

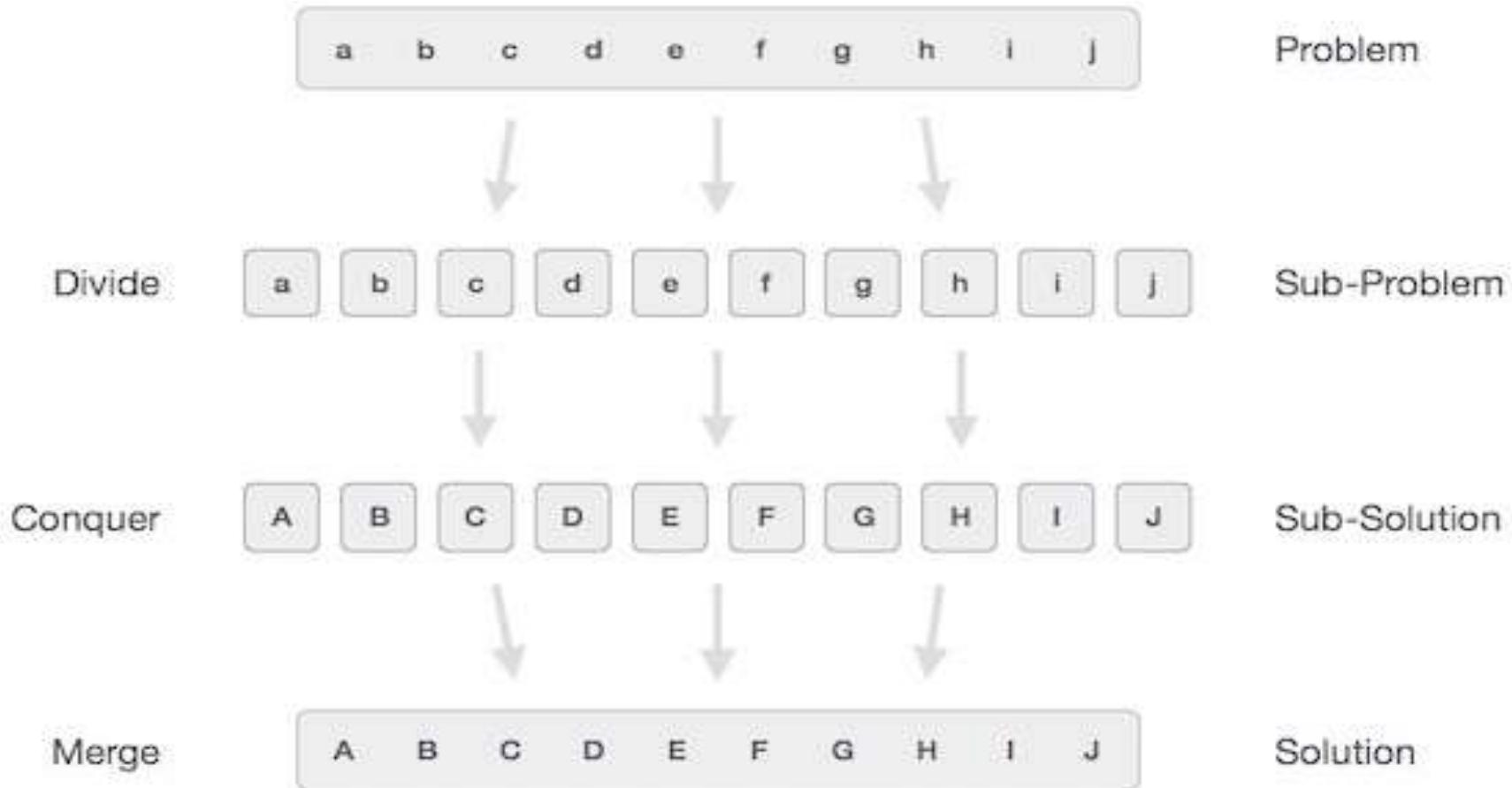
# UNIT-2

- Introduction
- Divide and Conquer
- Maximum Subarray Problem

## Introduction

- In divide and conquer method, the problem is divided into smaller sub-problems and then each problem is solved independently.
- When the subproblems is divided into even smaller sub-problems, it should reach a stage where no more division is possible.
- Those "atomic" smallest possible sub-problem (fractions) are solved.
- The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.

## Divide-and-conquer method in a three-step process:



## **Three parts of Divide-and-conquer method:**

**Divide:** This involves dividing the problem into some sub problem.

**Conquer:** Sub problem by calling recursively until sub problem solved.

**Combine:** The Sub problem Solved so that we will get find problem solution.

The following are some standard algorithms that follows Divide and Conquer algorithm.

1. Binary Search is a searching algorithm. In each step, the algorithm compares the input element  $x$  with the value of the middle element in array. If the values match, return the index of the middle. Otherwise, if  $x$  is less than the middle element, then the algorithm recurs for left side of the middle element, else recurs for the right side of the middle element.

2. Quicksort is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.

**3. Merge Sort** is also a sorting algorithm. The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.

**4. Closest Pair of Points** The problem is to find the closest pair of points in a set of points in x-y plane. The problem can be solved in  $O(n^2)$  time by calculating distances of every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solves the problem in  $O(n \log n)$  time.

**5. Strassen's Algorithm** is an algorithm to multiply two matrices. A simple method to multiply two matrices need 3 nested loops and is  $O(n^3)$ .

## Divide And Conquer algorithm :

```
DAC(a, i, j)
{
    if(small(a, i, j))
        return(Solution(a, i, j))
    else
        m = divide(a, i, j)                      // f1(n)
        b = DAC(a, i, mid)                       // T(n/2)
        c = DAC(a, mid+1, j)                     // T(n/2)
        d = combine(b, c)                        // f2(n)
    return(d)
}
```

## Recurrence Relation for DAC algorithm :

This is recurrence relation for above program.

$O(1)$  if  $n$  is small

$$T(n) = f_1(n) + 2T(n/2) + f_2(n)$$

## **Example:**

To find the maximum in a given array.

**Input:** { 100, 70, 60, 50, 80, 10, 1 }

**Output:** The maximum number in a given array is : 100

## **For Maximum:**

The recursive approach is used to find maximum where only two elements are left using condition i.e. if( $a[\text{index}] > a[\text{index}+1]$ .)

In a program line  $a[\text{index}]$  and  $a[\text{index}+1]$  condition will ensure only two elements in left.

```
if(index >= l-2)
{
    if(a[index]>a[index+1])
    {
        // (a[index] // the last element will be maximum in a given array.
    }
    else
    {
        // (a[index+1] // last element will be maximum in a given array.
    }
}
```

Recursive function to check the right side at the current index of an array.

*max = DAC\_Max(a, index+1, l); // Recursive call*

```
// Right element will be maximum.
if(a[index]>max)
return a[index];
// max will be maximum element in a given array.
else
return max;
}
```

- Maximum Subarray Problem

Generic problem:

Find a maximum subarray of  $A[\text{low} \dots \text{high}]$  with initial call:  $\text{low} = 1$  and  $\text{high} = n$

DC strategy:

1. Divide  $A[\text{low} \dots \text{high}]$  into two subarrays of as equal size as possible by finding the midpoint  $\text{mid}$
2. Conquer:
  - (a) finding maximum subarrays of  $A[\text{low} \dots \text{mid}]$  and  $A[\text{mid} + 1 \dots \text{high}]$
  - (b) finding a max-subarray that crosses the midpoint
3. Combine: returning the max of the three

Correctness:

This strategy works because any subarray must either lie entirely in one side of midpoint or cross the midpoint.

- Maximum Subarray Problem

Given an array **arr[ ]** of integers, the task is to find the maximum sum sub-array among all the possible sub-arrays.

**Examples:**

**Input:**  $\text{arr[ ]} = \{-2, 1, -3, 4, -1, 2, 1, -5, 4\}$

**Output:** 6

$\{4, -1, 2, 1\}$  is the required sub-array.

**Input:**  $\text{arr[ ]} = \{2, 2, -2\}$

**Output:** 4

- Maximum Subarray Problem

Divide and conquer algorithms generally involves dividing the problem into sub-problems and conquering them separately. For this problem, a structure is used to store the following values:

1. Total sum for a sub-array.
2. Maximum prefix sum for a sub-array.
3. Maximum suffix sum for a sub-array.
4. Overall maximum sum for a sub-array.(This contains the max sum for a sub-array).

- Maximum Subarray Problem

- During the recursion(Divide part) the array is divided into 2 parts from the middle. The left node structure contains all the above values for the left part of array and the right node structure contains all the above values. Having both the nodes, now we can merge the two nodes by computing all the values for resulting node.
- The max prefix sum for the resulting node will be maximum value among the maximum prefix sum of left node or left node sum + max prefix sum of right node or total sum of both the nodes (which is possible for an array with all positive values).

- Maximum Subarray Problem

- Similarly the max suffix sum for the resulting node will be maximum value among the maximum suffix sum of right node or right node sum + max suffix sum of left node or total sum of both the nodes (which is again possible for an array with all positive values).
- The total sum for the resulting node is the sum of both left node and right node sum. Now, the max subarray sum for the resulting node will be maximum among prefix sum of resulting node, suffix sum of resulting node, total sum of resulting node, maximum sum of left node, maximum sum of right node, sum of maximum suffix sum of left node and maximum prefix sum of right node.

- Maximum Subarray Problem

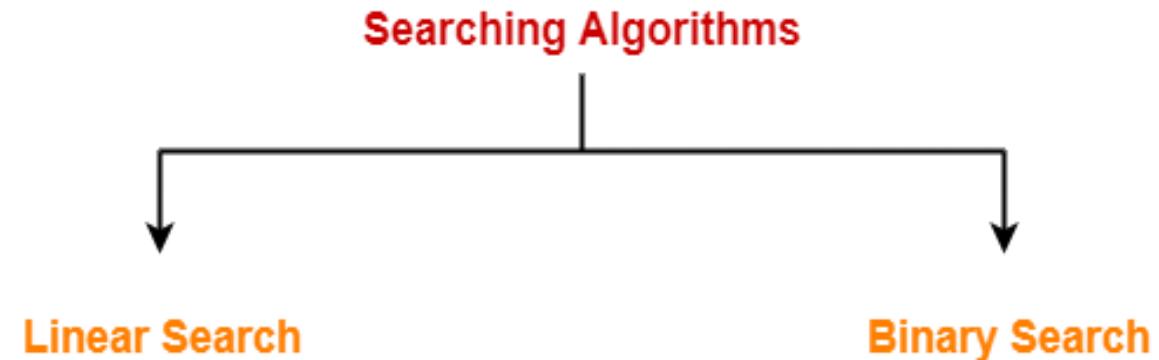
- **Time Complexity:** The recursive function generates the following recurrence relation.  
 $T(n) = 2 * T(n / 2) + O(1)$

# **BINARY SEARCH**

BinarySearch

Complexity of binary search

# BINARY SEARCH



# Binary Search

**Session Learning Outcome-SLO-**Solve problems using divide and conquer approaches

- **Motivation of the topic**

Binary Search is one of the fastest searching algorithms.

- It is used for finding the location of an element in a linear array.
- It works on the principle of divide and conquer technique.

*Binary Search Algorithm can be applied only on **Sorted arrays**.*

- So, the elements must be arranged in-
  - Either ascending order if the elements are numbers.
  - Or dictionary order if the elements are strings.
- To apply binary search on an unsorted array,
  - First, sort the array using some sorting technique.
  - Then, use binary search algorithm.

- Binary search is an efficient searching method. While searching the elements using this method the most essential thing is that the elements in the array should be a sorted one.
- An element which is to be searched from the list of elements stored in the array  $A[0\text{---}n-1]$  is called as Key element.
- Let  $A[m]$  be the mid element of array A. Then there are three conditions that need to be tested while searching the array using this method.
- They are given as follows
  - ❖ If  $\text{key}==A[m]$  then desired element is present in the list.
  - ❖ Otherwise if  $\text{key} <= A[m]$  then search the left sub list
  - ❖ Otherwise if  $\text{Key}>=A[m]$  then search the right sub list The following algorithm explains about binary search.

# Recursive Binary search algorithm

## Algorithm

Input: A array A[0...n-1] sorted in ascending orderand search key K.

Output: An index of array element which is equal to k

Low<- 0;high <-n-1

while low ≤ high do

    mid <- (low+high)/2

    if K=A[mid] return mid

    else if K,A[m] high <- mid-1

    else low<-mid+1

return

# EXAMPLE FOR BINARY SEARCH

**Step:1** Consider a list of elements sorted in array A as

0	1	2	3	4	5	6	
10	20	30	40	50	60	70	

Low                                  high

The Search key element is key=60

Now to obtain middle element we will apply formula

$$\text{mid} = (\text{low} + \text{high}) / 2$$

$$\text{mid} = (0 + 6) / 2$$

$$\text{mid} = 3$$

Then check  $A[\text{mid}] == \text{key}$ ,  $A[3] = 40$

$A[3] \neq 60$  Hence condition failed

Then check  $\text{key} > A[\text{mid}]$ ,  $A[3] = 40$

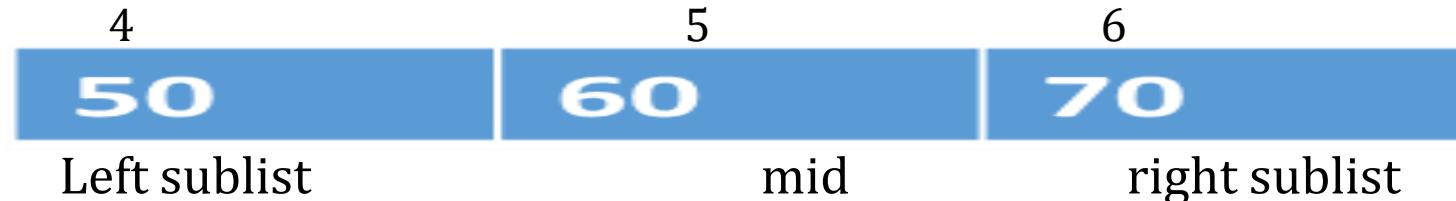
$60 > A[3]$  Hence condition satisfied so search the Right Sublist

## Step 2:

- The Right Sublist is



- Now we will again divide this list to check the mid element



- $\text{mid}=(\text{low}+\text{high})/2$
- $\text{mid}=(4+6)/2$
- $\text{mid}=5$
- Check if  $A[\text{mid}] == \text{key}$
- (i.e)  $A[5] == 60$ . Hence condition is satisfied. The key element is present in position 5.
- The number is present in the Array A[ ] at index position 5.**

Thus we can search the desired number from the list of the elements.

# ANALYSIS

- The basic operation in binary search is comparison of search key with array elements.
- To analyze efficiency of binary search we must count the number of times the search gets compared with the array elements.
- The comparison is also called three way comparisons because the algorithm makes the comparison to determine whether key is smaller, equal to or greater than  $A[m]$ .
- In this algorithm after one comparison the list of  $n$  elements is divided into  $n/2$  sub list.
- The worst case efficiency is that the algorithm compares all the array elements for searching the desired element.
- In this method one comparison is made and based on the comparison array is divided each time in  $n/2$  sub list.

- Hence worst case time complexity is given by

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

Time required to	one comparison made
Compare left or	with middle element
Right sub list	

Also  $C_{\text{worst}}(1) = 1$

- But as we consider the rounded down values when array gets divided the above equation can be written as

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

$$C_{\text{worst}}(1) = 1$$

- We can analyse the best case , Worst case and Average case . The complexity of binary search is given as follows

Best case	Average case	Worst Case
$\theta(1)$	$\theta(\log n)$	$\theta(\log n)$

- **Advantages of Binary search:**
  - Binary search is an optimal searching algorithm using which we can search the desired element very efficiently
- **Disadvantages of binary Search :**
  - This Algorithm requires the list to be sorted . Then only this method is applicable
- **Applications of binary search:**
  - The binary search is an efficient searching method and is used to search desired record from database
  - For solving with one un known this method is used

## Summary:

- Binary Search time complexity analysis is done below-
  - In each iteration or in each recursive call, the search gets reduced to half of the array.
  - So for  $n$  elements in the array, there are  $\log_2 n$  iterations or recursive calls.
- **Time Complexity of Binary Search Algorithm is  $O(\log_2 n)$ .**
  - Here,  $n$  is the number of elements in the sorted linear array.

## Home assignment:

- Search the Element 15 from the given array using Binary Search Algorithm.

3	10	15	20	35	40	60
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

## Binary Search Example

# Merge Sort with Time complexity analysis

Design and Analysis of Algorithms

# Session Learning Outcome-SLO

- At the end of this session, you should be able to perform merge sort and also evaluate its time complexity

# Introduction

- Attributed to a Hungarian mathematician John van Neumann
- Used for sorting unordered arrays
- Uses divide-and-conquer strategy
- **1<sup>st</sup> phase** is to **divide** the array of numbers into 2 equal parts
  - If necessary, these subarrays are divided further
- **2<sup>nd</sup> phase** is the **conquer** part
  - Involves sorting of subarrays recursively and combine the sorted arrays to give the final sorted list

# Merge sort – informal procedure

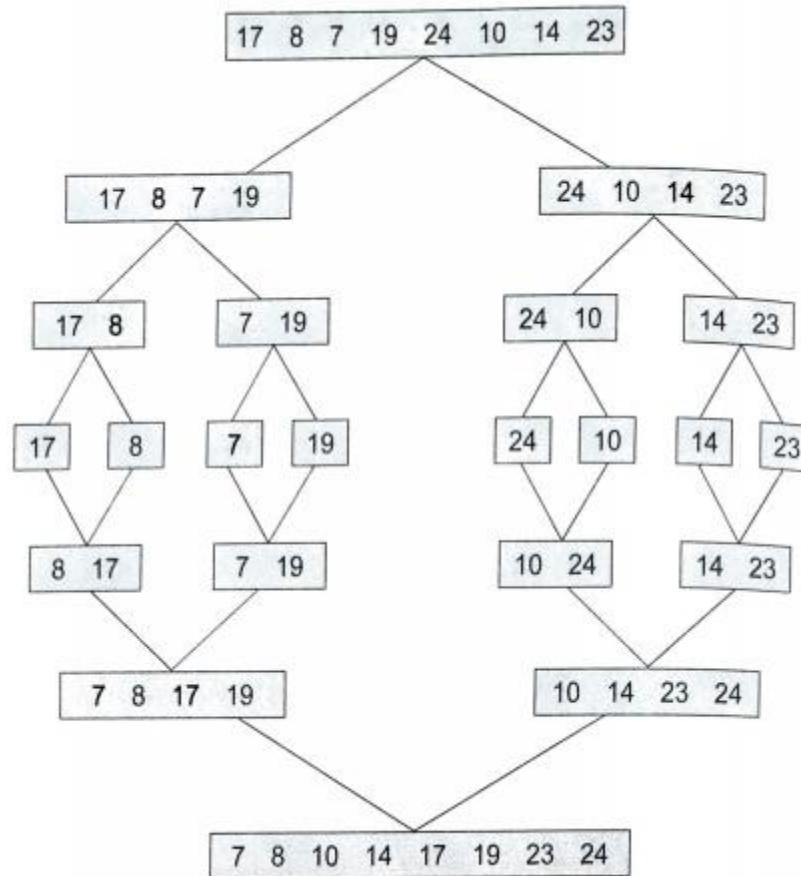
Step1: Divide an array into subarrays B and C

Step 2: Sort subarray D recursively; this yields B sorted subarray

Step 3: Sort subarray C recursively; this yields C sorted subarray

Step 4: Combine B and C sorted subarrays to obtain the final sorted array A

# MergeSort- An Example



# Algorithm mergesort(A[first .. Last])

**Input:** Unsorted array A with first =1 and last=n

**Output:** Sorted array A

Begin

```
    if (first == last) then
        return A[first]
    else
        mid = (first+last)/2
        for i ∈ {1,2, ..., mid}
            B[i] = A[i]
        End for
        for i ∈ {mid +1, ..., n}
            C[i] = A[i]
        End for
        mergesort(B[1 .. mid])
        mergesort(C[mid + 1 .. n])
        merge(B,C,A)
    End
```

# Algorithm merge(B,C,A)

**Input:** Two sorted arrays B and C

**Output:** Sorted array A

Begin

i=1

j=1

k=1

m = length(B)

n = length(C)

while ((i<=m) and (j <= n)) do

if(B[i] < C[j]) then

A[k] = B[i]

i=i+1

else

A[k] = C[i]

j=j+1

End if

k=k+1

end while

if (i > m) then

while k <= m + n do

A[k] = C[j]

j,k = j+1, k+1

end while

else if (j < n) then

while (k <= m+n) do

A[k] = B[j]

i,k = i+1,k+1

end while

end if

return (A)

end

# Complexity Analysis

- $T(n) = 2T\left(\frac{n}{2}\right) + n - 1, \text{ for } n \geq 2$   
= 1, for  $n=2$   
= 0, when  $n<2$

# Summary

- Uses divide and conquer strategy
- a comparison based sort
- out of place merge sort
- Stable Algorithm
- Merging method is used
- Variants of merge sort
  - in place
  - bottom up merge sort
  - top down merge sort

# Questions

- Why is merge sort an out-of-place sorting technique?
- Does it use recursive procedure?
- What are the best, worst and average case time complexities?

# Reference

- S. Sridhar, Design and Analysis of Algorithms, Oxford University Press, 2015

# Thank You

# Quick Sort

Prepared By

Dr. V. Elizabeth Jesi

Department of IT

# Introduction

- An efficient sorting algorithm
- Based on partitioning of array of **data** into smaller arrays
- Developed by British computer scientist Tony Hoare in 1959 and published in 1961
- Two or three times faster than other sorting algorithms
- Uses divide and conquer strategy
- Quicksort is a comparison sort
- Sometimes called **partition-exchange sort**

# Three operations performed in quick sort

- 1. Divide:** Select a 'pivot' element from the array and partition the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.
- 2. Conquer :** Recursively sort the two sub arrays
- 3. Combine :** Combine all the sorted elements in a group to form a list of sorted elements.

# How Quick sort works

- Select a PIVOT element from the array
- You can choose any element from the array as the pivot element.
- Here, we have taken the first element of the array as the pivot element.



- The elements smaller than the pivot element are put on the left and the elements greater than the pivot element are put on the right.



- Now sort the left and right arrays recursively.

# Algorithm Quicksort

```
Quicksort(A, L,H)
{
    // Array to be sorted
    // L – lower bound of array
    // H- upper bound of array

    If (L<H)
    {
        P=Partition(A,L,H)
        Quicksort(A,L,P)
        Quicksort(A,P+1,H)
    }
}
```

# Algorithm Partition

```
Partition(A,L,H)
{
    pivot = A[L]
    i=L, j=H+1
    repeat
        repeat i=i+1 until A[i]>=pivot
        repeat j=j-1 until A[j]<=pivot
        swap(a[i],a[j])
    until (i>j)
    swap(A[L],A[j])
}
```

# Method with Example (Partition)

- Numbers to be sorted

34	53	16	71	9	22	42	18
----	----	----	----	---	----	----	----

- Select 34(first element) as the pivot element

34	53	16	71	9	22	42	18
----	----	----	----	---	----	----	----

Find the element > 34 from the left, let its index be i

Find the element<34 from the right, let its index be j

34	53	16	71	9	22	42	18
----	----	----	----	---	----	----	----

34	18	16	71	9	22	42	53
----	----	----	----	---	----	----	----

- Do this process until the index l is greater than index j.

34	18	16	71	9	22	42	53
----	----	----	----	---	----	----	----

34	18	16	22	9	71	42	52
----	----	----	----	---	----	----	----

34	18	16	22	9	71	42	53
----	----	----	----	---	----	----	----

- i becomes > j , so exchange the pivot element and A(j)

9	18	16	22	34	71	42	53
---	----	----	----	----	----	----	----

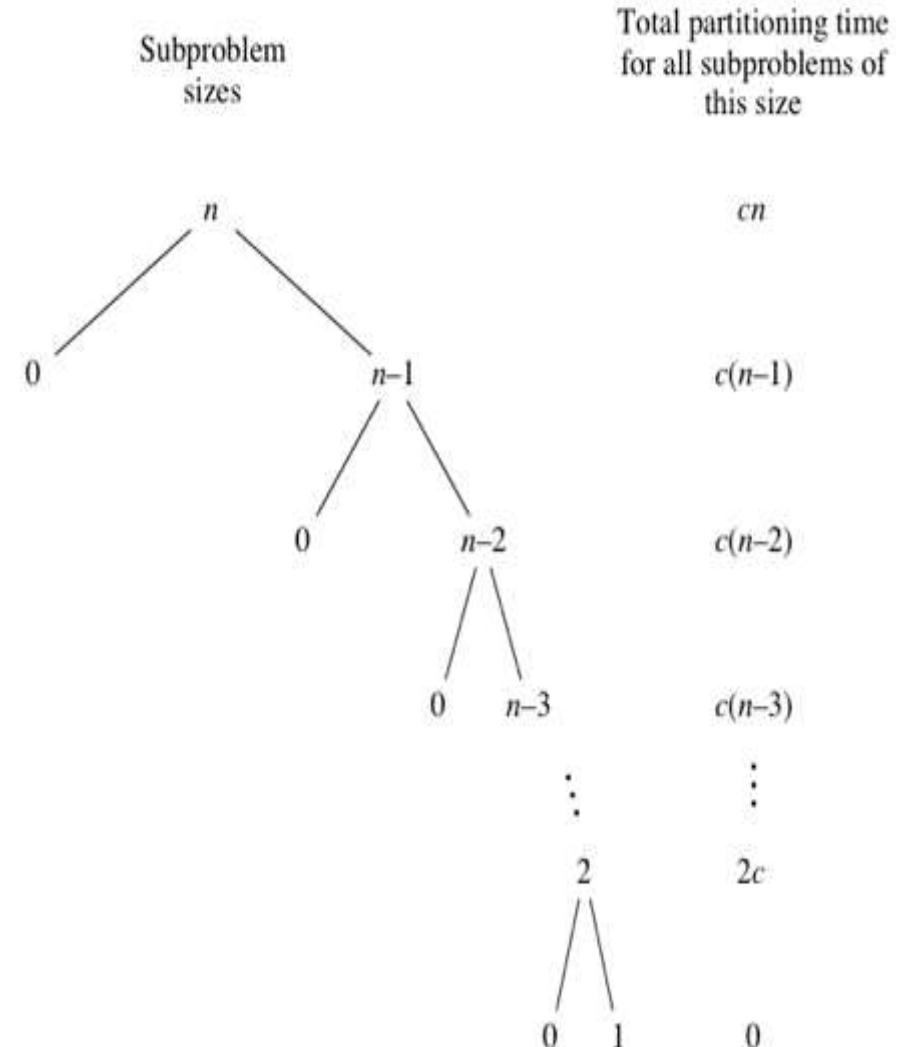
- Note all the elements to the left of pivot is lesser and the elements to the right of pivot is larger than the pivot

# Analysis of Quick Sort Algorithm

- If pivot element is either the largest or the smallest element
- Then one partition will have 0 elements and the other partition will have  $(n-1)$  elements.
- Eg : if pivot element is the largest element in the array. We will have all the elements except the pivot element in the left partition and no elements in the right partition
- The recursive calls will be on arrays of size 0 and  $(n-1)$

# Quick Sort - Worst-case running time

- Let the original call takes  $cn$  time for some constant  $c$ .
- Recursive call on  $n-1$  elements takes  $c(n-1)$ times
- Recursive call on  $n-2$  elements takes time  $c(n-2)$  times, and so on.
- Recursive call on 2 elements will take  $2c$  times
- Finally recursive call on 1 element will take no time.



- Sum up the partitioning times, we get

$$\begin{aligned} & cn + c(n-1) + c(n-2) + \dots + 2c \\ &= c(n + (n-1) + (n-2) + \dots + 2) \\ &= c((n+1)(n/2) - 1) \end{aligned}$$

[Note :  $1+2+\dots+(n-1)+n=(n+1)n/2$

So  $2+\dots+(n-1)+n=((n+1)n/2)-1$ ]

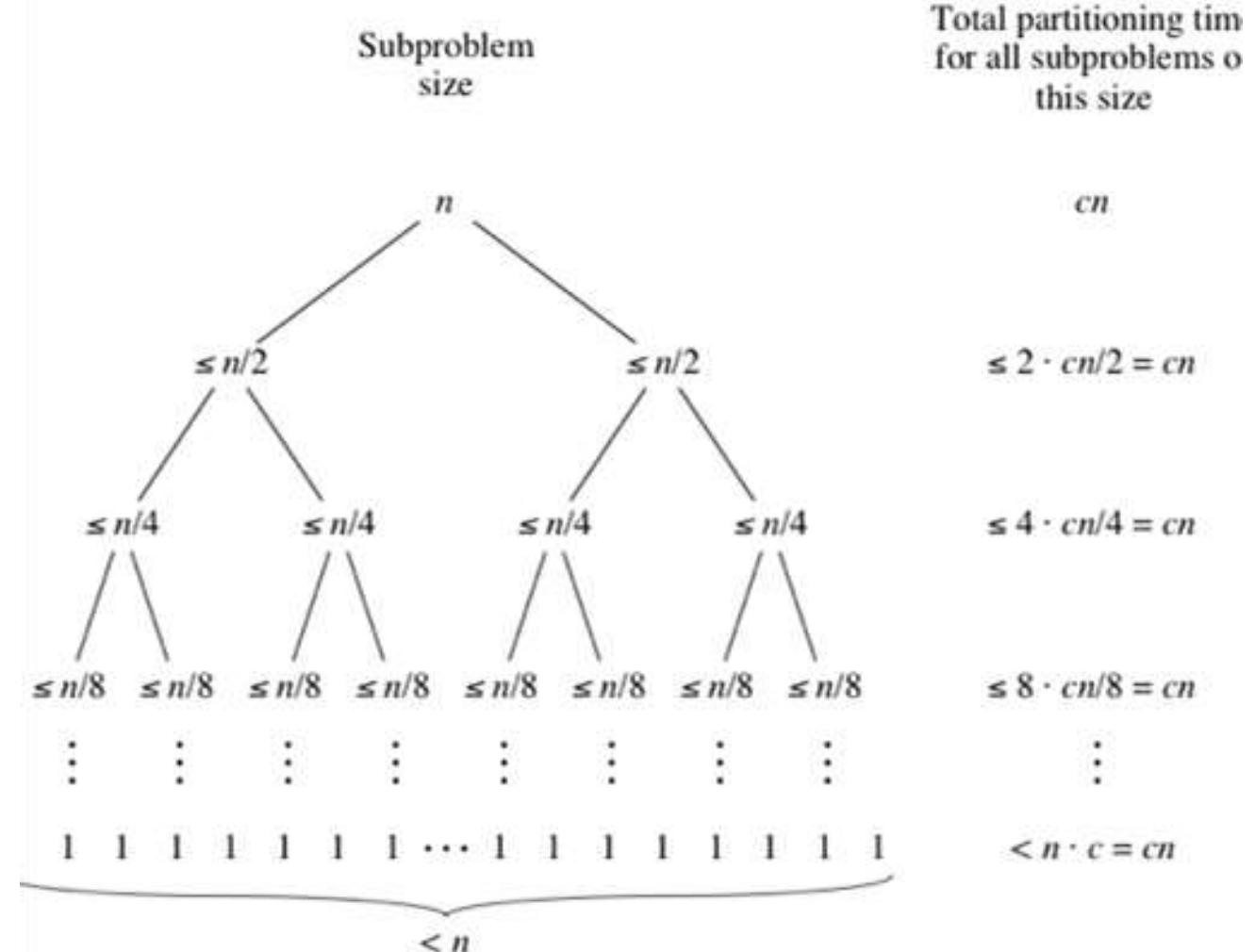
quicksort's worst-case running time is  $\Theta(n^2)$

# Quick Sort : Best-case running time

Occurs when the partitions are evenly partitioned.

quicksort's Best-case running time is

$$\Theta(n \log_2 n)$$



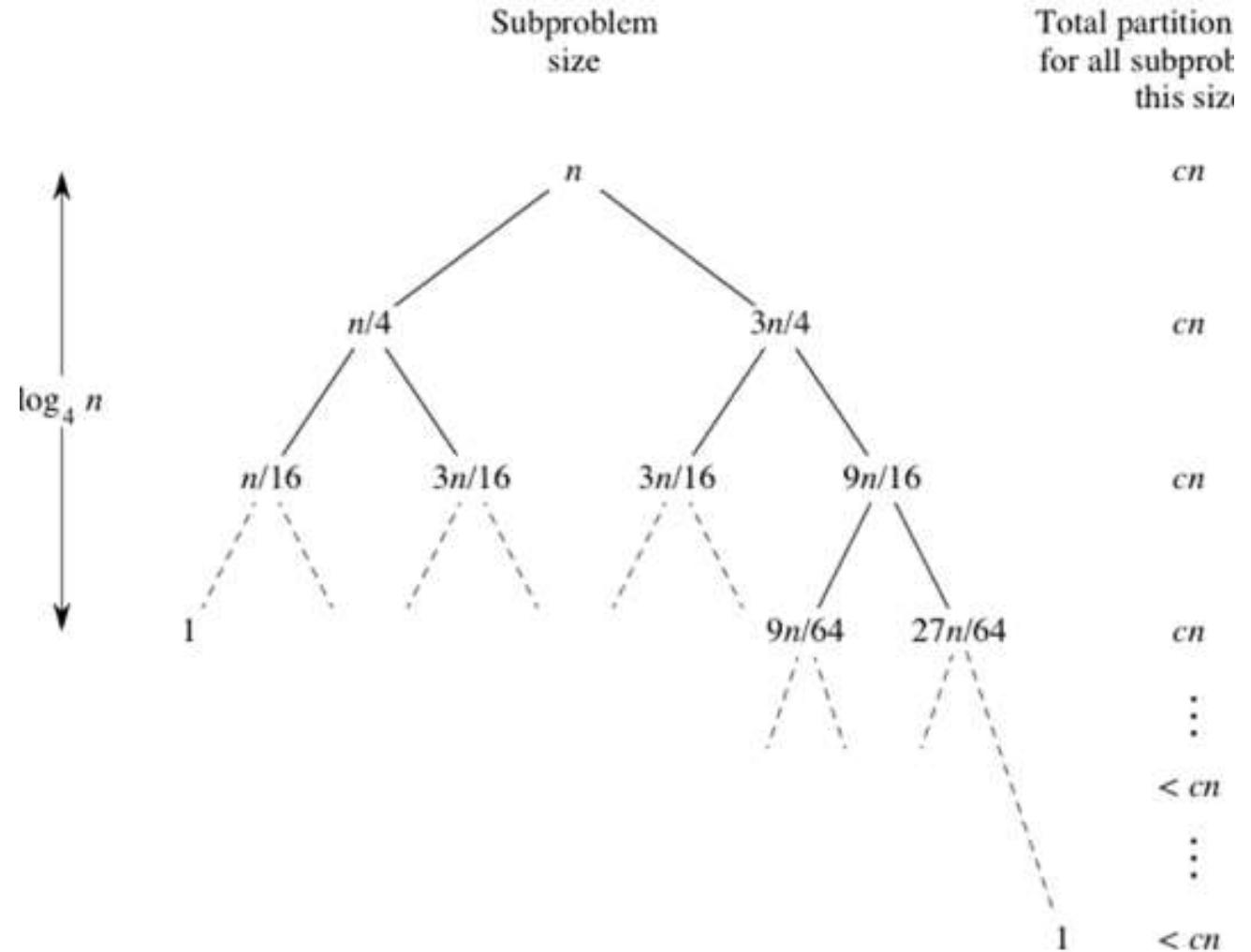
## Quick Sort : Average-case running time

imagine that we don't always get evenly balanced partitions but that we always get at worst a 3-to-1 split

That is, imagine that each time we partition, one side gets  $3n/4$  elements and the other side gets  $n/4$  elements

quicksort's Average-case running time is

$$\Theta(n \log_2 n)$$



# Home assignment

- Sort the following numbers using Quick sort
- 21, 45, 32, 76, 12, 83, 47, 153, 52
- 75, 34, 64, 82, 35, 79, 12, 53, 40, 61

# Master Theorem

# Master theorem

- Master theorem is used to determine running time of algorithms in terms of asymptotic notations.
- The master theorem is a formula for solving recurrences of the form  $T(n) = aT(n/b) + f(n)$ , where  $a \geq 1$  and  $b > 1$  and  $f(n)$  is asymptotically positive. (Asymptotically positive means that the function is positive for all sufficiently large  $n$ .)
- This recurrence describes an algorithm that divides a problem of size  $n$  into  $a$  subproblems, each of size  $n/b$ , and solves them recursively.

# Master Theorem - Proof

The theorem is as follows:

## Master Theorem:

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

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

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

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

# Master Theorem

- The master theorem compares the function  $n^{\log_b a}$  to the function  $f(n)$ . Intuitively, if  $n^{\log_b a}$  is larger (by a polynomial factor), then the solution is  $T(n) = \Theta(n^{\log_b a})$ . If  $f(n)$  is larger (by a polynomial factor), then the solution is  $T(n) = \Theta(f(n))$ . If they are the same size, then we multiply by a logarithmic factor.
- Be warned that these cases are not exhaustive – for example, it is possible for  $f(n)$  to be asymptotically larger than  $n^{\log_b a}$ , but not larger by a polynomial factor (no matter how small the exponent in the polynomial is). For example, this is true when  $f(n) = n^{\log_b a} \log n$ . In this situation, the master theorem would not apply, and you would have to use another method to solve the recurrence.

# Master Theorem

Master's theorem solves recurrence relations of the form-

$$T(n) = a T\left(\frac{n}{b}\right) + \theta(n^k \log^p n)$$

## Master's Theorem

### Case-01:

If  $a > b^k$ , then  $T(n) = \Theta(n^{\log_b a})$

### Case-02:

If  $a = b^k$  and

If  $p < -1$ , then  $T(n) = \Theta(n^{\log_b a})$

If  $p = -1$ , then  $T(n) = \Theta(n^{\log_b a} \cdot \log^2 n)$

If  $p > -1$ , then  $T(n) = \Theta(n^{\log_b a} \cdot \log^{p+1} n)$

### Case-03:

If  $a < b^k$  and

If  $p < 0$ , then  $T(n) = O(n^k)$

If  $p \geq 0$ , then  $T(n) = \Theta(n^k \log^p n)$

# Master Theorem

## Example 1:

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

We compare the given recurrence relation with

$$T(n) = aT(n/b) + \Theta(n^{k \log^p n}).$$

Then, we have-

$$a = 3, b = 2, k = 2, p = 0$$

Now,  $a = 3$  and  $b^k = 2^2 = 4$ .

Clearly,  $a < b^k$ .

So, we follow case- 03.

Since  $p = 0$ , so we have-

$$T(n) = \Theta(n^{k \log^p n})$$

$$T(n) = \Theta(n^2 \log^0 n)$$

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

## Example 2:

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

We compare the given recurrence relation with

$$T(n) = aT(n/b) + \Theta(n^{k \log^p n}).$$

Then, we have-

$$a = 2, b = 2, k = 1, p = 1$$

Now,  $a = 2$  and  $b^k = 2^1 = 2$ .

Clearly,  $a = b^k$ .

So, we follow case-02.

Since  $p = 1$ , so we have-

$$T(n) = \Theta(n^{\log_b a} \cdot \log^{p+1} n)$$

$$T(n) = \Theta(n^{\log_2 2} \cdot \log^{1+1} n)$$

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

# Master Theorem

## Example 3

$$T(n) = 8T(n/4) - n^2\log n$$

- The given recurrence relation does not correspond to the general form of Master's theorem.
- So, it can not be solved using Master's theorem.

## Example 4

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

- We write the given recurrence relation as  $T(n) = 3T(n/3) + n$ .
- This is because in the general form, we have  $\theta$  for function  $f(n)$  which hides constants in it.
- Now, we can easily apply Master's theorem.

We compare the given recurrence relation with  $T(n) = aT(n/b) + \theta(n^k \log^p n)$ .

Then, we have-

$$a = 3, b = 3, k = 1, p = 0$$

$$\text{Now, } a = 3 \text{ and } b^k = 3^1 = 3.$$

Clearly,  $a = b^k$ .

So, we follow case-02.

Since  $p = 0$ , so we have-

$$T(n) = \theta(n^{\log_a b} \cdot \log^{p+1} n)$$

$$T(n) = \theta(n^{\log_3 3} \cdot \log^{0+1} n)$$

$$T(n) = \theta(n^1 \cdot \log^1 n)$$

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

# Master Theorem

## Exercise Problems:

1.  $T(n)=4T(n/2)+n^2 \Rightarrow T(n)=\Theta(n^2\log n)$  (Case 2)
2.  $T(n)=T(n/2)+2n \Rightarrow \Theta(2n)$  (Case 3)
3.  $T(n) = 2nT(n/2) + nn \Rightarrow$  Does not apply (a is not constant)
4.  $T(n)=16T(n/4)+n \Rightarrow T(n)=\Theta(n^2)$  (Case 1)
5.  $T(n)=2T(n/2)+n\log n \Rightarrow T(n)=n\log_2 n$  (Case 2)

# Finding Minimum and Maximum

- **Session Learning Outcome-SLO:** Able to Solve Finding Minimum and Maximum using divide and conquer approaches
- **Motivation of the topic:** The problem is to find the ‘maximum’ and ‘minimum’ items in a set of ‘n’ elements
- Algorithm:
- *Algorithm MaxMin(A, n, max, min)// DIRECT APPROACH*  
  // Set *max* to the maximum and *min* to the minimum of A[1..n]  
  {  
    *max* = *min* = A[1];  
    for( i = 2 to n ) do  
      {  
        if (A[i]>*max*) then *max* = A[i];  
        if (A[i]<*min*) then *min* = A[i];  
      }  
  }
- The above algorithm requires  $2(n-1)$  element comparisons in the best, average and worst cases.

- A *divide-and-conquer* algorithm for this problem would proceed as follows: Let  $P = (n, a[i], \dots, a[j])$  denote an arbitrary instance of the problem. Here  $n$  is the number of elements in the list  $a[i], \dots, a[j]$  and we are interested in finding the maximum and minimum of this list. Let  $\text{small}(P)$  be true when  $n \leq 2$ . In this case, the maximum and minimum are  $a[i]$  if  $n = 1$ . If  $n = 2$ , the problem can be solved by **making one comparison**.
- If the list has more than two elements,  $P$  has to be divided into smaller instances. For example, we might divide  $P$  into the two instances  $P_1 = (n/2, a[1], \dots, a[n/2])$  and  $P_2 = (n - n/2, a[n/2 + 1], \dots, a[n])$ . After having divided  $P$  into two smaller sub problems, we can solve them by recursively invoking the same divide and conquer algorithm.
- Now the question is How can we combine the Solutions for  $P_1$  and  $P_2$  to obtain the solution for  $P$ ?
- If  $\text{MAX}(P)$  and  $\text{MIN}(P)$  are the maximum and minimum of the elements of  $P$ , then  $\text{MAX}(P)$  is the larger of  $\text{MAX}(P_1)$  and  $\text{MAX}(P_2)$  also  $\text{MIN}(P)$  is the smaller of  $\text{MIN}(P_1)$  and  $\text{MIN}(P_2)$ .
- MaxMin is a recursive algorithm that finds the maximum and minimum of the set of elements  $\{a(i), a(i+1), \dots, a(j)\}$ . The situation of set sizes one ( $i=j$ ) and two ( $i=j-1$ ) are handled separately.
- For sets containing more than two elements, the midpoint is determined and two new sub problems are generated. When the maxima and minima of this sub problems are determined, the two maxima are compared and the two minima are compared to achieve the solution for the entire set.

## *Algorithm for maximum and minimum using divide-and-conquer*

```
MaxMin(i, j, max, min)
```

```
// a[1:n] is a global array. Parameters i and j are integers, //  $1 \leq i \leq j \leq n$ . The effect is to set max and min to the largest and // smallest values in a[i:j].
```

```
{
```

```
    if (i=j) then max := min := a[i]; //Small(P)
```

```
    else if (i=j-1) then // Another case of Small(P)
```

```
{
```

```
        if (a[i] < a[j]) then max := a[j]; min := a[i];  
        else max := a[i]; min := a[j];
```

```
}
```

```
else
```

```
{
```

```
    // if P is not small, divide P into sub-problems.
```

```
    // Find where to split the set.
```

```
    mid := ( i + j )/2;
```

```
    // Solve the sub-problems.
```

```
    MaxMin( i, mid, max, min );
```

```
    MaxMin( mid+1, j, max1, min1 );
```

```
    // Combine the solutions.
```

```
    if (max < max1) then max := max1;
```

```
    if (min > min1) then min := min1;
```

```
}
```

# Analysis

what is the number of element comparisons needed for MaxMin? If  $T(n)$  represents this number, then the resulting recurrence relation is

$$T(n) = \begin{cases} 0 & n=1 \\ 1 & n=2 \\ T(n/2) + T(n/2) + 2 & n>2 \end{cases}$$

When  $n$  is a power of two,  $n = 2^k$

-for some positive integer  $k$ , then

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &\quad \cdot \\ &\quad \cdot \\ &= 2^{k-1} T(2) + \sum_{i=1}^{k-1} 2^k \\ &= 2^{k-1} + 2^k - 2 \\ &= 3n/2 - 2 = O(n) \end{aligned}$$

Note that  $3n/2 - 2$  is the best, average, worst case number of comparison when  $n$  is a power of two.

Comparisons with Straight Forward Method:

Compared with the  $2n - 2$  comparisons for the Straight Forward method, this is a saving of 25% in comparisons. It can be shown that no algorithm based on comparisons uses less than  $3n/2 - 2$  comparisons.

- **Activity:**
- The minimum number of comparisons required to find the minimum and the maximum of 100 numbers is \_\_\_\_\_.

# Closest pair problem

- Closest-pair problem calls for finding the two closest points in a set of  $n$  Points.
- Cluster Analysis in statistics deals with closest pair problem
- Euclidian distance is used to calculate the closest pair problem

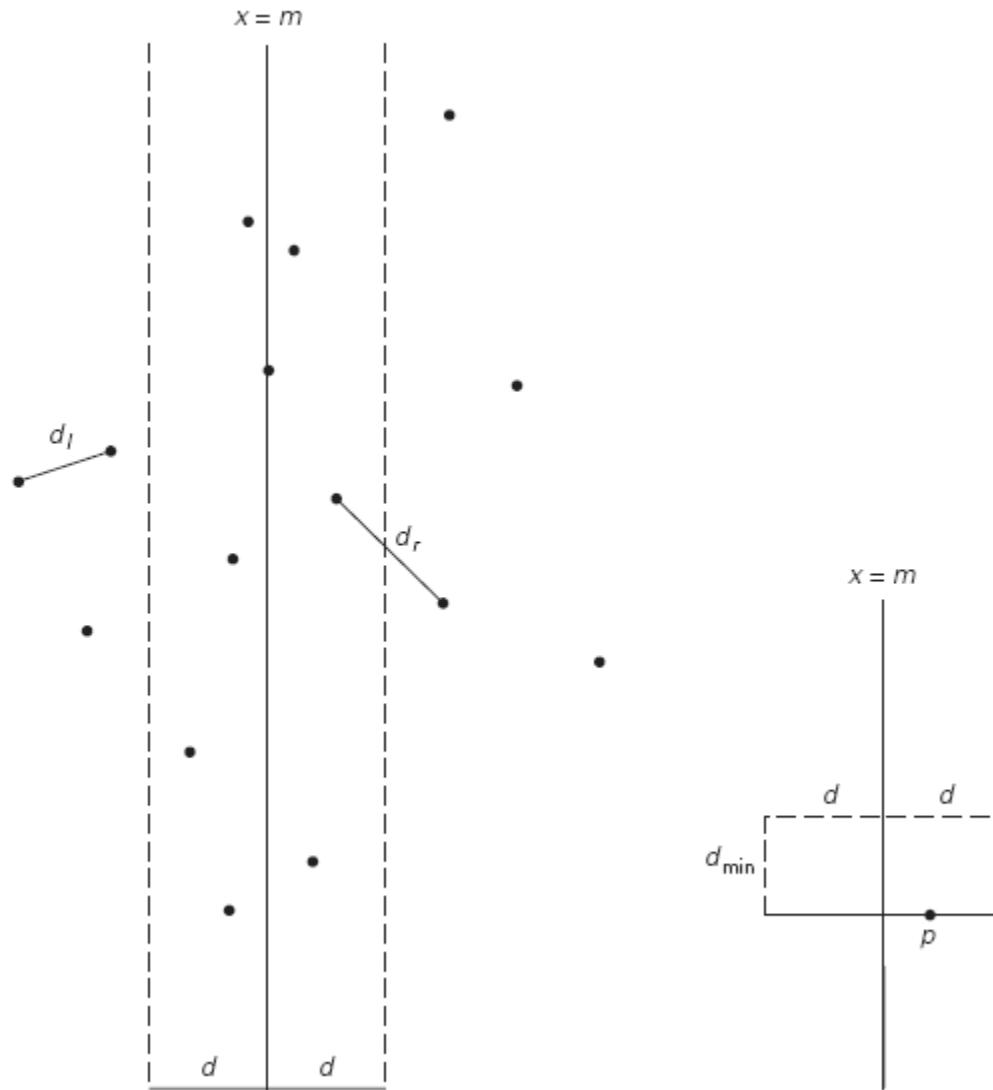
$$d_{xi,xj} = \sqrt{(xi - xj)^2 + (yi - yj)^2}$$

- Let  $P$  be a set of  $n > 1$  points in the Cartesian plane , we assume points are distinct
- Assume that the points are ordered in nondecreasing order of their  $x$  coordinate and  $y$  coordinate

# Closest pair problem

- Closest-pair problem calls for finding the two closest points in a set of  $n$  Points.
- Divide the points into equal halves
- After dividing the points into two equal halves find the closest point in each equal halves
- Find the distances among each closest point in each halves using Euclidean distance
- Store the distances calculated in a array
- Find the smallest distance in the array an the return the point for which we received minimum distance is closest pair

# Closest pair problem -Example



# Closest pair problem -Motivation

- When number of points or number of items are available we have to find the closest items or points available
- We find the distance between each pair of distinct points and find a pair with the smallest distance
- A naive algorithm of finding distances between all pairs of points in a space of dimension d and selecting the minimum requires  $O(n^2)$  time
- But while we go for calculating distance between two points using Euclidean distance , The total time taken is  $O(n \log n)$
- So Euclidean distance is used in closest pair using divide and conquer mechanism
- Time taken for computation using Brute Force approach is  $O(n^2)$

# Closest pair problem -Algorithm

```

if  $n \leq 3$ 
    return the minimal distance found by the brute-force algorithm
else
    copy the first  $\lfloor n/2 \rfloor$  points of  $P$  to array  $Pl$ 
    copy the same  $\lfloor n/2 \rfloor$  points from  $Q$  to array  $Ql$ 
    copy the remaining  $\lfloor n/2 \rfloor$  points of  $P$  to array  $Pr$ 
    copy the same  $\lfloor n/2 \rfloor$  points from  $Q$  to array  $Qr$ 
     $dl \leftarrow EfficientClosestPair(Pl, Ql)$ 
     $dr \leftarrow EfficientClosestPair(Pr, Qr)$ 
     $d \leftarrow \min\{dl, dr\}$ 
     $m \leftarrow P[\lfloor n/2 \rfloor - 1].x$ 
    copy all the points of  $Q$  for which  $|x - m| < d$  into array  $S[0..num - 1]$ 
     $dminsq \leftarrow d^2$ 
for  $i \leftarrow 0$  to  $num - 2$  do
     $k \leftarrow i + 1$ 
    while  $k \leq num - 1$  and  $(S[k].y - S[i].y)^2 < dminsq$ 
         $dminsq \leftarrow \min((S[k].x - S[i].x)^2 + (S[k].y - S[i].y)^2, dminsq)$ 
     $k \leftarrow k + 1$ 
return  $\sqrt{dminsq}$ 

```

# Closest pair problem -Analysis

- Running time of the algorithm (without sorting) is:

$$T(n) = 2T(n/2) + M(n), \text{ where } M(n) \in \Theta(n)$$

- By the Master Theorem (with  $a = 2, b = 2, d = 1$ )

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

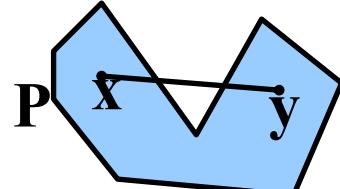
- So the total time is  $\Theta(n \log n)$ .

# Real time applications

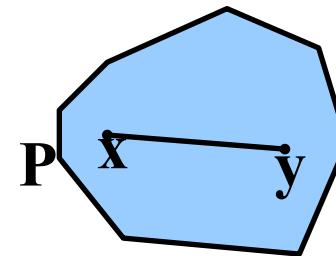
- Air/land/water traffic control system
- Collision avoidance

# Convex vs. Concave

- A polygon P is convex if for every pair of points x and y in P, the line xy is also in P; otherwise, it is called concave.



concave



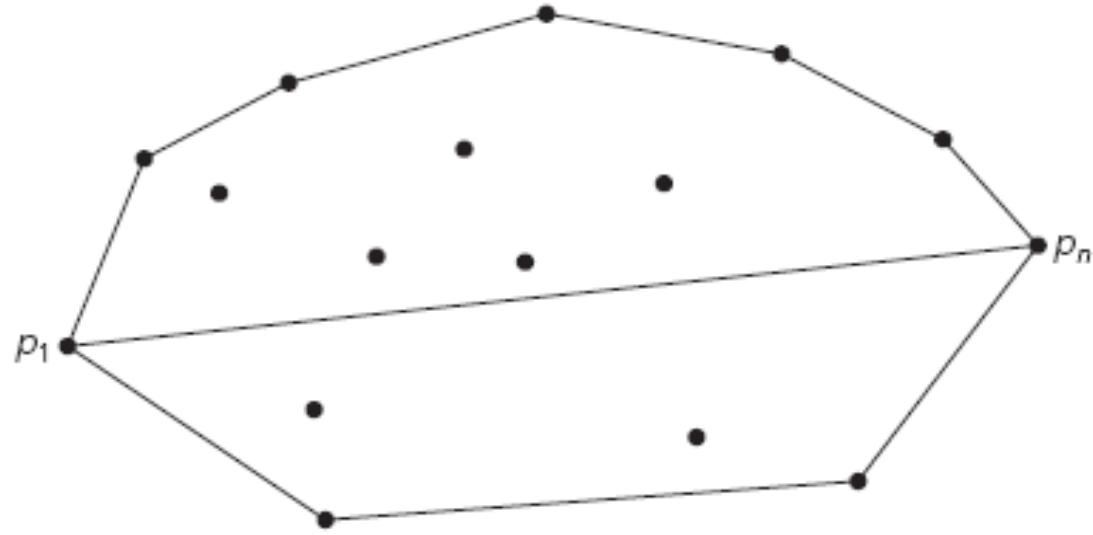
convex

# Convex hull Problem

- Convex hull or convex envelope or convex closure of a shape is the smallest convex set that contains it
- Convex Hull is the line completely enclosing a set of points in a plane so that there are no concavities in the line

# Convex hull Problem

- Let  $S$  be a set of  $n > 1$  points  $p_1(x_1, y_1), \dots, p_n(x_n, y_n)$  in the Cartesian plane



- Splitted in upper and lower half

# Convex hull Problem -Algorithm

```
vector<pair<int, int>> divide(vector<pair<int, int>> a)
{
    if (a.size() <= 5)
        return bruteHull(a);
    vector<pair<int, int>> left, right;
    for (int i=0; i<a.size()/2; i++)
        left.push_back(a[i]);
    for (int i=a.size()/2; i<a.size(); i++)
        right.push_back(a[i]);
    vector<pair<int, int>> left_hull = divide(left);
    vector<pair<int, int>> right_hull = divide(right);
    return merger(left_hull, right_hull);
}
```

# Convex Hull - Time Complexity



- Time complexity If points are not initially sorted  $O(n \log n)$
- Time efficiency:  $T(n) = T(n-1) + O(n)$   
 $T(n) = T(x) + T(y) + T(z) + T(v) + O(n)$ , where  $x + y + z + v \leq n$ .  
worst case:  $\Theta(n^2)$   
average case:  $\Theta(n)$

# Real time applications

- Collision avoidance

# Assignment

- Implementation of Closest pair problem
- Implementation of Convex hull problem

# **Strassen's Matrix Multiplication**

Dr. Anand M

Id: 102763

Assistant Professor

Department of Computer Science and Engineering  
SRM Institute of Science and Technology

# Basic Matrix Multiplication

Suppose we want to multiply two matrices of size  $N \times N$ : for example  $A \times B = C$ .

$$\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

$$C_{11} = a_{11}b_{11} + a_{12}b_{21}$$

$$C_{12} = a_{11}b_{12} + a_{12}b_{22}$$

$$C_{21} = a_{21}b_{11} + a_{22}b_{21}$$

$$C_{22} = a_{21}b_{12} + a_{22}b_{22}$$

2x2 matrix multiplication can be accomplished in 8 multiplication. ( $2^{\log_2 8} = 2^3$ )

# Basic Matrix Multiplication

```
void matrix_mult (){  
    for (i = 1; i <= N; i++) {  
        for (j = 1; j <= N; j++) {  
            compute Ci,j;  
        }  
    }  
}
```

algorithm

Time analysis

$$C_{i,j} = \sum_{k=1}^N a_{i,k} b_{k,j}$$

$$\text{Thus } T(N) = \sum_{i=1}^N \sum_{j=1}^N \sum_{k=1}^N c = cN^3 = O(N^3)$$

# Strassens's Matrix Multiplication

- Strassen showed that 2x2 matrix multiplication can be accomplished in 7 multiplication and 18 additions or subtractions. .( $2^{\log_2 7} = 2^{2.807}$ )
- This reduce can be done by Divide and Conquer Approach.

# Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data  $S$  in two or more disjoint subsets  $S_1, S_2, \dots$
  - Recur: solve the subproblems recursively
  - Conquer: combine the solutions for  $S_1, S_2, \dots$ , into a solution for  $S$
- The base case for the recursion are subproblems of constant size
- Analysis can be done using recurrence equations

# Divide and Conquer Matrix Multiply

$$A \times B = R$$

$$\begin{array}{|c|c|} \hline A_0 & A_1 \\ \hline A_2 & A_3 \\ \hline \end{array} \times \begin{array}{|c|c|} \hline B_0 & B_1 \\ \hline B_2 & B_3 \\ \hline \end{array} = \begin{array}{|c|c|} \hline A_0 \times B_0 + A_1 \times B_2 & A_0 \times B_1 + A_1 \times B_3 \\ \hline A_2 \times B_0 + A_3 \times B_2 & A_2 \times B_1 + A_3 \times B_3 \\ \hline \end{array}$$

- Divide matrices into sub-matrices:  $A_0, A_1, A_2$  etc
- Use blocked matrix multiply equations
- Recursively multiply sub-matrices

# Divide and Conquer Matrix Multiply

$$A \times B = R$$

$$\boxed{a_0} \times \boxed{b_0} = \boxed{a_0 \times b_0}$$

- Terminate recursion with a simple base case

# Strassens's Matrix Multiplication

$$\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

$$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$P_2 = (A_{21} + A_{22}) * B_{11}$$

$$P_3 = A_{11} * (B_{12} - B_{22})$$

$$P_4 = A_{22} * (B_{21} - B_{11})$$

$$P_5 = (A_{11} + A_{12}) * B_{22}$$

$$P_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$$

$$P_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$$

$$C_{11} = P_1 + P_4 - P_5 + P_7$$

$$C_{12} = P_3 + P_5$$

$$C_{21} = P_2 + P_4$$

$$C_{22} = P_1 + P_3 - P_2 + P_6$$

# Comparison

$$\begin{aligned} C_{11} &= P_1 + P_4 - P_5 + P_7 \\ &= (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22} * (B_{21} - B_{11}) - (A_{11} + A_{12}) * B_{22} + \\ &\quad (A_{12} - A_{22}) * (B_{21} + B_{22}) \\ &= A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} + A_{22}B_{21} - A_{22}B_{11} - \\ &\quad A_{11}B_{22} - A_{12}B_{22} + A_{12}B_{21} + A_{12}B_{22} - A_{22}B_{21} - A_{22}B_{22} \\ &= A_{11}B_{11} + A_{12}B_{21} \end{aligned}$$

# Strassen Algorithm

```
void matmul(int *A, int *B, int *R, int n) {  
    if (n == 1) {  
        (*R) += (*A) * (*B);  
    } else {  
        matmul(A, B, R, n/4);  
        matmul(A, B+(n/4), R+(n/4), n/4);  
        matmul(A+2*(n/4), B, R+2*(n/4), n/4);  
        matmul(A+2*(n/4), B+(n/4), R+3*(n/4), n/4);  
        matmul(A+(n/4), B+2*(n/4), R, n/4);  
        matmul(A+(n/4), B+3*(n/4), R+(n/4), n/4);  
        matmul(A+3*(n/4), B+2*(n/4), R+2*(n/4), n/4);  
        matmul(A+3*(n/4), B+3*(n/4), R+3*(n/4), n/4);  
    }  
}
```

Divide matrices in  
sub-matrices and  
recursively multiply  
sub-matrices

# Time Analysis

$$T(1) = 1 \quad (\text{assume } N = 2^k)$$

$$T(N) = 7T(N/2)$$

$$T(N) = 7^k T(N/2^k) = 7^k$$

$$T(N) = 7^{\log N} = N^{\log 7} = N^{2.81}$$

# Assignment Work

1. Verify the formulas of Strassen's algorithm for multiplying 2 matrices.
2. Apply Strassen's algorithm to compute.

$$\begin{bmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

# Largest Subarray Problem

Dr. Anand M

Id: 102763

Assistant Professor

Department of Computer Science and Engineering  
SRM Institute of Science and Technology

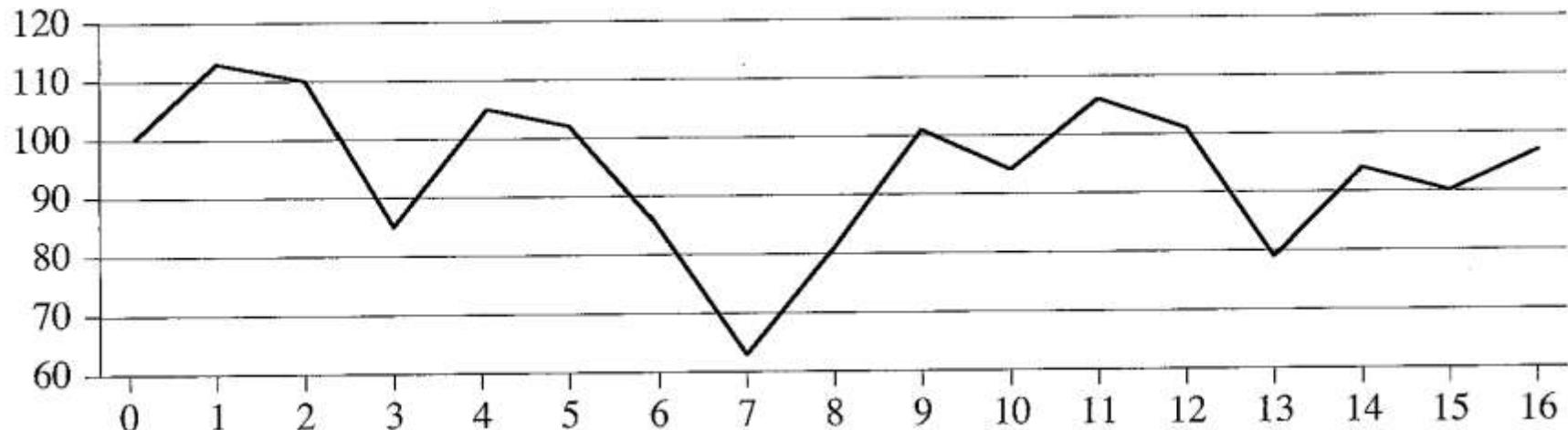
# Largest Subarray Problem

**Problem:** given an array of n numbers, find the (a) contiguous subarray whose sum has the largest value.

**Application:** an unrealistic stock market game, in which you decide when to buy and see a stock, with full knowledge of the past and future. ***The restriction*** is that you can perform just one ***buy*** followed by a ***sell***. The buy and sell both occur right after the close of the market.

**The interpretation of the numbers:** each number represents the stock value at closing on any particular day.

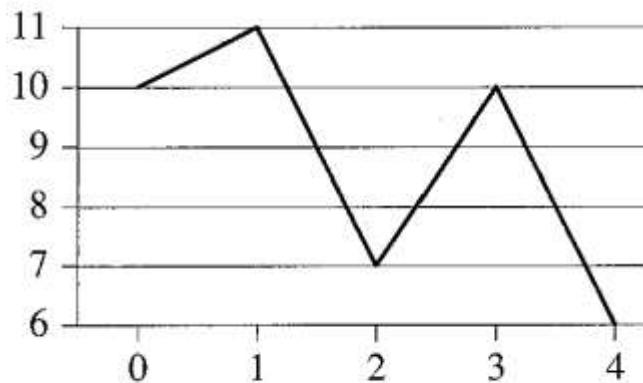
# Largest Subarray Problem



**Figure 4.1** Information about the price of stock in the Volatile Chemical Corporation after the close of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical axis shows the price. The bottom row of the table gives the change in price from the previous day.

# Largest Subarray Problem

**Another Example:** buying low and selling high, even with perfect knowledge, is not trivial:



Day	0	1	2	3	4
Price	10	11	7	10	6
Change	1	-4	3	-4	

**Figure 4.2** An example showing that the maximum profit does not always start at the lowest price or end at the highest price. Again, the horizontal axis indicates the day, and the vertical axis shows the price. Here, the maximum profit of \$3 per share would be earned by buying after day 2 and selling after day 3. The price of \$7 after day 2 is not the lowest price overall, and the price of \$10 after day 3 is not the highest price overall.

# A brute-force solution

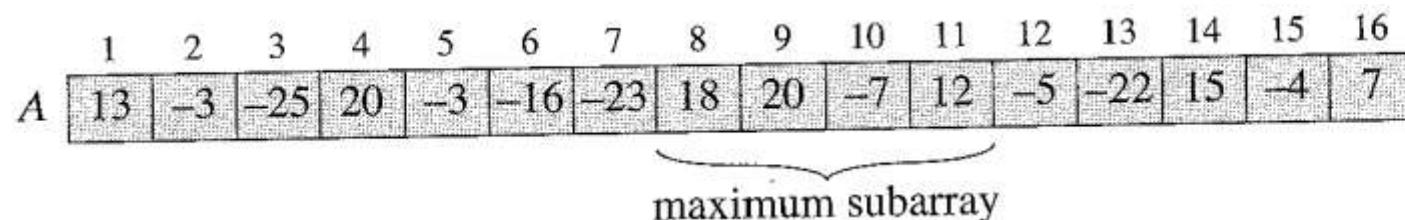
- $O(n^2)$  Solution
- Considering  $C_n^2$  pairs
- Not a pleasant prospect if we are rummaging through long time-series (Who told you it was easy to get rich???), even if you are allowed to post-date your stock options...

# A Better Solution: Max Subarray

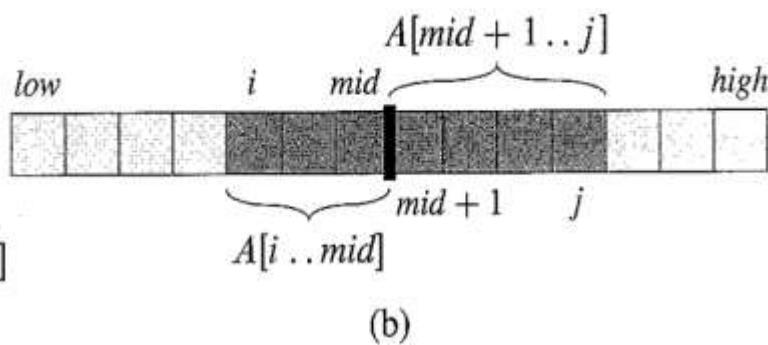
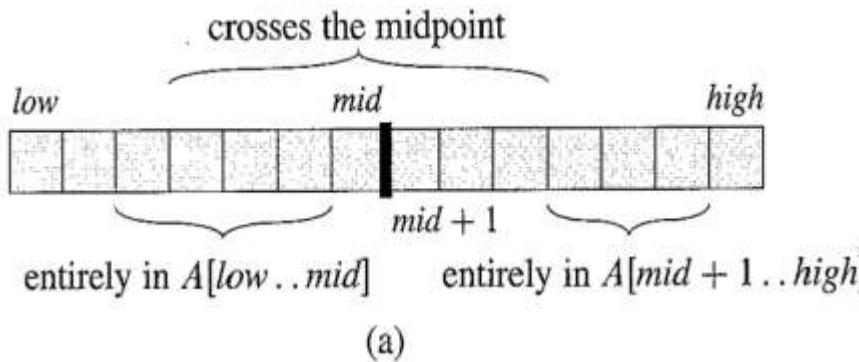
**Transformation:** Instead of the daily price, let us consider the daily change:  $A[i]$  is the difference between the closing value on day  $i$  and that on day  $i-1$ . The problem becomes that of finding a contiguous subarray the sum of whose values is maximum.

- On a first look this seems even worse: roughly the same number of intervals (one fewer, to be precise), and the requirement to add the values in the subarray rather than just computing a difference:  $\Omega(n^3)$ ?

# Max Subarray



**Figure 4.3** The change in stock prices as a maximum-subarray problem. Here, the subarray  $A[8..11]$ , with sum 43, has the greatest sum of any contiguous subarray of array  $A$ .



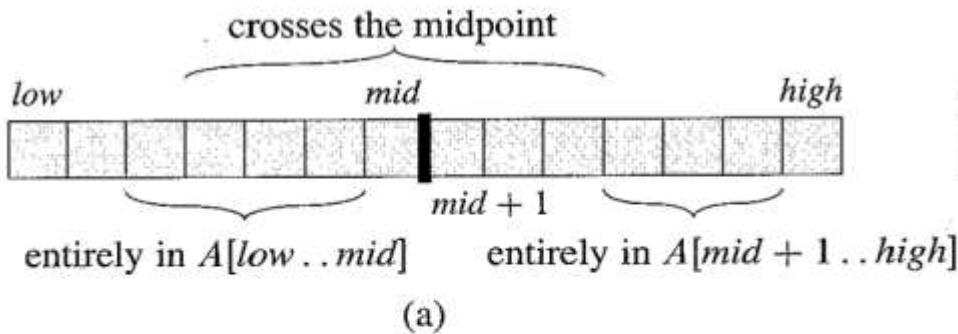
**Figure 4.4** (a) Possible locations of subarrays of  $A[low..high]$ : entirely in  $A[low..mid]$ , entirely in  $A[mid + 1..high]$ , or crossing the midpoint  $mid$ . (b) Any subarray of  $A[low..high]$  crossing the midpoint comprises two subarrays  $A[i..mid]$  and  $A[mid + 1..j]$ , where  $low \leq i \leq mid$  and  $mid < j \leq high$ .

# Max Subarray

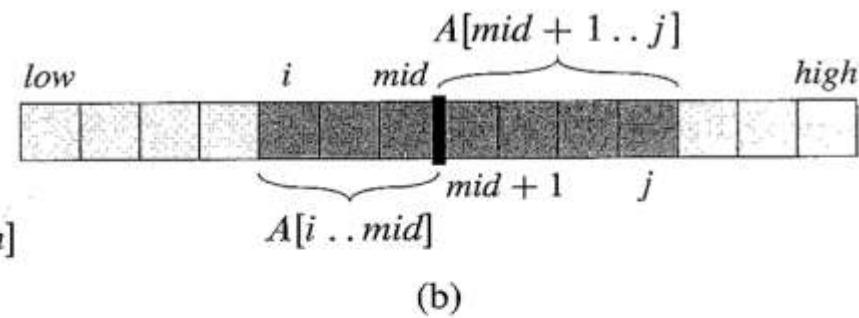
How do we divide?

We observe that a maximum contiguous subarray  $A[i\dots j]$  must be located as follows:

1. It lies entirely in the left half of the original array:  $[low\dots mid]$ ;
2. It lies entirely in the right half of the original array:  $[mid+1\dots high]$ ;
3. It straddles the midpoint of the original array:  $i \leq mid < j$ .



(a)



(b)

**Figure 4.4** (a) Possible locations of subarrays of  $A[low\dots high]$ : entirely in  $A[low\dots mid]$ , entirely in  $A[mid + 1\dots high]$ , or crossing the midpoint  $mid$ . (b) Any subarray of  $A[low\dots high]$  crossing the midpoint comprises two subarrays  $A[i\dots mid]$  and  $A[mid + 1\dots j]$ , where  $low \leq i \leq mid$  and  $mid < j \leq high$ .

# Max Subarray: Divide & Conquer

- The “left” and “right” subproblems are smaller versions of the original problem, so they are part of the standard Divide & Conquer recursion.
- The “middle” subproblem is not, so we will need to count its cost as part of the “combine” (or “divide”) cost.
  - The crucial observation (and it may not be entirely obvious) is that **we can find the maximum crossing subarray in time linear in the length of the  $A[low...high]$  subarray.**

**How?**  $A[i,...,j]$  must be made up of  $A[i...mid]$  and  $A[m+1...j]$  – so we find the largest  $A[i...mid]$  and the largest  $A[m+1...j]$  and combine them.

# The middle subproblem

FIND-MAX-CROSSING-SUBARRAY( $A, low, mid, high$ )

```
1  left-sum =  $-\infty$ 
2  sum = 0
3  for  $i = mid$  downto  $low$ 
4      sum = sum +  $A[i]$ 
5      if sum > left-sum
6          left-sum = sum
7          max-left =  $i$ 
8  right-sum =  $-\infty$ 
9  sum = 0
10 for  $j = mid + 1$  to  $high$ 
11     sum = sum +  $A[j]$ 
12     if sum > right-sum
13         right-sum = sum
14         max-right =  $j$ 
15 return (max-left, max-right, left-sum + right-sum)
```

# Algorithms: Max Subarray

FIND-MAXIMUM-SUBARRAY( $A, low, high$ )

```
1  if  $high == low$ 
2      return ( $low, high, A[low]$ )          // base case: only one element
3  else  $mid = \lfloor (low + high)/2 \rfloor$ 
4      ( $left-low, left-high, left-sum$ ) =
            FIND-MAXIMUM-SUBARRAY( $A, low, mid$ )
5      ( $right-low, right-high, right-sum$ ) =
            FIND-MAXIMUM-SUBARRAY( $A, mid + 1, high$ )
6      ( $cross-low, cross-high, cross-sum$ ) =
            FIND-MAX-CROSSING-SUBARRAY( $A, low, mid, high$ )
7      if  $left-sum \geq right-sum$  and  $left-sum \geq cross-sum$ 
8          return ( $left-low, left-high, left-sum$ )
9      elseif  $right-sum \geq left-sum$  and  $right-sum \geq cross-sum$ 
10         return ( $right-low, right-high, right-sum$ )
11     else return ( $cross-low, cross-high, cross-sum$ )
```

# Max Subarray: Algorithm Efficiency

We finally have:

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

The recurrence has the same form as that for MERGESORT, and thus we should expect it to have the same solution  $T(n) = \Theta(n \lg n)$ .

This algorithm is clearly substantially faster than any of the brute-force methods. It required some cleverness, and the programming is a little more complicated – but the payoff is large.

# Assignment Work

1. Write a pseudocode for a divide-and-conquer algorithm for finding the position of the largest element in an array of  $n$  numbers.
2. Set up and solve a recurrence relation for the number of key comparisons made by your algorithm.
3. How does this algorithm compare with the brute-force algorithm for this problem?

# Time complexity analysis of Largest sub-array sum

Dr. Anand M

Id: 102763

Assistant Professor

Department of Computer Science and Engineering  
SRM Institute of Science and Technology

# Max Subarray: Algorithm Efficiency

We finally have:

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

The recurrence has the same form as that for MERGESORT, and thus we should expect it to have the same solution  $T(n) = \Theta(n \lg n)$ .

This algorithm is clearly substantially faster than any of the brute-force methods. It required some cleverness, and the programming is a little more complicated – but the payoff is large.

# Time complexity analysis

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

$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\ &\leq 2[2T\left(\frac{n}{4}\right) + c\frac{n}{2}] + cn = 4T\left(\frac{n}{4}\right) + 2cn \quad \text{1<sup>st</sup> expansion} \\ &\leq 4[2T\left(\frac{n}{8}\right) + c\frac{n}{4}] + 2cn = 8T\left(\frac{n}{8}\right) + 3cn \quad \text{2<sup>nd</sup> expansion} \\ &\vdots \\ &\leq 2^k T\left(\frac{n}{2^k}\right) + kcn \quad \text{k<sup>th</sup> expansion} \end{aligned} \quad \begin{aligned} T(n) &\leq nT(1) + cn \log_2 n \\ &= O(n) + O(n \log n) \\ \text{The expansion stops when } 2^k &= n \quad = O(n \log n) \end{aligned}$$

# Time complexity analysis

- Theorem

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n) & \text{if } n \geq 2 \end{cases} \rightarrow T(n) = O(n \log n)$$

- Proof

- There exists positive constant  $a, b$  s.t.  $T(n) \leq \begin{cases} a & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + b \cdot n & \text{if } n \geq 2 \end{cases}$
- Use induction to prove  $T(n) \leq 2b \cdot n \log_2 n + a \cdot n$ 
  - $n = 1$ , trivial
  - $n > 1, \lceil \frac{n}{2} \rceil \leq \frac{n}{\sqrt{2}}$

$$T(n) \leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + b \cdot n$$

**Inductive hypothesis**

$$\begin{aligned} &\leq 2b \cdot (\lceil n/2 \rceil \log_2 \lceil n/2 \rceil) + a \cdot \lceil n/2 \rceil + 2b \cdot (\lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor) + a \cdot \lfloor n/2 \rfloor + b \cdot n \\ &\leq 2b \cdot (\lceil n/2 \rceil \log_2 \frac{n}{\sqrt{2}}) + a \cdot \lceil n/2 \rceil + 2b \cdot (\lfloor n/2 \rfloor \log_2 \frac{n}{\sqrt{2}}) + a \cdot \lfloor n/2 \rfloor + b \cdot n \\ &= 2b \cdot n(\log n - \log_2 \sqrt{2}) + a \cdot n + b \cdot n = 2b \cdot n \log_2 n + a \cdot n \end{aligned}$$

# UNIT 3

# Greedy & Dynamic

# Programming

## Session Outcome:

Idea to apply Both Dynamic Programming and Greedy algorithm to **solve optimization problems.**

# Greedy Algorithm

# Greedy Method

- "Greedy Method finds out of many options, but you have to choose the best option."
- Greedy Algorithm solves problems by making the best choice that seems best at the particular moment. Many optimization problems can be determined using a greedy algorithm. Some issues have no efficient solution, but a greedy algorithm may provide a solution that is close to optimal. A greedy algorithm works if a problem exhibits the following two properties:
- **Greedy Choice Property:** A globally optimal solution can be reached at by creating a locally optimal solution. In other words, an optimal solution can be obtained by creating "greedy" choices.
- **Optimal substructure:** Optimal solutions contain optimal subsolutions. In other words, answers to subproblems of an optimal solution are optimal.

# Greedy Method (Cont.)

Steps for achieving a Greedy Algorithm are:

- **Feasible:** Here we check whether it satisfies all possible constraints or not, to obtain at least one solution to our problems.
- **Local Optimal Choice:** In this, the choice should be the optimum which is selected from the currently available
- **Unalterable:** Once the decision is made, at any subsequent step that option is not altered.

# Greedy Method (Cont.)

Example:

- Machine scheduling
- Fractional Knapsack Problem
- Minimum Spanning Tree
- Huffman Code
- Job Sequencing
- Activity Selection Problem

# Dynamic Programming

# Dynamic Programming

- Dynamic Programming is the most powerful design technique for solving optimization problems.
- Divide & Conquer algorithm partition the problem into disjoint sub-problems solve the sub-problems recursively and then combine their solution to solve the original problems.
- Dynamic Programming is used when the subproblems are not independent, e.g. when they share the same subproblems. In this case, divide and conquer may do more work than necessary, because it solves the same sub problem multiple times.

# Dynamic Programming (Cont.)

- Dynamic Programming solves each subproblems just once and stores the result in a table so that it can be repeatedly retrieved if needed again.
- 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 paradigm of algorithm design in which an optimization problem is solved by a combination of achieving subproblem solutions and appealing to the "**principle of optimality**".

# Characteristics of Dynamic Programming

Dynamic Programming works when a problem has the following features:-

- **Optimal Substructure:** If an optimal solution contains optimal sub solutions then a problem exhibits optimal substructure.
- **Overlapping sub-problems:** When a recursive algorithm would visit the same sub-problems repeatedly, then a problem has overlapping sub-problems.

# Characteristics of Dynamic Programming

- If a problem has optimal substructure, then we can recursively define an optimal solution. If a problem has overlapping sub-problems, then we can improve on a recursive implementation by computing each sub-problem only once.
- If a problem doesn't have optimal substructure, there is no basis for defining a recursive algorithm to find the optimal solutions. If a problem doesn't have overlapping sub problems, we don't have anything to gain by using dynamic programming.
- If the space of sub-problems is enough (i.e. polynomial in the size of the input), dynamic programming can be much more efficient than recursion.

# Elements of Dynamic Programming

- **Substructure:** Decompose the given problem into smaller sub-problems. Express the solution of the original problem in terms of the solution for smaller problems.
- **Table Structure:** After solving the sub-problems, store the results to the sub problems in a table. This is done because sub-problem solutions are reused many times, and we do not want to repeatedly solve the same problem over and over again.
- **Bottom-up Computation:** Using table, combine the solution of smaller sub-problems to solve larger sub-problems and eventually arrives at a solution to complete problem.

**Note: Bottom-up means:-**

- Start with smallest sub-problems.
- Combining their solutions obtain the solution to sub-problems of increasing size.
- Until solving at the solution of the original problem.

# Components of Dynamic Programming

1. **Stages:** The problem can be divided into several sub-problems, which are called stages. A stage is a small portion of a given problem. For example, in the shortest path problem, they were defined by the structure of the graph.
2. **States:** Each stage has several states associated with it. The states for the shortest path problem was the node reached.
3. **Decision:** At each stage, there can be multiple choices out of which one of the best decisions should be taken. The decision taken at every stage should be optimal; this is called a stage decision.
4. **Optimal policy:** It is a rule which determines the decision at each stage; a policy is called an optimal policy if it is globally optimal. This is known as Bellman principle of optimality.
5. Given the current state, the optimal choices for each of the remaining states does not depend on the previous states or decisions. In the shortest path problem, it was not necessary to know how we got a node only that we did.
6. There exist a recursive relationship that identify the optimal decisions for stage  $j$ , given that stage  $j+1$ , has already been solved.
7. The final stage must be solved by itself.

# Development of Dynamic Programming Algorithm

It can be broken into four steps:

1. Characterize the structure of an optimal solution.
2. Recursively defined the value of the optimal solution. Like Divide and Conquer, divide the problem into two or more optimal parts recursively. This helps to determine what the solution will look like.
3. Compute the value of the optimal solution from the bottom up (starting with the smallest sub-problems)
4. Construct the optimal solution for the entire problem form the computed values of smaller sub-problems.

# Applications of Dynamic Programming

1. 0/1 knapsack problem
2. Mathematical optimization problem
3. All pair Shortest path problem
4. Reliability design problem
5. Longest common subsequence (LCS)
6. Flight control and robotics control
7. Time-sharing: It schedules the job to maximize CPU usage

# Divide & Conquer Method vs Dynamic Programming

Divide & Conquer Method	Dynamic Programming
<p><b>1.</b>It deals (involves) three steps at each level of recursion:  <b>Divide</b> the problem into a number of subproblems.  <b>Conquer</b> the subproblems by solving them recursively.  <b>Combine</b> the solution to the subproblems into the solution for original subproblems.</p>	<ul style="list-style-type: none"> <li><b>1.</b>It involves the sequence of four steps:Characterize the structure of optimal solutions.</li> <li>•Recursively defines the values of optimal solutions.</li> <li>•Compute the value of optimal solutions in a Bottom-up minimum.</li> <li>•Construct an Optimal Solution from computed information.</li> </ul>
<b>2.</b> It is Recursive.	<b>2.</b> It is non Recursive.
<b>3.</b> It does more work on subproblems and hence has more time consumption.	<b>3.</b> It solves subproblems only once and then stores in the table.
<b>4.</b> It is a top-down approach.	<b>4.</b> It is a Bottom-up approach.
<b>5.</b> In this subproblems are independent of each other.	<b>5.</b> In this subproblems are interdependent.
<b>6. For example:</b> Merge Sort & Binary Search etc.	<b>6. For example:</b> Matrix Multiplication.

# Greedy vs Dynamic

Feature

**Feasibility**

**Optimality**

**Recursion**

**Memorization**

**Time complexity**

**Fashion**

**Example**

Greedy method

In a greedy Algorithm, we make whatever choice seems best at the moment in the hope that it will lead to global optimal solution.

In Greedy Method, sometimes there is no such guarantee of getting Optimal Solution.

A greedy method follows the problem solving heuristic of making the locally optimal choice at each stage.

It is more efficient in terms of memory as it never look back or revise previous choices

Greedy methods are generally faster. For example, [Dijkstra's shortest path](#) algorithm takes  $O(E\log V + V\log V)$  time.

The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices.

Fractional knapsack .

Dynamic programming

In Dynamic Programming we make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution .

It is guaranteed that Dynamic Programming will generate an optimal solution as it generally considers all possible cases and then choose the best.

A Dynamic programming is an algorithmic technique which is usually based on a recurrent formula that uses some previously calculated states.

It requires dp table for memorization and it increases it's memory complexity.

Dynamic Programming is generally slower. For example, [Bellman Ford algorithm](#) takes  $O(VE)$  time.

Dynamic programming computes its solution bottom up or top down by synthesizing them from smaller optimal sub solutions.

0/1 knapsack problem

## **Course Learning Outcome :**

**Apply Greedy approach to solve Huffman coding**

**Comparison of Brute force and Huffman method of Encoding**

# Topics

- Huffman Problem
- Problem Analysis (Real Time Example)
  - Binary coding techniques
  - Prefix codes
- Algorithm of Huffman Coding Problem
- Time Complexity
  - Analysis and Greedy Algorithm
- Conclusion

# Huffman Coding using Greedy Algorithm

# Using ASCII Code: using Text Encoding

- Our objective is to develop a code that represents a given text as compactly as possible.
- A standard encoding is ASCII, which represents every character using 7 bits

Example

Represent “An English sentence” using ASCII code

- 1000001 (A) 1101110 (n) 0100000 ( ) 1000101 (E)  
1101110 (n) 1100111 (g) 1101100 (l) 1101001 (i)  
1110011 (s) 1101000 (h) 0100000 ( ) 1110011 (s)  
1100101 (e) 1101110 (n) 1110100 (t) 1100101 (e)  
1101110 (n) 1100011 (c) 1100101 (e)  
= 133 bits  $\approx$  17 bytes

# Refinement in Text Encoding

- Now a better code is given by the following encoding:

⟨space⟩ = 000, A = 0010, E = 0011, s = 010,  
c = 0110, g = 0111, h = 1000, i = 1001,  
l = 1010, t = 1011, e = 110, n = 111

- Then we encode the phrase as

0010 (A) 111 (n) 000 ( ) 0011 (E) 111 (n) 0111 (g)  
1010 (l) 1001 (i) 010 (s) 1000 (h) 000 ( ) 010 (s)  
110 (e) 111 (n) 1011 (t) 110 (e) 111 (n) 0110 (c)  
110 (e)

- This requires 65 bits  $\approx$  9 bytes. Much improvement.
- The technique behind this improvement, i.e., Huffman coding which we will discuss later on.

# Major Types of Binary Coding

There are many ways to represent a file of information.

## Binary Character Code (or Code)

- each character represented by a unique binary string.
- **Fixed-Length Code**
  - If  $\Sigma = \{0, 1\}$  then
  - All possible combinations of two bit strings
$$\Sigma \times \Sigma = \{00, 01, 10, 11\}$$
  - If there are less than four characters then two bit strings enough
  - If there are less than three characters then two bit strings not economical

# Fixed Length Code

- **Fixed-Length Code**
  - All possible combinations of three bit strings
$$\Sigma \times \Sigma \times \Sigma = \{000, 001, 010, 011, 100, 101, 110, 111\}$$
  - If there are less than nine characters then three bit strings enough
  - If there are less than five characters then three bit strings not economical and can be considered two bit strings
  - If there are six characters then needs 3 bits to represent, following could be one representation.
$$\begin{aligned}a &= 000, b = 001, c = 010, \\d &= 011, e = 100, f = 101\end{aligned}$$

# Variable Length Code

- Variable-Length Code
  - better than a fixed-length code
  - It gives short code-words for frequent characters and
  - long code-words for infrequent characters
- Assigning variable code requires some skill
- Before we use variable codes we have to discuss prefix codes to assign variable codes to set of given characters

# Prefix Code (Variable Length Code)

- A prefix code is a code typically a variable length code, with the “prefix property”
- Prefix property is defined as no codeword is a prefix of any other code word in the set.

## Examples

1. Code words  $\{0,10,11\}$  has prefix property
2. A code consisting of  $\{0, 1, 10, 11\}$  does not have, because “1” is a prefix of both “10” and “11”.

## Other names

- Prefix codes are also known as prefix-free codes, prefix condition codes, comma-free codes, and instantaneous codes etc.

# Why are prefix codes?

- Encoding simple for any binary character code;
- Decoding also easy in prefix codes. This is because no codeword is a prefix of any other.

## Example 1

- If  $a = 0$ ,  $b = 101$ , and  $c = 100$  in prefix code then the string: 0101100 is coded as  $0 \cdot 101 \cdot 100$

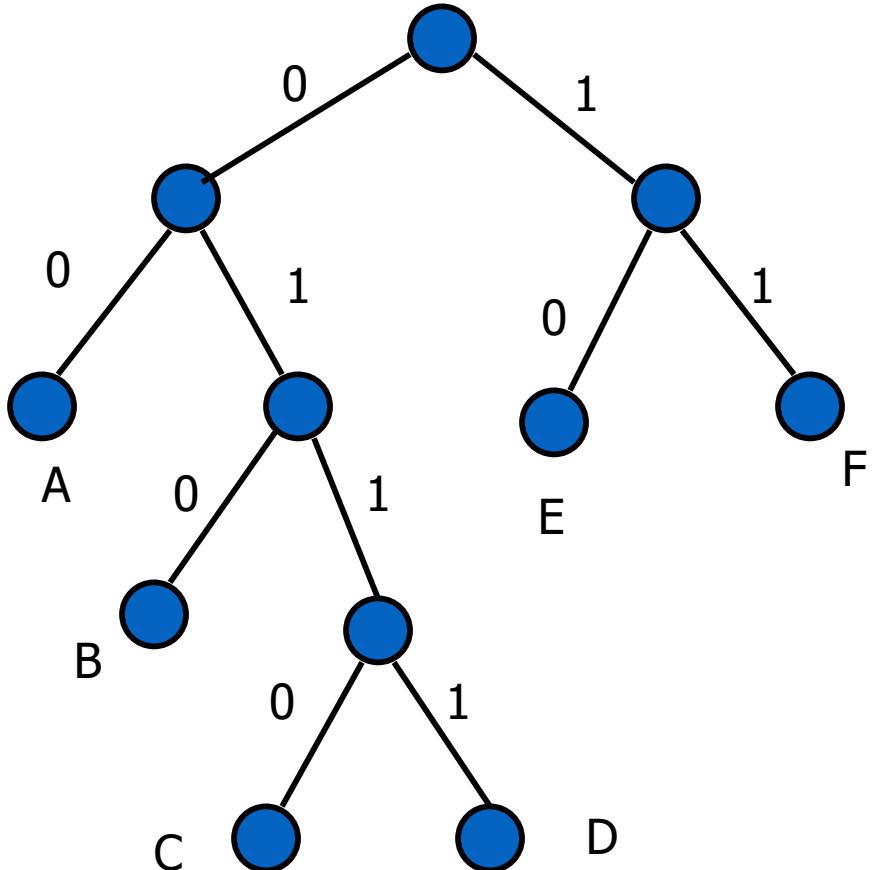
## Example 2

- In code words:  $\{0, 1, 10, 11\}$ , receiver reading “1” at the start of a code word would not know whether
  - that was complete code word “1”, or
  - prefix of the code word “10” or of “11”

# Prefix codes and binary trees

Tree representation of prefix codes

A	00
B	010
C	0110
D	0111
E	10
F	11



# Huffman Codes

- In Huffman coding, variable length code is used
- Data considered to be a sequence of characters.
- Huffman codes are a widely used and very effective technique for compressing data
  - Savings of 20% to 90% are typical, depending on the characteristics of the data being compressed.
- Huffman's greedy algorithm uses a table of the frequencies of occurrence of the characters to build up an optimal way of representing each character as a binary string.
- Now let us see an example to understand the concepts used in Huffman coding

## Example: Huffman Codes

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

A data file of 100,000 characters contains only the characters a–f,

with the frequencies indicated above.

- If each character is assigned a 3-bit **fixed-length codeword**, the file can be encoded in 300,000 bits.

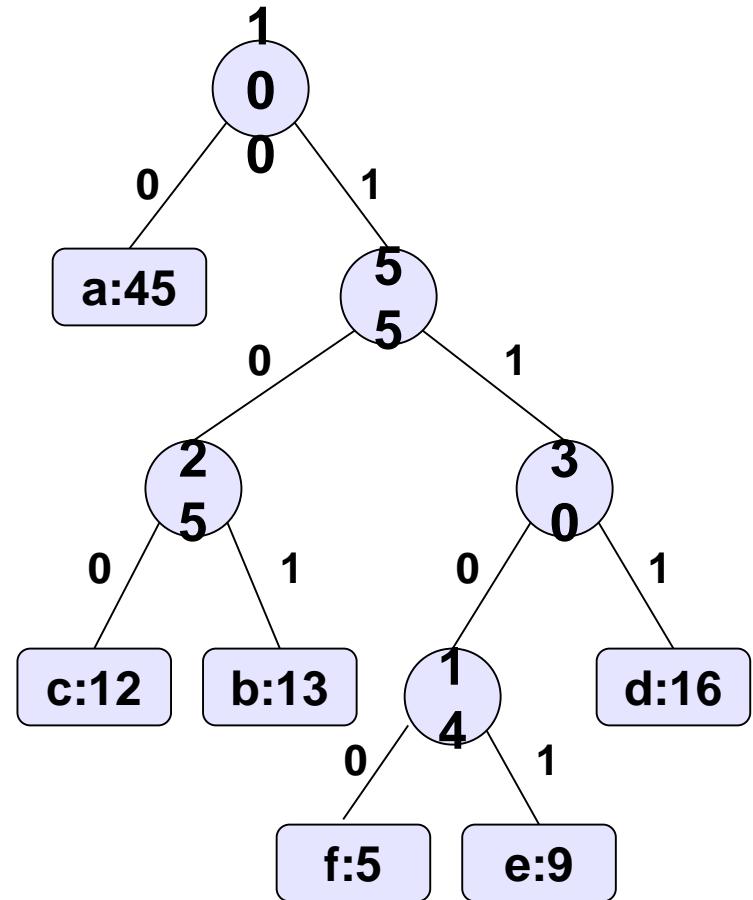
- Using the **variable-length code**

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1,000 = 224,000 \text{ bits}$$

which shows a savings of approximately 25%

# Binary Tree: Variable Length Codeword

	Frequency (in thousands)	Variable- length codeword
a	45	0
b	13	101
c	12	100
d	16	111
e	9	1101
f	5	1100



The tree corresponding to the variable-length code is shown for the data in table.

# Cost of Tree Corresponding to Prefix Code

- Given a tree  $T$  corresponding to a prefix code. For each character  $c$  in the alphabet  $C$ ,
  - let  $f(c)$  denote the frequency of  $c$  in the file and
  - let  $d_T(c)$  denote the depth of  $c$ 's leaf in the tree.
  - $d_T(c)$  is also the length of the codeword for character  $c$ .
  - The number of bits required to encode a file is

$$B(T) = \sum_{c \in C} f(c)d_T(c)$$

- which we define as the ***cost*** of the tree  $T$ .

# Algorithm: Constructing a Huffman Codes

Huffman (C)

```
1   $n \leftarrow |C|$ 
2   $Q \leftarrow C$ 
3  for  $i \leftarrow 1$  to  $n - 1$ 
4      do allocate a new node  $z$ 
5           $left[z] \leftarrow x \leftarrow \text{Extract-Min} (Q)$ 
6           $right[z] \leftarrow y \leftarrow \text{Extract-Min} (Q)$ 
7           $f[z] \leftarrow f[x] + f[y]$ 
8          Insert ( $Q, z$ )
9  return Extract-Min( $Q$ )   $\square$  Return root of the tree.
```

# Example: Constructing a Huffman Codes

**Q**    f:5    e:9    c:12    b:13    d:16    a:45  
:

The initial set of  $n = 6$  nodes, one for each letter.  
Number of iterations of loop are 1 to  $n-1$  ( $6-1 = 5$ )

# Constructing a Huffman Codes

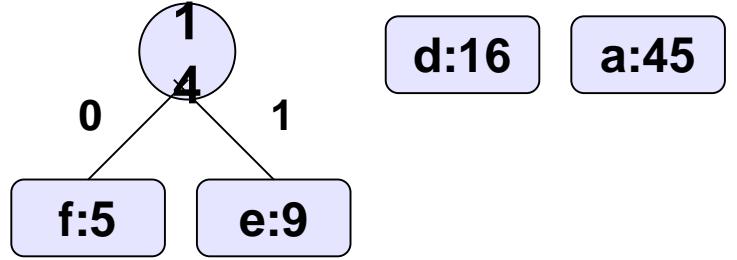
**Q:**

c:12

b:13

d:16

a:45



for  $i \leftarrow 1$

    Allocate a new node  $z$

$left[z] \leftarrow x \leftarrow \text{Extract-Min } (Q) = f:5$

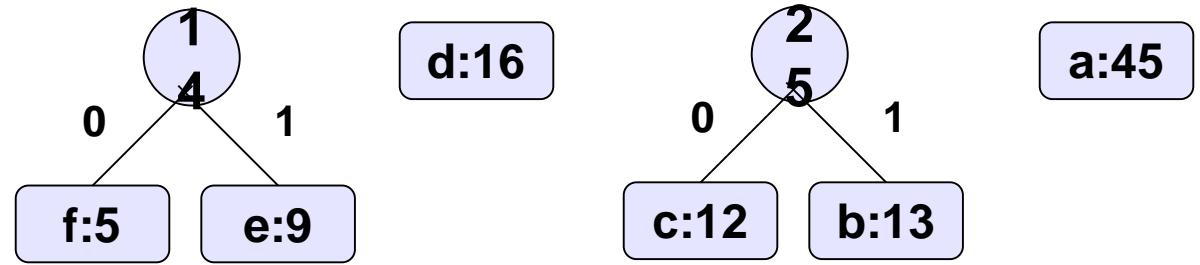
$right[z] \leftarrow y \leftarrow \text{Extract-Min } (Q) = e:9$

$f[z] \leftarrow f[x] + f[y] \quad (5 + 9 = 14)$

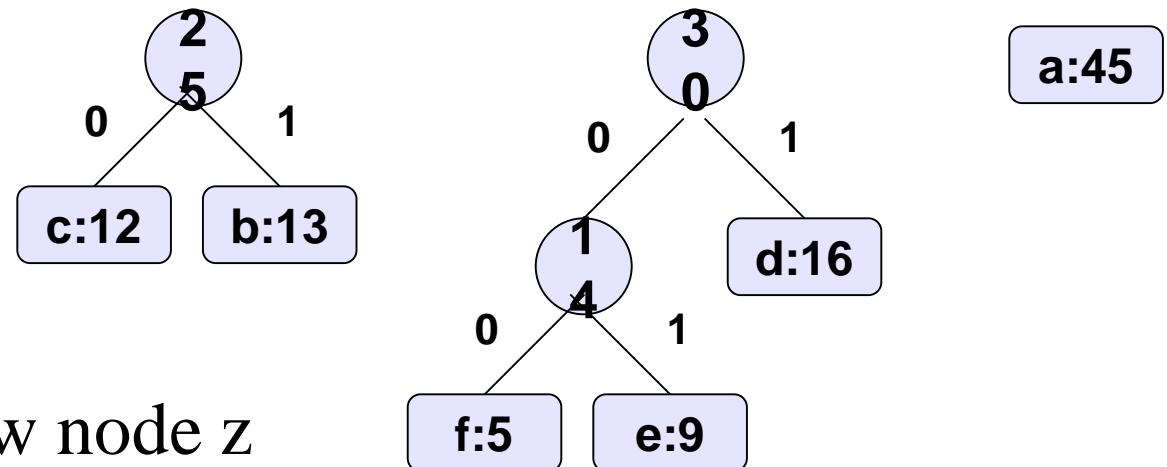
    Insert  $(Q, z)$

# Constructing a Huffman Codes

**Q:**



**Q:**



for  $i \leftarrow 3$

Allocate a new node  $z$

$left[z] \leftarrow x \leftarrow \text{Extract-Min } (Q) = z:14$

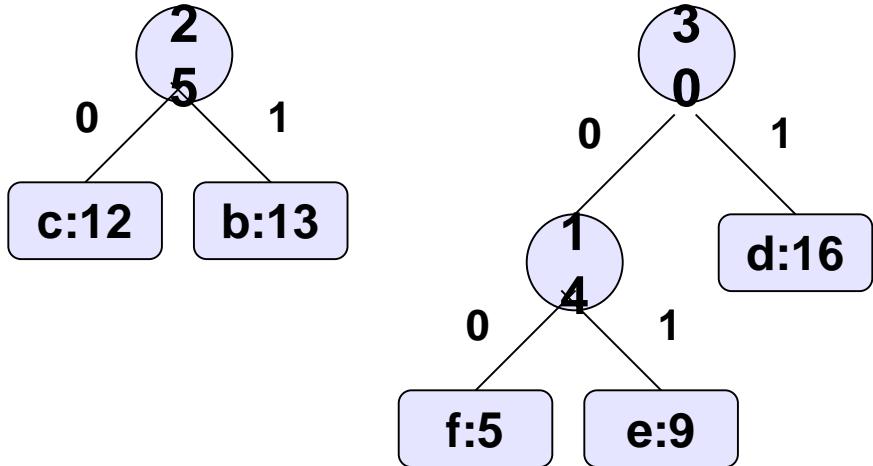
$right[z] \leftarrow y \leftarrow \text{Extract-Min } (Q) = d:16$

$f[z] \leftarrow f[x] + f[y] \quad (14 + 16 = 30)$

Insert  $(Q, z)$

# Constructing a Huffman Codes

**Q:**



a:45

**Q:** a:4  
5

for i  $\leftarrow$  4

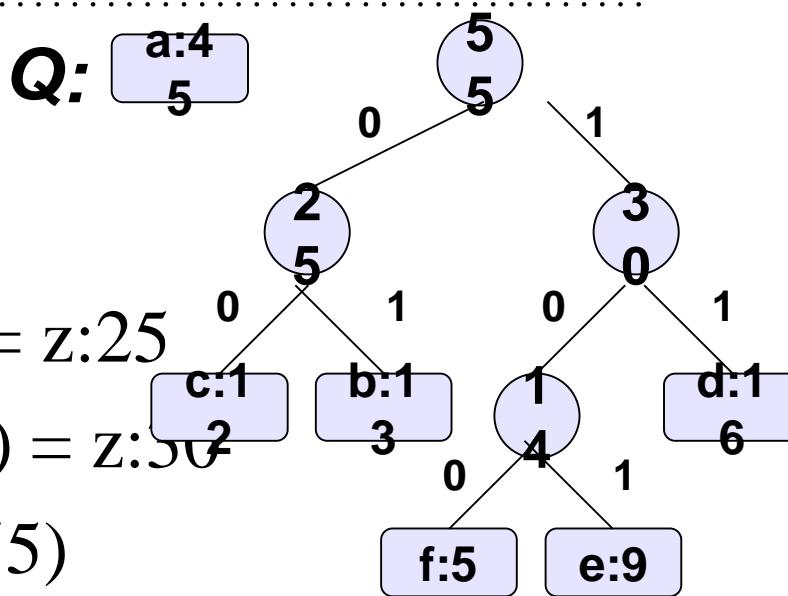
Allocate a new node z

$left[z] \leftarrow x \leftarrow \text{Extract-Min } (Q) = z:25$

$right[z] \leftarrow y \leftarrow \text{Extract-Min } (Q) = z:5$

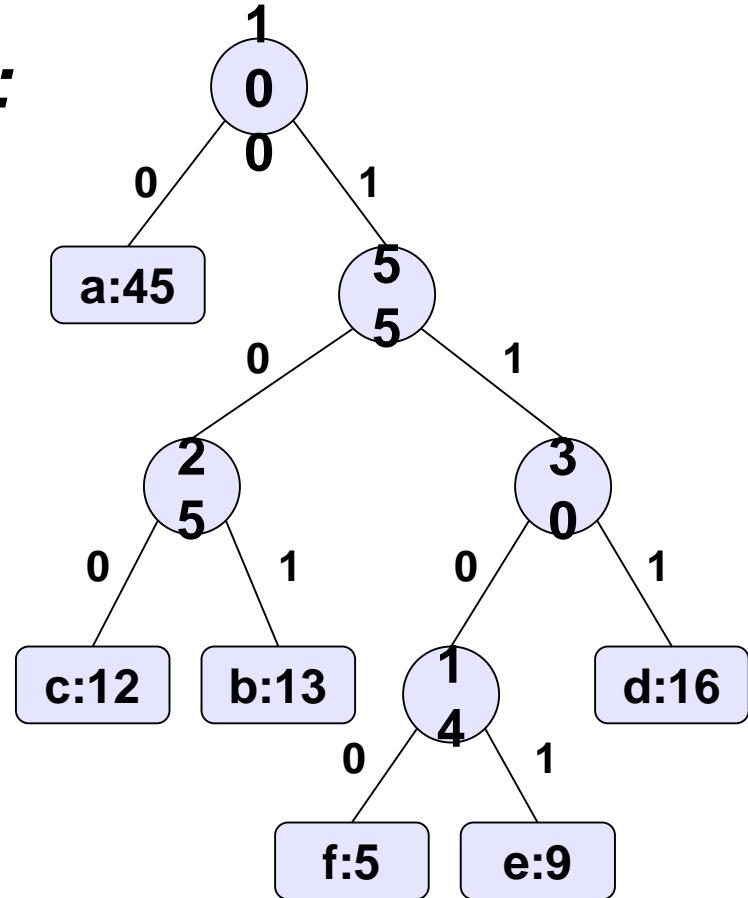
$f[z] \leftarrow f[x] + f[y] \quad (25 + 30 = 55)$

Insert  $(Q, z)$



# Constructing a Huffman Codes

**Q:**



for  $i \leftarrow 5$

Allocate a new node  $z$

$left[z] \leftarrow x \leftarrow \text{Extract-Min } (Q) = a:45$

$right[z] \leftarrow y \leftarrow \text{Extract-Min } (Q) = z:55$

$f[z] \leftarrow f[x] + f[y] \quad (45 + 55 = 100)$

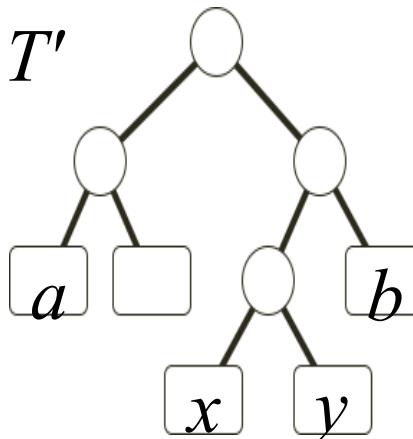
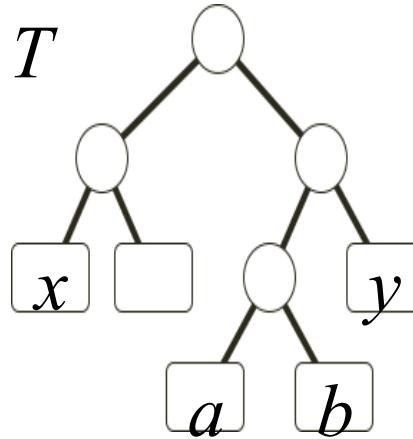
Insert  $(Q, z)$

# Lemma 1: Greedy Choice

*There exists an optimal prefix code such that the two characters with smallest frequency are siblings and have maximal depth in  $T$ .*

Proof:

- Let  $x$  and  $y$  be two such characters, and let  $T$  be a tree representing an optimal prefix code.
- Let  $a$  and  $b$  be two sibling leaves of maximal depth in  $T$ , and assume with out loss of generality that  $f(x) \leq f(y)$  and  $f(a) \leq f(b)$ .
- This implies that  $f(x) \leq f(a)$  and  $f(y) \leq f(b)$ .
- Let  $T'$  be the tree obtained by exchanging  $a$  and  $x$  and  $b$  and  $y$ .



# Conclusion

- Huffman Algorithm is analyzed
- Design of algorithm is discussed
- Computational time is given
- Applications can be observed in various domains e.g.  
data compression, defining unique codes, etc.  
generalized algorithm can used for other optimization  
problems as well

## Two questions

- Why does the algorithm produce the best tree ?
- How do you implement it efficiently ?

# Knapsack problem using greedy approach

# Session Learning Outcome-SLO

**Greedy is used** in optimization problems. The **algorithm** makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem

# GREEDY INTRODUCTION

- Greedy algorithms are simple and straightforward.
- They are shortsighted in their approach
- A greedy algorithm is similar to a dynamic programming algorithm, but the difference is that solutions to the sub problems do not have to be known at each stage,
- instead a "greedy" choice can be made of what looks best for the moment.

# GREEDY INTRODUCTION

- Many real-world problems are optimization problems in that they attempt to find an optimal solution among many possible candidate solutions.
- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution
- A “greedy algorithm” sometimes works well for optimization problems
- A greedy algorithm works in phases. At each phase: You take the best you can get right now, without regard for future consequences. It hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum.

# APPLICATIONS OF GREEDY APPROACH

- 1) Job sequenced with deadline.
- 2) Knapsack problem.
- 3) Huffman coding .
- 4) Optimal merge pattern.
- 5) Min cost spanning tree.
- 6) Dijkstra's algorithm.
- 7) Bellman ford
- 8) Breath first traversal(BFT)

## FRACTIONAL KNAPSACK PROBLEM

Now applying the greedy method to solve the knapsack problem. We are given with:-

- Objects n
- Weights  $W_i$
- Profits  $P_i$
- Knapsack of capacity 'm'

So here we have n objects and a knapsack of capacity 'm'. Object 'i' has weight ' $W_i$ '. If the fraction  $X_i$ ,  $0 \leq X_i \leq 1$ , of object 'i' is placed into the knapsack, then a profit of ' $P_i X_i$ ' is earned.

**THE OBJECTIVE** is to obtain a filling of the knapsack that maximizes the total profit earned. As the knapsack capacity is 'm' , we require the total weight of all objects to be at most 'm'. Formally the problem is stated as:-

**Maximize**       $\sum_{1 \leq i \leq n} P_i X_i$       (1)

**Subject to**       $\sum_{i \leq i \leq n} W_i X_i \leq m$       (2)

**And**       $0 \leq X_i \leq 1 , 1 \leq i \leq n$       (3)

And a feasible solution is any set satisfying equation (2) and equation (1). An optimal solution is a feasible solution for which equation (1) is maximized.

## FRACTIONAL KNAPSACK ALGORITHM

*Fractional knapsack ( a , p , w , c )*

```
//a is array of items,p is array for profits, w is array for weights and  
c is array for capacity of knapsack. //  
  
{  
    n := length [a] ;                                // finding total number of items given.  
    for i := 1 to n  
        do  
            ratio [ i ] := p [ i ] / w [ i ];  
            x [ i ] := 0 ;  
    done  
    sort ( a, ratio ) ;                            // to arrange item with highest value first  
    weight := 0  
    i := 1
```

```
while ( i <= n and weight < c )
{
    if ( weight +w [ i ] <= c )
        then
            x [ i ] := i;
            weight := weight + w [ i ];
        else
            r := ( c- weight ) / w [ i ];
            x [ i ] := r ;
            weight := c ;
            i := i + 1 ;
}
output ( x )
}
```

The fractional knapsack problem can be more clearly explained with the help of the following example:

### EXAMPLE :-

The total capacity of knapsack is 20.

The total number of items present is 3.

**Profit values Pi**

P1	P2	P3
25	24	15

**Weights values Wi**

W1	W2	W3
18	15	10

Now as we go according to the algorithm, we have :

$$n = 3$$

$$c = 20$$

$$p[i] = \{ 25, 24, 15 \}$$

$$w[i] = \{ 18, 15, 10 \}$$

Now the ratio array will have values:

$$\text{ratio [1]} = 25 / 18 = 1.4$$

$$\text{ratio [2]} = 24 / 15 = 1.6$$

$$\text{ratio [3]} = 15 / 10 = 1.5$$

simultaneously ,

$$x [1] = 0$$

$$x [2] = 0$$

$$x [3] = 0$$

then we sort the ratio array and it is as follows:-

1.6	1.5	1.4
1(item 2)	2 ( item 3)	3 ( item 1)

Weight = 0

i = 1

P

P2	P3	P1
24	15	25

w

W2	W3	W1
15	10	18

Now we check i for 1,2 3 and till weight (0) < 20

For i =1 and 0 < 20

check if ( 0 + 15 <= 20)

15 <= 20

=> x [1] = 1

weight = 15

For i = 2 and 15 < 20

else

$$=>r = (20 - 15) / 10 = 5 / 10$$

$$x [2] = 5 / 10$$

weight = 20

For i = 3 and 20 !<20      END

**So in this case order of consideration is 2 , 3 , 1.**

**We choose item 2 first whose weight is 15 and left space in knapsack is 5.**

**So we will add as much as possible from the next highest order that is item 3.**

**Which gives us 5/10.**

**Thus the profit is:**

$$24 + ( 5 / 10 ) * 15 = 31.5$$

This can also be explained by taking one more example as follows:-

**EXAMPLE :**

(w1,p1)	(w2,p2)	(w3,p3)	(w4,p4)	(w5,p5)
(120,5)	(150,5)	(200,4)	(150,8)	(140,3)

**Total capacity c = 600**

The profit / weight ratios are :

$$\underline{p_1/w_1 = 5/120 = .0417}; \quad \underline{p_2/w_2 = 5/150 = .0333}; \quad \underline{p_3/w_3 = 4/200 = .0200};$$

$$\underline{p_4/w_4 = 8/150 = .0533}; \quad \underline{p_5/w_5 = 3/140 = .0214};$$

Thus the order of consideration is 4, 1, 2, 5, 3

We take all of items 4, 1, 2 and 5 for a total weight of knapsack is

$$150 + 120 + 150 + 140 = 560$$

We now only have room for 40 units of weight, so we take  $40/200 = 1/5$  of object 3

The profit is thus  $5 + 5 + 0.2 \cdot 4 + 8 + 3 = 21.8$

**Thus the total profit gained is = 21.8**

## COMPLEXITY OF FRACTIONAL KNAPSACK PROBLEM

The complexity of fractional knapsack problem is  $O(n \log(n))$

## RUNNING TIME

Given a set of  $n$  tasks specified by their start and finish times, Algorithm Task Schedule produces a schedule of the tasks with the minimum number of machines in  $O(n \log n)$  time.

- Use heap-based priority queue to store tasks with the start time as the priorities
- Finding the earliest task takes  $O(\log n)$  time

## APPLICATIONS OF FRACTIONAL KNAPSACK PROBLEM

- It is used in automated data mining.
- Used in linear programming problems
- Real life applications of money-time relationships.
- Criteria for choosing feasible project

# Home assignment /Questions

**Write code for the 0-1 knapsack problem using greedy algorithm.  
Output should display:**

**Number of items = 5**

**Weights & values for each item:**

**Weights    Values**

**2              10**

**3              12**

**4              20**

**5              22**

**7              40**

**Capacity of knapsack: 15**

**Maximum value that can be chosen with capacity of knapsack = 15**

# Tree Traversals

# Session Learning Outcome-SLO

These usage patterns can be divided into the three ways that we access the nodes of the tree.

# Tree Traversal

- Traversal is the process of visiting every node once
- Visiting a node entails doing some processing at that node, but when describing a traversal strategy, we need not concern ourselves with what that processing is

# Binary Tree Traversal Techniques

- Three recursive techniques for binary tree traversal
- In each technique, the left subtree is traversed recursively, the right subtree is traversed recursively, and the root is visited
- What distinguishes the techniques from one another is the order of those 3 tasks

# Preorder, Inorder, Postorder

- In Preorder, the root is visited before (pre) the subtrees traversals
- In Inorder, the root is visited in-between left and right subtree traversal
- In Preorder, the root is visited after (pre) the subtrees traversals

## Preorder Traversal:

1. Visit the root
2. Traverse left subtree
3. Traverse right subtree

## Inorder Traversal:

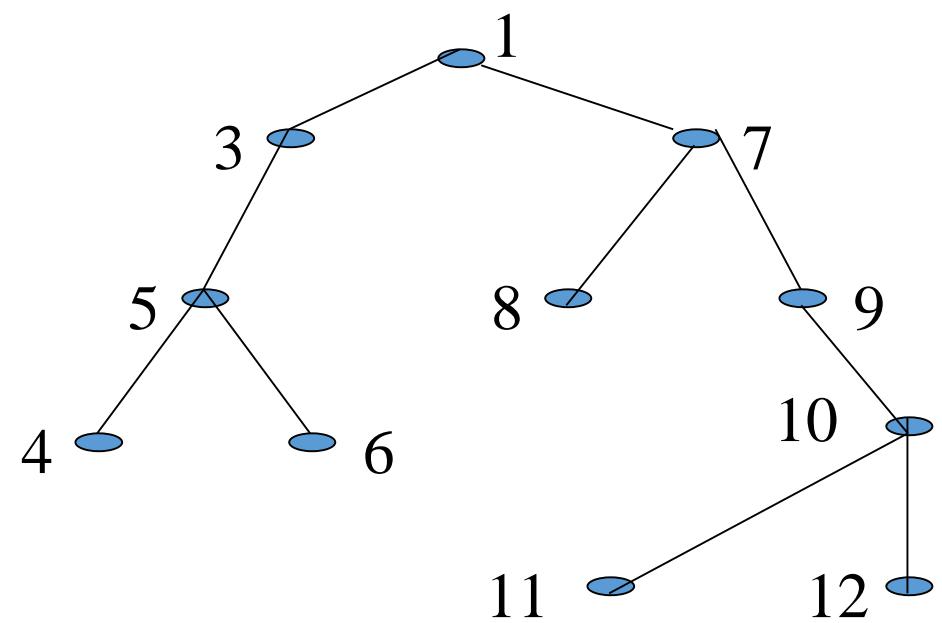
1. Traverse left subtree
2. Visit the root
3. Traverse right subtree

## Postorder Traversal:

1. Traverse left subtree
2. Traverse right subtree
3. Visit the root

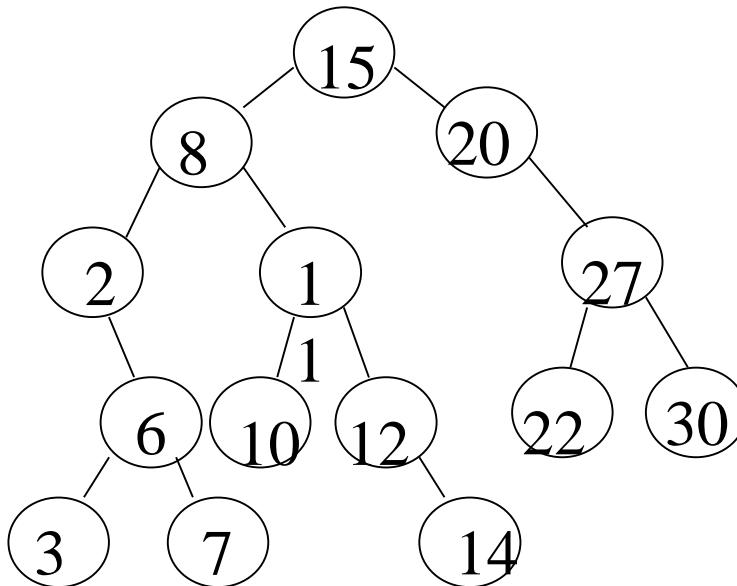
# Illustrations for Traversals

- Assume: visiting a node is printing its label
- Preorder:  
1 3 5 4 6 7 8 9 10 11 12
- Inorder:  
4 5 6 3 1 8 7 9 11 10 12
- Postorder:  
4 6 5 3 8 11 12 10 9 7 1



# Illustrations for Traversals (Contd.)

- Assume: visiting a node is printing its data
- Preorder: 15 8 2 6 3 7  
11 10 12 14 20 27 22 30
- Inorder: 2 3 6 7 8 10 11  
12 14 15 20 22 27 30
- Postorder: 3 7 6 2 10 14  
12 11 8 22 30 27 20 15



# Code for the Traversal Techniques

- The code for visit is up to you to provide, depending on the application
- A typical example for visit(...) is to print out the data part of its input node

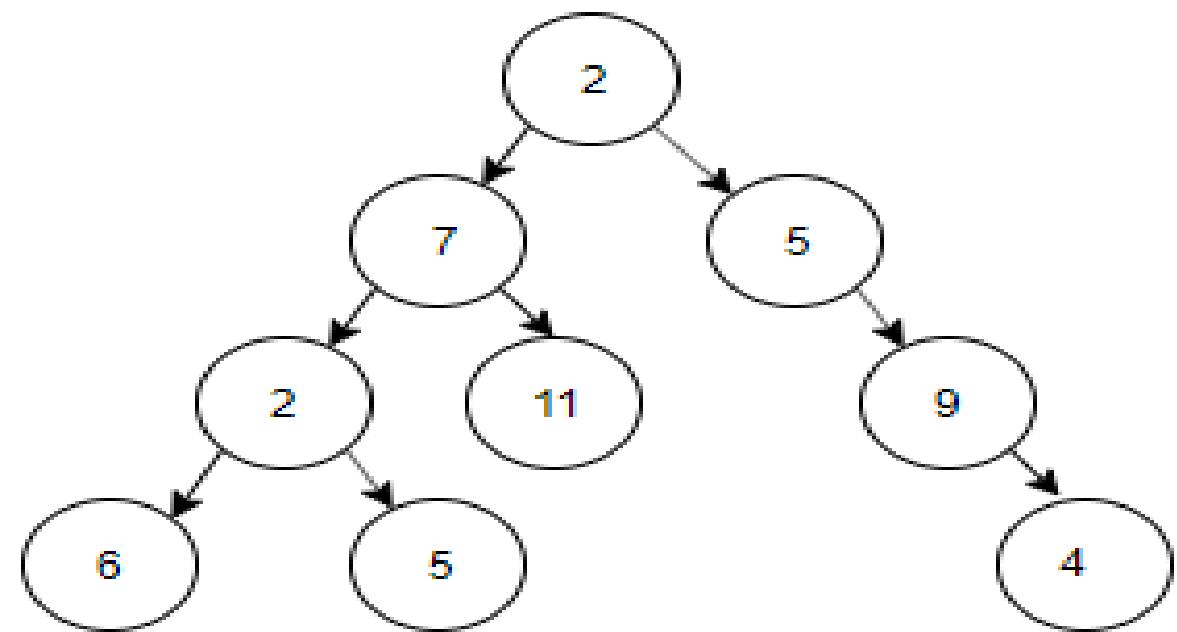
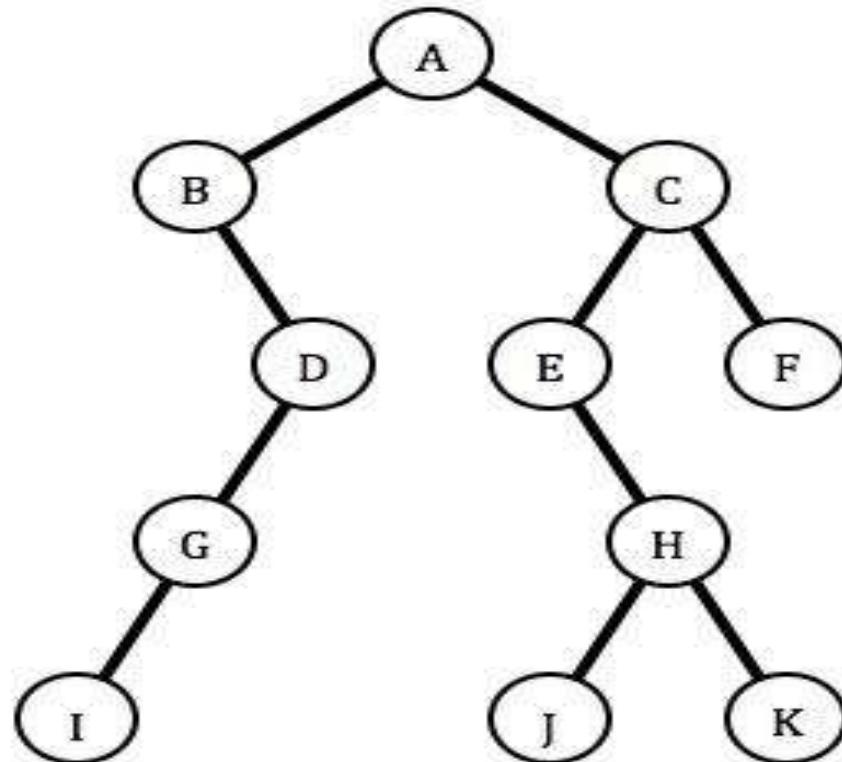
```
void preOrder(Tree *tree){
    if (tree->isEmpty( ))    return;
    visit(tree->getRoot( ));
    preOrder(tree->getLeftSubtree());
    preOrder(tree->getRightSubtree());
}
```

```
void inOrder(Tree *tree){
    if (tree->isEmpty( ))    return;
    inOrder(tree->getLeftSubtree( ));
    visit(tree->getRoot( ));
    inOrder(tree->getRightSubtree( ));
}
```

```
void postOrder(Tree *tree){
    if (tree->isEmpty( ))    return;
    postOrder(tree->getLeftSubtree( ));
    postOrder(tree->getRightSubtree( ));
    visit(tree->getRoot( ));
}
```

## Home assignment /Questions

Write the preorder, inorder and postorder traversals of the binary tree shown below.



# Minimum Spanning Tree

# Agenda

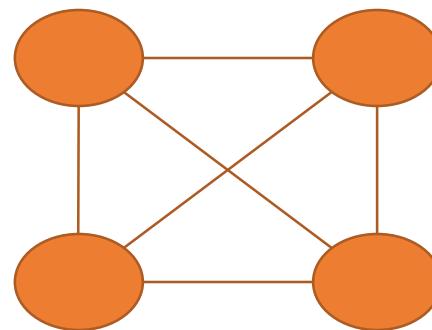
- Spanning Tree
- Minimum Spanning Tree
- Prim's Algorithm
- Kruskal's Algorithm
- Summary
- Assignment

# Session Outcomes

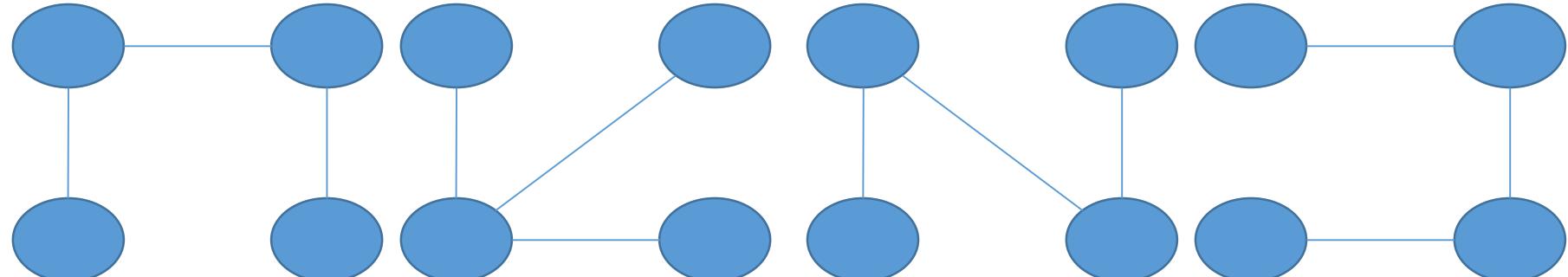
- Apply and analyse different problems using greedy techniques
- Evaluate the real time problems
- Analyse the different trees and find the low cost spanning tree

# Spanning Tree

- Given an undirected and connected graph  $G=(V,E)$ , a spanning tree of the graph  $G$  is a tree that spans  $G$  (that is, it includes every vertex of  $G$ ) and is a subgraph of  $G$  (every edge in the tree belongs to  $G$ )



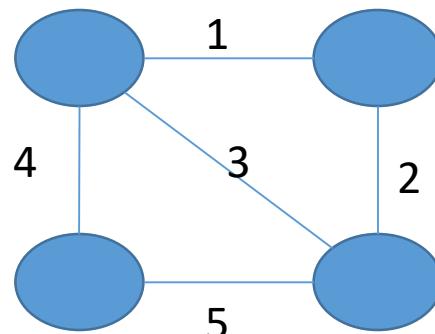
Undirected  
Graph



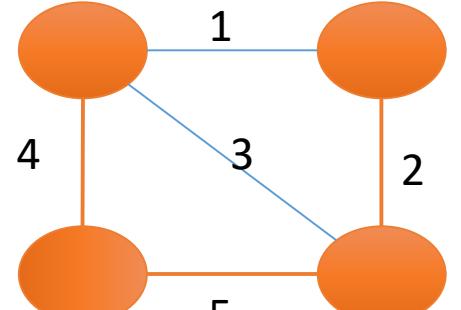
For of the spanning trees of the graph

# Minimum Spanning Tree

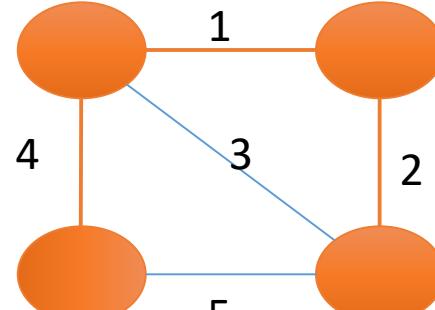
- The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.



Undirected  
Graph



Spanning Tree Cost=11  
 $(4+5+2)$



Minimum Spanning Tree  
Cost = 7  $(4+1+2)$

# Applications – MST

- Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:
- Cluster Analysis
- Handwriting recognition
- Image segmentation

# Prim's Algorithm

- Prim's Algorithm also use Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an **edge** in Kruskal's, we add **vertex** to the growing spanning tree in Prim's.
- Prim's Algorithm is preferred when-
  - The graph is dense.
  - There are large number of edges in the graph like  $E = O(V^2)$ .

# Steps for finding MST - Prim's Algorithm

***Step 1:*** Select any vertex

***Step 2:*** Select the shortest edge connected to that vertex

***Step 3:*** Select the shortest edge connected to any vertex already connected

***Step 4:*** Repeat step 3 until all vertices have been connected

# Prim's Algorithm

**Algorithm** Prim(G)

$V_T \leftarrow \{v_0\}$

$E_T \leftarrow \{ \} // \text{empty set}$

**for**  $i \leftarrow 1$  **to**  $|V|$  **do**

    find the minimum weight edge,  $e^* = (v^*, u^*)$       such      that       $v^*$       is  
    in  $V_T$  and  $u$  is in  $V - V_T$

$V_T \leftarrow V_T \text{ union } \{u^*\}$

$E_T \leftarrow E_T \text{ union } \{e^*\}$

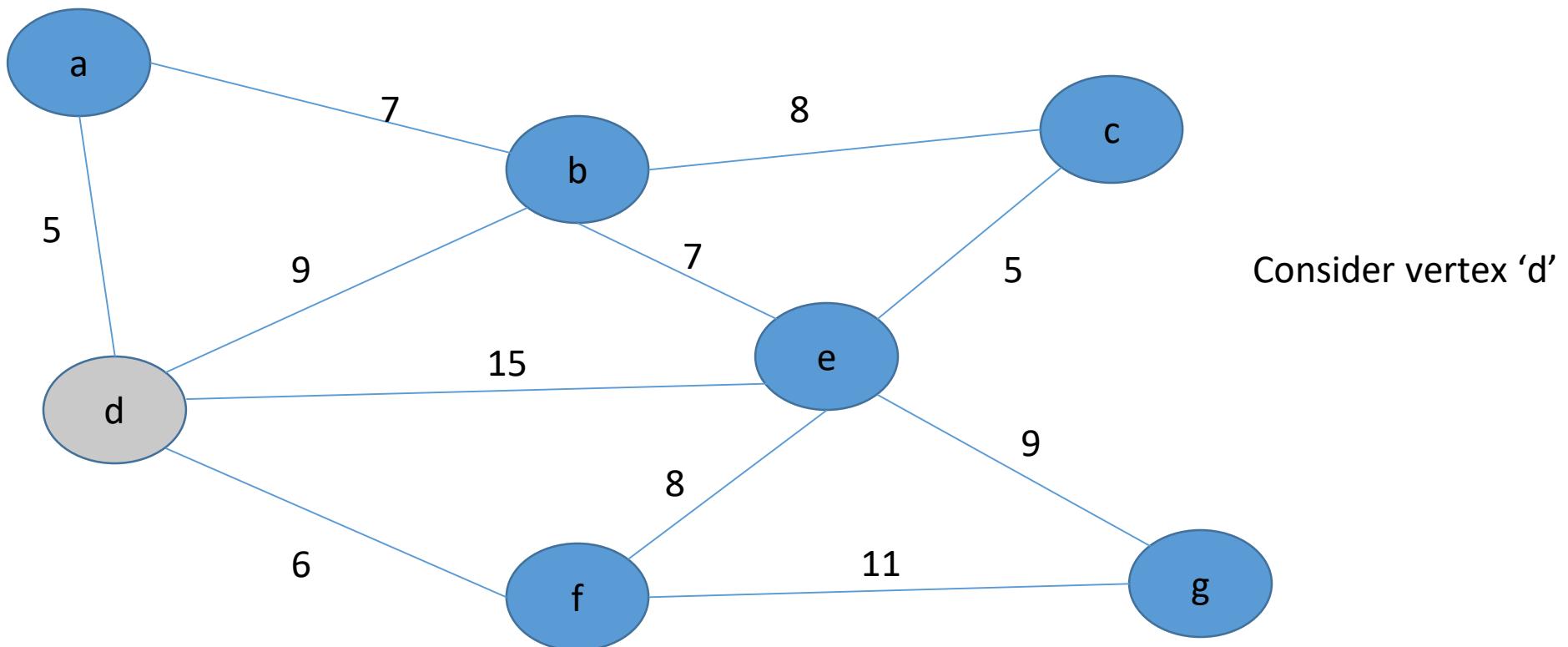
**return**  $E_T$

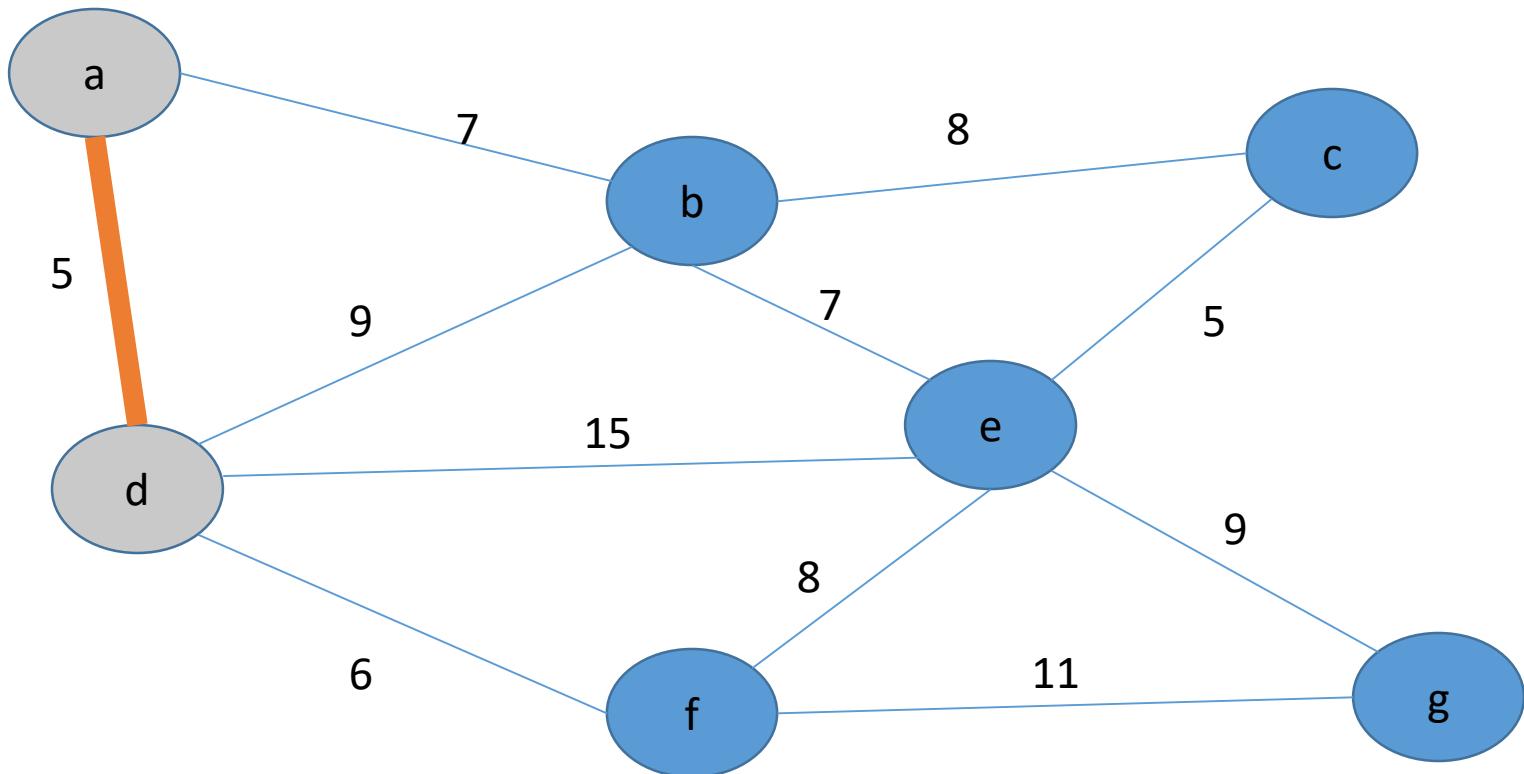
# Time Complexity – Prim's Algorithm

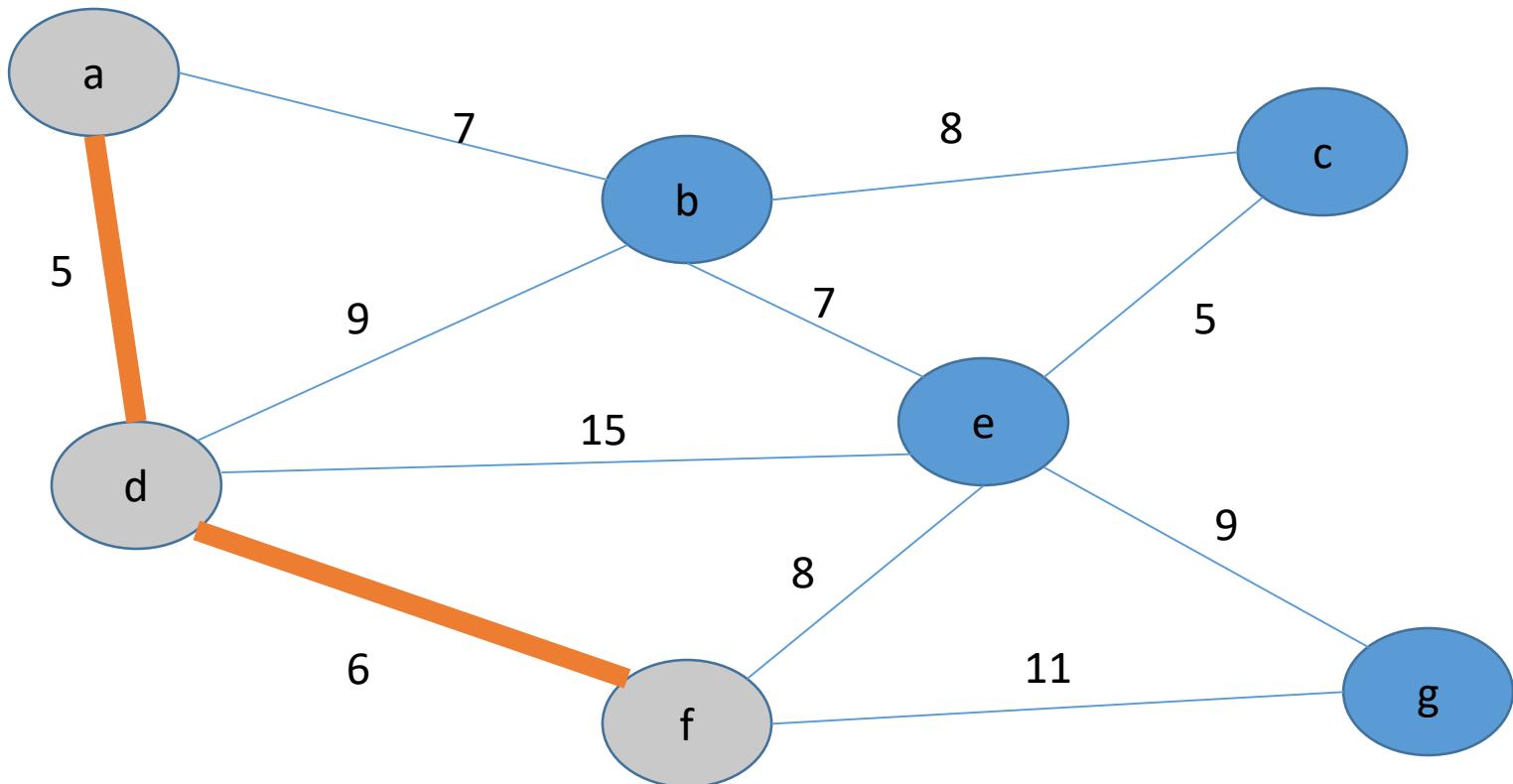
- If adjacency list is used to represent the graph, then using breadth first search, all the vertices can be traversed in  $O(V + E)$  time.
- We traverse all the vertices of graph using breadth first search and use a min heap for storing the vertices not yet included in the MST.
- To get the minimum weight edge, we use min heap as a priority queue.
- Min heap operations like extracting minimum element and decreasing key value takes  $O(\log V)$  time.
  
- So, overall time complexity
- $= O(E + V) \times O(\log V)$
- $= O((E + V)\log V)$
- $= O(E\log V)$

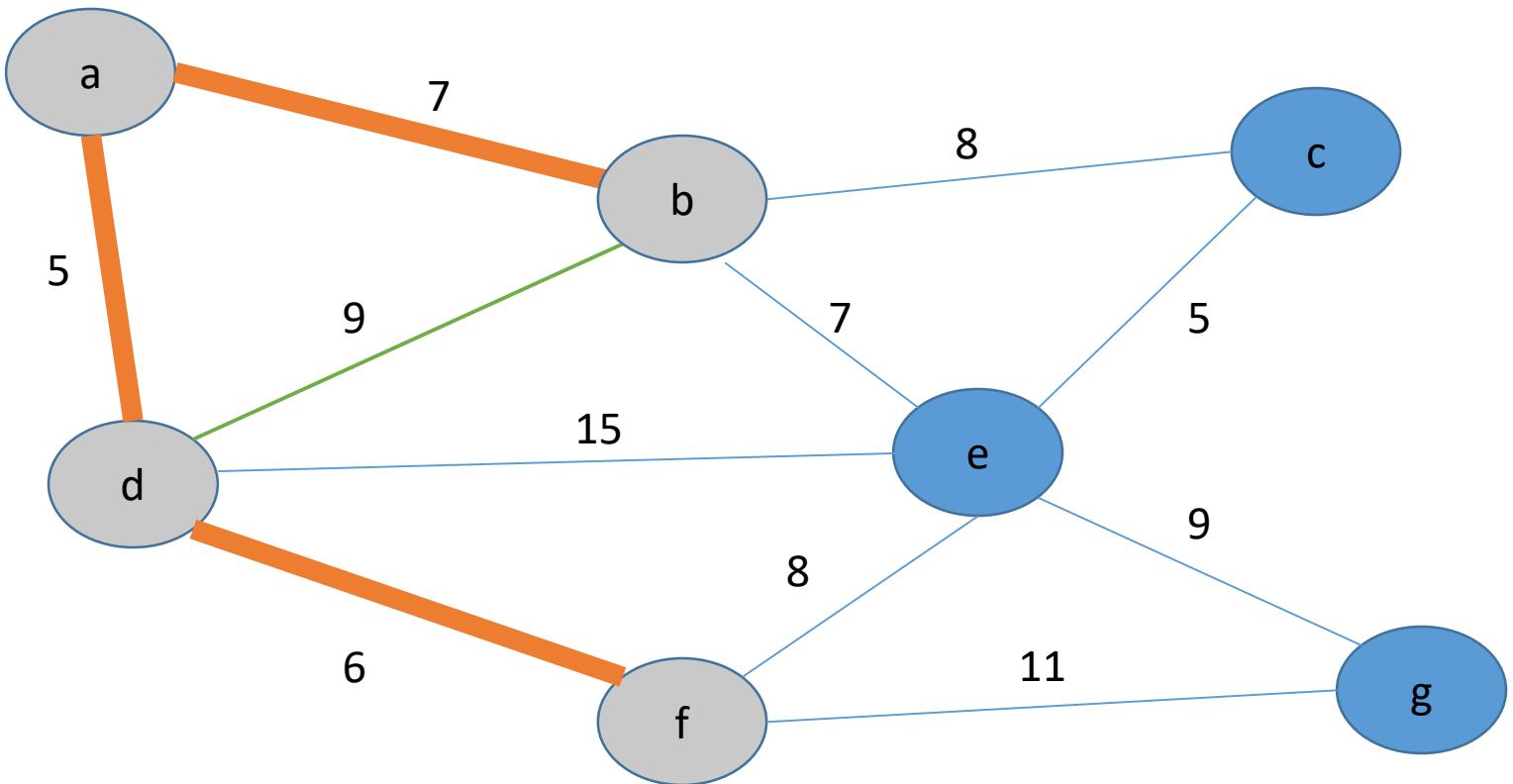
This time complexity can be improved and reduced to  $O(E + V\log V)$  using Fibonacci heap.

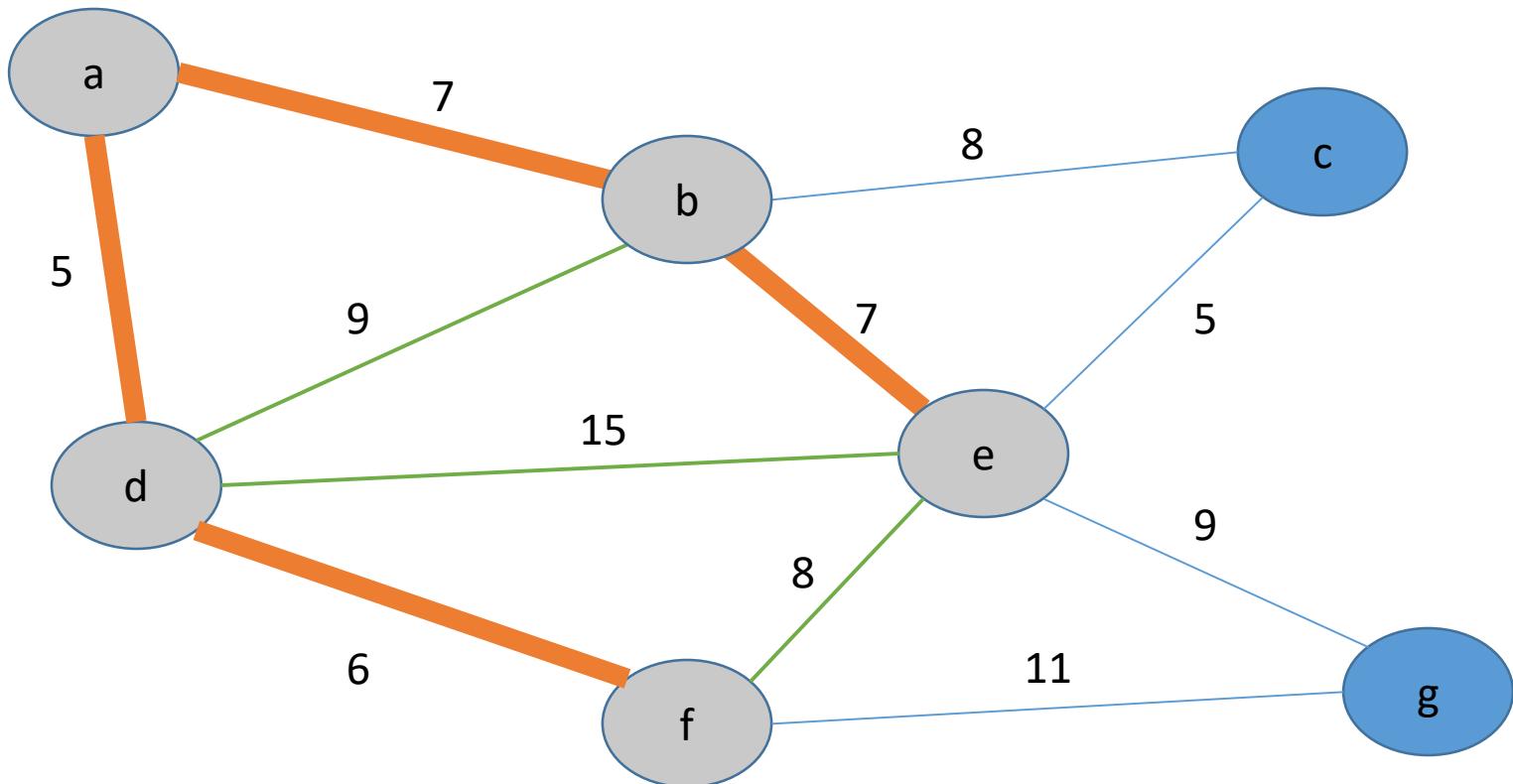
# Prim's Algorithm - Example

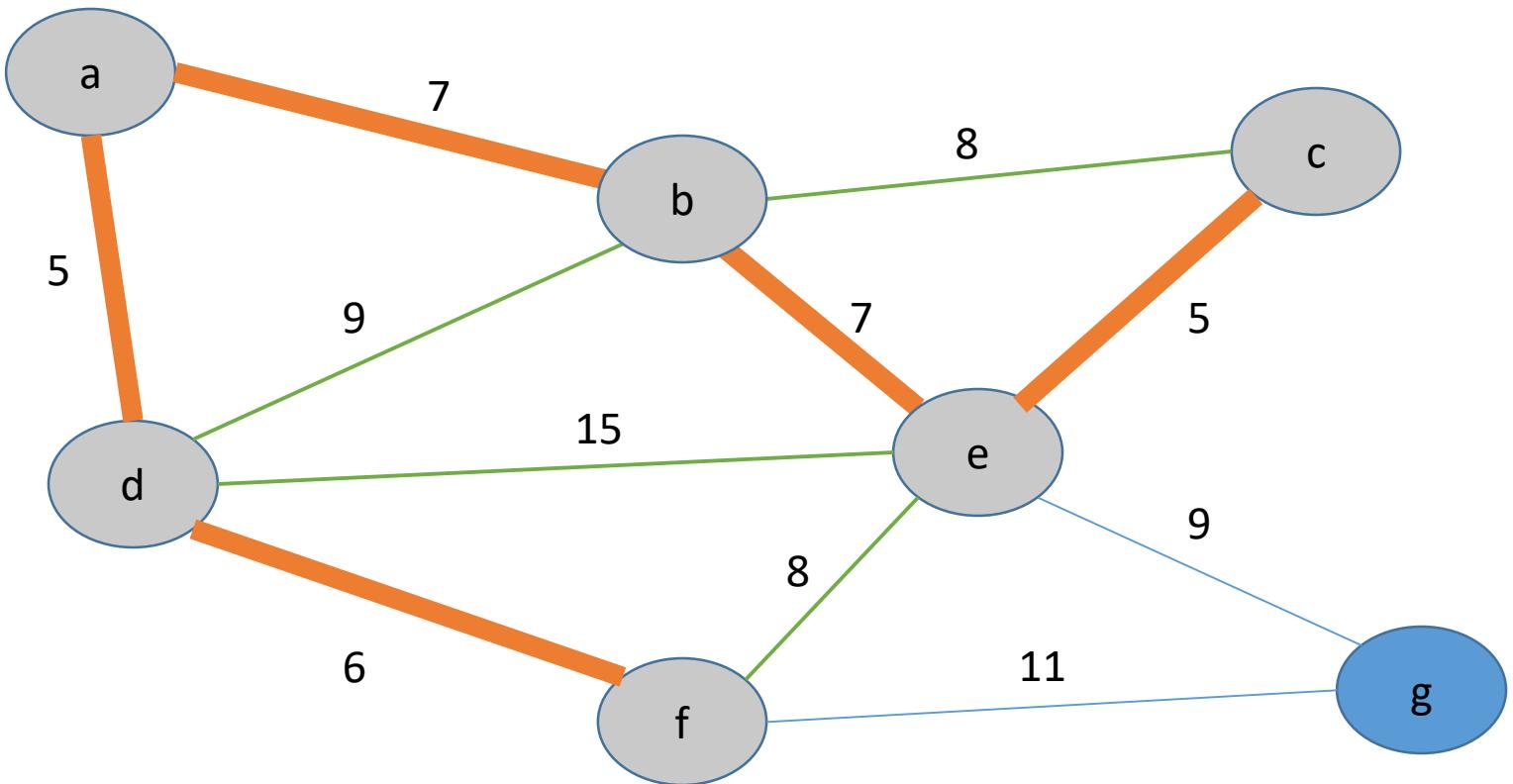


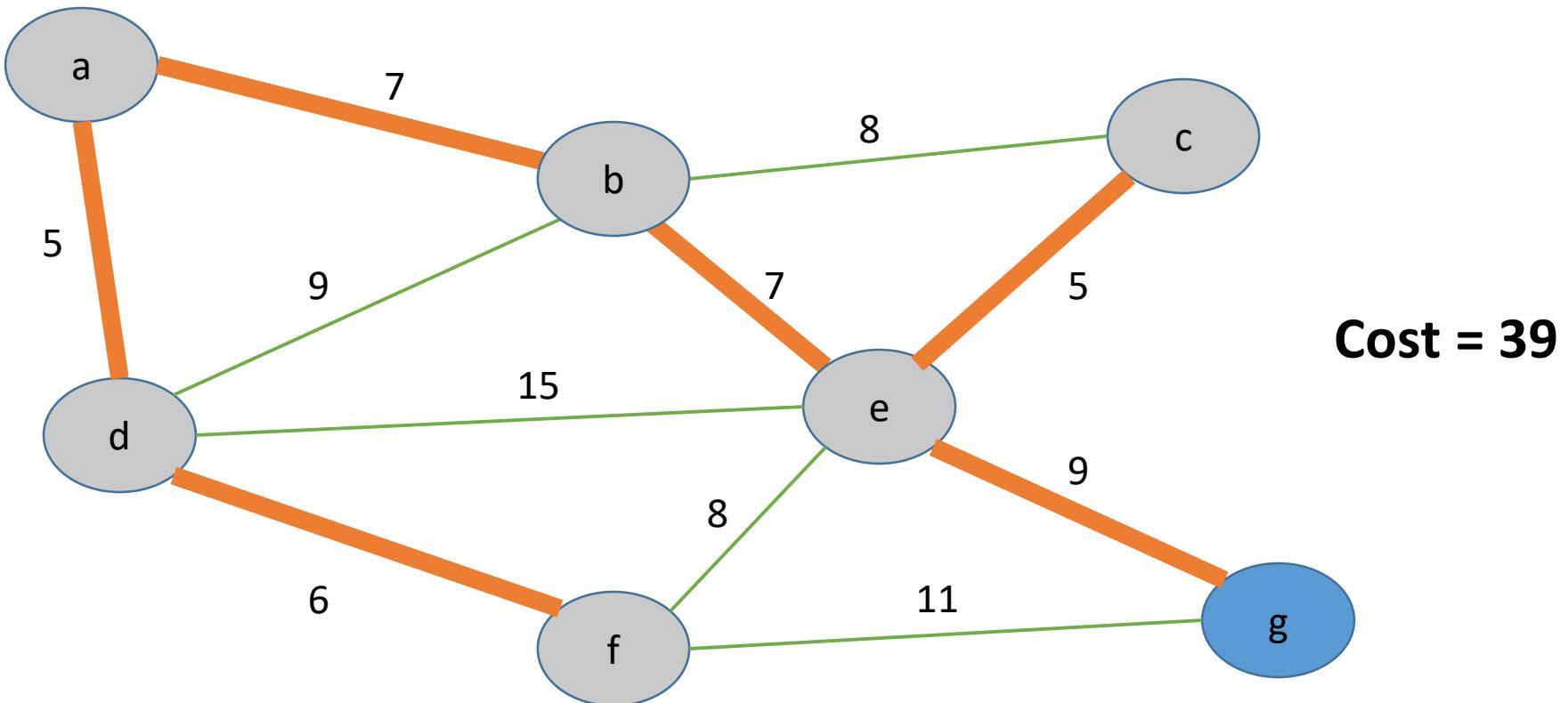












# Real-time Application – Prim's Algorithm

- Network for roads and Rail tracks connecting all the cities.
- Irrigation channels and placing microwave towers
- Designing a fiber-optic grid or ICs.
- Travelling Salesman Problem.
- Cluster analysis.
- Path finding algorithms used in AI.
- Game Development
- Cognitive Science

# Minimum Spanning tree – 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.
- 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.

# Steps for finding MST - Kruskal algorithm

**Step 1:** Sort all the edges in non-decreasing order of their weight.

**Step 2:** Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.

**Step 3:** Repeat step#2 until there are  $(V-1)$  edges in the spanning tree.

# Kruskal's Algorithm

**algorithm** Kruskal( $G$ ) **is**

$F := \emptyset$

**for each**  $v \in G.V$  **do**

        MAKE-SET( $v$ )

**for each**  $(u, v)$  **in**  $G.E$  ordered by weight( $u, v$ ), increasing **do**

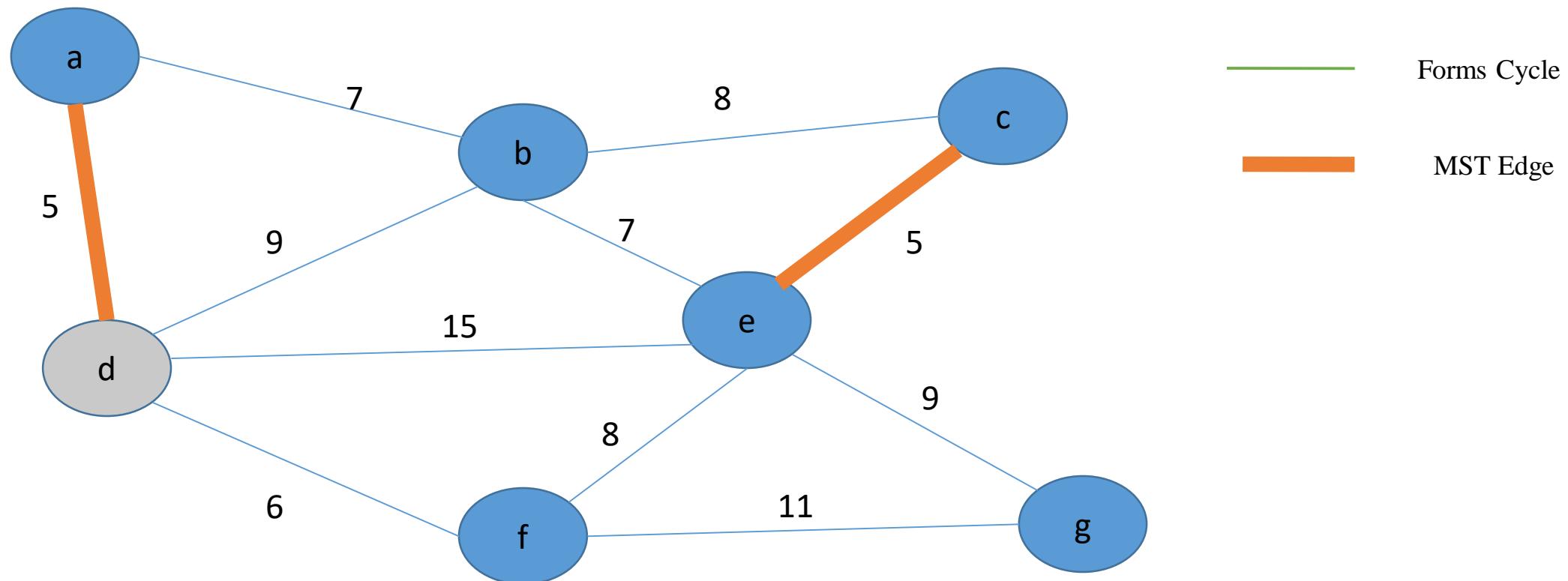
**if** FIND-SET( $u$ )  $\neq$

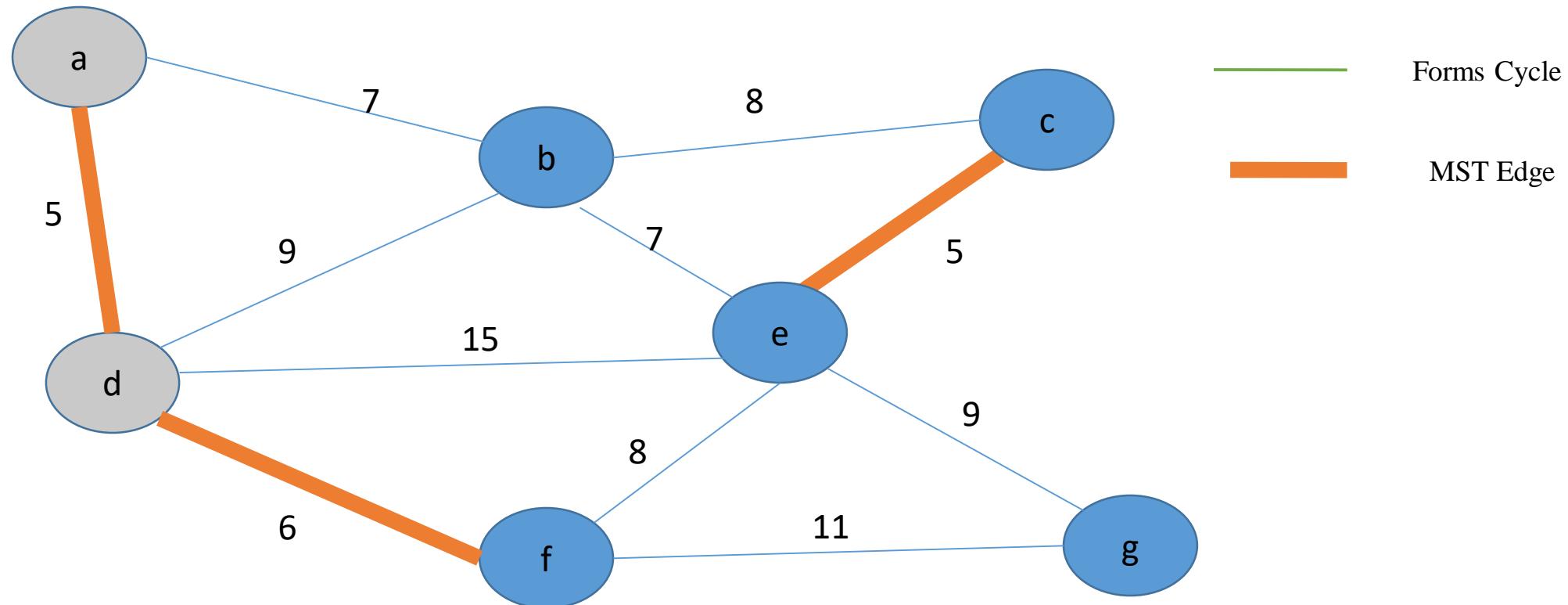
            FIND-SET( $v$ ) **then**

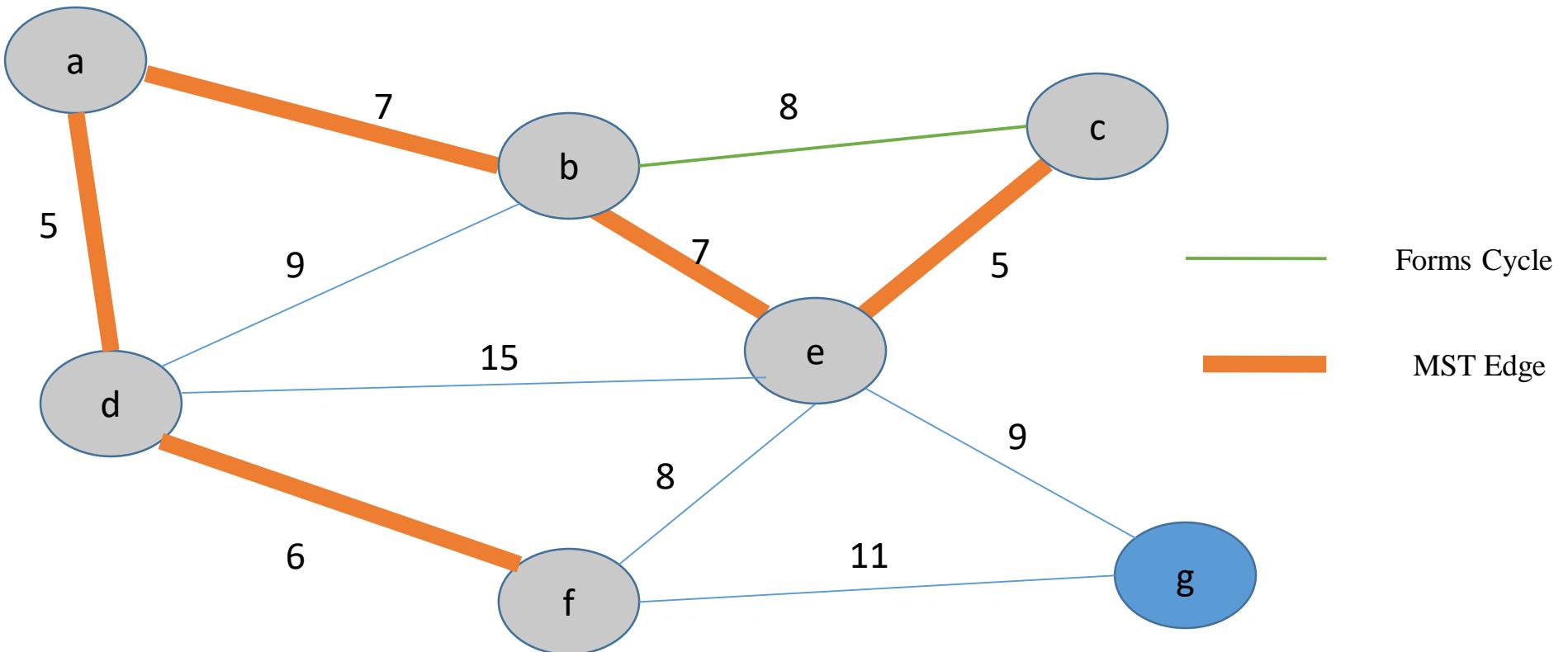
$F := F \cup \{(u, v)\}$  UNION(FIND-SET( $u$ ), FIND-SET( $v$ ))

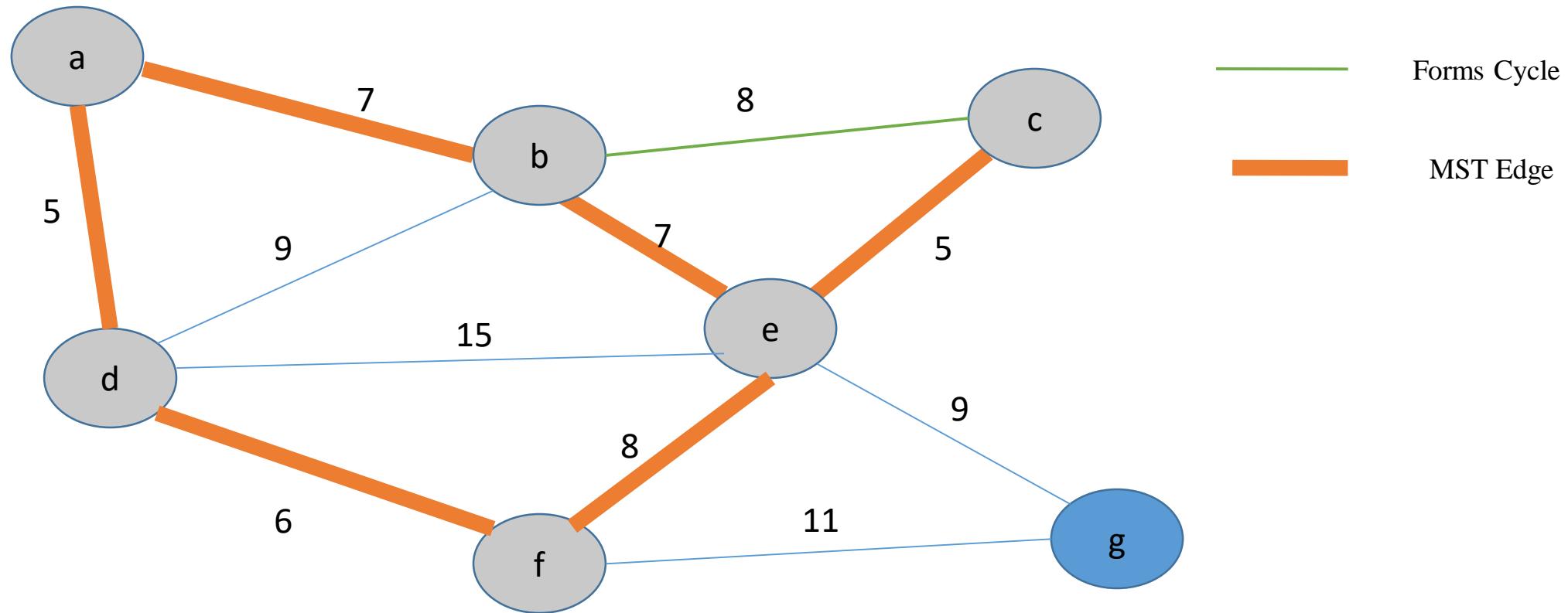
**return**  $F$

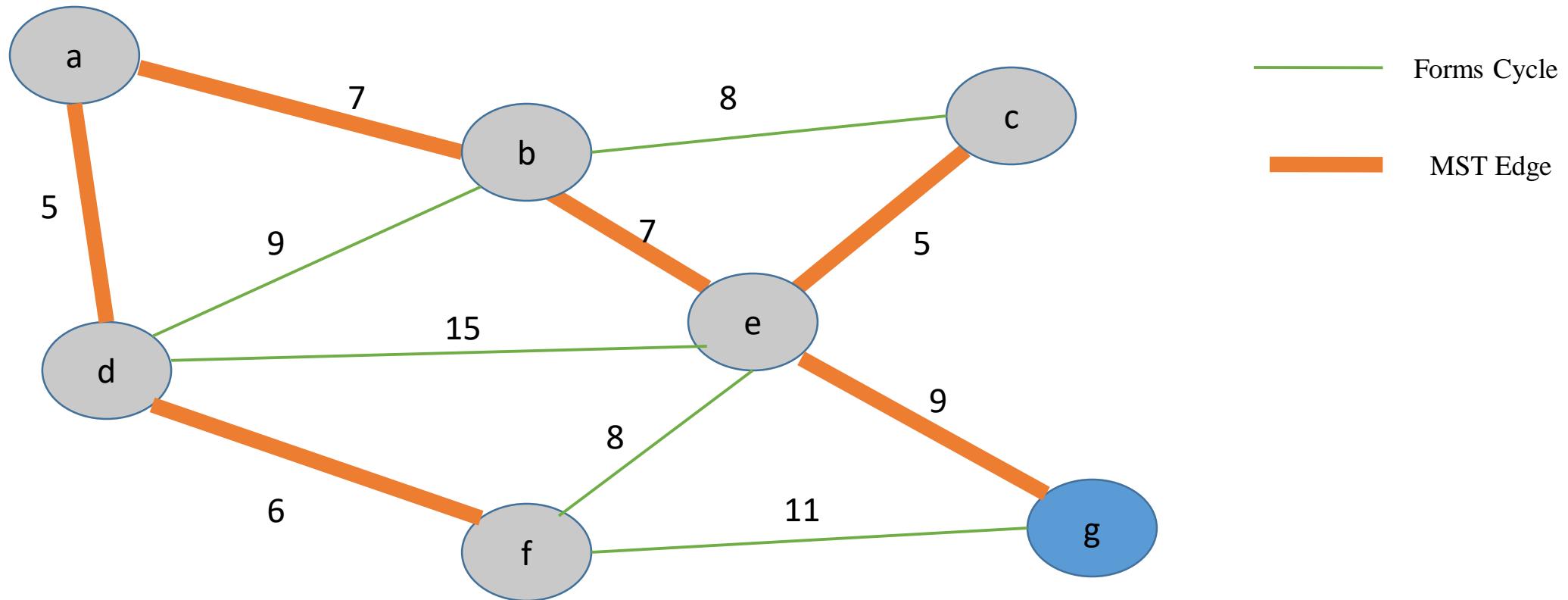
# Example – Kruskal's Algorithm

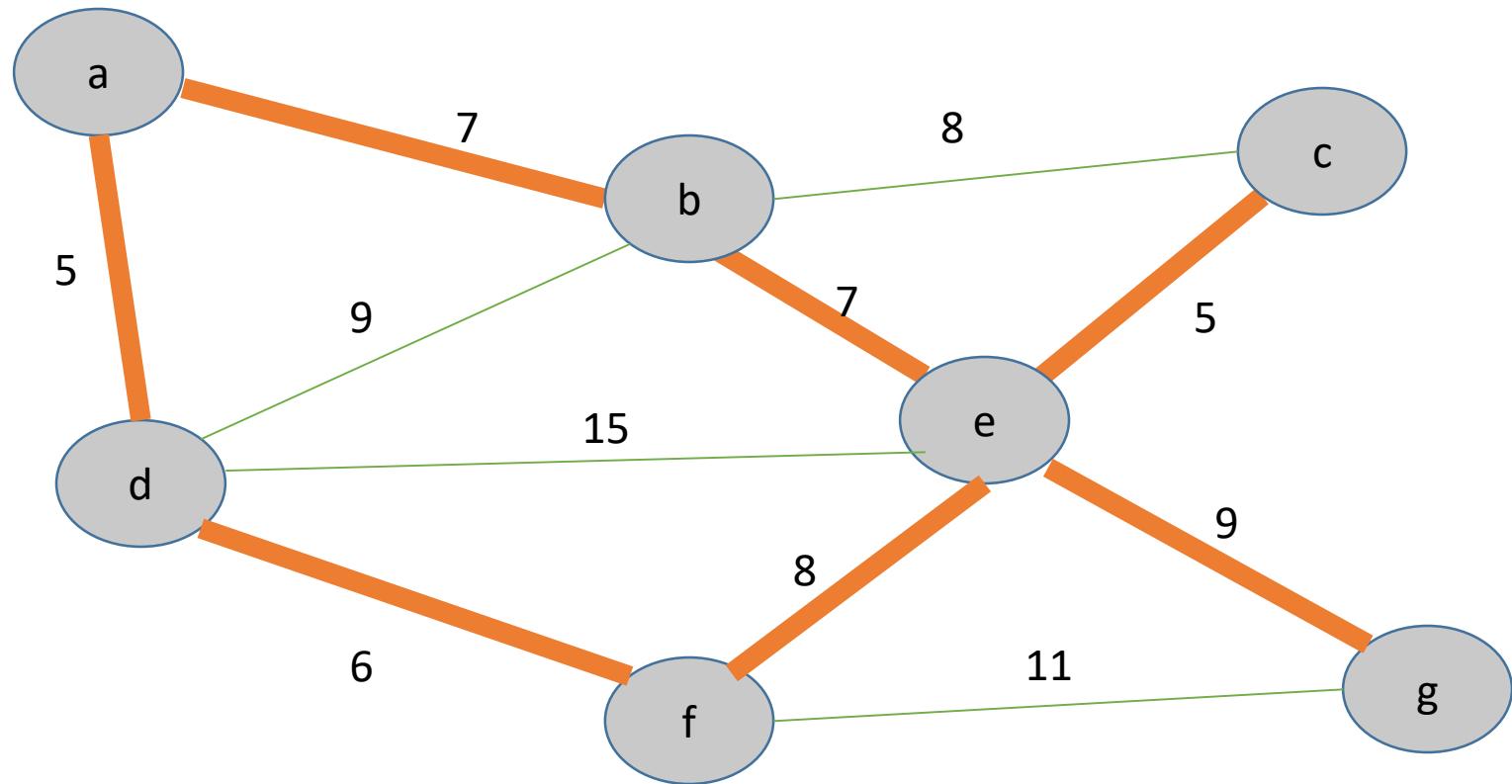












**Cost = 39**

# Time Complexity – Kruskal's Algorithm

- Sorting of edges takes  $O(E \log E)$  time.
- After sorting, we iterate through all edges and **Time Complexity – Kruskal's Algorithm**
- Apply find-union algorithm. The find and union operations can take atmost  $O(\log V)$  time.
- 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  $O(E \log V)$

# Real-time Application – Kruskal's Algorithm

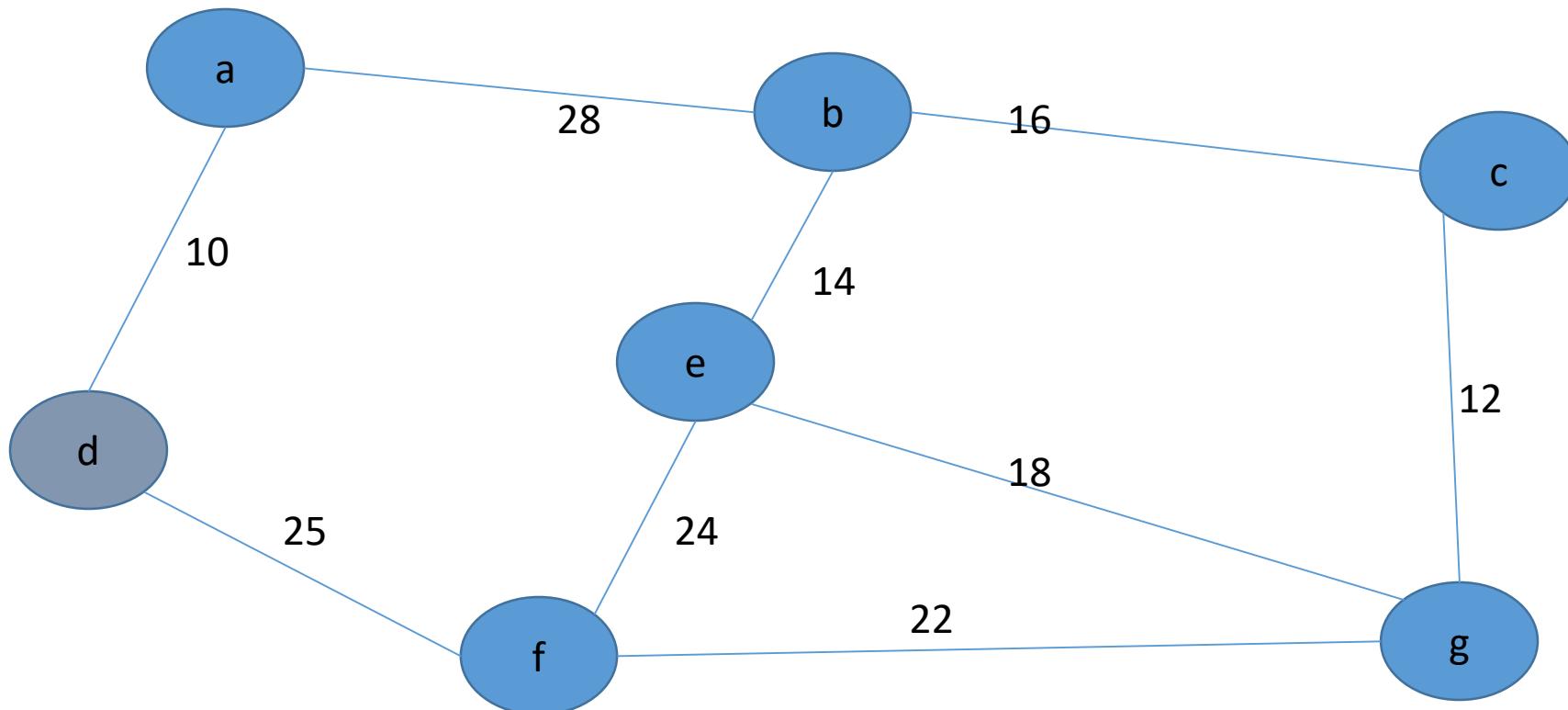
- Landing cables
- TV Network
- Tour Operations
- LAN Networks
- A network of pipes for drinking water or natural gas.
- An electric grid
- Single-link Cluster

# Difference between Prim's and Kruskal's Algorithm

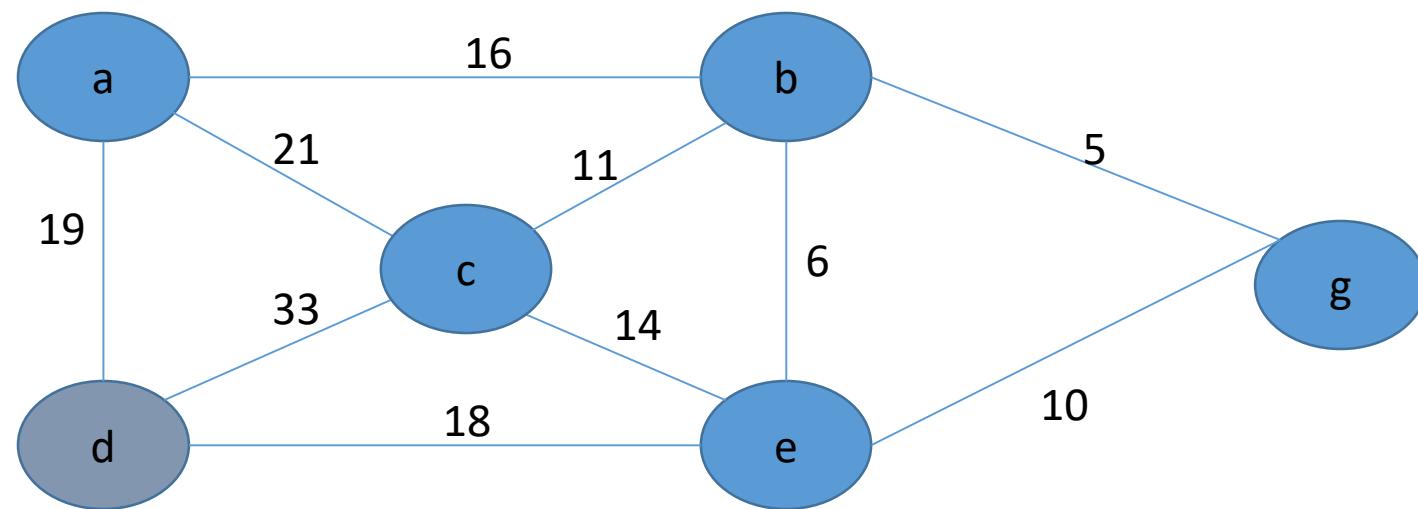
Prim's Algorithm	Kruskal's Algorithm
The tree that we are making or growing always remains connected.	The tree that we are making or growing usually remains disconnected.
Prim's Algorithm grows a solution from a random vertex by adding the next cheapest vertex to the existing tree.	Kruskal's Algorithm 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.

# Activity

- Find the minimum spanning tree of the graph given below using Prim's and Kruskal's Algorithms



2. Find the minimum spanning tree of the graph given below using Prim's and Kruskal's Algorithms



# Summary

- Both algorithms will always give solutions with the same length.
- They will usually select edges in a different order.
- Occasionally they will use different edges – this may happen when you have to choose between edges with the same length.

# Dynamic Programming

# Introduction to Dynamic Programming

## Session Learning Outcome-SLO

- Critically analyze the different algorithm design techniques for a given problem.
- To apply dynamic programming types techniques to solve polynomial time problems.

# What is Dynamic Programming?

- Dynamic programming is a method of solving complex problems by breaking them down into sub-problems that can be solved by working backwards from the last stage.
- Coined by Richard Bellman who described dynamic programming as the way of solving problems where you need to find the best decisions one after another

# Real Time applications

- 0/1 knapsack problem.
- Mathematical optimization problem.
- All pair Shortest path problem.
- Reliability design problem.
- Longest common subsequence (LCS)
- Flight control and robotics control.
- Time sharing: It schedules the job to maximize CPU usage.

# Steps to Dynamic Programming

- Every problem is divided into stages
- Each stage requires a decision
- Decisions are made to determine the state of the next stage
- The solution procedure is to find an optimal solution at each stage for every possible state
- This solution procedure often starts at the last stage and works its way forward

# 0/1 Knapsack Problem

Given  $n$  items of known weights  $w_1, \dots, w_n$  and values  $v_1, \dots, v_n$  and a knapsack of capacity  $W$ , find the most valuable subset of the items that fit into the knapsack.

# Example

- Let's illustrate that point with an example:

Item	Weight	Value
I <sub>0</sub>	3	10
I <sub>1</sub>	8	4
I <sub>2</sub>	9	9
I <sub>3</sub>	8	11

- The maximum weight the knapsack can hold is 20.**
- The best set of items from {I<sub>0</sub>, I<sub>1</sub>, I<sub>2</sub>} is {I<sub>0</sub>, I<sub>1</sub>, I<sub>2</sub>}
- BUT the best set of items from {I<sub>0</sub>, I<sub>1</sub>, I<sub>2</sub>, I<sub>3</sub>} is {I<sub>0</sub>, I<sub>2</sub>, I<sub>3</sub>}.
  - In this example, note that this optimal solution, {I<sub>0</sub>, I<sub>2</sub>, I<sub>3</sub>}, does NOT build upon the previous optimal solution, {I<sub>0</sub>, I<sub>1</sub>, I<sub>2</sub>}.
    - (Instead it builds upon the solution, {I<sub>0</sub>, I<sub>2</sub>}, which is really the optimal subset of {I<sub>0</sub>, I<sub>1</sub>, I<sub>2</sub>} with weight 12 or less.)

## Recursive formula

- Re-work the way that builds upon previous sub-problems
  - Let  $B[k, w]$  represent the maximum total value of a subset  $S_k$  with weight  $w$ .
  - Our goal is to find  $B[n, W]$ , where  $n$  is the total number of items and  $W$  is the maximal weight the knapsack can carry.
- Recursive formula for subproblems:  
$$\begin{aligned} B[k, w] &= B[k - 1, w], \text{ if } w_k > w \\ &= \max \{ B[k - 1, w], B[k - 1, w - w_k] + v_k \}, \text{ otherwise} \end{aligned}$$

# Recursive Formula

$$B[k, w] = \begin{cases} B[k - 1, w] & \text{if } w_k > w \\ \max \{ B[k - 1, w], B[k - 1, w - w_k] + b_k \} & \text{else} \end{cases}$$

- The best subset of  $S_k$  that has the total weight  $w$ , either contains item  $k$  or not.

**First case:**  $w_k > w$

- Item  $k$  can't be part of the solution. If it was the total weight would be  $> w$ , which is unacceptable.

**Second case:**  $w_k \leq w$

- Then the item  $k$  can be in the solution, and we choose the case with greater value.

# Algorithm

```
for w = 0 to W { // Initialize 1st row to 0's
    B[0,w] = 0
}
for i = 1 to n { // Initialize 1st column to 0's
    B[i,0] = 0
}
for i = 1 to n {
    for w = 0 to W {
        if wi <= w { //item i can be in the solution
            if vi + B[i-1,w-wi] > B[i-1,w]
                B[i,w] = vi + B[i-1,w- wi]
            else
                B[i,w] = B[i-1,w]
        }
        else B[i,w] = B[i-1,w] // wi > w
    }
}
```

## Example Problem

Let's run our algorithm on the following data:

- $n = 4$  (# of elements)
- $W = 5$  (max weight)
- Elements (weight, value):  
 $(2,3), (3,4), (4,5), (5,6)$

# Knapsack 0/1 Example

<b>i / w</b>	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
<b>0</b>	0	0	0	0	0	0
<b>1</b>	0					
<b>2</b>	0					
<b>3</b>	0					
<b>4</b>	0					

// Initialize the base cases

for  $w = 0$  to  $W$

$$B[0,w] = 0$$

for  $i = 1$  to  $n$

$$B[i,0] = 0$$

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

$$i = 1$$

$$v_i = 3$$

$$w_i = 2$$

$$w = 1$$

$$w - w_i = -1$$

if  $w_i \leq w$  //item i can be in the solution

if  $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else **B[i, w] = B[i-1, w]** //  $w_i > w$

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

## Knapsack 0/1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					

$$i = 1$$

$$v_i = 3$$

$$w_i = 2$$

$$w = 2$$

$$w - w_i = 0$$

if  $w_i \leq w$  //item i can be in the solution

if  $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

# Knapsack 0/1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3		
2	0					
3	0					
4	0					

$$i = 1$$

$$v_i = 3$$

$$w_i = 2$$

$$w = 3$$

$$w - w_i = 1$$

if  $w_i \leq w$  //item i can be in the solution

if  $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

# Knapsack 0/1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0					
3	0					
4	0					

$$i = 1$$

$$v_i = 3$$

$$w_i = 2$$

$$w = 4$$

$$w - w_i = 2$$

if  $w_i \leq w$  //item i can be in the solution

if  $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

## Knapsack 0/1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

$$i = 1$$

$$v_i = 3$$

$$w_i = 2$$

$$w = 5$$

$$w - w_i = 3$$

if  $w_i \leq w$  //item i can be in the solution

if  $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

# Knapsack 0/1 Example

Items:  
 1: (2,3)  
 2: (3,4)  
 3: (4,5)  
 4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0				
3	0					
4	0					



$$i = 2$$

$$v_i = 4$$

$$w_i = 3$$

$$w = 1$$

$$w - w_i = -2$$

if  $w_i \leq w$  //item i can be in the solution

if  $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

- Items:
- 1: (2,3)
  - 2: (3,4)**
  - 3: (4,5)
  - 4: (5,6)

## Knapsack 0/1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3			
3	0					
4	0					

$$i = 2$$

$$v_i = 4$$

$$w_i = 3$$

$$w = 2$$

$$w - w_i = -1$$

if  $w_i \leq w$  //item i can be in the solution

if  $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else **B[i, w] = B[i-1, w]** //  $w_i > w$

- Items:
- 1: (2,3)
  - 2: (3,4)
  - 3: (4,5)
  - 4: (5,6)

## Knapsack 0/1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4		
3	0					
4	0					

$$i = 2$$

$$v_i = 4$$

$$w_i = 3$$

$$w = 3$$

$$w - w_i = 0$$

if  $w_i \leq w$  //item i can be in the solution

if  $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

Items:  
 1: (2,3)  
 2: (3,4)  
 3: (4,5)  
 4: (5,6)

## Knapsack 0/1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	
3	0					
4	0					

$$\begin{aligned}
 i &= 2 \\
 v_i &= 4 \\
 w_i &= 3 \\
 w &= 4 \\
 w - w_i &= 1
 \end{aligned}$$

if  $w_i \leq w$  //item i can be in the solution

if  $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

- Items:
- 1: (2,3)
  - 2: (3,4)
  - 3: (4,5)
  - 4: (5,6)

## Knapsack 0/1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

$$i = 2$$

$$v_i = 4$$

$$w_i = 3$$

$$w = 5$$

$$w - w_i = 2$$

if  $w_i \leq w$  //item i can be in the solution

if  $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

## Knapsack 0/1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4		
4	0					

$$i = 3$$

$$v_i = 5$$

$$w_i = 4$$

$$w = 1..3$$

$$w - w_i = -3..-1$$

if  $w_i \leq w$  //item i can be in the solution

if  $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

## Knapsack 0/1 Example

i / w	0	1	2	3	4	5
<b>0</b>	0	0	0	0	0	0
<b>1</b>	0	0	3	3	3	3
<b>2</b>	0	0	3	4	4	7
<b>3</b>	0	0	3	4	5	
<b>4</b>	0					

$$i = 3$$

$$v_i = 5$$

$$w_i = 4$$

$$w = 4$$

$$w - w_i = 0$$

if  $w_i \leq w$  //item i can be in the solution

if  $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

Items:

1: (2,3)

2: (3,4)

3: (4,5)  

4: (5,6)

## Knapsack 0/1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

$$i = 3$$

$$v_i = 5$$

$$w_i = 4$$

$$w = 5$$

$$w - w_i = 1$$

if  $w_i \leq w$  //item i can be in the solution

if  $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

Items:  
 1: (2,3)  
 2: (3,4)  
 3: (4,5)  
 4: (5,6)

## Knapsack 0/1 Example

i / w	0	1	2	3	4	5
<b>0</b>	0	0	0	0	0	0
<b>1</b>	0	0	3	3	3	3
<b>2</b>	0	0	3	4	4	7
<b>3</b>	0	0	3	4	5	7
<b>4</b>	0	0	3	4	5	

$$i = 4$$

$$v_i = 6$$

$$w_i = 5$$

$$w = 1..4$$

$$w - w_i = -4..-1$$

if  $w_i \leq w$  //item i can be in the solution

if  $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

- Items:
- 1: (2,3)
  - 2: (3,4)
  - 3: (4,5)
  - 4: (5,6)

## Knapsack 0/1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$$i = 4$$

$$v_i = 6$$

$$w_i = 5$$

$$w = 5$$

$$w - w_i = 0$$

if  $w_i \leq w$  //item i can be in the solution

if  $v_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = v_i + B[i-1, w - w_i]$$

else

$$B[i, w] = B[i-1, w]$$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

- Items:
- 1: (2,3)
  - 2: (3,4)
  - 3: (4,5)
  - 4: (5,6)

## Knapsack 0/1 Example

i / w	0	1	2	3	4	5
<b>0</b>	0	0	0	0	0	0
<b>1</b>	0	0	3	3	3	3
<b>2</b>	0	0	3	4	4	7
<b>3</b>	0	0	3	4	5	7
<b>4</b>	0	0	3	4	5	<b>7</b>

The max possible value that can be carried in this knapsack is \$7

# Knapsack 0/1 Algorithm - Finding the Items

- This algorithm only finds the max possible value that can be carried in the knapsack
  - The value in  $B[n, W]$
- To know the *items* that make this maximum value, we need to trace back through the table.

# Knapsack 0/1 Algorithm

## Finding the Items

- Let  $i = n$  and  $k = W$   
if  $B[i, k] \neq B[i-1, k]$  then  
    mark the  $i^{\text{th}}$  item as in the knapsack  
     $i = i-1, k = k-w_i$   
else  
     $i = i-1$  // Assume the  $i^{\text{th}}$  item is not in the knapsack  
        // Could it be in the optimally packed knapsack?

# Finding the Items

Items:  
 1: (2,3)  
 2: (3,4)  
 3: (4,5)  
 4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i = 4  
 k = 5  
 $v_i = 6$   
 $w_i = 5$   
 $B[i,k] = 7$   
 $B[i-1,k] = 7$

i = n , k = W

while i, k > 0

if  $B[i, k] \neq B[i-1, k]$  then

*mark the  $i^{th}$  item as in the knapsack*

$i = i-1, k = k-w_i$

else

$i = i-1$

# Finding the Items

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$$i = 3$$

$$k = 5$$

$$v_i = 5$$

$$w_i = 4$$

$$B[i,k] = 7$$

$$B[i-1,k] = 7$$

$$i = n, k = W$$

while  $i, k > 0$

if  $B[i, k] \neq B[i-1, k]$  then

*mark the  $i^{th}$  item as in the knapsack*

$$i = i-1, k = k-w_i$$

else

$$i = i-1$$

# Finding the Items

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$$i = 2$$

$$k = 5$$

$$v_i = 4$$

$$w_i = 3$$

$$B[i,k] = 7$$

$$B[i-1,k] = 3$$

$$k - w_i = 2$$

$$i = n, k = W$$

while  $i, k > 0$

if  $B[i, k] \neq B[i-1, k]$  then

*mark the  $i^{th}$  item as in the knapsack*

$$i = i-1, k = k-w_i$$

else

$$i = i-1$$

# Finding the Items

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$$i = n, k = W$$

while  $i, k > 0$

if  $B[i, k] \neq B[i-1, k]$  then

*mark the  $i^{th}$  item as in the knapsack*

$$i = i-1, k = k - w_i$$

else

$$i = i-1$$

$$i = 1$$

$$k = 2$$

$$v_i = 3$$

$$w_i = 2$$

$$B[i, k] = 3$$

$$B[i-1, k] = 0$$

$$k - w_i = 0$$

# Finding the Items

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$$i = 1$$

$$k = 2$$

$$v_i = 3$$

$$w_i = 2$$

$$B[i,k] = 3$$

$$B[i-1,k] = 0$$

$$k - w_i = 0$$

$k = 0$ , it's completed

The optimal knapsack should contain:

*Item 1 and Item 2*

# Complexity calculation of knapsack problem

for  $w = 0$  to  $W$        $O(W)$   
 $B[0,w] = 0$

for  $i = 1$  to  $n$        $O(n)$   
 $B[i,0] = 0$

**Repeat  $n$  times**

for  $i = 1$  to  $n$        $O(W)$   
for  $w = 0$  to  $W$   
< the rest of the code >

What is the running time of this algorithm?  
 $O(n*W)$

# Home Assignment

1. Apply the bottom-up dynamic programming algorithm to the following instance of the knapsack problem. How to find out which items are in optimal subset?

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

, capacity  $W = 6$ .

2. Solve an instance of the knapsack problem by the dynamic programming algorithm.

item	weight	value	
1	2	\$12	
2	1	\$10	
3	3	\$20	
4	2	\$15	capacity $W = 5$ .

# Matrix-chain Multiplication

- Suppose we have a sequence or chain  $A_1, A_2, \dots, A_n$  of  $n$  matrices to be multiplied
  - That is, we want to compute the product  $A_1 A_2 \dots A_n$
- There are many possible ways (parenthesizations) to compute the product

# Matrix-chain Multiplication

...contd

- Example: consider the chain  $A_1, A_2, A_3, A_4$  of 4 matrices
  - Let us compute the product  $A_1A_2A_3A_4$
  - There are 5 possible ways:
    1.  $(A_1(A_2(A_3A_4)))$
    2.  $(A_1((A_2A_3)A_4))$
    3.  $((A_1A_2)(A_3A_4))$
    4.  $((A_1(A_2A_3))A_4)$
    5.  $(((A_1A_2)A_3)A_4)$

# Matrix-chain Multiplication

...contd

- To compute the number of scalar multiplications necessary, we must know:
  - Algorithm to multiply two matrices
  - Matrix dimensions
- Can you write the algorithm to multiply two matrices?

# Algorithm to Multiply 2 Matrices

**Input:** Matrices  $A_{p \times q}$  and  $B_{q \times r}$  (with dimensions  $p \times q$  and  $q \times r$ )

**Result:** Matrix  $C_{p \times r}$  resulting from the product  $A \cdot B$

**MATRIX-MULTIPLY**( $A_{p \times q}$ ,  $B_{q \times r}$ )

1.   **for**  $i \leftarrow 1$  **to**  $p$
2.       **for**  $j \leftarrow 1$  **to**  $r$
3.            $C[i, j] \leftarrow 0$
4.       **for**  $k \leftarrow 1$  **to**  $q$
5.            $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
6.   **return**  $C$

Scalar multiplication in line 5 dominates time to compute C  
Number of scalar multiplications =  $pqr$

# Matrix-chain Multiplication

...contd

- Example: Consider three matrices  $A_{10 \times 100}$ ,  $B_{100 \times 5}$ , and  $C_{5 \times 50}$
- There are 2 ways to parenthesize
  - $((AB)C) = D_{10 \times 5} \cdot C_{5 \times 50}$ 
    - $AB \Rightarrow 10 \cdot 100 \cdot 5 = 5,000$  scalar multiplications
    - $DC \Rightarrow 10 \cdot 5 \cdot 50 = 2,500$  scalar multiplications
  - $(A(BC)) = A_{10 \times 100} \cdot E_{100 \times 50}$ 
    - $BC \Rightarrow 100 \cdot 5 \cdot 50 = 25,000$  scalar multiplications
    - $AE \Rightarrow 10 \cdot 100 \cdot 50 = 50,000$  scalar multiplications

Total:  
7,500

Total:  
75,000

# Matrix-chain Multiplication

...contd

- Matrix-chain multiplication problem
  - Given a chain  $A_1, A_2, \dots, A_n$  of  $n$  matrices, where for  $i=1, 2, \dots, n$ , matrix  $A_i$  has dimension  $p_{i-1} \times p_i$
  - Parenthesize the product  $A_1A_2\dots A_n$  such that the total number of scalar multiplications is minimized
- Brute force method of exhaustive search takes time exponential in  $n$

# Dynamic Programming Approach

- The structure of an optimal solution
  - Let us use the notation  $A_{i..j}$  for the matrix that results from the product  $A_i A_{i+1} \dots A_j$
  - An optimal parenthesization of the product  $A_1 A_2 \dots A_n$  splits the product between  $A_k$  and  $A_{k+1}$  for some integer  $k$  where  $1 \leq k < n$
  - First compute matrices  $A_{1..k}$  and  $A_{k+1..n}$ ; then multiply them to get the final matrix  $A_{1..n}$

# Dynamic Programming Approach ...contd

- **Key observation:** parenthesizations of the subchains  $A_1A_2\dots A_k$  and  $A_{k+1}A_{k+2}\dots A_n$  must also be optimal if the parenthesization of the chain  $A_1A_2\dots A_n$  is optimal (why?)
- That is, the optimal solution to the problem contains within it the optimal solution to subproblems

# Dynamic Programming Approach ...contd

- Recursive definition of the value of an optimal solution
  - Let  $m[i, j]$  be the minimum number of scalar multiplications necessary to compute  $A_{i..j}$
  - Minimum cost to compute  $A_{1..n}$  is  $m[1, n]$
  - Suppose the optimal parenthesization of  $A_{i..j}$  splits the product between  $A_k$  and  $A_{k+1}$  for some integer  $k$  where  $i \leq k < j$

# Dynamic Programming Approach ...contd

- $A_{i..j} = (A_i A_{i+1} \dots A_k) \cdot (A_{k+1} A_{k+2} \dots A_j) = A_{i..k} \cdot A_{k+1..j}$
- Cost of computing  $A_{i..j}$  = cost of computing  $A_{i..k}$  + cost of computing  $A_{k+1..j}$  + cost of multiplying  $A_{i..k}$  and  $A_{k+1..j}$
- Cost of multiplying  $A_{i..k}$  and  $A_{k+1..j}$  is  $p_{i-1} p_k p_j$
- $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j \quad \text{for } i \leq k < j$
- $m[i, i] = 0$  for  $i=1,2,\dots,n$

# Dynamic Programming Approach ...contd

- But... optimal parenthesization occurs at one value of k among all possible  $i \leq k < j$
- Check all these and select the best one

$$m[i, j] = \begin{cases} 0 & \text{if } i=j \\ \min \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{if } i < j \\ i \leq k < j \end{cases}$$

# Dynamic Programming Approach ...contd

- To keep track of how to construct an optimal solution, we use a table  $s$
- $s[i, j] = \text{value of } k \text{ at which } A_i A_{i+1} \dots A_j \text{ is split for optimal parenthesization}$
- Algorithm: next slide
  - First computes costs for chains of length  $l=1$
  - Then for chains of length  $l=2, 3, \dots$  and so on
  - Computes the optimal cost bottom-up

# Algorithm to Compute Optimal Cost

**Input:** Array  $p[0 \dots n]$  containing matrix dimensions and  $n$

**Result:** Minimum-cost table  $m$  and split table  $s$

**MATRIX-CHAIN-ORDER**( $p[ ]$ ,  $n$ )

```

for  $i \leftarrow 1$  to  $n$ 
     $m[i, i] \leftarrow 0$ 
for  $l \leftarrow 2$  to  $n$ 
    for  $i \leftarrow 1$  to  $n-l+1$ 
         $j \leftarrow i+l-1$ 
         $m[i, j] \leftarrow \infty$ 
        for  $k \leftarrow i$  to  $j-1$ 
             $q \leftarrow m[i, k] + m[k+1, j] + p[i-1] p[k] p[j]$ 
            if  $q < m[i, j]$ 
                 $m[i, j] \leftarrow q$ 
                 $s[i, j] \leftarrow k$ 
return  $m$  and  $s$ 

```

Takes  $O(n^3)$  time

Requires  $O(n^2)$  space

# Constructing Optimal Solution

- Our algorithm computes the minimum-cost table  $m$  and the split table  $s$
- The optimal solution can be constructed from the split table  $s$ 
  - Each entry  $s[i, j] = k$  shows where to split the product  $A_i A_{i+1} \dots A_j$  for the minimum cost

# Example

- Show how to multiply this matrix chain optimally
- Solution on the board
  - Minimum cost 15,125
  - Optimal parenthesization  
 $((A_1(A_2A_3))((A_4 A_5)A_6))$



parenthesization

Matrix	Dimension
$A_1$	$30 \times 35$
$A_2$	$35 \times 15$
$A_3$	$15 \times 5$
$A_4$	$5 \times 10$
$A_5$	$10 \times 20$
$A_6$	$20 \times 25$



**SRM**  
INSTITUTE OF SCIENCE & TECHNOLOGY  
(Dedicated to be University w/o 3rd year UGC Act 1956)

# Longest Common Subsequence(LCS)

## Dynamic Programming

# LCS using Dynamic Programming

- SLO 1: To understand the Longest Common Subsequence(LCS) Problem and how it can be solved using Dynamic programming approach.

# LCS Problem Statement

- Given two sequences, find the length of longest subsequence present in both of them.

# Example

- A **subsequence** is a sequence that appears in the same relative order, but not necessarily contiguous.
- Let String S1= a b c d
- Subsequence

- |    |    |               |    |    |               |    |     |     |      |
|----|----|---------------|----|----|---------------|----|-----|-----|------|
| ab | bd | <del>ca</del> | ac | ad | <del>db</del> | ba | acd | bcd | abcd |
|----|----|---------------|----|----|---------------|----|-----|-----|------|
- Length of S1 = 4
  - Subsequence : abcd

# LCS Example

- $S_1 = a \ b \ c \ d \ e \ f \ g \ h \ i \ j$

- $S_2 = c \ d \ g \ i$

- Subsequence

- $S_1:$ 

a	b	c	d	e	f	g	h	i	j
---	---	---	---	---	---	---	---	---	---

- $S_2:$ 

c	d	g	i
---	---	---	---

- Subsequence, SST: c d g i

# Example

- $S_1 = a \ b \ c \ d \ e \ f \ g \ h \ i \ j$

- $S_2 = c \ d \ g \ i$

- Subsequence

- $S_1:$ 

a	b	c	d	e	f	g	h	i	j
---	---	---	---	---	---	---	---	---	---

- $S_2:$ 

c	d	g	i
---	---	---	---

- Subsequence,  $SS_2: d \ g \ i$

# Example

- $S_1 = a \ b \ c \ d \ e \ f \ g \ h \ i \ j$

- $S_2 = c \ d \ g \ i$

- Subsequence

- $S_1:$ 

a	b	c	d	e	f	g	h	i	j
---	---	---	---	---	---	---	---	---	---

- $S_2:$ 

c	d	g	i
---	---	---	---

- Subsequence, SS3: g i

# Example

- SS1= c d g i
- SS2: d g i
- SS3: g i
- **Longest Common Subsequence:** subsequence with maximum length.

maximum(length(subsequence1),  
length(subsequence2)....  
length(subsequence n)).

## Longest Common Subsequence :

maximum(length(c d g i), length(d g i),length(g i)).

- Therefore LCS (S1,S2) is **c d g i** with **length 4**

# Activity

- Give the LCS for the following strings
- X: ABCBDAB
- Y: BDCABA

# Motivation

**Brute Force Approach**-Check every subsequence of  $x[1 \dots m]$  to see if it is also a subsequence of  $y[1 \dots n]$ .

## Analysis

- $2^m$  subsequences of  $x$  (each bit-vector of length  $m$  determines a distinct subsequence of  $x$ ).
- Hence, the runtime would be exponential !

Example:

Consider S1 = ACCGGTCGAGTGCGCGGAAGGCCGGCCGAA.....

S2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA....

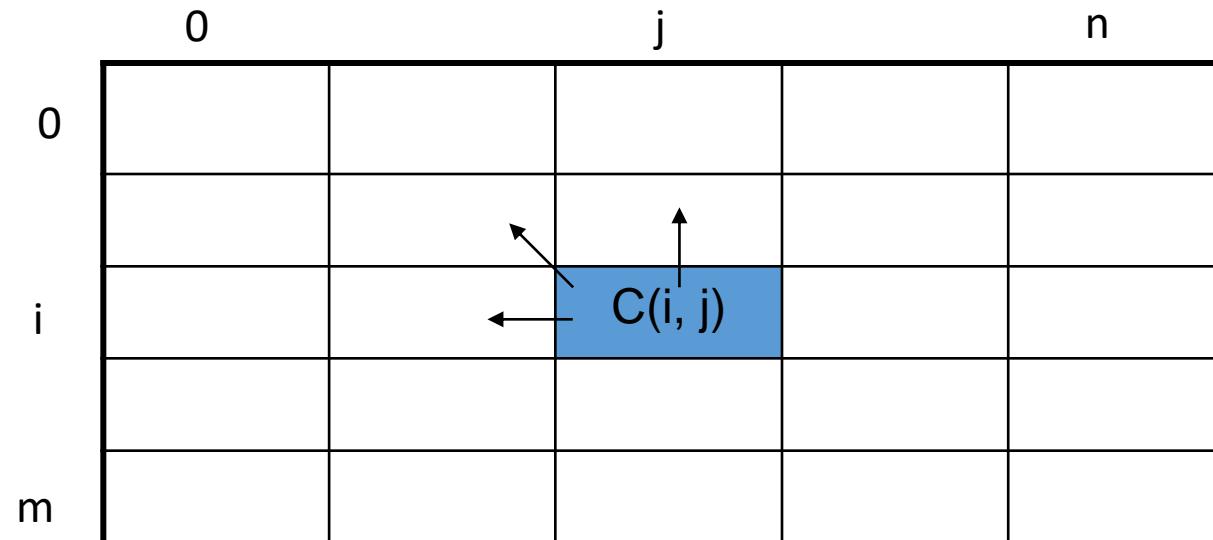
## Towards a better algorithm: a Dynamic Programming strategy

- Key: optimal substructure and overlapping sub-problems
- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.

# LCS Problem Formulation using DP Algorithm

- Key: find out the correct order to solve the sub-problems
- Total number of sub-problems:  $m * n$

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max \{ c[i-1, j], c[i, j-1] \} & \text{otherwise.} \end{cases}$$



# LCS Example

We'll see how LCS algorithm works on the following example:

- $X = ABCB$
- $Y = BDCAB$

What is the LCS of X and Y?

$$\text{LCS}(X, Y) = BCB$$

$X = A \textcolor{red}{B} \textcolor{blue}{C} \textcolor{magenta}{B}$

$Y = \textcolor{red}{B} D \textcolor{blue}{C} A \textcolor{magenta}{B}$

# Computing the Length of the LCS

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

		0	1	2	n	
		$x_0$	$y_1$	$y_2$	$y_n$	
0	$x_i$	0	0	0	0	0
	$x_1$	0				
1	$x_2$	0				
		0				
		0				
		0				
m	$x_m$	0				
		j				

first  
second  
i

# LCS Example (0)

ABCB  
BDCAB

		j	0	1	2	3	4	5
i		Y[j]	B	D	C	A	B	
0	X[i]							
1	A							
2	B							
3	C							
4	B							

$$X = ABCB; \ m = |X| = 4$$

$$Y = BDCAB; \ n = |Y| = 5$$

Allocate array c[5,6]

# LCS Example (1)

ABCB  
BDCAB

		j	0	1	2	3	4	5
i		Y[j]	B	D	C	A	B	
		X[i]	0	0	0	0	0	0
0		A	0					
1		B	0					
2		C	0					
3		B	0					

for  $i = 1$  to  $m$   
for  $j = 1$  to  $n$

$c[i,0] = 0$   
 $c[0,j] = 0$

# LCS Example (2)

ABC  
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	B	D	C	A	B
0	X[i]	0	0	0	0	0	0
1	A	0	0				
2	B	0					
3	C	0					
4	B	0					

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

# LCS Example (3)

ABCB  
BDCAB



**SRM**  
INSTITUTE OF SCIENCE & TECHNOLOGY  
(Deemed to be University w.e.f. UGC Act. 1956)

		j	0	1	2	3	4	5
i		Y[j]	B	D	C	A	B	
0	X[i]	0	0	0	0	0	0	0
1	A	0	0	0	0			
2	B	0						
3	C	0						
4	B	0						

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

# LCS Example (4)

ABCB  
BDC**A**B  
**4**



i	j	0	1	2	3	4	5	
		Y[j]	B	D	C			
0	X[i]	0	0	0	0	0	0	
1	<b>A</b>	0	0	0	0	1		
2	<b>B</b>	0						
3	<b>C</b>	0						
4	<b>B</b>	0						

if ( $X_i == Y_j$ )  
 $c[i,j] = c[i-1,j-1] + 1$   
else  $c[i,j] = \max(c[i-1,j], c[i,j-1])$

# LCS Example (5)

**A**BCB  
**B**DCAB



	j	0	1	2	3	4	5	
i		Y[j]	B	D	C	A		
	X[i]	0	0	0	0	0	0	
0		0	0	0	0	0	0	
1	<b>A</b>	0	0	0	0	1	→ 1	
2	<b>B</b>	0						
3	<b>C</b>	0						
4	<b>B</b>	0						

if (  $X_i == Y_j$  )  
 $c[i,j] = c[i-1,j-1] + 1$   
 else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$

# LCS Example (6)

A B C B  
B D C A B



**SRM**  
INSTITUTE OF SCIENCE & TECHNOLOGY  
(Deemed to be University w.e.f. 1998)

i	j	0	1	2	3	4	5
		Y[j]	B	D	C	A	B
0	X[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1				
3	C	0					
4	B	0					

if ( $X_i == Y_j$ )  
 $c[i,j] = c[i-1,j-1] + 1$   
else  $c[i,j] = \max(c[i-1,j], c[i,j-1])$

# LCS Example (7)

ABCB  
BDCAB



**SRM**  
INSTITUTE OF SCIENCE & TECHNOLOGY  
(Deemed to be University w.e.f. 1998)

i	j	0	1	2	3	4	5	
		Y[j]	B	D	C	A	B	
0	X[i]	0	0	0	0	0	0	
1	A	0	0	0	0	1	1	
2	B	0	1	1	1	1	1	
3	C	0						
4	B	0						

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

# LCS Example (8)

ABCB  
BDCAB  
**5**



**SRM**  
INSTITUTE OF SCIENCE & TECHNOLOGY  
(Deemed to be University w.e.f. 1998)

i	j	0	1	2	3	4	5
		Y[j]	B	D	C	A	B
0	X[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0					
4	B	0					

if ( $X_i == Y_j$ )  
 $c[i,j] = c[i-1,j-1] + 1$   
 else  $c[i,j] = \max(c[i-1,j], c[i,j-1])$

# LCS Example (9)

ABC  
BDCAB



**SRM**  
INSTITUTE OF SCIENCE & TECHNOLOGY  
(Deemed to be University w.e.f. 1998)

i	j	0	1	2	3	4	5	
		Y[j]	B	D		C	A	B
0	X[i]	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	
2	B	0	1	1	1	1	1	2
3	C	0	1	1				
4	B	0						

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

# LCS Example (10)

i	j	0	1	2	3	4	5	
		Y[j]	B	D		C	A	B
0	X[i]	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	
2	B	0	1	1		1	1	2
3	C	0	1	1		2		
4	B	0						

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

# LCS Example (11)

i	j	0	1	2	3	4	5	A B
		Y[j]	B	D	C			
0	X[i]	0	0	0	0	0	0	
1	A	0	0	0	0	1	1	
2	B	0	1	1	1	1		2
3	C	0	1	1	2	2	2	
4	B	0						

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

# LCS Example (12)

ABCB  
BDCAB



**SRM**  
INSTITUTE OF SCIENCE & TECHNOLOGY  
(Deemed to be University w.e.f. 1998)

i	j	0	1	2	3	4	5	
	Y[j]		B		D	C	A	B
0	X[i]	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	
2	B	0	1	1	1	1	2	
3	C	0	1	1	2	2	2	
4	B	0	1					

if ( $X_i == Y_j$ )  
 $c[i,j] = c[i-1,j-1] + 1$   
 else  $c[i,j] = \max(c[i-1,j], c[i,j-1])$

# LCS Example (13)

i	j	0	1	2	3	4	5
	Y[j]	B	D	C	A	B	
0	X[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

# LCS Example (14)

ABCB  
BDCAB  
5



i	j	0	1	2	3	4	5
		Y[j]	B	D	C	A	B
0	X[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

# Finding LCS

i	j	0	1	2	3	4	5
		Y[j]	B	D	C	A	B
0	X[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

Time for trace back:  $O(m+n)$ .

# Finding LCS (2)(Bottom Up approach)

i	j	0	1	2	3	4	5
		Y[j]	B	D	C	A	B
0	X[i]	0 0 0 0 0 0					
1	A	0 0 0 0 1 1					
2	B	0 1 1 1 1 2	1 ← 1	1 ← 2	2 ← 2	2 ← 3	3
3	C	0 1 1 2 2 2					
4	B	0 1 1 2 2 3					

B      C      B

LCS: BCB

# Algorithm steps for solving LCS problem using Dynamic Programming

- Step 1: Characterizing a longest common subsequence
- Step 2: A recursive solution
- Step 3: Computing the length of an LCS
- Step 4: Constructing an LCS

# Step 1: Characterizing a longest common subsequence

- Optimal substructure of LCS: a problem is said to have **optimal substructure** if an optimal solution can be constructed from optimal solutions of its sub problems.

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 let  $Z = \langle z_1, z_2, \dots, z_k \rangle$  be any LCS of  $X$  and  $Y$ .

1. If  $x_m = y_n$ , then  $z_k = x_m = y_n$  and  $Z_{k-1}$  is an LCS of  $X_{m-1}$  and  $Y_{n-1}$ .
2. If  $x_m \neq y_n$ , then  $z_k \neq x_m$  implies that  $Z$  is an LCS of  $X_{m-1}$  and  $Y$ .
3. If  $x_m \neq y_n$ , then  $z_k \neq y_n$  implies that  $Z$  is an LCS of  $X$  and  $Y_{n-1}$ .

## Step 2: A recursive solution

- Optimal substructure of LCS problem gives the recursive formula:

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

# Step 3: Computing the length of an LCS

**LCS-LENGTH( $X, Y$ )**

```

1    $m = X.length$ 
2    $n = Y.length$ 
3   let  $b[1..m, 1..n]$  and  $c[0..m, 0..n]$  be new tables
4   for  $i = 1$  to  $m$ 
5        $c[i, 0] = 0$ 
6   for  $j = 0$  to  $n$ 
7        $c[0, j] = 0$ 
8   for  $i = 1$  to  $m$ 
9       for  $j = 1$  to  $n$ 
10      if  $x_i == y_j$ 
11           $c[i, j] = c[i - 1, j - 1] + 1$ 
12           $b[i, j] = "\searrow"$ 
13      elseif  $c[i - 1, j] \geq c[i, j - 1]$ 
14           $c[i, j] = c[i - 1, j]$ 
15           $b[i, j] = "\uparrow"$ 
16      else  $c[i, j] = c[i, j - 1]$ 
17           $b[i, j] = "\leftarrow"$ 
18  return  $c$  and  $b$ 
```

# Step 4: Constructing an LCS

```

PRINT-LCS( $b, X, i, j$ )
1  if  $i == 0$  or  $j == 0$ 
    return
2  if  $b[i, j] == \nwarrow$ 
    PRINT-LCS( $b, X, i - 1, j - 1$ )
    print  $x_i$ 
6  elseif  $b[i, j] == \uparrow\uparrow$ 
    PRINT-LCS( $b, X, i - 1, j$ )
8  else PRINT-LCS( $b, X, i, j - 1$ )

```

	$j$	0	1	2	3	4	5	6
$i$	$y_j$	<b>B</b>	D	C	A	B	A	
0	$x_i$	0	0	0	0	0	0	0
1	A	0	0	0	0	1	-1	1
2	B	0	1	-1	-1	1	2	-2
3	C	0	1	1	2	-2	2	2
4	B	0	1	1	2	2	3	-3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

# Time Complexity Analysis of LCS Algorithm

- LCS algorithm calculates the values of each entry of the array  $c[m,n]$
- So what is the running time?

**O( $m \cdot n$ )**

since each  $c[i,j]$  is calculated in constant time, and there are  $m \cdot n$  elements in the array

## Asymptotic Analysis:

Worst case time complexity:  $O(n \cdot m)$

Average case time complexity:  $O(n \cdot m)$

Best case time complexity:  $O(n \cdot m)$

Space complexity:  $O(n \cdot m)$

# Real Time Application

- **Molecular biology.** DNA sequences (genes) can be represented as sequences of four letters ACGT, corresponding to the four sub molecules forming DNA.
  - To find a new sequences by generating the LCS of existing similar sequences
  - To find the similarity of two DNA Sequence by finding the length of their longest common subsequence.
- **File comparison.** The Unix program "diff" is used to compare two different versions of the same file, to determine what changes have been made to the file.
  - It works by finding a longest common subsequence of the lines of the two files; any line in the subsequence has not been changed, so what it displays is the remaining set of lines that have changed.
- **Screen redisplay.** Many text editors like "emacs" display part of a file on the screen, updating the screen image as the file is changed.
  - Can be a sort of common subsequence problem (the common subsequence tells you the parts of the display that are already correct and don't need to be changed)

# Summary

- Solving LCS problem by brute force approach requires  $O(2^m)$ .
- Applying Dynamic Programming to solve LCS problem reduces the time complexity to  $O(nm)$ .

# Review Questions

1. What is the time complexity of the brute force algorithm used to find the longest common subsequence?
2. What is the time complexity of the dynamic programming algorithm used to find the longest common subsequence?
3. If  $X[i]==Y[i]$  what is the value stored in  $c[i,j]$ ?
4. If  $X[i]!=Y[i]$  what is the value stored in  $c[i,j]$ ?
5. What is the value stored in zeroth row and zeroth column of the LCS table?

# Home Assignment

- Determine the LCS of  $(1,0,0,1,0,1,0,1)$  and  $(0,1,0,1,1,0,1,1,0)$ .

# OPTIMAL BINARY SEARCH TREE

- Session Learning Outcome-SLO
- Motivation of the topic
- Binary Search Tree
- Optimal Binary Search Tree
- Example Problem
- Analysis
- Summary
- Activity /Home assignment /Questions

# INTRODUCTION

## Session Learning Outcome:

- To Understand the concept of Binary Search tree and Optimal Binary Search Tree
- how to construct Optimal Binary Search Tree with optimal cost

# OPTIMAL BINARY SEARCH TREE(OBST)

- Binary Search tree(BST) which is mainly constructed for searching a key from it
- For searching any key from a given BST, it should take optimal time.
- For this, we need to construct a BST in such a way that it should take optimal time to search any of the key from given BST
- To construct OBST, frequency of searching of every key is required
- With the help of frequencies , construction of OBST is possible

## BINARY SEARCH TREE:

A binary search tree is a special kind of binary tree. In binary search tree, the elements in the left and right sub-trees of each node are respectively lesser and greater than the element of that node. Fig. 1 shows a binary search tree.

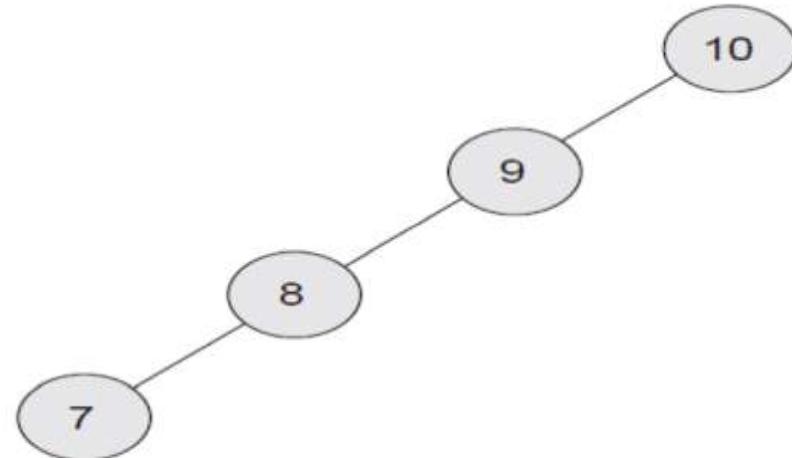


Fig. 1: Skewed Binary Search Tree

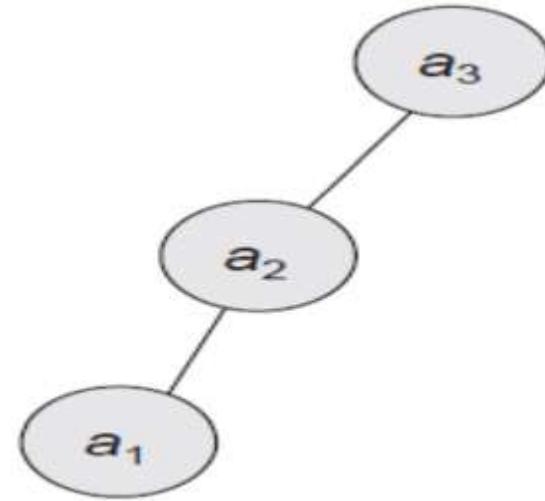
Fig. 1 is a binary search tree but is not balanced tree. On the other hand, this is a skewed tree where all the branches on one side. The advantage of binary search tree is that it facilitates search of a key easily. It takes  $O(n)$  to search for a key in a list. Whereas, search tree helps to find an element in logarithmic time.

## How an element is searched in binary search tree?

- Let us assume the given element is  $x$ . Compare  $x$  with the root element of the binary tree, if the binary tree is non-empty. If it matches, the element is in the root and the algorithm terminates successfully by returning the address of the root node. If the binary tree is empty, it returns a NULL value.
  - If  $x$  is less than the element in the root, the search continues in the left sub-tree
  - If  $x$  is greater than the element in the root, the search continues in the right sub-tree.
- This is by exploiting the binary search property. There are many applications of binary search trees. One application is construction of dictionary.
- There are many ways of constructing the binary search tree. Brute force algorithm is to construct many binary search trees and finding the cost of the tree. How to find cost of the tree? The cost of the tree is obtained by multiplying the probability of the item and the level of the tree. The following example illustrates the way of find this cost of the tree.

## HOW TO FIND THE COST OF BINARY SEARCH TREE

**Example 1:** Find the cost of the tree shown in Fig. 2 where the items probability is given as follows:  
 $a_1 = 0.4$  ,  $a_2 = 0.3$  ,  $a_3 = 0.3$



**Fig. 2: Sample Binary Search Tree**

### Solution

As discussed earlier, the cost of the tree is obtained by multiplying the item probability and the level of the tree. The cost of the tree is computed as follows;

$a_1$  level=3,  $a_2$  level=2,  $a_3$  level=1

$$\text{Cost of BST} = 3(0.4) + 2(0.3) + 1(0.3) = 2.1$$

It can be observed that the cost of the tree is 2.1.

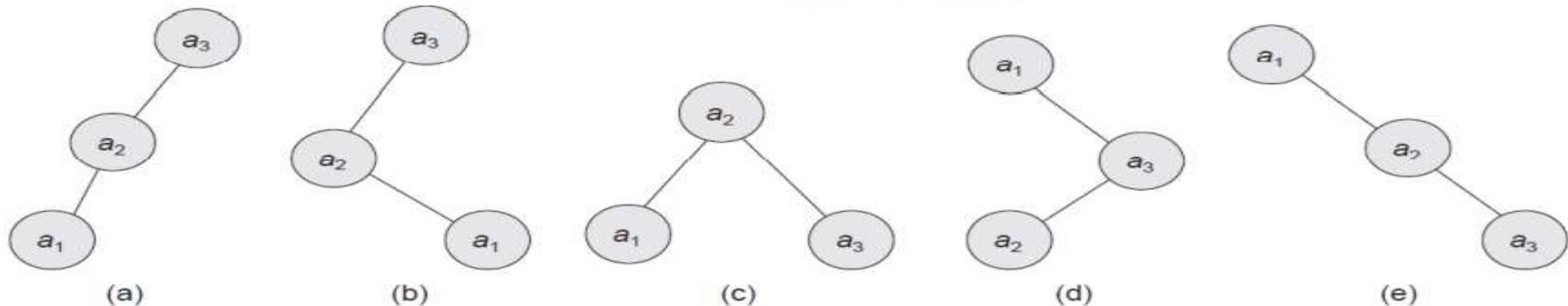
## OPTIMAL BINARY SEARCH TREE:

- What is an optimal binary search tree? An optimal binary search tree is a tree of optimal cost. This is illustrated in the following example

**Example 2: Construct optimal binary search tree for the three items  $a_1 = 0.4$ ,  $a_2 = 0.3$ ,  $a_3 = 0.3$  ?**

**Solution:**

- There are many ways one can construct binary search trees. Some of the constructed binary search trees and its cost are shown in Fig. 3.



**Fig. 3. : Some of the binary search trees**

- It can be seen the cost of the trees are respectively, 2.1, 1.3, 1.6, 1.9 and 1.9. So the minimum cost is 1.3. Hence, the optimal binary search tree is (b) Fig. 3.

## OPTIMAL BINARY SEARCH TREE(Contd..)

**How to construct optimal binary search tree? The problem of optimal binary search tree is given as follows:**

- Given sequence  $K = k_1 < k_2 < \dots < k_n$  of  $n$  sorted keys, with a search probability  $p_i$  for each key  $k_i$ . The aim is to build a binary search tree with minimum expected cost.
- One way is to use brute force method, by exploring all possible ways and finding the expected cost. But the method is not practical as the number of trees possible is Catalan sequence. The Catalan number is given as follows:

$$c(n) = \binom{2n}{n} \frac{1}{n+1} \text{ for } n > 0, \quad c(0) = 1.$$

If the nodes are 3, then the Catalan number is

$$C_3 = \frac{1}{3+1} \binom{6}{3} = 5$$

Hence, five search trees are possible. In general,  $\Omega(4^n/n^{3/2})$  different BSTs are possible with n nodes. Hence, alternative way is to explore dynamic programming approach.

**Dynamic programming Approach:**

- The idea is to One of the keys in  $a_1, \dots, a_n$ , say  $a_k$ , where  $1 \leq k \leq n$ , must be the root. Then , as per binary search rule, Left sub tree of  $a_k$  contains  $a_1, \dots, a_{k-1}$  and right subtree of  $a_k$  contains  $a_{k+1}, \dots, a_n$ .
- So, the idea is to examine all candidate roots  $a_k$  , for  $1 \leq k \leq n$  and determining all optimal BSTs containing  $a_1, \dots, a_{k-1}$  and containing  $a_{k+1}, \dots, a_n$

The informal algorithm for constructing optimal BST based on [1,3] is given as follows:

**Step 1:** Read  $n$  symbols with probability  $p_i$ .

**Step 2:** Create the table  $C[i, j]$ ,  $1 \leq i \leq j + 1 \leq n$ .

**Step 3:** Set  $C[i, i] = p_i$  and  $C[i - 1, j] = 0$  for all  $i \in [n]$ .

**Step 4:** Recursively compute the following relation:

$$C[i, j] = C[1 \dots k + 1] + C[k + 1 \dots j] + \sum_{m=1}^n p_m, \quad \text{for all } i \text{ and } j$$

**Step 5:** Return  $C[1 \dots n]$  as the maximum cost of constructing a BST.

**Step 6:** End.

The idea is to create a table as shown in below [2]

Table 2: Constructed Table for building Optimal BST

## OPTIMAL BINARY SEARCH TREE(Contd..)

The idea is to create a table as shown in below [2]

Table 2: Constructed Table for building Optimal BST

	0	1		j	n	
1	0	$p_1$				*
i		0	$p_2$			
n+1						0

The diagram shows a path from the top-left cell (0, 1) to the bottom-right cell (n+1, 0). The path consists of several vertical and horizontal steps. Arrows point upwards along the vertical steps and to the right along the horizontal steps. The path starts at (0, 1), goes up to (1, 1), then right to (1, 2), up to (2, 2), right to (2, 3), up to (3, 3), right to (3, 4), up to (4, 4), right to (4, 5), up to (5, 5), right to (5, 6), up to (6, 6), right to (6, 7), up to (7, 7), right to (7, 8), up to (8, 8), right to (8, 9), up to (9, 9), right to (9, 10), up to (10, 10), right to (10, 11), up to (11, 11), right to (11, 12), up to (12, 12), right to (12, 13), up to (13, 13), right to (13, 14), up to (14, 14), right to (14, 15), up to (15, 15), right to (15, 16), up to (16, 16), right to (16, 17), up to (17, 17), right to (17, 18), up to (18, 18), right to (18, 19), up to (19, 19), right to (19, 20), up to (20, 20), right to (20, 21), up to (21, 21), right to (21, 22), up to (22, 22), right to (22, 23), up to (23, 23), right to (23, 24), up to (24, 24), right to (24, 25), up to (25, 25), right to (25, 26), up to (26, 26), right to (26, 27), up to (27, 27), right to (27, 28), up to (28, 28), right to (28, 29), up to (29, 29), right to (29, 30), up to (30, 29), right to (30, 28), up to (28, 27), right to (27, 26), up to (26, 25), right to (25, 24), up to (24, 23), right to (23, 22), up to (22, 21), right to (21, 20), up to (20, 19), right to (19, 18), up to (18, 17), right to (17, 16), up to (16, 15), right to (15, 14), up to (14, 13), right to (13, 12), up to (12, 11), right to (11, 10), up to (10, 9), right to (9, 8), up to (8, 7), right to (7, 6), up to (6, 5), right to (5, 4), up to (4, 3), right to (3, 2), up to (2, 1), right to (1, 0), up to (0, 1), right to (1, 0).

## OPTIMAL BINARY SEARCH TREE(Contd..)

- The aim of the dynamic programming approach is to fill this table for constructing optimal BST.
- What should be entry of this table? For example, to compute  $C[2,3]$  of two items, say key 2 and key 3, two possible trees are constructed as shown below in Fig. 4 and filling the table with minimum cost.

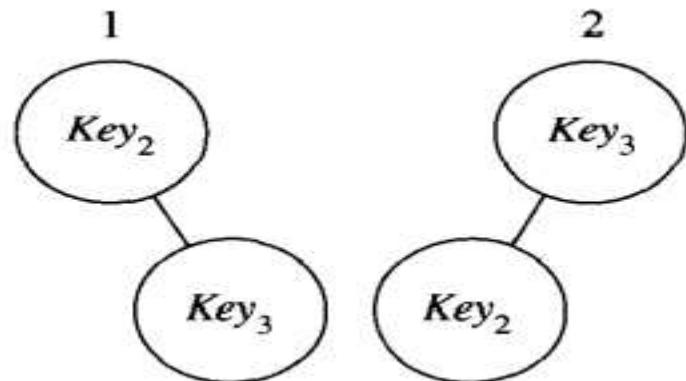


Fig. 4: Two possible ways of BST for key 2 and key 3.

## OPTIMAL BINARY SEARCH TREE(Contd..)

Example 3: Let there be four items A (Danny), B(Ian), C (Radha) , and D (zee) with probability  $2/7$   $1/7$   $3/7$   $1/7$  . Apply dynamic programming approach and construct optimal binary search trees?

**Solution:**

The initial Table is given below in Table 1,

**Table 1: Initial table**

	0	1	2	3	4
1	0	$2/7$			
2		0	$1/7$		
3			0	$3/7$	
4				0	$1/7$
5					0

It can be observed that the table entries are initial probabilities given. Then, using the recursive formula, the remaining entries are calculated.

### OPTIMAL BINARY SEARCH TREE(Contd..)

$$C[1, 2] = \min \begin{cases} C[1, 0] + C[2, 2] + p_1 + p_2 & \text{when } k = 1 \\ C[1, 1] + C[3, 2] + p_1 + p_2 & \text{when } k = 2 \end{cases}$$

$$C[2, 3] = \min \begin{cases} C[2, 1] + C[3, 3] + p_1 + p_3 & \text{when } k = 2 \\ C[2, 2] + C[4, 3] + p_1 + p_3 & \text{when } k = 3 \end{cases}$$

$$C[3, 4] = \min \begin{cases} C[3, 2] + C[4, 4] + p_3 + p_4 & \text{when } k = 3 \\ C[3, 3] + C[5, 4] + p_3 + p_4 & \text{when } k = 4 \end{cases}$$

The updated entries are shown below in Table 2.

## OPTIMAL BINARY SEARCH TREE(Contd..)

**Table 2: Updated table**

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

Similarly, the other entries are obtained as follows:

### OPTIMAL BINARY SEARCH TREE(Contd..)

$$C[1, 3] = \min \begin{cases} C[1, 0] + C[2, 3] + p_1 + p_2 + p_3, \\ \text{when } i = 1, j = 3, k = 1 \\ C[1, 1] + C[3, 3] + p_1 + p_2 + p_3, \\ \text{when } i = 1, j = 3, k = 2 \\ C[1, 2] + C[4, 3] + p_1 + p_2 + p_3, \\ \text{when } i = 1, j = 3, k = 3 \end{cases}$$

$$= \min \left\{ \frac{12}{7}, \frac{11}{7}, \frac{10}{7} \right\} = \frac{10}{7}$$

$$C[2, 4] = \min \begin{cases} C[2, 3] + C[5, 4] + p_2 + p_3 + p_4, \text{when } k = 2 \\ C[2, 2] + C[4, 4] + p_2 + p_3 + p_4, \text{when } k = 3 \\ C[2, 3] + C[5, 4] + p_2 + p_3 + p_4 \text{ when } k = 4 \end{cases}$$

$$= \min \left\{ \frac{11}{7}, \frac{7}{7}, \frac{11}{7} \right\} = \frac{7}{7}$$

## OPTIMAL BINARY SEARCH TREE(Contd..)

The updated table is given in Table 3.

	0	1	2	3	4
1	0	2/7	4/7	10/7	
2		0	1/7	6/7	7/7
3			0	3/7	5/7
4				0	1/7
5					0

Table 3: Updated  
table

The procedure is continued as

## OPTIMAL BINARY SEARCH TREE(Contd..)

$$C[1, 4] = \min \begin{cases} C[1, 0] + C[2, 4] + P_1 + P_2 + P_3 + P_4, \text{ when } k = 1 \\ C[1, 1] + C[3, 4] + P_1 + P_2 + P_3 + P_4, \text{ when } k = 2 \\ C[1, 2] + C[4, 4] + P_1 + P_2 + P_3 + P_4, \text{ when } k = 3 \\ C[1, 3] + C[4, 4] + P_1 + P_2 + P_3 + P_4 \} \text{ when } k = 4 \end{cases}$$

$$= \min \{ \frac{14}{7}, \frac{14}{7}, \frac{12}{7}, \frac{18}{7} \} = \frac{12}{7}$$

The updated final table is given as shown in Table 4.

	0	1	2	3	4
1	0	2/7	4/7	10/7	12/7
2		0	1/7	6/7	7/7
3			0	3/7	5/7
4				0	1/7
5					0

Table 4: Final  
Table

## OPTIMAL BINARY SEARCH TREE(Contd..)

- It can be observed that minimum cost is  $12/7$ . What about the tree structure? This can be reconstructed by noting the minimum k in another table as shown in Table 5

	0	1	2	3	4
1		1	1	3	3
2			2	3	3
3				3	3
4					4
5					

Table 4: Minimum k

- It can be seen from the table 5 that  $C(1,4)$  is 3. So the item 3 is root of the tree. Continuing this fashion, one can find the binary search tree as shown in Fig. 5.

## OPTIMAL BINARY SEARCH TREE(Contd..)

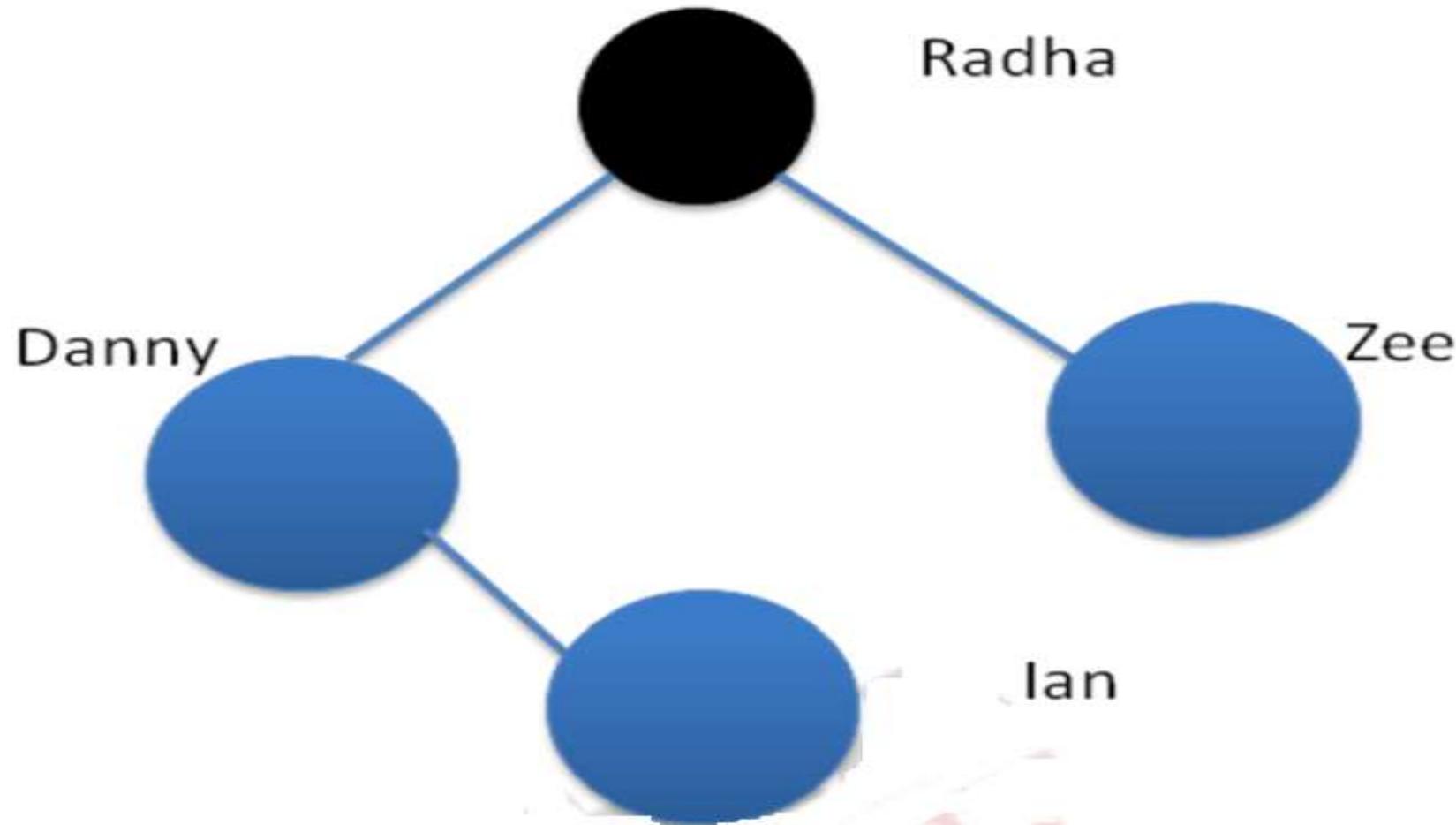


Fig. 5: Constructed Optimal BST

It can be seen that the constructed binary tree is optimal and balanced. The formal algorithm for constructing optimal BST is given as follows:

```
%% Initialize the table for C
for i = 1 to n do
    C[i, i - 1] = 0
    C[i, i] = p_i
End for
C[n + 1, n] = 0
```

```
%% Initialize the table for R
for i = 1 to n do
    R[i, i - 1] = 0
    R[i, i] = i
End for
C[n + 1, n] = 0
```

```
for diag = 1 to n - 1 do
    for i = 1 to n - diag do
        j = i + diag

        C[i,j] = mini < k ≤ j C[1...k - 1] + C[k + 1...j] +  $\sum_{m=1}^n p_m$ 

        R[i,j] = k
    End for
End for
Return C[1,n]
```

#### Complexity Analysis :

The time efficiency is  $\Theta(n^3)$  but can be reduced to  $\Theta(n^2)$  by taking advantage of monotonic property of the entries. The monotonic property is that the entry  $R[i,j]$  is always in the range between  $R[i,j-1]$  and  $R[i+1,j]$ . The space complexity is  $\Theta(n^2)$  as the algorithm is reduced to filling the table.

## SUMMARY

- Dynamic Programming is an effective technique.
- DP can solve LCS problem effectively.
- Edit Distance Problem can be solved effectively by DP approach.

## **HOME ASSIGNMENT:**

Construct the Optimal Binary Search Tree for the given values

Node No : 0	1	2	3
Nodes : 10	12	16	21
Freq : 4	2	6	3

# References

1. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, Introduction to Algorithms, 3<sup>rd</sup> ed., The MIT Press Cambridge, 2014.
2. <https://www.ics.uci.edu/~goodrich/teach/cs260P/notes/LCS.pdf>
3. <https://www.youtube.com/watch?v=sSno9rV8Rhg>
4. Mark Allen Weiss, Data Structures and Algorithm Analysis in C, 2nd ed., Pearson Education, 2006
5. Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran, Fundamentals of Computer Algorithms, Galgotia Publication, 2010
6. Anany Levitin, “Introduction to the Design and Analysis of Algorithms”, Third Edition, Pearson Education, 2012.
7. S.Sridhar , Design and Analysis of Algorithms , Oxford University Press, 2014.
8. A.Levitin, Introduction to the Design and Analysis of Algorithms, Pearson Education, New Delhi, 2012.
9. T.H.Cormen, C.E. Leiserson, and R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA 1992.
10. [www.javatpoint.com](http://www.javatpoint.com)