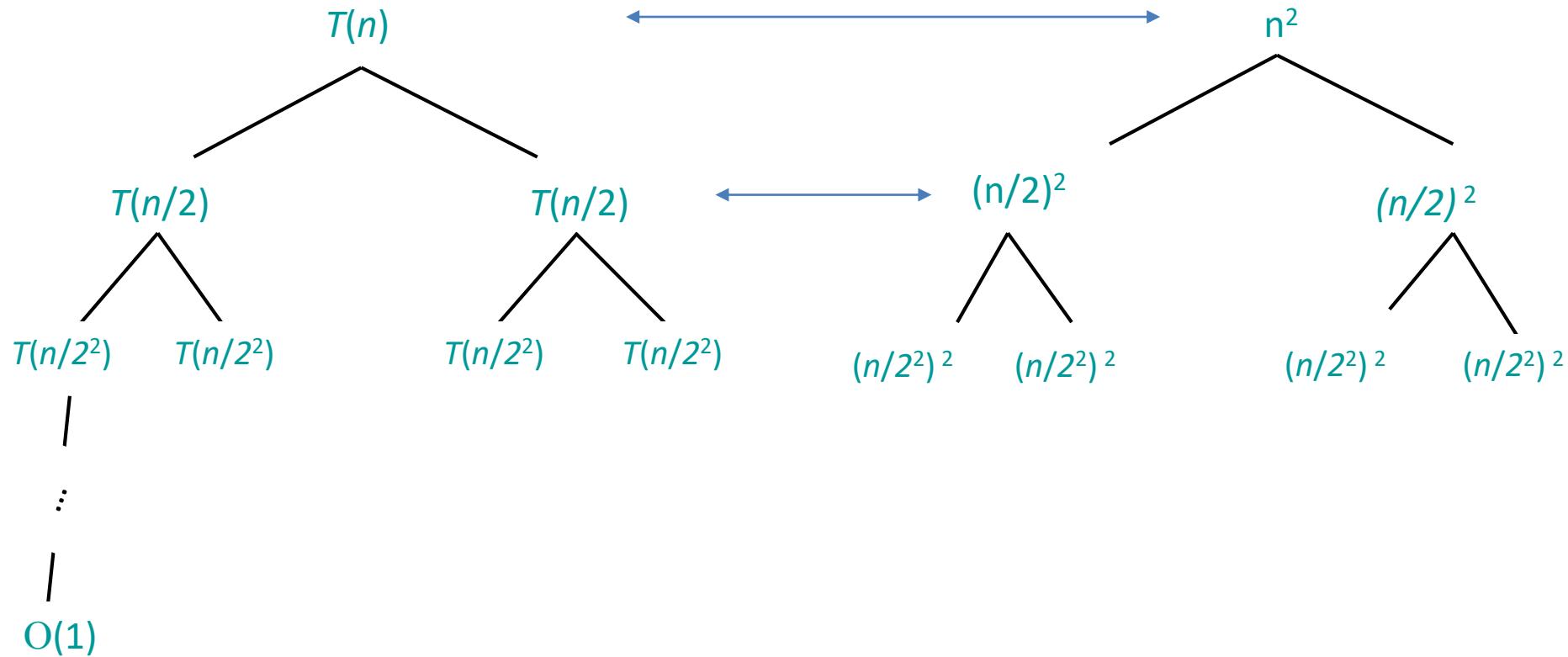
- Cost =  $4n * \text{depth}$
- At  $O(1)=T(n/2^i)$   $n/2^i=1$
- $i=\log_2 n$
- Cost =  $4n \log_2 n = O(n \log_2 n)$



## Example 6

---

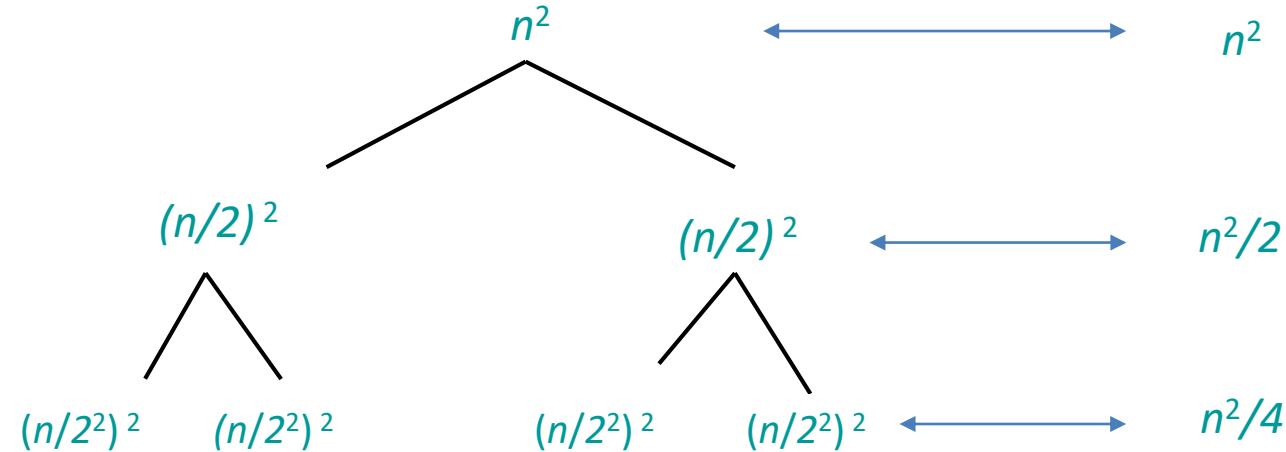
Solve  $T(n) = 2T(n/2) + n^2$



## Example 6

---

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



- Cost =  $n^2 * (1 + 1/2 + 1/2^2 + 1/2^3 + \dots + 1/2^n)$
- Cost =  $n^2 (1 / (1 - 1/2)) = 2 n^2$   
 $= O(n^2)$



## Module 1- Introduction

---

# Lecture No: 6 Master Theorem



# Master Method (cont..)

---

- In the standard recurrence relation each term has some significance,

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

where,

- n is the size of the problem.
- a is the number of sub problems in the recursion.
- n/b is the size of each sub problem. (Here it is assumed that all sub problems are essentially the same size.)
- f (n) is the sum of the work done outside the recursive calls, which includes the sum of dividing the problem and the sum of combining the solutions to the sub problems.
- It is not possible always bound the function according to the requirement, so we make three cases which will tell us what kind of bound we can apply on the function.



# The 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 non negative integers by the recurrence.
  - Then  $T(n)$  has the following asymptotic bounds,
- If  $f(n) = O(n^{\log_b a - \epsilon})$  for some constant  $\epsilon > 0$ , then  $\mathbf{T(n) = \Theta(n^{\log_b a})}$
  - If  $f(n) = \Theta(n^{\log_b a})$ , then  $\mathbf{T(n) = \Theta(n^{\log_b a} \log n)}$
  - If  $f(n) = \Omega(n^{\log_b a + \epsilon})$  for some constant  $\epsilon > 0$  and if  $a f(n/b) \leq c f(n)$  for some constant  $c < 1$  and all sufficient large  $n$ , then  $\mathbf{T(n) = \Theta(f(n))}$



# The Master theorem(cont..)

---

Each of the conditions can be interpreted as:

- If the cost of solving the sub-problems at each level increases by a certain factor, the value of  $f(n)$  will become polynomial smaller than  $n^{\log_b a}$ . Thus, the time complexity is oppressed by the cost of the last level i.e.  $n^{\log_b a}$
- If the cost of solving the sub-problem at each level is nearly equal, then the value of  $f(n)$  will be  $n^{\log_b a}$ . Thus, the time complexity will be  $f(n)$  times the total number of levels ie.  $n^{\log_b a} * \log n$
- If the cost of solving the sub problems at each level decreases by a certain factor, the value of  $f(n)$  will become polynomially larger than  $n^{\log_b a}$ . Thus, the time complexity is oppressed by the cost of  $f(n)$ .



# Master Theorem Limitations

---

The master theorem cannot be used if:

- $T(n)$  is not monotone. eg.  $T(n) = \sin n$
- $f(n)$  is not a polynomial. eg.  $f(n) = 2^n$
- $a$  is not a constant. eg.  $a = 2n$
- $a < 1$



# Using the master method

---

- To use the master method, we simply determine which case of the master theorem applies and write down the answer.

- Example,

$$T(n) = 9 T(n/3) + n$$

- Step 1 :

Compare given recurrence relation with standard form i.e.  
 $T(n)=a T(n/b) + f(n)$  to  
find out values of a, b and f(n).

So, we have a = 9, b = 3 , f(n) = n

- Step 2 :

Calculate value of  $n^{\log_b a}$  using above values,

$$n^{\log_b a} = n^{\log_3 9} = n^2$$

# Using the master method (cont..)

---

- **Step 3 :**

Compare the values of function  $f(n)$  with the value of  $n^{\log_b a}$

**Case 1 :** If function  $n^{\log_b a}$  is larger than  $f(n)$  then case 1 is applicable,  $T(n) = \Theta(n^{\log_b a})$

**Case 2 :** If two functions are the same size then we multiply by a logarithmic factor and

the solution is  $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$

**Case 3 :** If function  $f(n)$  is larger than  $n^{\log_b a}$  then case 3 is applicable,  $T(n) = \Theta(f(n))$

- **Step 4 :**

Based on this we get  $f(n) < n^{\log_b a}$  so we can say for the given example, **case 1** is applicable so  $\textcolor{red}{T(n) = \Theta(n^{\log_b a})}$

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

# Example 1 :

---

**Q. Solve the given recurrence relation using Master Method**

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

- **Step 1 :**

Compare given recurrence relation with standard form i.e.  $T(n)=a T(n/b) + f(n)$  to find out values of a, b and f(n).

So, we have  $a = 4$ ,  $b = 2$  ,  $f(n) = n^2$

- **Step 2 :**

Calculate the value of  $n^{\log_b a}$  using above values,

$$\begin{aligned}n^{\log_b a} &= n^{\log_2 4} \\&= n^{2 \log_2 2} \\&= n^2\end{aligned}$$

## Example 1 (cont..) :

---

- **Step 3 :**

Compare the values of function  $f(n)$  with the value of  $n^{\log_b a}$ .

$$f(n) = n^2 \text{ and } n^{\log_b a} = n^2$$

For given recurrence relation we have  $f(n) = n^{\log_b a}$

Hence **case 2** is applicable i.e. **If two functions are the same size then we multiply by a logarithmic factor and the solution is  $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(f(n) \log n)$**

- **Step 4 :**

Based on this **case 2**  $T(n) = \Theta(n^{\log_b a} \log n)$

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



## Example 2 :

---

**Q. Solve the given recurrence relation using Master Method**  
 **$T(n) = 16 T(n/4) + n!$ .**

- **Step 1 :**

Compare given recurrence relation with standard form i.e.  $T(n)=a T(n/b) + f(n)$  to find out values of a, b and f(n).

So, we have  $a = 16$ ,  $b = 4$  ,  $f(n) = n!$

- **Step 2 :**

Calculate the value of  $n^{\log_b a}$  using above values,

$$\begin{aligned}n^{\log_b a} &= n^{\log_4 16} \\&= n^{2 \log_4 4} \\&= n^2\end{aligned}$$

## Example 2 (cont..) :

---

- **Step 3 :**

Compare the values of function  $f(n)$  with the value of  $n^{\log_b a}$ .

$$f(n) = n! \quad \text{and} \quad n^{\log_b a} = n^2$$

For given recurrence relation we have  $f(n) > n^{\log_b a}$

Hence **case 3** is applicable i.e. **If function  $f(n)$  is larger than  $n^{\log_b a}$  then case 3 is applicable,  $T(n) = \Theta(f(n))$**

- **Step 4 :**

Based on this **case 3**  $T(n) = \Theta(f(n))$

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

## Example 3 :

**Q. Solve the given recurrence relation using Master Method**

$$T(n) = \sqrt{2} T(n/2) + \log n.$$

- **Step 1 :**

Compare given recurrence relation with standard form i.e.  $T(n)=a T(n/b) + f(n)$  to find out values of a, b and f(n).

So, we have  $a = \sqrt{2}$ ,  $b = 2$ ,  $f(n) = \log n$

- **Step 2 :**

Calculate the value of  $n^{\log_b a}$  using above values,

$$\begin{aligned}n^{\log_b a} &= n^{\log_2 \sqrt{2}} \\&= n^{1/2 \log_2 2} \\&= n^{1/2} = \sqrt{n}\end{aligned}$$

## Example 3 (cont..) :

---

- **Step 3 :**

Compare the values of function  $f(n)$  with the value of  $n^{\log_b a}$ .

$$f(n) = \log n \quad \text{and} \quad n^{\log_b a} = \sqrt[n]{n^a}$$

For given recurrence relation we have  $f(n) < n^{\log_b a}$

Hence **case 1** is applicable i.e. **If function  $n^{\log_b a}$  is larger than  $f(n)$  then case 1 is applicable,  $T(n) = \Theta(n^{\log_b a})$**

- **Step 4 :**

Based on this **case 1**  $T(n) = \Theta(n^{\log_b a})$

$$T(n) = \Theta(\sqrt[n]{n^a})$$

## Example 4 :

**Q. Solve the given recurrence relation using Master Method**

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

- **Step 1 :**

Compare given recurrence relation with standard form i.e.  $T(n)=a T(n/b) + f(n)$  to

find out values of a, b and f(n).

So, we have  $a = 2^n$ ,  $b = 2$ ,  $f(n) = n^n$

As per theorem value of 'a' should be constant and in given recurrence relation value of 'a' keeps on changing for all n. Therefore, a is not constant.

**Hence, Master Method can not be applied.**



## Example 5 :

**Q. Solve the given recurrence relation using Master Method**

$$T(n) = 0.5 T(n/2) + 1/n.$$

- **Step 1 :**

Compare given recurrence relation with standard form i.e.  $T(n)=a T(n/b) + f(n)$  to

find out values of a, b and f(n).

So, we have  $a = 0.5$ ,  $b = 2$  ,  $f(n) = 1/n$

To apply master theorem **value of a  $\geq 1$**  and in given recurrence relation **value of a  $< 1$** .

**Hence, Master Method can not be applied.**

## Example 6 :

---

**Q. Solve the given recurrence relation using Master Method**

$$T(n) = 64 T(n/8) - n^2 \log n.$$

• **Step 1 :**

Compare given recurrence relation with standard form i.e.  $T(n)=a T(n/b) + f(n)$  to find out values of  $a$ ,  $b$  and  $f(n)$ .

So, we have  $a = 64$ ,  $b = 8$  ,  $f(n) = - n^2 \log n$

To apply master theorem value of  $a \geq 1$  ,  $b > 1$  and  $f(n)$  should be positive function. But in given recurrence relation value of  $f(n)$  is not positive.

**Hence, Master Method can not be applied.**



# Practice Problems

---

1.  $T(n) = 3T(n/2) + n^2$
2.  $T(n) = 4T(n/2) + n^2$
3.  $T(n) = T(n/2) + 2^n$
4.  $T(n) = 16T(n/4) + n$
5.  $T(n) = 2T(n/2) + n \log n$
6.  $T(n) = 2T(n/4) + n^{0.58}$
7.  $T(n) = 4T(n/2) + cn$
8.  $T(n) = 3T(n/3) + n/2$
9.  $T(n) = 4T(n/2) + n^2 + n$
10.  $T(n) = T(n/2 + 5) + n^2$

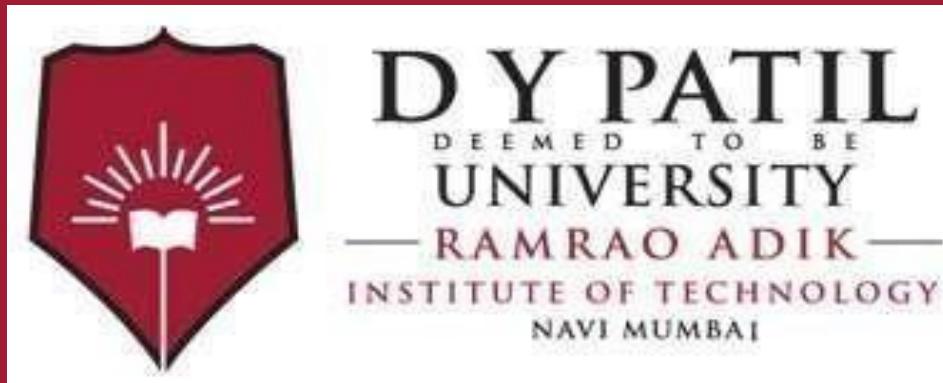
## Activity

---



<https://docs.google.com/forms/d/1PpTizWXcnPfJEZ17KgMoz3M79CjW3TV7wJzxI9AfdNY/edit?usp=sharing>

# Thank You



# Design and Analysis of Algorithms

## Module 2:

# Divide and Conquer Approach

## Index -

---

Lecture 09 - Introduction to Divide and Conquer Approach	3
Lecture 10 – Analysis of Binary Search	10
Lecture 11 – Analysis of Merge Sort	17
Lecture 12 – Analysis of Quick Sort	33

# Introduction to Divide and Conquer Approach



## Divide and Conquer General Method

---

- divide and conquer is an **algorithm design paradigm**.
- algorithm design paradigm is a generic model or framework which underlies the design of a class of algorithms.
- Strategy:
  - A **divide-and-conquer** algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.
  - The solutions to the sub-problems are then **combined** to give a solution to the original problem
- It is the basis of efficient algorithms for many problems such as merge sort, Quick sort.
- The correctness of a divide-and-conquer algorithm is usually proved by mathematical induction,
- The computational cost is often determined by solving recurrence relations.



## Divide and Conquer Approach

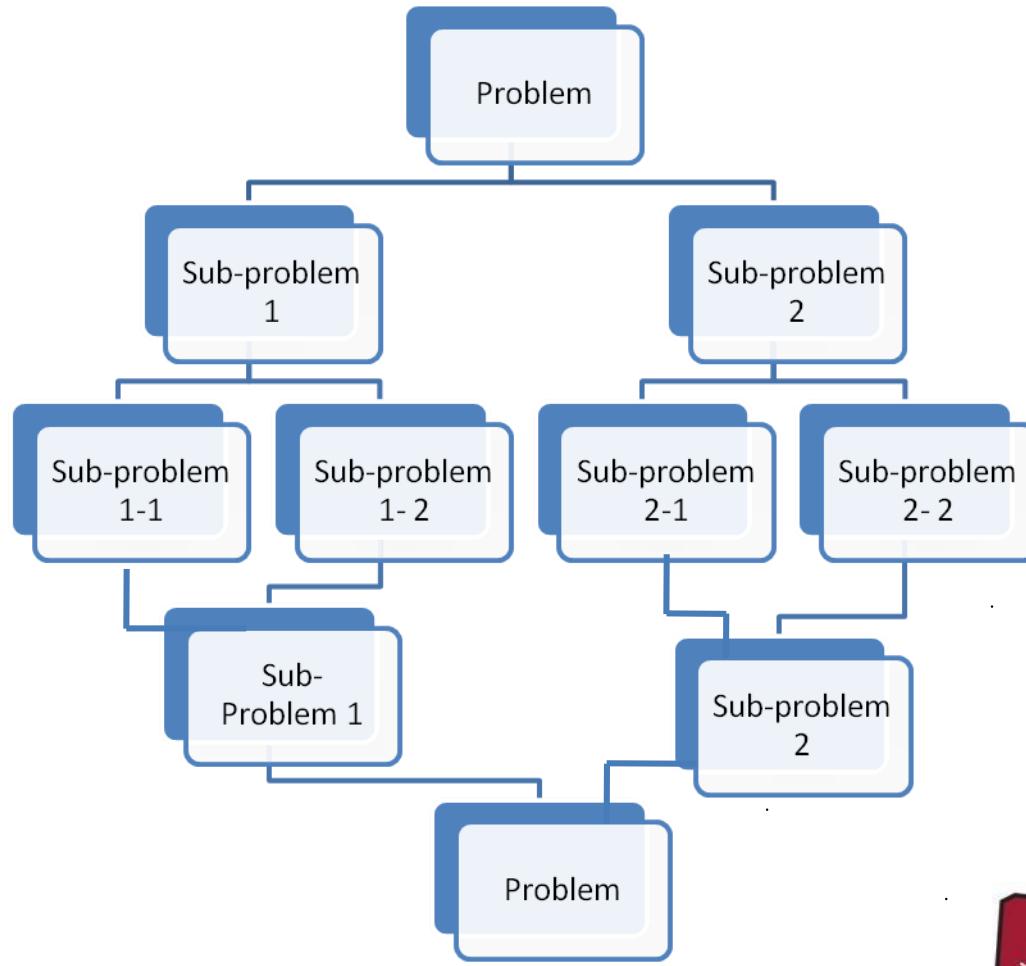
---

- think of divide-and-conquer algorithm as having three parts:
  1. Divide the problem into a number of subproblems that are smaller or instances of the same problem.
  2. Conquer the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
  3. Combine the solutions to the subproblems into the solution for the original problem.

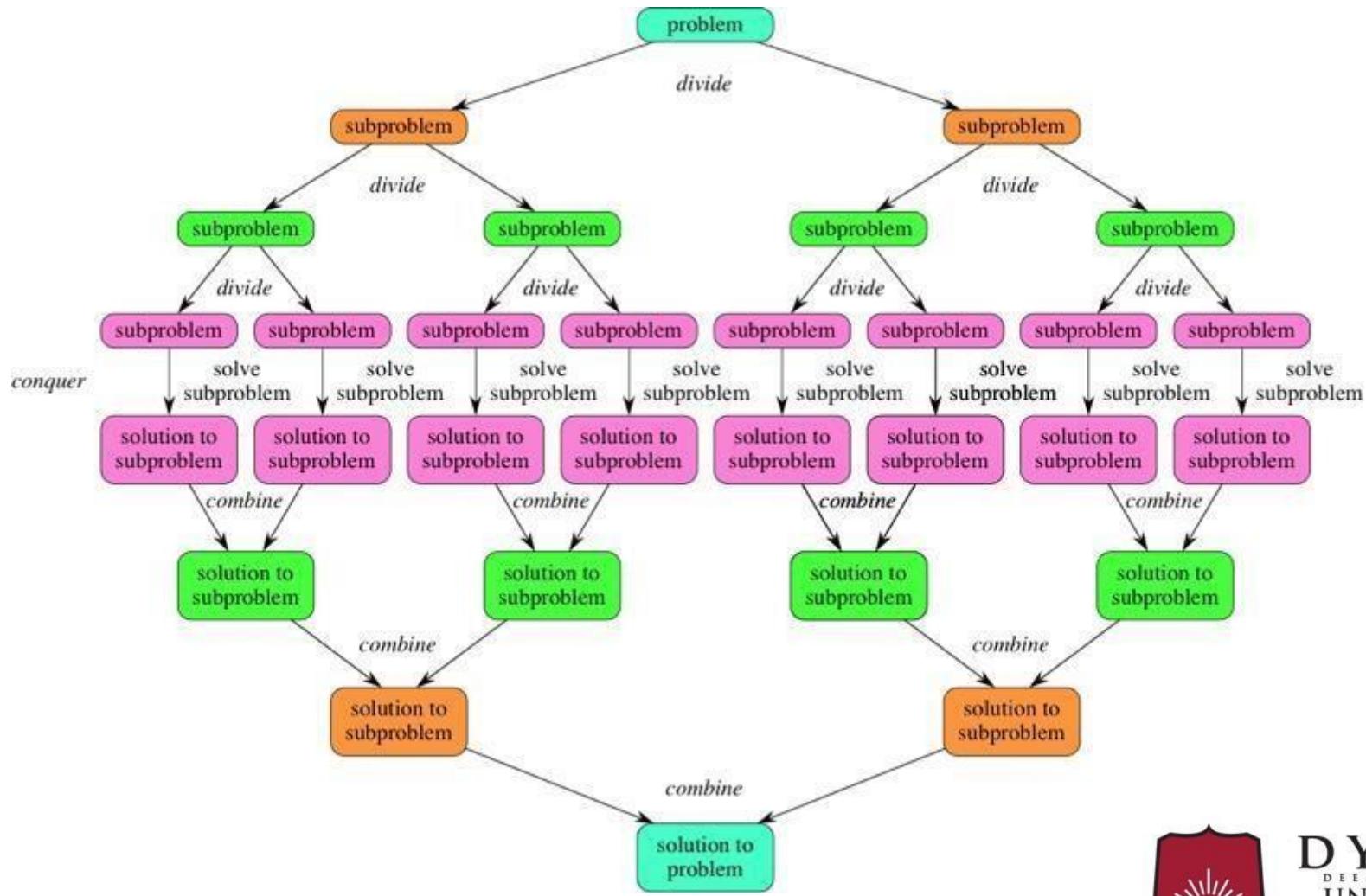


# Divide and Conquer Approach

---



# Divide and Conquer Approach



## Divide and Conquer - Advantages

---

- **Solving difficult problems**
- **Algorithm efficiency**
- **Parallelism**
- **Memory access**
- **Round off control**



## Lecture 10

---

# Analysis of Binary Search



## Binary Search - Example

---

- Binary Search, locates/find a element from the list of sorted items.
- Binary search, a divide-and-conquer algorithm where the sub problems are of roughly half the original size.
- Binary search compares the target value to the middle element of the array.
  - If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half.
  - again taking the middle element to compare to the target value, and repeating this until the target value is found.
  - If the search ends with the remaining half being empty, the target is not in the array



## Binary Search

---

- Find  $x$  in array [low.....high]

- Verify base case
- Compare  $x$  with middle element in the array
  - There are 3 possible outcome
  - Case 1 :  $x == \text{array}[\text{mid}]$

- return mid (index of middle element)

- Case 2:  $x < \text{array}[\text{mid}]$

- Find  $x$  in array [low.....mid-1]
- Note: this is same smaller sub-problem.

- Case 3:  $x > \text{array}[\text{mid}]$

- Find  $x$  in array [mid+1.....high]
- Note: this is same smaller sub-problem.



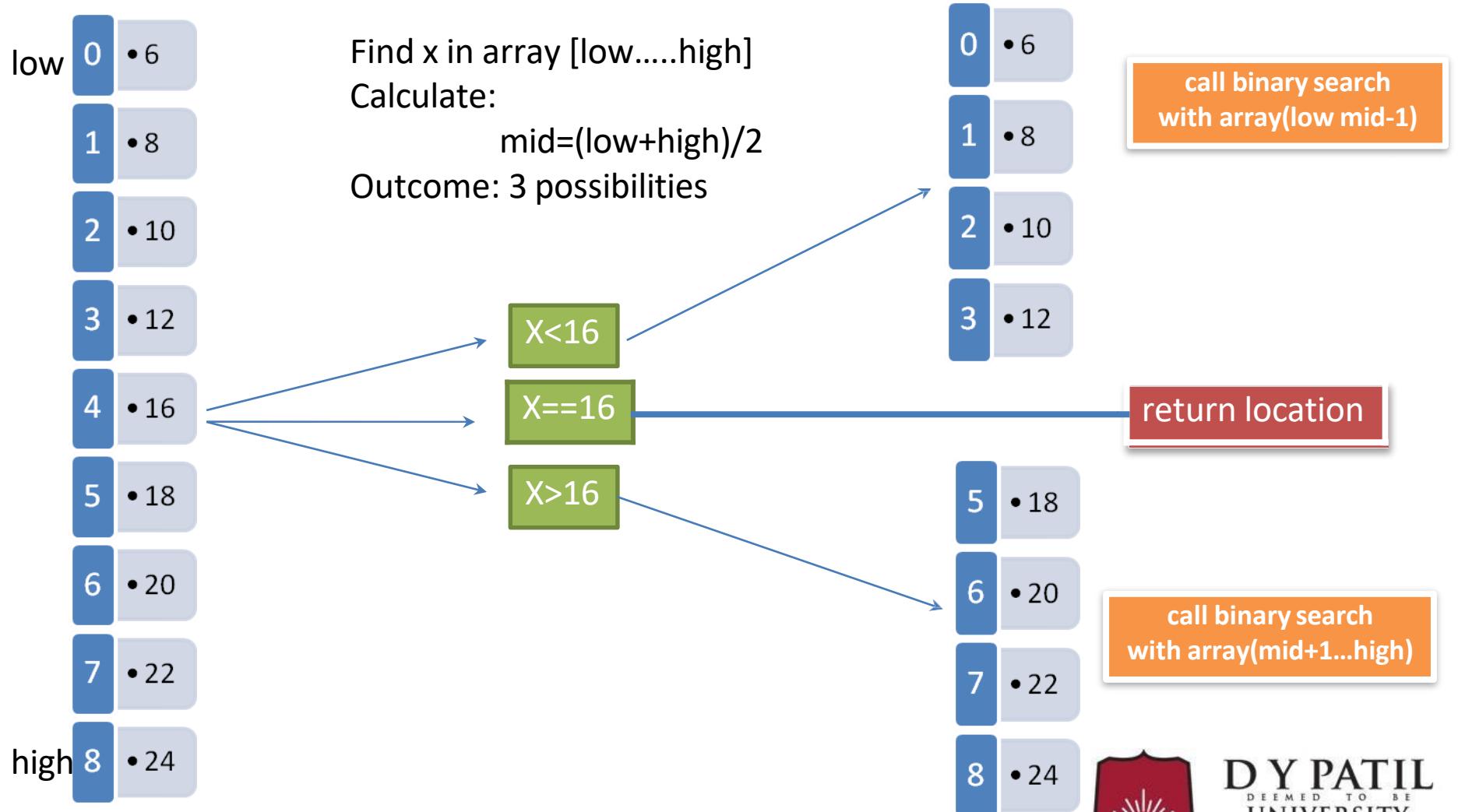
## Binary Search Algorithm

---

1. Read array elements in increasing order.
2. Read the element to be searched, say 'KEY'.
3. int BinarySearch(int array[], int start\_index, int end\_index, int KEY)  
    {  
        if (end\_index >= start\_index)  
        {  
            int middle = (start\_index+end\_index)/2;  
            if (array[middle] == KEY)  
                return middle;  
            if (array[middle] > KEY)  
                return BinarySearch(array, start\_index, middle-1, KEY);  
            elseif (KEY > array[middle])  
                return BinarySearch(array, middle+1, end\_index, KEY);  
        }  
        return -1;  
    }



## Binary Search - work



## Binary Search – Base case

---

- The recursive call base case - to stop recursion

```
If(low>high){  
    //stop recursion  
}
```

- We recursive divide and decrease search space, in case key is not equal to middle element by
  - Making search space between low and mid-1 if  $x < \text{array}[mid]$
  - Making search space between mid+1 and high if  $x > \text{array}[mid]$
- The above logic of divide and decreasing search space, if you have atleast one element in search space of array i.e  $\text{low} \leq \text{high}$



## Binary Search – Recurrence Equation

---

- $T(n)=T(n/2)+O(1)$
- Analysis using master's method
- Case 2 of master's method is application
  - $T(n)=aT(n/b) +f(n)$  where  $a \geq 1$  and  $b > 1$
  - If  $f(n) =\Theta(n^c)$  where  $c =\log_b a$  then  $T(n) =\Theta(n^c \log n)$
  - $a=1, b=2, c=0$ , also  $\log_b a =\log_2 1 =0$
  - Thus,  $T(n) =\Theta(n^0 \log n) =\Theta(\log n)$
  - Thus complexity of Binary search is  $\Theta(\log n)$  in worst case.



## Binary Search – Recurrence Equation

---

- $T(n)=T(n/2)+O(1)$
- Best case: in best the element we are looking is found and middle
- Thus, there is no recursive call further
- $T(n)=O(1)$



# Analysis of Merge Sort



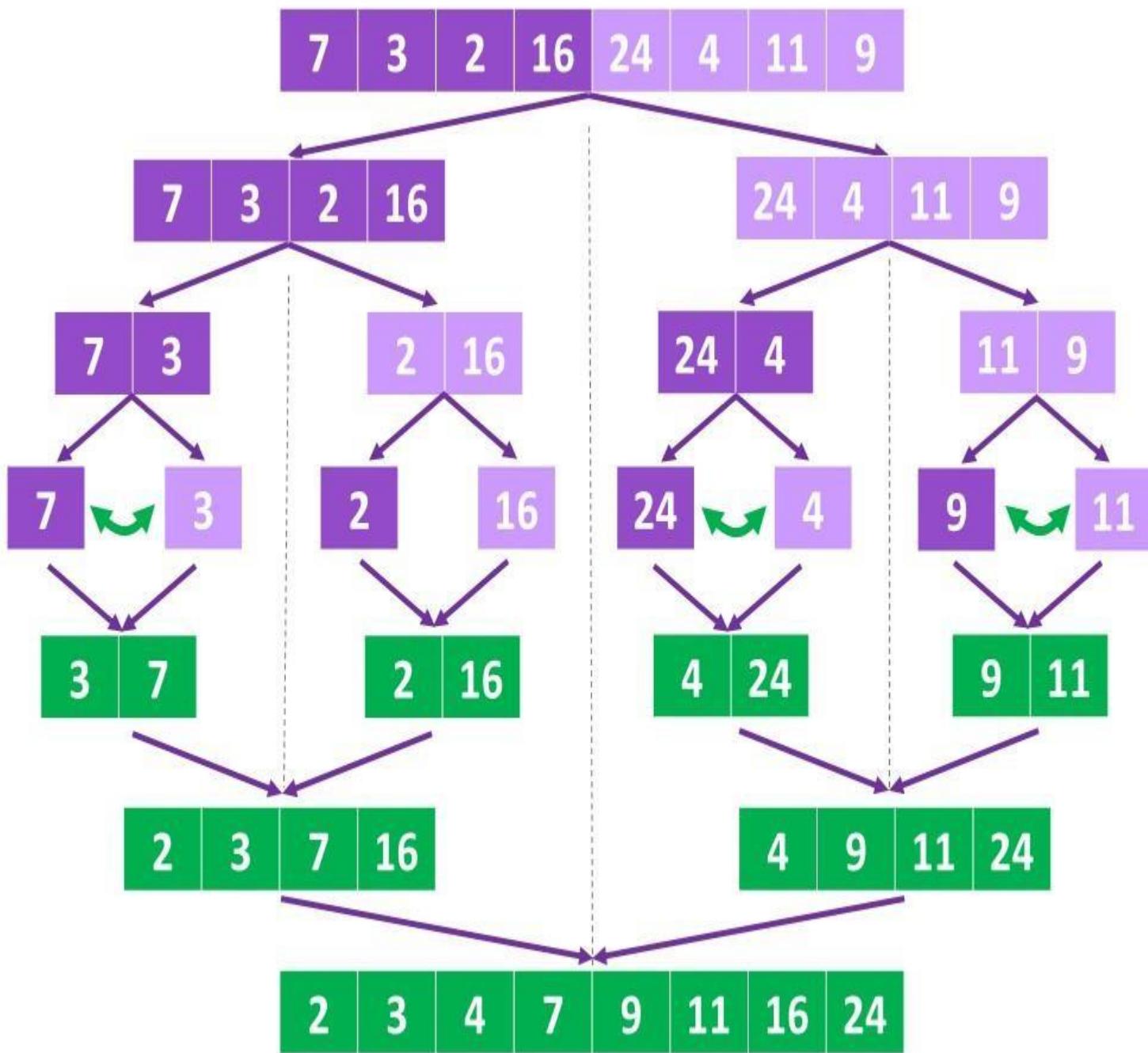
## Merge Sort

---

- The merge sort algorithm closely follows the divide-and-conquer methodology.
  - Divide: Divide the  $n$ -element sequence to be sorted into two subsequences of  $n/2$  elements each.
  - Conquer: Sort the two subsequences recursively using merge sort.
  - Combine: Merge the two sorted subsequences to produce the sorted answer
- The recursion "bottoms out" when the sequence to be sorted has length 1.



# Merge Sort



Step 1:  
Split sub-lists in two until you reach pair of values.

Step 3:  
Sort/swap pair of values if needed.

Step 4:  
Merge and sort sub-lists and repeat process till you merge to the full list.

## Merge Sort Algorithm

0	1	2	3	4	5	6	7
2	4	1	6	8	5	3	7

---

MERGESORT(A)

```
{  
    n<-length(A)  
    if(n<2)  
    {  
        return  
    }  
    mid<- n/2;  
    left =array of size (mid)  
    right=array of size(n-mid)  
    for(i=0 to mid-1)  
    {  
        left[i] <-A[i]  
    }  
    for(i=mid to n-1)  
    {  
        right[i-mid] <-A[i]  
    }  
    MERGESORT (L)  
    MERGESORT (R)  
    MERGESORT (L, R, A)  
}
```

0	1	2	3	4	5	6	7
2	4	1	6	8	5	3	7

## Merge Sort Algorithm

MERGESORT(L, R, A)  
{

```

nL < length(L)
nR < length(R)
i = 0, j = 0, k = 0
while(i < nL && j < nR)
{
    if(L[i] <= R[j])
    {
        A[k] <- L[i]; i <- i+1
    }
    else
    {
        A[k] <- R[j]; j <- j+1
    }
    k = k + 1
}
while(i < nL)
{
    A[k] = L[i]; i <- i+1; k <- k+1;
}
while(j < nR)
{
    A[k] = R[j]; j <- j+1; k <- k+1;
}
}
```

Module 2- Divide and  
Conquer

0	1	2	3	4	5	6	7
38	27	43	3	9	82	10	7

---

Best Case, Average Case and Worst case are same in Merge sort:

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

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

Time taken by left sublist to get sorted

Time taken for combine sublist

Time taken by right sublist to get sorted

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

## Analysis

---

- Recurrence Equation of Merge sort
- $T(n) = 2T(n/2) + O(n)$
- How we got above equation:
  - Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus,  $D(n) = \Theta(1)$ .
  - Conquer: We recursively solve two subproblems, each of size  $n/2$ , which contributes  $2T(n/2)$  to the running time.
  - Combine: We have already noted that the MERGE procedure on an  $n$ -element subarray takes time  $\Theta(n)$ , so  $C(n) = \Theta(n)$ .



## Analysis

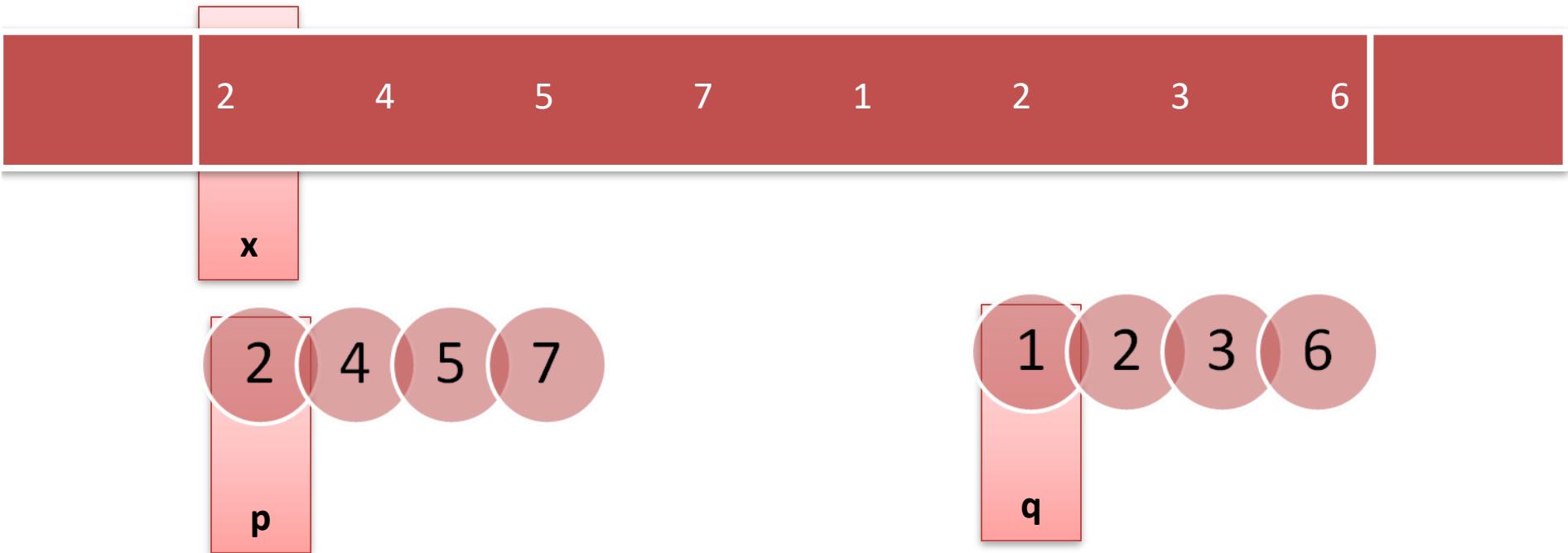
---

- Recurrence Equation of Merge sort
- $T(n)=2T(n/2)+O(n)$
- Analysis using master's method
- Case 2 of master's method is application
  - $T(n)=aT(n/b) +f(n)$  where  $a \geq 1$  and  $b > 1$
  - If  $f(n) =\Theta(n^c)$  where  $c =\log_b a$  then  $T(n) =\Theta(n^c \log n)$
  - $a=2, b=2, c=0$ , also  $\log_b a =\log_2 2 =1$
  - Thus,  $T(n) =\Theta(n^1 \log n) =\Theta(n \log n)$

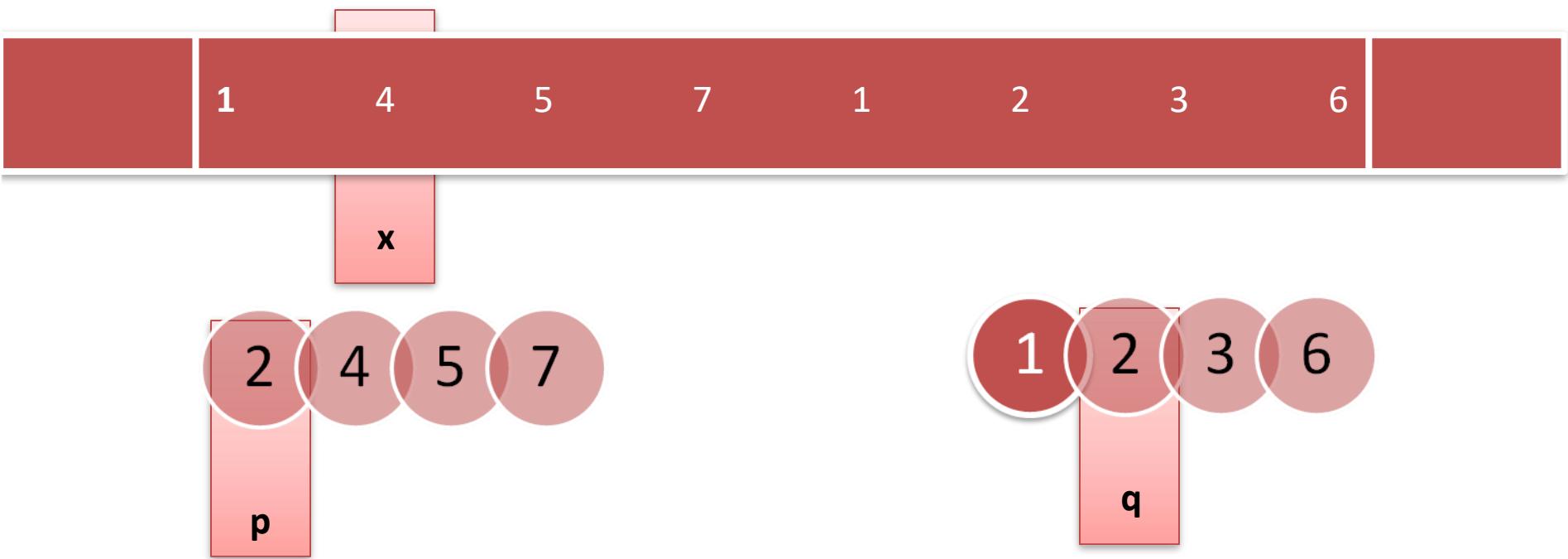


## Merge Algorithm - working

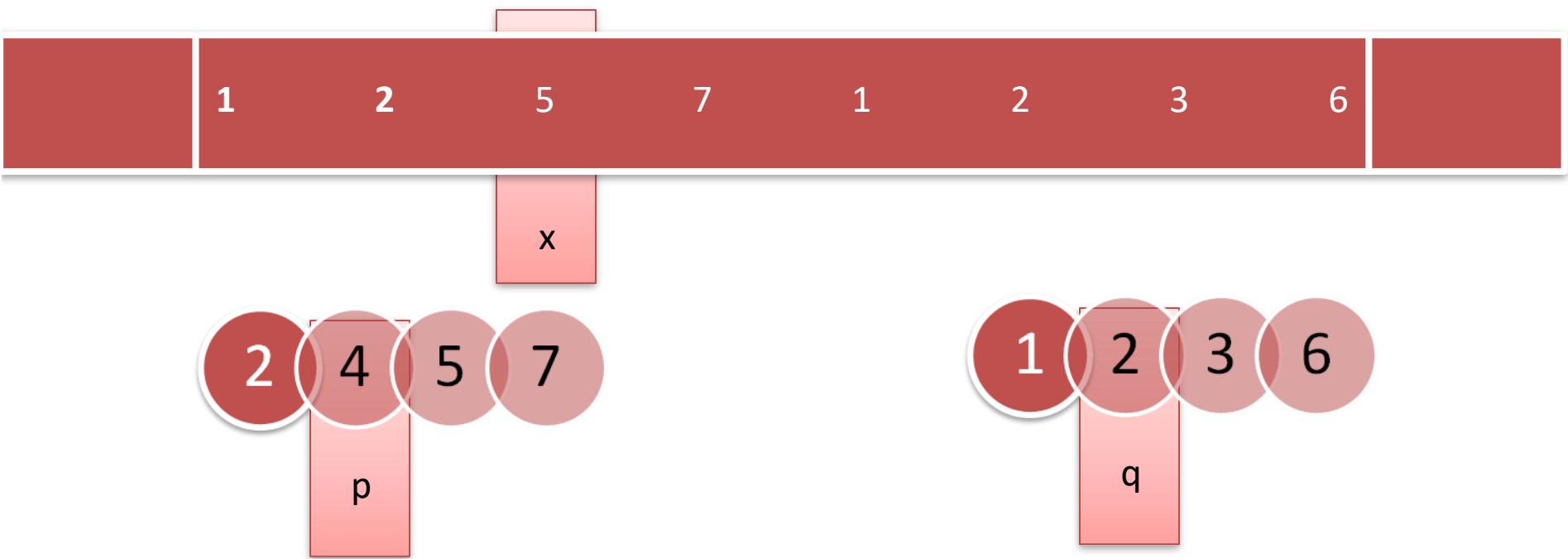
---



## Merge Algorithm - working

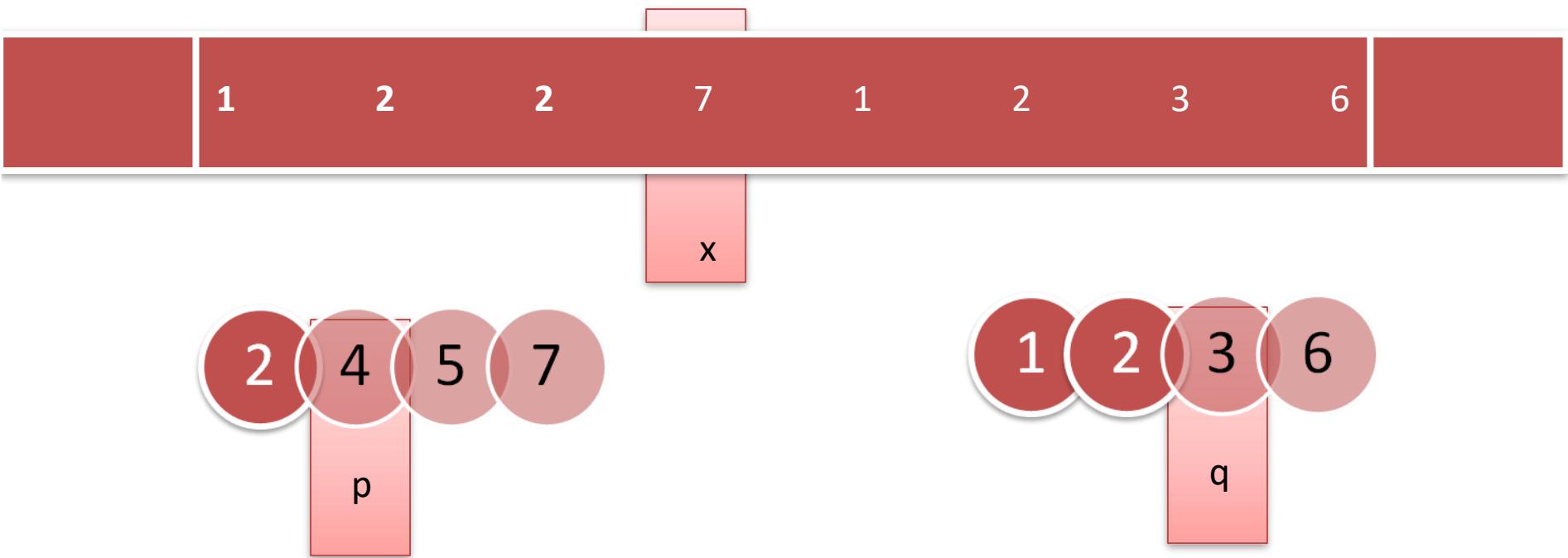


## Merge Algorithm - working



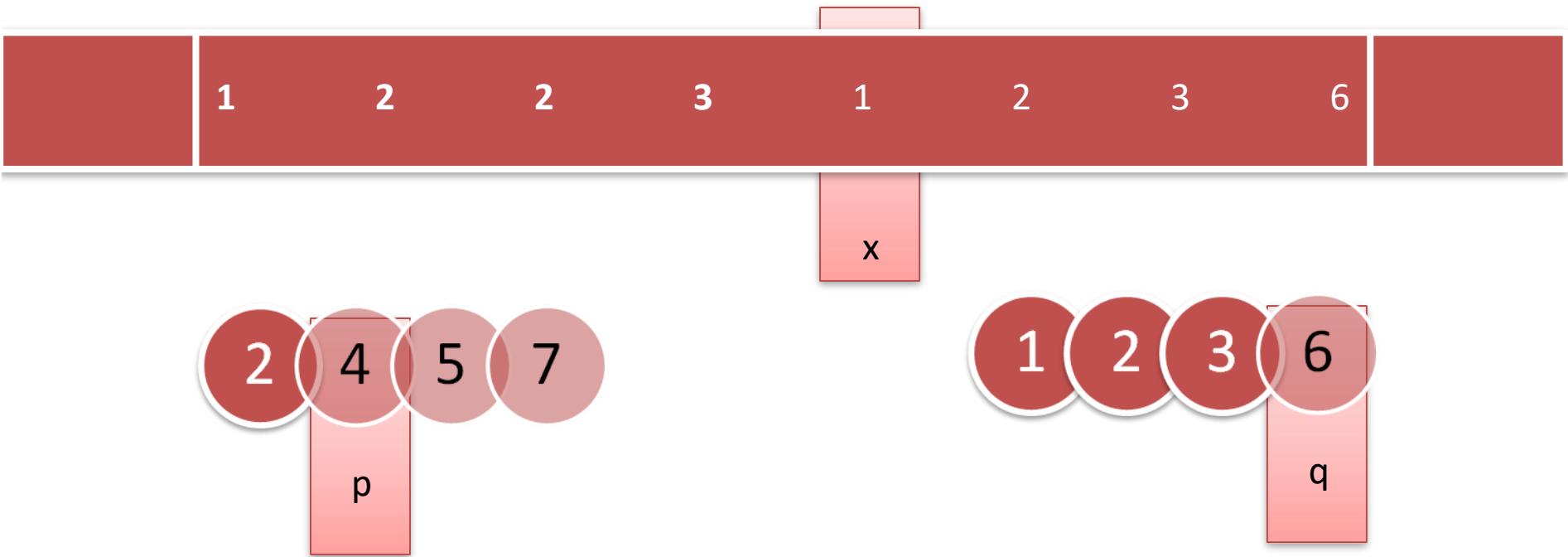
## Merge Algorithm - working

---

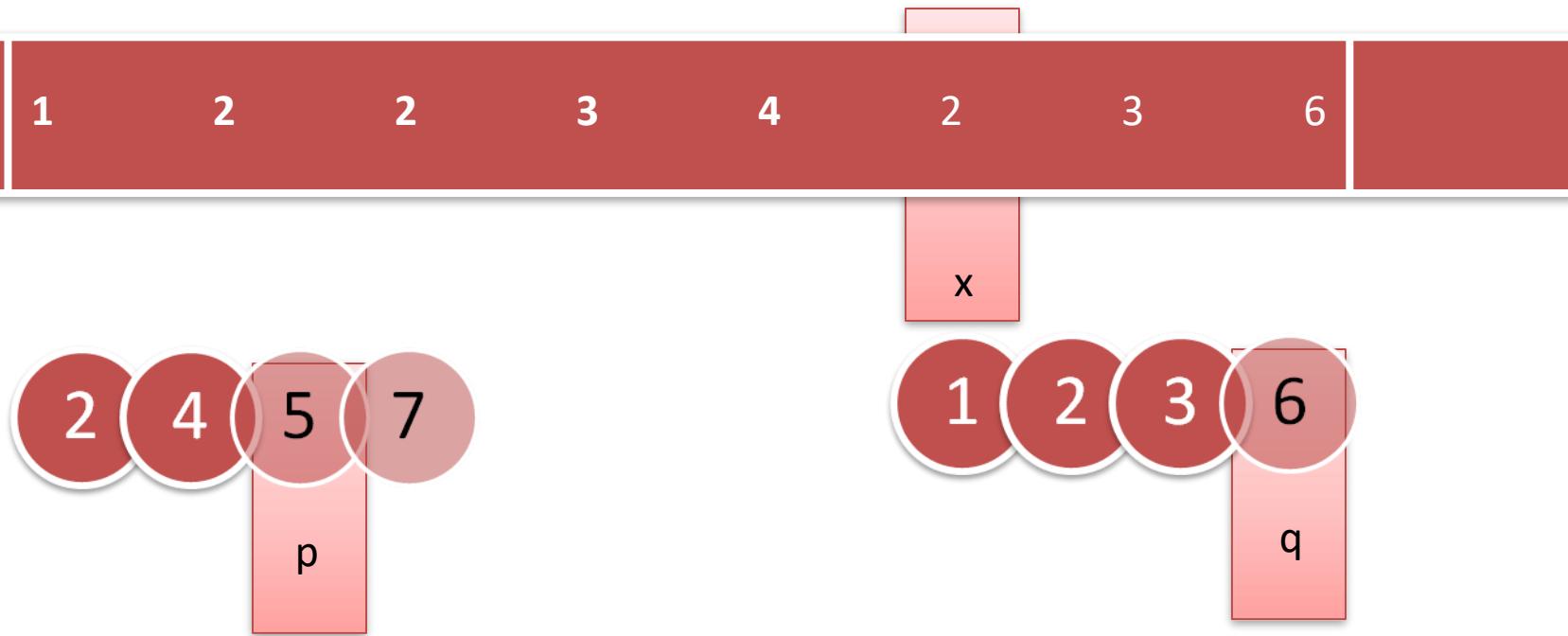


## Merge Algorithm - working

---

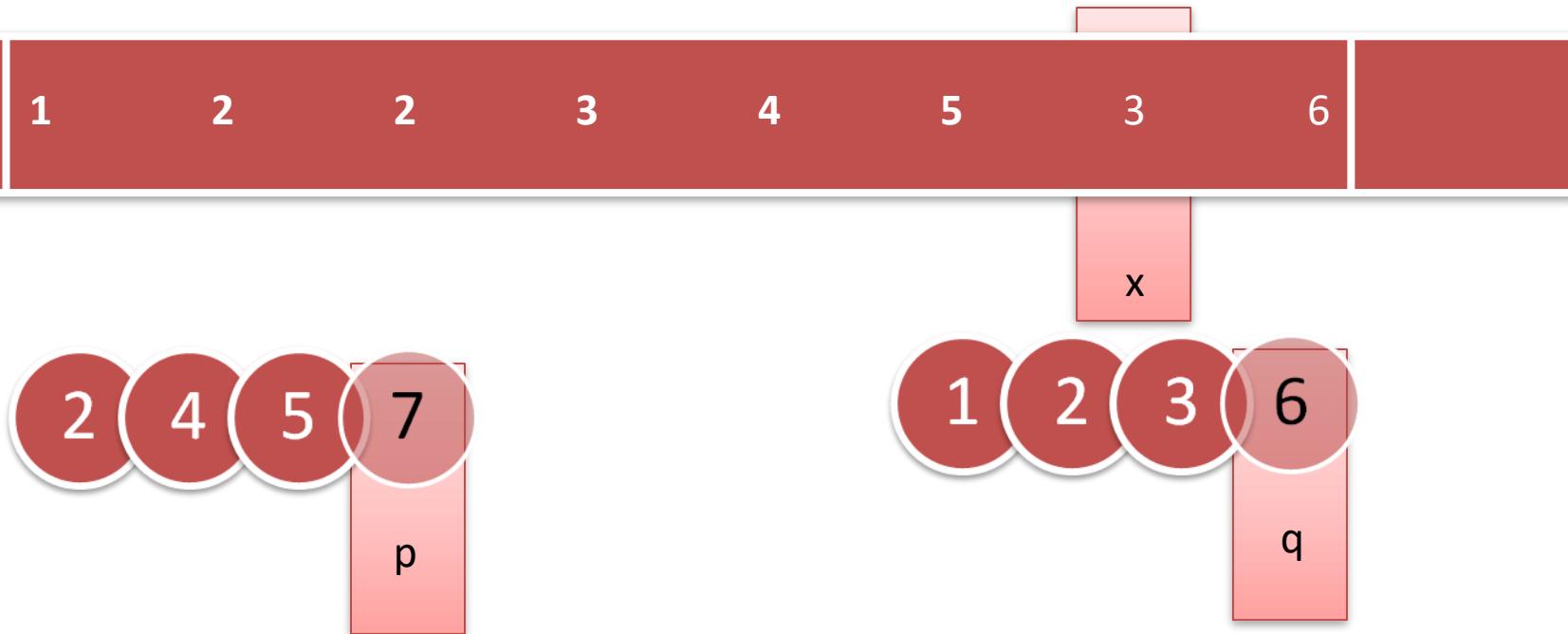


## Merge Algorithm - working

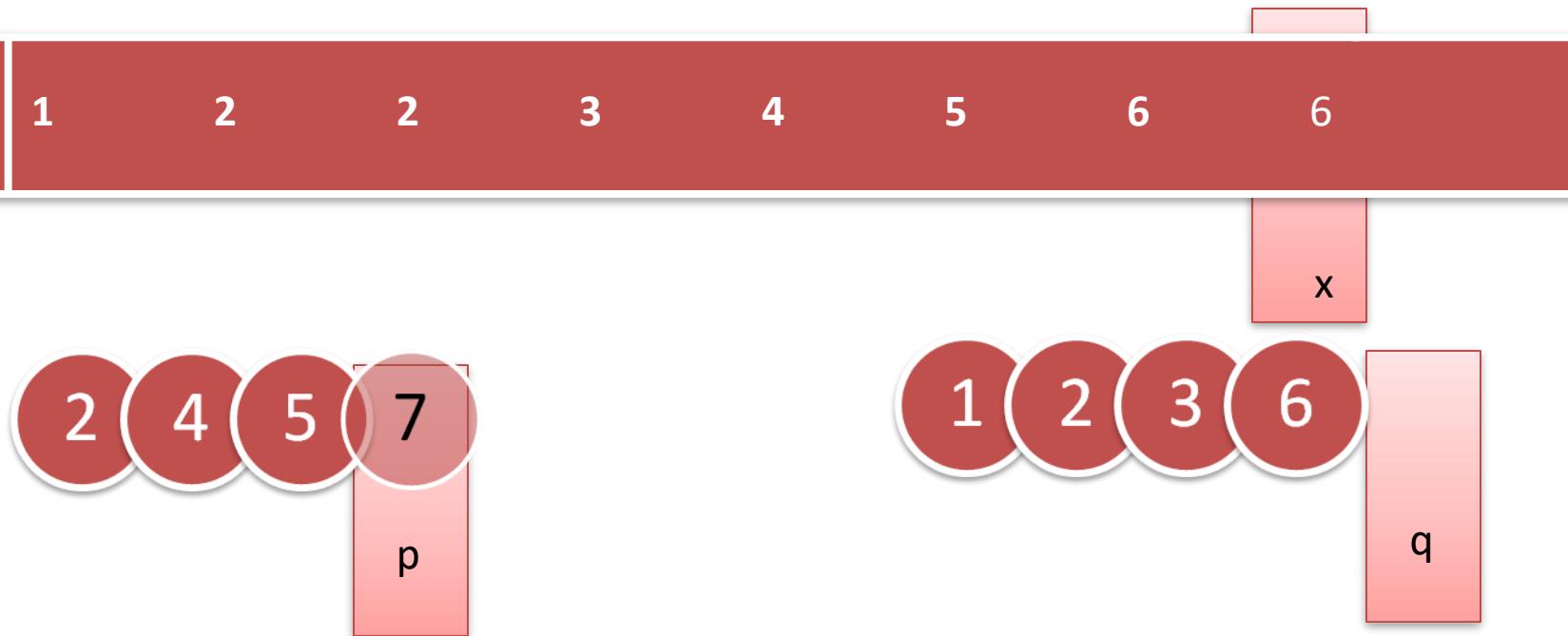


## Merge Algorithm - working

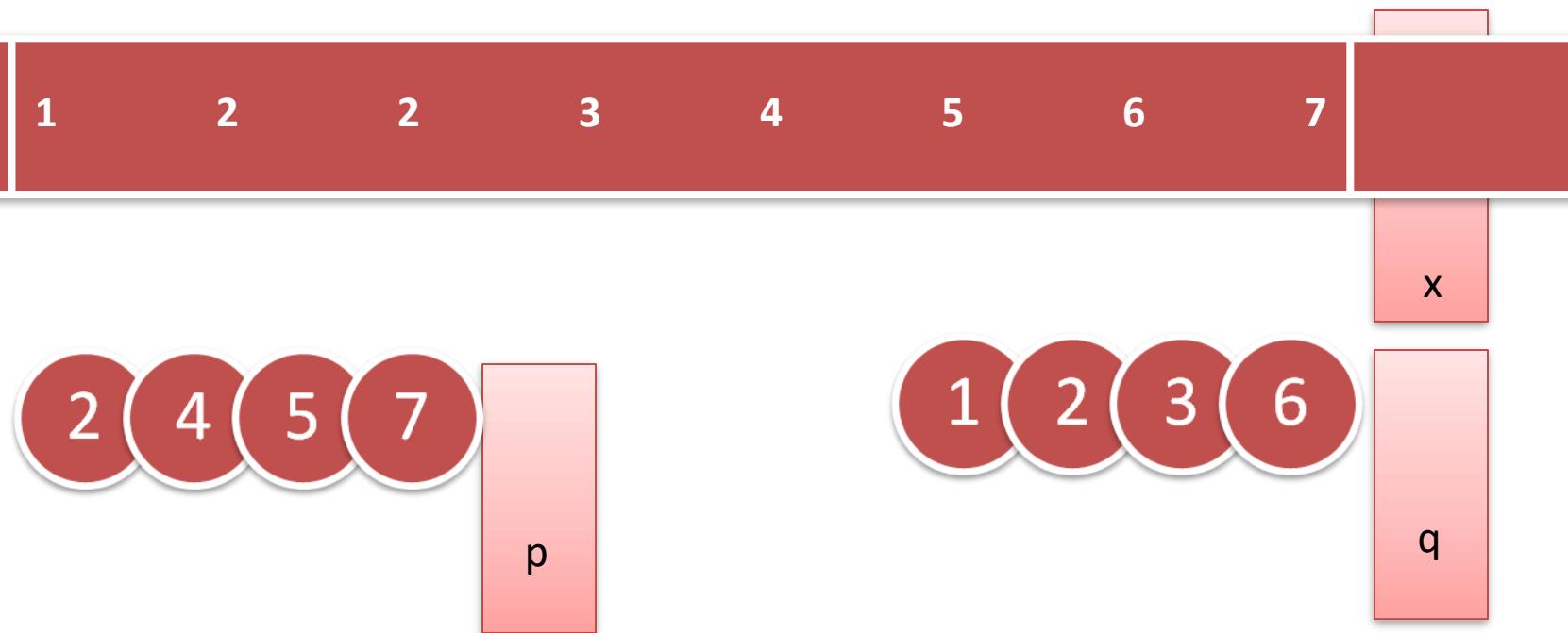
---



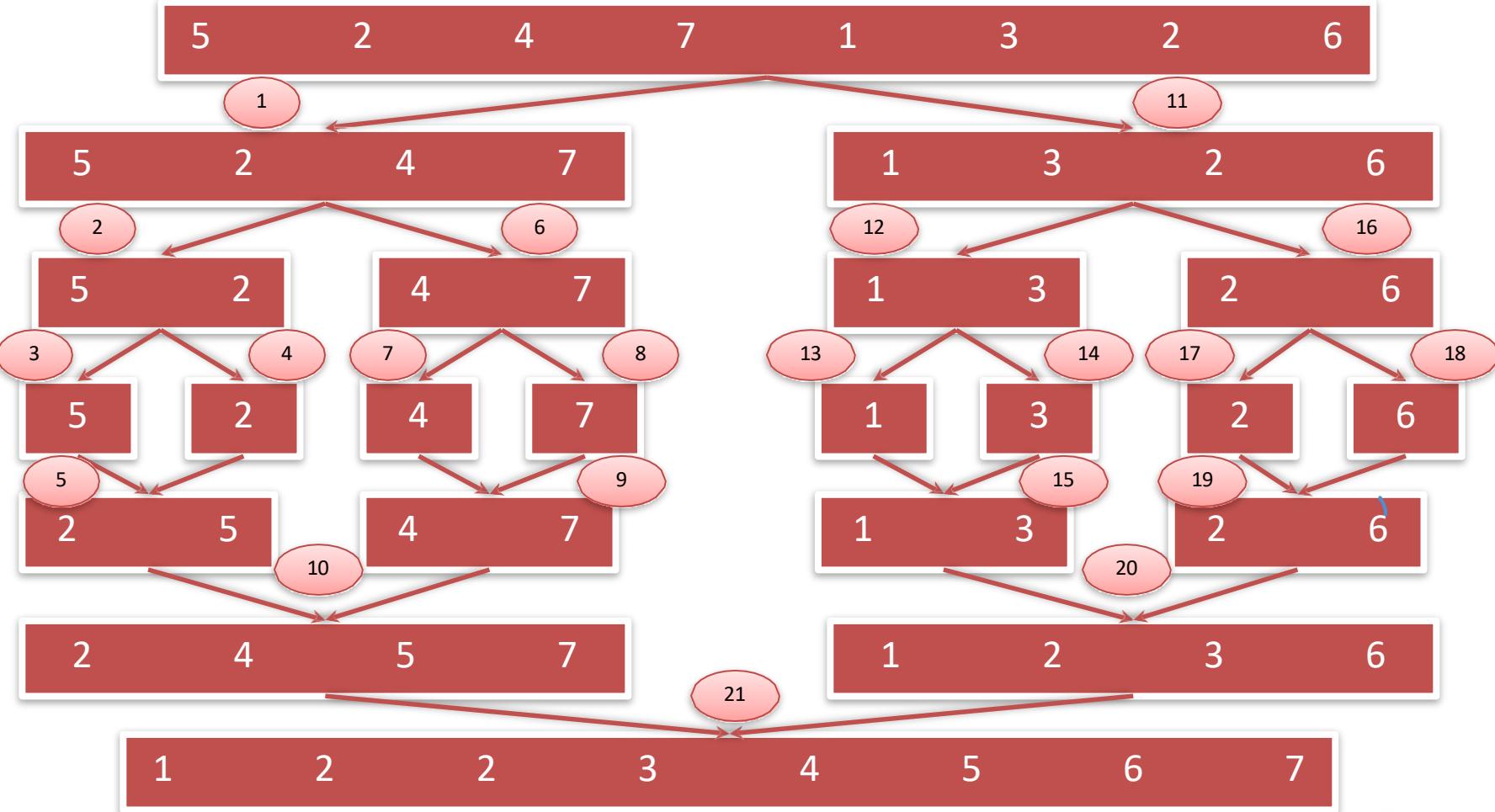
## Merge Algorithm - working



## Merge Algorithm - working



## Example....



## Lecture 12

---

# Analysis of Quick Sort



## Quick Sort

---

- Quicksort, like merge sort, is based on the divide-and-conquer paradigm
- It is the three-step divide-and-conquer process for sorting a typical subarray  $A[p \dots r]$ .
  - **Divide:** Partition (rearrange) the array  $A[p \dots r]$  into two (possibly empty) subarrays  $A[p \dots q - 1]$  and  $A[q + 1 \dots r]$ 
    - such that each element of  $A[p \dots q - 1]$  is less than or equal to  $A[q]$ , which is, in turn, less than or equal to each element of  $A[q + 1 \dots r]$ .
    - Compute the index  $q$  as part of this partitioning procedure.
  - **Conquer:** Sort the two subarrays  $A[p \dots q - 1]$  and  $A[q + 1 \dots r]$  by recursive calls to quicksort.
  - **Combine:** Since the subarrays are sorted in place, no work is needed to combine them: the entire array  $A[p \dots r]$  is now sorted.



## Quick Sort Algorithm

0	1	2	3	4	5	6	7	
35	50	15	25	80	20	90	45	$+\infty$



## Quick Sort Algorithm- Method 1

```
Quick_sort(l, h)
{
    if(l<h)
    {
        j=partition(l, h);
        Quick_sort(l, j)
        Quick_sort(j+1, h)
    }
}
```

0	1	2	3	4	5	6	7	
35	50	15	25	80	20	90	45	+∞



# Quick Sort Algorithm- Method 1

```
partition(l, h)
```

```
{
```

```
    pivot=A[l];
```

```
    i=l; j=h;
```

```
    while(i < j)
```

```
{
```

```
        do
```

```
            {i++;}
```

```
            while(A[i] <=pivot);
```

```
            while(A[j] >pivot)
```

```
                {j--;}
```

```
                if(i < j)
```

```
                    {swap(A[i], A[j]);}
```

```
}
```

```
        swap(A[l], A[j]);
```

```
        return j;
```

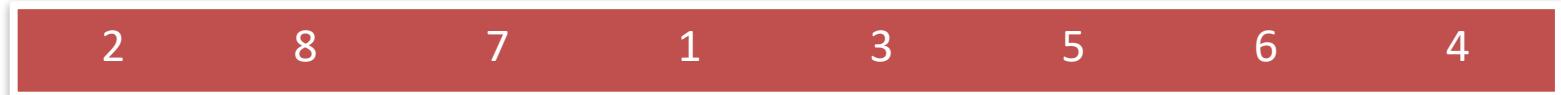
```
}
```

0	1	2	3	4	5	6	7	
35	50	15	25	80	20	90	45	+∞

## Quick Sort – Algorithm Method 2

- **QUICKSORT(A, p, r)**
  1. if  $p < r$
  2. then  $q \leftarrow \text{PARTITION}(A, p, r)$
  3.  $\text{QUICKSORT}(A, p, q - 1)$
  4.  $\text{QUICKSORT}(A, q + 1, r)$

- Quick sort divides array into subarrays:



$p$  \_\_\_\_\_ |

$q$

\_\_\_\_\_ |  $r$

Recursively call Quick sort  $A[q+1\dots r]$

Recursively call Quick sort  $A[p\dots q-1]$

## Quick Sort – Algorithm Method 2

- **QUICKSORT(A, p, r)**
  1. if  $p < r$
  2. then  $q \leftarrow \text{PARTITION}(A, p, r)$
  3.  $\text{QUICKSORT}(A, p, q - 1)$
  4.  $\text{QUICKSORT}(A, q + 1, r)$

- Recurrence Equation:

$$T(n) = T(q) + T(n-q) + f(n)$$

Call to quicksort A[p...q-1]

Call to quicksort A[q+1...r]

Call to partition



## Quick Sort – Algorithm Method 2

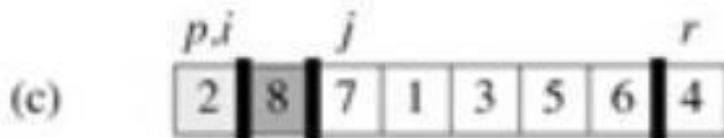
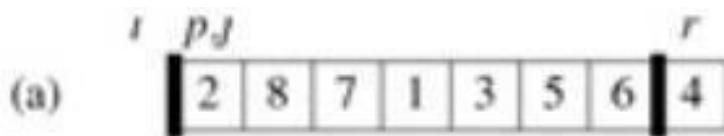
---

- PARTITION( $A, p, r$ )
  1.  $x \leftarrow A[r]$
  2.  $i \leftarrow p - 1$
  3. for  $j \leftarrow p$  to  $r - 1$ 
    4. do if  $A[j] \leq x$ 
      5. then  $i \leftarrow i + 1$
      6. exchange  $A[i] \leftrightarrow A[j]$
    7. exchange  $A[i + 1] \leftrightarrow A[r]$
    8. return  $i + 1$



## Quick Sort - Algorithm

- operation of PARTITION function on an 8-element array



## Quick Sort - Algorithm

- PARTITION( $A, p, r$ )
  1.         $x \leftarrow A[r]$
  2.         $i \leftarrow p - 1$
  3.        for  $j \leftarrow p$  to  $r - 1$ 
    4.              do if  $A[j] \leq x$
    5.                      then  $i \leftarrow i + 1$
    6.                      exchange  $A[i] \leftrightarrow A[j]$
  7.        exchange  $A[i + 1] \leftrightarrow A[r]$
  8.        return  $i + 1$

- The running time of PARTITION on the subarray  $A[p.....r]$  is  $\Theta(n)$
- Thus recurrence equation becomes:

$$T(n) = T(q) + T(n-q) + \Theta(n)$$



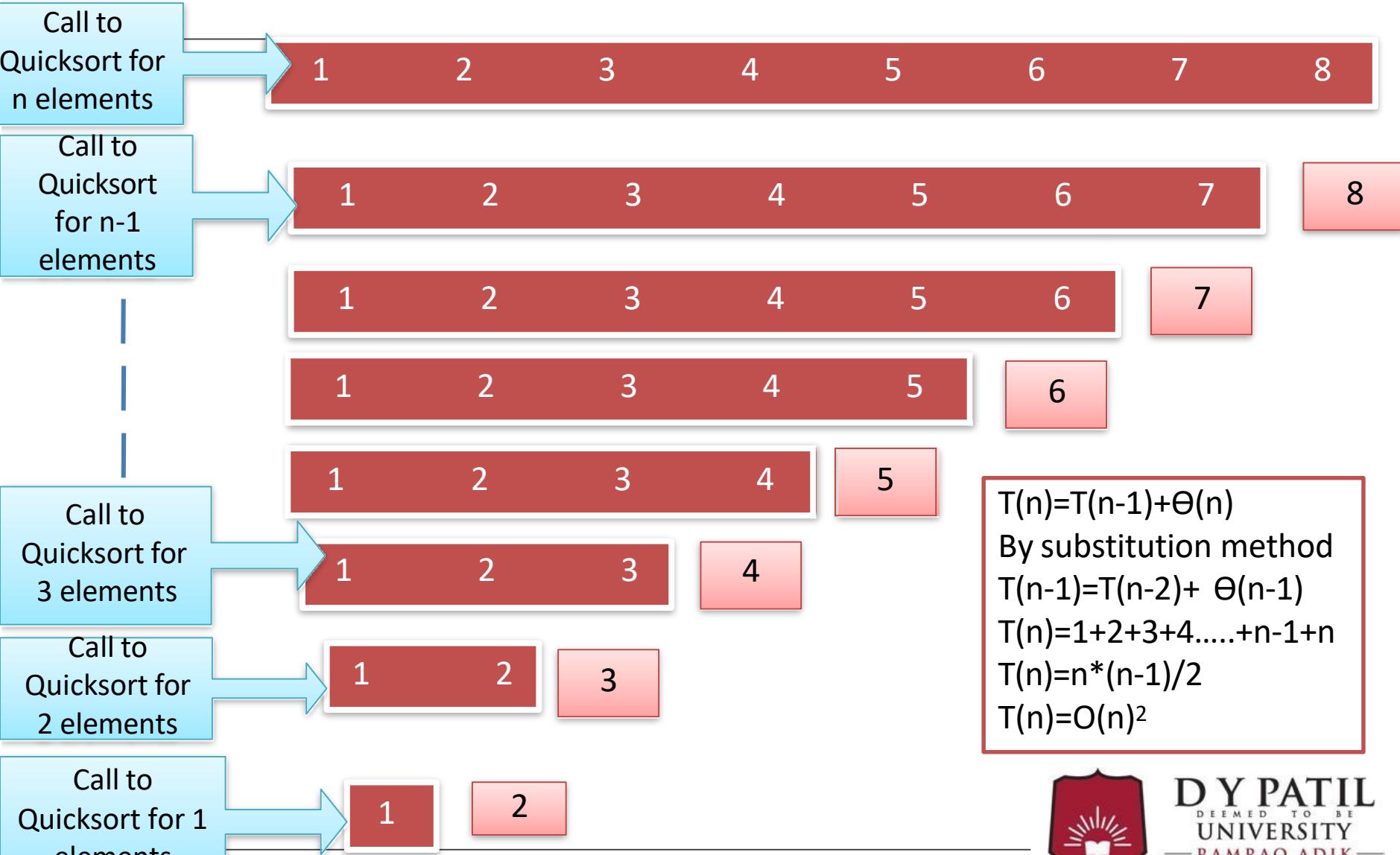
## Quick Sort - Analysis

---

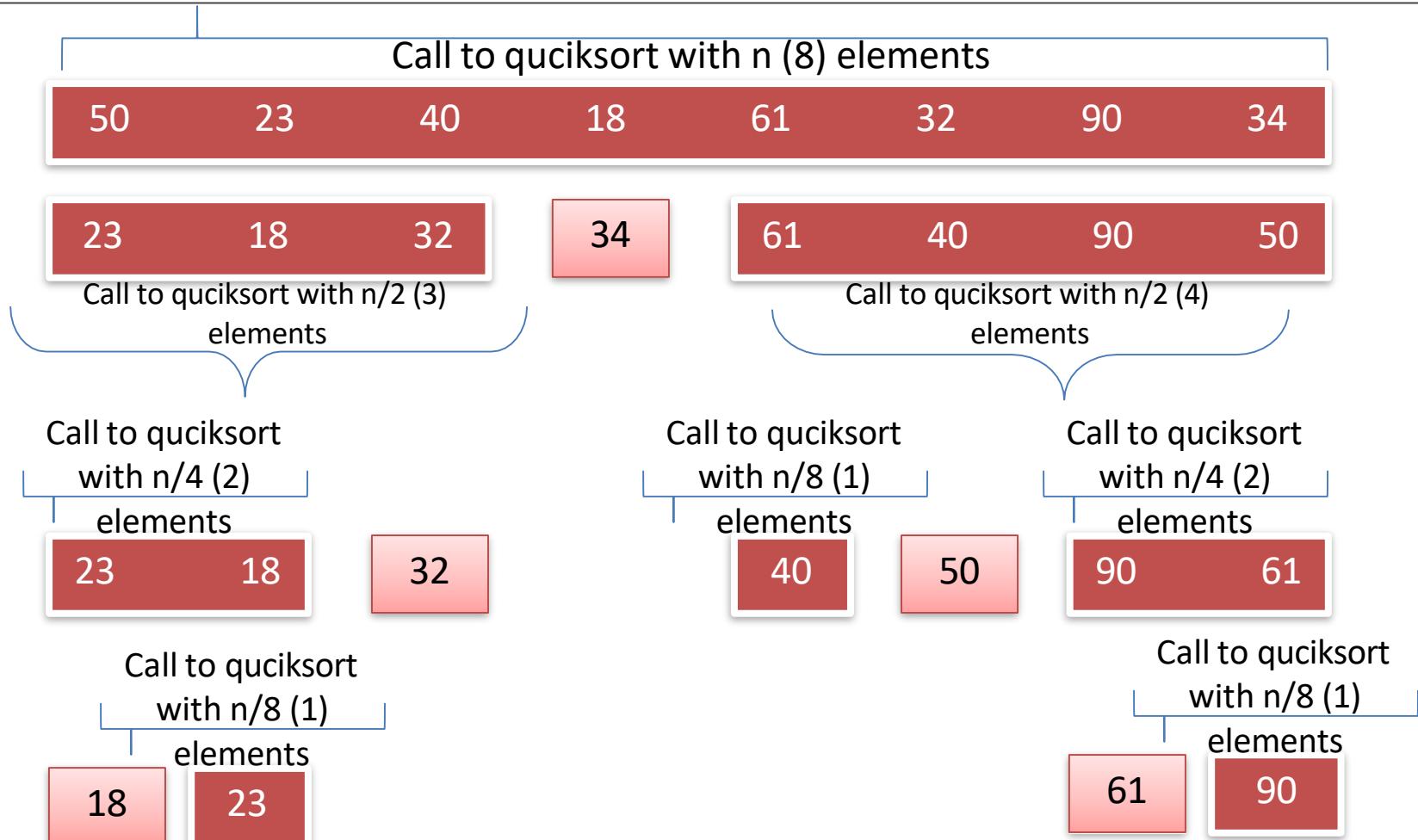
- The complexity of Quick sort depends upon partitioning.
- If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort (Best Case).
- If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort (Worst Case).



## Worst Case



## Best Case



## Best Case

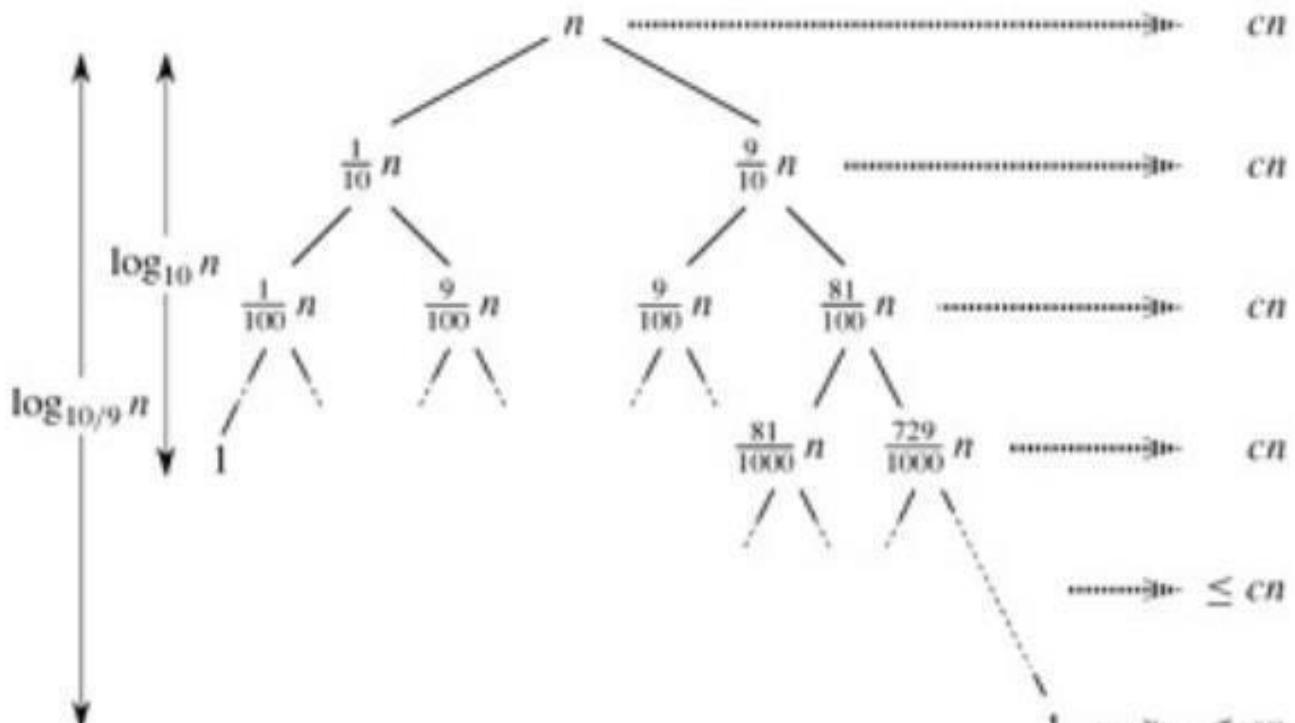
---

- Looking example, every time the array is divide in to half
- Thus,  $T(n)=2T(n/2)+\Theta(n)$
- The above equation is similar to merge sort
- Analysis using master's method
- Case 2 of master's method is application
  - $T(n)=aT(n/b)+f(n)$  where  $a \geq 1$  and  $b > 1$
  - If  $f(n)=\Theta(n^c)$  where  $c = \log_b a$  then  $T(n)=\Theta(n^c \log n)$
  - $a=2, b=2, c=0$ , also  $\log_b a = \log_2 2 = 1$
  - Thus,  $T(n)=\Theta(n^1 \log n)=\Theta(n \log n)$



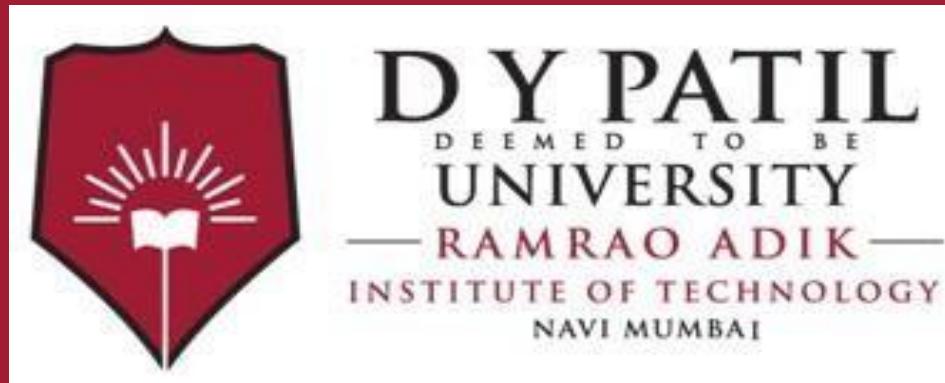
## Average Case

- Suppose on calling quick sort the size of left subarray is  $9n/10$  and right subarray is  $n/10$ .

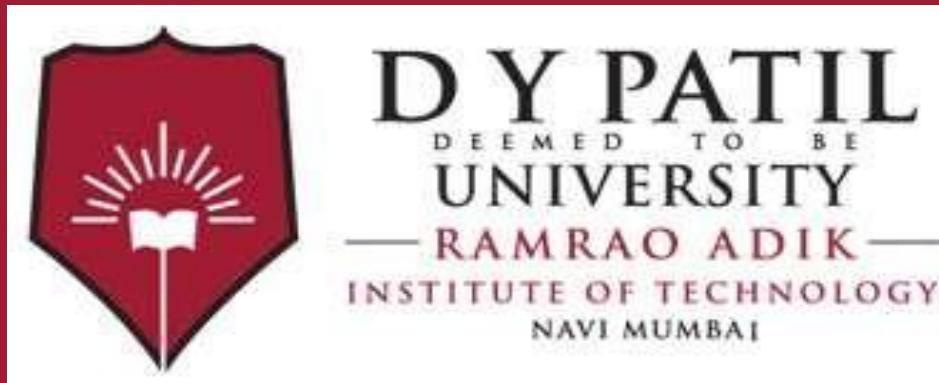


$$O(n \lg n)$$





# Thank You



# Design and Analysis of Algorithms

## Unit 2:

# Greedy Method Approach

# **Index -**

---

Lecture 12 - Introduction to Greedy Approach

Lecture 13 – Knapsack Problem

Lecture 14 – Minimum cost spanning trees: Kruskal’s Algorithm

Lecture 15 – Minimum cost spanning trees: Prim’s Algorithm

Lecture 16 – Single source shortest path

Lecture 17 – Job sequencing with deadlines

---



# Introduction to Greedy Approach



## Greedy Approach

---

- General method: Given  $n$  inputs choose a subset that satisfies some constraints.
- A subset that satisfies the constraints is called a feasible solution.
- A feasible solution that maximises or minimises a given (objective) function is said to be optimal.
- Often it is easy to find a feasible solution but difficult to find the optimal solution.

## Greedy Approach

---

- A greedy approach works in stage
- always makes the choice that seems to be the best at that moment
- it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

## Optimal ??

---

- How do you decide which choice is optimal?
- Based on objective function that needs to be optimized (either maximized or minimized) at a given point.
- A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized.
- The Greedy algorithm has only one shot to compute the optimal solution so that it never goes back and reverses the decision.

## Advantages and Disadvantages

---

- It is quite easy to come up with a greedy algorithm (or even multiple greedy algorithms) for a problem.
- Analyzing the run time for greedy algorithms will generally be much easier than for other techniques (like Divide and conquer).
- For the Divide and conquer technique, it is not clear whether the technique is fast or slow. This is because at each level of recursion the size of gets smaller and the number of sub-problems increases.
- The difficult part is that for greedy algorithms you have to work much harder to understand correctness issues.
- Note: Most greedy algorithms are not correct.



## Example 1

---

- Coin exchange problem,
- optimization criteria— have minimum number of coins
- Eg denomination in Rs.={1,3,4,5,7}
- Get exchange for amount say Rs. 10
- So you will use greedy approach, always take a coin higher denomination possible
- So solution amount =  $10 - 7 - 3 = 0$
- So you got minimum number of coins and feasible as well as optimal solution, other solution is {5,5}



## Example 2

---

- Coin exchange problem,
- optimization criteria— have minimum number of coins
- Eg denomination in rs.={1,3,4,5}
- Get exchange for amount say Rs. 7
- So you will use greedy approach, always take a coin higher denomination possible
- So solution amount =  $7 - 5 - 1 - 1 = 0$
- Here, the feasible solution you got 3 coins in exchange for amount of Rs.7.
- Optimal solution was with 2 coins as {3,4}



## Example 3

---

- Coin exchange problem, optimization criteria– have minimum number of coins
- Eg. denomination in rs.={3,4,5}
- Get exchange for amount say Rs. 7
- So you will use greedy approach, always take a coin higher denomination possible
- So solution amount =  $7 - 5 = 2$
- Here, you are not able to get feasible solution, even though it exist.
- Optimal solution was with 2 coins as {3,4}
- Note: Most greedy algorithms are not correct.



## Summarize

---

- Greedy algorithms are simple and straightforward
- A greedy algorithm always makes the choice that looks best at the moment.
- They are short-sighted in their approach in the sense that they take decisions on the basis of information at hand without worrying about the effect these decisions may have in the future.
- They are easy to invent, easy to implement and most of the time quite efficient.
- Greedy algorithms are used to solve optimization problems.
- That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.



# Knapsack Problem



## Knapsack Problem

---

- Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
- It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.
- It refers to the commonplace problem of packing the most valuable or useful items without overloading the luggage.

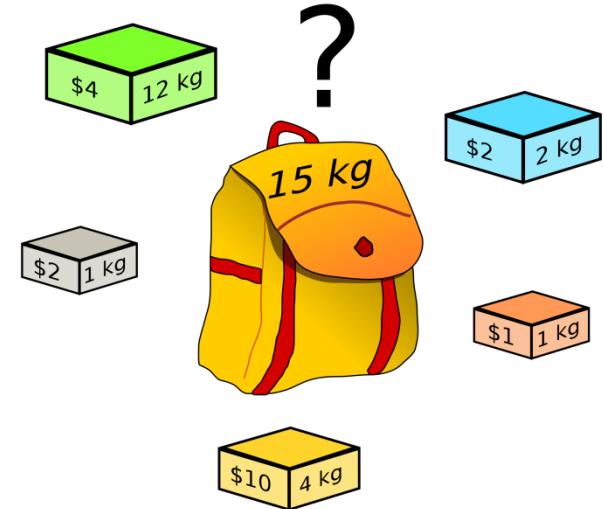


## Knapsack Problem

- Knapsack has 2 variations



Fractional Knapsack  
Greedy Approach



0-1 Knapsack  
Dynamic Programming  
Approach

## Knapsack Problem

---

- Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
- It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.
- It refers to the commonplace problem of packing the most valuable or useful items without overloading the luggage.



## Knapsack Problem

---

- In Fractional Knapsack, we can break items for maximizing the total value of knapsack.
  - This problem in which we can break an item is also called the fractional knapsack problem.
  - The fractional knapsack can be solved using greedy/dynamic programming approach .
- 
- In the 0-1 Knapsack problem, we are not allowed to break items. We either take the whole item or don't take it.
  - 0-1 Knapsack problem, **may** get feasible solution with greedy approach, but with dynamic programming approach surely **gets** the optimal solution.



## Knapsack Problem

---

- In Fractional Knapsack, we can break items for maximizing the total value of knapsack.
- Problem Scenario

A thief is robbing a store and can carry a maximal weight of  $W$  into his knapsack. There are  $n$  items available in the store and weight of  $i^{\text{th}}$  item is  $w_i$  and its profit is  $p_i$ . What items should the thief take?
- Thus, according to problem statement
  - There are  $n$  items
  - Weight of  $i^{\text{th}}$  is  $w_i$ ,  $w_i > 0$
  - profit of  $i^{\text{th}}$  is  $p_i$ ,  $p_i > 0$
  - Capacity of knapsack  $W$



## Knapsack Problem

---

- Since items can be broken down, i.e can be taken in fraction
- $x_i$ , will be between  $0 \leq x_i \leq 1$
- The  $i^{\text{th}}$  contributes the weight  $x_i * w_i$  and profit  $x_i * p_i$
- Objective is:

$$\text{maximize} \sum_{n=1}^n (x_i \cdot p_i)$$

- constraint

$$\sum_{n=1}^n (x_i \cdot w_i) \leq W$$



## Knapsack Problem - Algorithm

---

- Get the ratio of  $p_i / w_i$  for each item in store
- Sort the list of items based on  $p_i / w_i$
- Keep on picking up the item as whole by the time  $\sum w_i \leq W$ , for all items added in knapsack
- If there is a space remaining in knapsack take the fraction of next item, to fill the knapsack completely
- 



## Knapsack Problem - Algorithm

---

### Algorithm: Greedy-Fractional-Knapsack ( $w[1..n]$ , $p[1..n]$ , $W$ )

```
■for i = 1 to n
    do x[i] = 0
■weight = 0
■for i = 1 to n
    if weight + w[i] ≤ W then
        x[i] = 1
        weight = weight + w[i]
    else
        x[i] = (W - weight) / w[i]
        weight = W
        break
■return x
```



## Knapsack Problem - Analysis

---

- 1) Compute the value per pound  $p_i / w_i$  for each item –takes linear time  $O(n)$
  - 2) Sort the list based on  $p_i / w_i$  – sorting takes  $O(n \log n)$  time
  - 3) The thief begins by taking, as much as possible, of the item with the greatest value per pound
  - 4) If the supply of that item is exhausted before filling the knapsack he takes, as much as possible, of the item with the next greatest value per pound
  - 5) Repeat ( 3 - 4) until his knapsack becomes full – step 3 and 4 takes linear time
- 
- time complexity  $T(n)=$ ratio calculation + sorting list + adding individual item to list
  - $T(n)= O(n)+O(n \log n)+O(n)$
  
  - Thus, by sorting the items by value per pound the greedy algorithm runs in  $O( n \log n )$  time



## Knapsack Problem - Example

---

**Example:** Consider Knapsack Capacity  $W = 30$

ITEM	A	B	C	D
VALUE	50	140	60	60
SIZE	5	20	10	12
RATIO	10	7	6	5

### Solution:

All of A, all of B, and  $((30-25)/10)$  of C (and none of D)

$$\text{Size: } 5 + 20 + 10 * (5/10) = 30$$

$$\text{Value: } 50 + 140 + 60 * (5/10) = 190 + 30 = 220$$

**Time:**  $\Theta(n)$ , if already sorted



## Knapsack Problem - Example

---

**Example:** Consider Knapsack Capacity  $W = 25$

ITEM	A	B	C	D
VALUE	12	9	9	5
SIZE	24	10	10	7
RATIO	0.5	0.9	0.9	0.714

**Solution:**

Find which items among the given are added to knapsack ?



# Minimum cost spanning trees: Kruskal's Algorithm



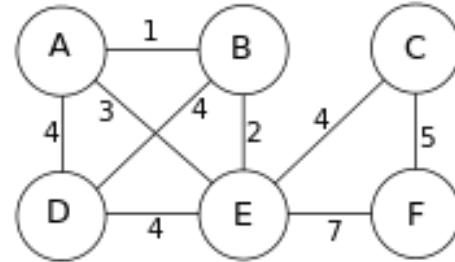
## Minimum cost spanning trees: Kruskal's Algorithm

---

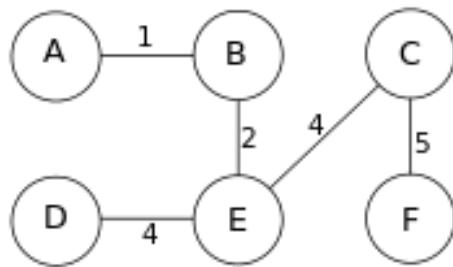
- A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.
- it is a spanning tree whose sum of edge weights is as small as possible.
- there might be several spanning trees possible.
- If the graph has  $n$  vertices then it has  $n-1$  edges.



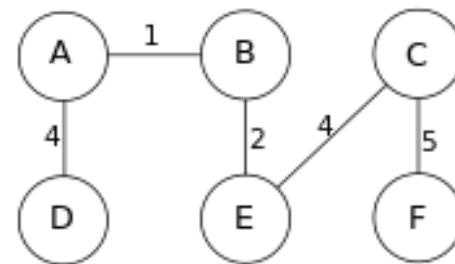
## Minimum cost spanning trees: Kruskal's Algorithm



Graph G



Minimum spanning tree 1



Minimum spanning tree 2

## Minimum cost spanning trees: Kruskal's Algorithm

---

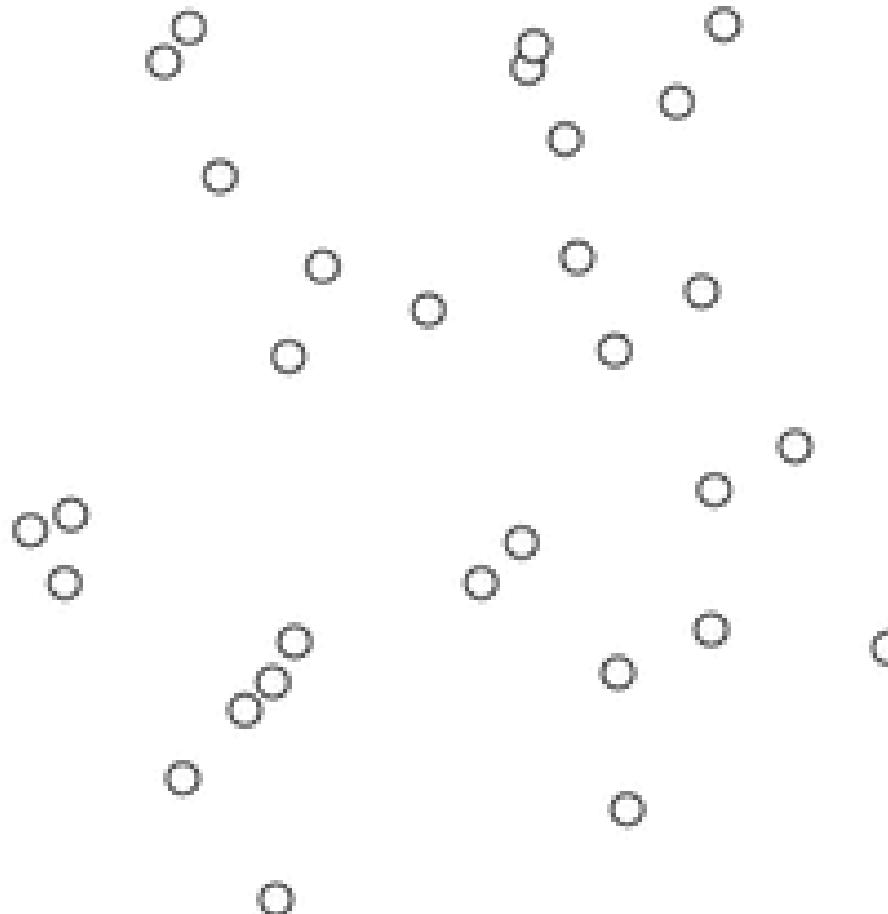
- Kruskal's algorithm finds a minimum spanning forest of an undirected edge-weighted graph.
- A minimum spanning tree of a connected graph is a subset of the edges that forms a tree that includes every vertex, where the sum of the weights of all the edges in the tree is minimized.
- It is a greedy algorithm in graph theory as in each step it adds the next lowest-weight edge that will not form a cycle to the minimum spanning forest.



## Minimum cost spanning trees: Kruskal's Algorithm-Demo

---

- A demo for Kruskal's algorithm on a complete graph with weights based on Euclidean distance.



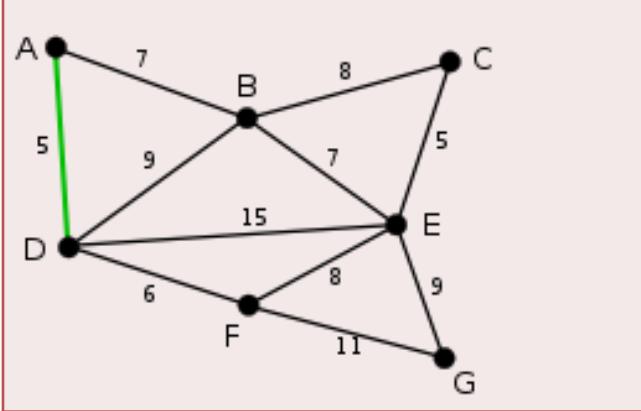
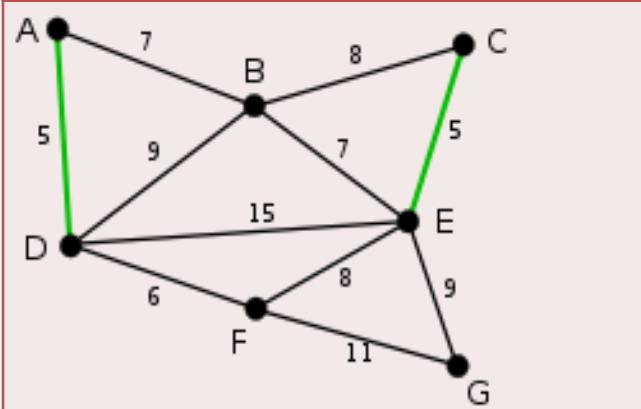
## Minimum cost spanning trees: Kruskal's Algorithm

---

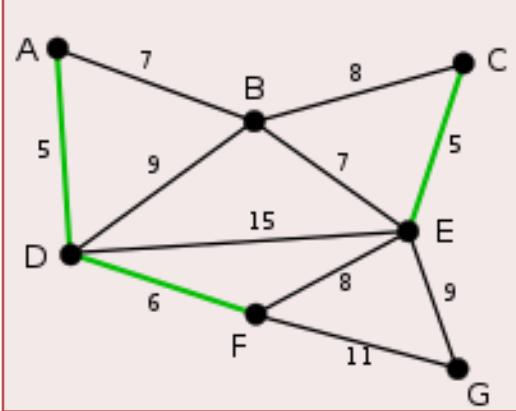
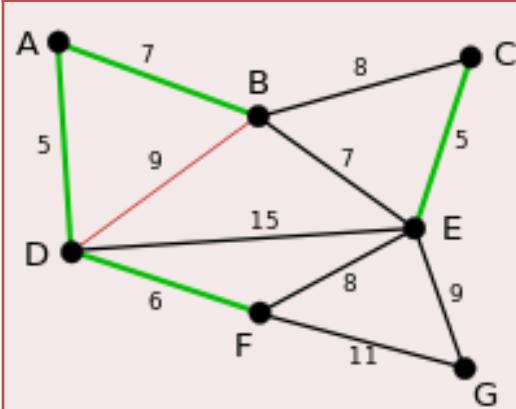
- Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree.
- Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.
- Algorithm Steps:
  1. Sort the graph edges with respect to their weights.
  2. Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
  3. Only add edges which doesn't form a cycle , edges which connect only disconnected components.



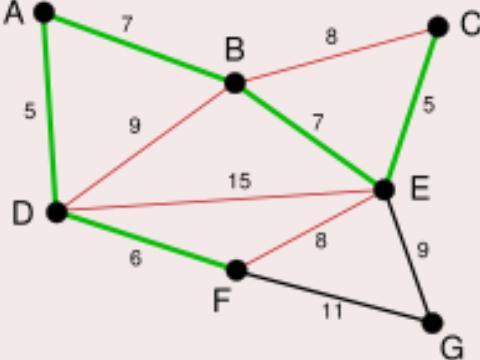
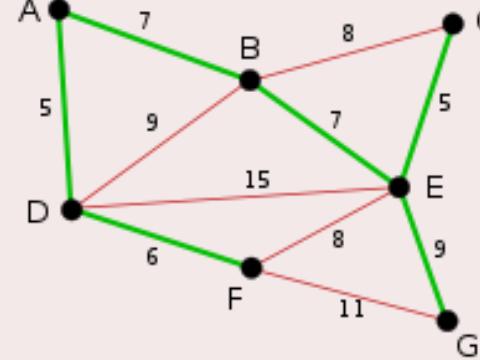
## Minimum cost spanning trees: Kruskal's Algorithm

Image	Description
	<p>AD and CE are the shortest edges, with length 5, and AD has been arbitrarily chosen, so it is highlighted.</p>
	<p>CE is now the shortest edge that does not form a cycle, with length 5, so it is highlighted as the second edge.</p>

## Minimum cost spanning trees: Kruskal's Algorithm

Image	Description
	<p>The next edge, <b>DF</b> with length 6, is highlighted using much the same method.</p>
	<p>The next-shortest edges are <b>AB</b> and <b>BE</b>, both with length 7. <b>AB</b> is chosen arbitrarily, and is highlighted. The edge <b>BD</b> has been highlighted in red, because there already exists a path (in green) between <b>B</b> and <b>D</b>, so it would form a cycle (<b>ABD</b>) if it were chosen.</p>

## Minimum cost spanning trees: Kruskal's Algorithm

Image	Description
 <p>A graph with 7 vertices (A, B, C, D, E, F, G) and 9 edges. The edges and their weights are: AB (7), AC (5), AD (5), BD (9), BE (7), BC (8), DE (15), DF (6), and EG (9). The edges BE and BC are highlighted in green, while all other edges are highlighted in red.</p>	<p>The process continues to highlight the next-smallest edge, <b>BE</b> with length 7. Many more edges are highlighted in red at this stage: <b>BC</b> because it would form the loop <b>BCE</b>, <b>DE</b> because it would form the loop <b>DEBA</b>, and <b>FE</b> because it would form <b>FEBAD</b>.</p>
 <p>The same graph as above, but now the edge <b>EG</b> is highlighted in green, while all other edges are red.</p>	<p>Finally, the process finishes with the edge <b>EG</b> of length 9, and the minimum spanning tree is found.</p>



## Minimum cost spanning trees: Kruskal's Algorithm

---

**Mst-Kruskal( $G, w$ )**

1.  $F := \emptyset$
2. **for each**  $v \in G.V$
3.     MAKE-SET( $v$ )
4. sort the edges of  $G.E$  into non-decreasing order by weight  $w$
5. **for each**  $(u, v)$  in  $G.E$  ordered by weight( $u, v$ ), increasing
6. **if** FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ ) **then**
7.      $F := F \cup \{(u, v)\}$
8.     UNION(FIND-SET( $u$ ), FIND-SET( $v$ ))
9. **return**  $F$



## Minimum cost spanning trees: Kruskal's Algorithm- Analysis

**Mst-Kruskal( $G, w$ )**

1.  $F := \emptyset$
2. **for each**  $v \in G.V$
3.     MAKE-SET( $v$ )
4. sort the edges of  $G.E$  into non-decreasing order by weight  $w$
5. **for each**  $(u, v)$  **in**  $G.E$  ordered by weight( $u, v$ ), increasing
6. **if** FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ ) **then**
7.      $F := F \cup \{(u, v)\}$
8.     UNION(FIND-SET( $u$ ), FIND-SET( $v$ ))
9. **return**  $F$

$O(1)$

A very slow growing function

$O(E \log E)$

$O(E \log V)$

- $T(n) = O(1) + O(E \log E) + O(E \log V)$
- So overall complexity is  $O(E \log E + E \log V)$  time.
- The value of  $E$  can be atmost  $O(V^2)$ ,
- so  $O(\log V)$  are  $O(\log E)$  same.
- Therefore, overall time complexity is  $O(E \log E)$  or  $O(E \log V)$

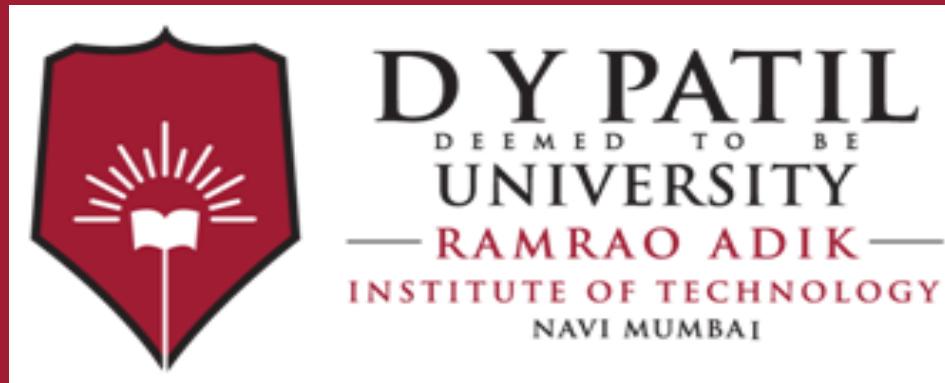


## Reading Material

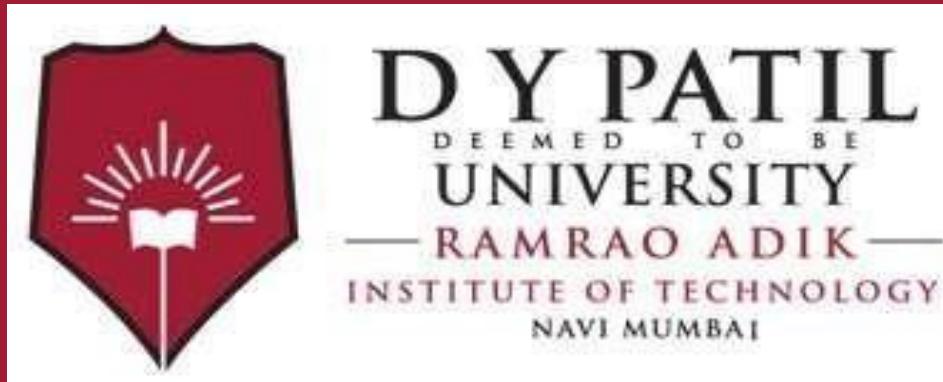
---

- Greedy method -  
<https://www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/tutorial/>
- Kruskal algo : <https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/tutorial/>
- Prims Algo  
<https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/>
- Dijktra;s  
<https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>  
<https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/>
  
- Video-
- Kruskal compexity: [https://youtu.be/3rrNH\\_AizMA](https://youtu.be/3rrNH_AizMA)
- Dijiktra's : <https://youtu.be/ba4YGd7S-TY>
- Prims : <https://youtu.be/eB61LXLZVqs>





# Thank You



# Design and Analysis of Algorithms

## Unit 2:

# Greedy Method Approach

## Index -

---

Lecture 12 - Introduction to Greedy Approach

Lecture 13 – Knapsack Problem

Lecture 14 – Minimum cost spanning trees: Kruskal’s Algorithm

Lecture 15 – Minimum cost spanning trees: Prim’s Algorithm

Lecture 16 – Single source shortest path

Lecture 17 – Job sequencing with deadlines

---



# Minimum cost spanning trees: Prim's Algorithm



# Prims Algorithm

---

- Prim's Algorithm is a famous greedy algorithm.
- It is used for finding the Minimum Spanning Tree (MST) of a given graph.
- To apply Prim's algorithm, the given graph must be weighted, connected and undirected.

## Prims Algorithm

---

Step 1: Select a starting vertex

Step 2: Repeat Steps 3 and 4 until there are fringe vertices

Step 3: Select an edge e connecting the tree vertex and fringe vertex that has minimum weight

Step 4: Add the selected edge and the vertex to the minimum spanning tree T  
[END OF LOOP]

Step 5: EXIT

# **Prim's Algorithm Implementation-**

---

## **Step-01:**

Randomly choose any vertex.

The vertex connecting to the edge having least weight is usually selected.

## **Step-02:**

Find all the edges that connect the tree to new vertices.

Find the least weight edge among those edges and include it in the existing tree.

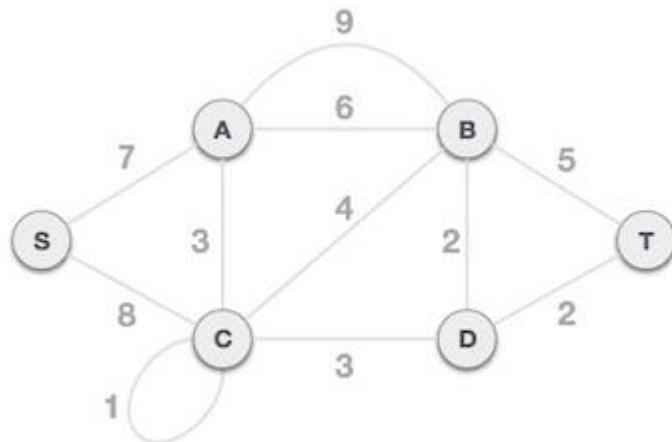
If including that edge creates a cycle, then reject that edge and look for the next least weight edge.

## **Step-03:**

Keep repeating step-02 until all the vertices are included and Minimum Spanning Tree (MST) is obtained

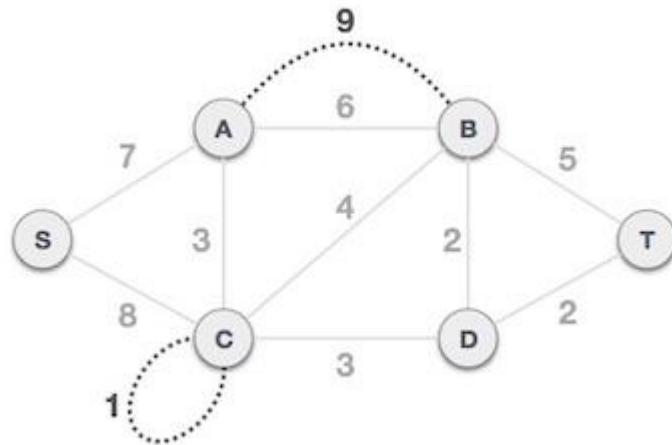
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –(Example 1)

---

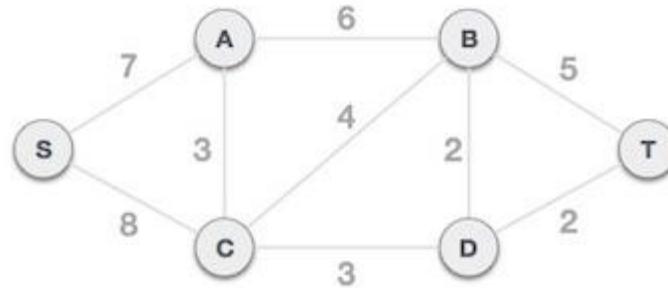


## Step 1 - Remove all loops and parallel edges

---



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.



## Step 2 - Choose any arbitrary node as root node

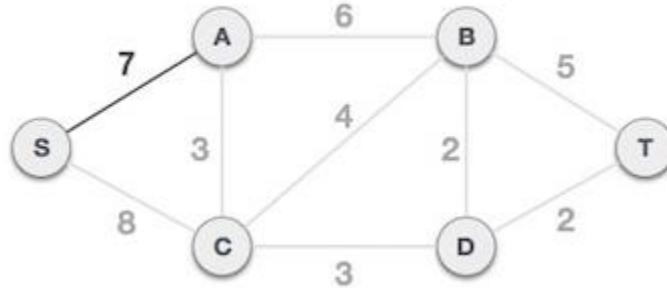
---

In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node.

So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

## Step 3 - Check outgoing edges and select the one with less cost

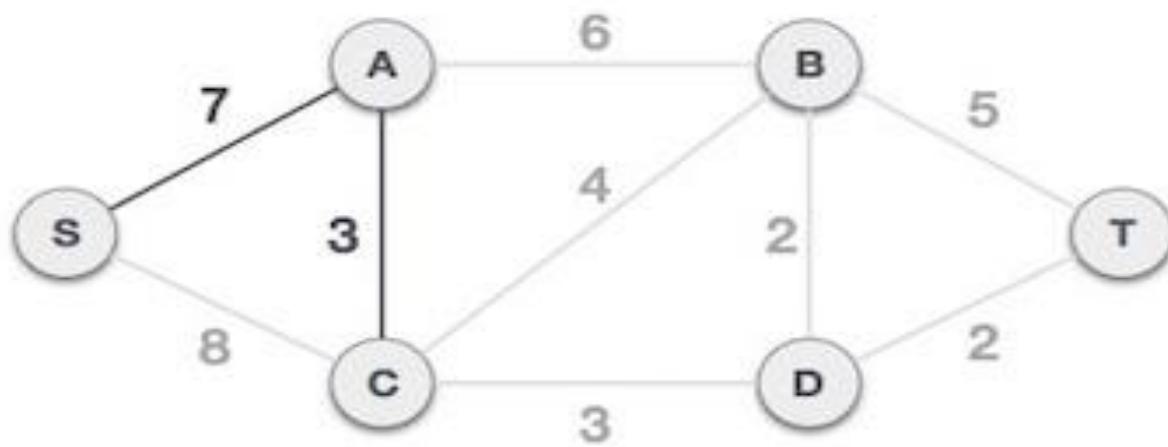
---



After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.

Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.

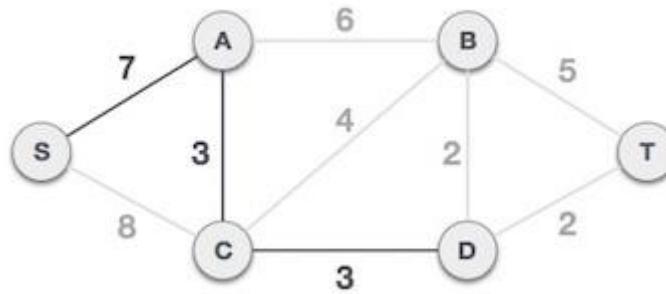
---



## Contd..

---

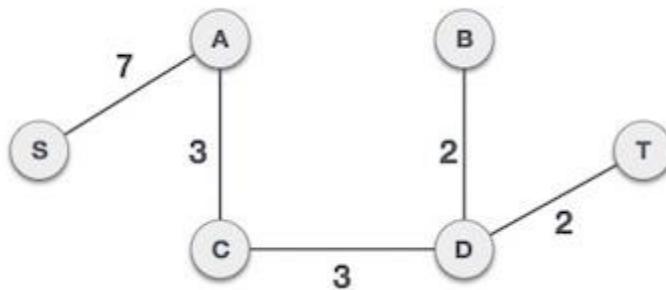
After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



## Final Step

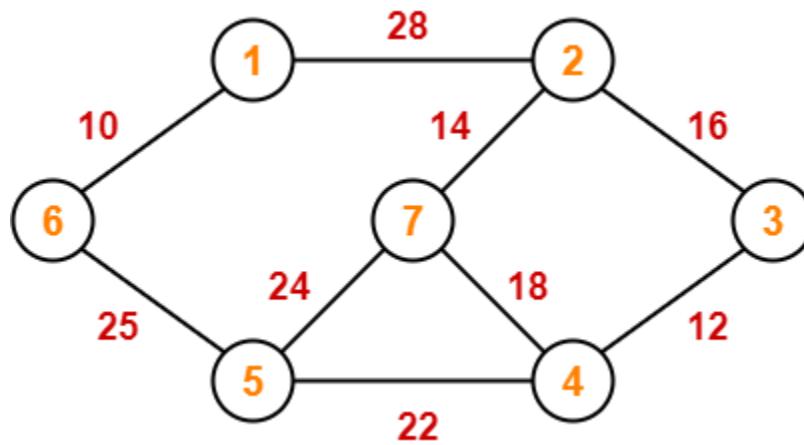
---

After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



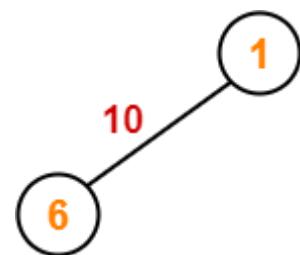
## Example 2

---



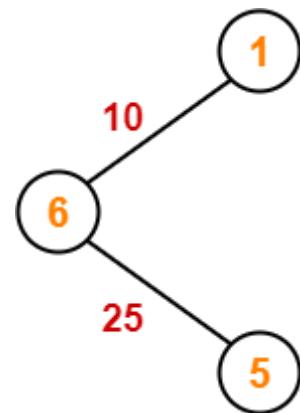
## Step 1:

---



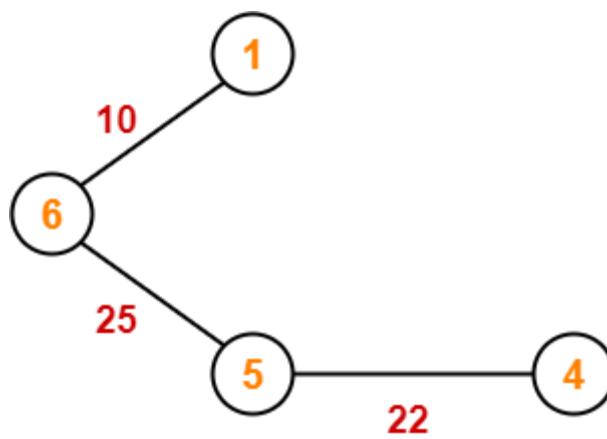
## Step 2

---



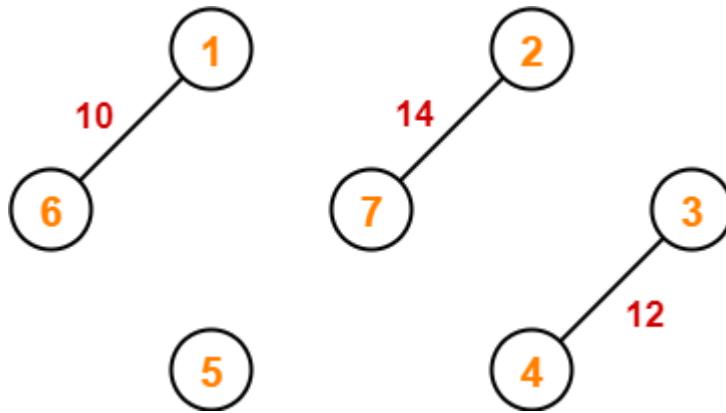
## Step 03

---



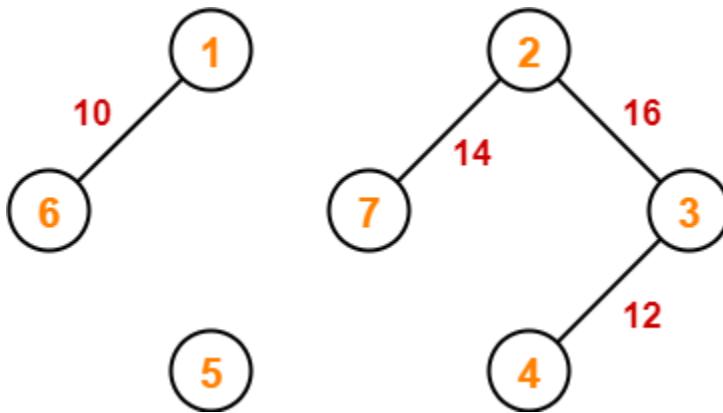
## Step 04

---



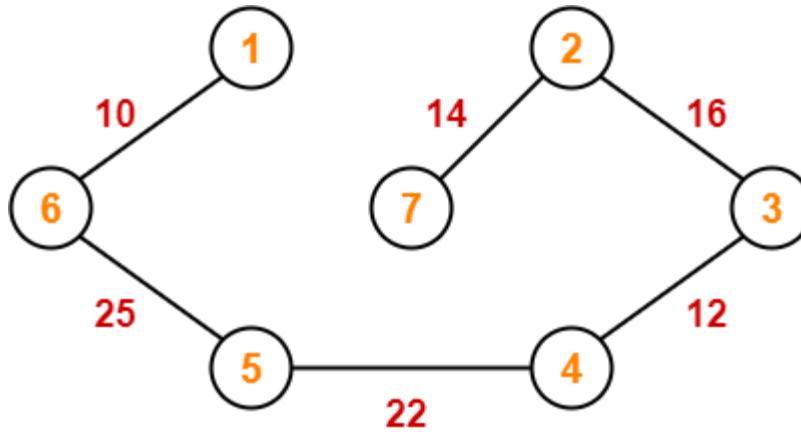
## Step 05

---



## Step 06

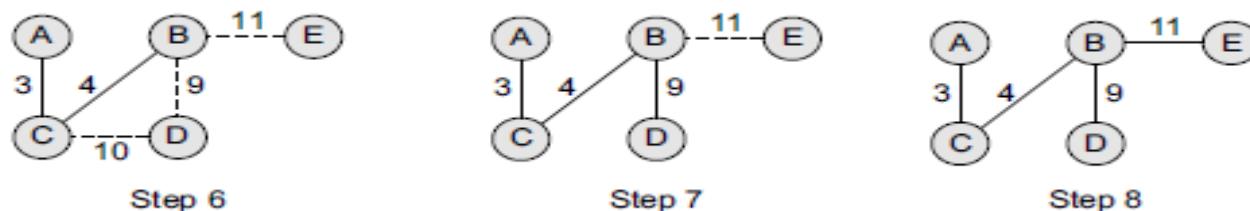
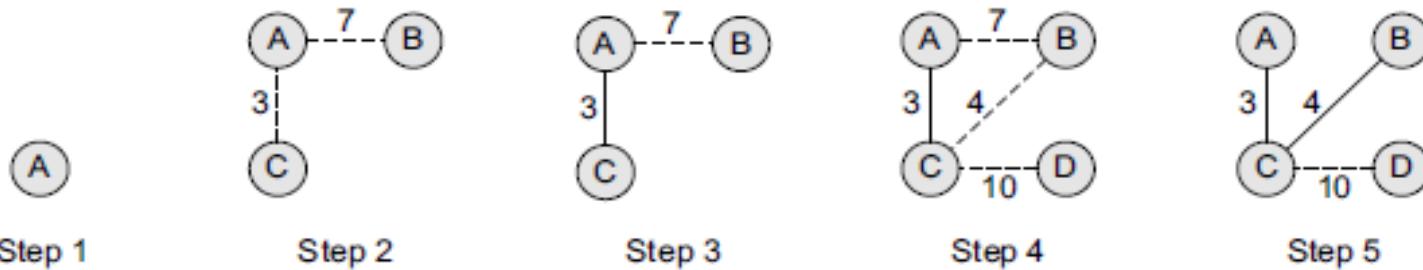
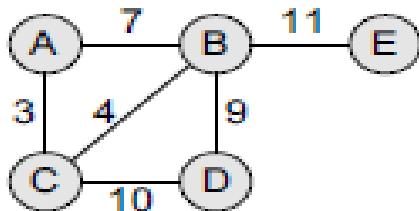
---



Since all the vertices have been included in the MST, so we stop.

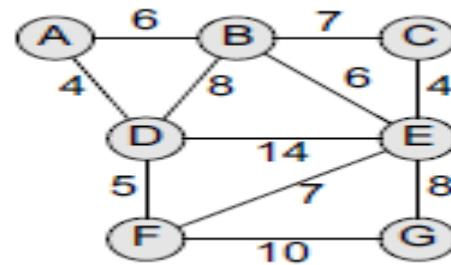
Now, Cost of Minimum Spanning Tree  
= Sum of all edge weights  
=  $10 + 25 + 22 + 12 + 16 + 14$   
= 99 units  
‘

## Example 3



## Solve

---



# Application of Minimum spanning Tree

---

The biggest application of minimum cost spanning trees is connecting multiple nodes to a single network with the smallest cost.

This can be seen with computers and a network, using wire to connect each computer, or consider a single phone line that you want multiple phones to be connected to via physical wire. You would want to minimize the costs to connect each computer or phone to the network.

## Kruskal's Algorithm is preferred when-

---

- The graph is sparse.
- There are less number of edges in the graph like  $E = O(V)$
- The edges are already sorted or can be sorted in linear time.

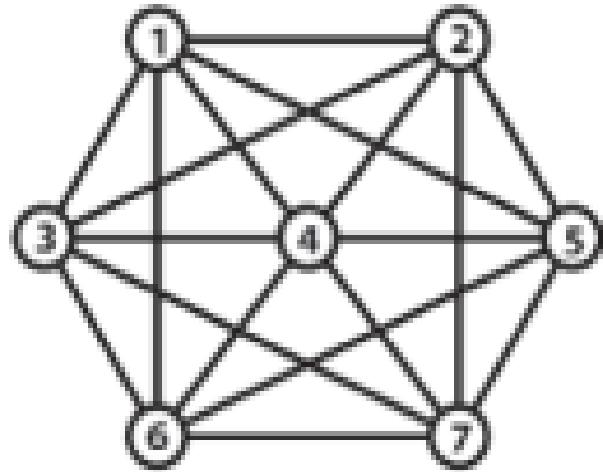
Prim's Algorithm is preferred when-

---

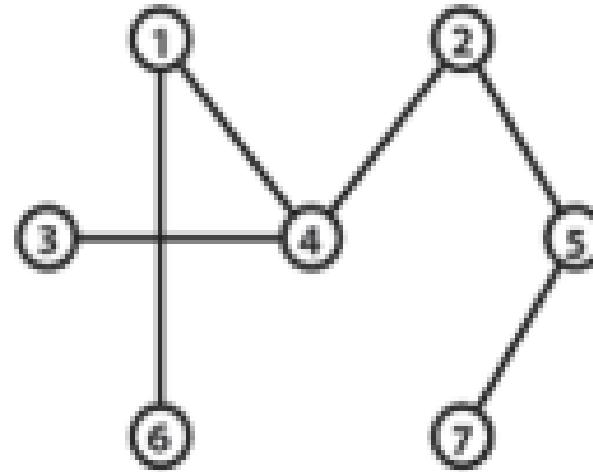
- The graph is dense.
- There are large number of edges in the graph like  $E = O(V^2)$ .

## Dense v/s Sparse

---



Dense



Sparse

# Difference between Prims and Kruskal's algorithm

---

Prim's Algorithm	Kruskal's Algorithm
The tree that we are making or growing always remains connected.	The tree that we are making or growing usually remains disconnected.
Prim's Algorithm grows a solution from a random vertex by adding the next cheapest vertex to the existing tree.	Kruskal's Algorithm grows a solution from the cheapest edge by adding the next cheapest edge to the existing tree / forest.
Prim's Algorithm is faster for dense graphs.	Kruskal's Algorithm is faster for sparse graphs.

## Minimum cost spanning trees: Prim's Algorithm

- Algorithm
  - 1) Create a set mstSet that keeps track of vertices already included in MST.
  - 2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
  - 3) While mstSet doesn't include all vertices
    - ....a) Pick a vertex u which is not there in mstSet and has minimum key value.
    - ....b) Include u to mstSet.
    - ....c) Update key value of all adjacent vertices of u.
      - To update the key values, iterate through all adjacent vertices.
      - For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v
- The idea of using key values is to pick the minimum weight edge from cut.
- The key values are used only for vertices which are not yet included in MST.
- The key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.



## Minimum cost spanning trees: Prim's Algorithm

- Algorithm

- 1) Create a set mstSet that keeps track of vertices already included in MST. O(1)
- 2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first. O(V)
- 3) While mstSet doesn't include all vertices
  - a) Pick a vertex u which is not there in mstSet and has minimum key value.
  - b) Include u to mstSet. O(V)
  - c) Update key value of all adjacent vertices of u.
    - To update the key values, iterate through all adjacent vertices.
    - For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v O(V)\*  
+O(1)  
+O(V)

- If used adjacency matrix for implementation
- $T(n)=O(1)+O(V)+O(V)*(O(V)+O(1)+O(V))$
- $T(n)=O(V)+O(V)*(2*O(V))=O(V)+O(V^2) = O(V^2)$



## Minimum cost spanning trees: Prim's Algorithm

---

- If used adjacency matrix for representation of input graph
  - $T(n)=O(1)+O(V)+O(V)*(O(V)+O(1)+O(V))$
  - $T(n)=O(V)+O(V)*(2*O(V))=O(V)+O(V^2) = O(V^2)$
- If the input graph is represented using adjacency list, then the time complexity of Prim's algorithm can be reduced to  $O(E \log V)$  with the help of binary heap.



## Lecture 16

---

# Single source shortest path



## Single Source Shortest Path - Dijkstra's Algorithm

---

- With Dijkstra's Algorithm, you can find the shortest path between nodes in a graph
- Particularly, you can find the shortest path from a node (called the "source node") to all other nodes in the graph, producing a shortest-path tree.
- Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree.
- Like Prim's MST, we generate a *SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree.
- Similarly in dijktra's at every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

## Single Source Shortest Path - Dijkstra's Algorithm

---

- Dijkstra's Algorithm can only work with graphs that have positive weights. This is because, during the process, the weights of the edges have to be added to find the shortest path.
- If there is a negative weight in the graph, then the algorithm will not work properly.
- Once a node has been marked as "visited", the current path to that node is marked as the shortest path to reach that node.
- And negative weights can alter this if the total weight can be decremented after this step has occurred.
- To deal with graph with negative weighted edges, we have bellman ford algorithm, in dynamic programming approach



## Single Source Shortest Path - Dijkstra's Algorithm

```
DIJKSTRA( $G, w, s$ )
1      INITIALIZE-SINGLE-SOURCE( $G, s$ )
2       $S \leftarrow \emptyset$ 
3       $Q \leftarrow V[G]$ 
4      while  $Q \neq \emptyset$  do
5           $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
6           $S \leftarrow S + \{u\}$ 
7          for each vertex  $v$  adjacent to  $u$  do
8              RELAX( $u, v, w$ )
```

- Time complexity
- $T(n) = O(V^2)$
- If the input graph is represented using adjacency list, it can be reduced to  $O(E \log V)$  with the help of binary heap.



## Single Source Shortest Path - Dijkstra's Algorithm

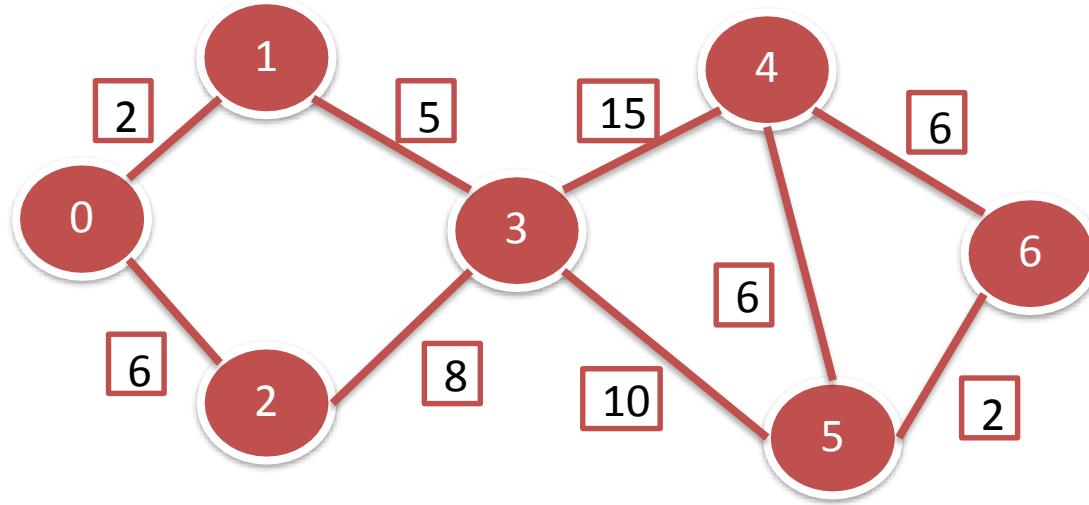
```
INITIALIZE-SINGLE-SOURCE(G, s)
1      for each vertex v in V[G]
2          do  $d[v] \leftarrow \infty$ 
3           $\pi[v] \leftarrow \text{NIL}$ 
4       $d[s] 0$ 
```

```
RELAX( $u, v, w$ )
1      if  $d[v] > d[u] + w(u, v)$  then
2           $d[v] \leftarrow d[u] + w(u, v)$ 
3           $\pi[v] \leftarrow u$ 
```



## Single Source Shortest Path - Dijkstra's Algorithm - Example

- Start with vertex with minimum distance  $d[i]=\min$  and not visited  $v[i]=F$ , mark it visited and update distance of adjacent node if  $v[j]>v[i]+w[i,j]$
- Repeat above steps, till we have all connected nodes visited as true,  $v[i]=T$



## Single Source Shortest Path - Dijkstra's Algorithm

```
INITIALIZE-SINGLE-SOURCE(G, s)
1      for each vertex v in V[G]
2          do  $d[v] \leftarrow \infty$ 
3           $\pi[v] \leftarrow \text{NIL}$ 
4       $d[s] 0$ 
```

O(V)

```
RELAX( $u, v, w$ )
1      if  $d[v] > d[u] + w(u, v)$  then
2           $d[v] \leftarrow d[u] + w(u, v)$ 
3           $\pi[v] \leftarrow u$ 
```

O(1)



# Single Source Shortest Path - Dijkstra's Algorithm

```
DIJKSTRA( $G, w, s$ )
1   INITIALIZE-SINGLE-SOURCE( $G, s$ )
2    $S \leftarrow \emptyset$ 
3    $Q \leftarrow V[G]$ 
4   while  $Q \neq \emptyset$  do
5        $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
6        $S \leftarrow S + \{u\}$ 
7       for each vertex  $v$  adjacent to  $u$  do
8           RELAX( $u, v, w$ )
```

$O(V)$   
 $O(1)$   
 $O(V)$

$O(V) *$   
 $O(\log V)$   
+  $O(1)$   
+  $O(V)$

- Time complexity
- $T(n)=O(V^2)$
- If the input graph is represented using adjacency list, it can be reduced to  $O(E \log V)$  with the help of binary heap.

# Job sequencing with deadlines



## Job sequencing with deadlines

---

- problem consists of n jobs each associated with a deadline and profit and our objective is to earn maximum profit.
- We will earn profit only when job is completed on or before deadline.
- We assume that each job will take unit time to complete.
- The objective is to earn maximum profit when only one job can be scheduled or processed at any given time.
- We will greedy approach, to earn more profit, will try to complete job with more profit first.



## Job sequencing with deadlines - Algorithm

---

- 1) Sort all jobs in decreasing order of profit.
- 2) Initialize the result sequence as first job in sorted jobs.
- 3) Do following for remaining n-1 jobs  
    If the current job can fit in the current result sequence  
        without missing the deadline  
        add current job to the result.  
    else  
        ignore the current job.



## Job sequencing with deadlines

---

index	1	2	3	4	5
JOB	j1	j2	j3	j4	j5
DEADLINE	2	1	4	4	1
PROFIT	60	100	20	40	20

- Sort the job in the decreasing order of profit

index	1	2	3	4	5
JOB	j2	j1	j4	j3	j5
DEADLINE	1	2	4	4	1
PROFIT	100	60	40	20	20



## Job sequencing with deadlines

---

- Looking at the jobs we can say the max deadline value is 3. So,  $d_{max} = 3$
- As  $d_{max} = 4$  so we will have THREE slots to keep track of free time slots. Set the time slot status to EMPTY

time slot	1	2	3	4
status	EMPTY	EMPTY	EMPTY	EMPTY

- Number of jobs are 5, so  $n=5$



## Job sequencing with deadlines

---

index	1	2	3	4	5
JOB	j2	j1	j4	j3	j5
DEADLINE	1	2	4	4	1
PROFIT	100	60	40	20	20

- Select first j2, check slot 1 is empty or any other slot before slot 1, if yes add to table of job completed

time slot	1	2	3	4
status	j2	EMPTY	EMPTY	EMPTY



## Job sequencing with deadlines

---

index	1	2	3	4	5
JOB	j2	j1	j4	j3	j5
DEADLINE	1	2	4	4	1
PROFIT	100	60	40	20	20

- Select j1, check slot 2 is empty or any other slot before slot 2, if yes add to table of job completed

time slot	1	2	3	4
status	j2	j1	EMPTY	EMPTY

## Job sequencing with deadlines

---

index	1	2	3	4	5
JOB	j2	j1	j4	j3	j5
DEADLINE	1	2	2	3	1
PROFIT	100	60	40	20	20

- Select j4, check slot 4 is empty or any other slot before slot 4, if yes add to table of job completed, otherwise discard it

time slot	1	2	3	4
status	j2	j1	EMPTY	j4



## Job sequencing with deadlines

---

index	1	2	3	4	5
JOB	j2	j1	j4	j3	j5
DEADLINE	1	2	2	3	1
PROFIT	100	60	40	20	20

- Select j3, check slot 4 is empty or any other slot before slot 4, if yes add to table of job completed, otherwise discard it
- Since slot 3 as it is empty.

time slot	1	2	3	4
status	j2	j1	j3	j4



## Job sequencing with deadlines

---

index	1	2	3	4	5
JOB	j2	j1	j4	j3	j5
DEADLINE	1	2	2	3	1
PROFIT	100	60	40	20	20

- Select j5, check slot 1 is empty or any other slot before slot 1, if yes add to table of job completed, otherwise discard it
- Since all slots are full, stop processing.

time slot	1	2	3	4
status	j2	j1	j3	j4



## Job sequencing with deadlines

---

- Our objective is to select jobs that will give us higher profit, without missing the deadline
- Total profit earned: profit =  $p[2] + p[1] + p[3] + p[4]$   
=  $100 + 60 + 40 + 20$   
= 220

time slot	1	2	3	4
status	j2	j1	j3	j4



## Job sequencing with deadlines- Solve

Task	Deadline	Profit
T1	7	15
T2	2	20
T3	5	30
T4	3	18
T5	4	18
T6	5	10
T7	2	23
T8	7	16
T9	3	25



Sort as per the profit

Verify :  
Answer = 147

Task	Deadline	Profit
T3	5	30
T9	3	25
T7	2	23
T2	2	20
T4,T5	3,4	18,18
T8	7	16
T1	7	15
T6	5	10



## Job sequencing with deadlines- Solve

Task	Deadline	Profit
T3	5	30
T9	3	25
T7	2	23
T2	2	20
T4,T5	3,4	18,18
T8	7	16
T1	7	15
T6	5	10

Sort as per the profit

time slot	status	Profit
1	T2	20
2	T7	23
3	T9	25
4	T5	18
5	T3	30
6	T1	15
7	T8	16
	<b>Total</b>	<b>147</b>

## Job sequencing with deadlines- Analysis

---

- Complexity :  $O(n^2)$
- How?
- Sort the job based on decreasing order of profit -  $O(n \log n)$
- Then pick the job, from highest profit to lowest profit, one by one -  $O(n)$ 
  - For each find empty slot, we can have maximum  $n$  slots.
- Thus complexity :  $T(n) = O(n^2)$

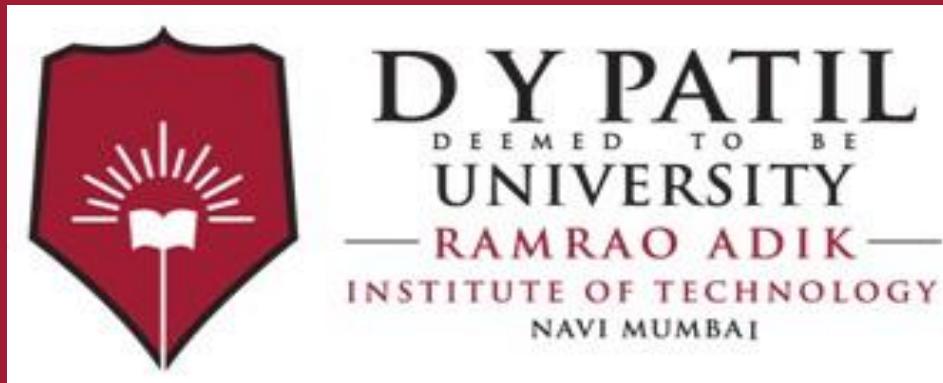


## Reading Material

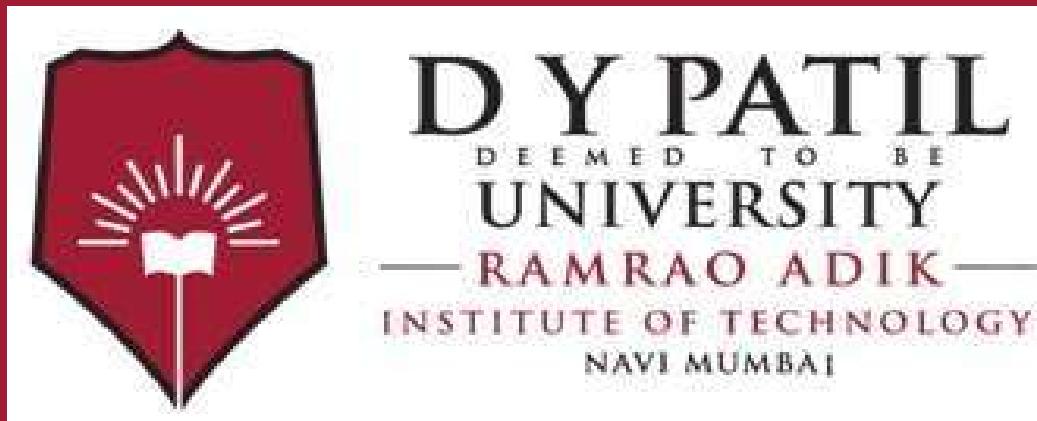
---

- Greedy method -  
<https://www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/tutorial/>
- Kruskal algo : <https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/tutorial/>
- Prims Algo  
<https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/>
- Dijktra;s  
<https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>  
<https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/>
  
- Video-
- Kruskal compexity: [https://youtu.be/3rrNH\\_AizMA](https://youtu.be/3rrNH_AizMA)
- Dijiktra's : <https://youtu.be/ba4YGd7S-TY>
- Prims : <https://youtu.be/eB61LXLZVqs>





# Thank You



# Design and Analysis of Algorithms

## Unit 3: Dynamic Programming Approach

## Index -

---

Lecture 18 - Introduction to Dynamic Programming Approach

Lecture 19 – Multistage Graphs

Lecture 20 – Single Source Shortest Path

Lecture 21- All Pair Shortest Path

---



# Introduction to Dynamic Programming Approach



# Dynamic Programming Approach

---

- Dynamic Programming is also used in optimization problems.
- The core idea of Dynamic Programming is to avoid repeated work by remembering partial results and this concept finds its application in a lot of real life situations.
- Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.
- Dynamic Programming is used when the subproblems are not independent, e.g. when they share the same subproblems.
- Dynamic Programming is a **Bottom-up approach**- we solve all possible small problems and then combine to obtain solutions for bigger problems.
- Dynamic Programming is a powerful technique that allows one to solve different types of problems in time  $O(n^2)$  or  $O(n^3)$  for which a naive approach would take exponential time.



# Dynamic Programming Approach (cont..)

---

- Dynamic Programming uses concept of Memoization.
- Memoization means memorizing the results of some specific states, which can then be later accessed to solve other sub-problems.
- **The intuition behind dynamic programming is that we trade space for time**, i.e. instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.
- Let's take an example of Fibonacci numbers :

Fibonacci (n) = 1; if n = 0

Fibonacci (n) = 1; if n = 1

Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)

So, the first few numbers in this series will be: **1, 1, 2, 3, 5, 8, 13, 21...** and so on!



# Dynamic Programming Approach (cont..)

- A code for Fibonacci Series using pure recursion:

```
int fib (int n) {  
    if (n < 2)  
        return 1;  
    return fib(n-1) + fib(n-2);  
}
```

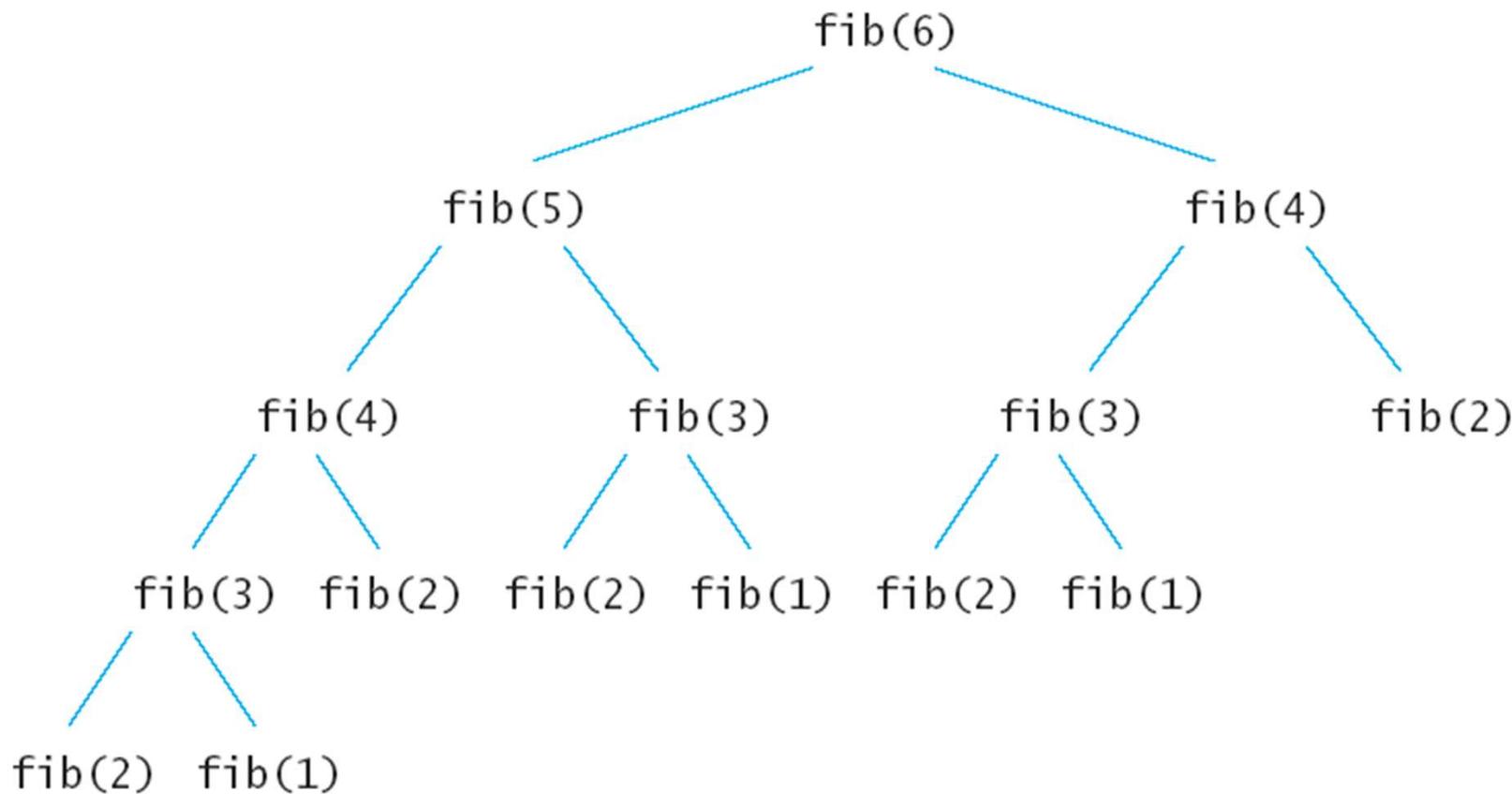
- Using Dynamic Programming approach with memorization:

```
void fib () {  
    fibresult[0] = 1;  
    fibresult[1] = 1;  
    for (int i = 2; i<n; i++)  
        fibresult[i] = fibresult[i-1] +  
    fibresult[i-2];  
}
```



# Dynamic Programming Approach (cont..)

- In the recursive code, a lot of values are being recalculated multiple times.
- We could do good with calculating each unique quantity only once.



# Properties of Dynamic Programming Approach

---

## 1. Overlapping sub-problems :

- Similar to Divide-and-Conquer approach, Dynamic Programming also combines solutions to sub-problems.
- It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table, so that these don't have to be re-computed.
- Hence, this technique is needed where overlapping sub-problem exists.
- For example,
  - Binary Search does not have overlapping sub-problem. Whereas Fibonacci numbers have many overlapping sub-problems.



# Properties of Dynamic Programming Approach

## (cont..)

---

### 2. Optimal substructure

- A given problem has Optimal Substructure Property, if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems.
- For example, the Shortest Path problem has the following optimal substructure property  
If a node  $x$  lies in the shortest path from a source node  $u$  to destination node  $v$ , then the shortest path from  $u$  to  $v$  is the combination of the shortest path from  $u$  to  $x$ , and the shortest path from  $x$  to  $v$ .
- The standard All Pair Shortest Path algorithms like Floyd-Warshall and Bellman-Ford are typical examples of Dynamic Programming.

# Steps of Dynamic Programming Approach

---

Dynamic Programming algorithm is designed using the following four steps –

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from the computed information.



# Applications of Dynamic Programming Approach

---

- Single Source Shortest Path
- Longest common subsequence
- Floyd Warshall's algorithm for all-pairs shortest paths
- Assembly Line Scheduling
- Travelling Salesman Problem
- 0/1 Knapsack Problem



# Difference between Divide & Conquer and Greedy Method

Dynamic Programming	Greedy Method
Dynamic Programming is used to obtain the optimal solution.	Greedy Method is also used to get the optimal solution.
In Dynamic Programming, we choose at each step, but the choice may depend on the solution to sub-problems.	In a greedy Algorithm, we make whatever choice seems best at the moment and then solve the sub-problems arising after the choice is made.
Less efficient as compared to a greedy approach	More efficient as compared to a greedy approach
Example: 0/1 Knapsack	Example: Fractional Knapsack
It is guaranteed that Dynamic Programming will generate an optimal solution using Principle of Optimality.	In Greedy Method, there is no such guarantee of getting Optimal Solution.
It is a top-down approach.	It is a Bottom-up approach.

# Multistage Graphs



# Multistage Graph

---

- A multistage graph is a directed graph having a number of multiple stages.
- The goal is to find the shortest distance from source to sink.
- The number on the directed edge denotes the weight/distance between the two vertices.
- In multistage graph problem we have to find the shortest path from source to sink.
- The cost of a path from source (denoted by S) to sink (denoted by T) is the sum of the costs of edges on the path.
- The multistage graph can be solved using forward and backward approach.
- In forward approach we will find the path from destination to source, in backward approach we will find the path from source to destination.



# Multistage Graph (cont..)

---

- Let  $G=(V, E)$  be a directed graph. In this we divide the problem into no. of stages or multiple stages then we try to solve whole problem.
- A multistage graph  $G=(V,E)$  is a directed graph in which the vertices are partitioned into  $k \geq 2$  disjoint sets  $V_i$ ,  $i \leq i \leq k$ .
- The vertex  $s$  is source and  $t$  is the sink. Let  $c(i, j)$  be the cost of edge .
- The cost of a path from  $s$  to  $t$  is the sum of costs of the edges on the path.
- The multistage graph problem is to find a minimum-cost path from  $s$  to  $t$ .
- In multistage graph problem we obtain the minimum path at each current stage by considering the path length of each vertex obtained in earlier stage.
- Thus the sequence of decisions is taken by considering overlapping solutions.



# Multistage Graph (cont..)

---

- A dynamic programming formulation for a k-stage graph problem is obtained by first noticing that every s to t path is the result of a sequence of k-2 decisions.
- The  $i^{\text{th}}$  decision involves determining which vertex in  $V_{i+1}$ ,  $1 \leq i \leq k-2$ , is on the path.
- It is easy to see that principle of optimality holds.
- Let  $p(i, j)$  be a minimum-cost path from vertex  $j$  in  $V_i$  to vertex  $t$ . Let  $\text{cost}(i, j)$  be the cost of this path.
- Using forward approach to find cost of the path.

$\text{Cost}(i, j) = \min \{ c(j, l) + \text{cost}(i+1, l) \}$  for  $i$ -stage number where  $j$ -vertices available at particular stage  $l$ - vertices away from the vertex

# Algorithm of Multistage Graph (Forward Approach)

Algorithm Fgraph (graph G, int k, int n, int p[])

// The input is a k-stage graph G=(V, E)with n vertices indexed on order of stages. E is a set of

// edges and cost[i, j] is cost of <i, j> , p[1:k] is a minimum cost path

{

    float cost [max size]; int d [max size], r;

    Cost [n] = 0.0

    for (int j = n-1; j>= 1; j--)

        { // Compute cost[j]

Let r be a vertex such that <j, r> is an edge of G and C[j][r] + cost[r] is minimum;

        Cost [j] = C[j][r] + Cost[r]

        d [j] = r

    }

// Find a minimum cost path

    P [1] = 1 , P[k] = n

    for (j = 2 ; j <= k-1; j++)

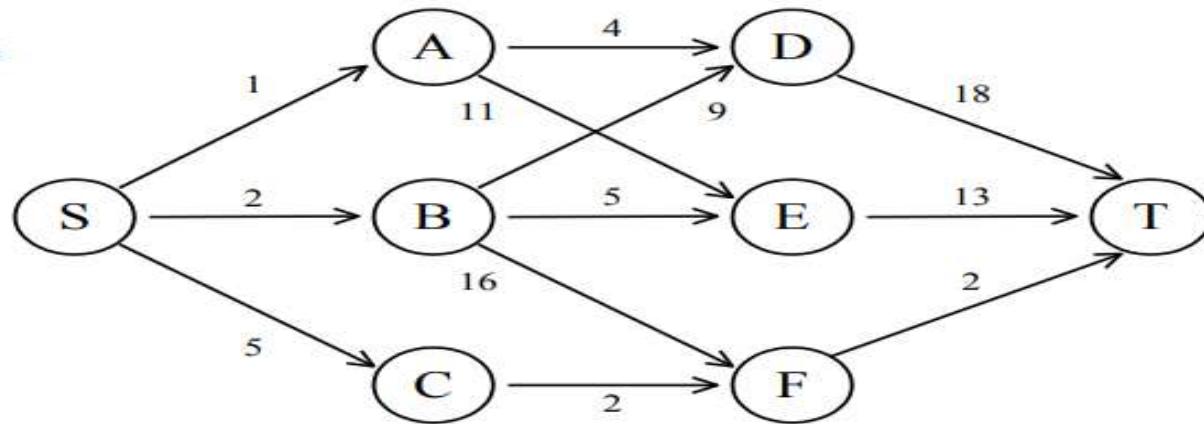
        P[j] = d[P(j-1)];

}



## Example 1:

Q. Find shortest path in given multistage graph.

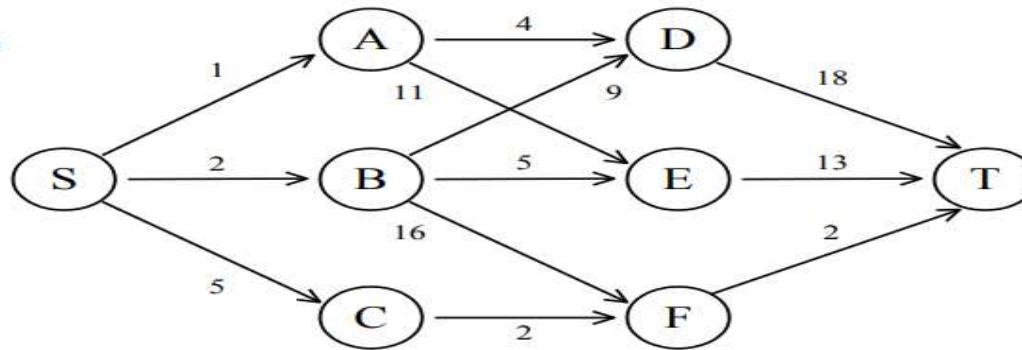


- If we go with greedy approach will get shortest path as:  $(S, A, D, T) = 1+4+18 = 23$
- But the real shortest path is:  $(S, C, F, T) = 5+2+2 = 9$ .
- Hence, the greedy method can not be applied to this case.

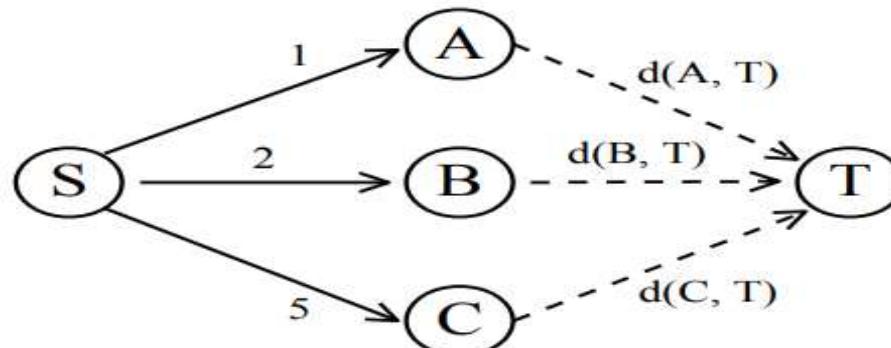


## Example 1 (Forward Approach) :

Q. Find shortest path in given multistage graph from S to T.



- **Step 1:** Starting with Source 'S', we have shortest edge as S-A with weight 1. Hence starting with this edge find out next shortest path to reach 'T'.

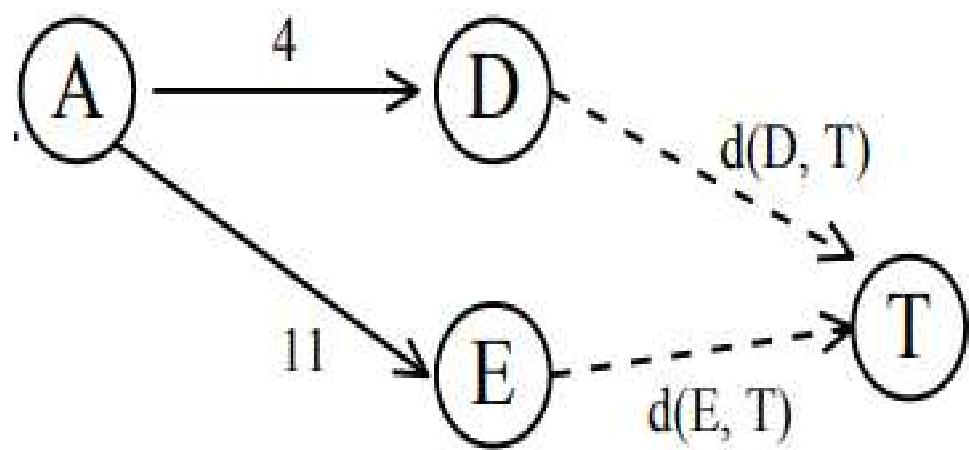


$$d(S, T) = \min\{1 + d(A, T), 2 + d(B, T), 5 + d(C, T)\}$$



## Example 1 (Forward Approach)(cont..)

- **Step 2 :** Now, we have to find  $d(A, T)$ ,  $d(B, T)$  and  $d(C, T)$ .
- **Step 3 :** Find the distance  $d(A, T)$ .

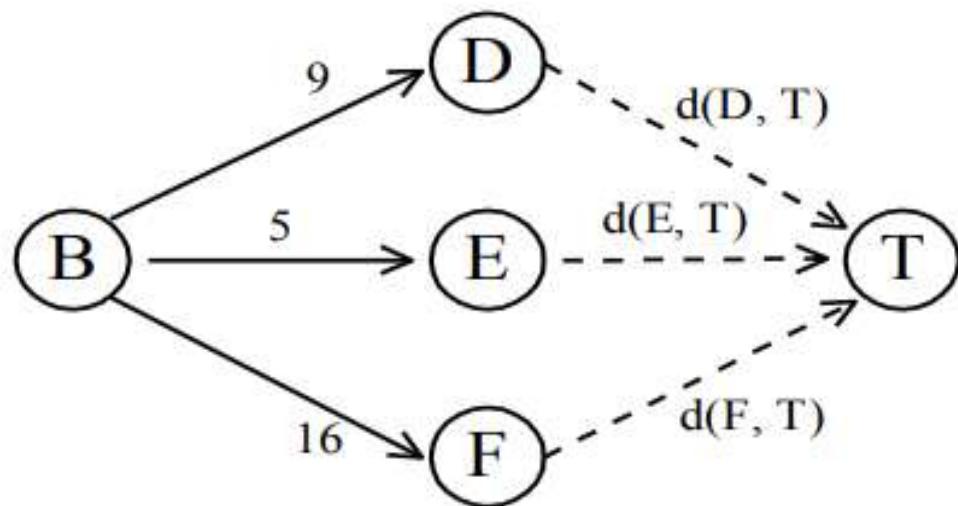


$$\begin{aligned} d(A, T) &= \min\{4+d(D, T), 11+d(E, T)\} \\ &= \min\{4+18, 11+13\} \\ &= 22 \end{aligned}$$



## Example 1(Forward Approach) (cont..)

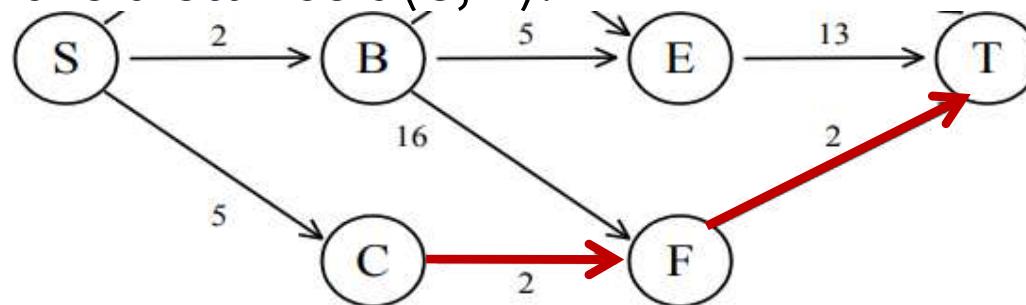
- Step 4 : Find the distance  $d(B, T)$ .



$$\begin{aligned}d(B, T) &= \min\{9+d(D, T), 5+d(E, T), 16+d(F, T)\} \\&= \min\{9+18, 5+13, 16+2\} \\&= 18\end{aligned}$$

## Example 1(Forward Approach) (cont..)

- **Step 5 :** Find the distance  $d(C, T)$ .



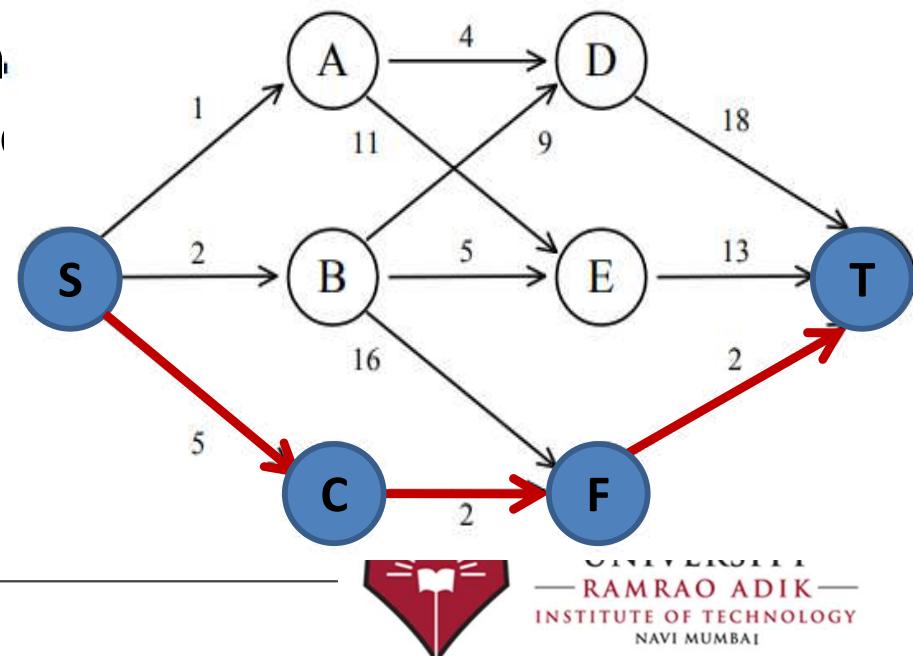
$$\begin{aligned}d(C, T) &= \min\{ 2+d(F, T) \} \\&= 2+2 \\&= 4\end{aligned}$$

- **Step 6 :** Finally calculating shortest pa

$$\begin{aligned}d(S, T) &= \min\{1+d(A, T), 2+d(B, T), 5+\\&\quad d(C, T)\} \\&= \min\{1+22, 2+18, 5+4\} \\&= 9\end{aligned}$$

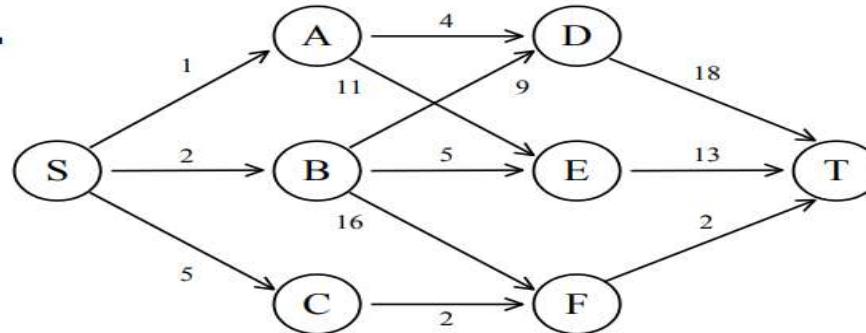
**Shortest Distance =  $d(S, T) = 9$**

**Shortest Path =  $S \rightarrow C \rightarrow F \rightarrow T$**



## Example 2(Backward Approach) :

**Q. Find shortest path in given multistage graph from S to T.**



- **Step 1** : Starting with Sink 'S', find distance to reach next vertex.

$$d(S, A) = 1$$

$$d(S, B) = 2$$

$$d(S, C) = 5$$

- **Step 2** : Now, find distance from sink 'S' to vertices D, E and F respectively.

$$d(S, D) = \min\{d(S, A)+d(A, D), d(S, B)+d(B, D)\} = \min\{ 1+4, 2+9 \} = 5$$

$$d(S, E) = \min\{d(S, A)+d(A, E), d(S, B)+d(B, E)\} = \min\{ 1+11, 2+5 \} = 7$$

$$\begin{aligned} d(S, F) &= \min\{d(S, B)+d(B, F), d(S, C)+d(C, F)\} \\ &= \min\{ 2+16, 5+2 \} = 7 \end{aligned}$$

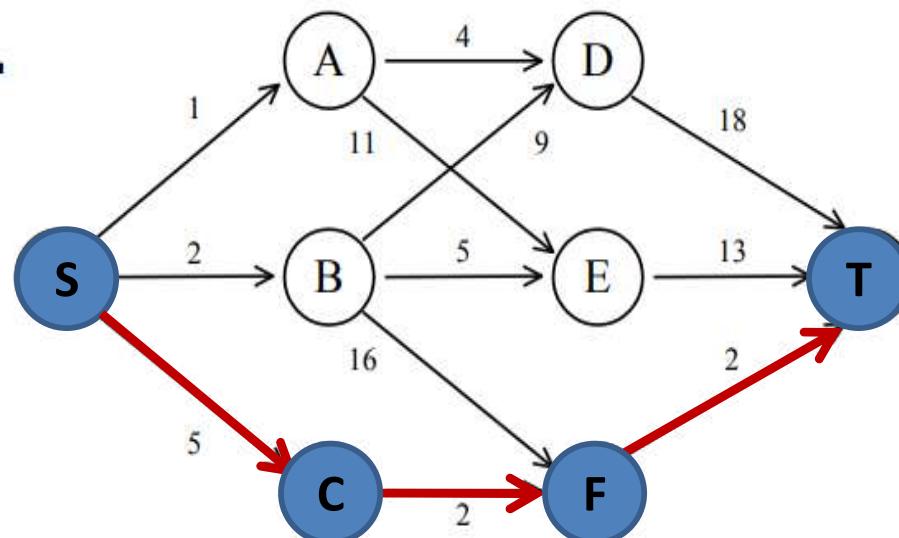


## Example 2(Backward Approach) (cont..)

- **Step 3 :** Finally calculating shortest distance from source 'S' to target 'T'.

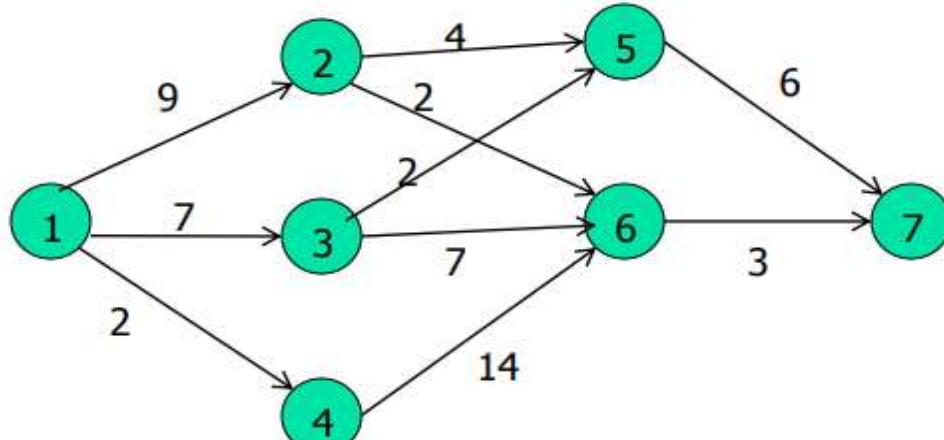
$$\begin{aligned} d(S, T) &= \min\{ d(S, D)+d(D, T), d(S, E)+d(E, T), d(S, F)+d(F, T) \} \\ &= \min\{ 5+18, 7+13, 7+2 \} \\ &= 9 \end{aligned}$$

**Shortest Distance =  $d(S, T) = 9$**   
**Shortest Path =  $S \rightarrow C \rightarrow F \rightarrow T$**



## Example 3 :

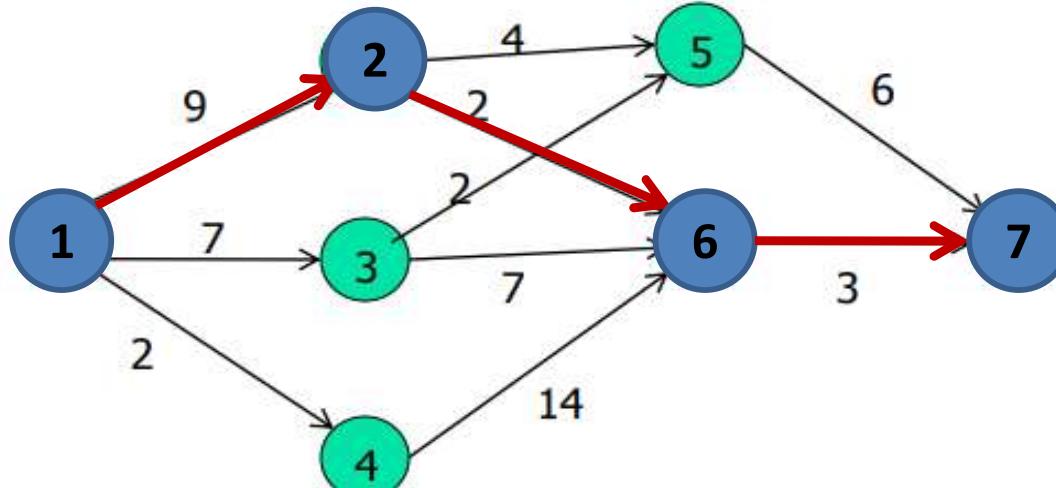
Q. Find shortest path in given multistage graph from 1 to 7.



→Solution :

Shortest Distance =  $d(1, 7) = 14$

Shortest Path =  $1 \rightarrow 2 \rightarrow 6 \rightarrow 7$



# Analysis of Multistage Graph

---

- In a multi-stage graph algorithm for shortest path, we minimise cost for every edge exactly once.
- So the Time Complexity is  $O(E)$ .
- However, in the worst case, we get a complete graph, which has edges  $E = n*(n-1)/2$ .
- So worst time complexity then becomes  $O(E) = O(n^2)$ .

# Single Source Shortest Path (Bellman Ford) Algorithm



# Single Source Shortest Path Algorithm

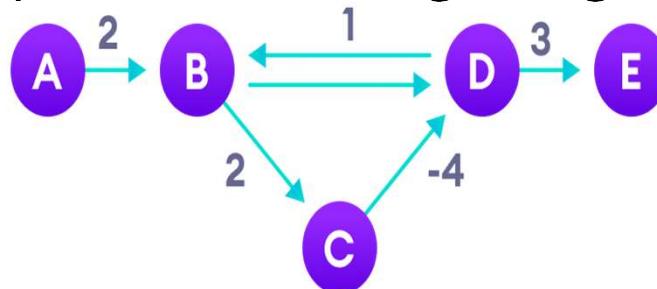
---

- Given a graph and a source vertex  $src$  in graph, find shortest paths from  $src$  to all vertices in the given graph.
- Algorithms used:
  1. Dijkstra's Algorithm
  2. Bellman Ford Algorithm
- Sometimes graph may contain negative weight edges.
- Dijkstra's algorithm is a Greedy algorithm. Dijkstra doesn't work for Graphs with negative weight edges.
- Bellman Ford algorithm is used to compute the shortest path from a single source vertex to all the other vertices in a given weighted digraph.
- Bellman Ford algorithm is slower than Dijkstra's algorithm but it can handle negative weight edges in the graph unlike Dijkstra's.
- The Bellman-Ford algorithm uses the principle of relaxation to find increasingly accurate path length.



## Bellman Ford Algorithm (cont..)

- Negative weight edges can create negative weight cycles i.e. a cycle that will reduce the total path distance(edge weight) by coming back to the same point.



- If a graph contains a ‘negative cycle’ that is reachable from the source, then there is no shortest path.
- Shortest path algorithms like Dijkstra's Algorithm that aren't able to detect such a cycle and can give an incorrect result because they can go through a negative weight cycle and reduce the path length.
- Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices.
- Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths.



# Algorithm for Bellman Ford Concept

---

## Bellman-Ford ( $G, w, s$ )

1. Initialize Single Source ( $G, s$ )
2. for  $i \rightarrow 1$  to  $|G.V| - 1$
3.     do for each edge  $(u, v) \in |G.E|$
4.         do RELAX( $u, v, w$ )
5.     for each edge  $(u, v) \in |G.E|$
6.         do if  $d[v] > d[u] + w[u, v]$
7.             then return False
8.     return True

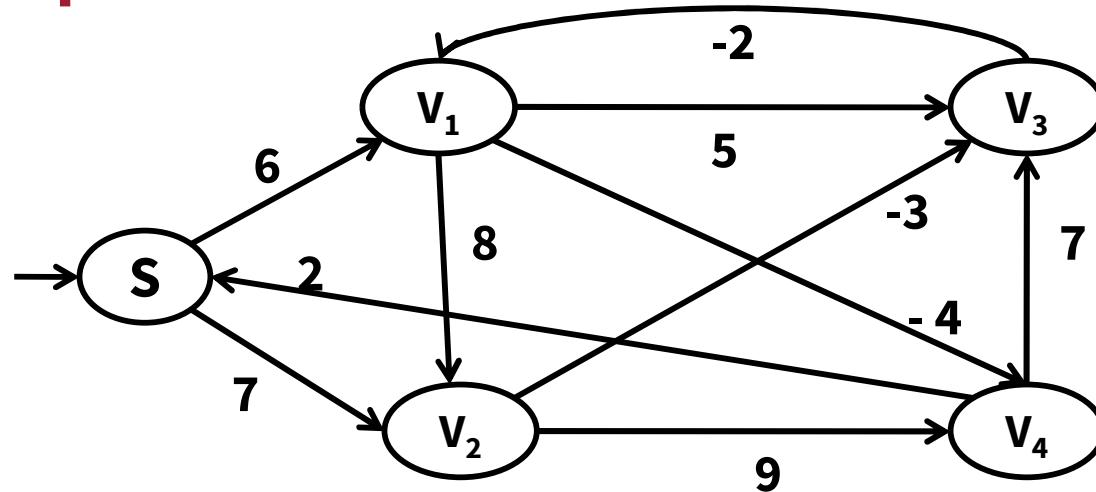
## RELAX( $u, v, w$ )

1. if  $d[v] > d[u] + w[u][v]$
2.     then  $d[v] = d[u] + w[u][v]$  // Edge Relaxation
3.      $\Pi[v] = u$



## Example 1 :

**Q. Find shortest path from source vertex to all other vertices.**



→ **Solution :**

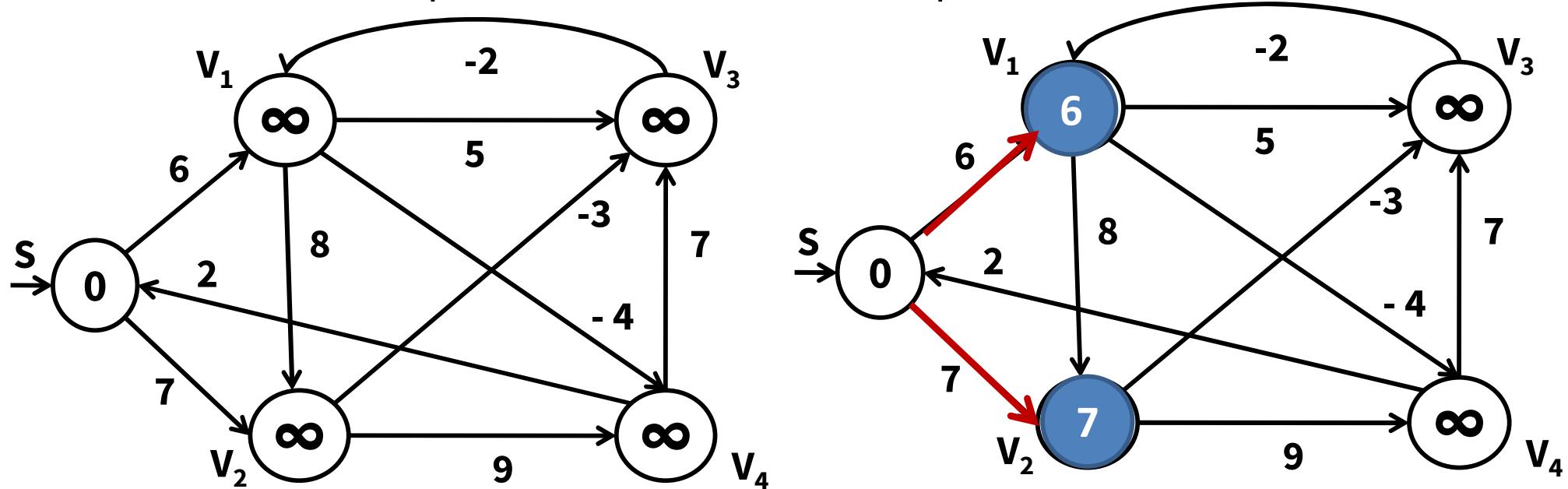
- **Step 1 :** Initialize all vertices at  $\infty$  distance from the source vertex and distance of source vertex = 0.

V	S	v <sub>1</sub>	v <sub>2</sub>	v <sub>3</sub>	v <sub>4</sub>
d[v]	0	$\infty$	$\infty$	$\infty$	$\infty$
$\Pi$ [v]	-	/	/	/	/



## Example 1 (cont..)

- **Step 2 :** Start with source vertex, update the distances to minimum value and do relaxation procedure to find shortest path.

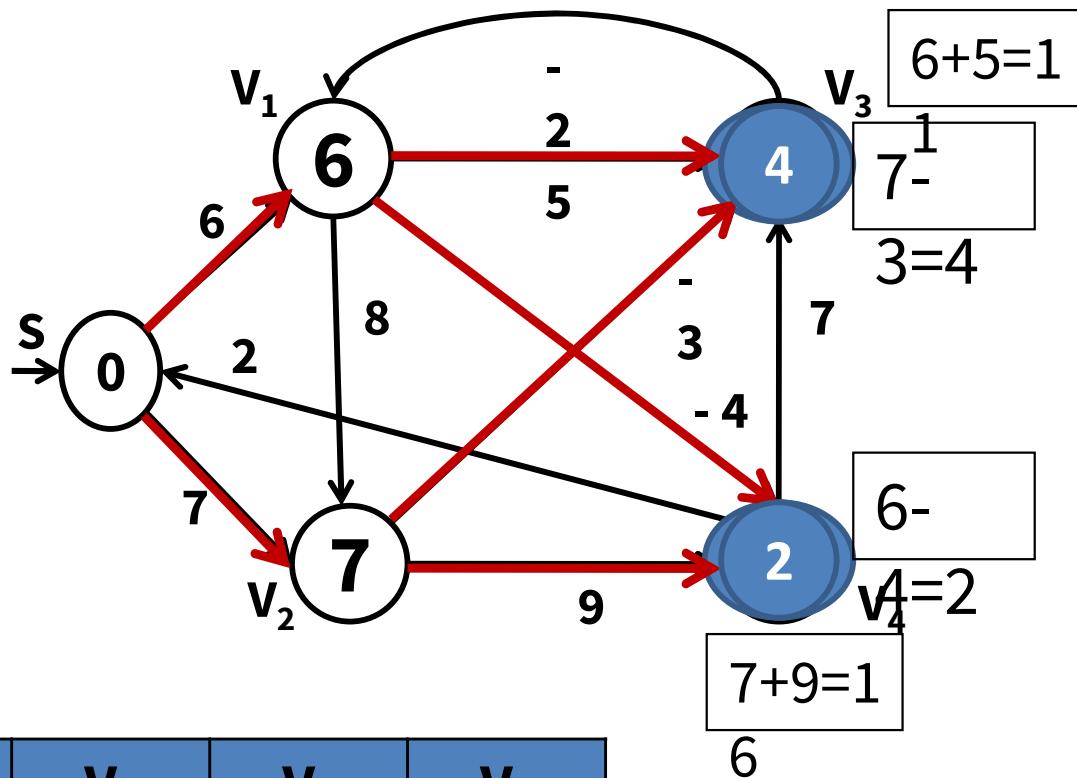
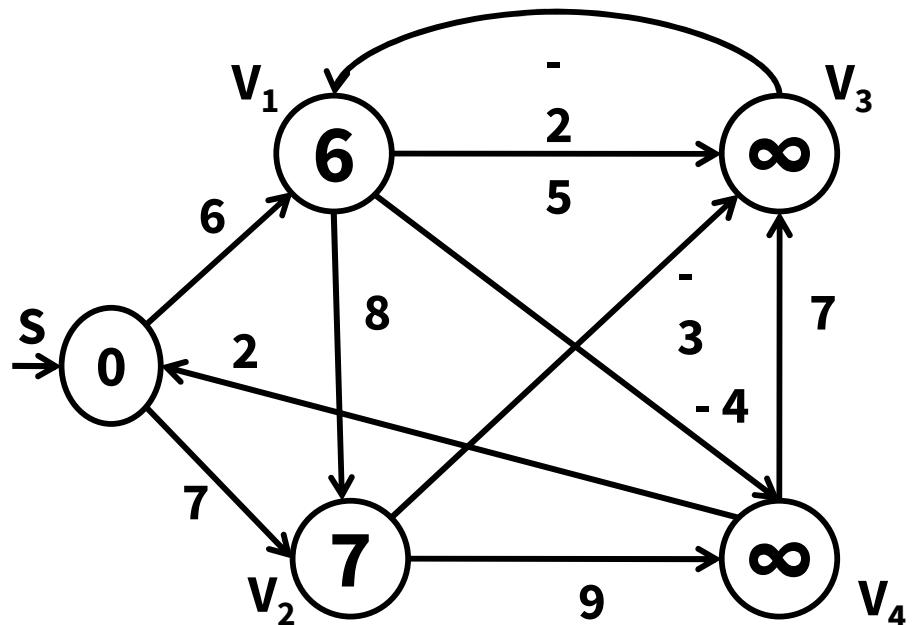


V	S	v <sub>1</sub>	v <sub>2</sub>	v <sub>3</sub>	v <sub>4</sub>
d[v]	0	6	7	∞	∞
Π[v]	-	S	S	/	/



## Example 1 (cont..)

- Step 3 :

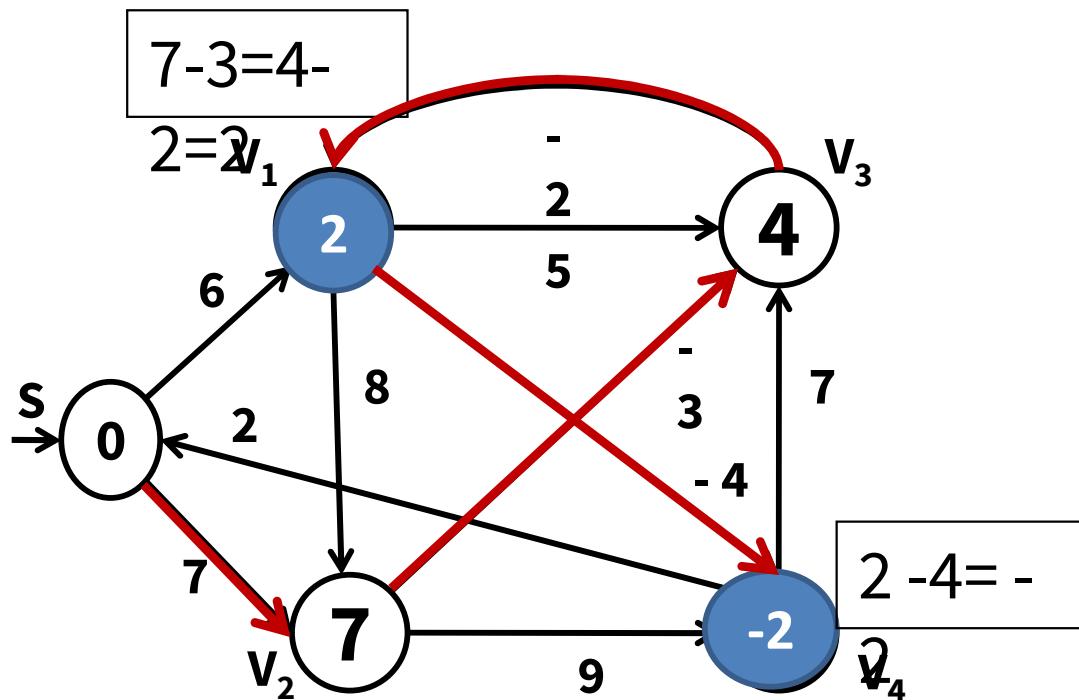
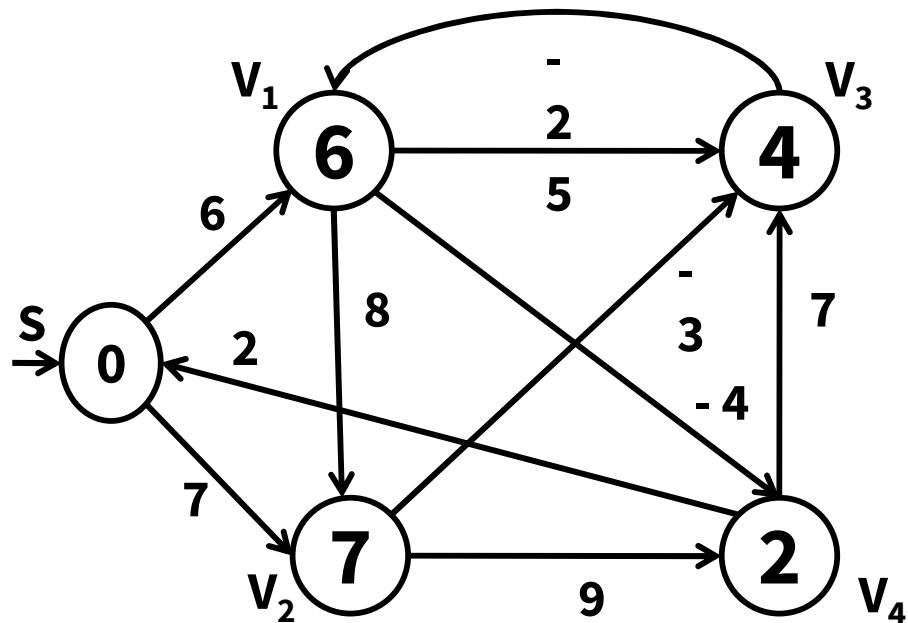


V	S	v <sub>1</sub>	v <sub>2</sub>	v <sub>3</sub>	v <sub>4</sub>
d[v]	0	6	7	4	2
$\Pi[v]$	-	s	s	$v_2$	$v_1$



## Example 1 (cont..)

- Step 4 :

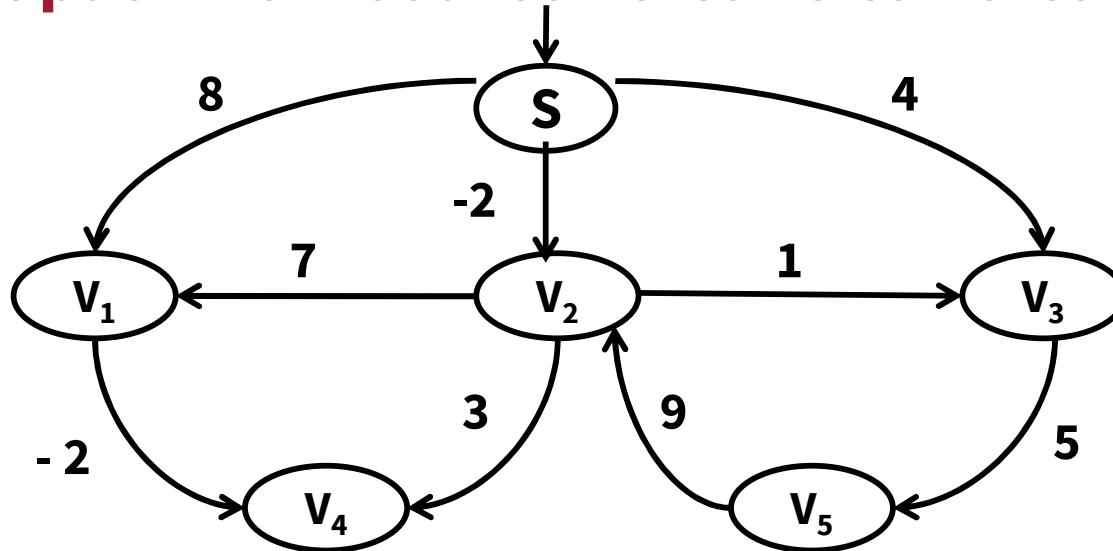


V	S	V <sub>1</sub>	V <sub>2</sub>	V <sub>3</sub>	V <sub>4</sub>
d[v]	0	2	7	4	-2
$\Pi[v]$	-	V <sub>3</sub>	S	V <sub>2</sub>	V <sub>1</sub>



## Example 2:

Q. Find shortest path from source vertex S to vertex  $V_5$ .



→Solution :

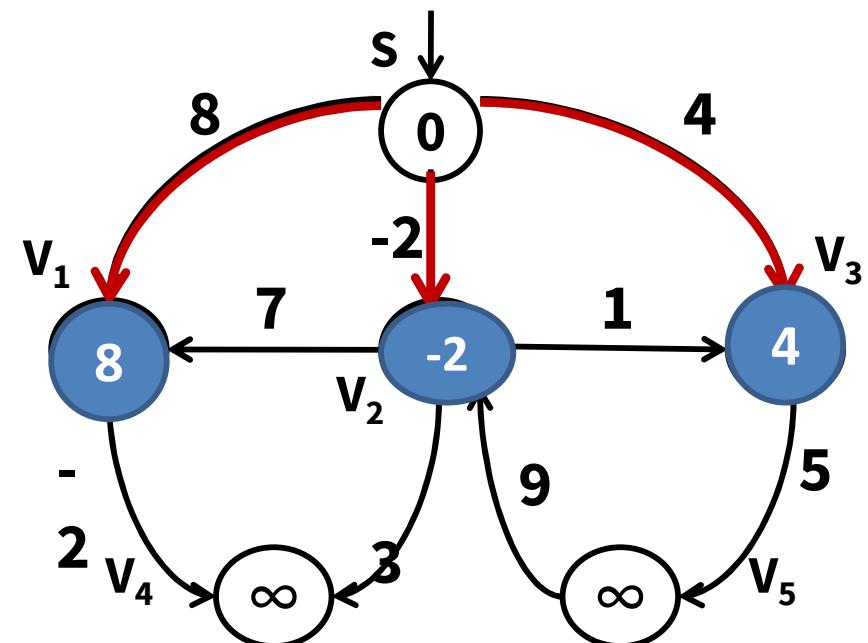
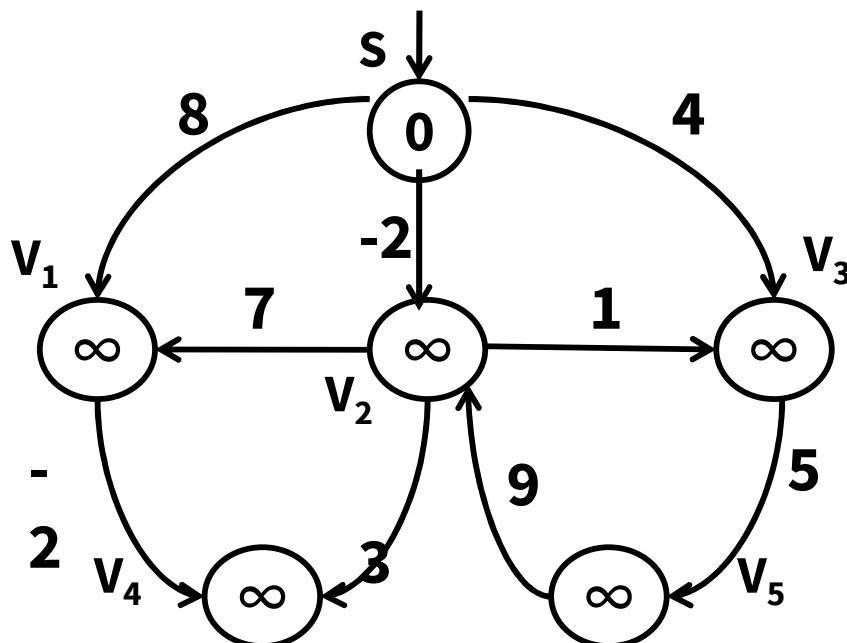
- **Step 1 :** Initialize all vertices at  $\infty$  distance from the source vertex and distance of source vertex =0.

V	S	$V_1$	$V_2$	$V_3$	$V_4$	$V_5$
d[v]	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
$\Pi[v]$	-	/	/	/	/	/



## Example 2 (cont..)

- **Step 2:** Start with source vertex, update the distances to minimum value and do relaxation procedure to find shortest path.

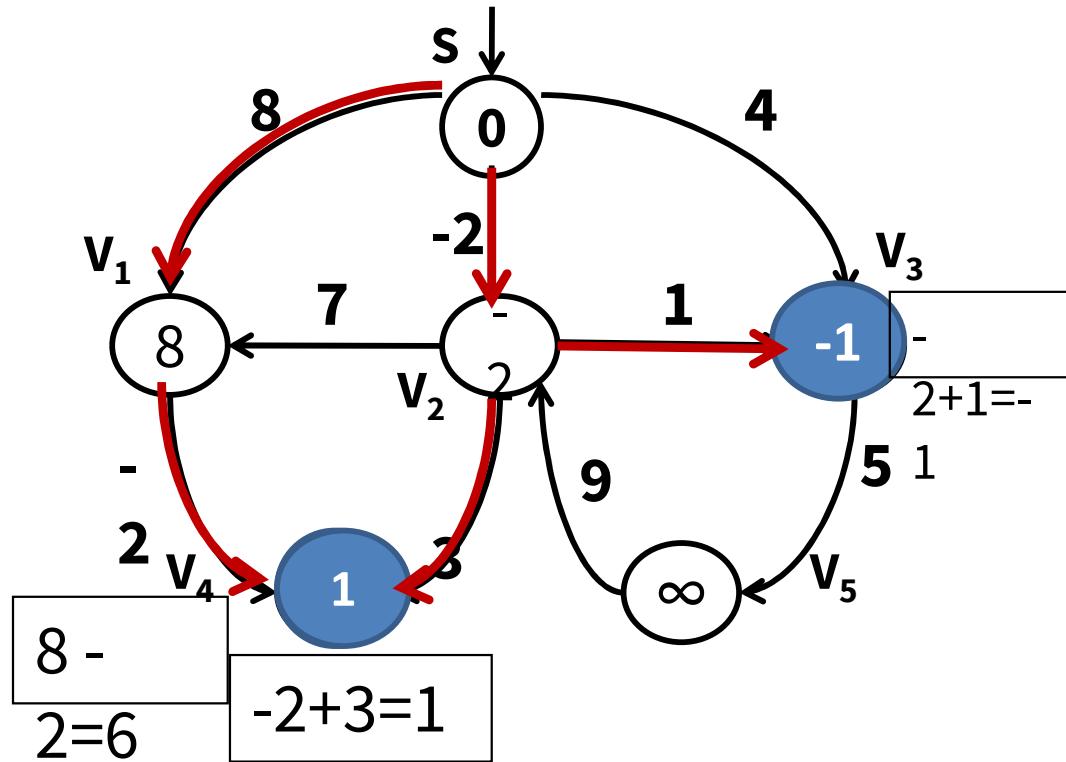
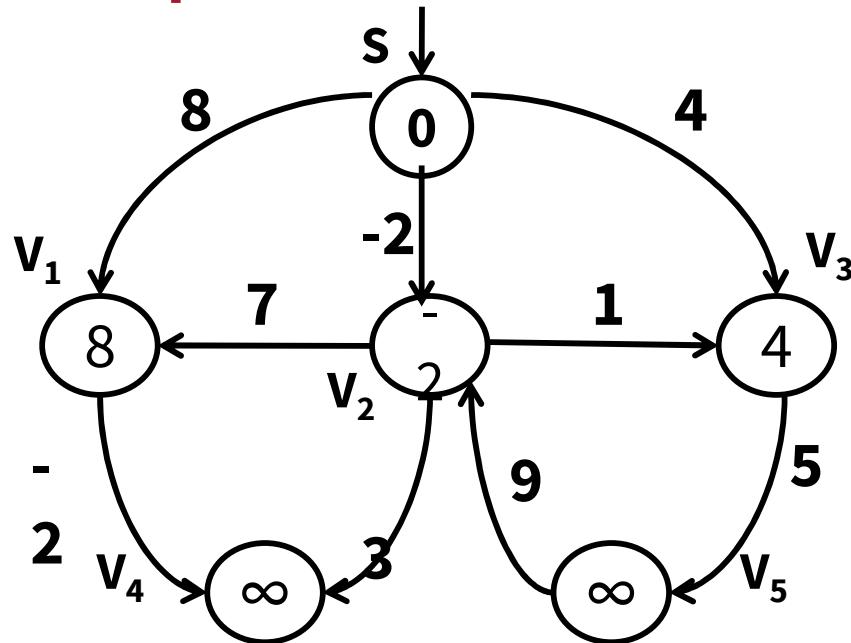


$v$	$s$	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$
$d[v]$	0	8	-2	4	$\infty$	$\infty$
$\Pi[v]$	-	$s$	$s$	$s$	/	/



## Example 2 (cont..)

- Step 3:

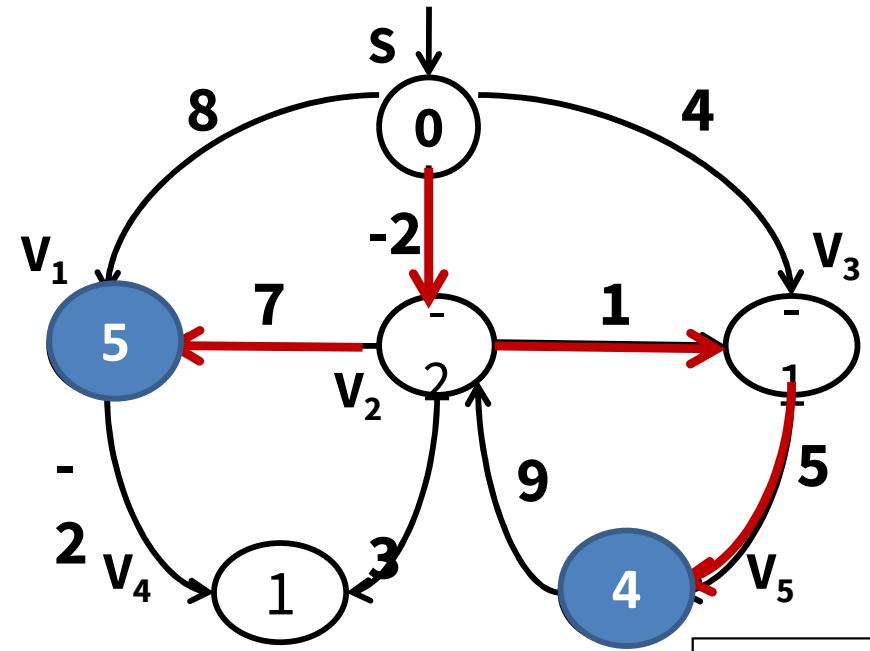
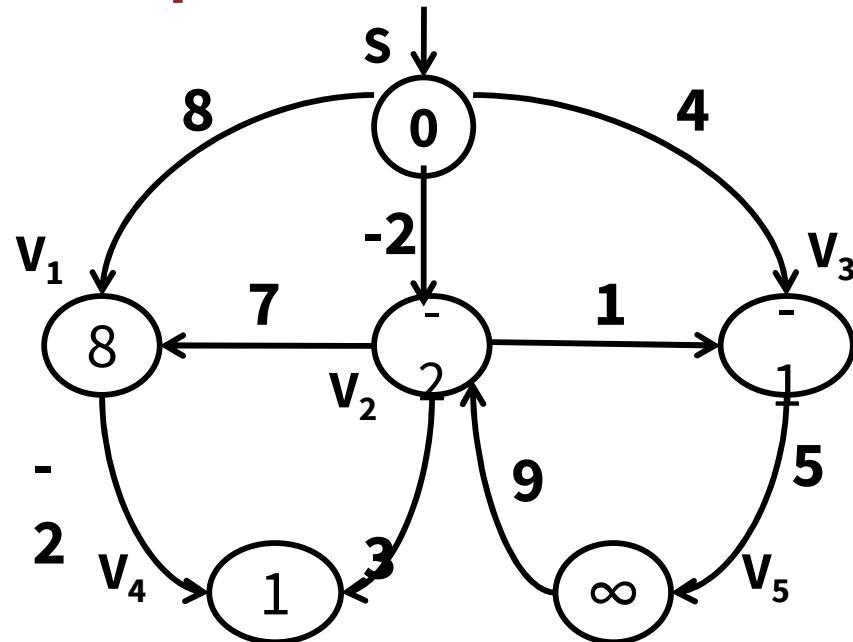


V	S	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$
$d[v]$	0	8	-2	1	-1	infinity
$\Pi[v]$	-	S	S	$v_2$	$v_2$	/



## Example 2 (cont..)

- ## • Step 4 :

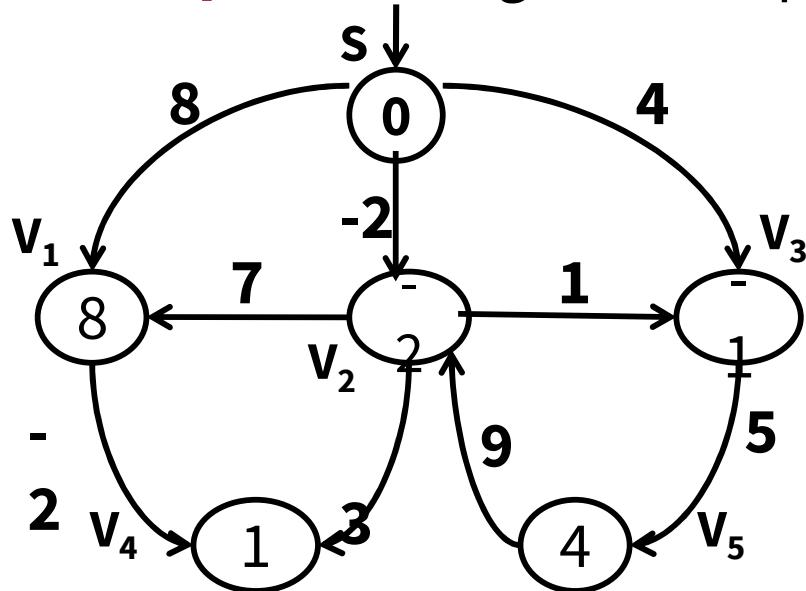


<b>V</b>	<b>S</b>	<b>V<sub>1</sub></b>	<b>V<sub>2</sub></b>	<b>V<sub>3</sub></b>	<b>V<sub>4</sub></b>	<b>V<sub>5</sub></b>
d[v]	0	5	-2	1	-1	4
$\Pi[v]$	-	$V_2$	$S$	$V_2$	$V_2$	$V_3$

$$\begin{array}{r} 1+5= \\ 4 \end{array}$$

## Example 2 (cont..)

- **Step 5 :** Finding shortest path from source to  $V_5$  .



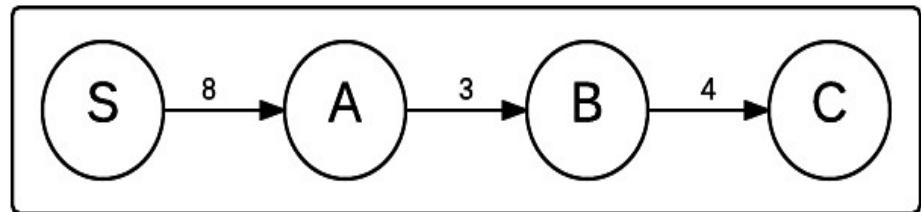
V	S	$V_1$	$V_2$	$V_3$	$V_4$	$V_5$
$d[v]$	0	5	-2	1	-1	4
$\Pi[v]$	-	$V_2$	S	$V_2$	$V_2$	$V_3$

**Check path array starting from vertex 5.**

$$\Pi[5] = V_3$$

$$\Pi[3] = V_2$$

$$\Pi[2] = S$$



**Shortest Path =  $S \rightarrow V_2 \rightarrow V_3 \rightarrow V_5$**

# Analysis of Bellman Ford Algorithm

---

- Bellman-Ford makes  $|E|$  relaxations for every iteration, and there are  $|V| - 1$  iterations.
- Therefore, the worst-case scenario is that Bellman-Ford runs in  $O(|V|.|E|)$  time.
- However, in some scenarios, the number of iterations can be much lower.
- For certain graphs, only one iteration is needed, and hence in the best case scenario, only  $O(|E|)$  time is needed.
- An example of a graph that would only need one round of relaxation is a graph where each vertex only connects to the next one in a linear fashion, like the graphic below:



# Applications of Bellman Ford Algorithm

---

- Bellman-Ford is used in the distance-vector routing protocol.
- This protocol decides how to route packets of data on a network.
- The distance equation (to decide weights in the network) is the number of routers a certain path must go through to reach its destination.
- A second example is the interior gateway routing protocol.
- This proprietary protocol is used to help machines exchange routing data within a system.



# All Pair Shortest Path



# All Pair Shortest Path

---

- It aims to figure out the shortest path from each vertex ‘v’ to every other vertex ‘u’.
- Floyd-Warshall algorithm is used to find all pair shortest path for a given weighted graph.
- As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.
- At first the output matrix is same as given cost matrix of the graph. After that the output matrix will be updated with all vertices ‘k’ as the intermediate vertex.

# Floyd Warshall Algorithm

- Let the vertices of  $G$  be  $V = \{1, 2, \dots, n\}$  and consider a subset  $\{1, 2, \dots, k\}$  of vertices for some  $k$ .
- For any pair of vertices  $i, j \in V$ , consider all paths from  $i$  to  $j$  whose intermediate vertices are all drawn from  $\{1, 2, \dots, k\}$ , and let  $p$  be a minimum weight path from amongst them.
- The Floyd-Warshall algorithm exploits a link between path  $p$  and shortest paths from  $i$  to  $j$  with all intermediate vertices in the set  $\{1, 2, \dots, k-1\}$ .
- If  $k$  is an intermediate vertex of path  $p$ , then we break  $p$  down into  $i \rightarrow k \rightarrow j$ .
- Let  $d_{ij}^{(k)}$  be the weight of the shortest path from vertex  $i$  to vertex  $j$  with all intermediate vertices in  $\{1, 2, \dots, k\}$ .  
with all  $d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1 \end{cases}$ .



# Algorithm of Floyd Warshall Concept

---

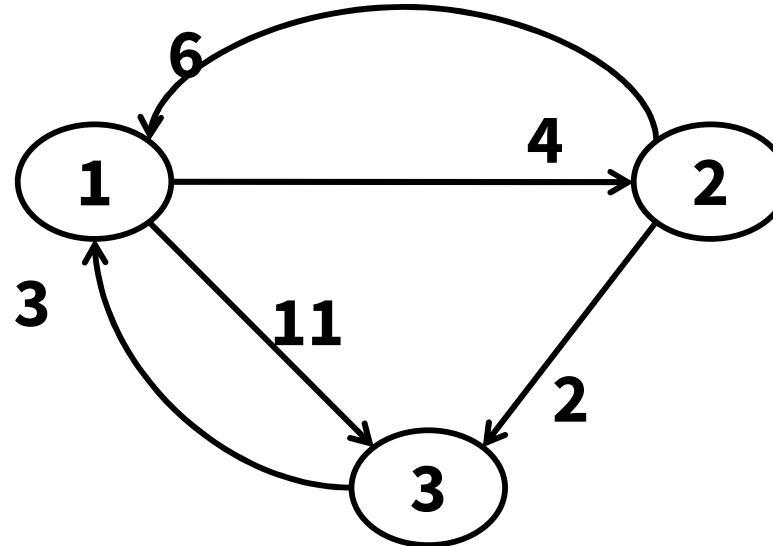
## FLOYD - WARSHALL (W)

1.  $n \leftarrow \text{rows}[W]$
2.  $D^0 \leftarrow W$
3. for  $k \leftarrow 1$  to  $n$
4.   do for  $i \leftarrow 1$  to  $n$
5.     do for  $j \leftarrow 1$  to  $n$
6.       do  $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
7. return  $D^{(n)}$

- The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3-6.
- Each execution of line 6 takes  $O(1)$  time.
- The algorithm thus runs in time  $\theta(n^3)$ . ( $n \rightarrow$  no. of vertices)

## Example 1 :

Q. Find all pair shortest path for the given graph.

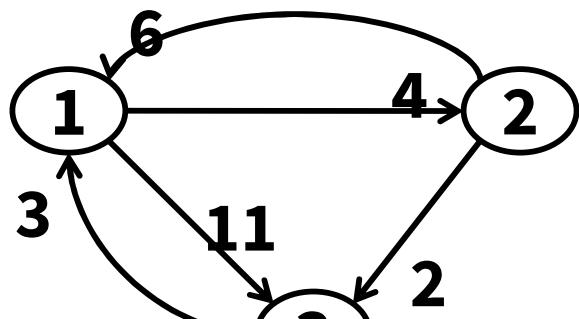


→ Solution :

- **Step 1 :** Create a distance matrix  $D[0]$  and path matrix  $\Pi[0]$  of dimension  $n \times n$  where  $n$  is the number of vertices. The row and the column are indexed as  $i$  and  $j$  respectively.
  - $i$  and  $j$  are the vertices of the graph.
  - Each cell  $D[i][j]$  is filled with the distance from the  $i^{\text{th}}$  vertex to the  $j^{\text{th}}$  vertex.
  - If there is no path from  $i^{\text{th}}$  to  $j^{\text{th}}$  vertex, the cell is left as infinity.

## Example 1 : (cont..)

- For the given example we have,



$$D[0] = \begin{vmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{vmatrix} \quad \Pi[0] = \begin{vmatrix} \text{NIL} & 1 & 1 \\ 2 & \text{NIL} & 2 \\ 3 & \text{NIL} & \text{NIL} \end{vmatrix}$$

- Step 2 :** Now, create a matrix  $D[1]$  using matrix  $D[0]$ . The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

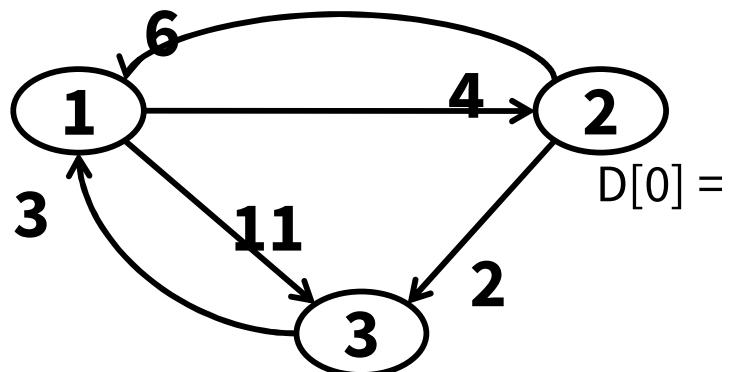
→ Let  $k$  be the intermediate vertex in the shortest path from source to destination.  $D[i][j]$  is filled with  $(D[i][k] + D[k][j])$  if  $(D[i][j] > D[i][k] + D[k][j])$ . i.e. if the direct distance from the source to the destination is greater than the path through the vertex  $k$ , then the cell is filled with  $D[i][k] + D[k][j]$ .

→ In this step,  $k$  is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex  $k$ .



## Example 1 : (cont..)

- **Step 2 :** Now,  $k=1$



$$D[0] = \begin{vmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{vmatrix}$$

$$\Pi[0] = \begin{vmatrix} \text{NIL} & 1 & 1 \\ 2 & \text{NIL} & 2 \\ 3 & \text{NIL} & \text{NIL} \end{vmatrix}$$

$D[2][3] = 2$  which is  $\min(D[2][3], D[2][1]+D[1][3])$  i.e.  $\min(2, 6+11)$

$D[3][2] = \infty$  so updating this through vertex 1,

$D[3][2] = \min(D[3][2], D[3][1]+D[1][2]) = \min(\infty, 3+4) = 7$

$\therefore D[3][2] = 7$

$$D[1] = \begin{vmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{vmatrix}$$

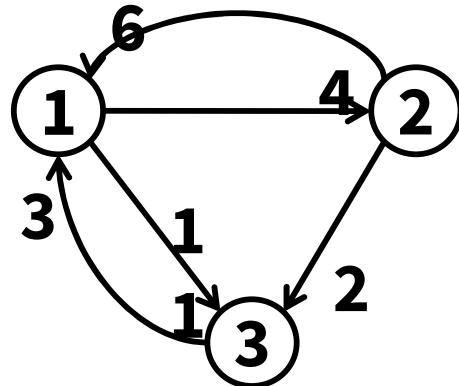
$$\Pi[1] = \begin{vmatrix} \text{NIL} & 1 & 1 \\ 2 & \text{NIL} & 2 \\ 3 & 1 & \text{NIL} \end{vmatrix}$$

Distance is updated through vertex 1 so update that in path matrix also.



## Example 1 : (cont..)

- Step 3 :** Consider  $k=2$  and calculate  $D[2]$  and  $\Pi[2]$ . The elements in the 2<sup>nd</sup> column and the 2<sup>nd</sup> row are left as they are.



$$D[1] = \begin{vmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{vmatrix} \quad \Pi[1] = \begin{vmatrix} \text{NIL} & 1 & 1 \\ 2 & \text{NIL} & 2 \\ 3 & 1 & \text{NIL} \end{vmatrix}$$

$D[1][3] = 11$  which can be updated through vertex 2 as  $\min(D[1][3], D[1][2]+D[2][3])$   
 $= \min(11, 4+2) = 6$

$\therefore D[1][3] = 6$

$D[3][1] = 3$  and we can not update that through vertex 2 as it is already minimum.

∴  $D[3][1] = 3$

0	4	6
6	0	2
3	7	0

$$\Pi[2] =$$

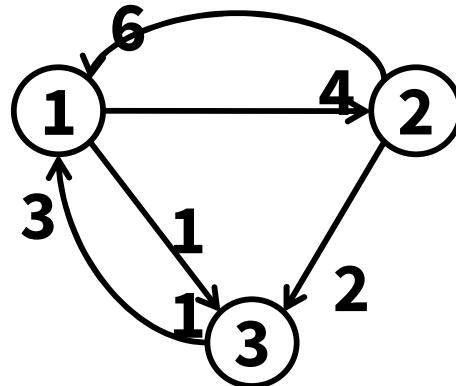
Nil	1	2
2	Nil	2
3	1	Nil

Distance is updated through vertex 2 so update that in path matrix also.



## Example 1 : (cont..)

- Step 4 :** Consider  $k=3$  and calculate  $D[3]$  and  $\Pi[3]$ . The elements in the 3<sup>rd</sup> column and the 3<sup>rd</sup> row are left as they are.



$$D[2] = \begin{vmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{vmatrix}$$

$$\Pi[2] = \begin{vmatrix} \text{NIL} & 1 & 2 \\ 2 & \text{NIL} & 2 \\ 3 & 1 & \text{NIL} \end{vmatrix}$$

$D[1][2] = 4$  and we can not update that through vertex 3 as we don't have any path.

$$\therefore D[1][2] = 4$$

$D[2][1] = 6$  which can be updated through vertex 3 as  $\min(D[2][1],$

$$D[2][3]+D[3][1])$$

$$\cdot D[2] = \begin{vmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{vmatrix}$$

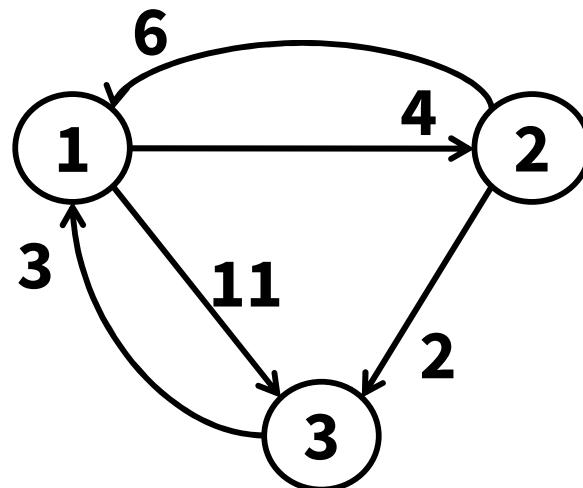
$$\Pi[3] = \begin{vmatrix} \text{NIL} & 1 & 2 \\ 3 & \text{NIL} & 2 \\ 3 & 1 & \text{NIL} \end{vmatrix}$$

Distance is updated through vertex 3 so update that in path matrix also.



## Example 1 : (cont..)

- **Step 5 :** In given graph we have only 3 vertices.



∴ D[3] gives the shortest path between each pair of vertices.

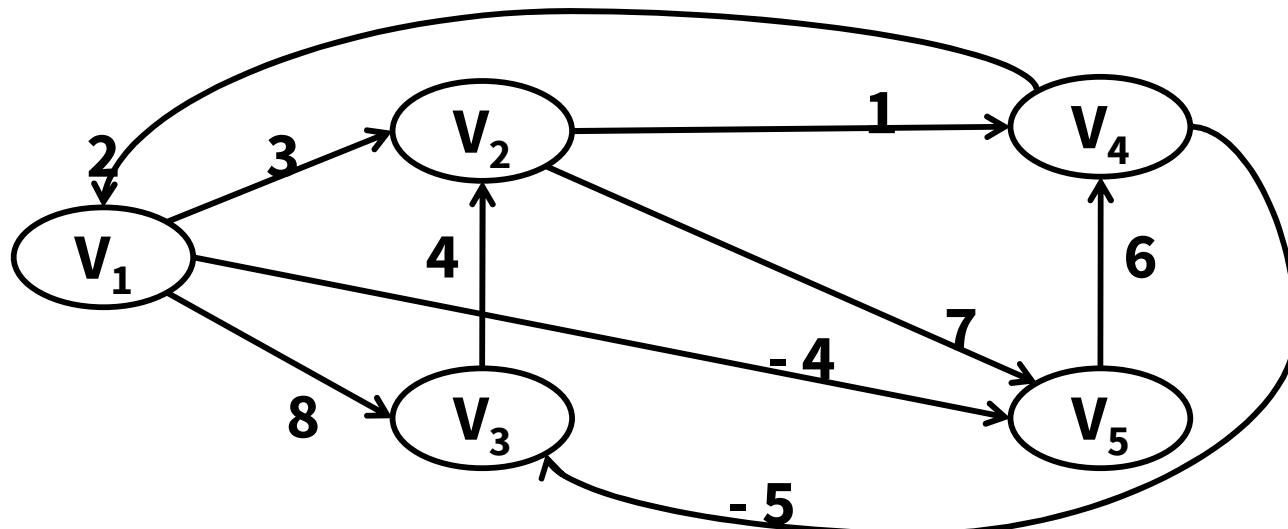
$$D[3] = \begin{vmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{vmatrix}$$

$$\Pi[3] = \begin{vmatrix} NIL & 1 & 2 \\ 3 & NIL & 2 \\ 3 & 1 & NIL \end{vmatrix}$$



## Example 2 :

Q. Find all pair shortest path for the given graph.



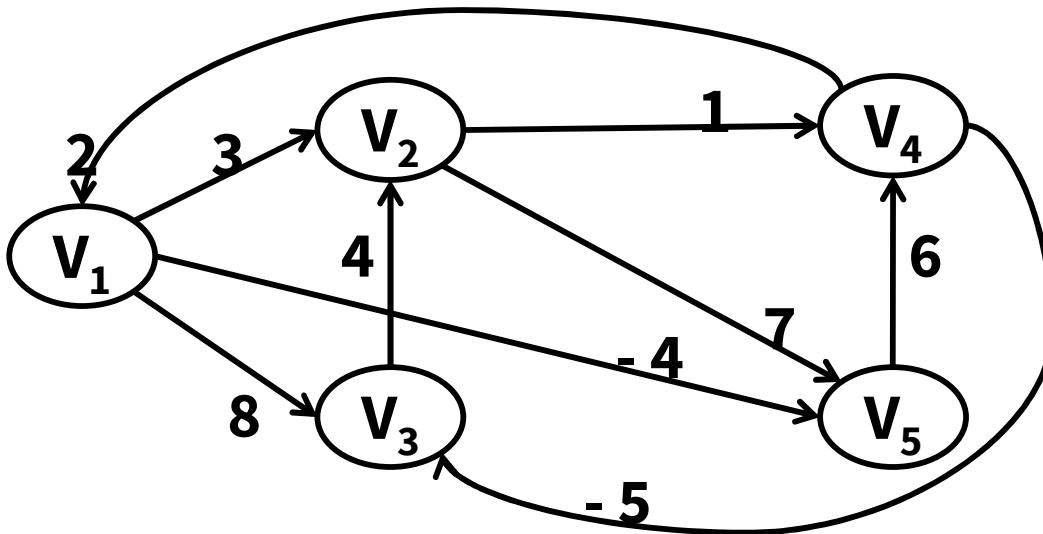
→ Solution :

- **Step 1 :** Create a distance matrix  $D[0]$  and path matrix  $\Pi[0]$  of dimensions

$$D[0] = \begin{array}{ccccccc} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{array}$$
$$\Pi[0] = \begin{array}{ccccc} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{array}$$



## Example 2 : (cont..)



$$D[0] = \begin{array}{cccccc} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{array}$$

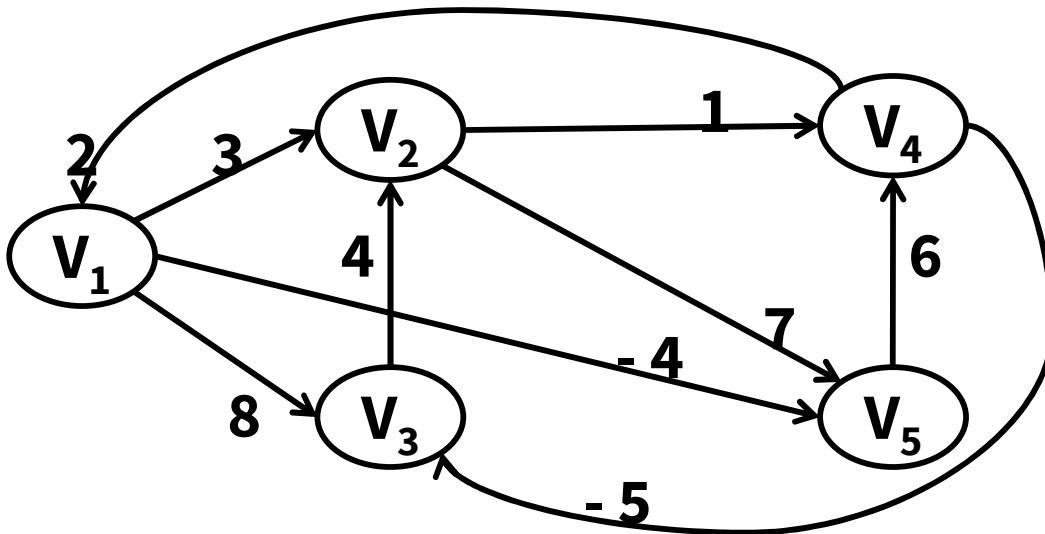
- Step 2:** Create a matrix  $D[1]$  using matrix  $D[0]$ . The elements in the first column and the first row are left as they are. Now,  $k=1$

$$D[1] = \begin{array}{ccccc} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \min(\infty, (2+3)) & \infty & \infty & 6 & 0 \end{array}$$

$\Pi[1] = \begin{array}{ccccc} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{array}$



## Example 2 : (cont..)



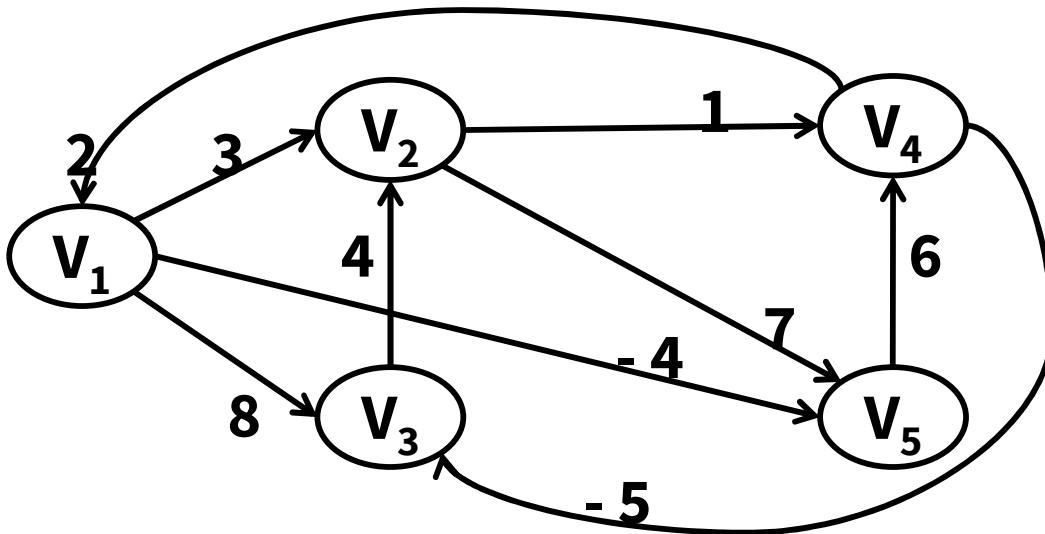
$D[1] =$	0	3	8	$\infty$	-4
	$\infty$	0	$\infty$	1	7
	$\infty$	4	0	$\infty$	$\infty$
	2	5	-5	0	-2
	$\infty$	$\infty$	$\infty$	6	0

- **Step 3 :** Create a matrix  $D[2]$  using matrix  $D[1]$ . The elements in the 2<sup>nd</sup> column and the 2<sup>nd</sup> row are left as they are. Now,  $k=2$

$D[2] =$	0	3	8	4	-4
	$\infty$	0	$\infty$	1	7
	$\infty$	4	0	5	1
	2	5	-5	0	-1
	$\infty$	$\infty$	$\infty$	6	0

$\Pi[2] =$	NIL	1	1	2	1
	NIL	NIL	NIL	2	2
	NIL	3	NIL	2	2
	4	1	4	NIL	1
	NIL	NIL	NIL	5	NIL

## Example 2 : (cont..)



$$D[2] = \begin{vmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{vmatrix}$$

- **Step 4 :** Create a matrix  $D[3]$  using matrix  $D[2]$ . The elements in the 3<sup>rd</sup> column and the 3<sup>rd</sup> row are left as they are. Now,  $k=3$

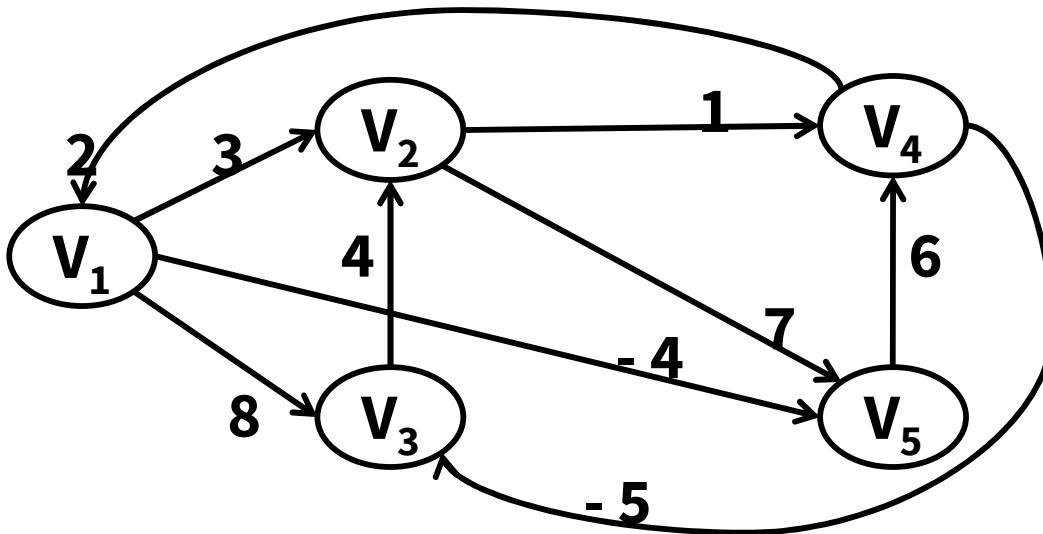
$$D[3] = \begin{vmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{vmatrix}$$

min( 5, (-5+4) )

$$\Pi[3] = \begin{vmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{vmatrix}$$



## Example 2 : (cont..)



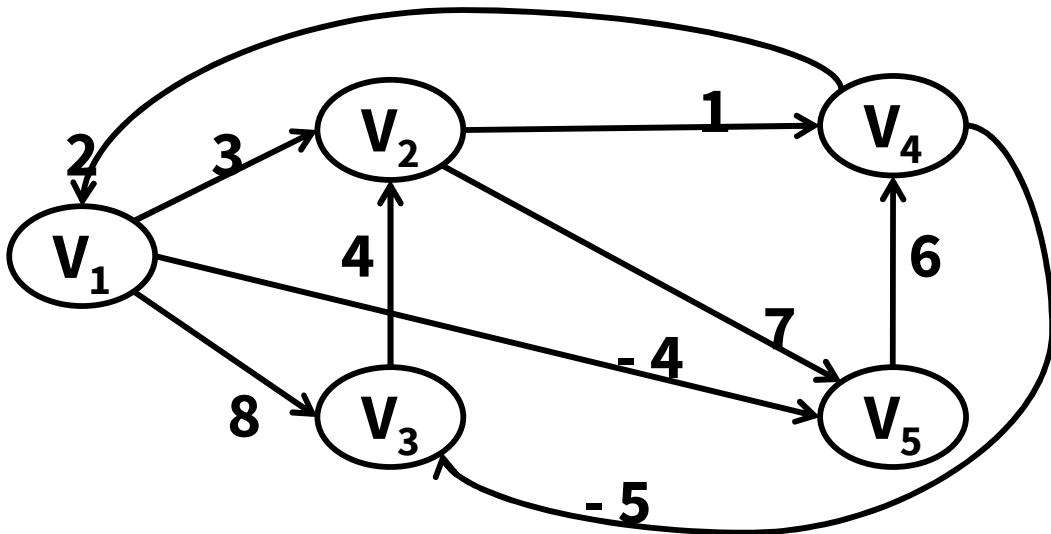
$$D[3] = \begin{vmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{vmatrix}$$

- **Step 5 :** Create a matrix  $D[4]$  using matrix  $D[3]$ . The elements in the 4<sup>th</sup> column and the 4<sup>th</sup> row are left as they are. Now,  $k=4$

$$D[4] = \begin{vmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{vmatrix}$$

$$\Pi[4] = \begin{vmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 4 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{vmatrix}$$

## Example 2 : (cont..)



$$D[4] = \begin{vmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{vmatrix}$$

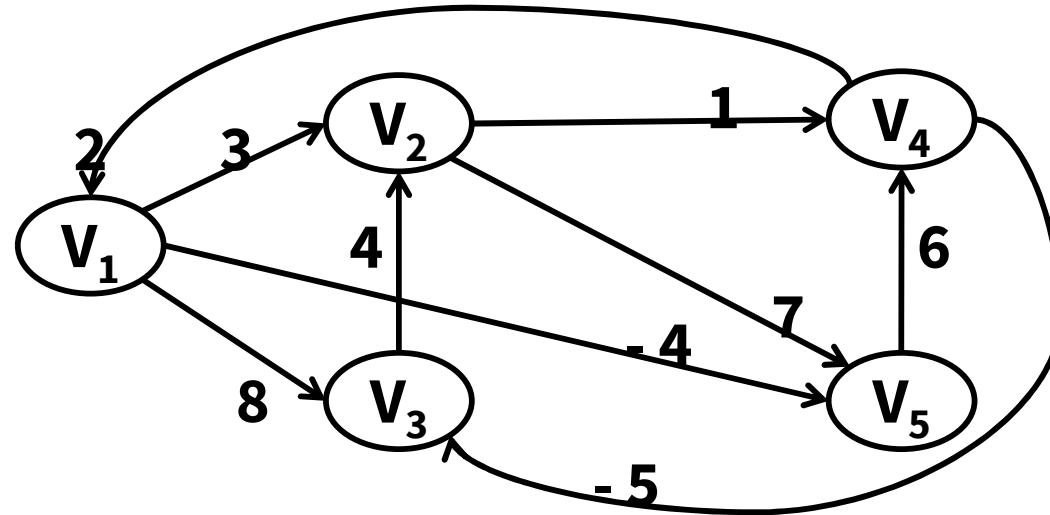
- **Step 6 :** Create a matrix  $D[5]$  using matrix  $D[4]$ . The elements in the 5<sup>th</sup> column and the 5<sup>th</sup> row are left as they are. Now,  $k=5$

$$D[5] = \begin{vmatrix} \min(3, (-4+6-5+4)) & 1 & -3 & 2 & -4 \\ 0 & \min(-1, (-4+6-5+4)) & & & \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{vmatrix}$$

$$\Pi[5] = \begin{vmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{vmatrix}$$

## Example 2 : (cont..)

- **Step 7:** In given graph we have only 5 vertices.



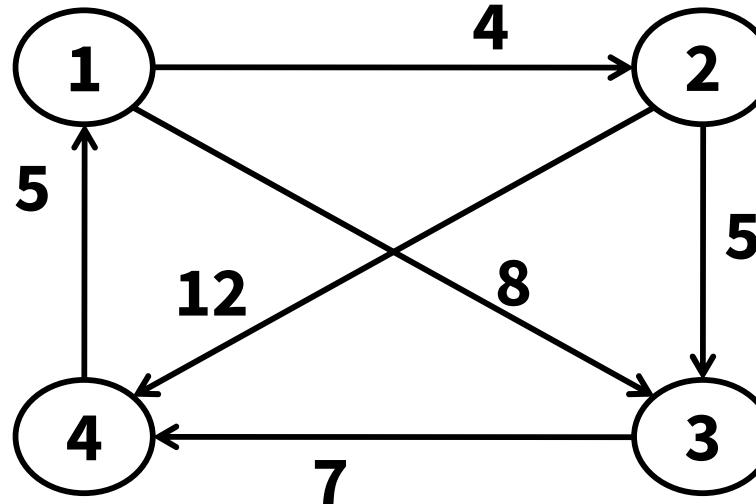
∴ D[5] gives the shortest path between each pair of vertices.

$D[5] =$	<table border="1"> <tr> <td>0</td><td>1</td><td>-3</td><td>2</td><td>-4</td></tr> <tr> <td>3</td><td>0</td><td>-4</td><td>1</td><td>-1</td></tr> <tr> <td>7</td><td>4</td><td>0</td><td>5</td><td>3</td></tr> <tr> <td>2</td><td>-1</td><td>-5</td><td>0</td><td>-2</td></tr> <tr> <td>8</td><td>5</td><td>1</td><td>6</td><td>0</td></tr> </table>	0	1	-3	2	-4	3	0	-4	1	-1	7	4	0	5	3	2	-1	-5	0	-2	8	5	1	6	0	<table border="1"> <tr> <td>NIL</td><td>3</td><td>4</td><td>5</td><td>1</td></tr> <tr> <td>4</td><td>NIL</td><td>4</td><td>2</td><td>1</td></tr> <tr> <td>4</td><td>3</td><td>NIL</td><td>2</td><td>1</td></tr> <tr> <td>4</td><td>3</td><td>4</td><td>NIL</td><td>1</td></tr> <tr> <td>4</td><td>3</td><td>4</td><td>5</td><td>NIL</td></tr> </table>	NIL	3	4	5	1	4	NIL	4	2	1	4	3	NIL	2	1	4	3	4	NIL	1	4	3	4	5	NIL
0	1	-3	2	-4																																																
3	0	-4	1	-1																																																
7	4	0	5	3																																																
2	-1	-5	0	-2																																																
8	5	1	6	0																																																
NIL	3	4	5	1																																																
4	NIL	4	2	1																																																
4	3	NIL	2	1																																																
4	3	4	NIL	1																																																
4	3	4	5	NIL																																																



## Example 3 :

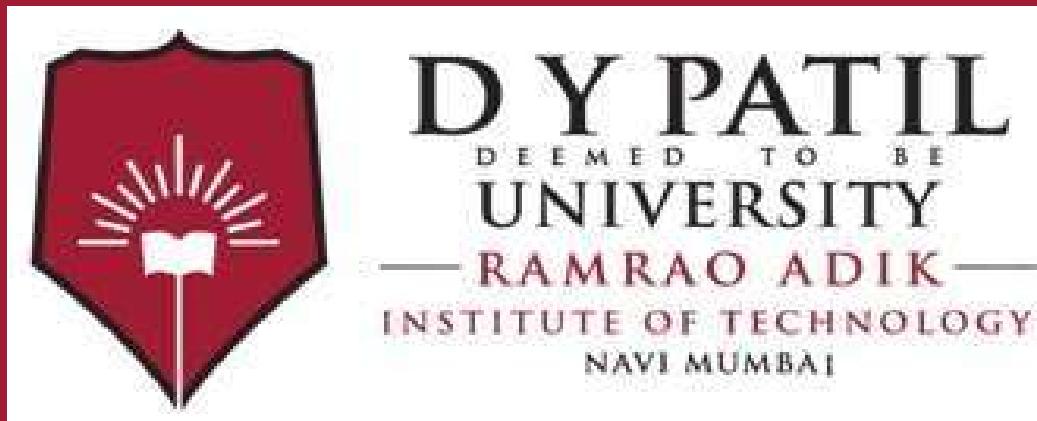
Q. Find all pair shortest path for the given graph.



→Solution :

$$D[4] = \begin{vmatrix} 0 & 4 & 8 & 15 \\ 17 & 0 & 5 & 12 \\ 12 & 16 & 0 & 7 \\ 5 & 9 & 13 & 0 \end{vmatrix}$$
$$\Pi[4] = \begin{vmatrix} \text{NIL} & 4 & 1 & 2 \\ 4 & \text{NIL} & 2 & 2 \\ 4 & 1 & \text{NIL} & 3 \\ 4 & 1 & 1 & \text{NIL} \end{vmatrix}$$





# Design and Analysis of Algorithms

## Unit 4: Dynamic Programming Approach

## Index -

---

Lecture 22 0/1 Knapsack Problem

---

Lecture 23 – Travelling Salesman Problem

Lecture 24 – Longest Common Subsequence

---



# Travelling Salesman Problem



# Travelling Salesman Problem

---

- A traveller needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once.
- The challenge of the problem is that the travelling salesman needs to minimize the total length of the trip.
- Find the shortest possible route that he visits each city exactly once and returns to the origin city.
- **Example:** A newspaper agent daily drops the newspaper to the area assigned in such a manner that he has to cover all the houses in the respective area with minimum travel cost.



# Solution to Travelling Salesman Problem

---

- We can use brute-force approach to evaluate every possible tour and select the best one.
- For  $n$  number of vertices in a graph, there are  $(n - 1)!$  number of possibilities.
- Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.



# Solution to Travelling Salesman Problem (cont..)

---

- Let us consider a graph  $G = (V, E)$ , where  $V$  is a set of cities and  $E$  is a set of weighted edges.
- An edge  $e(u, v)$  represents that vertices  $u$  and  $v$  are connected.
- Distance between vertex  $u$  and  $v$  is  $d(u, v)$ , which should be non-negative.
- Suppose we have started at city  $1$  and after visiting some cities now we are in city  $j$ . This is a partial tour.
- We need to know  $j$ , since this will determine which cities are most convenient to visit next.
- We also need to know all the cities visited so far, so that we don't repeat any of them.

# Solution to Travelling Salesman Problem (cont..)

---

- For a subset of cities  $S \in \{1, 2, 3, \dots, n\}$  that includes  $1$ , and  $i \in S$ , let  $C(i, S)$  be the length of the shortest path visiting each node in  $S$  exactly once, starting at  $1$  and ending at  $S$ .
- When  $|S| > 1$ , we define  $C(S, 1) = \infty$  since the path cannot start and end at  $1$ .
- Let us define  $C(i, S)$  in terms of smaller sub-problem.
- We should select the next city in such a way that

$$C(i, S) = \min \{ C(S - \{j\}, j) + d(i, j) \} \quad \text{where } i, j \in S \text{ and } i \neq j$$



# Algorithm Travelling Salesman Problem

---

$$C(\{1\}, 1) = 0$$

for  $s = 2$  to  $n$  do

    for all subsets  $S \in \{1, 2, 3, \dots, n\}$  of size  $s$  and containing 1

$$C(S, 1) = \infty$$

    for all  $j \in S$  and  $j \neq 1$

$$C(S, j) = \min \{ C(S - \{j\}, i) + d(i, j) \text{ for } i \in S \text{ and } i \neq j\}$$

$$\text{Return } \min C(\{1, 2, 3, \dots, n\}, j) + d(j, 1)$$

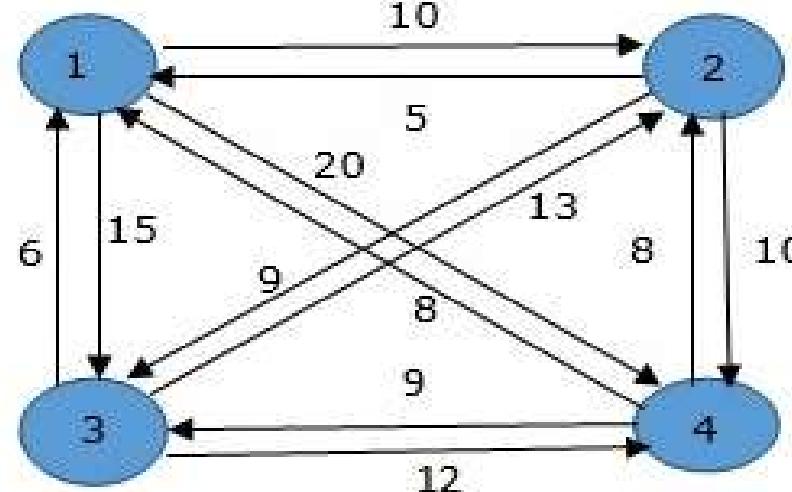
## Analysis :

- There are at the most  $2^n \cdot n$  sub-problems and each one takes linear time to solve.
- Therefore, the total running time is  $O(2^n \cdot n^2)$ .



## Example 1 :

**Q. Find the optimum cost tour for the given graph.**



**Solution :**

Step 1 : Find the distance matrix for the given graph.

	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>
<b>1</b>	0	10	15	20
<b>2</b>	5	0	9	10
<b>3</b>	6	13	0	12
<b>4</b>	8	8	9	0



## Example 1 : (cont..)

**Step 2 :** We will consider start vertex as 1. Let  $S = \emptyset$  then we have,

$$\text{Cost}(2, \emptyset, 1) = d(2, 1) = 5$$

$$\text{Cost}(3, \emptyset, 1) = d(3, 1) = 6$$

$$\text{Cost}(4, \emptyset, 1) = d(4, 1) = 8$$

**Step 3 :** Now,  $S=1$  ( means intermediate set contains 1 vertex)

Applying formula,  $\text{Cost}(i, S) = \min\{\text{Cost}(j, S - \{j\}) + d[i, j]\}$

$$\text{Cost}(2, \{3\}, 1) = d[2, 3] + \text{Cost}(3, \emptyset, 1) = 9 + 6 = 15$$

$$\text{Cost}(2, \{4\}, 1) = d[2, 4] + \text{Cost}(4, \emptyset, 1) = 10 + 8 = 18$$

$$\text{Cost}(3, \{2\}, 1) = d[3, 2] + \text{Cost}(2, \emptyset, 1) = 13 + 5 = 18$$

$$\text{Cost}(3, \{4\}, 1) = d[3, 4] + \text{Cost}(4, \emptyset, 1) = 12 + 8 = 20$$

$$\text{Cost}(4, \{3\}, 1) = d[4, 3] + \text{Cost}(3, \emptyset, 1) = 9 + 6 = 15$$

$$\text{Cost}(4, \{2\}, 1) = d[4, 2] + \text{Cost}(2, \emptyset, 1) = 8 + 5 = 13$$



## Example 1 : (cont..)

**Step 4 :** Consider  $S = 2$  then we have,

$$\begin{aligned}\text{Cost}(2,\{3, 4\}, 1) &= \min \{ [d[2,3] + \text{Cost}(3,\{4\},1)], [d[2,4] + \text{Cost}(4,\{3\},1)] \} \\ &= \min \{ (9 + 20), (10 + 15) \} \\ &= 25\end{aligned}$$

$$\begin{aligned}\text{Cost}(3,\{2, 4\}, 1) &= \min \{ [d[3,2] + \text{Cost}(2,\{4\},1)], [d[3,4] + \text{Cost}(4,\{2\},1)] \} \\ &= \min \{ (13 + 18), (12 + 13) \} \\ &= 25\end{aligned}$$

$$\begin{aligned}\text{Cost}(4,\{2, 3\}, 1) &= \min \{ [d[4,2] + \text{Cost}(2,\{3\},1)], [d[4,3] + \text{Cost}(3,\{2\},1)] \} \\ &= \min \{ (8 + 15), (9 + 18) \}\end{aligned}$$



## Example 1 : (cont..)

**Step 5 :** Consider  $S = 3$  i.e. We have to find  $\text{Cost}(1, \{2, 3, 4\})$ .

In TSP we have to complete the tour where we have started i.e. start and end vertex should be same. In this case start and end vertex =1.

$$\begin{aligned}\text{Cost}(1, \{2, 3, 4\}, 1) &= \min \{ [d[1, 2] + \text{Cost}(2, \{3, 4\}, 1)], \\ &\quad [d[1, 3] + \text{Cost}(3, \{2, 4\}, 1)], \\ &\quad [d[1, 4] + \text{Cost}(4, \{2, 3\}, 1)] \\ &= \min \{ (10 + 25), (15 + 25), (20 + 23) \} \\ &= 35\end{aligned}$$

**Thus, the optimal tour is of length =35**

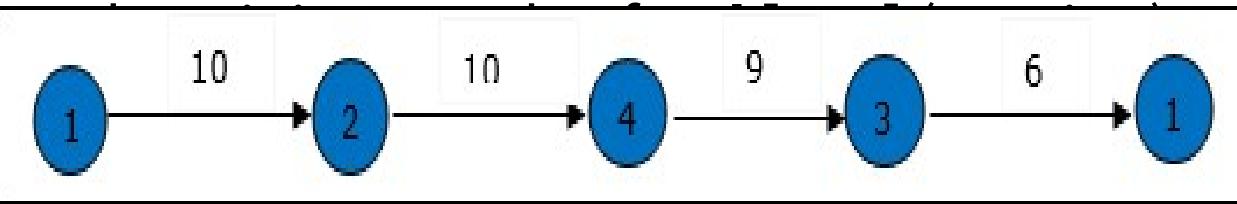


## Example 1 : (cont..)

### Step 6 : Finding the tour path.

- Start from cost  $\{1, \{2, 3, 4\}, 1\}$ , we get the minimum value for  $\mathbf{d [1, 2]}$ .
- i.e. When  $s = 3$ , we get optimum path from 1 to 2. So select vertex 2 (cost is 10) then go backwards.
- When  $s = 2$ , we get the minimum value for  $\mathbf{d [4, 2]}$ . So select vertex 4 (cost is 10) again go backwards.
- When  $s = 1$ , we get the minimum value for  $\mathbf{d [4, 3]}$ . Selecting path 4 to 3 (cost is 9) and vertex 4. Go backward.
- Now,  $S = \Phi$  means we don't have any vertex. Going back to vertex 1 from vertex 3.

• We g

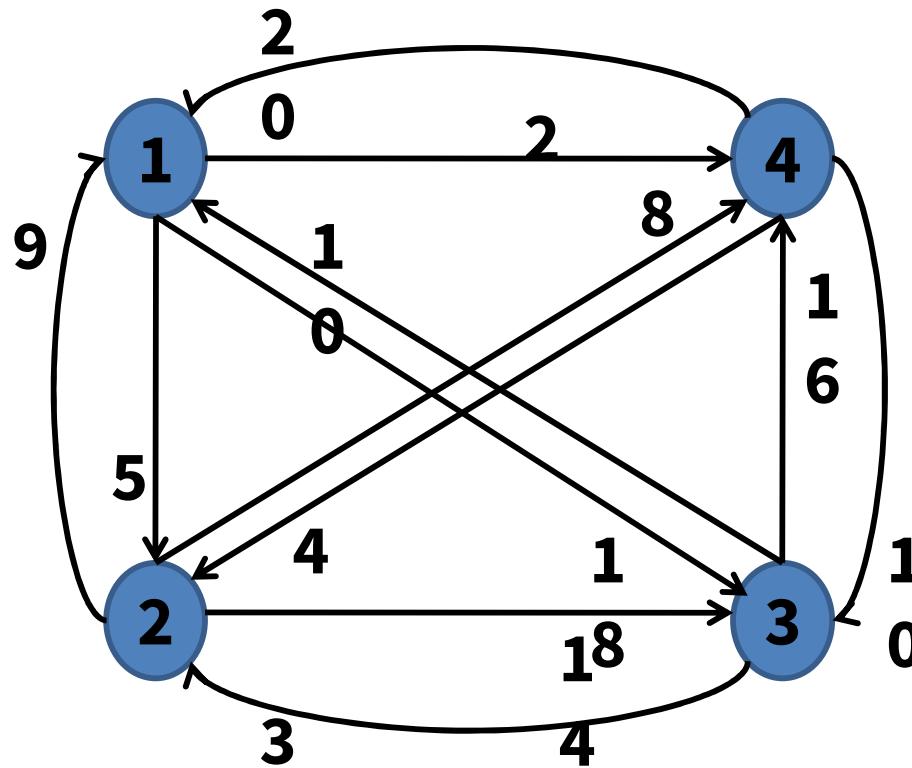


• Hen



## Example 2 :

Q. Find the optimum cost tour for the given graph.



Solution :

Optimal Tour Path Cost = 24

Optimal Tour Path = 1 → 4 → 3 → 2 → 1



D Y PATIL  
DEEMED TO BE  
UNIVERSITY  
RAMRAO ADIK  
INSTITUTE OF TECHNOLOGY  
NAVI MUMBAI

# Applications of TSP

---

- ❑ Vehicle routing.
- ❑ Robotic welding in the car industry.
- ❑ Logistics and packaging
- ❑ Planning
- ❑ Scheduling



# Longest Common Subsequence



# Longest Common Subsequence

---

- A subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements. Longest common subsequence (*LCS*) of 2 sequences is a subsequence, with maximal length, which is common to both the sequences.
- **Subsequence** and **Sub-String** are different.
- A substring **s'** of a string **s** is an ordered sequence of consecutive elements of **s**.
  - E.g : **010** is substring of **01111010000**
- Whereas **Subsequence s'** of a string **s** is an ordered sequence of elements of **s** in increasing order but not necessarily in consecutive order.
  - E.g : **010** is subsequence of **011111110**
- In the longest-common-subsequence problem, we are given two sequences  $X = x_1, x_2, \dots, x_m$  and  $Y = y_1, y_2, \dots, y_n$  and wish to find a **maximum-length common subsequence** of  $X$  and  $Y$ .



## Longest Common Subsequence (cont..)

---

- For example, if  $X = \langle A, B, C, B, D, A, B \rangle$  and  $Y = \langle B, D, C, A, B, A \rangle$  , the sequence  $B, C, A$  is a common subsequence of both X and Y.
- But the sequence  $B, C, A$  is not a longest common subsequence (LCS) of X and Y, however, since it has length 3 and the sequence  $B, C, B, A$  , which is also common to both X and Y, has length 4.
- The sequence  $B, C, B, A$  is an LCS of X and Y, as is the sequence  $B, D, A, B$  ,
- However there is no common subsequence of length 5 or greater.

# Longest Common Subsequence Algorithm –

## Recursive Approach

- Start comparing strings in reverse order one character at a time.
- Now we have 2 cases –
  - Both characters are same
    - Add 1 to the result and remove the last character from both the strings and make recursive call to the modified strings.
  - Both characters are different
    - Remove the last character of String 1 and make a recursive call and remove the last character from String 2 and make a recursive and then return the max from returns of both recursive calls. see example below



# Longest Common Subsequence Algorithm –

## Recursive Approach (cont..)

**Example:**

- **Case 1:**

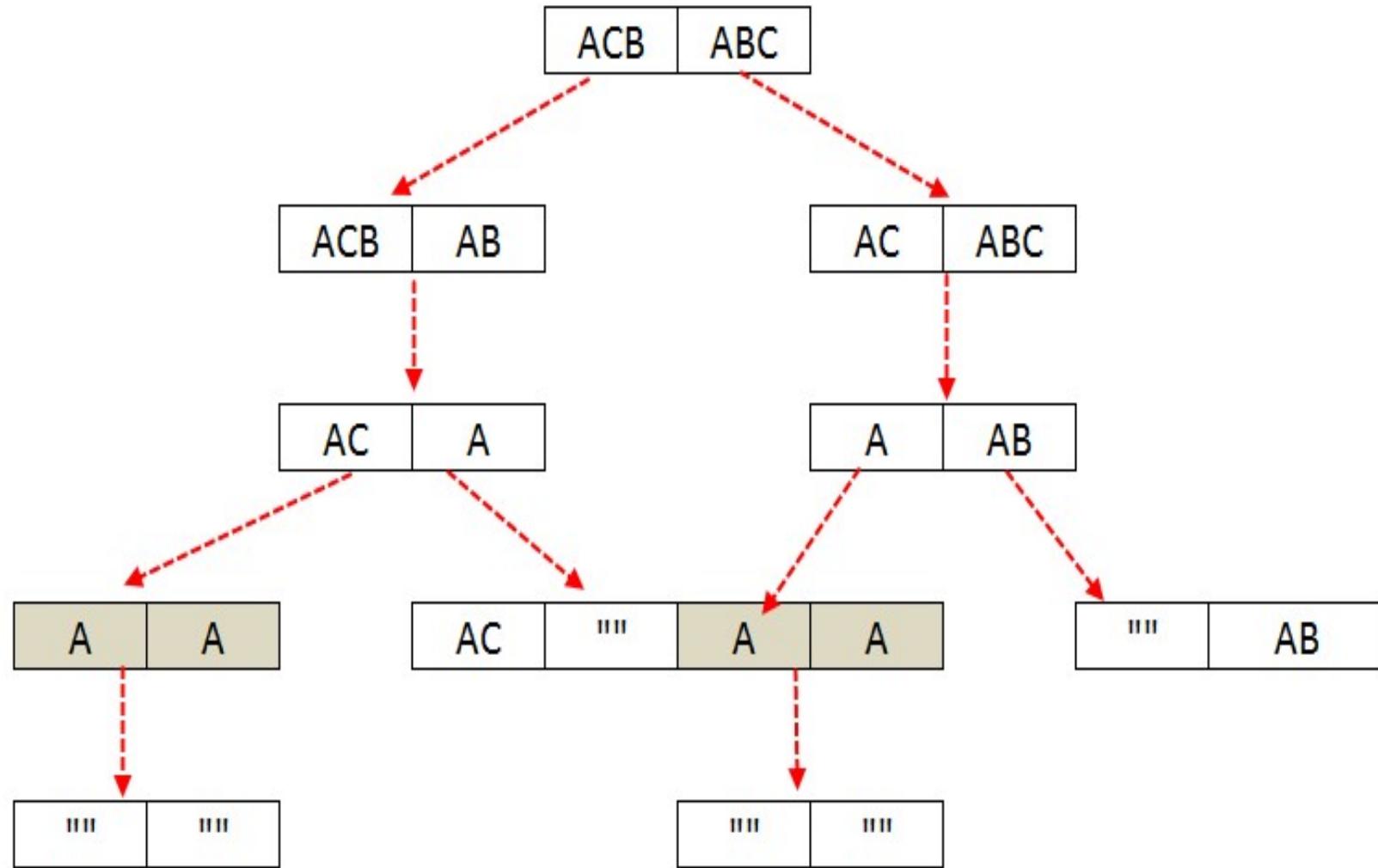
- String A: "ABCD", String B: "AEBD"
- $\text{LCS}(\text{"ABCD"}, \text{"AEBD"}) = 1 + \text{LCS}(\text{"ABC"}, \text{"AEB"})$

- **Case 2:**

- String A: "ABCDE", String B: "AEBDF"
- $\text{LCS}(\text{"ABCDE"}, \text{"AEBDF"}) = \text{Max}(\text{LCS}(\text{"ABCDE"}, \text{"AEBD"}), \text{LCS}(\text{"ABCD"}, \text{"AEBDF"}))$
- In a given string of length n, there can be  $2^n$  subsequences can be made, so if we do it by recursion then Time complexity will  $O(2^n)$  since we will solving sub problems repeatedly.



# Longest Common Subsequence Algorithm – Recursive Approach (cont..)



# Longest Common Subsequence Algorithm – Dynamic Programming Approach

---

- The Longest Common Subsequence problem is a very well known problem, and has a classic example of a dynamic programming solution.
- **Dynamic Programming** will store the solution of the sub problems in a solution array and use it when ever needed, This technique is called Memorization.
- Let  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Y = \langle y_1, y_2, \dots, y_n \rangle$  be sequences, and we wish to find LCS of X and Y.
- The optimal substructure of the LCS problem gives the recursive formula as,

$$c[i, j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \end{cases}$$



# Longest Common Subsequence Algorithm

## //Computing the Length of LCS

```
LCS-LENGTH ( $X, Y$ )
1  $m \leftarrow \text{length}[X]$ 
2  $n \leftarrow \text{length}[Y]$ 
3 for  $i \leftarrow 1$  to  $m$ 
4   do  $c[i, 0] \leftarrow 0$ 
5 for  $j \leftarrow 0$  to  $n$ 
6   do  $c[0, j] \leftarrow 0$ 
7 for  $i \leftarrow 1$  to  $m$ 
8   do for  $j \leftarrow 1$  to  $n$ 
9     do if  $x_i = y_j$ 
10       then  $c[i, j] \leftarrow c[i - 1, j - 1] + 1$ 
11        $b[i, j] \leftarrow "\square"$ 
12     else if  $c[i - 1, j] \geq c[i, j - 1]$ 
13       then  $c[i, j] \leftarrow c[i - 1, j]$ 
14        $b[i, j] \leftarrow "\uparrow"$ 
15     else  $c[i, j] \leftarrow c[i, j - 1]$ 
16        $b[i, j] \leftarrow \leftarrow$ 
17 return  $c$  and  $b$ 
```

TIL  
TECHNICAL INSTITUTE OF TECHNOLOGY



RAMRAO ADIK  
INSTITUTE OF TECHNOLOGY  
NAVI MUMBAI

# Longest Common Subsequence Algorithm

---

## //Constructing the LCS

```
PRINT-LCS( $b, X, i, j$ )
1 if  $i = 0$  or  $j = 0$ 
2   then return
3 if  $b[i, j] = \square$ 
4   then PRINT-LCS( $b, X, i - 1, j - 1$ )
5     print  $x_i$ 
6 elseif  $b[i, j] = \uparrow$ 
7   then PRINT-LCS( $b, X, i - 1, j$ )
8 else PRINT-LCS( $b, X, i, j - 1$ )
```



## Example 1 :

Q. Find the longest subsequence for string ABCBDAB and BDCABA.

Solution :

- **Step 1 :** Create a table of dimension  $(n+1) \times (m+1)$  where n and m are the lengths of X and Y respectively. The first row and the first column are filled with zeros.

- In given

	0	0	0	0	0	0	0
0							
0							
0							
0							
0							
0							

## Example 1 : (cont..)

---

- **Step 2 :** Fill each cell of the table using the following logic.

**1.** If the character corresponding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.

**2.** If the characters are not equal then compare values of previous column and previous row element for filling the current cell.

**2.A]** If values are equal then keep that value and mark direction as upward i.e. Point to upward element.

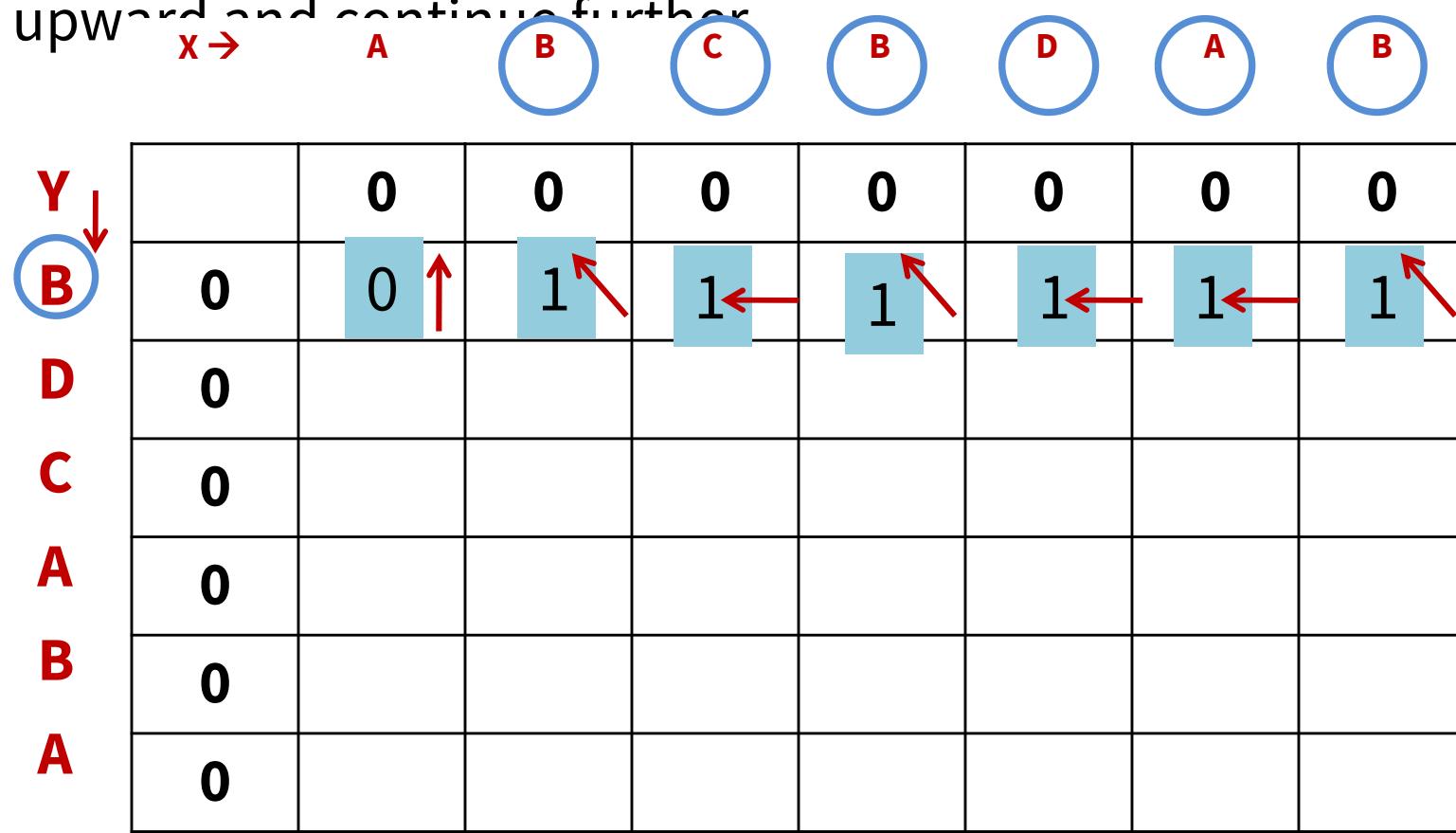
**2.B]** If upper value is greater then keep that value and mark direction as upward.

**2.C]** If LHS is greater than upper value then keep LHS value and mark direction as left arrow pointing to LHS element.

<sup>26</sup> Based on this logic will start filling the table by comparing 2  
Lecture No 26: Longest common  
Subsequence

## Example 1 : (cont..)

- **Step 3:** Comparing X[1] with Y[1] i.e. comparing “A” and “B”.
  - Both characters are not equal so comparing values.
  - Both values are same so keep that as the same and mark direction as upward and continue further



## Example 1 : (cont..)

- **Step 4 :** Comparing Y[2] with X sequence characters and based on that will update data in cell.

$x \rightarrow$       

	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1
D	0	0	1	1	1	2	2
C	0						
A	0						
B	0						
A	0						



## Example 1 : (cont..)

- **Step 5 :** Comparing Y[3] with X sequence characters and based on that will update data in cell.

$x \rightarrow$       

	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1
D	0	0	1	1	1	2	2
	0	0	1	2	2	2	2
A							
B							
A							



## Example 1 : (cont..)

- **Step 6 :** Comparing Y[4] with X sequence characters and based on that will update data in cell.

$x \rightarrow$       A    B    C    B    D    A    B

	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1
D	0	0	1	1	1	2	2
C	0	0	1	1	1	2	2
(A)	0	0	1	2	2	2	2
A	0	1	1	2	2	3	3
B	0						
A	0						



## Example 1 : (cont..)

- **Step 7 :** Comparing Y[5] with X sequence characters and based on that will update data in cell.

$x \rightarrow$       

	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1
D	0	0	1	1	1	2	2
C	0	0	1	2	2	2	2
A	0	1	1	2	2	3	3
	1	2	2	3	3	3	4
A	0						



## Example 1 : (cont..)

- **Step 8 :** Comparing Y[6] with X sequence characters and based on that will update data in cell.

$x \rightarrow$       

	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1
D	0	0	1	1	1	2	2
C	0	0	1	2	2	2	2
A	0	1	1	2	2	3	3
B	0	1	2	2	3	3	4
(A)	0	1	2	2	3	3	4



## Example 1 : (cont..)

- **Step 9 :** The value in the last row and the last column(i.e. Bottom right corner) is the length of the longest common subsequence.  
Here,  $\text{LCS}(x, y) = 4$

	0	0	0	0	0	0	0
B	0	1↑	1←	1↑	1←	1←	1↑
D	0	0↑	1↑	1↑	1↑	2↑	2↑
C	0	0↑	1↑	2↑	2↑	2↑	2↑
A	0	1↑	1↑	2↑	2↑	2↑	3↑
B	0	1↑	2↑	2↑	3↑	3↑	4↑
A	0	1↑	2↑	2↑	3↑	3↑	4↑

## Example 1 : (cont..)

- Step 10 :** In order to find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to diagonal arrow in the cell form the longest

x →      A      B      C      B      D      A      B

	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1
D	0	0	1	1	1	2	2
C	0	0	1	1	2	2	2
A	0	1	1	2	2	2	3
B	0	1	2	2	3	3	3
A	0	1	2	2	3	3	4

Y ↓

Start here

**Longest Common Subsequence =**  
**BDAB**



## Example 2 :

Q. Find the longest subsequence for given string.

$X[ ] = \text{"AGGTAB"}$

$Y[ ] = \text{"GXTXAYB"}$

Solution :

		G	X	T	X	A	Y	B
	0	0	0	0	0	0	0	0
A	0	0	0	0	0	1	1	1
G	0	1	1	1	1	1	1	1
G	0	1	1	1	1	1	1	1
T	0	1	1	2	2	2	2	2
A	0	1	1	2	2	3	3	3
B	0	1	1	2	2	3	3	4

LCS Length = 4

LCS = GTAB



## Example 3 :

**Q. Find the longest subsequence for string ABCDA and ACBDEA.**

**Solution :**

	A	B	C	D	A	
A	0	0	0	0	0	0
C	0	1	1	1	1	1
B	0	1	2	2	2	2
D	0	1	2	2	3	3
E	0	1	2	2	3	3
A	0	1	2	2	3	4

**LCS Length =  
4 LCS = ACDA**

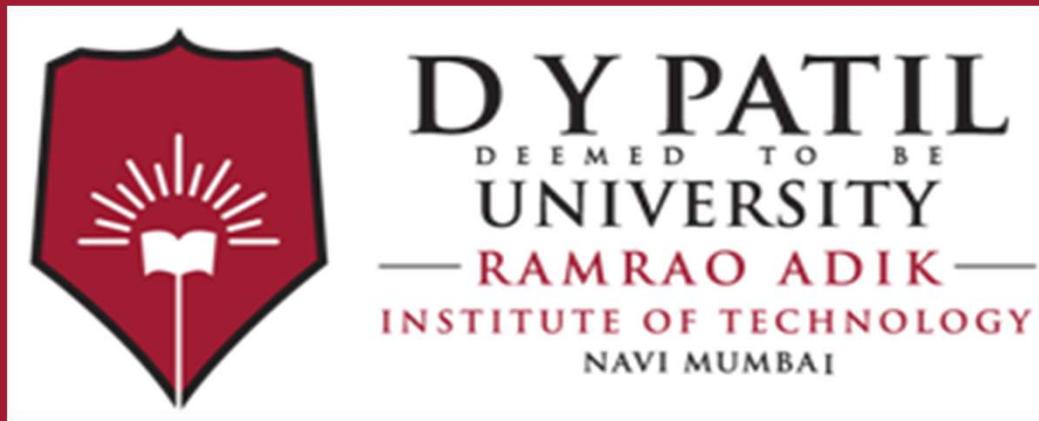


# Analysis of Longest Common Subsequence Algorithm

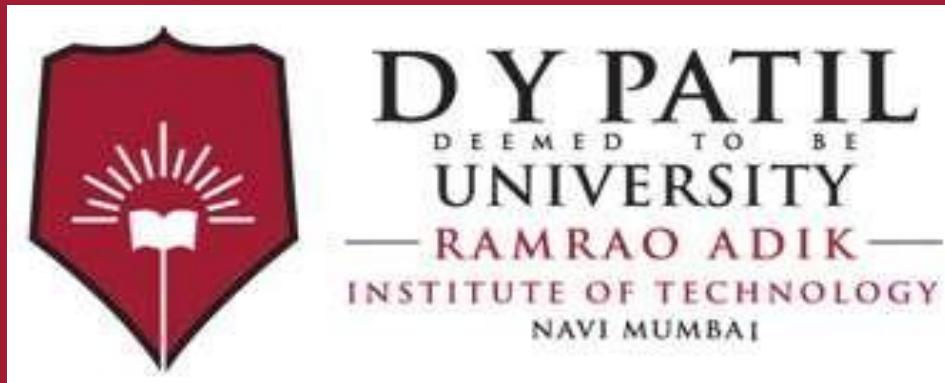
---

- The method of dynamic programming reduces the number of function calls.
- It stores the result of each function call so that it can be used in future calls without the need for redundant calls.
- In the dynamic algorithm, the results obtained from each comparison between elements of X and the elements of Y are stored in a table so that they can be used in future computations.
- So, the time taken by a dynamic approach is the time taken to fill the table i.e.  $O(mn)$ . Whereas, the recursion algorithm has the complexity of  $2^{\max(m, n)}$ .





# Thank You



# Analysis of Algorithms

## Unit 4: Backtracking and Branch and Bound

## **Index -**

---

Lecture 25 - Introduction to Backtracking

Lecture 26 – 8 queen problem( N-queen problem)

Lecture 27 – Sum of subsets

---



## Module No 4: Backtracking and Branch and Bound

---

# Lecture No: 25

# Introduction to Backtracking



# Backtracking

---

- Suppose you have to make a series of *decisions*, among various *choices*, where
  - You don't have enough information to know what to choose
  - Each decision leads to a new set of choices
  - Some sequence of choices (possibly more than one) may be a solution to your problem
- **Backtracking** is a methodical way of trying out various sequences of decisions, until you find one that “works”.
- Backtracking technique can be considered as an organized exhaustive search that often avoids searching all possibilities.



# Backtracking algorithm (contd..)

---

- It uses recursive calling to find the solution by building a solution step by step increasing values with time.
- It removes the solutions that doesn't give rise to the solution of the problem based on the constraints given to solve the problem.
- In backtracking problem, the algorithm tries to find a sequence path to the solution which has some small checkpoints from where the problem can backtrack if no feasible solution is found for the problem.
- Backtracking algorithm is applied to some specific types of problems,
  - Decision problem used to find a feasible solution of the problem.
  - Optimisation problem used to find the best solution that can be applied.
  - Enumeration problem used to find the set of all feasible solutions of the problem.



# Example 1: Sudoku puzzle

---

- Common puzzle that can be solved by backtracking is a Sudoku puzzle.
- The basic idea behind the solution is as follows:
  - Scan the board to look for an empty square that could take on the fewest possible values based on the simple game constraints.
  - If you find a square that can only be one possible value, fill it in with that one value and continue the algorithm.
  - If no such square exists, place one of the possible numbers for that square in the number and repeat the process.
  - If you ever get stuck, erase the last number placed and see if there are other possible choices for that slot and try those next.



## Example 2: Maze

---

- Find path through maze
  - Start at beginning of maze
  - If at exit, return true
  - Else for each step from current location
    - Recursively find path
    - Return with first successful step
    - Return false if all steps fail



# Backtracking Approach

---

- Construct the **State Space Tree**:
  - Root represents an initial state
  - Nodes reflect specific choices made for a solution's components.
    - Promising and non promising nodes
    - leaves
- Explore the state space tree using depth-first search
- “Prune” non-promising nodes
  - DFS stops exploring sub tree rooted at nodes leading to no solutions and “backtracks” to its parent node



# Backtracking Algorithm – Recursive Approach

---

**Algorithm** Backtrack( $k$ )

```
// This schema describes the backtracking process using
// recursion. On entering, the first  $k - 1$  values
//  $x[1], x[2], \dots, x[k - 1]$  of the solution vector
//  $x[1 : n]$  have been assigned.  $x[ ]$  and  $n$  are global.
{
    for (each  $x[k] \in T(x[1], \dots, x[k - 1])$ ) do
    {
        if ( $B_k(x[1], x[2], \dots, x[k]) \neq 0$ ) then
        {
            if ( $x[1], x[2], \dots, x[k]$  is a path to an answer node)
                then write ( $x[1 : k]$ );
            if ( $k < n$ ) then Backtrack( $k + 1$ );
        }
    }
}
```



# Backtracking Algorithm - Iterative Approach

---

**Algorithm** |Backtrack( $n$ )

```
// This schema describes the backtracking process.  
// All solutions are generated in  $x[1 : n]$  and printed  
// as soon as they are determined.  
{  
     $k := 1;$   
    while ( $k \neq 0$ ) do  
    {  
        if (there remains an untried  $x[k] \in T(x[1], x[2], \dots, x[k-1])$  and  $B_k(x[1], \dots, x[k])$  is true) then  
        {  
            if ( $x[1], \dots, x[k]$  is a path to an answer node)  
                then write ( $x[1 : k]$ );  
             $k := k + 1;$  // Consider the next set.  
        }  
        else  $k := k - 1;$  // Backtrack to the previous set.  
    }  
}
```



# Backtracking - Conclusion

---

- Backtracking provides the hope to solve some problem instances of nontrivial sizes by pruning non-promising branches of the state-space tree.
- The success of backtracking varies from problem to problem and from instance to instance.
- Backtracking possibly generates all possible candidates in an exponentially growing state-space tree.



## Module No 4: Backtracking and Branch and Bound

---

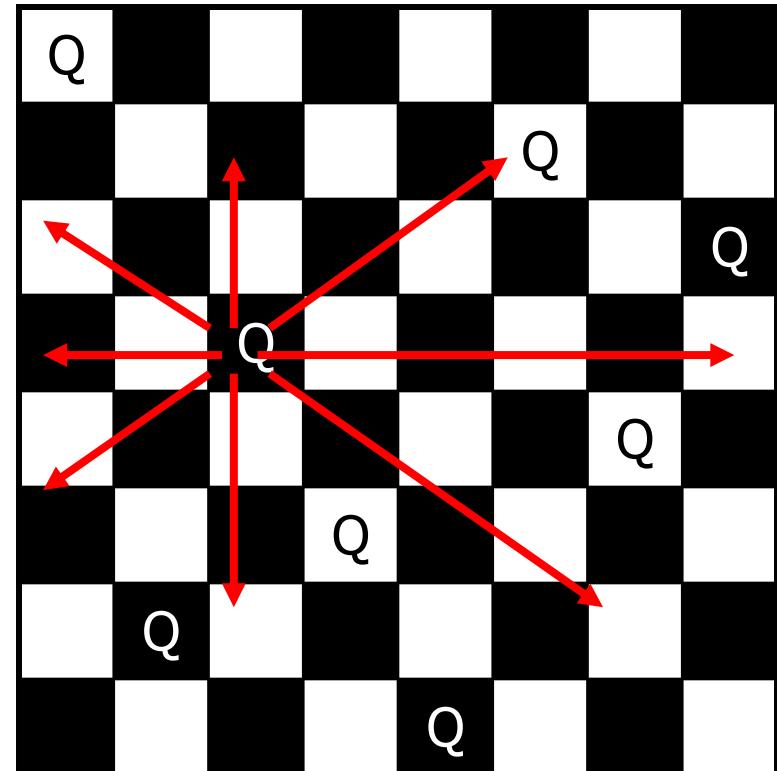
# Lecture No: 26

# N Queen Problem



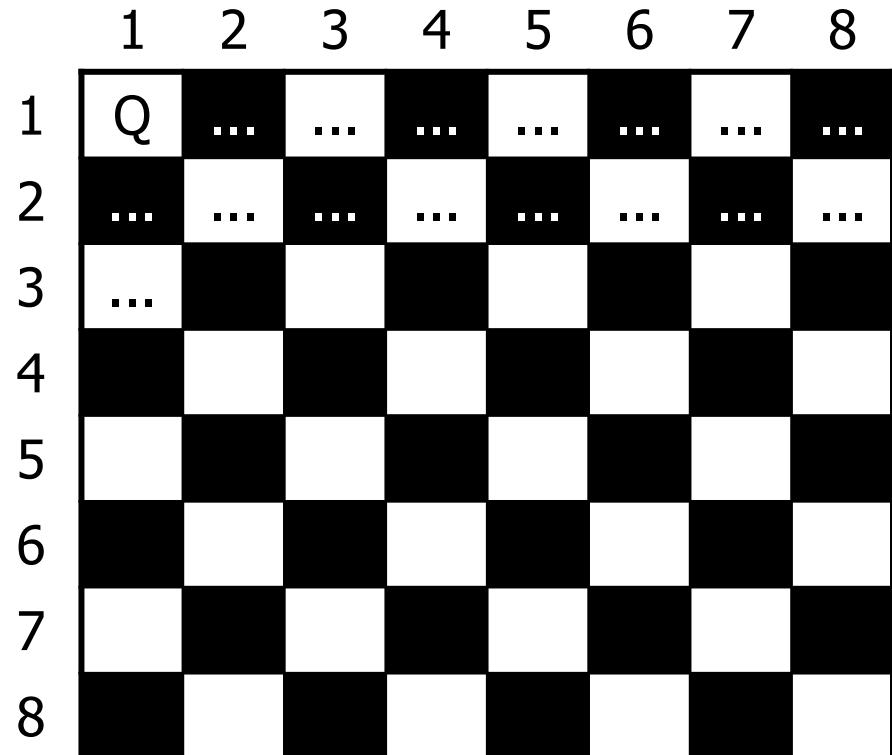
# The N Queen Problem

- **N-Queens Problem:** Given a chess board having  $N \times N$  cells, we need to place  $N$  queens in such a way that no queen is attacked by any other queen.
- A queen can attack horizontally, vertically and diagonally.
- Consider the problem of trying to place 8 queens on a chess board such that no queen can attack another queen.
  - What are the "choices"?
  - How do we "make" or "un-make" a choice?
  - How do we know when to stop?



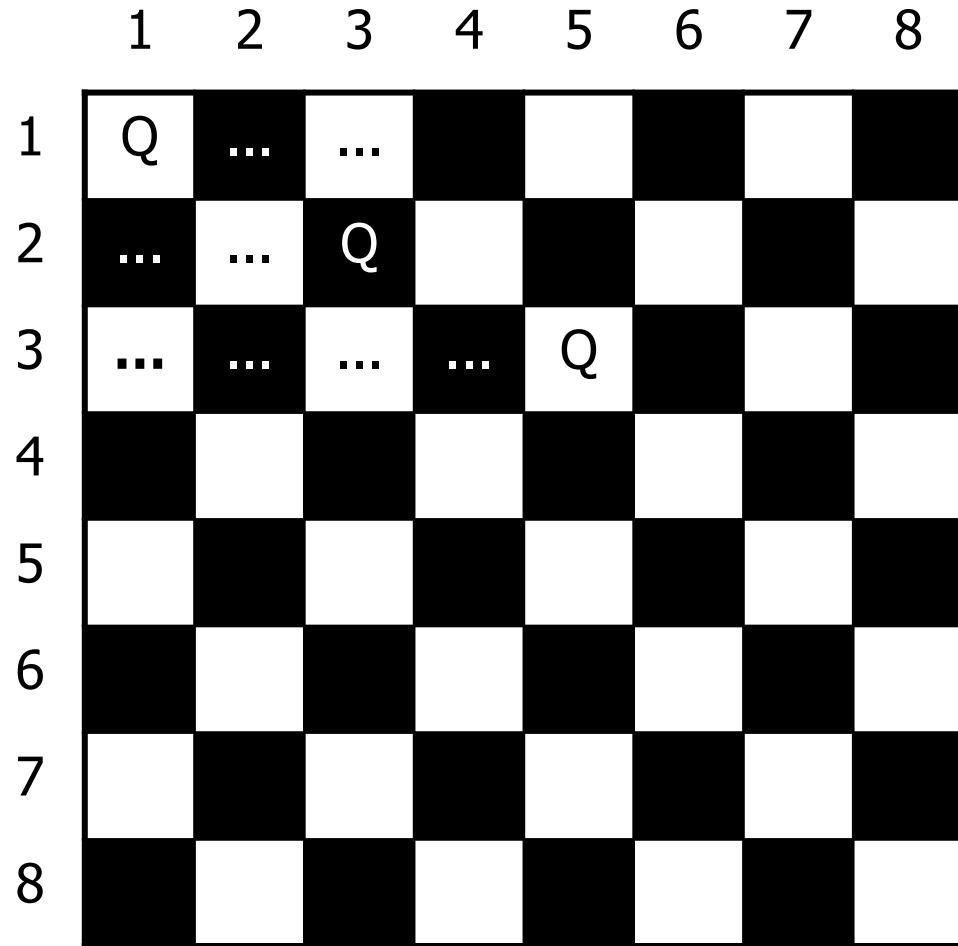
# Naïve Approach

- for (each square on board):
  - Place a queen there.
  - Try to place the rest of the queens.
  - Un-place the queen.
  - How large is the solution space for this algorithm?
    - $64 * 63 * 62 * \dots$



# Improved Approach

- Efficiency can be improved if we place exactly 1 queen in each row and in each column.
  - Redefine a "choice" to be valid placement of a queen in a particular column.
  - In this case, how large is the solution space now?
    - $8 * 8 * 8 * \dots$



# Backtracking Approach

---

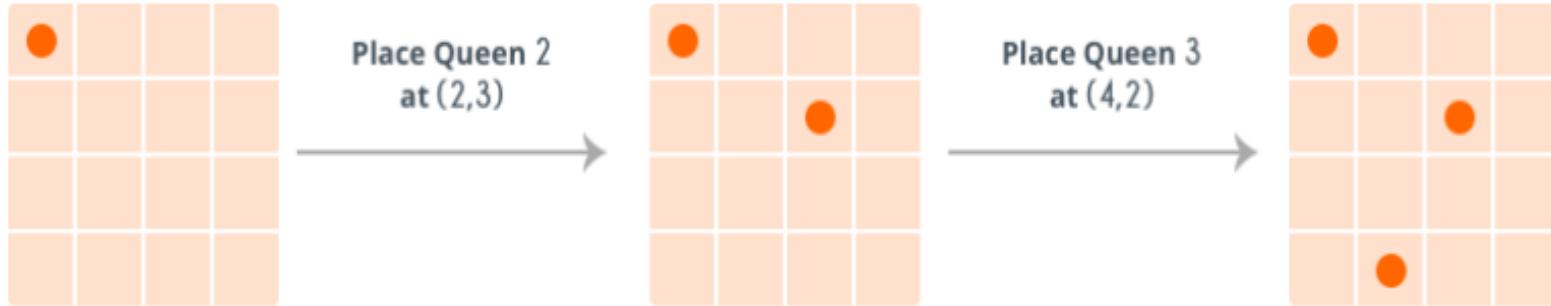
- Initially we are having  $N \times N$  unattacked cells where we need to place  $N$  queens.
- Let's place the first queen at a cell  $(i,j)$ , so now the number of unattacked cells is reduced, and number of queens to be placed is  $N-1$ .
- Place the next queen at some unattacked cell.
- This again reduces the number of unattacked cells and number of queens to be placed becomes  $N-2$ .
- Continue doing this, as long as following conditions hold.
  - The number of unattacked cells is not 0.
  - The number of queens to be placed is not 0.
- If the number of queens to be placed becomes 0, then it's over, we found a solution.
- If the number of unattacked cells become 0, then we need to backtrack, i.e. remove the last placed queen from its current cell, and place it at some other cell.

• !)

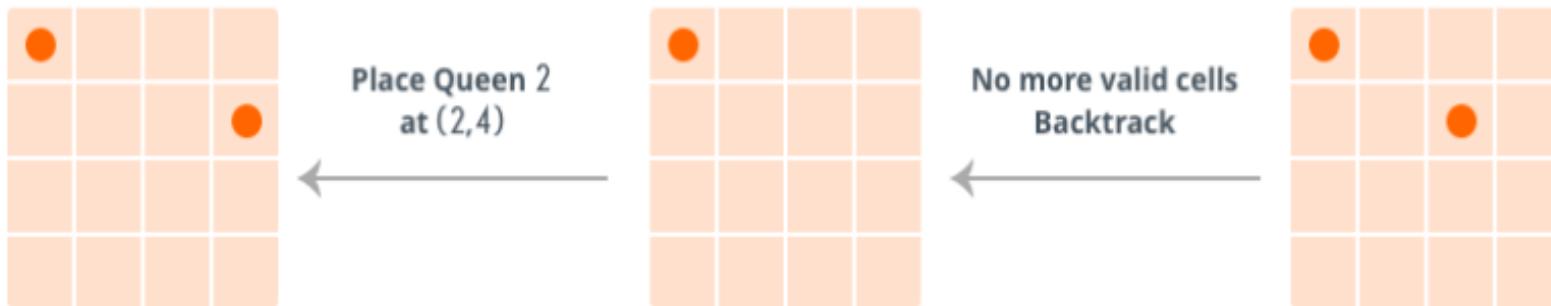


# 4 X 4 Queen

- Let's see how it works when no. of queen, N =4.
- Consider a 4 x 4 chessboard and Place 1<sup>st</sup> Queen at (1,1).

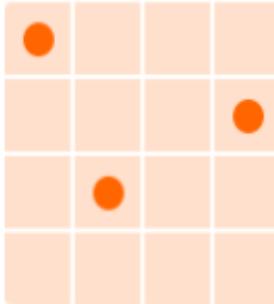


- We don't have any valid cells to place 4<sup>th</sup> Queen so backtrack.
- Remove 3<sup>rd</sup> Queen from chessboard and update the position of 2<sup>nd</sup> Queen.

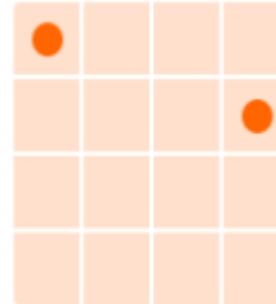


# 4 X 4 Queen (cont..)

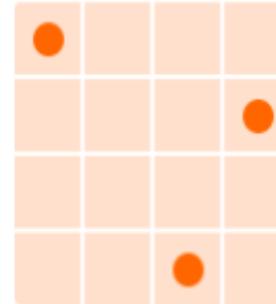
- Place 3<sup>rd</sup> Queen at (3, 2).



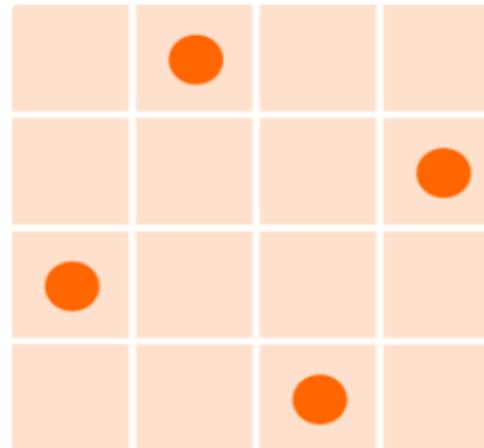
No more valid cells  
Backtrack



Place Queen 3  
at (4,3)



- Still we don't have any valid cells to place 4<sup>th</sup> Queen so backtrack.
- Remove 3<sup>rd</sup> Queen from chessboard and backtrack.
- Remove 2<sup>nd</sup> Queen from chessboard and backtrack.
- Now, update the position of 1<sup>st</sup> Queen. At the end we get solution as,



# Algorithm for N Queens Problem

---

// This algorithm prints all possible placements of n queens on nXn chessboard so that they are non-attacking.

**Algorithm N - Queens (k, n)**

{

    For i  $\leftarrow$  1 to n

        do if Place (k, i) then

{

            x [k]  $\leftarrow$  i;

            if (k ==n) then

                write (x [1....n]);

            else

                N - Queens (k + 1, n);

}

}



# Algorithm for N Queens Problem (cont..)

---

## Algorithm Place (k, i)

```
{  
    For j ← 1 to k - 1  
    do if (x [j] = i)  
        or (Abs (x [j] - i)) = (Abs (j - k))  
    then return false;  
    return true;  
}
```

- Place (k, i) returns a Boolean value that is true if the  $k^{\text{th}}$  queen can be placed in column i.
- It tests both whether i is distinct from all previous costs  $x_1, x_2, \dots, x_{k-1}$  and whether there is no other queen on the same diagonal.
- Using place, we give a precise solution to then n- queens problem.
- $x []$  is a global array whose final  $k - 1$  values have been set.
- Abs (r) returns the absolute value of r.

## Analysis of N-Queen Problem

---

- Number of possible arrangements of N Queens on  $N \times N$  chessboard is  $N!$ , as we are skipping row or column, already having a queen placed.
- So average and worst case complexity of the solution is  $O(N!)$ .
- The best case occurs if you find your solution before exploiting all possible arrangements.
- If we need all the possible solutions, the best, average and worst case complexity remains  $O(N!)$ .



## **Module No 4: Backtracking and Branch and Bound**

---

# **Lecture No: 27**

## **Sum of Subsets**



# Sum of Subset Problem

---

- In this problem, there is a given set with some integer elements. And another some value is also provided, we have to find a subset of the given set whose sum is the same as the given sum value.
- In this problem we have to find a subset  $s'$  of the given set  $S = (S_1, S_2, S_3, \dots, S_n)$  where the elements of the set 'S' are 'n' positive integers in such a manner that  $s' \in S$  and sum of the elements of subset  $s'$  is equal to some positive integer 'X'.
- **For example:**

We have a set of numbers, and a sum value.

→ Set  $S = \{10, 7, 5, 18, 12, 20, 15\}$

→ Sum Value = 35

→ All possible subsets of the given set, where sum of each element for every subsets is same as the given sum value.

→  $\{10, 7, 18\} \{10, 5, 20\} \{5, 18, 12\} \{20, 15\}$



# Sum of Subset Problem (cont..)

---

- A naive solution would be to cycle through all subsets of n numbers and for every one of them, check if the subset sums to the right number.
- The running time is of order  $O(2^n \cdot n)$  since there are  $2^n$  subsets and to check each subset we need to sum at most n elements.
- The Subset-Sum Problem can be solved by using the backtracking approach.
- Here backtracking approach is used for trying to select a valid subset when an item is not valid, we will backtrack to get the previous subset and add another element to get the solution.
- The running time complexity is  $O(2^n)$ .



# Procedure to Solve Sum of Subset Problem

---

1. Start with an empty set.
2. Add the next element from the list to the set
3. If the subset is having sum M, then stop with that subset as solution.
4. If the subset is not feasible or if we have reached the end of the set, then backtrack through the subset until we find the most suitable value.
5. If the subset is feasible ( $\text{sum of subset} < M$ ) then go to step 2.
6. If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.



# Example 1:

Q. Solve following problem to find subsets from the given set of elements,  $W = \{5, 10, 12, 13, 15, 18\}$  which will give sum,  $M=30$  and draw portion of state space tree.

Solution :

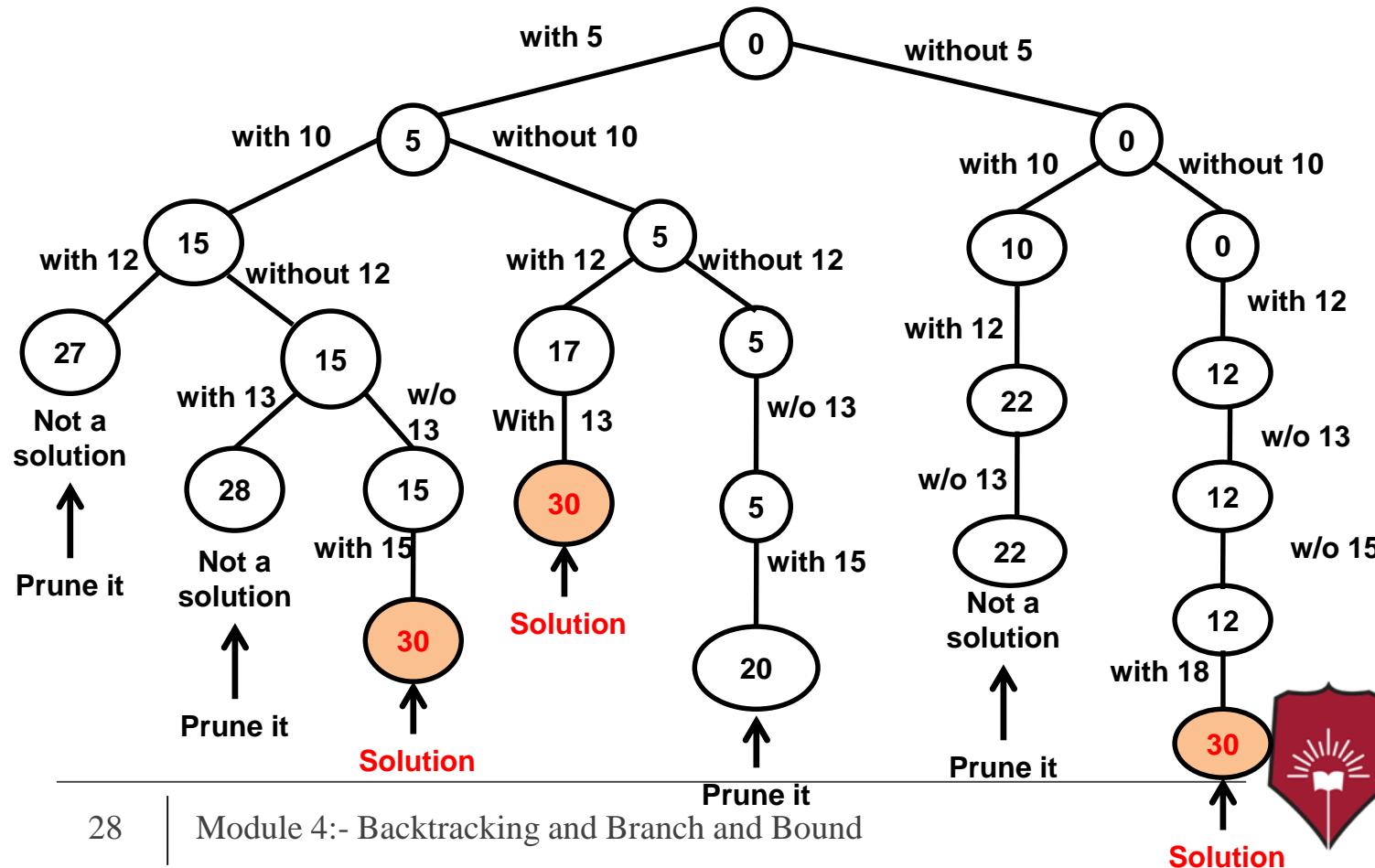
Initially Subset = {}	Sum=0	Description
5	5	Now add next element.
5, 10	15 i.e. $15 < 30$	Add next element.
5, 10, 12	27 i.e. $27 < 30$	Add next element.
5, 10, 12, 13	40 i.e. $40 > 30$	Sum exceeds $M=30$ . Hence backtrack.
5, 10, 12, 15	42	Sum exceeds $M=30$ . Hence backtrack.
5, 10, 12, 18	45	Sum exceeds $M=30$ . Hence backtrack.
5, 10, 13	28 i.e. $28 < 30$	Add next element.
5, 10, 13, 15	43 i.e. $43 > 30$	Sum exceeds $M=30$ . Hence backtrack.
5, 10, 13, 18	46 i.e. $46 > 30$	Sum exceeds $M=30$ . Hence backtrack.
5, 10, 15	30	Solution obtained as $M=30$ .

# Example 1: (cont..)

→ Possible sum of subsets are :

$$\{ 5, 10, 15 \} \quad \{ 5, 12, 13 \} \quad \{ 12, 18 \}$$

The state space tree is shown as below in figure.  $\{5, 10, 12, 13, 15, 18\}$ .



## Example 2:

Q. Solve following problem to find subsets from the given set of elements,  $W = \{2, 7, 8, 15\}$  which will give sum,  $M=17$  and draw portion of state space tree.

Solution :

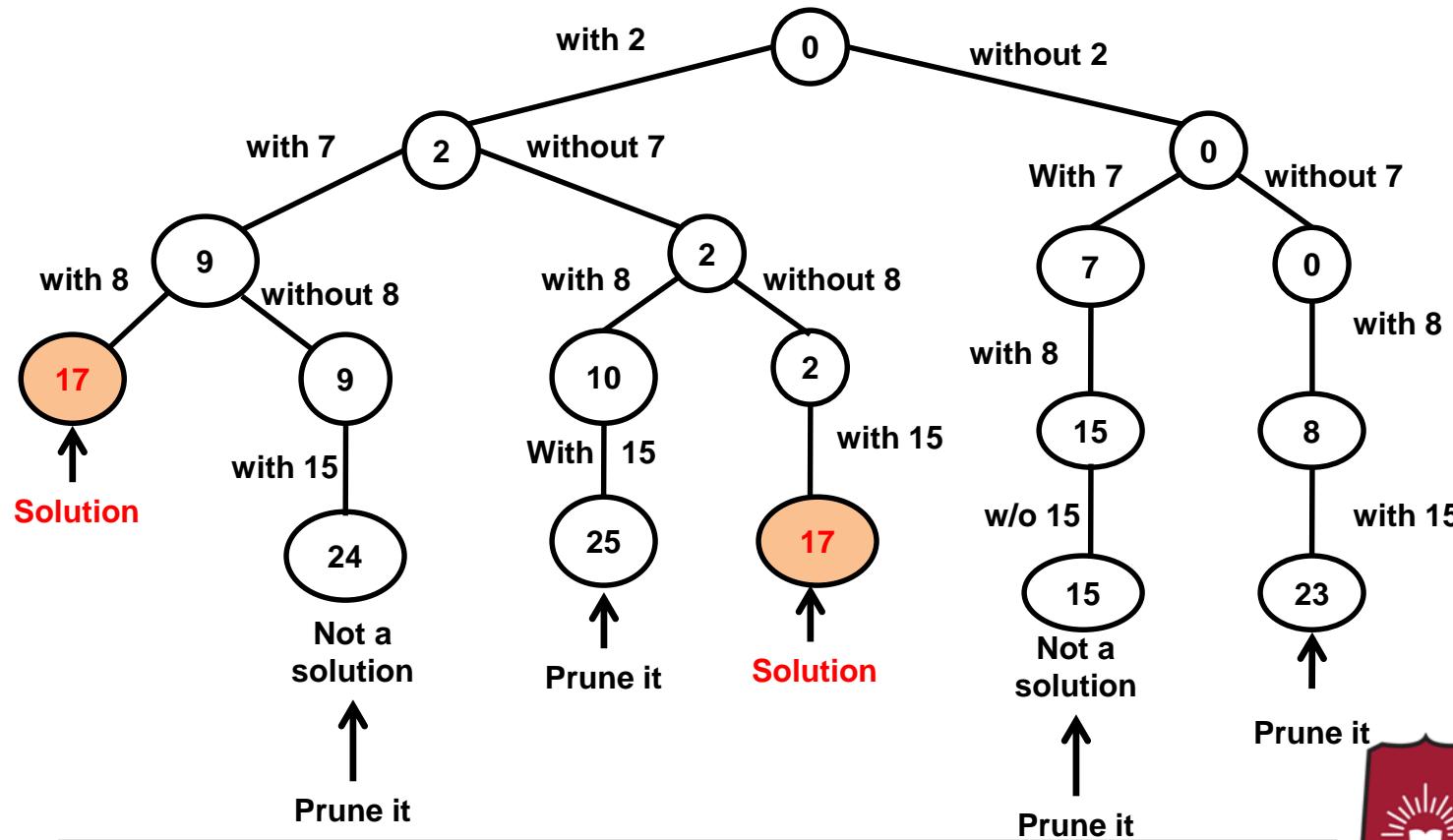
Initially Subset = {}	Sum=0	Description
2	2	Now add next element.
2, 7	9 i.e. $9 < 17$	Add next element.
2, 7, 8	17 i.e. $17 = 17$	Sum of elements=17. Hence solution obtained. Find new subset.
2	2	Now add next element.
2, 8	10 i.e. $10 < 17$	Add next element.
2, 8, 15	25 i.e. $25 > 17$	Sum exceeds M=17. Hence backtrack.
2, 8	10 i.e. $10 < 17$	Already calculated. Backtrack to find next solution.
2	2	Now add next element.
2, 15	17 i.e. $17 = 17$	Sum of elements=17. Hence solution obtained.

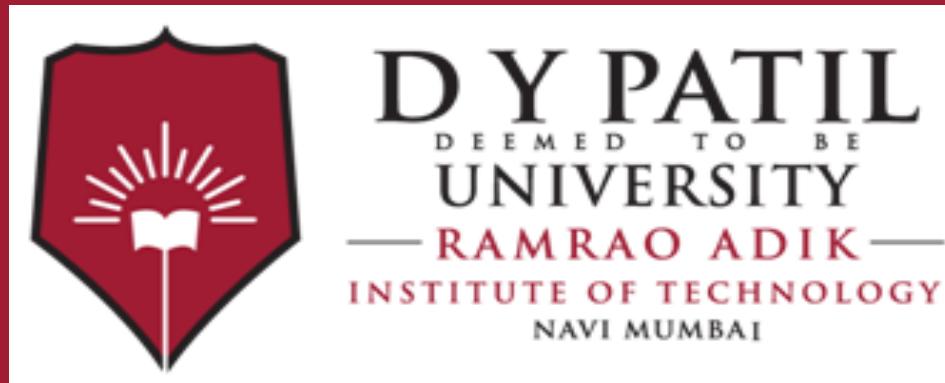
## Example 2: (cont..)

→ Possible sum of subsets are :

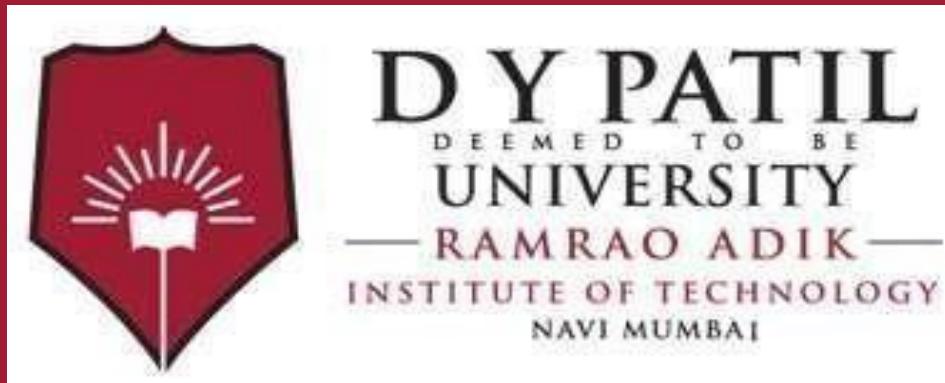
$$\{ 2, 7, 8 \} \quad \{ 2, 15 \}$$

The state space tree is shown as below in figure.  $\{2, 7, 8, 15\}$ .





# Thank You



# Design & Analysis of Algorithms

## Unit 4: Backtracking and Branch and Bound

# **Index -**

---

Lecture 28– Graph coloring

---

Lecture 29– Introduction to Branch and Bound Concept and 15 Puzzle Problem

---

Lecture 30 – Travelling Salesman Problem

---



**Module No 4: Backtracking and Branch and Bound**

---

# **Lecture No: 28**

## **Graph Coloring**



# Graph Coloring Problem

---

- Graph coloring problem involves assigning colors to certain elements of a graph subject to certain restrictions and constraints.
- This has found applications in numerous fields in computer science. For example:
- **Geographical maps:** There can be cases when no two adjacent cities/states can be assigned same color in the maps of countries or states. In this case, only four colors would be sufficient to color any map.
- **Vertex coloring** is the most commonly encountered graph coloring problem.
- The problem states that given  $m$  colors, determine a way of coloring the vertices of a graph such that no two adjacent vertices are assigned same color.
- The smallest number of colors needed to color a graph  $G$  is called its **chromatic number**.

# Solution to Graph Coloring Problem

---

## → Naive Approach :

- In this approach using the brute force method, we find all permutations of color combinations that can color the graph.
- After generating a configuration, check if the adjacent vertices have the same colour or not.
- If the conditions are met, add the combination to the result and break the loop otherwise not.
- This method is not efficient in terms of time complexity because it finds all colors combinations rather than a single solution.
- Since each node can be colored by using any of the  $m$  colors, the total number of possible color configurations are  $m^v$ .
- The complexity is exponential which is very huge.

# Solution to Graph Coloring Problem (cont..)

---

## → Backtracking Approach :

- The backtracking algorithm makes the process efficient by avoiding many bad decisions made in naïve approaches.
- In this approach, we color a single vertex and then move to its adjacent (connected) vertex to color it with different color.
- After coloring, we again move to another adjacent vertex that is uncolored and repeat the process until all vertices of the given graph are colored.
- In case, we find a vertex that has all adjacent vertices colored and no color is left to make it color different, we backtrack and change the color of the last colored vertices and again proceed further.
- If by backtracking, we come back to the same vertex from where we started and all colors were tried on it, then it means the given number of colors (i.e. ‘m’) is insufficient to color the given graph and we require more colors (i.e. a bigger chromatic number).

# Time Complexity of Graph Coloring Problem

---

- Since backtracking is also a kind of brute force approach, there would be total  $O(m^v)$  possible color combinations.
- The upperbound time complexity remains the same but the average time taken will be less due to the refined approach.

# Steps To color graph using the Backtracking Algorithm

---

## 1. Different colors:

- A) Confirm whether it is valid to color the current vertex with the current color (by checking whether any of its adjacent vertices are colored with the same color).
- B) If yes then color it and otherwise try a different color.
- C) Check if all vertices are colored or not.
- D) If not then move to the next adjacent uncolored vertex.

## 2. If no other color is available then backtrack (i.e. un-color last colored vertex).

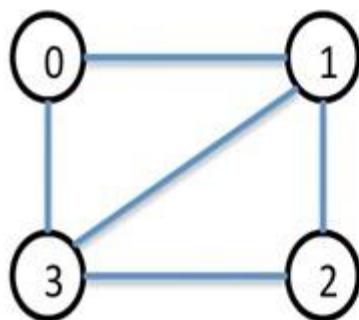
- Here **backtracking means to stop further recursive calls** on adjacent vertices by returning false. In this algorithm Step-1.B (Continue) and Step-2 (backtracking) is causing the program to try different color option.
- **Continue** – try a different color for current vertex.  
**Backtrack** – try a different color for last colored vertex.

# Algorithm for Graph Coloring Problem

```
graphColor(int k)
{
    for ( int c=1; c<=m; c++)
    {
        if(isSafe(k, c))
        {
            x[k]=c;
            if((k+1)<n)
                graphColor(k+1);
            else
                print x[];
                return;
        }
    }
}

isSafe(int k, int c)
{
    for ( int i=0; i<n; i++)
    {
        if (G[k][i]==1 && c==x[i] )
        {
            return false;
        }
    }
    return true;
}
```

## Example Problem: n=4, m=3



Adjacency Matrix (G)

n	0	1	2	3
0	1	1	0	1
1	1	1	1	1
2	0	1	1	1
3	1	1	1	1

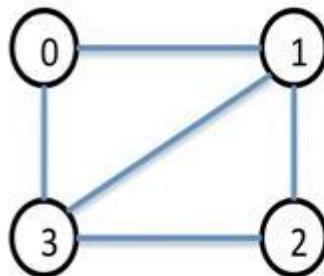
```
graphColour(int k){  
    for(int c = 1; c<=m; c++){  
        if(isSafe(k,c)){  
            x[k] = c;  
            if((k+1)<n)  
                graphColour(k+1);  
            else  
                print x[]; return;  
        }  
    }  
}
```

k = the node that we're going to colour in this level of the recursion

x[k] = Is an array that holds the current colour at each node.

# Example : (cont..)

Example Problem:  $n=4, m=3$



Adjacency Matrix (G)

$n$	0	1	2	3
0	1	1	0	1
1	1	1	1	1
2	0	1	1	1
3	1	1	1	1

```
graphColour(int k){  
    for(int c = 1; c<=m; c++){  
        if(isSafe(k,c)){  
            x[k] = c;  
            if(k+1 <n))  
                graphColour(k+1);  
            else  
                print x[ ]; return;  
        }  
    }  
}
```

```
isSafe(int k, int c){  
    for(int i = 0; i<n; i++){  
        if(G[k][i] == 1 && c == x[i])  
            return false;  
    }  
    return true;  
}
```

$i = 0$

$G[0][0] == 1$

$c \neq x[i], 1 \neq 0$

graphColour(0);

$k = 0$

$i = 1$

$c = 1$  (red)

$G[0][1] == 1$

$x[k] = 1$  (red)

$c \neq x[i], 1 \neq 0$

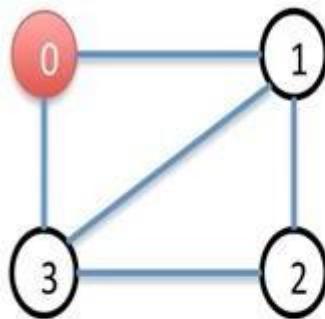
Loop continues for all n

Means vertex is  
not yet coloured

Returns true

# Example : (cont..)

Example Problem:  $n=4, m=3$



Adjacency Matrix (G)

$n$	0	1	2	3
0	1	1	0	1
1	1	1	1	1
2	0	1	1	1
3	1	1	1	1

```
graphColour(int k){
    for(int c = 1; c<=m; c++){
        if(isSafe(k,c)){
            x[k] = c;
            if(k+1 <n))
                graphColour(k+1);
            else
                print x[];
                return;
        }
    }
}
```

```
isSafe(int k, int c){
    for(int i = 0; i<n; i++){
        if(G[k][i] == 1 && c == x[i]){
            return false;
        }
    }
    return true;
}
```

$i = 0$

$G[1][0] == 1$

$c == x[i], 1 == 1$

Vertex is coloured

graphColour(1);

$K = 1$

Returns false

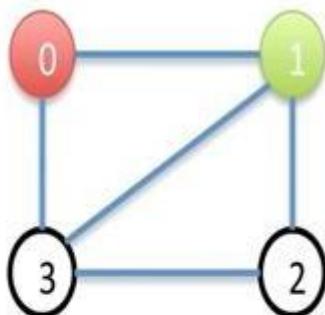
$c = 1$  (red)

isSafe() called  
with  $c=2$

$c = 2$  (green)

$x[1] = 2$  (green)

Example Problem:  $n=4, m=3$



Adjacency  
Matrix (G)

n	0	1	2	3
0	1	1	0	1
1	1	1	1	1
2	0	1	1	1
3	1	1	1	1

```
graphColour(int k){  
    for(int c = 1; c<=m; c++){  
        if(isSafe(k,c)){  
            x[k] = c;  
            if(k+1 <n)  
                graphColour(k+1);  
            else  
                print x[]; return;  
        }  
    }  
}
```

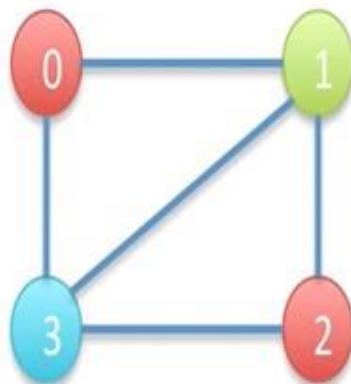
```
isSafe(int k, int c){  
    for(int i = 0; i<n; i++){  
        if(G[k][i] == 1 && c == x[i])  
            return false;  
    }  
    return true;  
}
```

The recursion continues for all the nodes in the graph, trying the different colours.

If no colour is safe, and not all nodes are filled, it'll back track and try a different colour on the last node set.

## Example : (cont..)

Example Problem:  $n=4, m=3$



Red = 1

Green = 2

Blue = 3

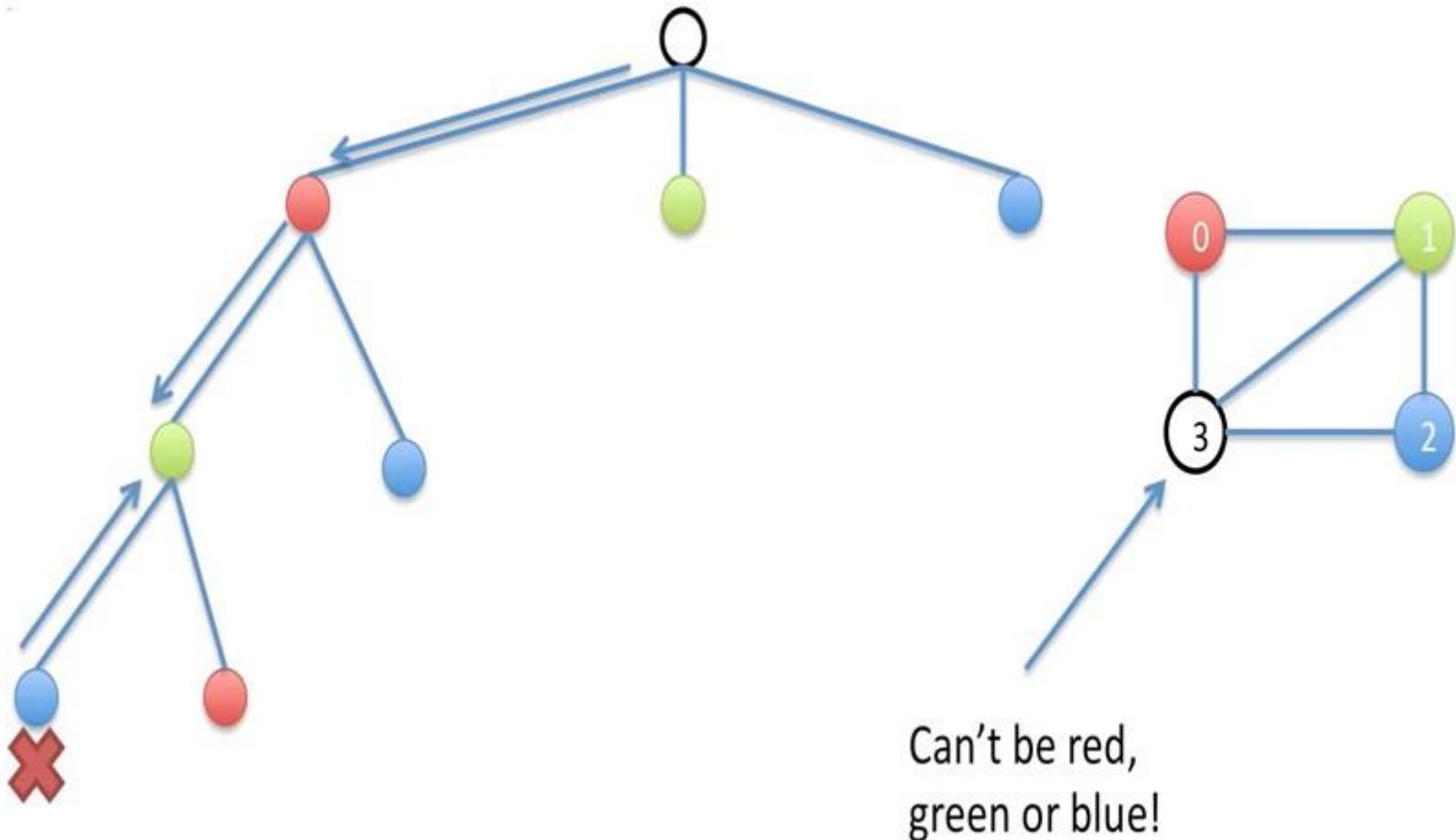
$x[k]$

0	1	2	3
1	2	1	3

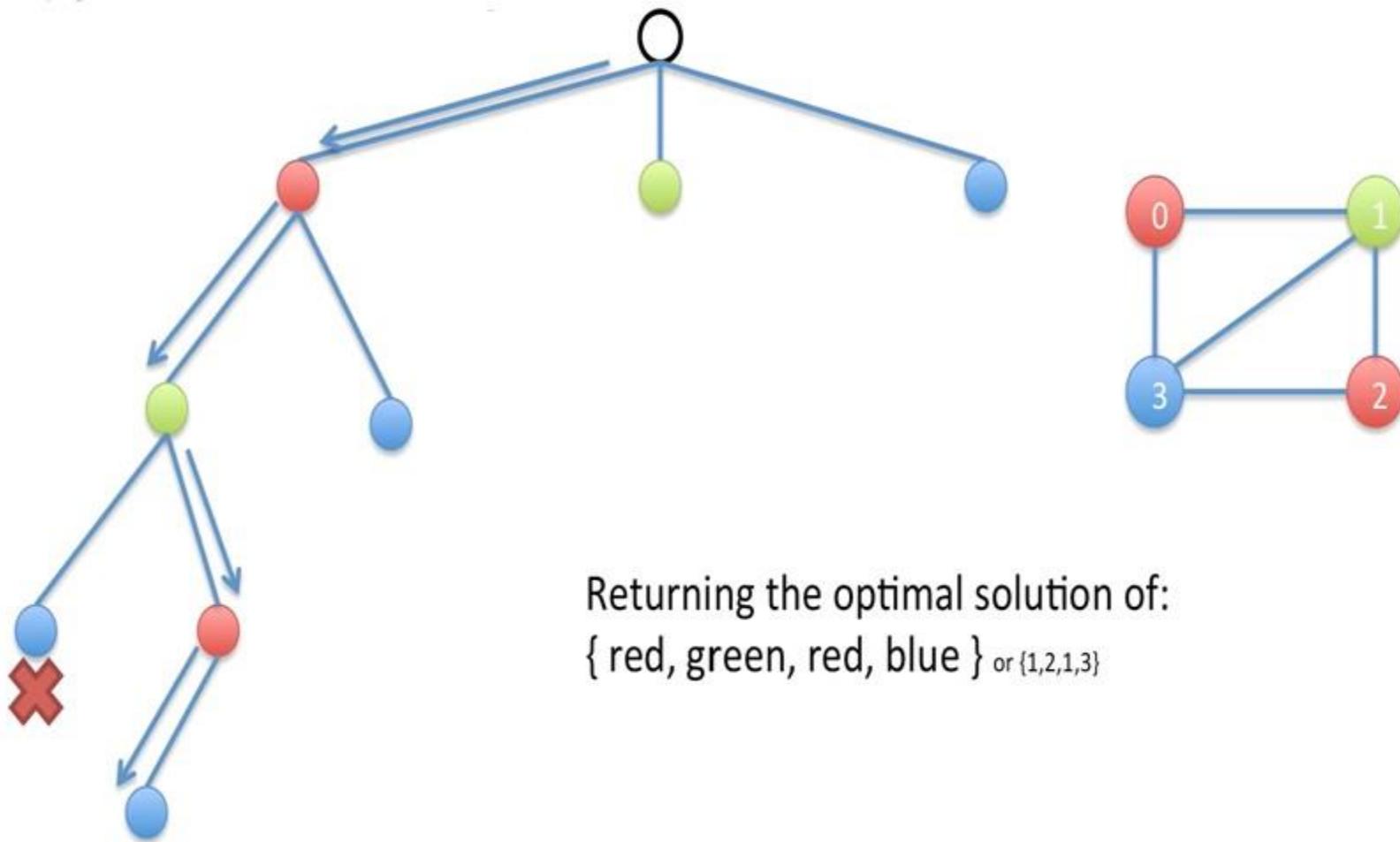
$$x[0] = 1, x[1] = 2, x[2] = 1, x[3] = 3$$

## Example : (cont..)

---



## Example : (cont..)



# Introduction to Branch and Bound and 15 Puzzle Problem



# Introduction to Branch & Bound Approach

---

- In computer science, there is a large number of optimization problems which has a finite but extensive number of feasible solutions.
- Among these, some problems like finding the shortest path in a graph or Minimum Spanning Tree can be solved in polynomial time.
- A significant number of optimization problems like production planning, crew scheduling, 15 Puzzle can't be solved in polynomial time, and they belong to the NP-Hard class.
- These problems are the example of NP-Hard combinatorial optimization problem.
- Branch and bound (B&B) is an algorithm paradigm widely used for solving such problems.
- Branch and bound algorithms are used to find the optimal solution for combinatory, discrete, and general mathematical optimization problems.



# Introduction to Branch & Bound Approach (cont..)

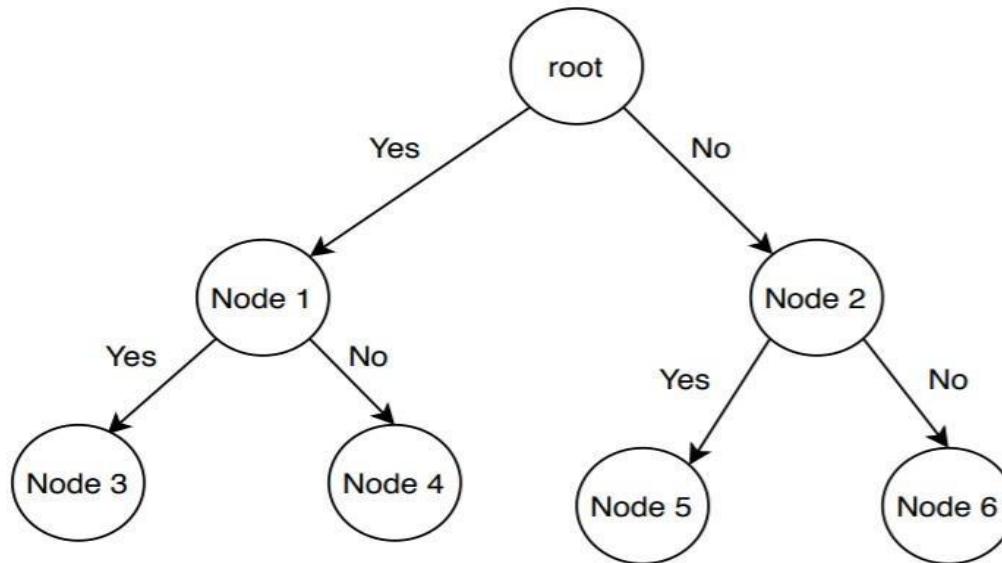
---

- Branch and bound is a systematic method for solving optimization problems
- B&B is a rather general optimization technique that applies where the greedy method and dynamic programming fail.
- However, it is much slower. Indeed, it often leads to exponential time complexities in the worst case.
- On the other hand, if applied carefully, it can lead to algorithms that run reasonably fast on average.
- The general idea of B&B is a BFS-like search for the optimal solution, but not all nodes get expanded (i.e., their children generated). Rather, a carefully selected criterion determines which node to expand and when, and another criterion tells the algorithm when an optimal solution has been found.

# Introduction to Branch & Bound Approach (cont..)

- In general, given an NP-Hard problem, a branch and bound algorithm explores the entire search space of possible solutions and provides an optimal solution.
- A branch and bound algorithm consist of stepwise enumeration of possible candidate solutions by exploring the entire search space.

- Wid
- Th



d a rooted  
space:



# Introduction to Branch & Bound Approach (cont..)

---

- Here, each child node is a partial solution and part of the solution set.
- Before constructing the rooted decision tree, we set an upper and lower bound for a given problem based on the optimal solution.
- At each level, we need to make a decision about which node to include in the solution set.
- At each level, we explore the node with the best bound. In this way, we can find the best and optimal solution fast.
- Now it is crucial to find a good upper and lower bound in such cases.
- We can find an upper bound by using any local optimization method or by picking any point in the search space.
- In general, we want to partition the solution set into smaller subsets of solution.
- Then we construct a rooted decision tree, and finally, we choose the best possible subset (node) at each level to find the best possible solution set.



# Introduction to Branch & Bound Approach (cont..)

---

- Here, each child node is a partial solution and part of the solution set.
- Before constructing the rooted decision tree, we set an upper and lower bound for a given problem based on the optimal solution.
- At each level, we need to make a decision about which node to include in the solution set.
- At each level, we explore the node with the best bound. In this way, we can find the best and optimal solution fast.
- Now it is crucial to find a good upper and lower bound in such cases.
- We can find an upper bound by using any local optimization method or by picking any point in the search space.
- In general, we want to partition the solution set into smaller subsets of solution.
- Then we construct a rooted decision tree, and finally, we choose the best possible subset (node) at each level to find the best possible solution set.



# Advantages of Branch & Bound Approach

---

- In a branch and bound algorithm, we don't explore all the nodes in the tree. That's why **the time complexity of the branch and bound algorithm is less when compared with other algorithms.**
- If the problem is not large and if we can do the branching in a reasonable amount of time, **it finds an optimal solution for a given problem.**
- The branch and bound algorithm find a minimal path to reach the optimal solution for a given problem. **It doesn't repeat nodes while exploring the tree.**



# Disadvantages of Branch & Bound Approach

---

- **The branch and bound algorithm are time-consuming.** Depending on the size of the given problem, the number of nodes in the tree can be too large in the worst case.
- Also, parallelization is extremely difficult in the branch and bound algorithm.



# Difference between Backtracking and Branch & Bound Approach

Backtracking	Branch and Bound
Solution for backtracking is traced using DFS.	It is not necessary to use DFS for obtaining the solution even BFS can be applied.
Decision problem can be solved using Backtracking.	Optimization problem can be solved using Branch and Bound.
While finding the solution to the problem bad choices can be made.	It proceeds on better solutions. So there can not be a bad solution.
<b>Applications:</b> Graph Coloring N-Quen Problem	<b>Applications:</b> 15 Puzzle Travelling Salesman Problem



# 15 Puzzle Problem



# 15 Puzzle Problem

- This problem was invented by Sam Loyd in 1878.
- There are 15 numbered tiles placed on a square frame with capacity of 16 tiles.
- **Objective of this problem is :** Given an initial arrangement and objective to transform any initial arrangement to a goal arrangement through a series of legal moves.

1	2	3	4
5	6	7	
9	10	12	8
11	13	14	15

Initial  
Arrangement



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Goal Arrangement



# 15 Puzzle Problem (cont..)

---

- There is always one empty slot in the initial arrangement i.e. ES.
- A legal move is the one that move a tile adjacent to an Empty Space(ES) are moved to either left, right, up or down.
- Each move creates a new arrangement of tile called states of the puzzle.
- Initial and final arrangements are called initial state and goal states respectively.
- A state space tree can be drawn for representing various states.
- In the state space tree the path from initial state to goal state represents the answer.



# Solution to 15 Puzzle Problem

- The lower bound cost of node  $x$  is calculated using formula,  
$$\hat{c}(x) = f(x) + \hat{g}(x)$$
where,  
 $f(x)$  : length of the path from root to node  $x$ .  
 $\hat{g}(x)$  : number of non blank tiles which are not in their goal position for node  $x$
- **Step 1:**
  - Start from initial arrangement ie. Initial state which becomes E-node i.e. Root node.
  - Generate child nodes 2, 3, 4 and 5.
  - Once child node gets generated they become live nodes and node 1 dies.



# Solution to 15 Puzzle Problem (cont..)

- **Step 2:**
  - Starting with Initial arrangement will generate node 2, 3, 4 and 5 by placing ES(Empty slot) tile at different positions.
    - Consider tile positions for node 2 as shown where ES is moved in **Up** direction.
    - Calculating cost for node 2 using formula ,  $\hat{c}(x) = f(x) + \hat{g}(x)$

Initial  
Arrangement

1	2	3	4
5	6	ES	8
9	10	7	11
13	14	15	12



1	2	ES	4
5	6		8
9	10		
13	14	15	



# Solution to 15 Puzzle Problem (cont..)

## • Step 2: (cont..)

-Calculating cost for node 3 using formula ,  $\hat{c}(x) = f(x) + \hat{g}(x)$

- Consider tile position for node 3 as  $h$  where ES is

moved in **Right** direction.

Initial  
Arrangement

6	9	10	11
13	14	15	12
1	2	3	4



3 <sub>1</sub>	h	n	w
5	6	7	ES
9	10	8	ES
13	14	15	?

- Path length from root to node 3 is 1.  $\therefore f(3) = 1$

- The shaded tiles indicate that they are not in their goal positions. There are four such tiles.  $\therefore \hat{g}(3) = 4$

$$\begin{aligned}\hat{c}(3) &= f(3) + \hat{g}(3) \\ &= 1+4\end{aligned}$$

$$\therefore \hat{c}(3) = 5$$



# Solution to 15 Puzzle Problem (cont..)

## • Step 2: (cont..)

-Calculating cost for node 4 using formula ,  $\hat{c}(x) = f(x) + \hat{g}(x)$

- Consider tile positions for node 4 as

where ES is moved in Down direction.

Initial Arrangement

5	6	7	8
9	10	11	
13	14	15	12



4	5	6	7
9	10	11	
13	14	15	8

- Path length from root to node 4 is 1.  $\therefore f(4) = 1$

- The shaded tiles indicate that they are not in their goal positions. There are two such tiles.  $\therefore \hat{g}(4) = 2$

$$\begin{aligned}\hat{c}(4) &= f(4) + \hat{g}(4) \\ &= 1+2\end{aligned}$$

$$\therefore \hat{c}(4) = 3$$



# Solution to 15 Puzzle Problem (cont..)

## • Step 2: (cont..)

-Calculating cost for node 5 using formula ,  $\hat{c}(x) = f(x) + \hat{g}(x)$

- Consider tile positions for node 5 as shown where ES is moved in **Left** direction.

Initial  
Arrangement

5	9	10	11
13	14	15	12
16	7	ES	8



5	9	10	11
13	14	15	12
16	7	ES	8

5	9	10	11
13	14	15	12
16	7	ES	8

- Path length from root to node 5 is 1.  $\therefore f(5) = 1$

- The shaded tiles indicate that they are not in their goal positions. There are four such tiles.  $\therefore \hat{g}(5) = 4$

$$\begin{aligned}\hat{c}(5) &= f(5) + \hat{g}(5) \\ &= 1+4\end{aligned}$$

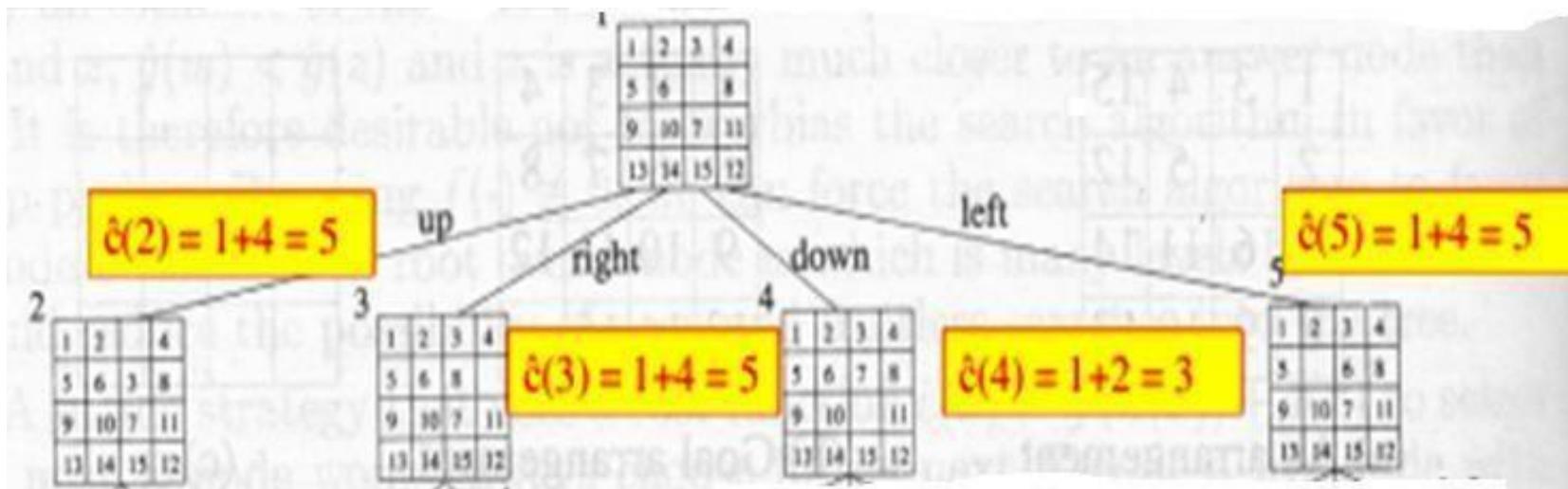
$$\therefore \hat{c}(5) = 5$$



# Solution to 15 Puzzle Problem (cont..)

- **Step 2: (cont..)**

- From node 2, 3, 4 and 5 node 4 is having minimum cost i.e.  $\hat{c}(4) = 3$ . Hence node 4 becomes E node.
- Now, ES can be moved in only 3 directions i.e. Right, left and down as first two row tiles are properly placed.
- Therefore, expand node 4 to generate new nodes 10, 11 and 12.



# Solution to 15 Puzzle Problem (cont..)

- **Step 3:**
  - Calculating cost for node 10 using formula ,  $\hat{c}(x) = f(x) + \hat{g}(x)$
  - Consider is moved in node 10

er til	e <sub>2</sub> po	s <sub>3</sub> iti	node	as	h	n <sub>4</sub> where
n R	ht	direc	n <sub>8</sub>	s <sub>2</sub>	..	ES
ig			o	5	6	7
9	10	ES	11	9	10	11
13	14	15	12	13	14	15
th from	root to no	t to no	de	8 is 2.	$f(10) = 2$	

- Path length from root to node 10 is 2.  $\therefore f(10) = 2$
- The shaded tiles indicate that they are not in their goal positions. There is one such tile.  $\therefore \hat{g}(10) = 1$
- $\hat{c}(10) = f(10) + \hat{g}(10)$   
 $= 2+1$

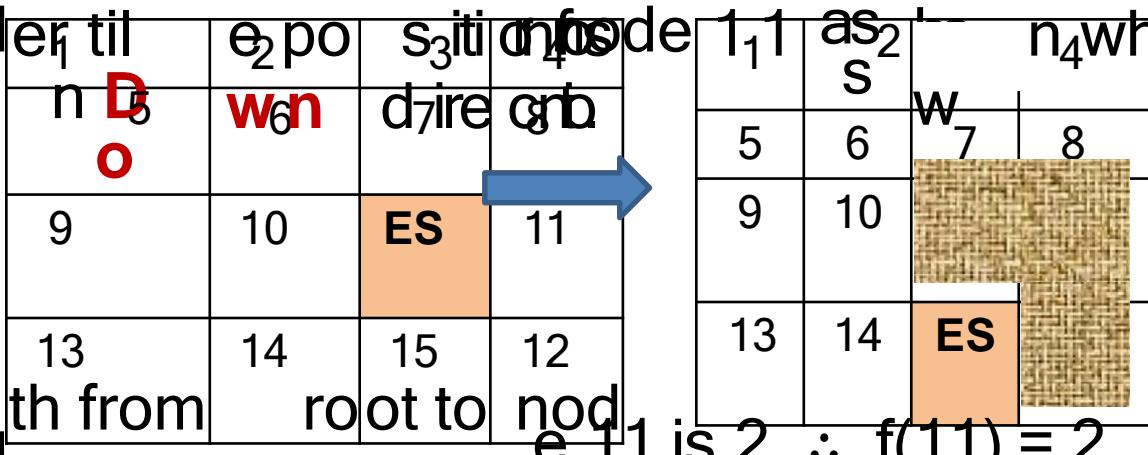
$$\therefore \hat{c}(10) = 3$$



# Solution to 15 Puzzle Problem (cont..)

- **Step 3: (cont..)**
    - Calculating cost for node 11 using formula ,  $\hat{c}(x) = f(x) + \hat{g}(x)$ 
      - Consider tile 11 is moved in position 11
- Path length from root to node 11 is 2.  $\therefore f(11) = 2$
- The shaded tiles indicate that they are not in their goal positions. There are 3 such tiles.  $\therefore \hat{g}(11) = 3$
- $\hat{c}(11) = f(11) + \hat{g}(11)$   
 $= 2+3$

$$\therefore \hat{c}(11) = 5$$

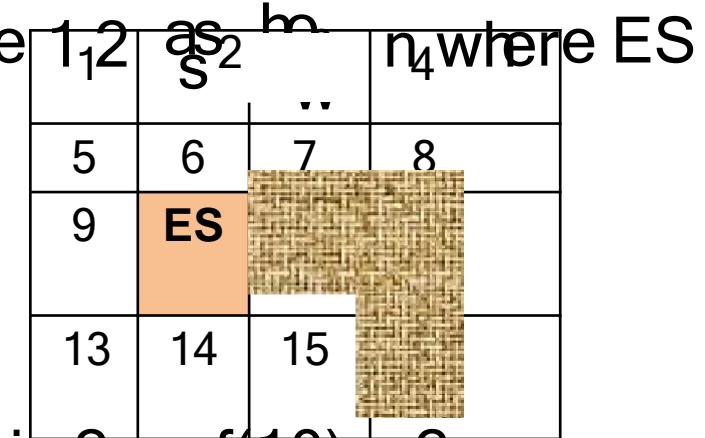


# Solution to 15 Puzzle Problem (cont..)

- **Step 3: (cont..)**
  - Calculating cost for node 12 using formula ,  $\hat{c}(x) = f(x) + \hat{g}(x)$

- Consider tile  $n_5$  is moved in the direction of node 12 as shown below.

$n_5$	$t_6$	$e_2$	$p_0$	$s_3$	$i_4$	$t_7$	$f_8$
$n_5$	$t_6$	$e_2$	$p_0$	$s_3$	$i_4$	$t_7$	$f_8$
9	10	ES		11			
13	14	15	12				



- Path length from root to node 12 is 2.  $\therefore f(10) = 2$

- The shaded tiles indicate that they are not in their goal positions. There are 3 such tiles.  $\therefore \hat{g}(12) = 3$

$$\begin{aligned}\hat{c}(12) &= f(12) + \hat{g}(12) \\ &= 2+3\end{aligned}$$

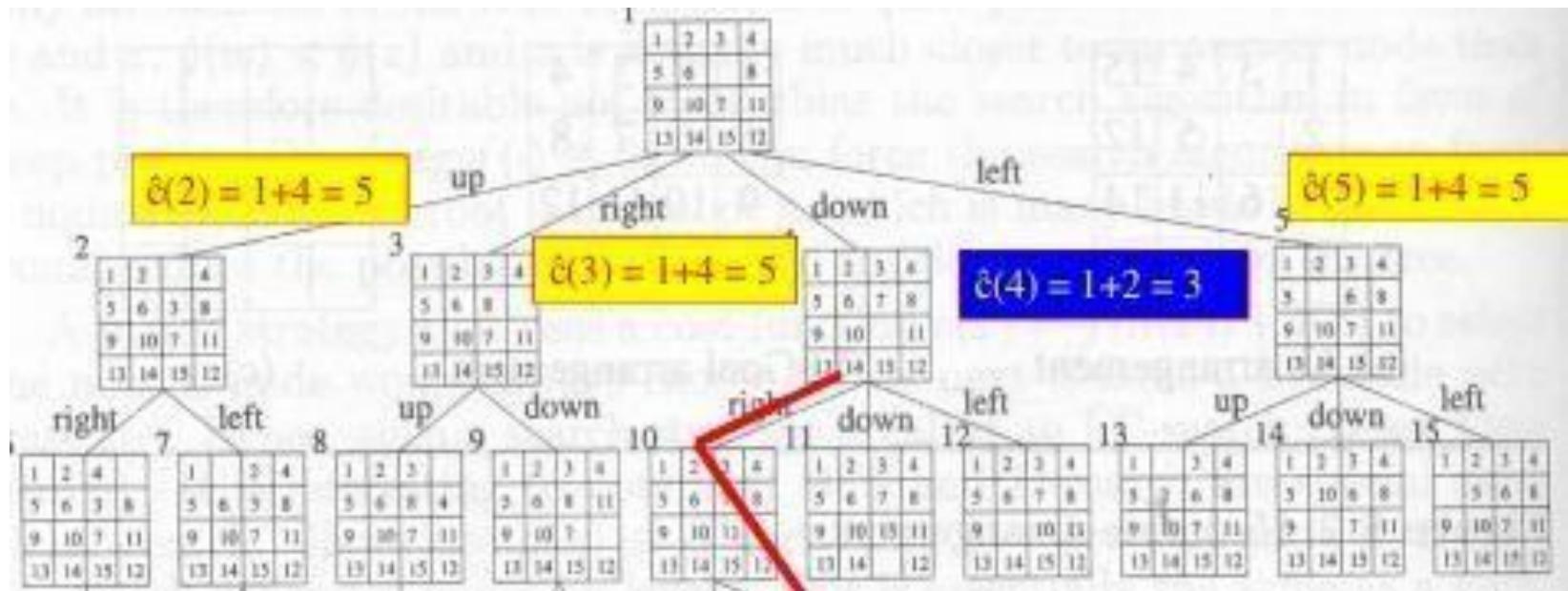
$$\therefore \hat{c}(12)=5$$



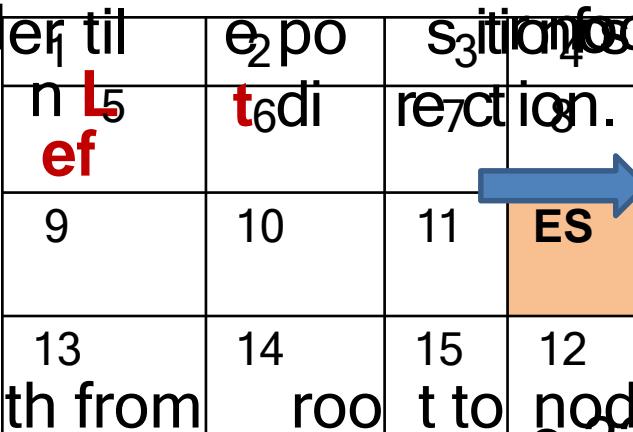
# Solution to 15 Puzzle Problem (cont..)

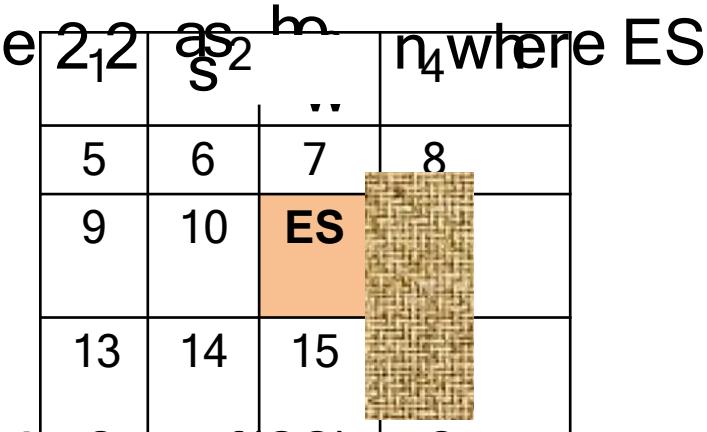
## • Step 3: (cont..)

- From node 10, 11 and 12 node 10 is having minimum cost i.e.  $\hat{c}(10) = 3$ . Hence node 10 becomes E node.
- Now, ES can be moved in only 2 directions i.e. Down and left as most of the tiles are properly placed.
- Therefore, expand node 10 to generate new nodes 22 and 23.



# Solution to 15 Puzzle Problem (cont..)

- **Step 4:**
  - Calculating cost for node 22 using formula ,  $\hat{c}(x) = f(x) + \hat{g}(x)$ 
    - Consider tile 5 is moved in position 11. 



- Path length from root to node 22 is 3.  $\therefore f(22) = 3$

- The shaded tiles indicate that they are not in their goal positions. There are 2 such tiles.  $\therefore \hat{g}(22) = 2$

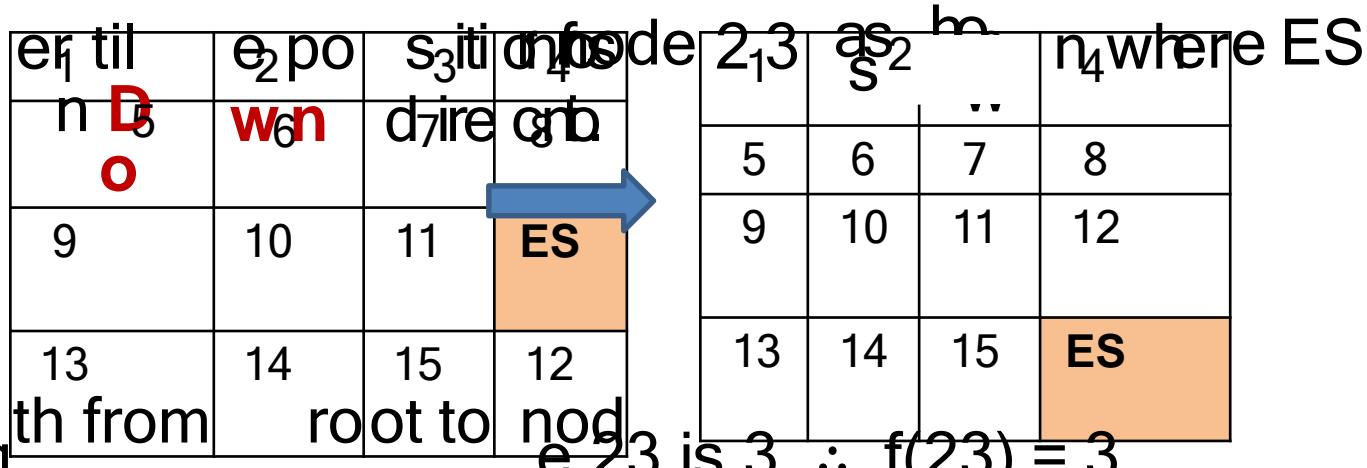
$$\begin{aligned}\hat{c}(22) &= f(22) + \hat{g}(22) \\ &= 3+2\end{aligned}$$

$$\therefore \hat{c}(22) = 5$$

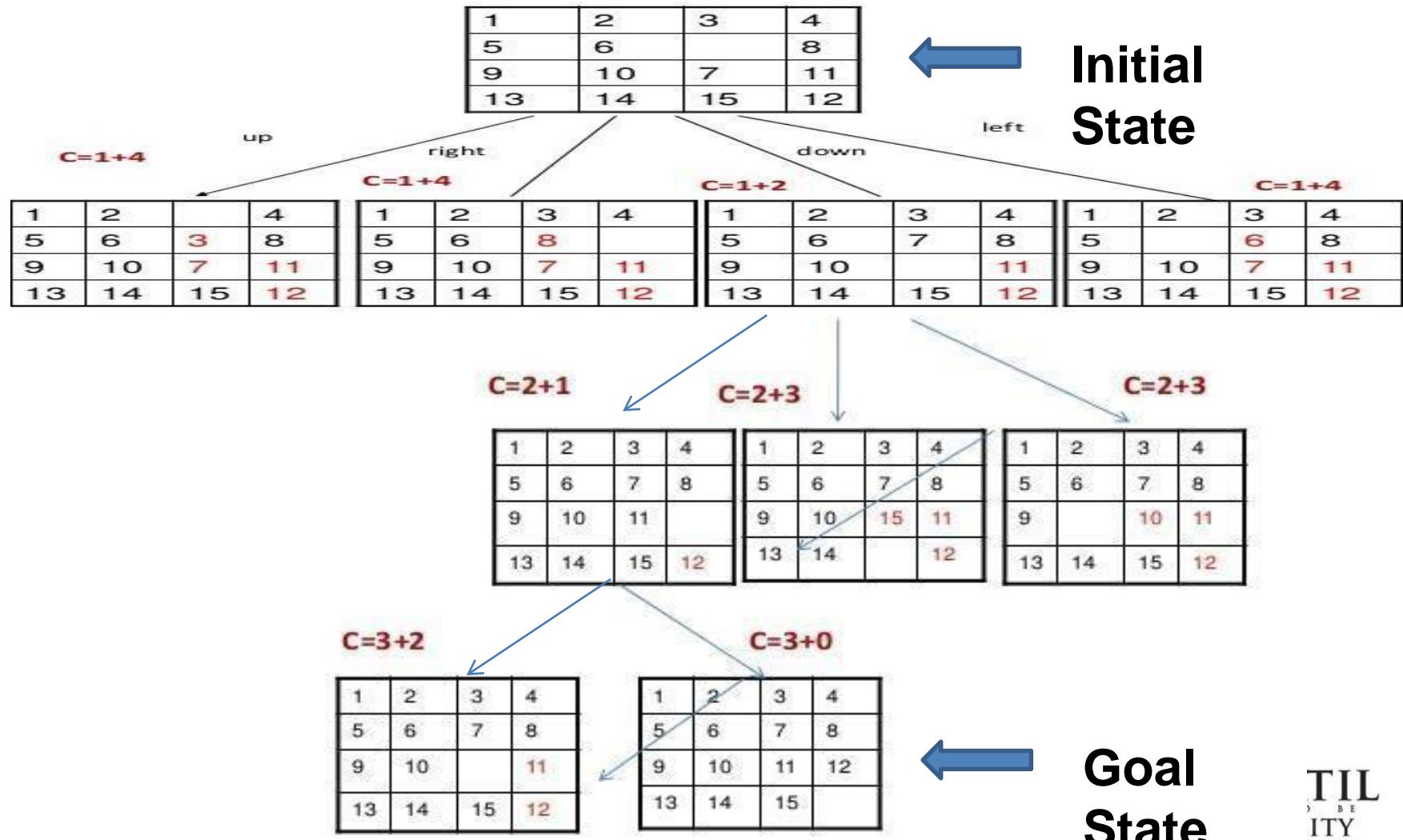


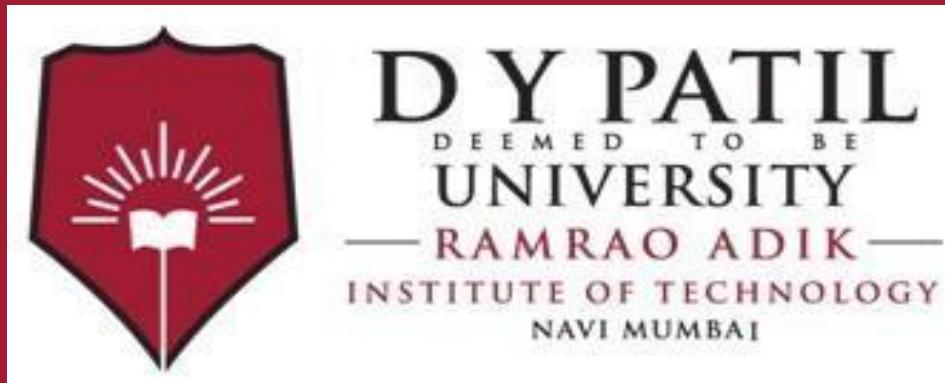
# Solution to 15 Puzzle Problem (cont..)

- **Step 4: (cont..)**
  - Calculating cost for node 23 using formula ,  $\hat{c}(x) = f(x) + g(x)$
  - Consider tile 5 is moved in node 23 as shown in figure
  - Path length from root to node 23 is 3.  $\therefore f(23) = 3$
  - All tiles are in their goal positions.  $\therefore g(23) = 0$
  - $\hat{c}(23) = f(23) + g(23)$  $= 3+0$  $\therefore \hat{c}(23) = 3$

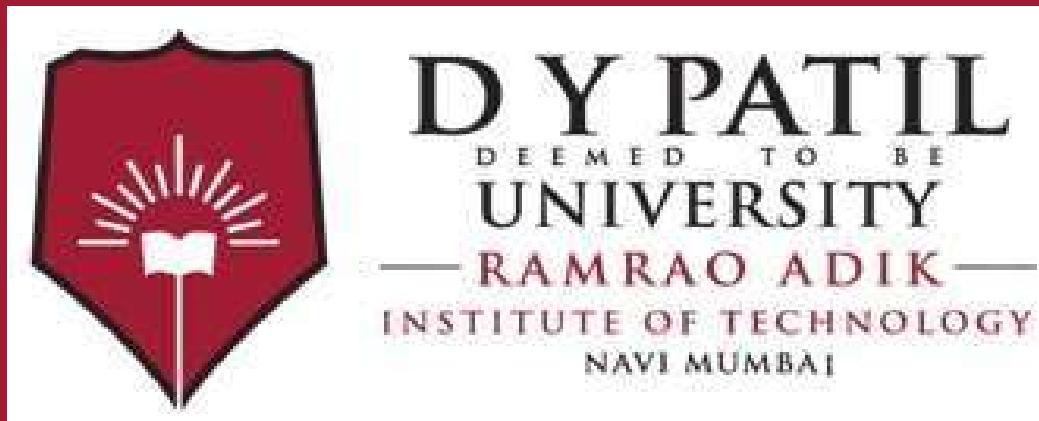


# State Space Tree of 15 Puzzle Problem





# Thank You



# Design and Analysis of Algorithms

## Unit 5: String Matching Algorithms

# Index -

---

Lecture 31 Naïve String Matching Algorithm

Lecture 32 Rabin Karp algorithm

Lecture 33 Knuth-Morris-Pratt Algorithm



# Naïve String Matching Algorithm



# What is *string matching*?

---

- ❖ Finding all occurrences of a *pattern* in a given *text* (or *body of text*)  
or
- ❖ In computer science, string searching algorithms, sometimes called string matching algorithms, that try to find a place where one or several string (also called pattern) are found within a larger string or text.ing or text

# Many applications

---

- ❖ While using editor/word processor/browser
- ❖ Login name & password checking
- ❖ Virus detection
- ❖ Header analysis in data communications
- ❖ DNA sequence analysis

# **TYPES OF STRING MATCHING:-**

---

- ØThe Naive string-matching algorithm
- ØThe Rabin-Krap algorithm
- ØString matching with finite automata
- ØThe Knuth-Morris-Pratt algorithm

# The Naive string-matching algorithm

---

The idea of the naive solution is just to make a comparison character by character of the text for all pattern.

It returns all the valid shifts found.

P=Pattern

T=Text

S=Shift



# The Naive string-matching algorithm

**cont....**

---

**Input:** Text strings  $T[1..n]$  and  $P[1..m]$

**Result:** All valid shifts displayed

**NAÏVE-STRING-MATCHER** ( $T, P$ )

$n \leftarrow \text{length}[T] \ m \leftarrow \text{length}[P]$

**for**  $s \leftarrow 0$  **to**  $n-m$

**if**  $P[1..m] = T[(s+1)..(s+m)]$

print “pattern occurs with shift”  $s$

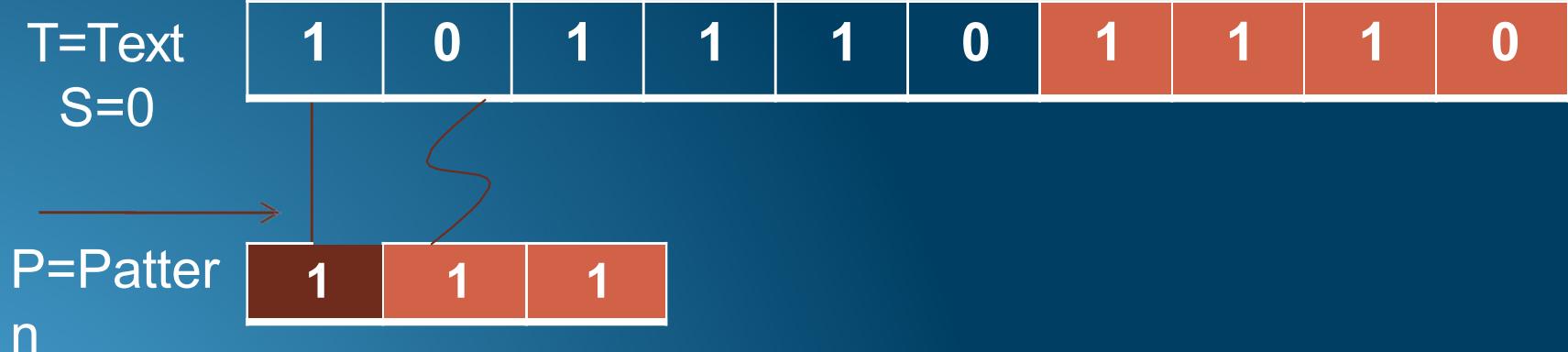
# EXAMPLE

□ SUPPOSE,

$T=1011101110$

$P=111$

FIND ALL VALID SHIFT.....

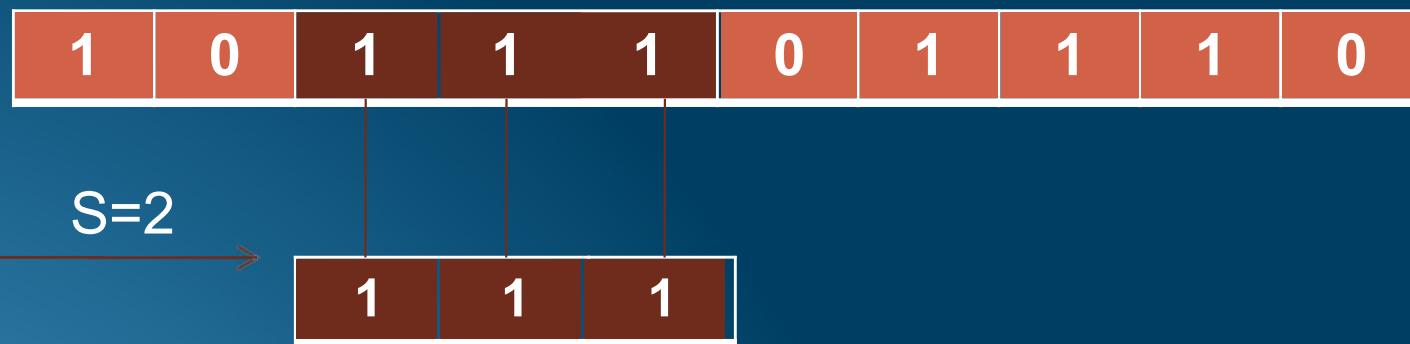


1	0	1	1	1	0	1	1	1	0
---	---	---	---	---	---	---	---	---	---

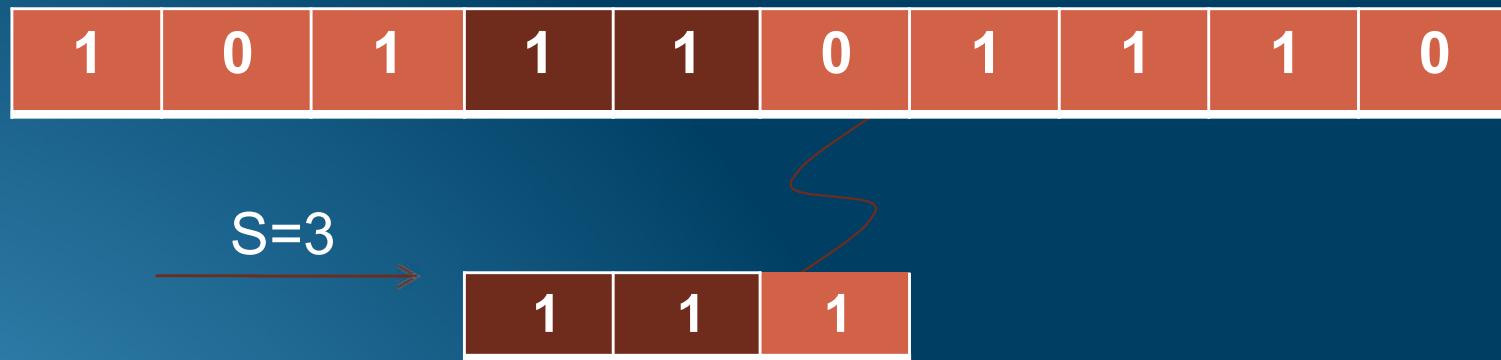
S=1

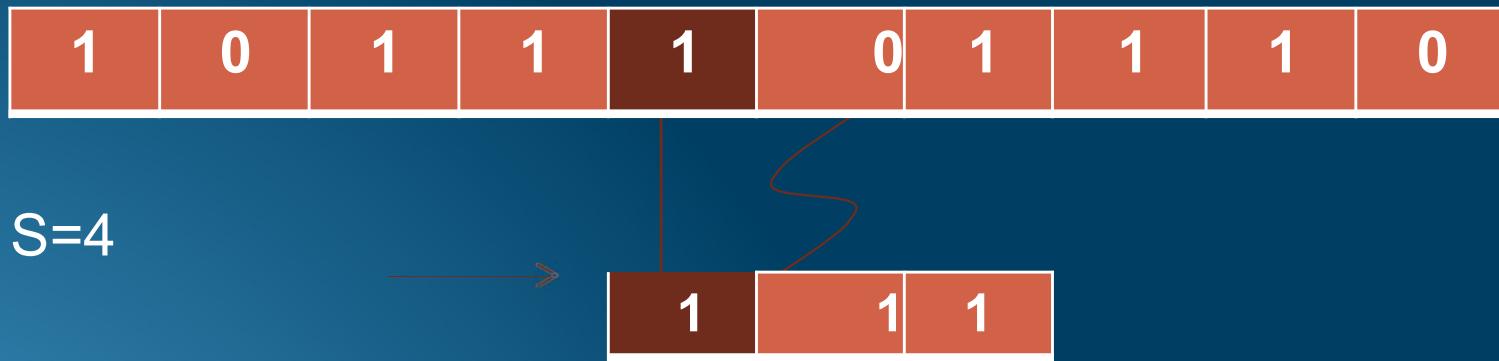


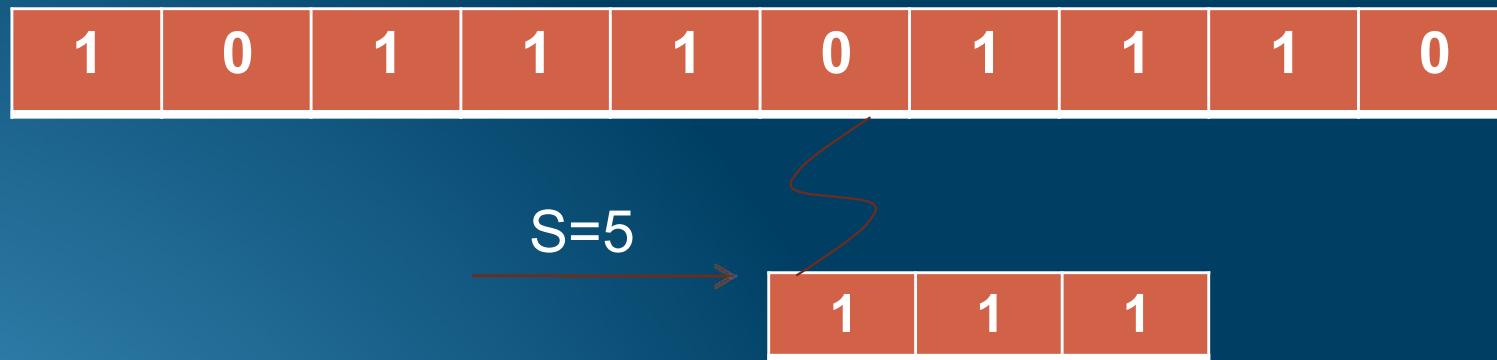
1	1	1
---	---	---



So, S=2 is a valid shift...









So, S=6 is a valid shift...

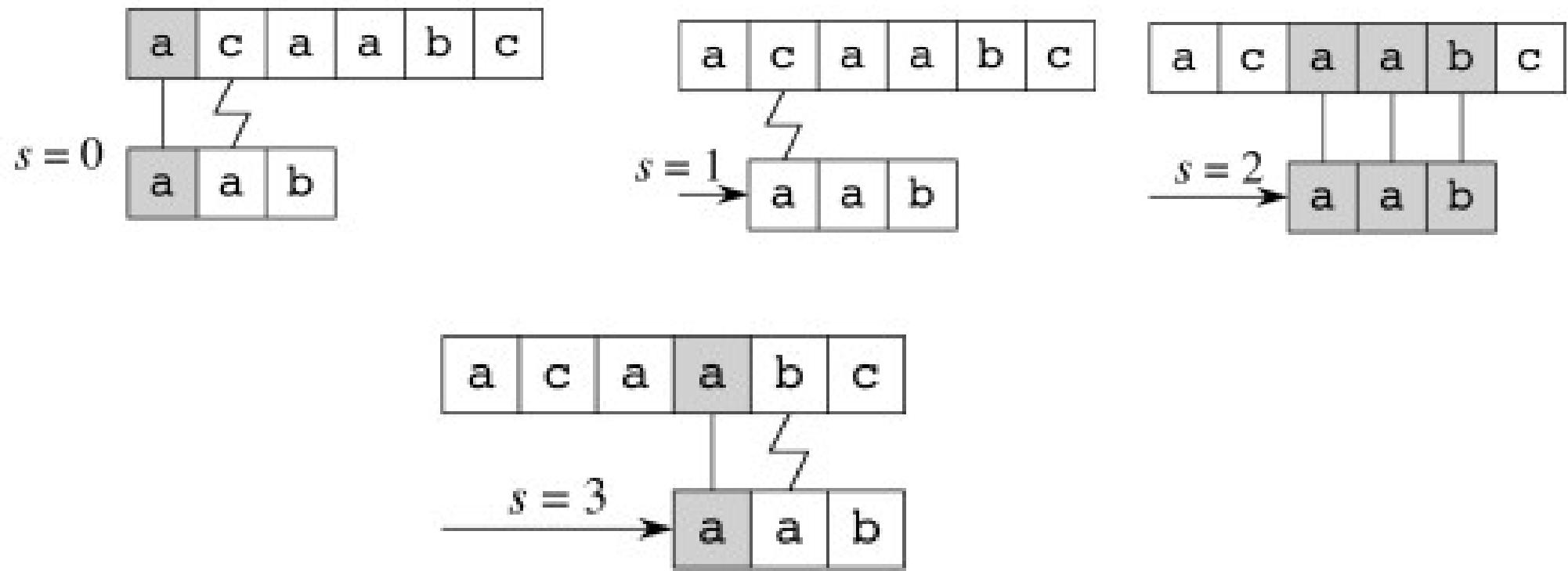
1	0	1	1	1	0	1	1	1	0
---	---	---	---	---	---	---	---	---	---

S=7

1	1	1
---	---	---

# Example

Suppose  $P = \text{aab}$  and  $T = \text{acaabc}$ . There are four passes:



# Worst-case Analysis

---

There are  $m$  comparisons for each shift in the worst case

- ☒ There are  $n-m+1$  shifts
- ☒ So, the worst-case running time is  $\Theta((n-m+1)m)$  , which is  $\Theta(n^2)$  if  $m = \text{floor}(n/2)$
- ☒ In the example on previous slide, we have  $(13-4+1)4$  comparisons in total
- ☒ Naïve method is inefficient because information from a shift is not used again



## Lecture 33

---

# Knuth-Morris-Pratt Algorithm



# The Knuth-Morris-Pratt Algorithm

---

- Knuth, Morris and Pratt proposed a linear time algorithm for the string matching problem.
- A matching time of  $O(n)$  is achieved by avoiding comparisons with elements of ‘S’ that have previously been involved in comparison with some element of the pattern ‘p’ to be matched. i.e., backtracking on the string ‘S’ never occurs

# Components of KMP algorithm

---

- The prefix function,  $\Pi$

The prefix function,  $\Pi$  for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. This information can be used to avoid useless shifts of the pattern ‘p’. In other words, this enables avoiding backtracking on the string ‘S’.

- The KMP Matcher

With string ‘S’, pattern ‘p’ and prefix function ‘ $\Pi$ ’ as inputs, finds the occurrence of ‘p’ in ‘S’ and returns the number of shifts of ‘p’ after which occurrence is found.

# The prefix function, $\Pi$

---

Following pseudo code computes the prefix function,  $\Pi$ :

## Compute-Prefix-Function (p)

```
m ← length[p]           //'p' pattern to be matched
Π[1] ← 0
k ← 0
    for q ← 2 to m
        do while k > 0 and p[k+1] != p[q]
            do k ← Π[k]
            If p[k+1] = p[q]
                then k ← k +1
            Π[q] ← k
return Π
```



## Example: compute $\Pi$ for the pattern 'p' below:

p

a	b	a	b	a	c	a
---	---	---	---	---	---	---

Initially:  $m = \text{length}[p] = 7$

$$\Pi[1] = 0$$

$$k = 0$$

Step 1:  $q = 2, k=0$

$$\Pi[2] = 0$$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
$\Pi$	0	0					

Step 2:  $q = 3, k = 0,$   
 $\Pi[3] = 1$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
$\Pi$	0	0	1				

Step 3:  $q = 4, k = 1$   
 $\Pi[4] = 2$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
$\Pi$	0	0	1	2			

Step 4:  $q = 5, k = 2$

$$\Pi[5] = 3$$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
$\Pi$	0	0	1	2	3		

Step 5:  $q = 6, k = 3$

$$\Pi[6] = 1$$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
$\Pi$	0	0	1	2	3	1	

Step 6:  $q = 7, k = 1$

$$\Pi[7] = 1$$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
$\Pi$	0	0	1	2	3	1	1

After iterating 6 times, the prefix function computation is complete:  $\rightarrow$

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
$\Pi$	0	0	1	2	3	1	1

# The KMP Matcher

The KMP Matcher, with pattern ‘p’, string ‘S’ and prefix function ‘ $\Pi$ ’ as input, finds a match of p in S. Following pseudocode computes the matching component of KMP algorithm:

KMP-Matcher(S,p)

```
1 n ← length[S]
2 m ← length[p]
3 Π ← Compute-Prefix-Function(p)
4 q ← 0                                //number of characters matched
5 for i ← 1 to n                         //scan S from left to right
6   do while q > 0 and p[q+1] != S[i]
     do q ← Π[q]                          //next character does not match
     if p[q+1] = S[i]
       then q ← q + 1                    //next character matches
     if q = m                            //is all of p matched?
       then print "Pattern occurs with shift" i - m
     q ← Π[ q ]                          // look for the next match
```

*Note: KMP finds every occurrence of a ‘p’ in ‘S’. That is why KMP does not terminate in step 12, rather it searches remainder of ‘S’ for any more occurrences of ‘p’.*



**D Y PATIL**  
DEEMED TO BE  
**UNIVERSITY**  
— RAMRAO ADIK —  
INSTITUTE OF TECHNOLOGY  
NAVI MUMBAI

Illustration: given a String ‘S’ and pattern ‘p’ as follows:

S

b	a	c	b	a	b	a	b	a	c	a	c	a
---	---	---	---	---	---	---	---	---	---	---	---	---

p

a	b	a	b	a	c	a
---	---	---	---	---	---	---

Let us execute the KMP algorithm to find whether ‘p’ occurs in ‘S’.

*For ‘p’ the prefix function,  $\Pi$  was computed previously and is as follows:*

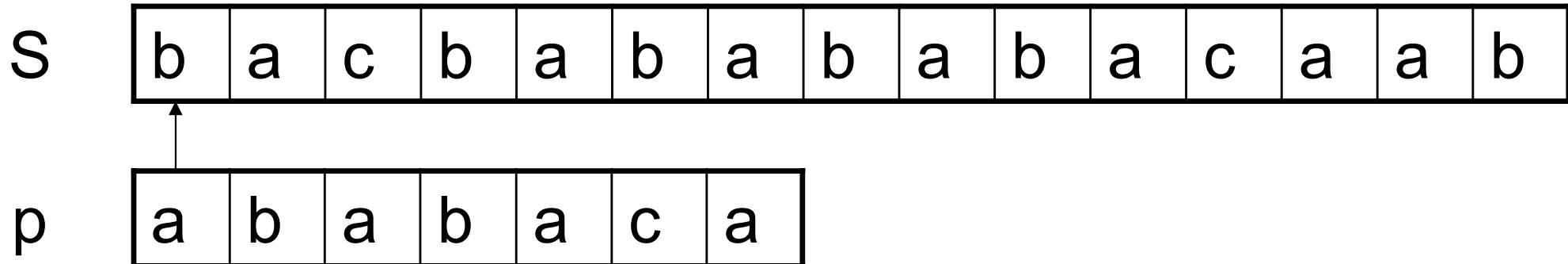
q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
$\Pi$	0	0	1	2	3	1	1



Initially: n = size of S = 15;  
m = size of p = 7

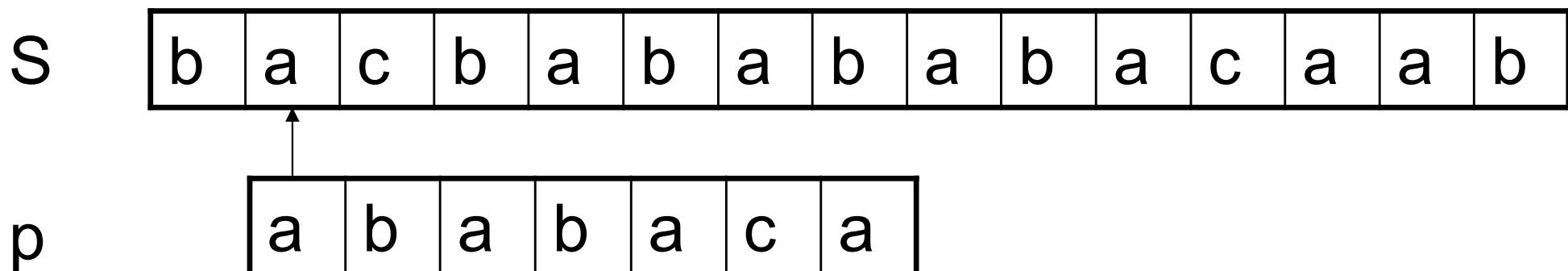
Step 1: i = 1, q = 0  
comparing p[1] with S[1]

---



P[1] does not match with S[1]. 'p' will be shifted one position to the right.

Step 2: i = 2, q = 0  
comparing p[1] with S[2]

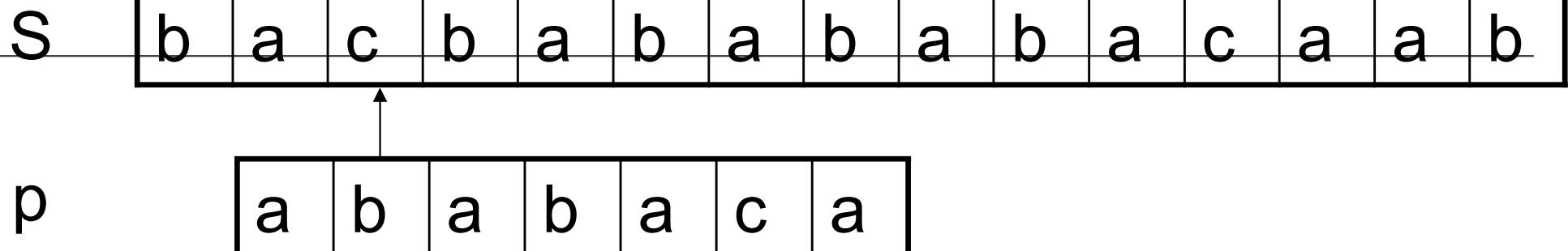


P[1] matches S[2]. Since there is a match, p is not shifted.

Step 3:  $i = 3, q = 1$

Comparing  $p[2]$  with  $S[3]$

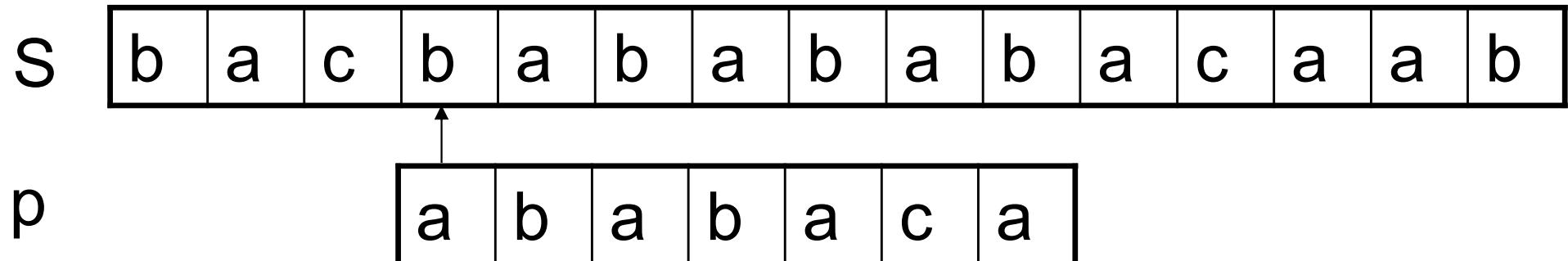
$p[2]$  does not match with  $S[3]$



Step 4:  $i = 4, q = 0$

comparing  $p[1]$  with  $S[4]$

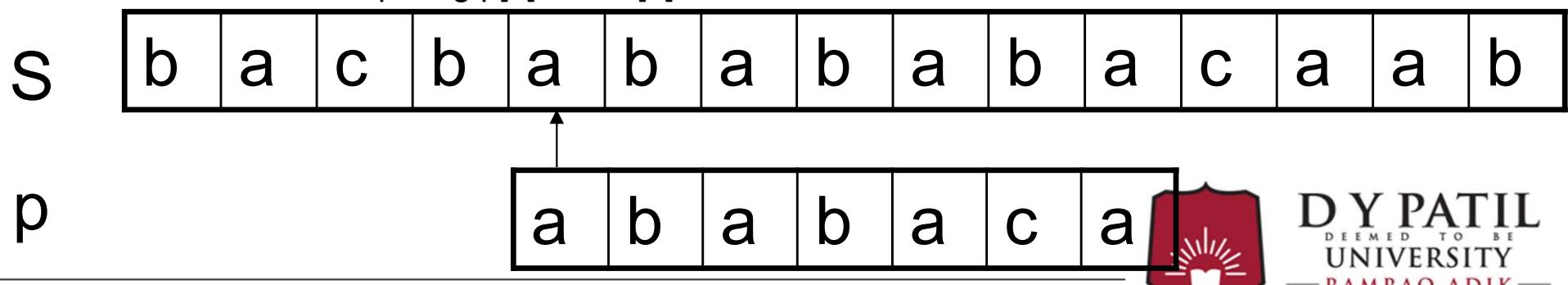
$p[1]$  does not match with  $S[4]$



Step 5:  $i = 5, q = 0$

comparing  $p[1]$  with  $S[5]$

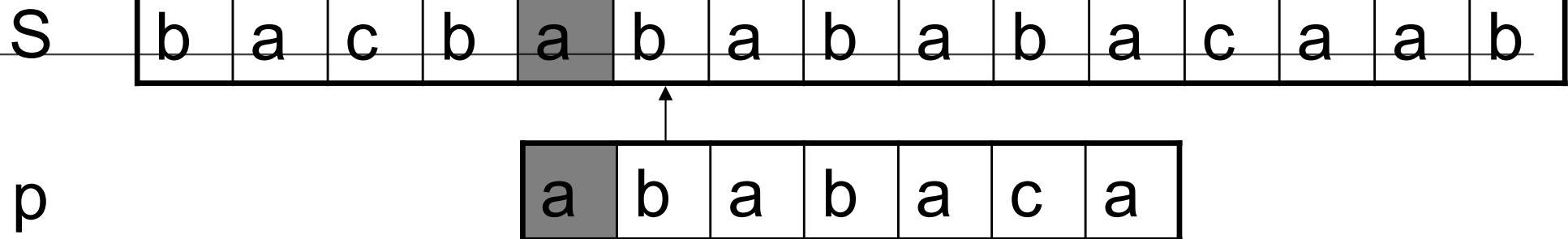
$p[1]$  matches with  $S[5]$



Step 6:  $i = 6, q = 1$

Comparing  $p[2]$  with  $S[6]$

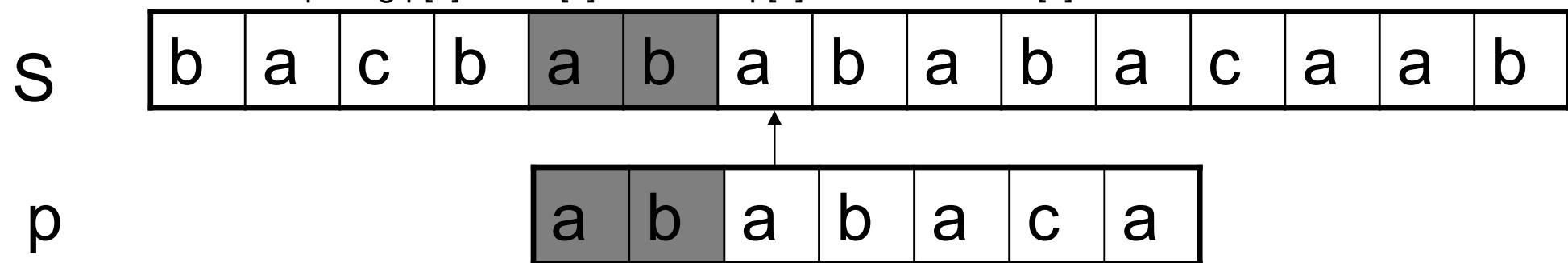
$p[2]$  matches with  $S[6]$



Step 7:  $i = 7, q = 2$

Comparing  $p[3]$  with  $S[7]$

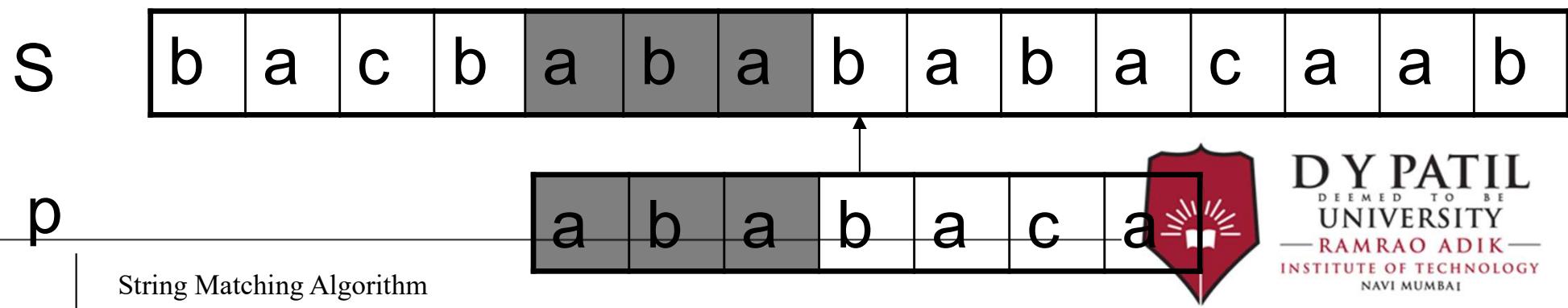
$p[3]$  matches with  $S[7]$



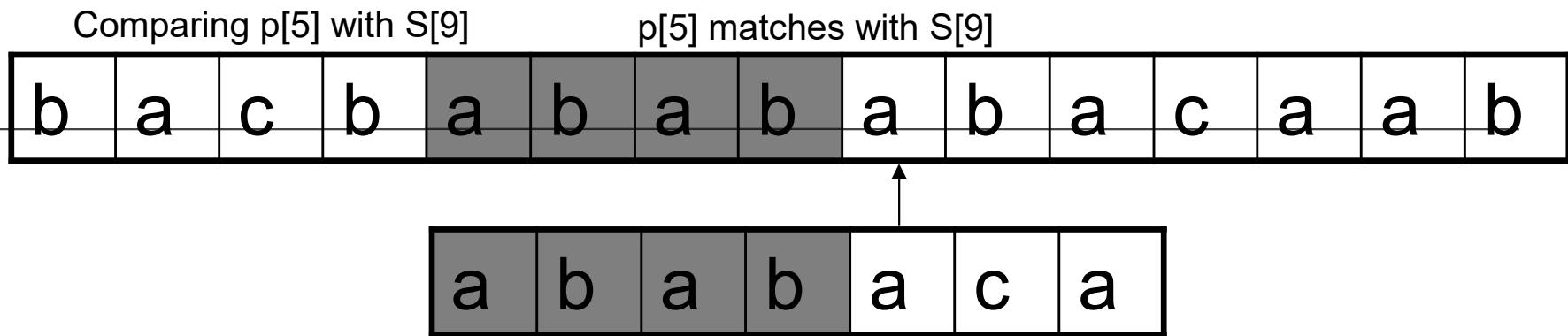
Step 8:  $i = 8, q = 3$

Comparing  $p[4]$  with  $S[8]$

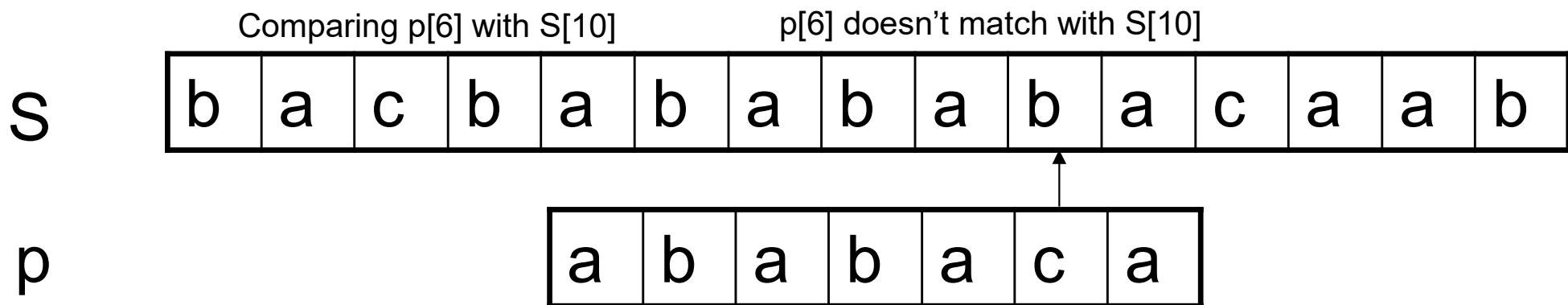
$p[4]$  matches with  $S[8]$



Step 9:  $i = 9$ ,  $q = 4$

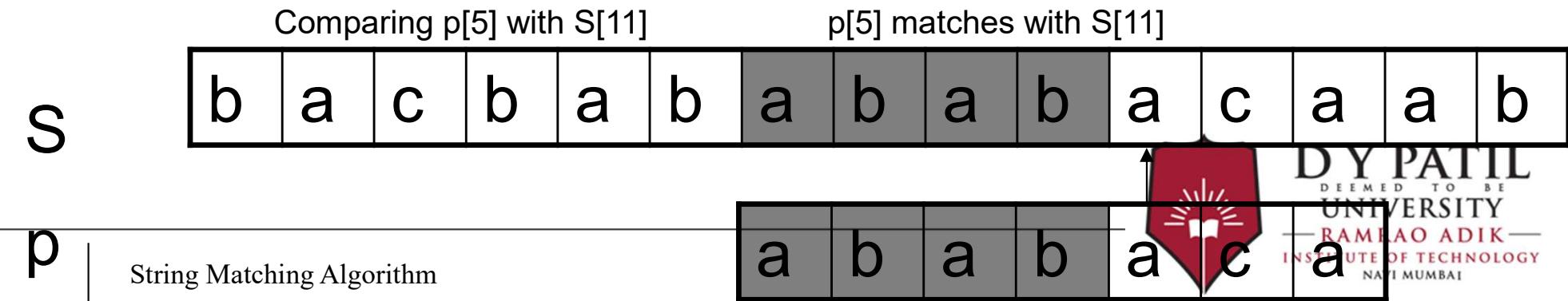


Step 10:  $i = 10$ ,  $q = 5$

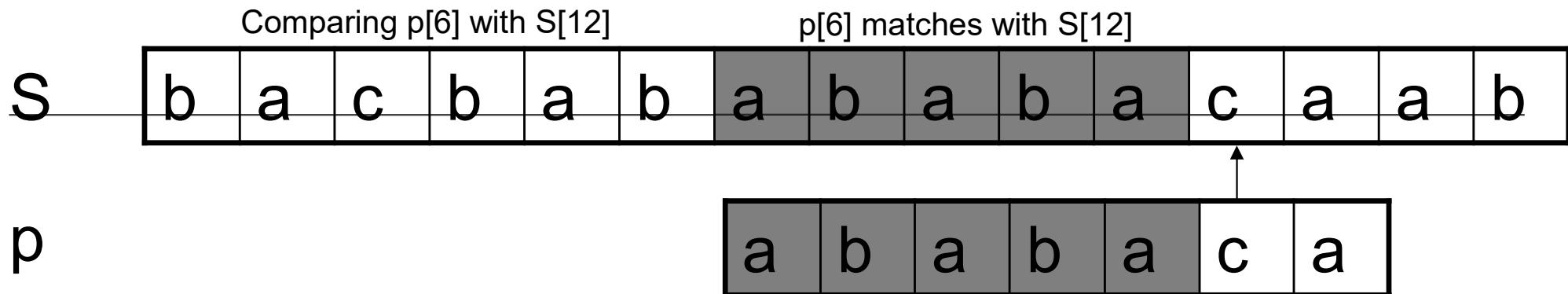


Backtracking on p, comparing  $p[4]$  with  $S[10]$  because after mismatch  $q = \Pi[5] = 3$

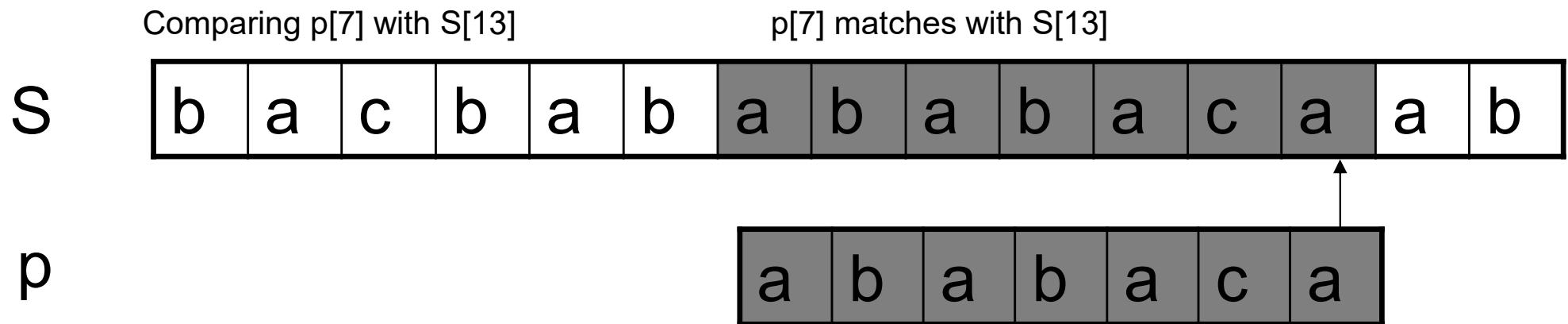
Step 11:  $i = 11$ ,  $q = 4$



Step 12:  $i = 12$ ,  $q = 5$



Step 13:  $i = 13$ ,  $q = 6$



Pattern 'p' has been found to completely occur in string 'S'. The total number of shifts that took place for the match to be found are:  $i - m = 13 - 7 = 6$  shifts.



D Y PATIL  
DEEMED TO BE  
UNIVERSITY  
RAMRAO ADIK  
INSTITUTE OF TECHNOLOGY  
NAVI MUMBAI

# Running - time analysis

- Compute-Prefix-Function ( $\Pi$ )

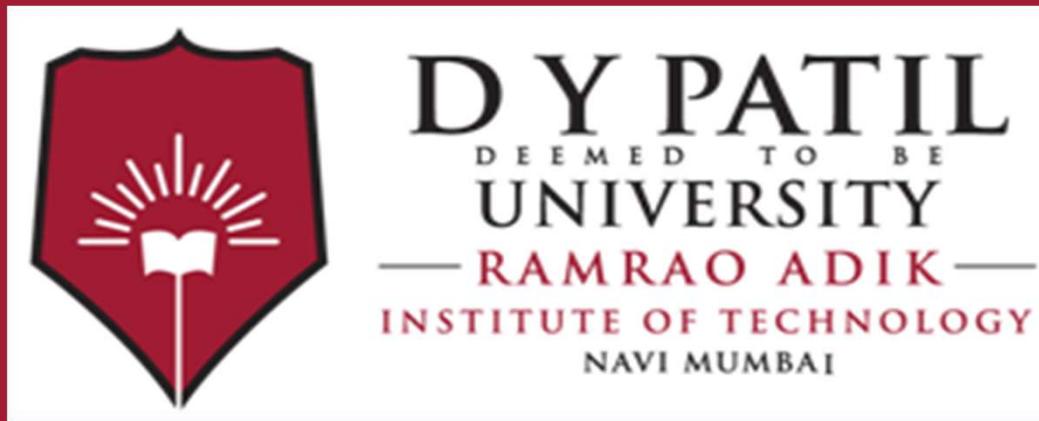
```
m ← length[p]           // 'p' pattern to be  
matched  
 $\Pi[1] \leftarrow 0$   
 $k \leftarrow 0$   
for q ← 2 to m  
    do while k > 0 and p[k+1] != p[q]  
        do k ←  $\Pi[k]$   
        If p[k+1] = p[q]  
            then k ← k + 1  
         $\Pi[q] \leftarrow k$   
return  $\Pi$ 
```

In the above pseudocode for computing the prefix function, the for loop from step 4 to step 10 runs ‘m’ times. Step 1 to step 3 take constant time. Hence the running time of compute prefix function is  $\Theta(m)$ .

- KMP Matcher

```
n ← length[S]  
m ← length[p]  
 $\Pi \leftarrow \text{Compute-Prefix-Function}(p)$   
 $q \leftarrow 0$   
for i ← 1 to n  
    do while q > 0 and p[q+1] != S[i]  
        do q ←  $\Pi[q]$   
        if p[q+1] = S[i]  
            then q ← q + 1  
        if q = m  
            then print “Pattern occurs with shift” i – m  
         $q \leftarrow \Pi[q]$ 
```

The for loop beginning in step 5 runs ‘n’ times, i.e., as long as the length of the string ‘S’. Since step 1 to step 4 take constant time, the running time is dominated by this for loop. Thus running time of matching function is  $\Theta(n)$ .



# Thank You



Expert Talk  
on  
*Tractable and Intractable Problems*

By  
Dr. Vanita Mane

04/04/2025

---

# Basic concepts of P, NP, NP Hard and NP Complete problems



# Deterministic Algorithms

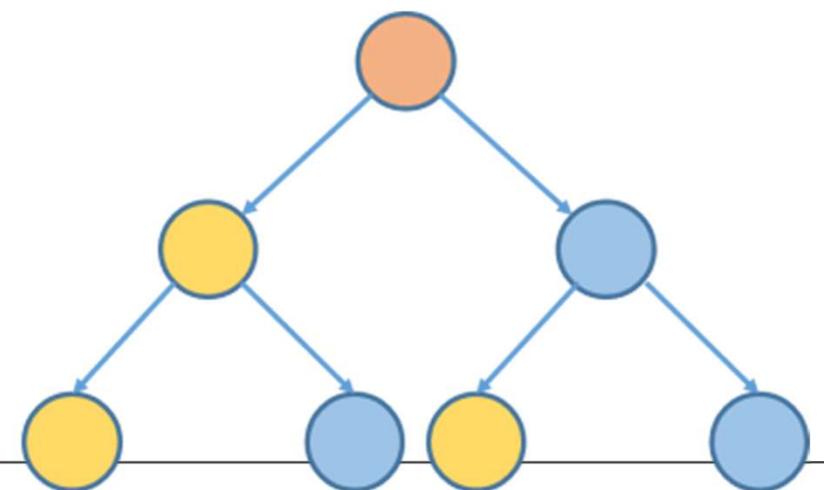
- If we run **deterministic algorithms** multiple times on the same input, they produce the same result every time.
- Deterministic algorithms are typically represented by the state machine, where the machine moves from one state to another state on a specific input.
- The underlying machine always goes through the same states during execution.
- Finding a random number is deterministic, it always returns a random number. Finding odd or even, sorting, finding max, etc. are deterministic. Most of the algorithms used in practice are deterministic in nature.



Deterministic

# Non-deterministic Algorithms

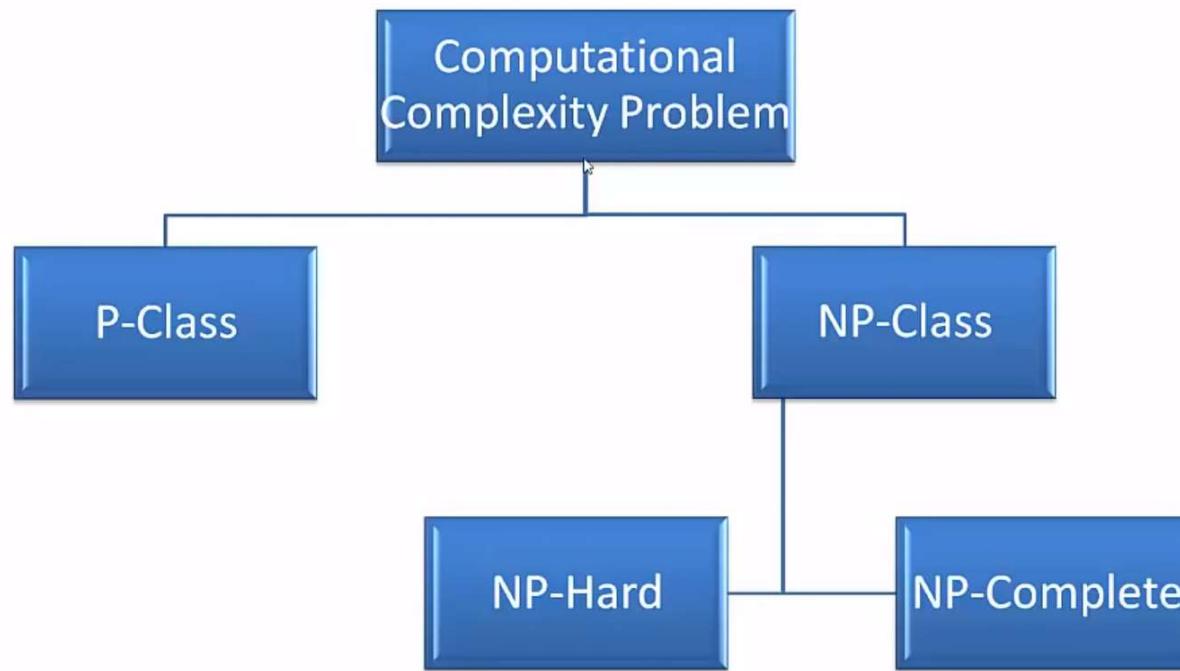
- The **non-deterministic algorithm** works on guessing. For some input, nondeterministic algorithms may produce different outcomes every time.
  - The nondeterministic algorithm operates in two phases: guessing and verification. After guessing the answer, the algorithm verifies its correctness.
  - In the non-deterministic algorithm, each state has a probability of a different outcome on the same input. The vending machine is an example of the deterministic approach, in which the outcome against given input is always deterministic.
  - Randomly picking some element from the list and checking if it is maximum is non-deterministic.
  - Non-deterministic algorithm behaves differently for the same input on different runs.



## Introduction

---

- **NP-Completeness** is a fundamental concept in **computational complexity theory**.
- It helps classify problems based on their computational difficulty.



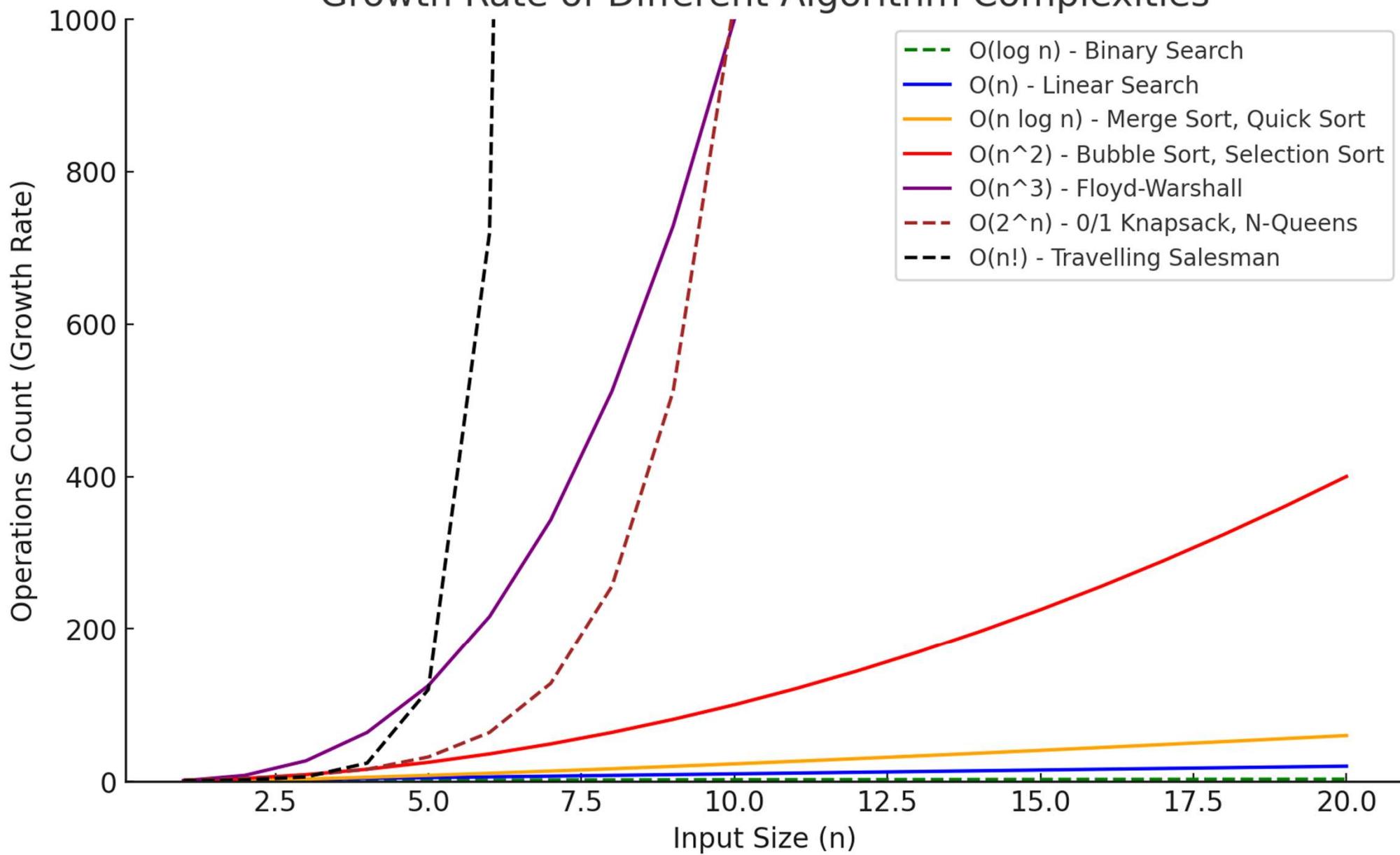
<b>Algorithm</b>	<b>Time Complexity</b>
<b>Sorting Algorithms</b> (Merge Sort, Quick Sort, etc.)	$O(n \log n)$
<b>Searching Algorithms</b> (Binary Search, Linear Search)	$O(\log n)$ or $O(n)$
<b>Dijkstra's Shortest Path Algorithm</b>	$O(V^2)$ (Adj. matrix) or $O((V + E) \log V)$ (Adj. list with priority queue)
<b>Job Sequencing with Deadline</b>	$O(n \log n)$
<b>Fractional Knapsack Problem</b>	$O(n \log n)$
<b>0/1 Knapsack Problem (Dynamic Programming)</b>	$O(nW)$ (Pseudo-polynomial)
<b>Minimum Cost Spanning Tree Algorithms</b> (Prim's, Kruskal's)	$O(E \log V)$
<b>Bellman-Ford Single Source Shortest Path</b>	$O(VE)$
<b>Floyd-Warshall's All Pair Shortest Path</b>	$O(V^3)$
<b>String Matching Algorithms:</b>	
- <b>Naïve String Matching</b>	$O(mn)$
- <b>Rabin-Karp Algorithm</b>	$O(n)$ (on average, worst case $O(mn)$ )
- <b>Finite Automata-based String Matching</b>	$O(n)$
- <b>Knuth-Morris-Pratt (KMP) Algorithm</b>	$O(n)$



<b>Algorithm</b>	<b>Time Complexity</b>
<b>Travelling Salesman Problem (TSP) (Exact Solution)</b>	$O(n!)$ (Brute Force), $O(2^n n^2)$ (DP)
<b>N-Queens Problem</b>	$O(n!)$ (Backtracking)
<b>Sum of Subsets Problem</b>	$O(2^n)$
<b>Graph Coloring</b>	$O(k^n)$ (Backtracking)
<b>Traveling Salesperson Problem (Decision Version)</b>	$O(2^n n^2)$
<b>15-Puzzle Problem</b>	$O(n!)$



## Growth Rate of Different Algorithm Complexities



---

# **Polynomial(P) and non deterministic polynomial(NP)algorithms**



# Polynomial-time algorithm (P Class)

---

- A **polynomial-time algorithm** is an algorithm whose execution time is either given by a polynomial on the size of the input, or can be bounded by such a polynomial. Problems that can be solved by a polynomial-time algorithm are called *tractable* problems.
- These algorithms can be solved in polynomial time  **$O(n^k)$**  for some constant **k**, making them efficiently solvable.

P Class Problem:

A Problem which can be solved on polynomial time is known as P-Class Problem.

Ex: All sorting and searching algorithms.

# Non-deterministic Polynomial Time (NP Class)

- These problems are either **NP-complete or NP-hard**, meaning they have no known polynomial-time solutions, and their solutions can only be verified in polynomial time.
- **NP, for non-deterministic polynomial time**, is one of the best-known complexity classes in theoretical computer science.
- **Exponential Time ( $O(2^n)$ ,  $O(k^n)$ ), Factorial Time ( $O(n!)$ ), Super-Exponential Time (e.g.,  $O(n^n)$ )**

NP Class Problem:

A Problem which cannot be solved on polynomial time but is verified in polynomial time is known as Non Deterministic Polynomial or NP-Class Problem.

Ex: Su-Doku, Prime Factor, Scheduling, Travelling Salesman

# Various problems in NP

---

## Optimization Problems:

» An optimization problem is one which asks, “What is the optimal solution to problem X?”

### » Examples:

- 0-1 Knapsack
- Fractional Knapsack
- Minimum Spanning Tree
- Decision Problems

» An decision problem is one which asks, “Is there a solution to problem X with property Y?”

### » Examples:

- Does a graph G have a MST of weight  $\leq W$ ?



## Various problems in NP (cont...)

---

- An optimization problem tries to find an optimal solution
- A decision problem tries to answer a yes/no question
- Many problems will have decision and optimization versions.
  - » Eg: Traveling salesman problem
- optimization: find hamiltonian cycle of minimum weight
- decision: find hamiltonian cycle of weight  $< k$



# Tractability

---

- In contrast, P (polynomial time) is the set of all decision problems which can be solved in polynomial time by a Turing machine.
- Roughly speaking, if a problem is in P, then it's considered **tractable**, i.e. there exists an algorithm that can solve it in a reasonable amount of time on a computer.
- If a problem is not in P, then it's said to be **intractable**, meaning that for large values it would take far too long for even the best supercomputer to solve it - in some cases, this means millions or even billions of years i.e. as they grow large, we are unable to solve them in reasonable time.

**Polynomial time:  $O(n^2)$ ,  $O(n^3)$ ,  $O(1)$ ,  $O(n \lg n)$**

**Not in polynomial time:  $O(2^n)$ ,  $O(n^n)$ ,  $O(n!)$**

# Polynomial Time Algorithms (P-Class)

Algorithm	Time Complexity	Category
Sorting Algorithms (Merge Sort, Quick Sort, etc.)	$O(n \log n)$	P
Searching Algorithms (Binary Search, Linear Search)	$O(\log n)$ or $O(n)$	P
Dijkstra's Shortest Path Algorithm	$O(V^2)$ (Adj. matrix) or $O((V + E) \log V)$ (Adj. list with priority queue)	P
Job Sequencing with Deadline	$O(n \log n)$	P
Fractional Knapsack Problem	$O(n \log n)$	P
0/1 Knapsack Problem (Dynamic Programming)	$O(nW)$ (Pseudo-polynomial)	P (But exponential in general case)
Minimum Cost Spanning Tree Algorithms (Prim's, Kruskal's)	$O(E \log V)$	P
Bellman-Ford Single Source Shortest Path	$O(VE)$	P
Floyd-Warshall's All Pair Shortest Path	$O(V^3)$	P
String Matching Algorithms:		
- Naïve String Matching	$O(mn)$	P
- Rabin-Karp Algorithm	$O(n)$ (on average, worst case $O(mn)$ )	P
- Finite Automata-based String Matching	$O(n)$	P
- Knuth-Morris-Pratt (KMP) Algorithm	$O(n)$	P

# Non-Deterministic Polynomial Time Algorithms (NP Class)

Algorithm	Time Complexity	Category
Travelling Salesman Problem (TSP) (Exact Solution)	$O(n!)$ (Brute Force), $O(2^n n^2)$ (DP)	NP-Hard
N-Queens Problem	$O(n!)$ (Backtracking)	NP-Complete
Sum of Subsets Problem	$O(2^n)$	NP-Complete
Graph Coloring	$O(k^n)$ (Backtracking)	NP-Complete
Traveling Salesperson Problem (Decision Version)	$O(2^n n^2)$	NP-Complete
15-Puzzle Problem	$O(n!)$	NP-Complete

# Classification of Algorithms

P	NP
Sorting Searching Dijkstra's Shortest Path Algorithm Job Sequencing with Deadline Fractional Knapsack Problem 0/1 Knapsack Problem Minimum cost spanning tree algorithms	Travelling salesman problem N-Queens problem Sum of subsets Graph coloring Traveling Salesperson Problem 15-Puzzle Problem



# Intractable Problems

---

- Can be classified in various categories based on their degree of difficulty, e.g.,
  - NP
  - NP-complete
  - NP-hard
- Let's define NP algorithms and NP problems ...

The NP Class Problems, it is verified in polynomial time.

The P Class Problems, not only it is solved on polynomial time but it is verified also in polynomial time.

NP Class

Hard to Solve  
&  
Easy to Verify  
Exponential  
time

NP

P

P Class

Easy to Solve  
&  
Easy to Verify  
Polynomial time

Tractable

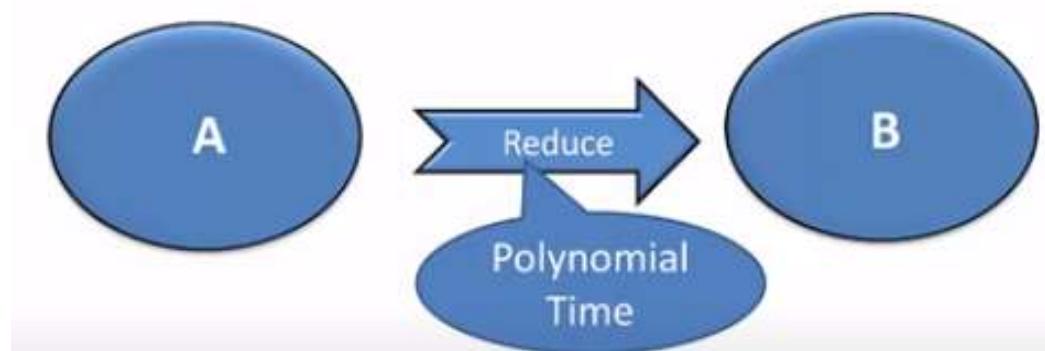
$P \subseteq NP$

Intractable

## Reduction

---

- We have two problems, A and B, and we know problem B is a P class problem. If problem A can be reduced, or converted to problem B, and this reduction takes a polynomial amount of time, then we can say that A is also a P class problem (A is reducible to B).

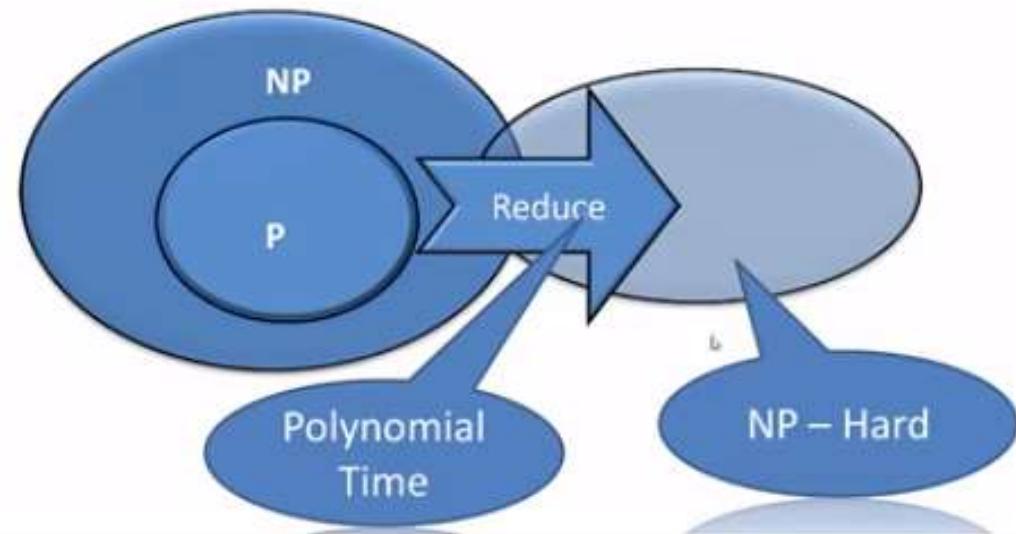


## NP-Hard Problems

- A problem is classified as NP-Hard when an algorithm for solving it can be translated to solve *any* NP problem. Then we can say, this problem is *at least* as hard as any NP problem, but it could be much harder or more complex.
- **NP-hard**-- Now suppose we found that A is reducible to B, then it means that B is at least as hard as A.

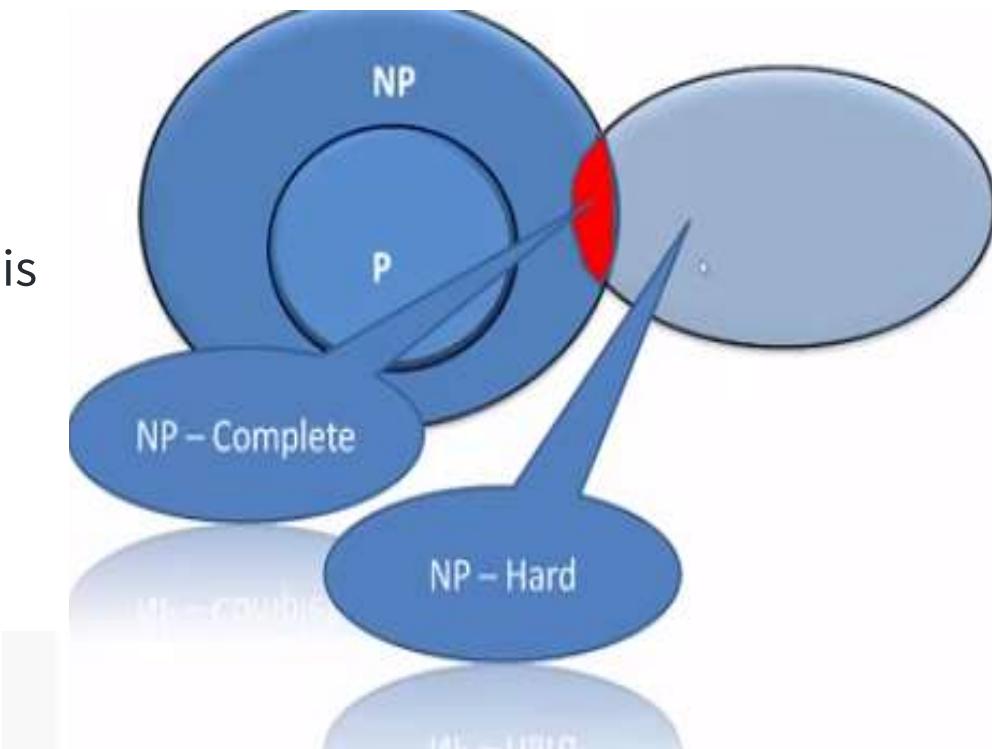
### NP Hard Problem:

A Problem is NP-Hard if every problem in NP can be polynomial reduced to it.



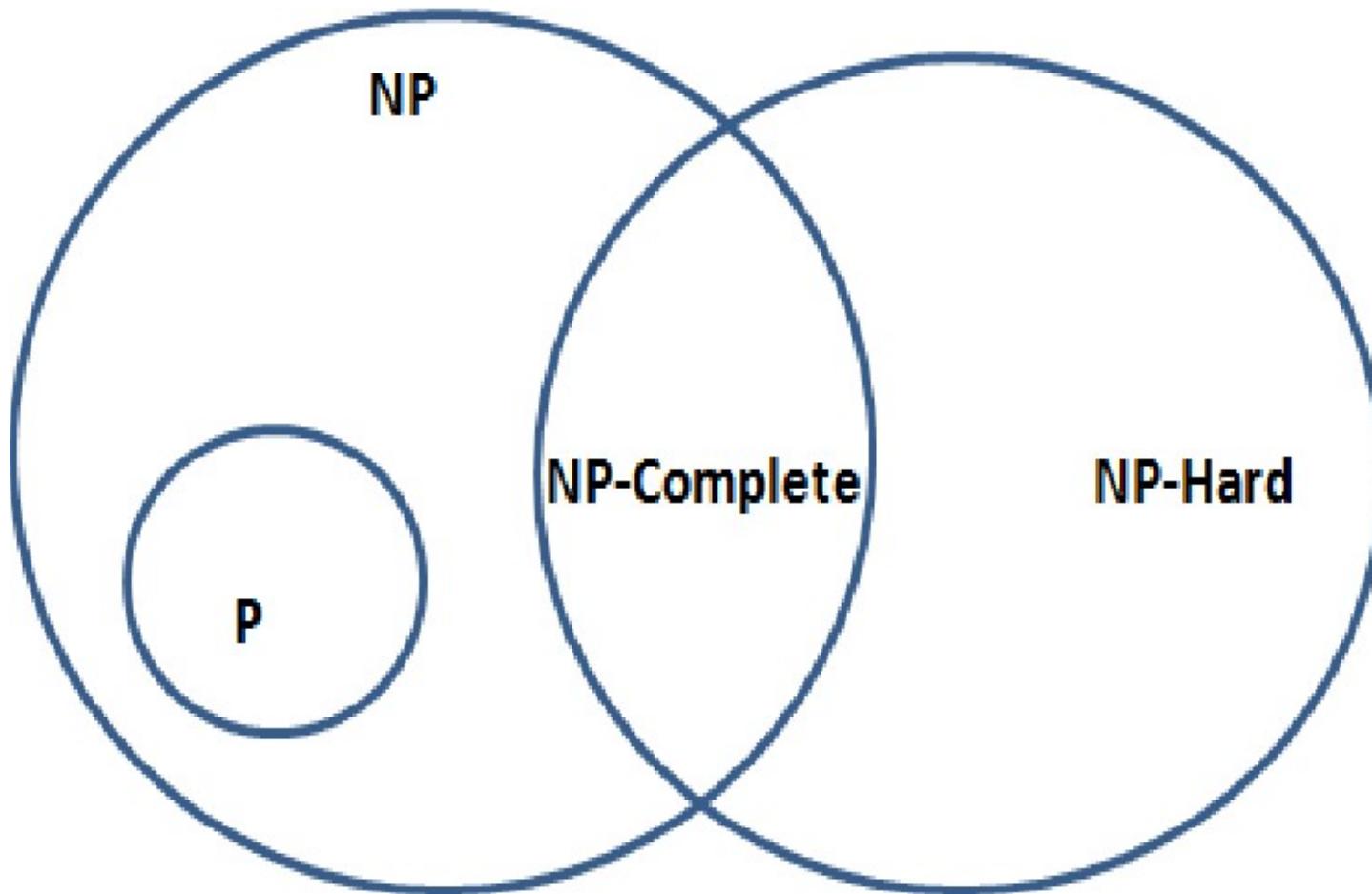
## NP-Complete Problems

- NP-Complete problems are problems that live in both the NP and NP-Hard classes. This means that NP-Complete problems can be verified in polynomial time and that any NP problem can be reduced to this problem in polynomial time.
- The group of problems which are both in NP and NP-hard are known as NP-Complete problem.
- Now suppose we have a NP-Complete problem R and it is reducible to Q then Q is at least as hard as R and since R is an NP-hard problem. therefore Q will also be at least NP-hard , it may be NP-complete also.



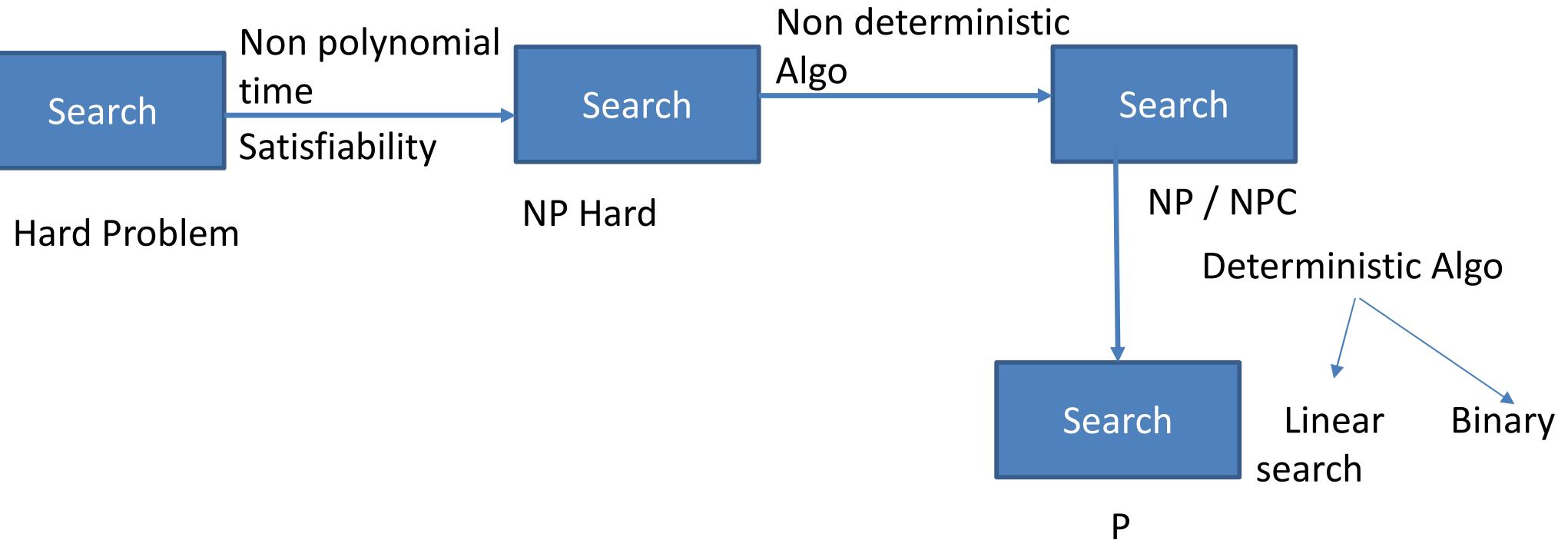
## Venn diagram

---



# Example

---



---

# NP-Completeness



# **NP-complete problems**

---

- A decision problem  $D$  is *NP-complete* iff
  1.  $D \in NP$
  2. every problem in  $NP$  is *polynomial-time reducible* to  $D$  ( $D \in NP\text{-Hard}$ )
- Other *NP-complete problems* obtained through polynomial-time reductions of known *NP-complete* problems
- **Reduction**
  - ❖ A problem  $P$  can be *reduced* to another problem  $Q$  if any instance of  $P$  can be rephrased to an instance of  $Q$ , the solution to which provides a solution to the instance of  $P$ 
    - » This rephrasing is called a *transformation*
  - ❖ Intuitively: If  $P$  reduces in polynomial time to  $Q$ ,  $P$  is “no harder to solve” than  $Q$

# Polynomial-time reductions

---

- **Informal explanation of reductions:**
- We have two problems, X and Y. Suppose we have a black-box solving problem X in polynomial-time. Can we use the black-box to solve Y in polynomial-time?
- If yes, we write  $Y \leq_P X$  and say that Y is **polynomial-time reducible to X**.
- More precisely, we take any input of Y and in polynomial number of steps translate it into an input (or a set of inputs) of X. Then we call the black-box for each of these inputs. Finally, using a polynomial number of steps we process the output information from the boxes to output the answer to problem Y.

# NP-complete and NP-hard: how to prove

---

- **The steps to prove NP-hardness of a problem X:**
  1. Find an already known NP-hard problem Y.
  2. Show that  $Y \leq_P X$ .
- **The steps to prove NP-completeness of a problem X:**
  1. Show that Y is NP-hard.
  2. Show that Y is in NP.

# Proving NP-Completeness

---

*What steps do we have to take to prove a problem Q is NP-Complete?*

- » Pick a known NP-Complete problem P
- » Reduce P to Q
  - Describe a transformation that maps instances of P to instances of Q, s.t. “yes” for Q = “yes” for P
  - Prove the transformation works
  - Prove it runs in polynomial time
- » Prove  $Q \in \text{NP}$

---

# Proving Travelling Salesman Problem to NP Complete Problem

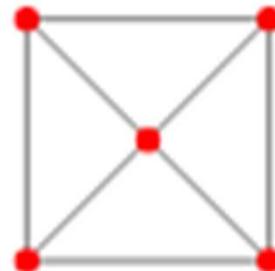


## A Hamiltonian cycle

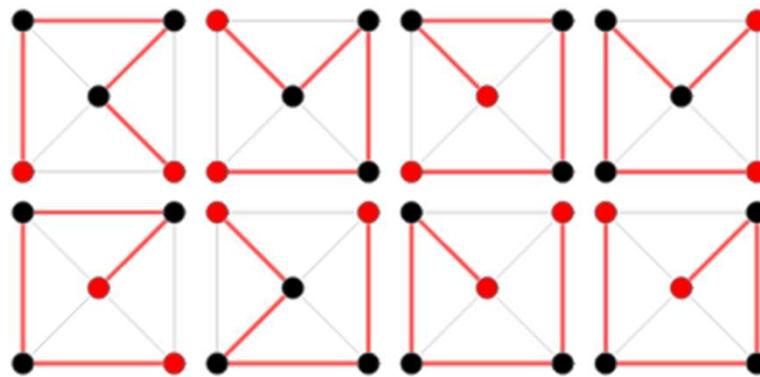
---

- A *Hamiltonian cycle*, *Hamiltonian circuit*, *vertex tour* or *graph cycle* is a cycle that visits each vertex exactly once.

*5-wheel graph*

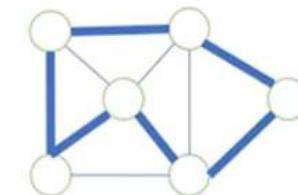
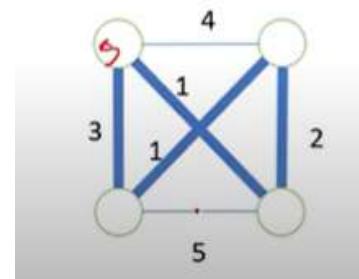


*A Hamiltonian cycles of above graph*



# Travelling Salesman Problem

- The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip

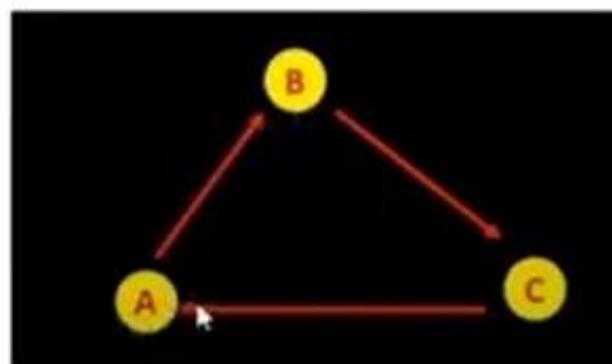


- Check all permutations: about  $O(n!)$  extremely slow
- Dynamic programming:  $O(n^2 2^n)$
- No significantly better upper bound is known

Brute Force Approach

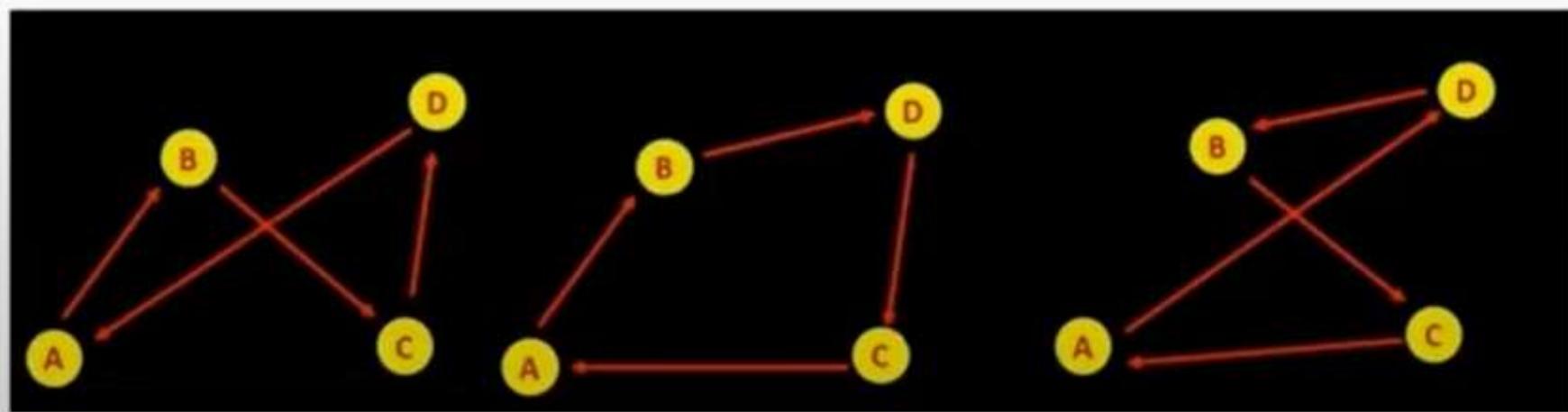
# Why is TSP – NP Hard ?

- For  $n=3$  nodes, there is one only path for TSP.



No of TSP solutions =  
 $(n-1)!/2$

- For  $n=4$  nodes, no of paths is  $(4-1)!/2 = 3!/2 = 3$  paths.



# Why is TSP – NP Hard ?

- For  $n=31$ , no of paths =  $(n-1)!/2 = 30!/2 = 1.3262642990609552931815424e+32$   
    ↳
- Given Processor speed = 1GHz (No of instructions executed/sec)
- Hence, 1 sec =  $1 * 10^9$
- Time taken for  $1.3262642990609552931815424e+32$  instructions =  $30!/(2 * 10^9) = 1,32,62,64,29,90,60,95,52,93,18,154.24$  seconds
- $1,32,62,64,29,90,60,95,52,93,18,154.24$  seconds/ $(24 * 60) = 9,21,01,68,74,34,78,85,62,026.496$  days
- $9,21,01,68,74,34,78,85,62,026.496$  days/ $365 = 2,52,33,33,90,23,22,97,430.20957808219178$  years

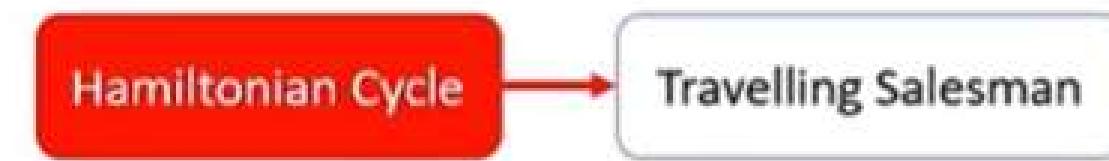
So ,it is impossible to find all possible solutions and then get the optimal solution. Hence , it is NP Hard.

# Proving Travelling Salesman Problem to NP Complete Problem

---

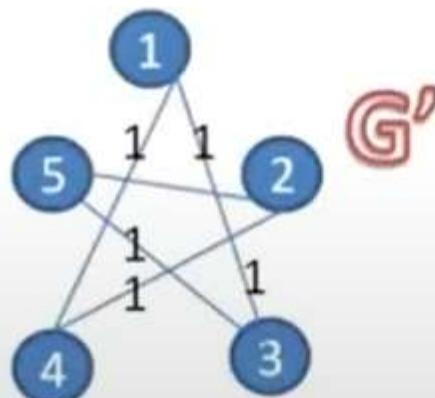
- To prove ***TSP is NP-Complete***,
  1. ***TSP belongs to NP.***
  2. ***TSP is NP-hard***

To prove this, one way is to show that ***Hamiltonian cycle*  $\leq_p$  *TSP*** (as we know that the Hamiltonian cycle problem is NP complete).

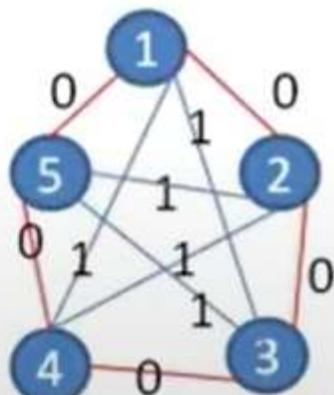


# Reduction of Hamiltonian circuit (G) to TSP(G) in polynomial time

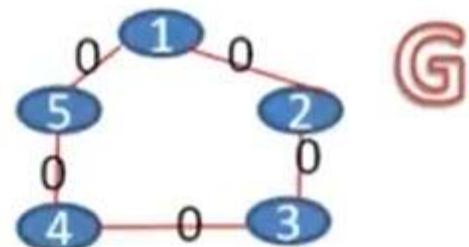
- $G(V, E)$ , Hamiltonian Path = 1->2->3->4->5
- To form a TSP:
- Step 1 : Construct a complement graph  $G'$  (takes  $O(n)$  time)
- Step 2: Construct a complete Graph by combining  $G \& G'$
- Define Cost function:  $C(i,j) = \begin{cases} 0, & \text{if } (i,j) \in G, \\ 1, & \text{if } (i,j) \in G' \end{cases}$



1->4->2->5->3



Converting unweighted  $G$  to weighted  $G'$  takes linear time  $O(n)$



	1	2	3	4	5
1	NL	1	0	0	1
2		NL	1	0	0
3			NL	1	0
4				NL	1
5					NL

	1	2	3	4	5
1	NL	0	1	1	0
2		NL	0	1	1
3			NL	0	1
4				NL	0
5					NL

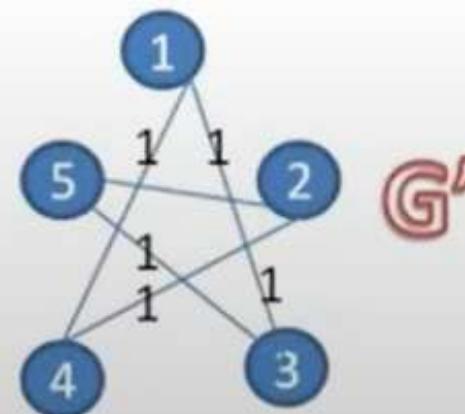
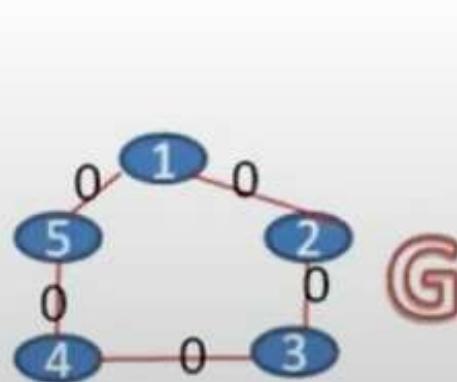
Activate V

Setting

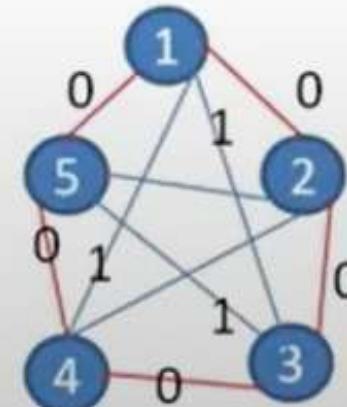
# Reduction of Hamiltonian circuit ( $G$ ) to TSP( $G$ ) in polynomial time

Hamiltonian Cycle problem reduced to an instance of TSP:

1. The actual reduction is quite simple, with the translation from unweighted to weighted graph easily performed in linear time.
2. Further, this translation is designed to ensure that the answers of the two problems will be identical.
3. If the graph  $G$  has a Hamiltonian cycle , then this exact same tour will correspond to  $n$  edges in  $E'$ , each with weight 1. Therefore, this gives a TSP tour of  $G'$  of weight exactly  $n$ .
4. If  $G$  does not have a Hamiltonian cycle, then there can be no such TSP tour in  $G'$ , because the only way to get a tour of cost  $n$  in  $G'$  would be to use only edges of weight 1, which implies a Hamiltonian cycle in  $G$ .



TSP Tour : 1 -> 4 -> 2 -> 5 -> 3->1  
Weight = n=5



## Proving Travelling Salesman Problem to NP Complete Problem(cont..)

---

- Thus we can say that the graph  $G'$  contains a TSP if graph  $G$  contains Hamiltonian Cycle. Therefore, any instance of the Travelling salesman problem can be reduced to an instance of the hamiltonian cycle problem. Thus, the TSP is NP-Hard.
- As TSP is both NP and NP Hard it is NP Complete problem.

---

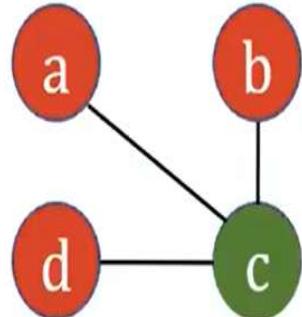
# Proving Vertex Cover Problem to NP Complete Problem



# Vertex Cover Problem

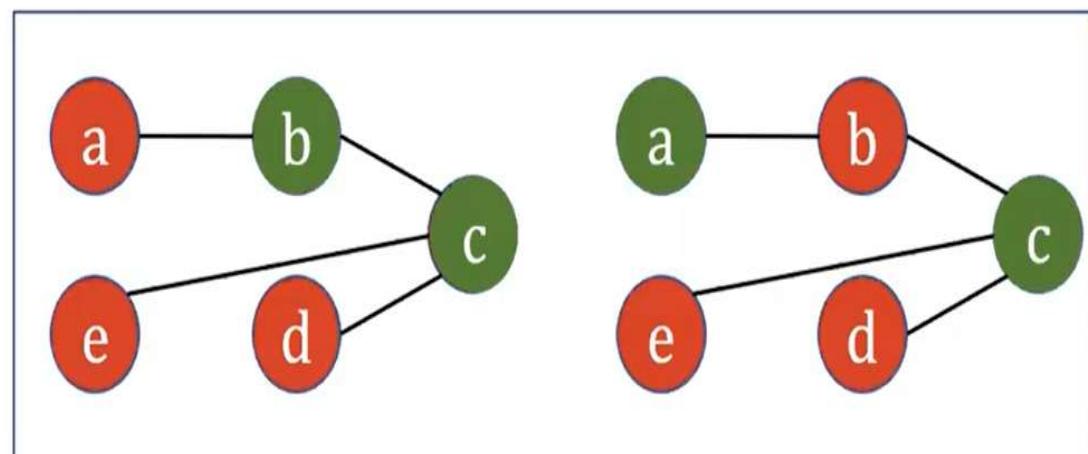
## Introduction

- A Vertex cover of a Graph is a subset of vertices that can cover every edge
  - An edge is covered if one of the endpoint is chosen
- **Vertex Cover Problem**
  - Given an undirected Graph, find Minimum Size Vertex Cover



Minimum Vertex Cover  
is Empty []

Minimum Vertex Cover  
is [c]



Minimum Vertex Cover is [c, b] or [c, a]

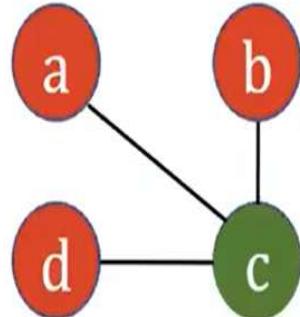
# Vertex Cover Problem

## Introduction

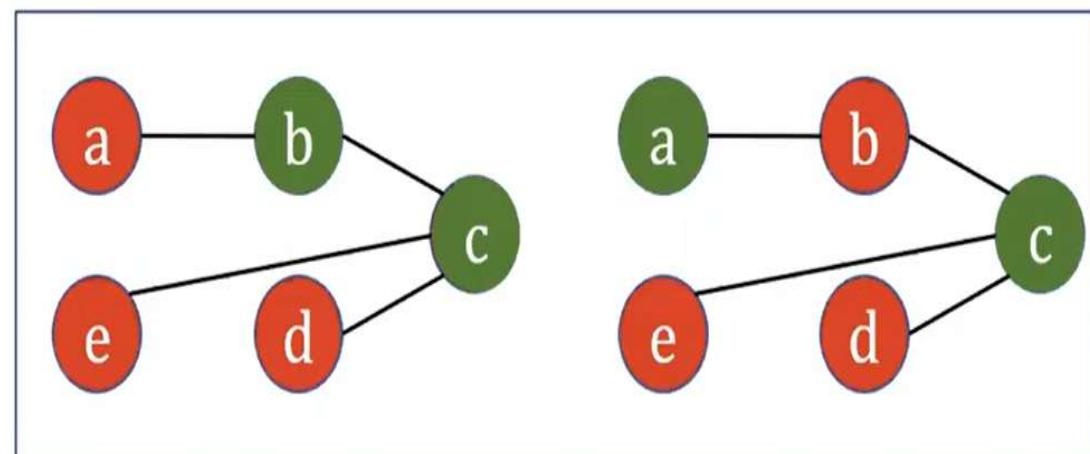
- A Vertex cover of a Graph is a subset of vertices that can cover every edge
  - An edge is covered if one of the endpoint is chosen
- **Vertex Cover Problem**
  - Given an undirected Graph, find Minimum Size Vertex Cover



Minimum Vertex Cover  
is Empty []

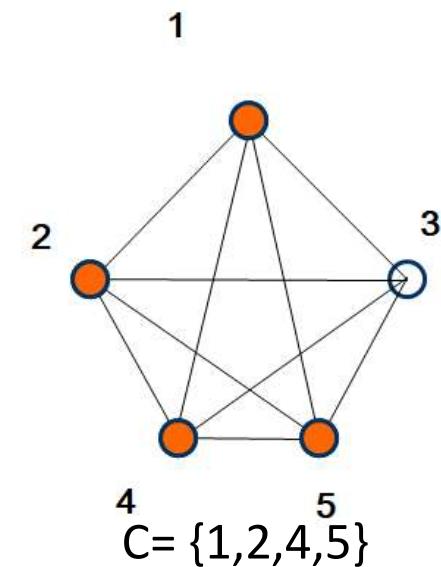
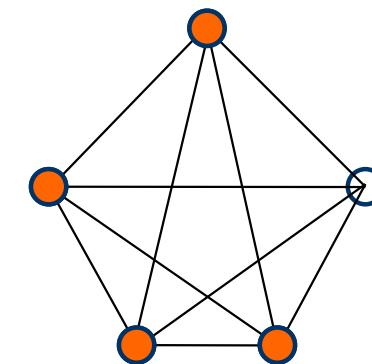
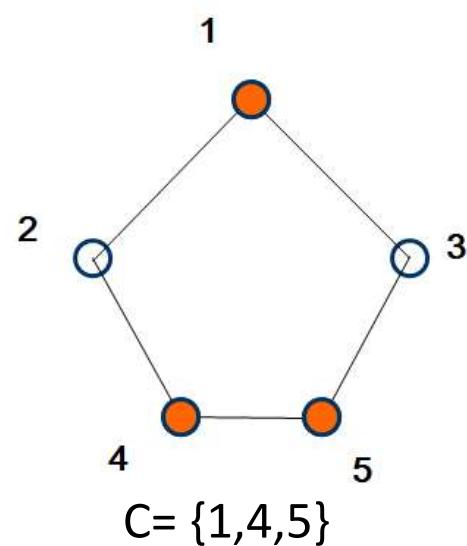
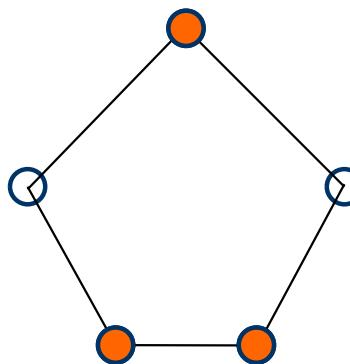
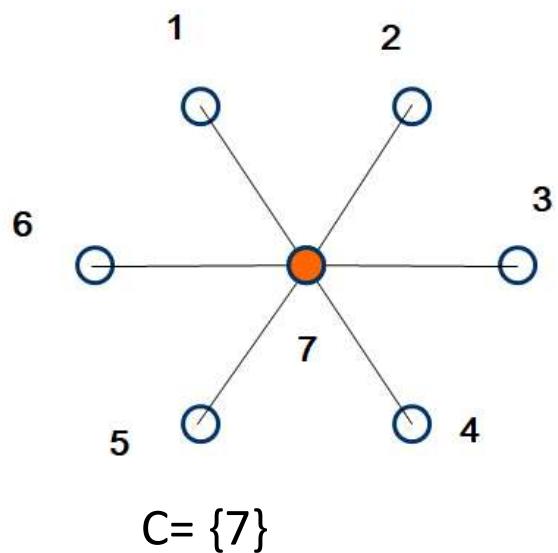
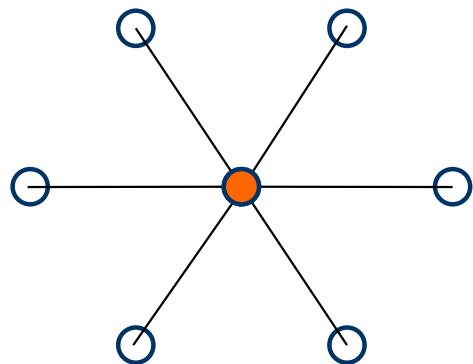


Minimum Vertex Cover  
is [c]



Minimum Vertex Cover is [c, b] or [c, a]

## Vertex Cover Problem- Example



# Vertex Cover Problem

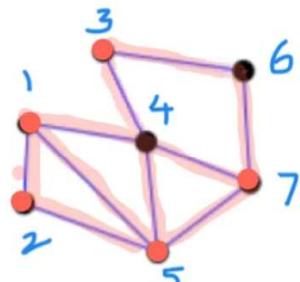
---

## ***Definition***

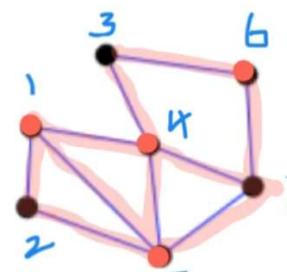
- A vertex cover of an undirected graph is a subset of its vertices such that for every edge  $(u, v)$  of the graph, either 'u' or 'v' is in vertex cover
  
- Solutions offered to Vertex Cover Problem
  - 1. Brute Force Approach
  - 2. Greedy Approach
  - 3. Approximation Algorithm



## 1. Brute Force Approach

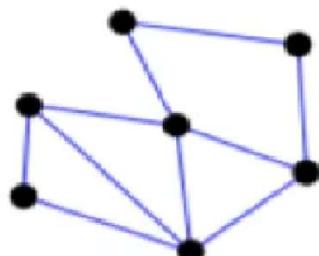


$\{1, 2, 3, 5, 7\}$

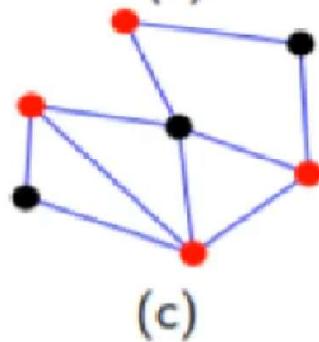


$C = \{1, 4, 5, 6\}$   
valid vertex cover

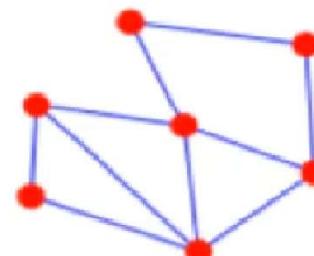
$$|C| = 4$$



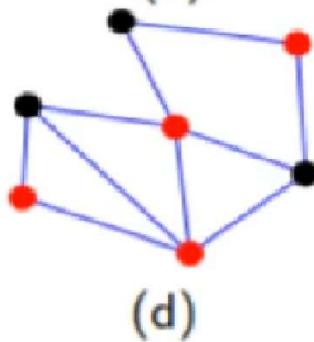
(a)



(c)



(b)



(d)

$\{1, 2, 3, \dots, n\}$

$$\downarrow 2^n$$

$O(2^n)$

Figure: (a) An undirected graph (b) A trivial vertex cover (c) A vertex cover (d) An other vertex cover

## Proving Vertex Cover Problem to NP Complete Problem

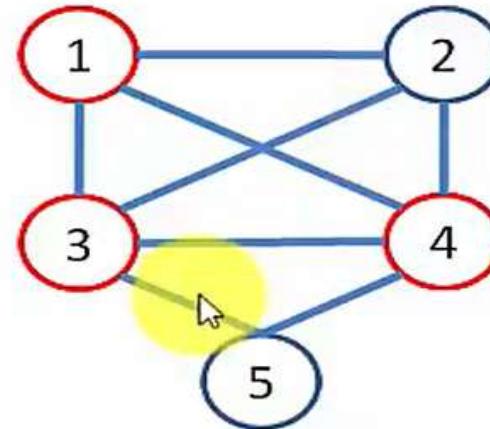
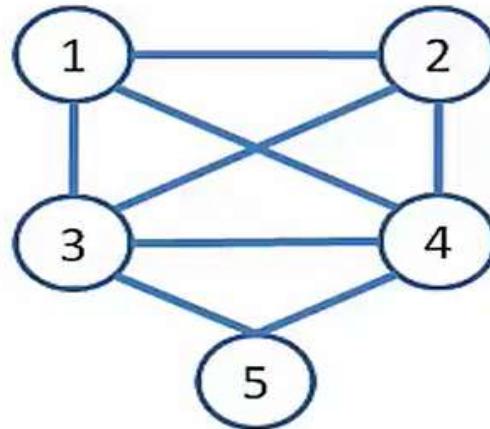
---

### Vertex Cover Problem

A vertex cover of a graph is a set of vertices that touches every edge in the graph.

A vertex cover of a graph  $G = (V, E)$  is a subset  $V_c \subseteq V$  such that if  $(a, b) \in E$  then either  $a \in V_c$  OR  $b \in V_c$  OR both.

# Vertex Cover Problem



A vertex cover of a graph  $G = (V, E)$  is a subset  $V_c \subseteq V$  such that if  $(a, b) \in E$  then either  $a \in V_c$  OR  $b \in V_c$  OR both  $a, b \in V_c$ .

## Steps for proving NP-Complete:

Step 1: Prove that B is in NP

Step 2: Select an NP-Complete Language A.

Step 3: Construct a function  $f$  that maps members of A to members of B.

Step 4: Show that  $x$  is in A iff  $f(x)$  is in B.

Step 5: Show that  $f$  can be computed in polynomial time.



## NP-Completeness of Vertex Cover Problem:

Vertex Cover Problem is in NP.

Vertex Cover Problem is NP- Hard.

## NP-Completeness of Vertex Cover Problem:

To Show Vertex Cover Problem is in NP.

A Problem which cannot be solved on polynomial time but is verified in polynomial time is known as Non Deterministic Polynomial or NP-Class Problem.

Given  $V_c$ , vertex cover of  $G = (V, E)$ ,  $|V_c| = k$ . We can check in  $O(|V| + |E|)$  that  $V_c$  is a vertex cover for  $G$ .

For each  $v \in V_c$ , remove all incident edges.  
Check if all edges were removed from  $G$ .

Thus Vertex Cover Problem is in NP.



## NP-Completeness of Vertex Cover Problem:

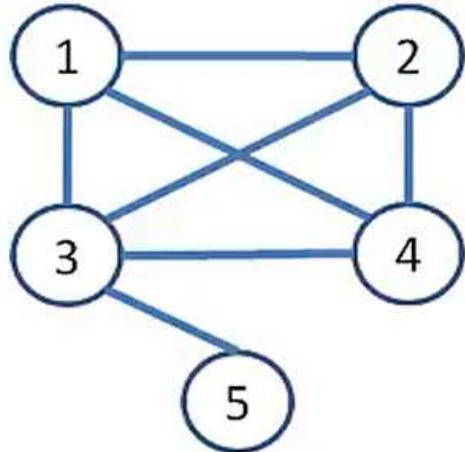
To show Vertex Cover Problem is NP-Hard.

we need to show that Vertex Cover is at least as hard any other problem in NP.

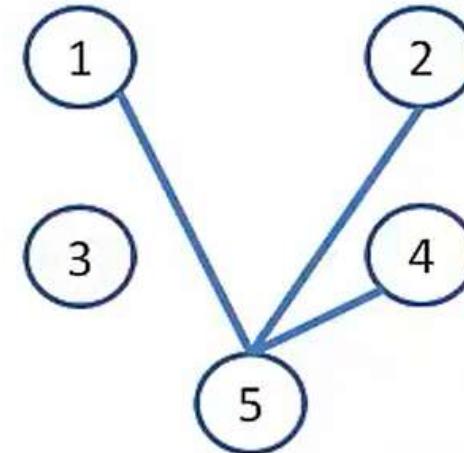
we give a reduction from Clique to Vertex Cover Problem.

It means, given an instance  $I$  of Clique, we will produce a graph  $G(V,E)$  and an integer  $k$  such that  $G$  has a maximum clique of  $k$  if and only if  $I$  in  $\bar{G}(V,\bar{E})$  has a vertex cover of size  $|V|-k$ .

## Proving Vertex Cover Problem to NP Complete Problem

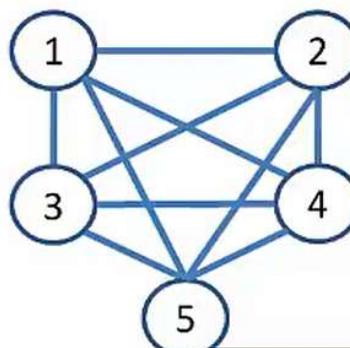


$G(V, E)$



$\bar{G}(V, \bar{E})$

$G(V, E)$



$\bar{G}(V, \bar{E})$

Complement  $\bar{G}(V, \bar{E})$  of a graph  $G(V, E)$  is a graph such that  $\bar{E} = \{(u, v) \in G \mid (u, v) \notin \bar{G}\}$

OR

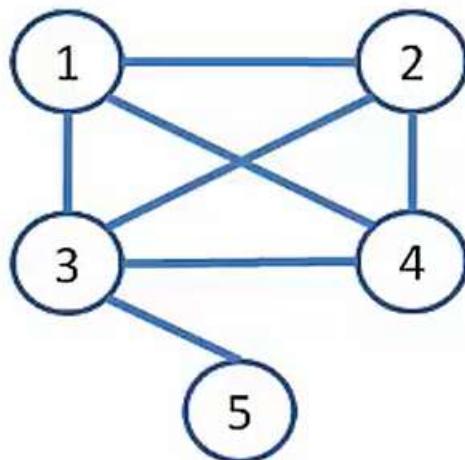
$G(V, E) \cup \bar{G}(V, \bar{E})$  is a complete graph

$G(V, E)$ ,  
 $|V| = n$ ,  
Clique =  $k$

Reduce

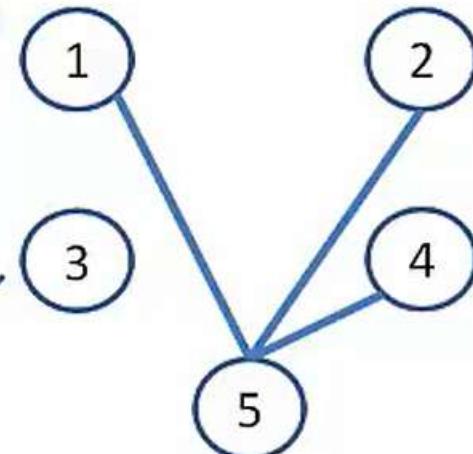
Polynomial  
Time

$\bar{G}(\bar{V}, \bar{E})$   
Vertex Cover  
 $= |V| - k$



|V|=5  
Max. Clique( $k$ )=4

Vertex Cover is  
 $|V| - k = 1$



Let  $G$  has clique  $V'$  of size  $k$ .

$\Rightarrow G$  has vertex cover of size  $|V|-k$

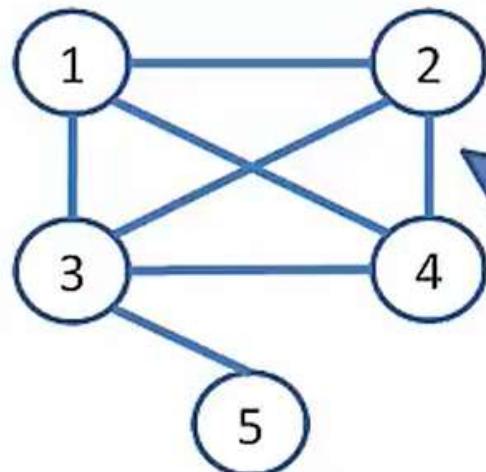
$(a, b) \in E \Rightarrow (a, b) \notin \bar{E}$

If  $(a, b) \in \bar{E}$ , then at least a or b  $\notin V'$ .

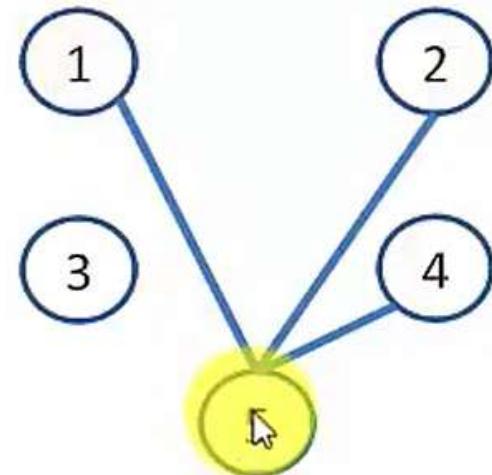
Every pair in  $V'$  is connected by an edge in  $E$ .

$\Rightarrow$  At least one of a or b is in  $V-V'$

$\Rightarrow$  Edge  $(a,b)$  is covered by  $V-V'$



$$V = \{1, 2, 3, 4, 5\}$$
$$V' = \{1, 2, 3, 4\}$$
$$V - V' = \{5\}$$



## Proving Vertex Cover Problem to NP Complete Problem

---

- Thus, we can say that there is a clique of size  $k$  in graph  $G$  if and only if there is a vertex cover of size  $|V| - k$  in  $G'$ , and hence, any instance of the clique problem can be reduced to an instance of the vertex cover problem. Thus, vertex cover is NP Hard. Since vertex cover is in both NP and NP Hard classes, it is NP Complete.

<https://www.geeksforgeeks.org/proof-that-vertex-cover-is-np-complete/>

# Summary

Polynomial Time (P): Efficient –Runs in Polynomial Time

Non-Deterministic Polynomial Time (NP Class): Inefficient- Not running in Polynomial Time

Feature	Polynomial Time (P)	Non-Polynomial Time
Efficiency	Fast for large inputs	Impractical for large inputs
Examples	Sorting, Searching, Shortest Path, Job Sequencing, Fractional Knapsack, MST, String Matching.	TSP, N-Queens, Sum of Subsets, Graph Coloring, 15-Puzzle.
Complexity	$O(n^k)$	$O(2^n)$ , $O(n!)$ , $O(n^n)$
Tractability	Tractable	Intractable



# Thank You



**D Y PATIL**  
— RAMRAO ADIK —  
**INSTITUTE OF**  
**TECHNOLOGY**  
NAVI MUMBAI