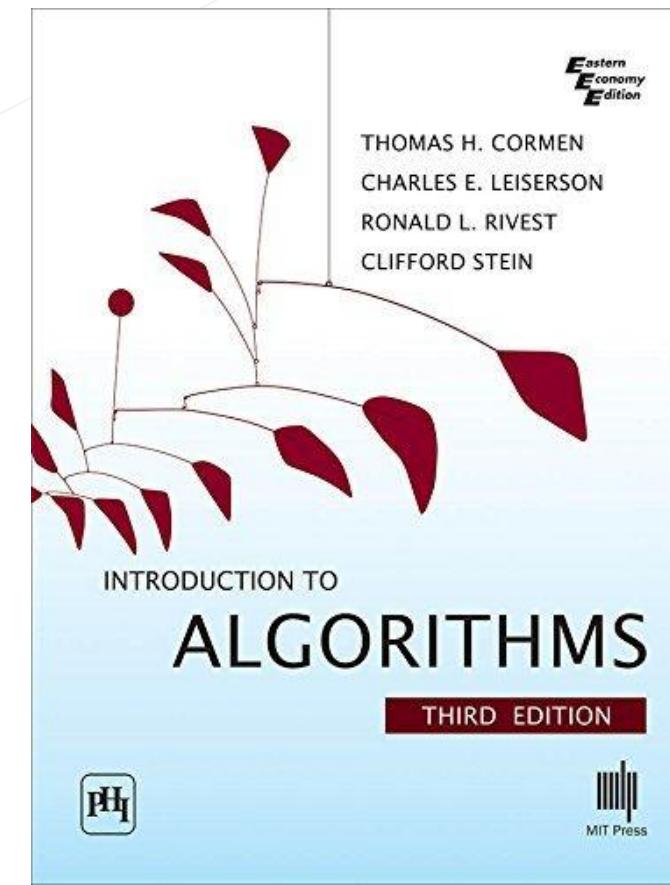


# Algorithm

- Core subjects for CS/IT Students. In GATE 8-10 Marks out of 100 Marks, and 5-6 questions on an average. Most of questions are Numerical. Needs a little time, good scoring. Applied in Industry.

## Syllabus

- Introduction, Searching, Sorting.
- Algorithm design techniques
  - Divide and conquer, Greedy, Dynamic programming
- Graph search
  - Minimum spanning trees, Shortest paths
- Asymptotic worst case time and space complexity.



Q Find the Largest Number Among Three Numbers ?

knowledgeGate

# Q Find the Largest Number Among Three Numbers ?

1. Start
2. Read the three numbers to be compared, as A, B and C.
3. Check if A is greater than B.
  - 3.1 If true, then check if A is greater than C.
    - 3.1.1 If true, print 'A' as the greatest number.
    - 3.1.2 If false, print 'C' as the greatest number.
  - 3.2 If false, then check if B is greater than C.
    - 3.2.1 If true, print 'B' as the greatest number.
    - 3.2.2 If false, print 'C' as the greatest number.
4. End

```
#include <stdio.h>
int main()
{
    int A, B, C;

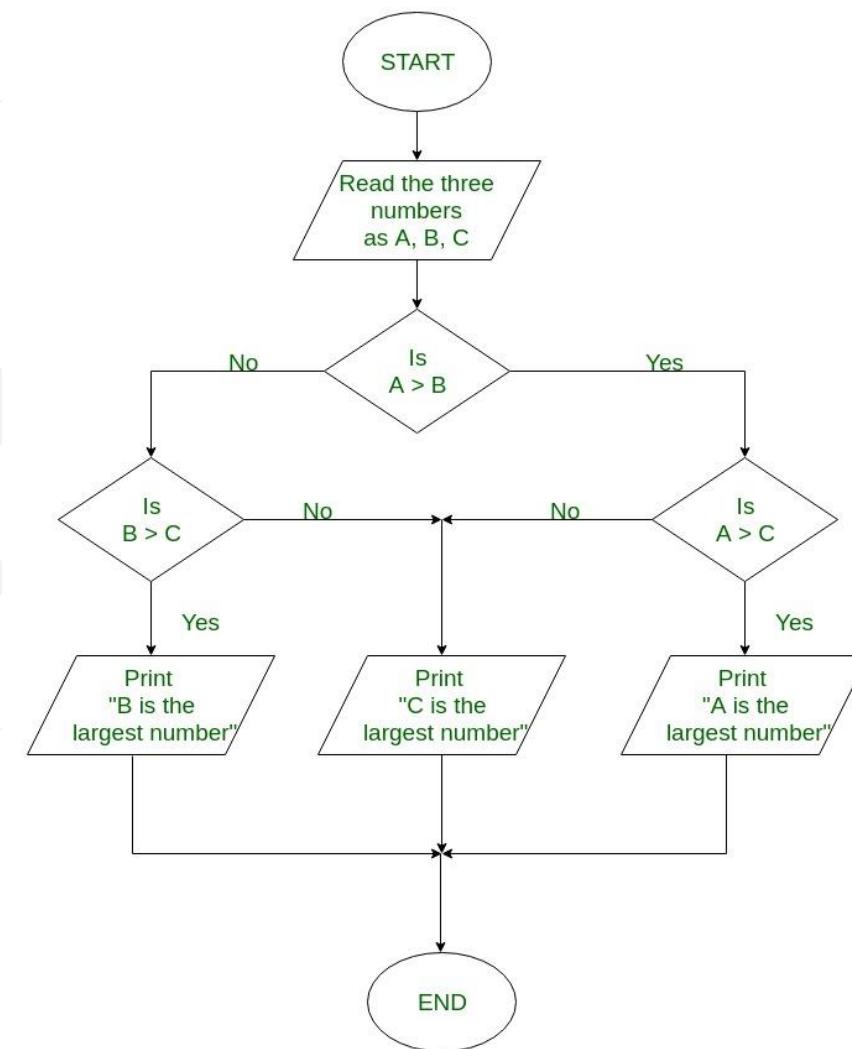
    printf("Enter the numbers A, B and C: ");
    scanf("%d %d %d", &A, &B, &C);

    if(A >= B && A >= C)
        printf("%d is the largest number.", A);

    if(B >= A && B >= C)
        printf("%d is the largest number.", B);

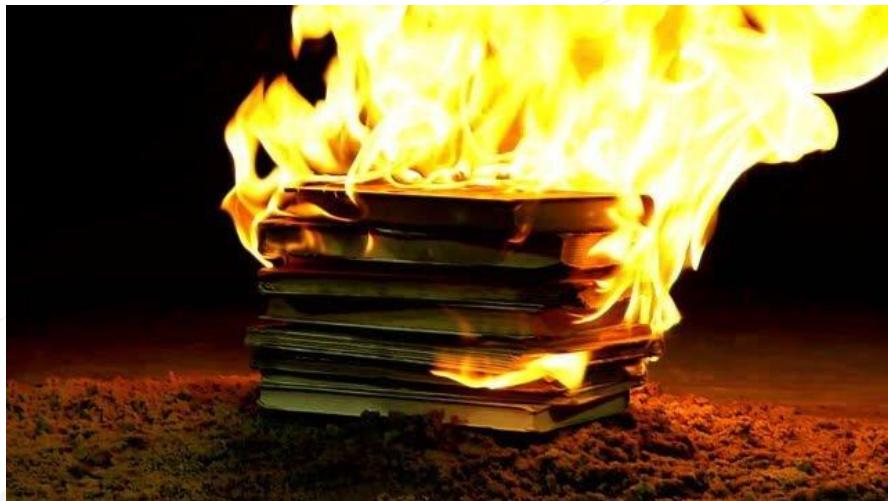
    if(C >= A && C >= B)
        printf("%d is the largest number.", C);

    return 0;
}
```



## Introduction to Algorithm

- In mathematics and computer science, an algorithm is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation. A stem by step Procedure.
- Algorithms are unambiguous specifications for performing calculation, data processing, automated reasoning, and other tasks.
- Will accept Zero or more input, but generate at least one output.
- Every instruction in algo should be effective



0	5	6	6	8	9	0	8	6	8	9
5	3	3	0	3	6	8	9	9	8	8
4	8	3	0	8	0	9	9	2	8	3
6	8	0	2	9	8	6	9	0	9	9
6	9	8	8	6	6	0	5	0	9	9
4	4	0	8	3	0	0	0	2	3	2
4	4	3	8	3	8	9	8	8	9	9
8	0	0	9	0	8	8	6	9	2	3
8	8	0	9	9	9	9	8	8	0	6
8	8	6	0	4	9	9	0	3	6	9
8	5	2	3	5	6	9	8	5	9	0

बेटा किताबों को आग लगा दो

# Problem Solving Cycle

- Problem Definition: Understand Problem
- Constraints & Conditions: Understand constraints if any
- Design Strategies (Algorithmic Strategy)
- Express & Develop the algo
- Validation (Dry run)
- Analysis (Space and Time analysis)
- Coding
- Testing & Debugging
- Installation
- Maintenance

## Need for Analysis

- What parameters can be considered for comparison between cars?



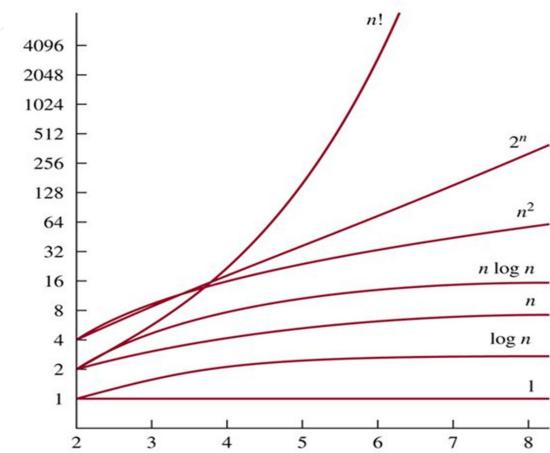
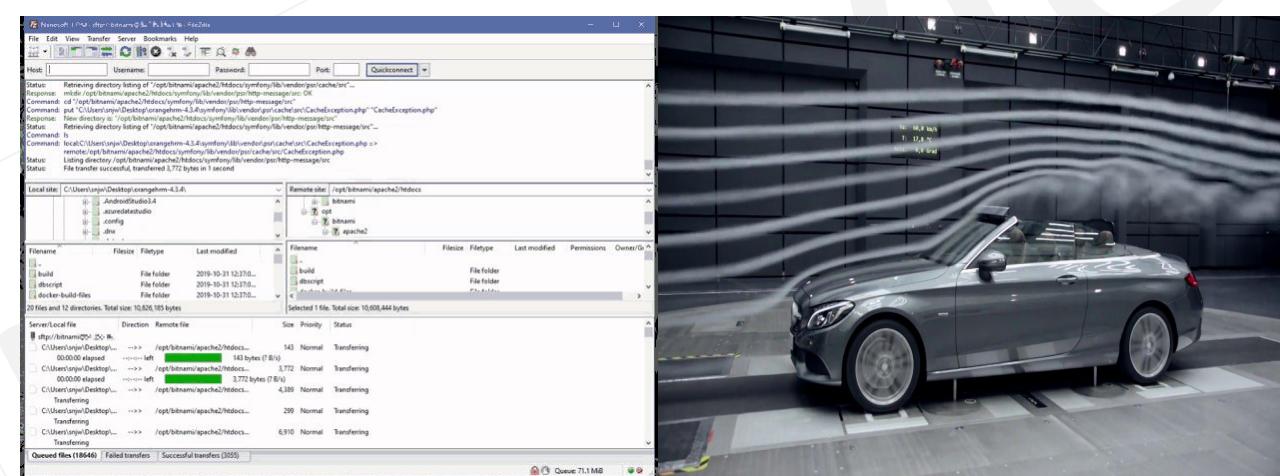
- We do analysis of algorithm to do a performance comparison between different algorithm to figure out which one is best possible option. Following are the parameters which can be considered while analysis of an algorithm

- Time
- Space
- Bandwidth
- Register
- Battery power

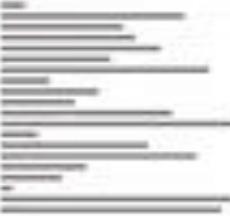
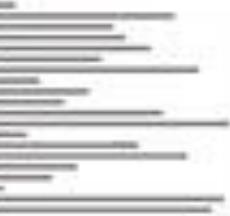
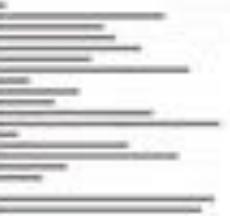
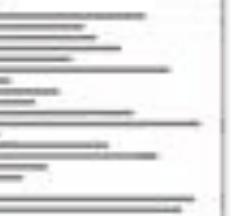
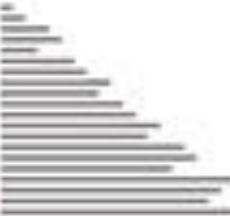
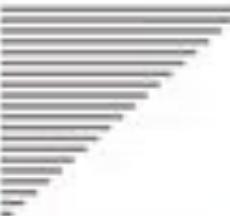
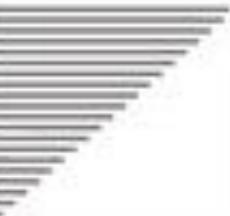
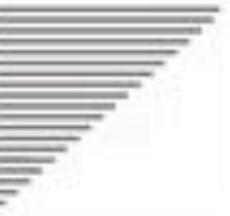
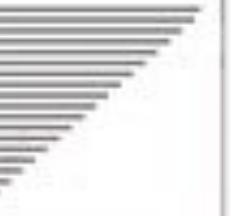


# Types of Analysis

Aspect	Experimental (A Posteriori) Analysis	Apriori (Asymptotic) Analysis
Timing	Performed after code implementation and execution.	Done before implementation, purely theoretical.
Result Type	Measures actual time or space usage.	Estimates time or space complexity.
Influencing Factors	Affected by hardware, software, environment, etc.	Independent of hardware or software factors.
Accuracy	Provides exact, real-world results.	Provides approximate, theoretical results.
Use Case	Useful for real-world performance comparison.	Useful for analysing algorithm efficiency for large inputs.



# Sorting

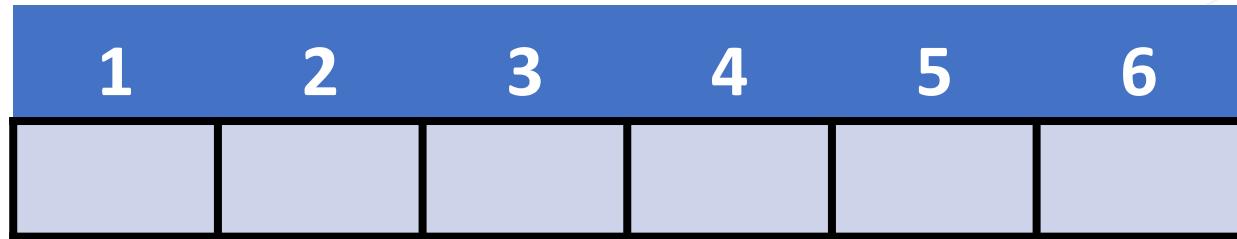
	 <u>Insertion</u>	 <u>Selection</u>	 <u>Bubble</u>	 <u>Shell</u>	 <u>Merge</u>	 <u>Heap</u>	 <u>Quick</u>	
	<u>Random</u>							
	<u>Nearly Sorted</u>							
	<u>Reversed</u>							
	<u>Few Unique</u>							

# Sorting

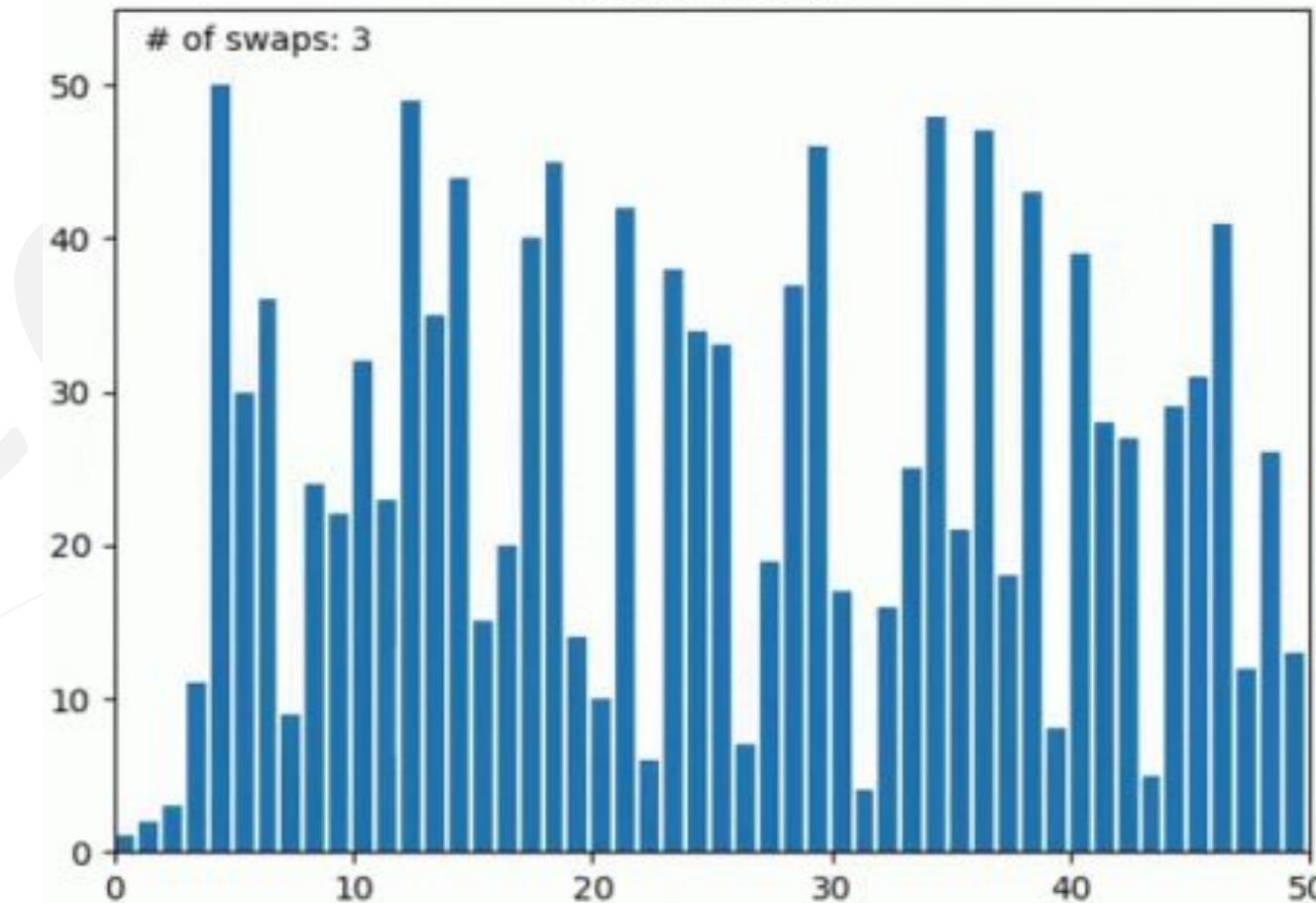
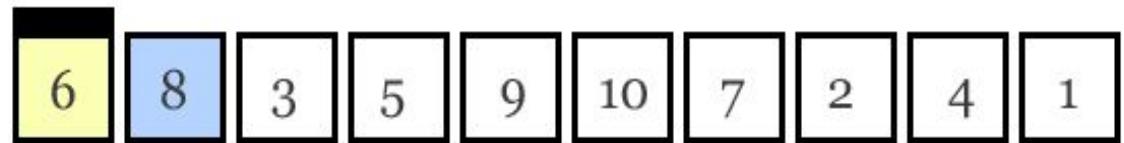
- Sorting is the process of arranging data (numbers or characters) in a specific order (increasing or decreasing). Sorting is crucial in many applications that require data to be in order. There are number of approaches available for sorting and some parameter based on which we judge the performance of these algorithm.
- **Space Complexity:**
  - **Internal Sorting (In-Place):** Sorting that requires no extra memory beyond what is needed for the problem itself (e.g., Heap Sort).
  - **External Sorting:** Sorting that requires additional memory to store data (e.g., Merge Sort).
- **Stability:**
  - A sorting algorithm is **Stable** if it preserves the relative order of equal elements (e.g., Bubble Sort).
  - **Unstable** sorting algorithms do not preserve the order of equal elements (e.g., Insertion Sort).

## Selection Sort

- The algorithm divides the input list into two parts: a sorted sublist of items which is built up from left to right at the front (left) of the list and a sublist of the remaining unsorted items that occupy the rest of the list.



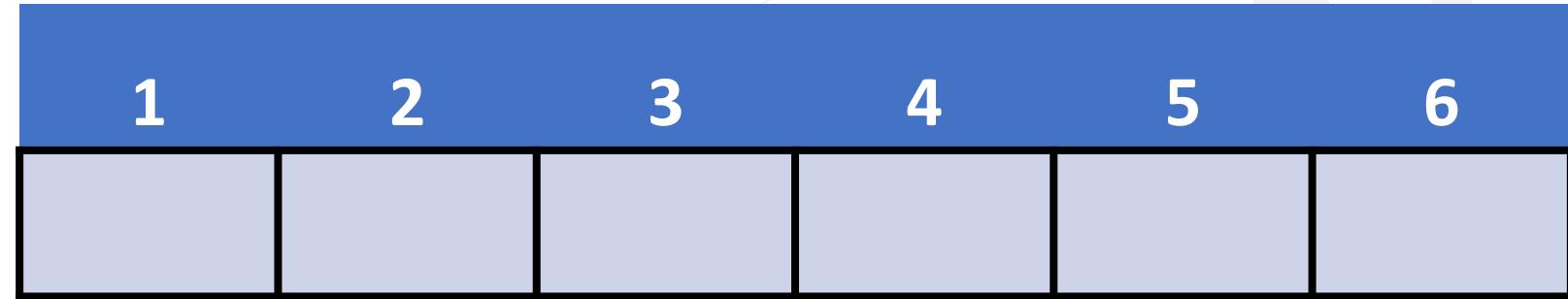
- Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.



## Selection Sort(Algo)

Selection sort (A, n)

```
{  
    for k □ 1 to n-1  
    {  
        min = A[k]  
        Loc = k  
        for j □ k+1 to n  
        {  
            if(min > A[j])  
            {  
                min = A[j]  
                Loc = j  
            }  
        }  
        swap(A[k],A[Loc])  
    }  
}
```



## Selection Sort(Analysis)

```
Selection sort (A, n)
{
    for k □ 1 to n-1
    {
        min = A[k]
        Loc = k
        for j □ k+1 to n
        {
            if(min > A[j])
            {
                min = A[j]
                Loc = j
            }
        }
        swap(A[k],A[Loc])
    }
}

• Selection sort is noted for its simplicity and has performance advantages over more complicated algorithms in certain situations(number of swaps, which is  $n - 1$  in the worst case). It has an  $O(n^2)$  time complexity, which makes it inefficient on large lists.
```

- Depends on structure or content ?
  - Structure
- Internal/External sort algorithm ?
  - Internal
- Stable/Unstable sort algorithm ?
  - Unstable
- Best case time complexity ?
  - $O(n^2)$
- Worst case time complexity ?
  - $O(n^2)$
- Algorithmic Approach?
  - Subtract and Conquer

**Q** Which one of the following is the tightest upper bound that represents the number of swaps required to sort n numbers using selection sort? **(Gate-2013) (1 Marks)**

A)  $O(\log n)$

B)  $O(n)$

C)  $O(n \log n)$

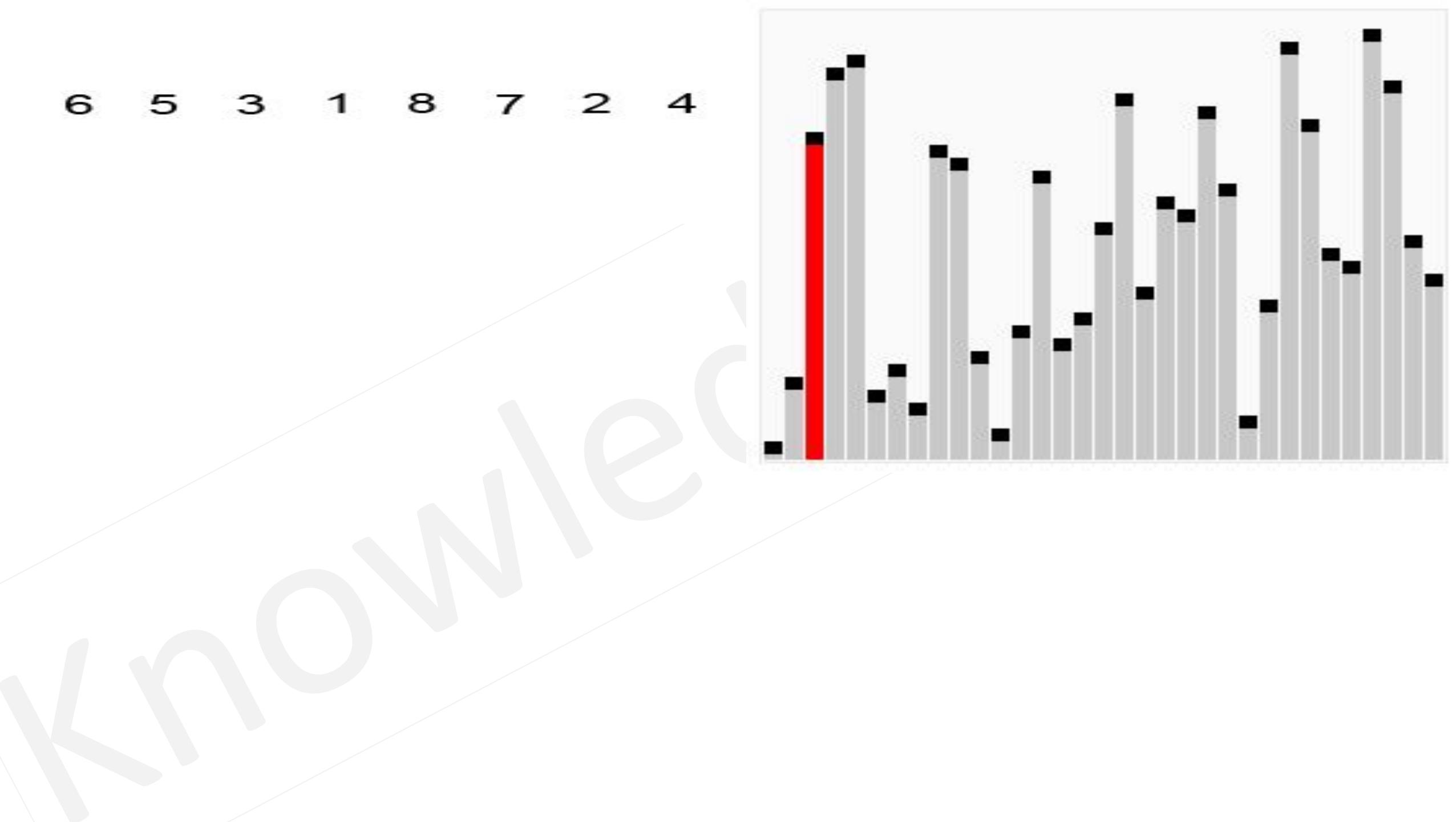
D)  $O(n^2)$

**Q** What is the number of swaps required to sort n elements using selection sort, in the worst case? **(Gate-2009) (1 Marks)**

- a)  $\Theta(n)$
- b)  $\Theta(n \log n)$
- c)  $\Theta(n^2)$
- d)  $\Theta(n^2 \log n)$

# Bubble / Shell / Sinking Sort

- Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list.



# Bubble / Shell / Sinking Sort(Algo without flag)

```
Bubble sort (A, n)
```

```
{
```

```
    for k □ 1 to n-1
```

```
{
```

```
        ptr = 1
```

```
        while(ptr <= n-k)
```

```
{
```

```
            if(A[ptr] > A[ptr+1])
```

```
{
```

```
                exchange(A[ptr],A[ptr+1])
```

```
}
```

```
                ptr = ptr+1
```

```
}
```

```
}
```



# Bubble / Shell / Sinking Sort (Analysis with flag)

Bubble sort (A, n)

```
{  
    for k = 1 to n-1  
    {  
        ptr = 1  
        while(ptr <= n-k)  
        {  
            if(A[ptr] > A[ptr+1])  
            {  
                exchange(A[ptr],A[ptr+1])  
                flag = 1  
            }  
            ptr = ptr+1  
        }  
        if(!flag)  
        {  
            break;  
        }  
    }  
}
```

- Depends on structure or content ?
- Internal/External sort algorithm ?
- Stable/Unstable sort algorithm ?
- Best case time complexity ?
- Worst case time complexity ?
- Algorithmic Approach?

# Bubble / Shell / Sinking Sort (Analysis with flag)

Bubble sort (A, n)

```
{  
    for k = 1 to n-1  
    {  
        ptr = 1  
        while(ptr <= n-k)  
        {  
            if(A[ptr] > A[ptr+1])  
            {  
                exchange(A[ptr],A[ptr+1])  
                flag = 1  
            }  
            ptr = ptr+1  
        }  
        if(!flag)  
        {  
            break;  
        }  
    }  
}
```

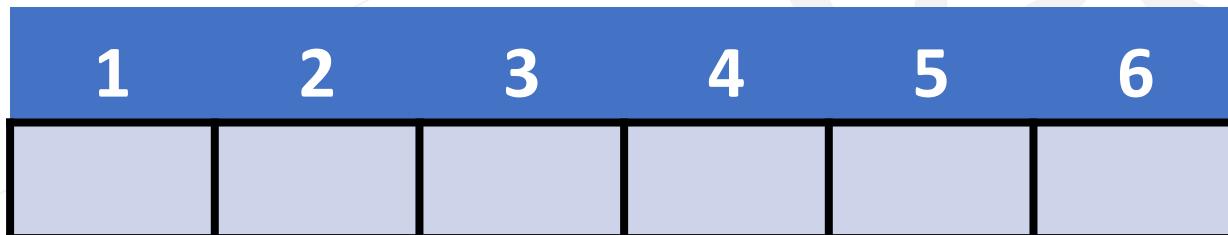
- Depends on structure or content ?
  - Both
- Internal/External sort algorithm ?
  - Internal
- Stable/Unstable sort algorithm ?
  - Stable
- Best case time complexity ?
  - $O(n)$
- Worst case time complexity ?
  - $O(n^2)$
- Algorithmic Approach?
  - Subtract and Conquer

## Bubble / Shell / Sinking Sort (Conclusion)

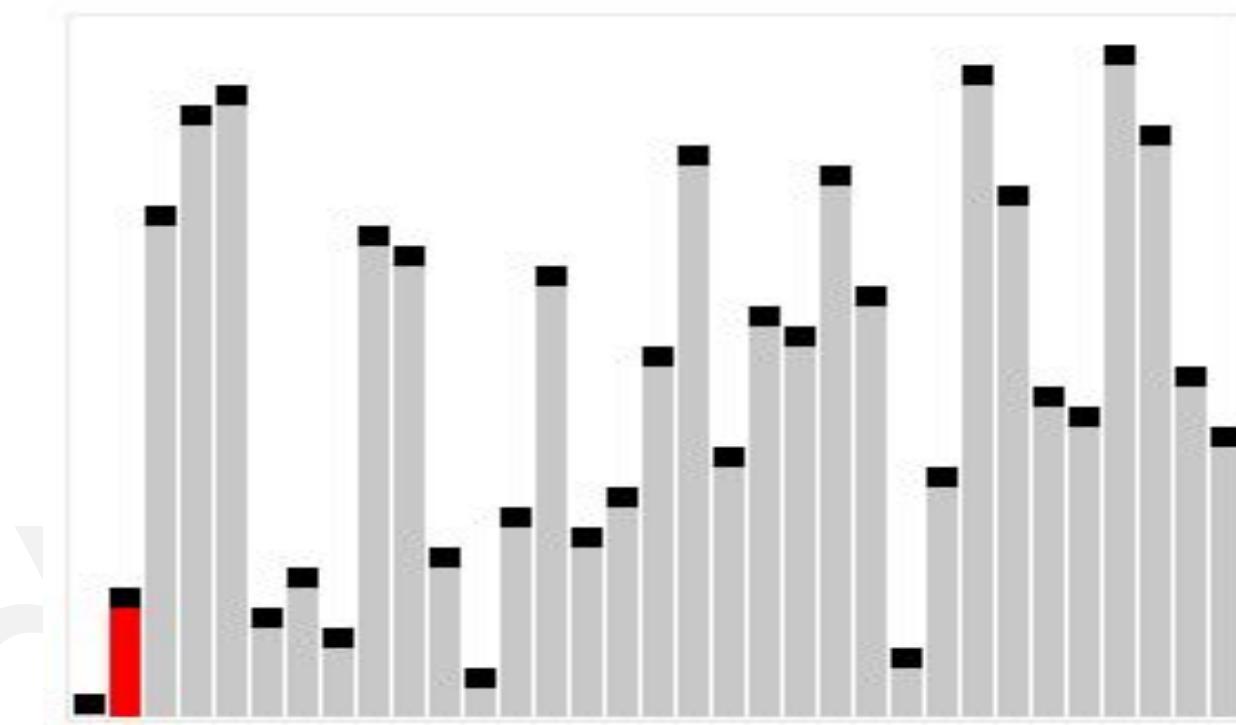
- **Efficiency:** Bubble Sort performs poorly in real-world scenarios compared to other  $O(n^2)$  algorithms like Insertion Sort and Selection Sort, which generally run faster and have similar complexity.
- **Practical Use:** It's not a practical sorting algorithm and is mostly used for educational purposes.
- **Comparison with Efficient Algorithms:** Efficient algorithms like Heap Sort and Merge Sort are preferred in real-world applications and are used in sorting libraries of languages like Python and Java.
- **Best Case:** When the list is already sorted, Bubble Sort has a time complexity of  $O(n)$ , which is a benefit over algorithms that continue their full sorting process even in best-case scenarios.

# Insertion Sort

- Process: Insertion Sort removes one element from the input at a time, finds its correct position in the sorted list, and inserts it there.
- Repetition: The process repeats until no input elements are left.
- Comparison: At each position, the algorithm compares the element with the largest value in the sorted part of the list (adjacent element).
- If Larger: If the element is larger than the adjacent sorted value, it stays in place, and the next element is checked.
- If Smaller: If the element is smaller, the larger values are shifted to make room, and the element is inserted into its correct position.



6 5 3 1 8 7 2 4



# Insertion Sort (Analysis)

```
Insertion sort (A, n)
{
    for j □ 2 to n
    {
        key = A[j]
        i = j - 1
        while(i>0 and A[i] > key)
        {
            A[i+1] = A[i]
            i = i-1
        }
        A[i+1]=key
    }
}
```



## Insertion Sort (Analysis)

```
Insertion sort (A, n)
{
    for j □ 2 to n
    {
        key = A[j]
        i = j - 1
        while(i>0 and A[i] > key)
        {
            A[i+1] = A[i]
            i = i-1
        }
        A[i+1]=key
    }
}
```

- Depends on structure or content ?
  - Both
- Internal/External sort algorithm ?
  - Internal
- Stable/Unstable sort algorithm ?
  - Stable
- Best case time complexity ?
  - $O(n)$
- Worst case time complexity ?
  - $O(n^2)$
- Algorithmic Approach?
  - Subtract and Conquer

## Insertion Sort (Conclusion)

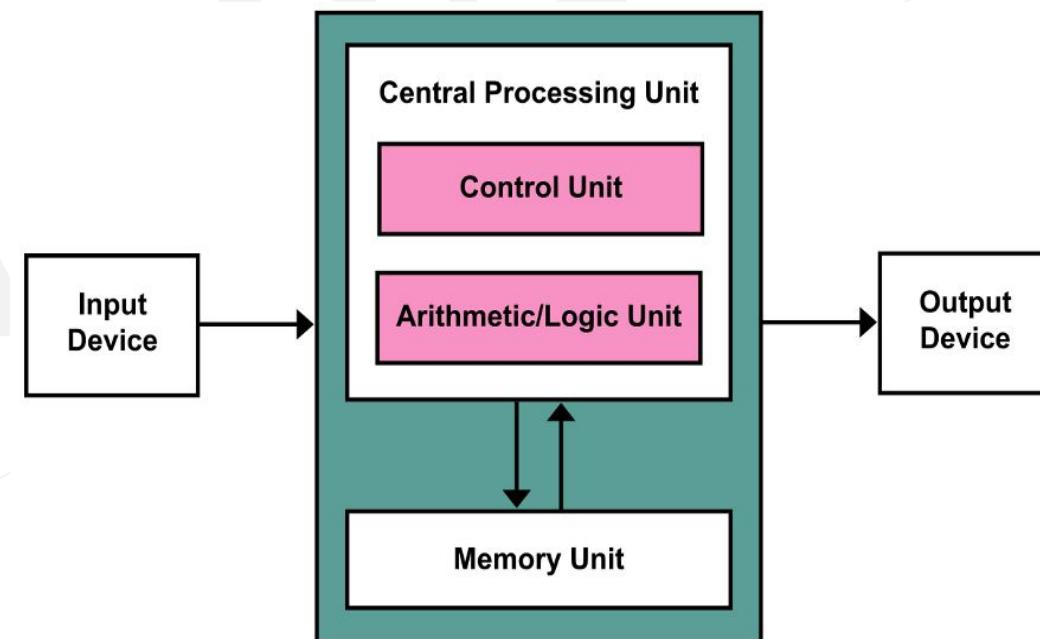
- **Less Efficient for Large Lists:** Insertion Sort is less efficient compared to advanced algorithms like Heap Sort and Merge Sort, which both have  $O(n\log n)$  time complexity.
- **Advantages for Small Data Sets:** Insertion Sort performs well on small data sets and is more efficient than other quadratic algorithms like Selection Sort and Bubble Sort.
- **Practical Use:** Despite its inefficiency on large lists, Insertion Sort remains useful for smaller or nearly sorted datasets.

**Q** What is the worst-case time complexity of insertion sort where position of the data to be inserted is calculated using binary search?

- (A) N
- (B)  $N \log N$
- (C)  $N^2$
- (D)  $N(\log N)^2$

# Merge Sort

- In computer science, merge sort is an efficient, general-purpose, comparison-based sorting algorithm. Merge sort is a divide and conquer algorithm that was invented by John von Neumann in 1945.
- Conceptually, a merge sort works as follows:
  - Divide the unsorted list into  $n$  sublists, each containing one element (a list of one element is considered sorted).
  - Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.



Von Neumann Architecture

38	27	43	3	9	82	10
----	----	----	---	---	----	----

38	27	43	3
----	----	----	---

9	82	10
---	----	----

38	27
----	----

43	3
----	---

9	82
---	----

10
----

38

27

43

3

9

82

10

27	38
----	----

3	43
---	----

9	82
---	----

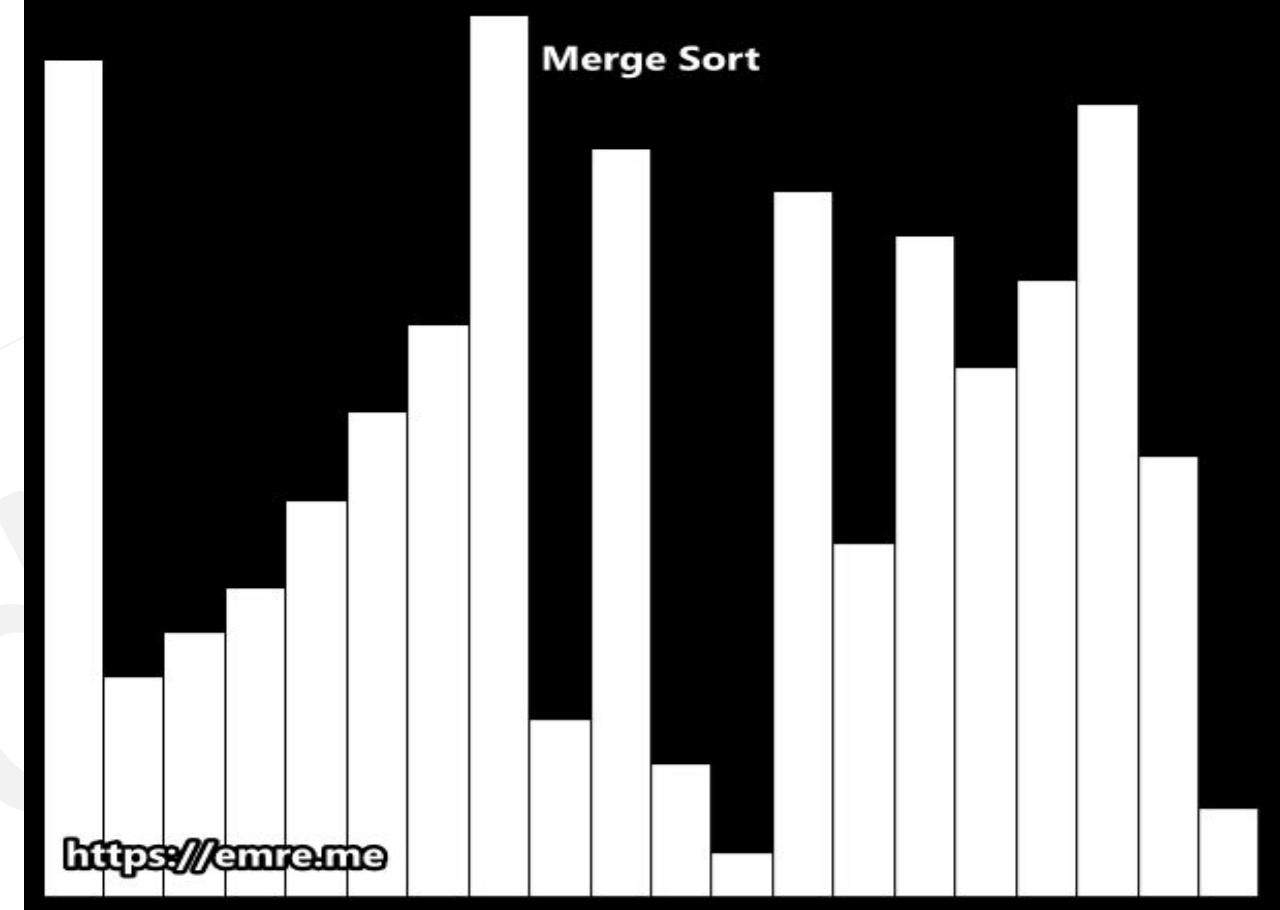
10
----

3	27	38	43
---	----	----	----

9	10	82
---	----	----

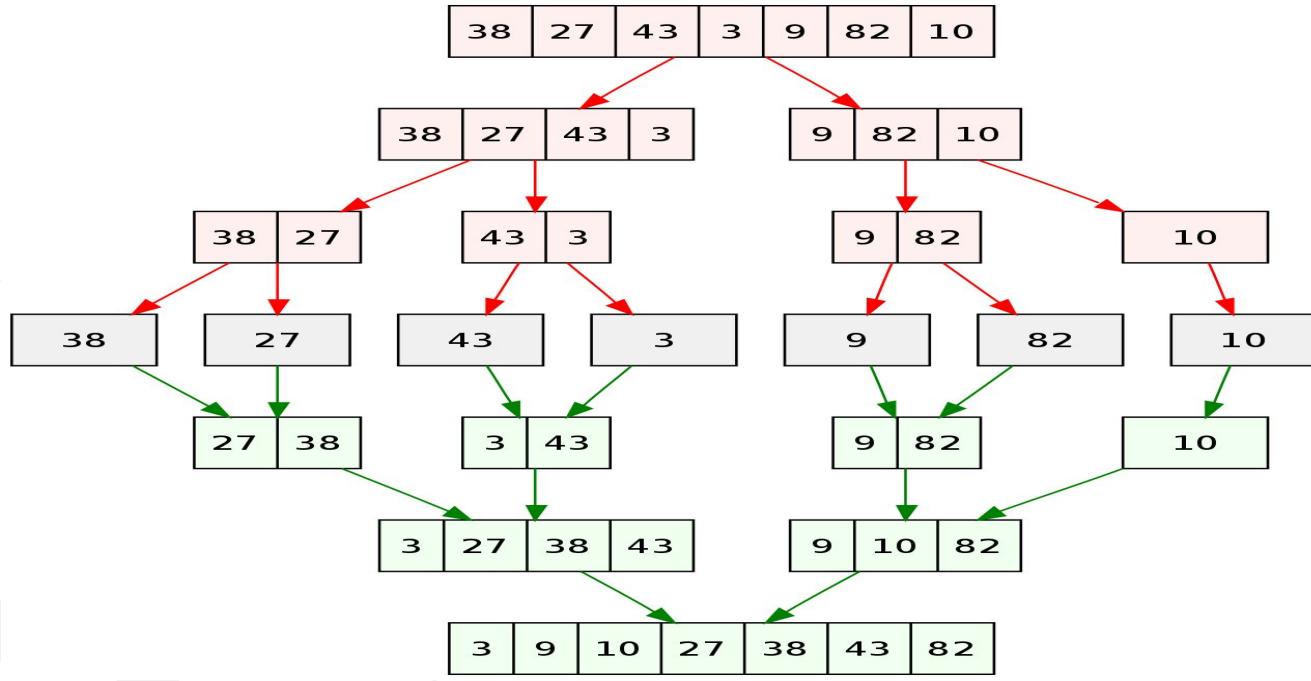
3	9	10	27	38	43	82
---	---	----	----	----	----	----

know



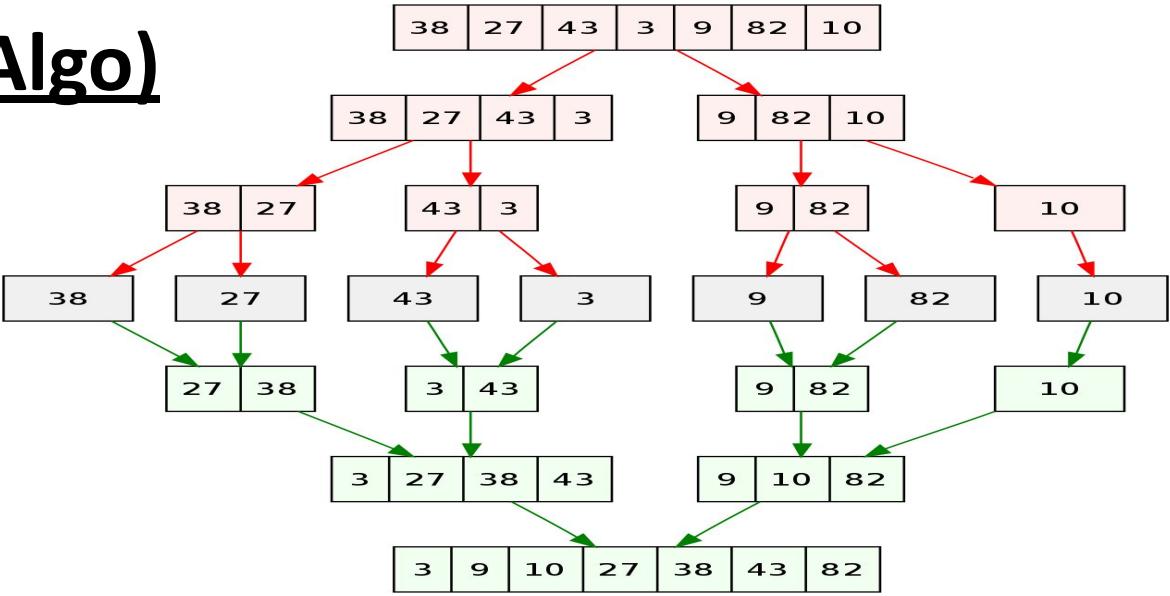
# Merge Sort(Algo)

```
Merge_Sort(A, p, r)
{
    if(p < r)
    {
        q = ⌊(p + r)/2⌋
        Merge_Sort (A, p, q)
        Merge_Sort (A, q + 1, r)
        Merge (A, p, q, r)
    }
}
```



# Merge Sort(Algo)

```
Merge (A, p, q, r)
{
    n1 □ q - p + 1
    n2 □ r - q
    Create array L [1.....n1+1] and R [1.....n2+1]
    for i □ 1 to n1
        do L[i] = A [p + i - 1]
    for j □ 1 to n2
        do R[j] = A [j + q]
    L [n1+1] □ ∞
    R [n2+1] □ ∞
    i □ 1
    j □ 1
    for k □ p to r
    {
        if(L[i] <= R[j])
        {
            A[k] = L[i]
            i = i + 1
        }
        Else
        {
            A[k] = R[j]
            j = j + 1
        }
    }
}
```



## Merge Sort(Analysis)

- Depends on structure or content ?
  - Structure
- Internal/External sort algorithm ?
  - External
- Stable/Unstable sort algorithm ?
  - Stable
- Best case time complexity ?
  - $O(n \log n)$
- Worst case time complexity ?
  - $O(n \log n)$
- Algorithmic Approach?
  - Divide and Conquer

## Merge Sort(Conclusion)

- **Recurrence Relation**: The time complexity of Merge Sort for a list of length  $n$  is expressed by the recurrence relation  $T(n)=2T(n/2)+n$ , reflecting the divide-and-conquer approach.
- **Comparison with Quick Sort**: In the worst case, Merge Sort performs about 39% fewer comparisons than Quick Sort in its average case.
- **Sequential Access Efficiency**: Merge Sort is more efficient than Quick Sort for lists that are accessed sequentially, making it popular in languages like Lisp, where such data structures are common.
- **Space Complexity**: Merge Sort requires  $O(n)$  extra space, as it needs additional memory for the merging process.

**Q** Assume that a Merge\_Sort algorithm in the worst case takes 30 seconds for an input of size 64. Which of the following most closely approximates the maximum input size of a problem that can be solved in 6 minutes? **(Gate-2015) (1 Marks)**  
[Asked in Capgemini 2021]

- a) 256
- b) 512
- c) 1024
- d) 2048

**Q** A list of  $n$  strings, each of length  $n$ , is sorted into lexicographic order using the merge-sort algorithm. The worst case running time of this computation is  
**(Gate-2012) (1 Marks)**

(A)  $O(n \log n)$

(B)  $O(n^2 \log n)$

(C)  $O(n^2 + \log n)$

(D)  $O(n^2)$

# Heap Sort

- **Invention:** Heap Sort was invented by J.W.J. Williams in 1964, introducing the heap data structure as a useful tool in its own right.
- **Process:** Heap Sort divides the input into a sorted and unsorted region. It repeatedly extracts the largest element from the unsorted region and inserts it into the sorted region.
- **Efficiency:** It avoids a linear-time scan by maintaining the unsorted region in a heap data structure, allowing for quicker extraction of the largest element.
- **Steps:**
  - First, a heap is built from the data (typically in an array with the layout of a complete binary tree).
  - Then, elements are removed from the heap one by one (starting with the largest) and placed into a sorted array.
- **Heap Property Maintenance:** After each extraction of the largest element, the heap is updated to maintain its structure until all elements are sorted.

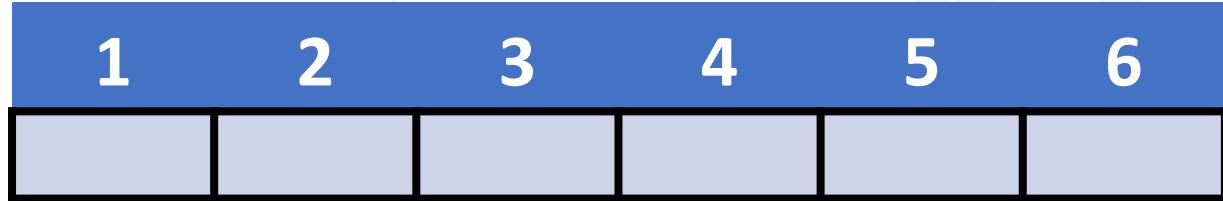


## Heap Sort(Algo)

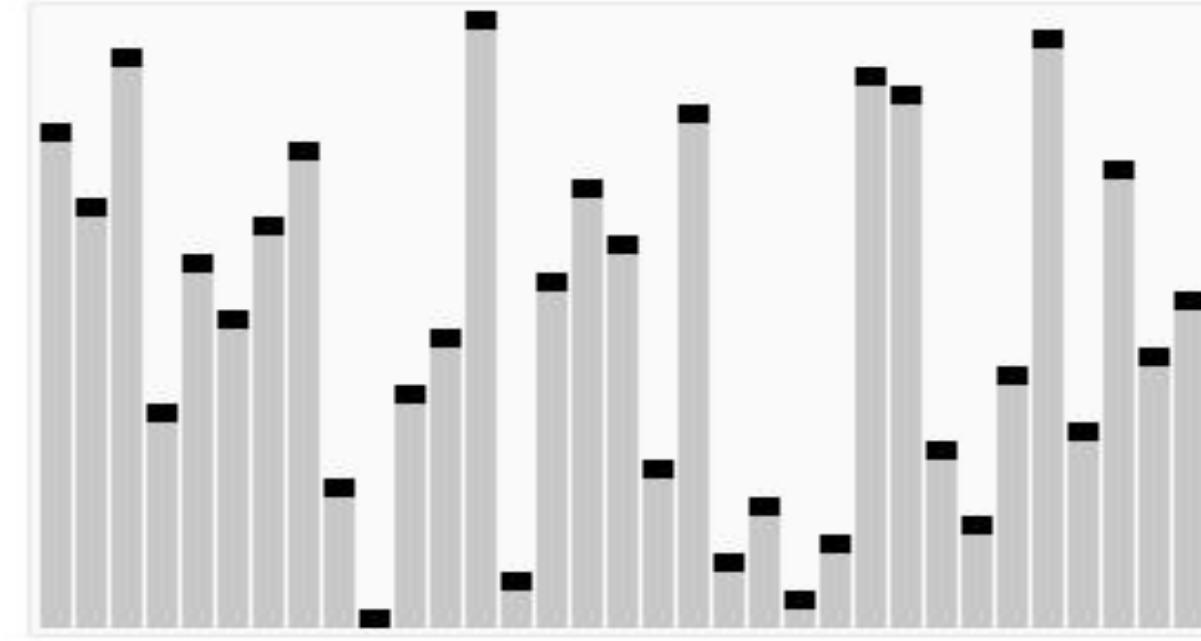
```
Heap_Sort(A)
{
    Build_Max_heap(A)
    for i □ length[A] down to 2
    {
        do exchange (A[1] □□ A[i])
        Heap-size[A] □ Heap-size[A] – 1
        Max-Heapify(A,1)
    }
}
```

```
Build_Max_Heap(A)
{
    Heap-size[A] □ length[A]
    for i □ L length[A]/2 ↓ down to 1
    {
        do Max-Heapify (A, i)
    }
}
```

```
Max-Heapify(A, i)
{
    L □ Left[i]
    R □ Right[i]
    if( L <= Heap_size[A] and A[L] > A[i])
        Largest □ L
    Else
        Largest □ i
    if(R <= Heap_size[A] and A[r] > A[Largest])
        Largest □ R
    if(Largest != i)
    {
        Exchange( A[i] □□ A[Largest])
        Max-Heapify(A, Largest)
    }
}
```



6 5 3 1 8 7 2 4



know

## Heap Sort(Analysis)

- Depends on structure or content ?
  - Both
- Internal/External sort algorithm ?
  - Internal
- Stable/Unstable sort algorithm ?
  - Unstable
- Best case time complexity ?
  - $O(n \log n)$
- Worst case time complexity ?
  - $O(n \log n)$
- Algorithmic Approach?
  - Mixed Approach

**Q Consider the following statements: (Gate-2019) (1 Marks)**

- I. The smallest element in a max-heap is always at a leaf node.
- II. The second largest element in a max-heap is always a child of the root node.
- III. A max-heap can be constructed from a binary search tree in  $\Theta(n)$  time.
- IV. A binary search tree can be constructed from a max-heap in  $\Theta(n)$  time.

Which of the above statements is/are TRUE?

- (A) II, III and IV
- (B) I, II and III
- (C) I, III and IV
- (D) I, II and IV

**Q** An operator `delete(i)` for a binary heap data structure is to be designed to delete the item in the  $i$ -th node. Assume that the heap is implemented in an array and  $i$  refers to the  $i$ -th index of the array. If the heap tree has depth  $d$  (number of edges on the path from the root to the farthest leaf), then what is the time complexity to re-fix the heap efficiently after the removal of the element? **(Gate-2016) (1 Marks)**

- a)  $O(1)$
- b)  $O(d)$  but not  $O(1)$
- c)  $O(2^d)$  but not  $O(d)$
- d)  $O(d2^d)$  but not  $O(2^d)$

**Q** Consider a complete binary tree where the left and the right subtrees of the root are max-heaps. The lower bound for the number of operations to convert the tree to a heap is **(GATE-2015) (1 Marks)**

a)  $\Omega(\log n)$

b)  $\Omega(n)$

c)  $\Omega(n \log n)$

d)  $\Omega(n^2)$

**Q** Consider the process of inserting an element into a Max Heap, where the Max Heap is represented by an array. Suppose we perform a binary search on the path from the new leaf to the root to find the position for the newly inserted element, the number of comparisons performed is: **(Gate-2007) (1 Marks)**

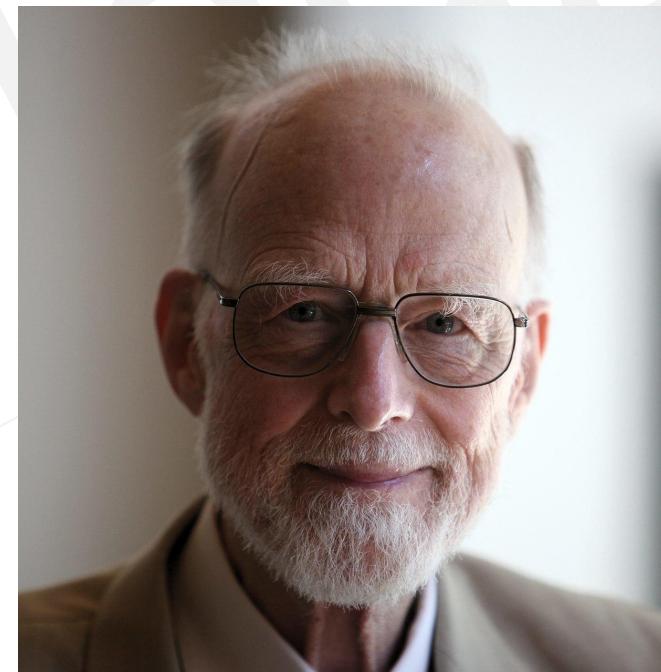
- a)  $\Theta(\log_2 n)$
- b)  $\Theta(\log_2 \log_2 n)$
- c)  $\Theta(n)$
- d)  $\Theta(n \log_2 n)$

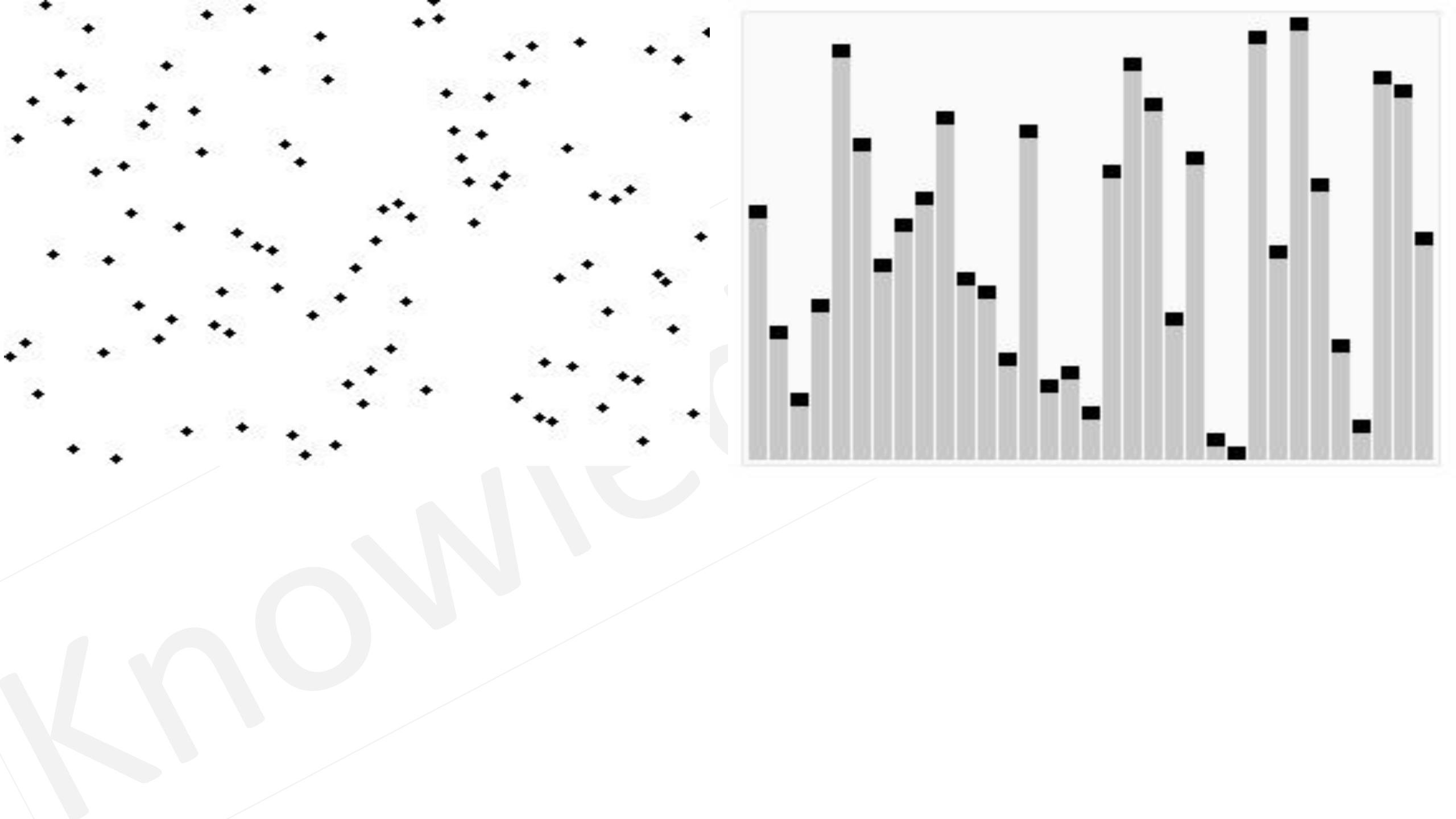
**Q** Which of the following is true about merge sort?

- (A) Merge Sort works better than quick sort if data is accessed from slow sequential memory.
- (B) Merge Sort is stable sort by nature
- (C) Merge sort outperforms heap sort in most of the practical situations.
- (D) All of the above.

# Quick Sort

- **Development:** Quick Sort was developed by British computer scientist Tony Hoare and published in 1961. It remains a widely used sorting algorithm today.
- **Performance:** When implemented efficiently, Quick Sort is faster than Merge Sort and approximately two to three times faster than Heap Sort in practice.
- **Recognition:** Tony Hoare received the Turing Award in 1980, considered the highest distinction in computer science, for his contributions.
- **Algorithm Type:** Quick Sort is a divide-and-conquer algorithm that selects a "pivot" element and partitions the array into two sub-arrays—one with elements smaller than the pivot and the other with larger elements.
- **In-Place Sorting:** Quick Sort can be performed in-place, requiring minimal additional memory, and sorts sub-arrays recursively.

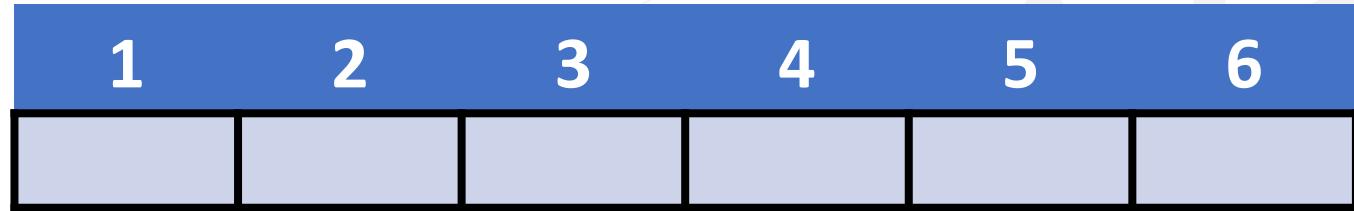




```

Quick_Sort(A, p, r)
{
    if(p < r)
    {
        q = partition (A, p, r)
        quick_Sort(A, p, q - 1)
        quick_Sort(A, q + 1, r)
    }
}
Partition (A, p, r)
{
    x = A[r]
    i = p - 1
    for j = p to r - 1
    {
        if(A[j] <= x)
        {
            i = i + 1
            Exchange( A[i] ↔ A[j])
        }
    }
    Exchange( A[i + 1] ↔ A[r])
    return i+1
}

```



## Quick Sort(Analysis)

- Depends on structure or content ?
  - Both
- Internal/External sort algorithm ?
  - Internal
- Stable/Unstable sort algorithm ?
  - Unstable
- Best case time complexity ?
  - $O(n \log n)$
- Worst case time complexity ?
  - $O(n^2)$
- Algorithmic Approach?
  - Divide and Conquer

**Q** An array of 25 distinct elements is to be sorted using quicksort. Assume that the pivot element is chosen uniformly at random. The probability that the pivot element gets placed in the worst possible location in the first round of partitioning (rounded off to 2 decimal places) is \_\_\_\_\_.  
**(Gate-2019) (1 Marks) [Asked in CoCubes 2020]**

**Q** Which one of the following is the recurrence equation for the worst case time complexity of the Quicksort algorithm for sorting  $n(\geq 2)$  numbers? In the recurrence equations given in the options below, c is a constant. **(Gate-2015) (1 Marks)**

(a)  $T(n) = 2T(n/2) + cn$

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

(c)  $T(n) = 2T(n - 1) + cn$

(d)  $T(n) = T(n/2) + cn$

**Q** You have an array of  $n$  elements. Suppose you implement quicksort by always choosing the central element of the array as the pivot. Then the tightest upper bound for the worst-case performance is **(Gate-2014) (1 Marks)**

a)  $O(n^2)$

b)  $O(n\log n)$

c)  $\Theta(n\log n)$

d)  $O(n^3)$

**Q** Consider the Quicksort algorithm. Suppose there is a procedure for finding a pivot element which splits the list into two sub-lists each of which contains at least one-fifth of the elements. Let  $T(n)$  be the number of comparisons required to sort  $n$  elements. Then **(Gate-2008) (2 Marks)**

(A)  $T(n) \leq 2T(n/5) + n$

(B)  $T(n) \leq T(n/5) + T(4n/5) + n$

(C)  $T(n) \leq 2T(4n/5) + n$

(D)  $T(n) \leq 2T(n/2) + n$

**Q** Suppose we are sorting an array of eight integers using quicksort, and we have just finished the first partitioning with the array looking like this:

2 5 1 7 9 12 11 10

Which statement is correct?

- (A) The pivot could be either the 7 or the 9
- (B) The pivot could be the 7, but it is not the 9
- (C) The pivot is not the 7, but it could be the 9
- (D) Neither the 7 nor the 9 is the pivot.

**Q** What is recurrence for worst case of QuickSort and what is the time complexity in Worst case?

- (A)** Recurrence is  $T(n) = T(n-2) + O(n)$  and time complexity is  $O(n^2)$
- (B)** Recurrence is  $T(n) = T(n-1) + O(n)$  and time complexity is  $O(n^2)$
- (C)** Recurrence is  $T(n) = 2T(n/2) + O(n)$  and time complexity is  $O(n \log n)$
- (D)** Recurrence is  $T(n) = T(n/10) + T(9n/10) + O(n)$  and time complexity is  $O(n \log n)$

Sorting Algorithm	Best Case	Worst Case
Selection	$O(n^2)$	$O(n^2)$
Bubble	$O(n^2) / O(n)$	$O(n^2)$
Insertion	$O(n)$	$O(n^2)$
Merge	$O(n \log n)$	$O(n \log n)$
Heap	$O(n \log n)$	$O(n \log n)$
Quick	$O(n \log n)$	$O(n^2)$

**Q** There are  $n$  unsorted arrays:  $A_1, A_2, \dots, A_n$ . Assume that  $n$  is odd. Each of  $A_1, A_2, \dots, A_n$  contains  $n$  distinct elements. There are no common elements between any two arrays. The worst-case time complexity of computing the median of the medians of  $A_1, A_2, \dots, A_n$  is **(Gate-2019) (1 Marks)**

- a)  $O(n)$
- b)  $O(n \log n)$
- c)  $O(n^2)$
- d)  $\Omega(n^2 \log n)$

**Q** Let  $A$  be an array of 31 numbers consisting of a sequence of 0's followed by a sequence of 1's. The problem is to find the smallest index  $i$  such that  $A[i]$  is 1 by probing the minimum number of locations in  $A$ . The worst-case number of probes performed by an *optimal* algorithm is \_\_\_\_\_. (Gate-2017) (1 Marks)

**Q** The worst-case running times of Insertion sort, Merge sort and Quick sort, respectively, are: **(Gate-2016) (1 Marks)**

a)  $\Theta(n \log n)$ ,  $\Theta(n \log n)$  and  $\Theta(n^2)$

b)  $\Theta(n^2)$ ,  $\Theta(n^2)$  and  $\Theta(n \log n)$

c)  $\Theta(n^2)$ ,  $\Theta(n \log n)$  and  $\Theta(n \log n)$

d)  $\Theta(n^2)$ ,  $\Theta(n \log n)$  and  $\Theta(n^2)$

**Q** Assume that the algorithms considered here sort the input sequences in ascending order. If the input is already in ascending order, which of the following are TRUE? **(Gate-2016) (1 Marks)**

- I. Quicksort runs in  $\Theta(n^2)$  time
  - II. Bubble sort runs in  $\Theta(n^2)$  time
  - III. Merge\_Sort runs in  $\Theta(n)$  time
  - IV. Insertion sort runs in  $\Theta(n)$  time
- 
- a) I and II only
  - b) I and III only
  - c) II and IV only
  - d) I and IV only

**Q** An element in an array X is called a leader if it is greater than all elements to the right of it in X. The best algorithm to find all leaders in an array (**GATE-2006**) (2 Marks)

- (A) Solves it in linear time using a left to right pass of the array
- (B) Solves it in linear time using a right to left pass of the array
- (C) Solves it using divide and conquer in time  $\theta(n \log n)$
- (D) Solves it in time  $\theta(n^2)$

**Q** Which of the following sorting algorithms has the lowest worst-case complexity?

(A) Merge Sort

(B) Bubble Sort

(C) Quick Sort

(D) Selection Sort

**Q** Which of the following is not a stable sorting algorithm in its typical implementation.

- (A)** Insertion Sort
- (B)** Merge Sort
- (C)** Quick Sort
- (D)** Bubble Sort

**Q** Which of the following sorting algorithms in its typical implementation gives best performance when applied on an array which is sorted or almost sorted (maximum 1 or two elements are misplaced).

- (A) Quick Sort
- (B) Heap Sort
- (C) Merge Sort
- (D) Insertion Sort

**Q** Consider a situation where swap operation is very costly. Which of the following sorting algorithms should be preferred so that the number of swap operations are minimized in general?

- (A) Heap Sort
- (B) Selection Sort
- (C) Insertion Sort
- (D) Merge Sort

**Q** You have to sort 1 GB of data with only 100 MB of available main memory. Which sorting technique will be most appropriate?

(A) Heap sort

(B) Merge sort

(C) Quick sort

(D) Insertion sort

**Q** Which sorting algorithm will take least time when all elements of input array are identical? Consider typical implementations of sorting algorithms.

(A) Insertion Sort

(B) Heap Sort

(C) Merge Sort

(D) Selection Sort

**Q** An array of  $n$  numbers is given, where  $n$  is an even number. The maximum as well as the minimum of these  $n$  numbers needs to be determined. Which of the following is TRUE about the number of comparisons needed? **(Gate-2007) (2 Marks)**

- a) At least  $2n - c$  comparisons, for some constant  $c$ , are needed.
- b) At most  $1.5n - 2$  comparisons are needed.
- c) At least  $n \log_2 n$  comparisons are needed.
- d) None of the above

**Q** The minimum number of comparisons required to determine if an integer appears more than  $n/2$  times in a sorted array of  $n$  integers is **(Gate-2008) (2 Marks)**

- (A)  $\Theta(n)$
- (B)  $\Theta(\log n)$
- (C)  $\Theta(\log^* n)$
- (D)  $\Theta(1)$

**Q** The minimum number of comparisons required to find the minimum and the maximum of 100 numbers is \_\_\_\_\_. **(Gate-2014) (1 Marks)**

**Q** An unordered list contains  $n$  distinct elements. The number of comparisons to find an element in this list that is neither maximum nor minimum is **(Gate-2015) (1 Marks)**

(A)  $\Theta(n \log n)$

(B)  $\Theta(n)$

(C)  $\Theta(\log n)$

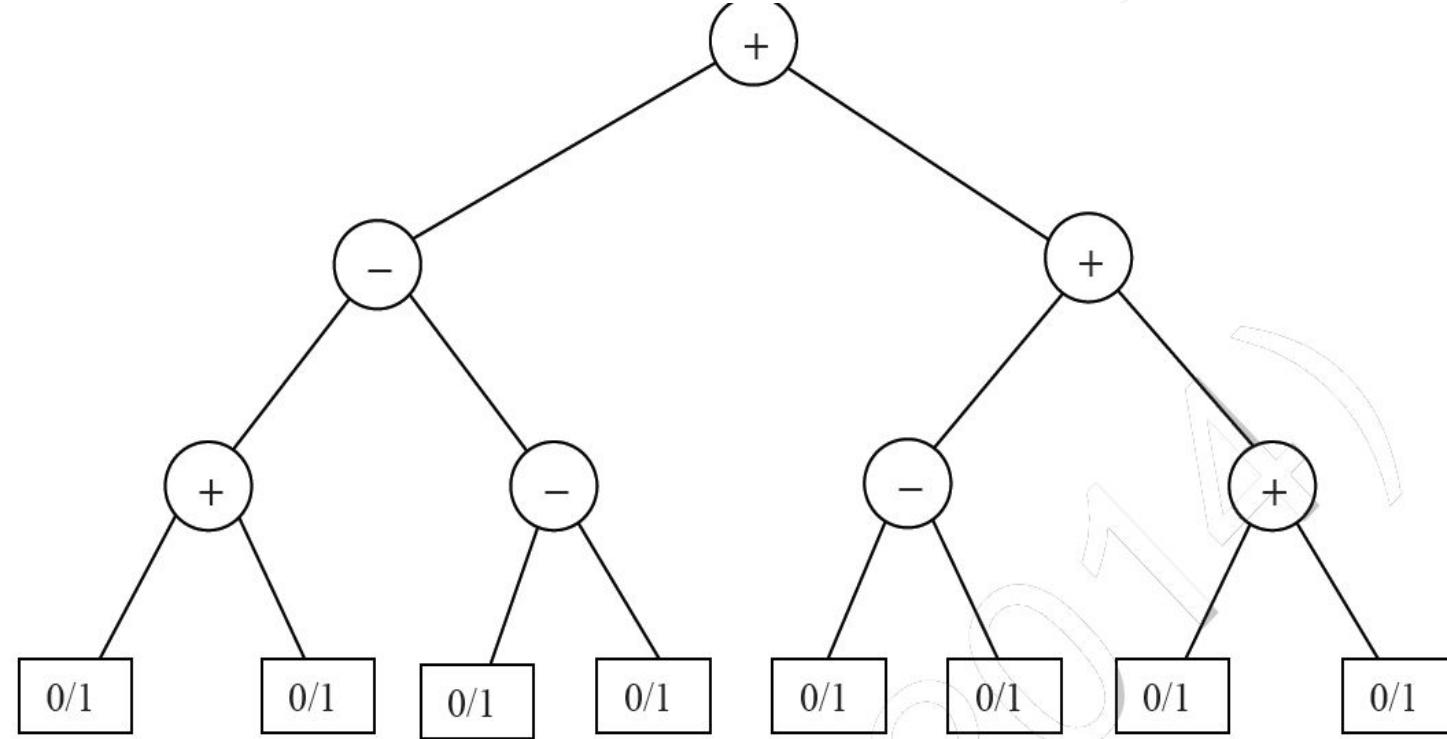
(D)  $\Theta(1)$

**Q Match the algorithms with their time complexities: (Gate-2017) (1 Marks)**

<u>Algorithm</u>	<u>Time complexity</u>
(P) Towers of Hanoi with $n$ disks	(i) $\Theta(n^2)$
(Q) Binary search given $n$ sorted numbers	(ii) $\Theta(n \log n)$
(R) Heap sort given $n$ numbers at the worst case	(iii) $\Theta(2^n)$
(S) Addition of two $n \times n$ matrices	(iv) $\Theta(\log n)$

- a)** P-> (iii), Q -> (iv), R -> (i), S -> (ii)
- b)** P-> (iv), Q -> (iii), R -> (i), S -> (ii)
- c)** P-> (iii), Q -> (iv), R -> (ii), S -> (i)
- d)** P-> (iv), Q -> (iii), R -> (ii), S -> (i)

**Q** Consider the expression tree shown. Each leaf represents a numerical value, which can either be 0 or 1. Over all possible choices of the values at the leaves, the maximum possible value of the expression represented by the tree is \_\_\_\_\_. (Gate-2014) (2 Marks)



**Q** Consider the following array.

23	32	45	69	72	73	89	97
----	----	----	----	----	----	----	----

Which algorithm out of the following options uses the least number of comparisons (among the array elements) to sort the above array in ascending order? **(GATE 2021)**

- (a) Selection sort
- (b) Mergesort
- (c) Insertion sort
- (d) Quicksort using the last element as

**Q** What is the worst-case number of arithmetic operations performed by recursive binary search on a sorted array of size n? **(GATE 2021)(1 MARKS)**

(a)  $\Theta(\sqrt{n})$

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

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

(d)  $\Theta(n)$

**Q.** Let  $A$  be an array containing integer values. The distance of  $A$  is defined as the minimum number of elements in  $A$  that must be replaced with another integer so that the resulting array is sorted in non-decreasing order. The distance of the array [2, 5, 3, 1, 4, 2, 6] is \_\_\_\_\_ **(Gate 2024 CS)(1 Mark)(NAT)**

**Q.** Consider an unordered list of  $N$  distinct integers.

What is the minimum number of element comparisons required to find an integer in the list that is NOT the largest in the list? **(Gate 2025)**

- A) 1
- B)  $N - 1$
- C)  $N$
- D)  $2N - 1$

**Q.** An array  $A$  of length  $n$  with distinct elements is said to be bitonic if there is an index  $1 \leq i \leq n$  such that  $A[1..i]$  is sorted in the non-decreasing order and  $A[i+1..n]$  is sorted in the non-increasing order.

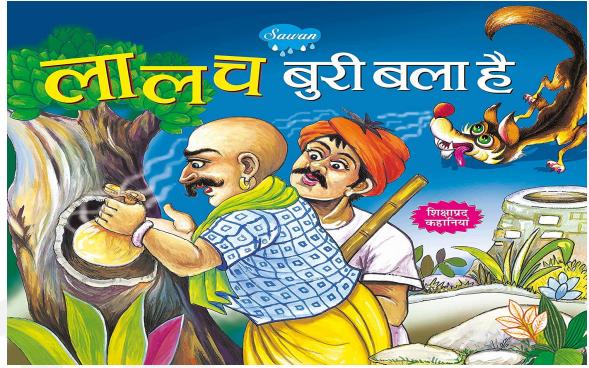
Which ONE of the following represents the best possible asymptotic bound for the worst-case number of comparisons by an algorithm that searches for an element in a bitonic array  $A$ ?

**(Gate 2025)**

- A)  $\Theta(n)$
- B)  $\Theta(1)$
- C)  $\Theta(\log^2 n)$
- D)  $\Theta(\log n)$

# Greedy Algorithm

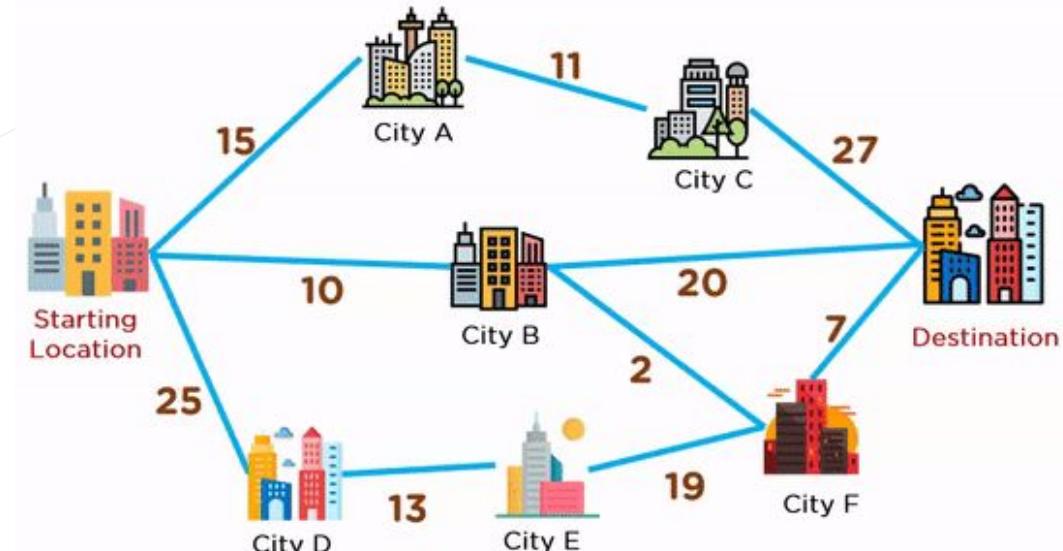
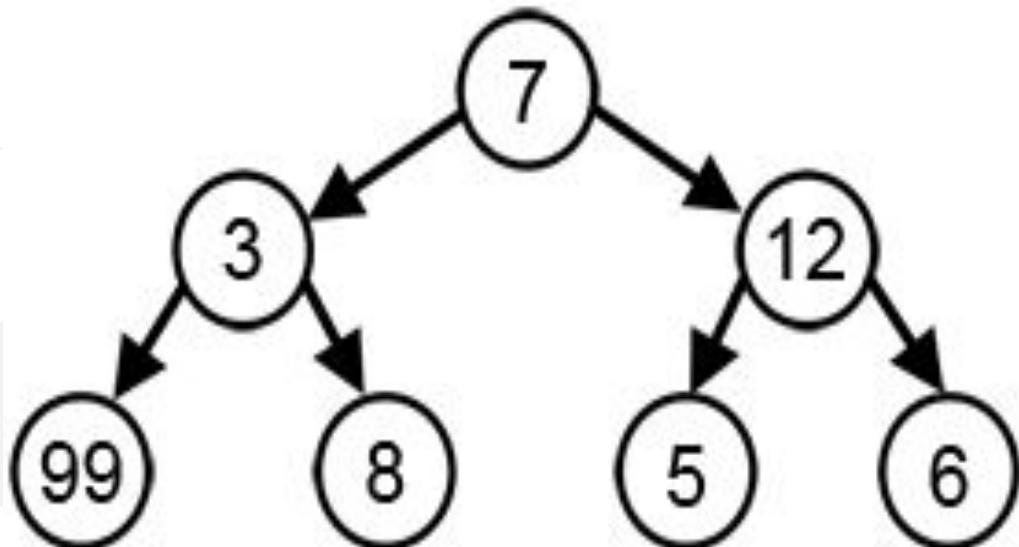
- A **greedy algorithm** is a problem-solving approach like Subtract and conquer, divide and conquer and dynamic programming, which is used for solving optimality problem(one Solution), out of all feasible solution.
- Knapsack Problem
- Job sequencing with Deadline
- Huffman Coding
- Optimal Merge Pattern
- Minimum Spanning Tree
- Single source shortest path



लालच बुरी बला है,  
अगर बुरे काम के लिए किया गया हो तो

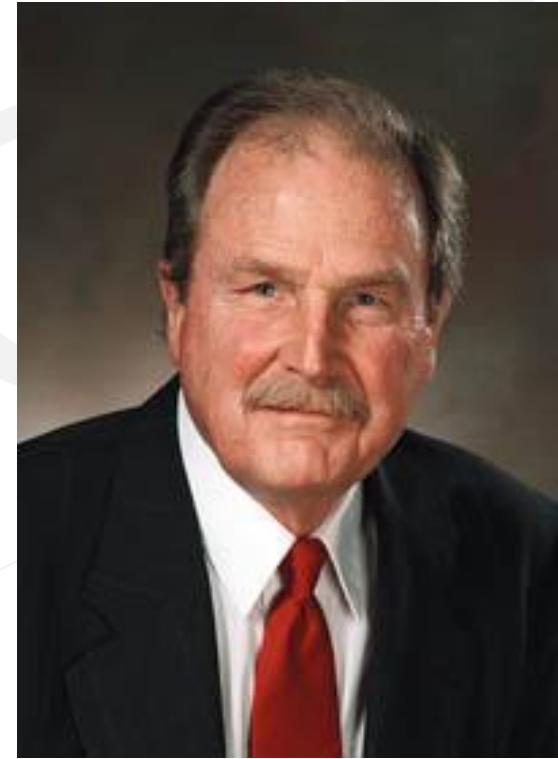
# Greedy Algorithm

- **Definition:** A greedy algorithm makes locally optimal choices at each stage with the hope of finding a global optimum.
- **Effectiveness:** While a greedy strategy doesn't always produce the optimal solution, it often provides good approximations in a reasonable amount of time.
- **Example:** In the Traveling Salesman Problem, a greedy strategy would involve visiting the nearest unvisited city at each step. Though it doesn't guarantee the best solution, it finds a reasonable one quickly.
- **Process:** Greedy algorithms make one decision at a time and don't reconsider previous choices. This is different from **dynamic programming**, which makes decisions based on previous stages and may revise earlier choices.
- **Local vs Global Optimum:** A greedy algorithm may reach a local maximum (e.g., "m") but miss the global maximum (e.g., "M") if the local choice doesn't lead to the global best solution.



# Huffman coding

- **Definition:** Huffman coding is an optimal prefix code used for **lossless data compression**. It minimizes redundancy when encoding data, ensuring that no code is a prefix of another.
- **Development:** The algorithm was developed by **David A. Huffman** while he was a student at MIT in 1952, as a solution to a coding efficiency problem presented by his professor.
- **Process:** Huffman's algorithm builds a **binary tree** based on symbol frequencies. The tree is built from the bottom up, ensuring an optimal coding structure, unlike the top-down approach used in Shannon-Fano coding.
- **Optimality:** It generates a **variable-length code table** based on the frequency of symbols. More frequent symbols get shorter codes, and less frequent symbols get longer codes.
- **Limitations:** While Huffman coding is optimal for encoding symbols separately, it is **not always the most efficient** method compared to other advanced compression techniques that encode multiple symbols together.

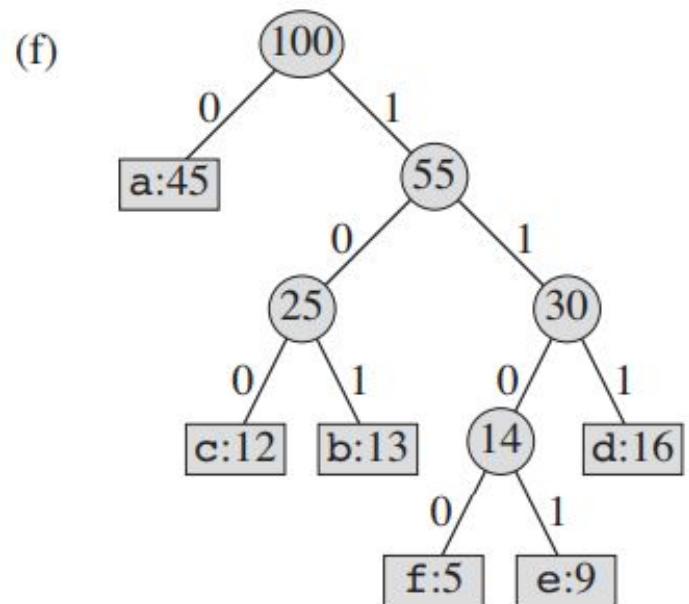
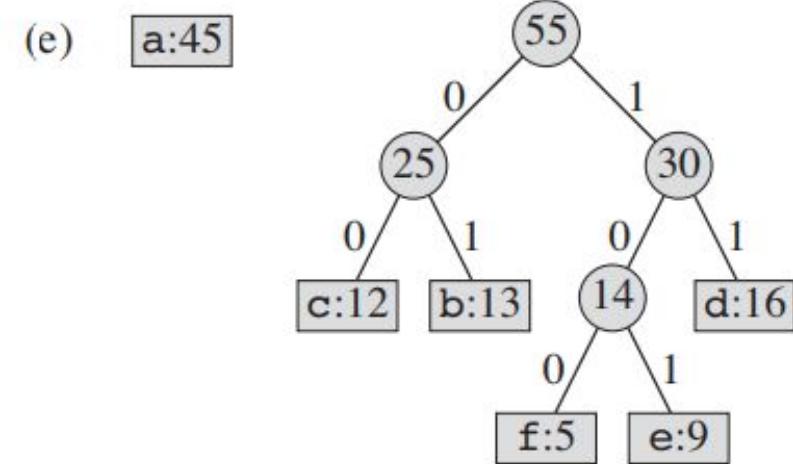
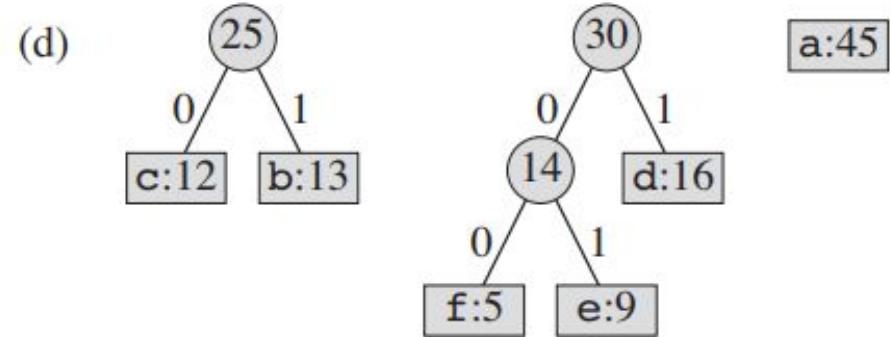
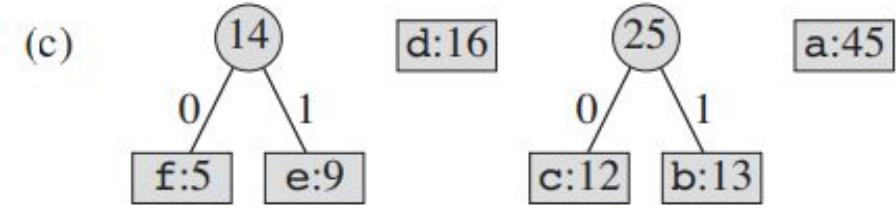


In alphabetical order

A	8.15 %	N	7.10 %
B	1.44 %	O	8.00 %
C	2.76 %	P	1.98 %
D	3.79 %	Q	0.12 %
E	13.11 %	R	6.83 %
F	2.92 %	S	6.10 %
G	1.99 %	T	10.47 %
H	5.26 %	U	2.46 %
I	6.35 %	V	0.92 %
J	0.13 %	W	1.54 %
K	0.42 %	X	0.17 %
L	3.39 %	Y	1.98 %
M	2.54 %	Z	0.08 %

HUFFMAN( $C$ )

```
1   $n = |C|$ 
2   $Q = C$ 
3  for  $i = 1$  to  $n - 1$ 
4      allocate a new node  $z$ 
5       $z.left = x = \text{EXTRACT-MIN}(Q)$ 
6       $z.right = y = \text{EXTRACT-MIN}(Q)$ 
7       $z.freq = x.freq + y.freq$ 
8       $\text{INSERT}(Q, z)$ 
9  return EXTRACT-MIN( $Q$ )    // return the root of the tree
```



**Q** Consider the following character with frequency and generate Huffman tree, find Huffman code for each character, find the number of bits required for a message of 100 characters?

Character	Frequency
M <sub>1</sub>	12
M <sub>2</sub>	4
M <sub>3</sub>	45
M <sub>4</sub>	17
M <sub>5</sub>	23

**Q** A message is made up entirely of characters from the set  $X = \{P, Q, R, S, T\}$ . The table of probabilities for each of the characters is shown below: If a message of 100 characters over  $X$  is encoded using Huffman coding, then the expected length of the encoded message in bits is \_\_\_\_\_.

**(Gate-2017) (2 Marks)**

Character	Probability
P	0.22
Q	0.34
R	0.17
S	0.19
T	0.08
Total	1.00

**Q** Suppose the letters a, b, c, d, e, f has probabilities  $1/2$ ,  $1/4$ ,  $1/8$ ,  $1/16$ ,  $1/32$ ,  $1/32$  respectively. Which of the following is the Huffman code for the letter a, b, c, d, e, f? **(Gate - 2007) (2 Marks)**

- (A) 0, 10, 110, 1110, 11110, 11111
- (B) 11, 10, 011, 010, 001, 000
- (C) 11, 10, 01, 001, 0001, 0000
- (D) 110, 100, 010, 000, 001, 111

**Q** Suppose the letters a, b, c, d, e, f has probabilities  $1/2$ ,  $1/4$ ,  $1/8$ ,  $1/16$ ,  $1/32$ ,  $1/32$  respectively. What is the average length of Huffman codes? **(Gate - 2007) (2 Marks)**

- (A) 3      (B) 2.1875      (C) 2.25      (D) 1.9375

**Q** The characters a to h have the set of frequencies based on the first 8 Fibonacci numbers as follows

a : 1, b : 1, c : 2, d : 3, e : 5, f : 8, g : 13, h : 21

A Huffman code is used to represent the characters. What is the sequence of characters corresponding to the following code? **(Gate-2006) (2 Marks)**

110111100111010

- (A) fdheg
- (B) ecgdf
- (C) dchfg
- (D) fehdg

**Q** A networking company uses a compression technique to encode the message before transmitting over the network. Suppose the message contains the following characters with their frequency: If the compression technique used is Huffman Coding, how many bits will be saved in the message?

(A) 224

(B) 800

(C) 576

(D) 324

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

**Q** In question #2, which of the following represents the word “dead”?

(A) 1011111100101

(B) 0100000011010

(C) Both A and B

(D) None of these

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

**Q** What is the time complexity of Huffman Coding?

**(A)**  $O(N)$

**(B)**  $O(N \log N)$

**(C)**  $O(N(\log N)^2)$

**(D)**  $O(N^2)$

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

**Q** Consider the string abbccddeee. Each letter in the string must be assigned a binary code satisfying the following properties:

- For any two letters, the code assigned to one letter must not be a prefix of the code assigned to the other letter.
- For any two letters of the same frequency, the letter which occurs earlier in the dictionary order is assigned a code whose length is at most the length of the code assigned to the other letter.

Among the set of all binary code assignments which satisfy the above two properties, what is the minimum length of the encoded string? **(GATE 2021) (2 MARKS)**

- (A) 21
- (B) 23
- (C) 25
- (D) 30

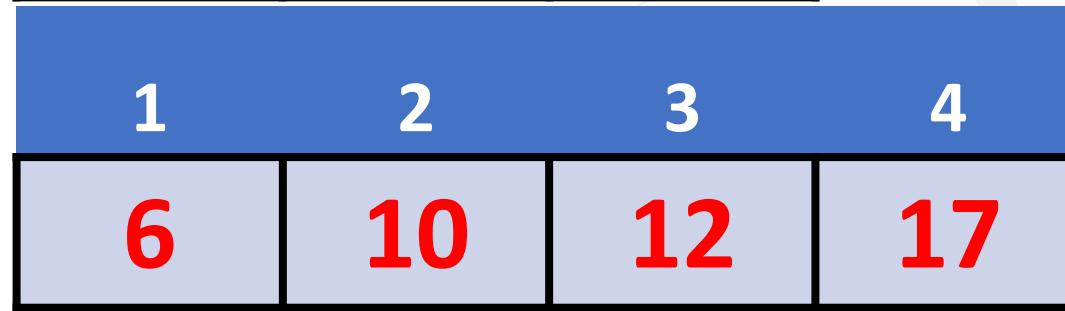
## Optimal Merge Pattern

**Problem:** Merging of two sorted list into a single sorted list.

A



B



C



Total time/Record movement =

# **Problem:** Merging multiple sorted list into a single sorted list

A	B	C	D
2	3	5	7

A	B	C	D
2	3	5	7

A	B	C	D
2	3	5	7

Case1: Total time/Record movement =

Case2: Total time/Record movement =

Case3: Total time/Record movement =

## Optimal Merge Pattern

- **Definition:** An optimal merge pattern involves merging two or more sorted files into a single sorted file using a two-way merge with the minimum number of record movements or computations.
- **Computation Time:** If two files of size  $m$  and  $n$  are merged, the total computation time will be  $m+n + nm + n$ .
- **Greedy Strategy:** The greedy strategy is used by always merging the two smallest size files first, ensuring the minimum total computation time.
- **Conclusion:** This method ensures an optimal solution by minimizing the overall number of computations and movements during the merge process.

**Q** Given a set of 8 files from  $F_1$  to  $F_8$  with following number of pages find the minimum number of record movements to merge them into single file?

$F_1$	$F_2$	$F_3$	$F_4$	$F_5$	$F_6$	$F_7$	$F_8$
18	3	15	12	10	11	7	9

**Q** Suppose P, Q, R, S, T are sorted sequences having lengths 20, 24, 30, 35, 50 respectively. They are to be merged into a single sequence by merging together two sequences at a time. The number of comparisons that will be needed in the worst case by the optimal algorithm for doing this is \_\_\_\_\_. **(Gate-2014) (2 Marks)**

# Knap Sack Problem

- **Definition:** The knapsack problem is a combinatorial optimization problem where given a set of items with specific weights and values, you need to determine the best combination of items to include in a knapsack such that the total weight is within a given limit, and the total value is maximized.
- **Versions:** There are two versions of the problem:
- **Fractional Knapsack:** Items can be divided, meaning fractions of an item can be selected.
- **0/1 Knapsack:** Items cannot be divided; you either take the whole item or none.
- **History:** The problem has been studied for over a century, with some of the earliest research dating back to 1897. The name is attributed to mathematician **Tobias Dantzig**.



x<sub>1</sub>

$x_2$

X<sub>3</sub>

X<sub>N-1</sub>

X N

**Q** Consider the weights and values of items listed below. Note that there is only one unit of each item.

Greedy by Profit

Object	O <sub>1</sub>	O <sub>2</sub>	O <sub>3</sub>
Profit	25	24	15
Weight	18	15	10

Object	O <sub>1</sub>	O <sub>2</sub>	O <sub>3</sub>
Profit	25	24	15
Weight	18	15	10
Solution			

**Q** Consider the weights and values of items listed below. Note that there is only one unit of each item.

Greedy by Weight

Object	O <sub>1</sub>	O <sub>2</sub>	O <sub>3</sub>
Profit	25	24	15
Weight	18	15	10

Object	O <sub>1</sub>	O <sub>2</sub>	O <sub>3</sub>
Profit	25	24	15
Weight	18	15	10
Solution			

**Q** Consider the weights and values of items listed below. Note that there is only one unit of each item.

Greedy by Profit/Weight

Object	O <sub>1</sub>	O <sub>2</sub>	O <sub>3</sub>
Profit	25	24	15
Weight	18	15	10

Object	O <sub>1</sub>	O <sub>2</sub>	O <sub>3</sub>
Profit	25	24	15
Weight	18	15	10
Profit/Weight	1.38	1.6	1.5
Solution			

- **Applications**: The problem is commonly encountered in **resource allocation** scenarios with financial constraints and has applications in fields like:
  - Combinatorics
  - Computer science
  - Complexity theory
  - Cryptography
  - Applied mathematics
- **Conclusion**: The knapsack problem models real-life situations where a limited capacity (e.g., knapsack) needs to be filled with the most valuable items, maximizing utility while adhering to constraints.

**Q** Consider the weights and values of items listed below. Note that there is only one unit of each item. The task is to pick a subset of these items such that their total weight is no more than 11 Kgs and their total value is maximized. Moreover, no item may be split. The total value of items picked by an optimal algorithm is denoted by  $V_{opt}$ . A greedy algorithm sorts the items by their value-to-weight ratios in descending order and packs them greedily, starting from the first item in the ordered list. The total value of items picked by the greedy algorithm is denoted by  $V_{greedy}$ . The value of  $V_{opt} - V_{greedy}$  is \_\_\_\_\_.

**(Gate-2018) (2 Marks)**

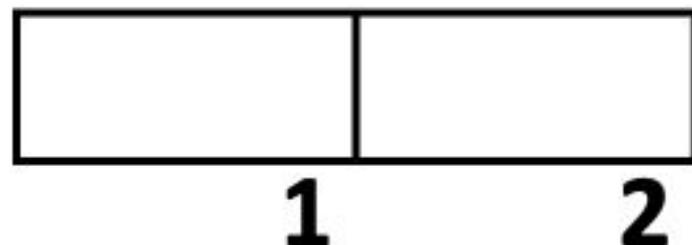
Item number	Weight (in Kgs)	Value (in Rupees)
1	10	60
2	7	28
3	4	20
4	2	24

# Job sequencing with Deadline

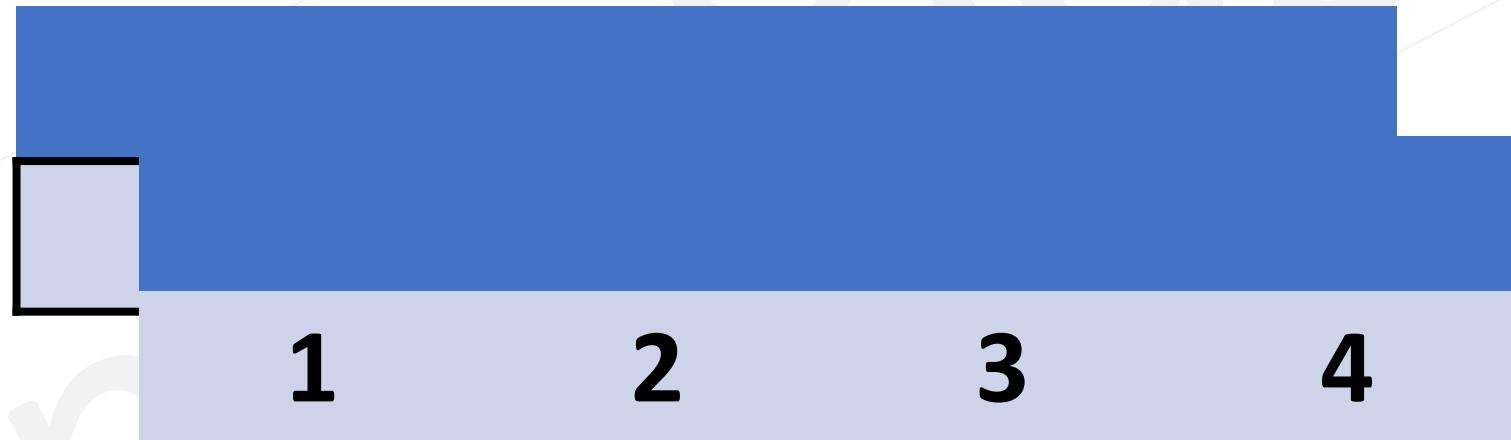
- **Problem Definition:**
  - Given  $n$  jobs, each with a deadline  $D_i$  and profit  $P_i$  if completed before the deadline.
  - The goal is to schedule jobs on a single CPU with non-preemptive scheduling to maximize profit.
- **Assumptions:**
  - All jobs arrive at time 0.
  - Each job takes exactly 1 time unit to complete (burst time = 1).
- **Objective:**
  - Select a subset of jobs such that they can all be completed within their respective deadlines and the total profit is maximized.

**Q** if we have for task  $T_1, T_2, T_3, T_4$ , having Deadline  $D_1 = 2, D_2 = 1, D_3 = 2, D_4 = 1$ , and profit  $P_1 = 100, P_2 = 10, P_3 = 27, P_4 = 15$ , find the maximum profit possible?

Task	$T_1$	$T_2$	$T_3$	$T_4$
Profit	100	10	27	15
Deadline	2	1	2	1



Task	$T_1$	$T_2$	$T_3$	$T_4$	$T_5$	$T_6$	$T_7$
Profit	35	30	25	20	15	12	5
Deadline	3	4	4	2	3	1	2



- **Solution Approach:**
  - **Step 1:** Sort jobs in decreasing order of profit.
  - **Step 2:** For each job in this order, find the latest available time slot  $i$  such that  $i < D_i$  and the slot is empty.
  - **Step 3:** Assign the job to the slot and mark the slot as filled.
  - **Step 4:** If no such slot is available, ignore the job.
- **Conclusion:** The greedy strategy of scheduling the most profitable jobs first, within available time slots, ensures the maximization of profit.

**Q** We are given 9 tasks T<sub>1</sub>, T<sub>2</sub>... T<sub>9</sub>. The execution of each task requires one unit of time. We can execute one task at a time. Each task T<sub>i</sub> has a profit P<sub>i</sub> and a deadline d<sub>i</sub>. Profit P<sub>i</sub> is earned if the task is completed before the end of the d<sub>i</sub>th unit of time.

Task	T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>	T <sub>5</sub>	T <sub>6</sub>	T <sub>7</sub>	T <sub>8</sub>	T <sub>9</sub>
Profit	15	20	30	18	18	10	23	16	25
Deadline	7	2	5	3	4	5	2	7	3

Are all tasks completed in the schedule that gives maximum profit? (Gate-2005) (2 Marks)

- (A) All tasks are completed
- (B) T<sub>1</sub> and T<sub>6</sub> are left out
- (C) T<sub>1</sub> and T<sub>8</sub> are left out
- (D) T<sub>4</sub> and T<sub>6</sub> are left out

Q We are given 9 tasks T1, T2.... T9. The execution of each task requires one unit of time. We can execute one task at a time. Each task  $T_i$  has a profit  $P_i$  and a deadline  $d_i$ . Profit  $P_i$  is earned if the task is completed before the end of the  $\frac{d_i}{\text{unit}}$  unit of time.

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

What is the maximum profit earned? (Gate-2005) (2 Marks)

- (A) 147      (B) 165      (C) 167      (D) 175

**Q** Consider  $n$  jobs  $J_1, J_2, \dots, J_n$  such that job  $J_i$  has execution time  $t_i$  and a non-negative integer weight  $w_i$ . The weighted mean completion time of the jobs is defined to be,

$$\frac{\sum_{i=1}^n w_i T_i}{\sum_{i=1}^n w_i},$$

where  $T_i$  is the completion time of job  $J_i$ . Assuming that there is only one processor available, in what order must the jobs be executed in order to minimize the weighted mean completion time of the jobs? **(Gate-2007) (2 Marks)**

- (A) Non-decreasing order of  $t_i$
- (B) Non-increasing order of  $w_i$
- (C) Non-increasing order of  $w_i t_i$
- (D) None-increasing order of  $w_i/t_i$

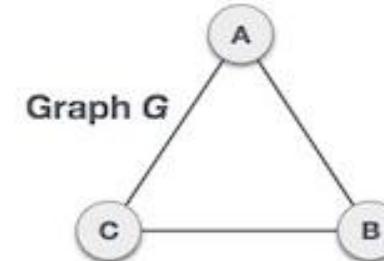
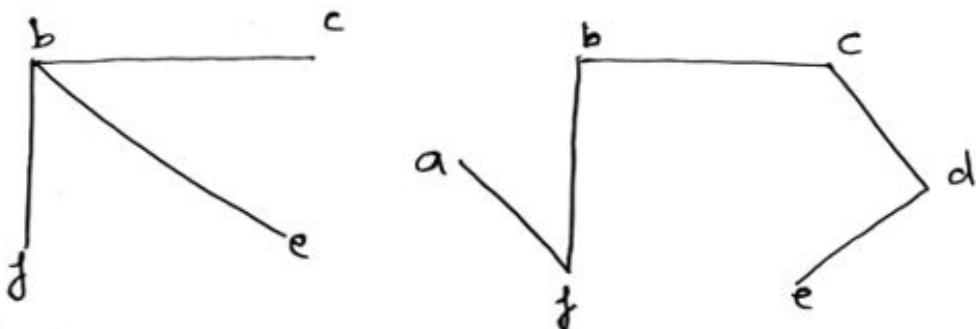
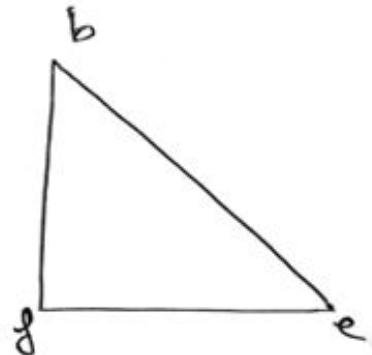
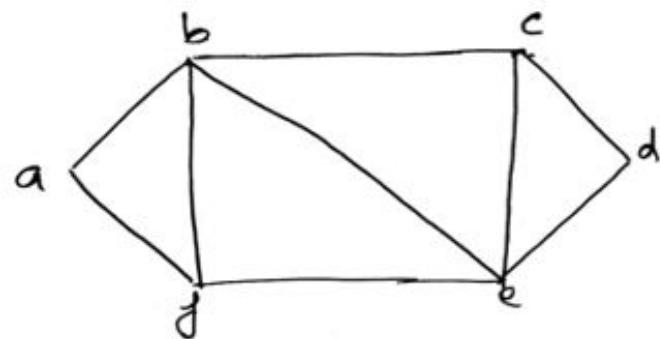
**Q** The following are the starting and ending times of activities A, B, C, D, E, F, G and H respectively in chronological order: “ $a_s b_s c_s a_e d_s c_e e_s f_s b_e d_e g_s e_e f_e h_s g_e h_e$ ” Here,  $x_s$  denotes the starting time and  $x_e$  denotes the ending time of activity X. We need to schedule the activities in a set of rooms available to us. An activity can be scheduled in a room only if the room is reserved for the activity for its entire duration. What is the minimum number of rooms required?

**(Gate-2003) (2 Marks)**

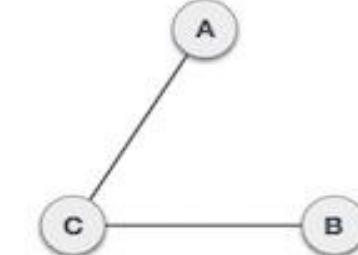
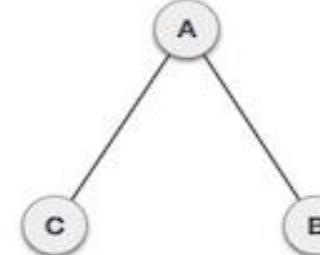
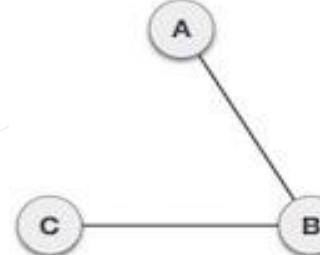
- (A) 3
- (B) 4
- (C) 5
- (D) 6

# Spanning tree

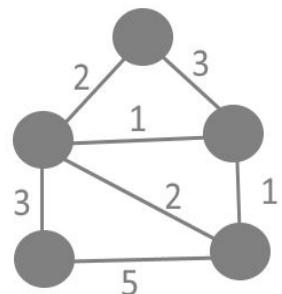
- A tree  $T$  is said to be spanning tree of a connected graph  $G$ , if  $T$  is a subgraph of  $G$  and  $T$  contains all vertices of  $G$ .



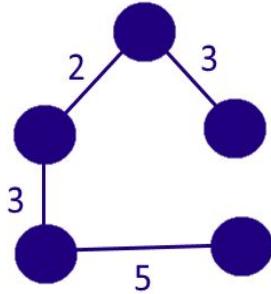
Spanning Trees



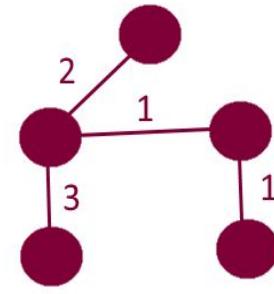
- A **minimum spanning tree (MST)** or **minimum weight spanning tree** is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the **minimum possible total edge weight**.



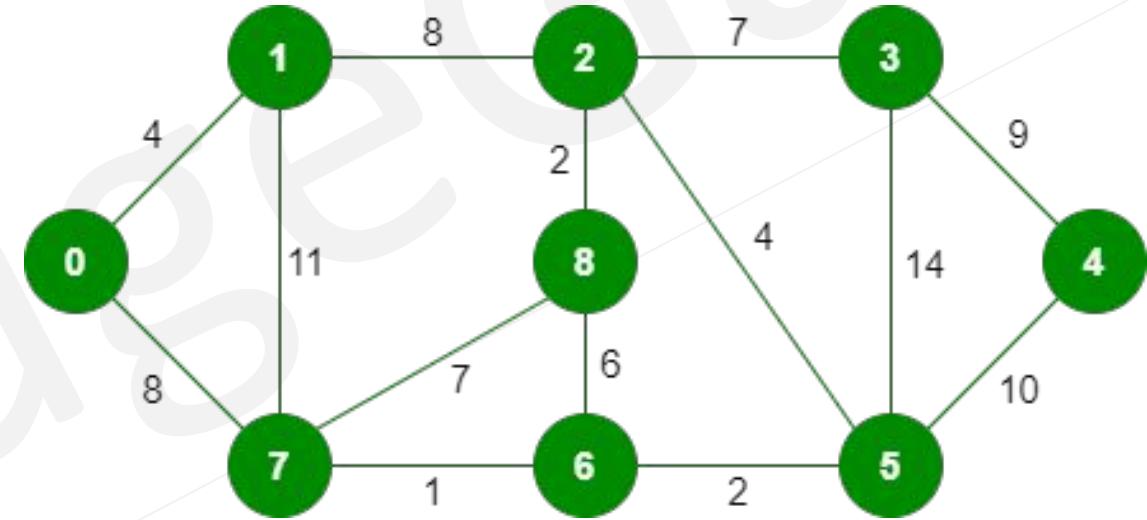
Graph



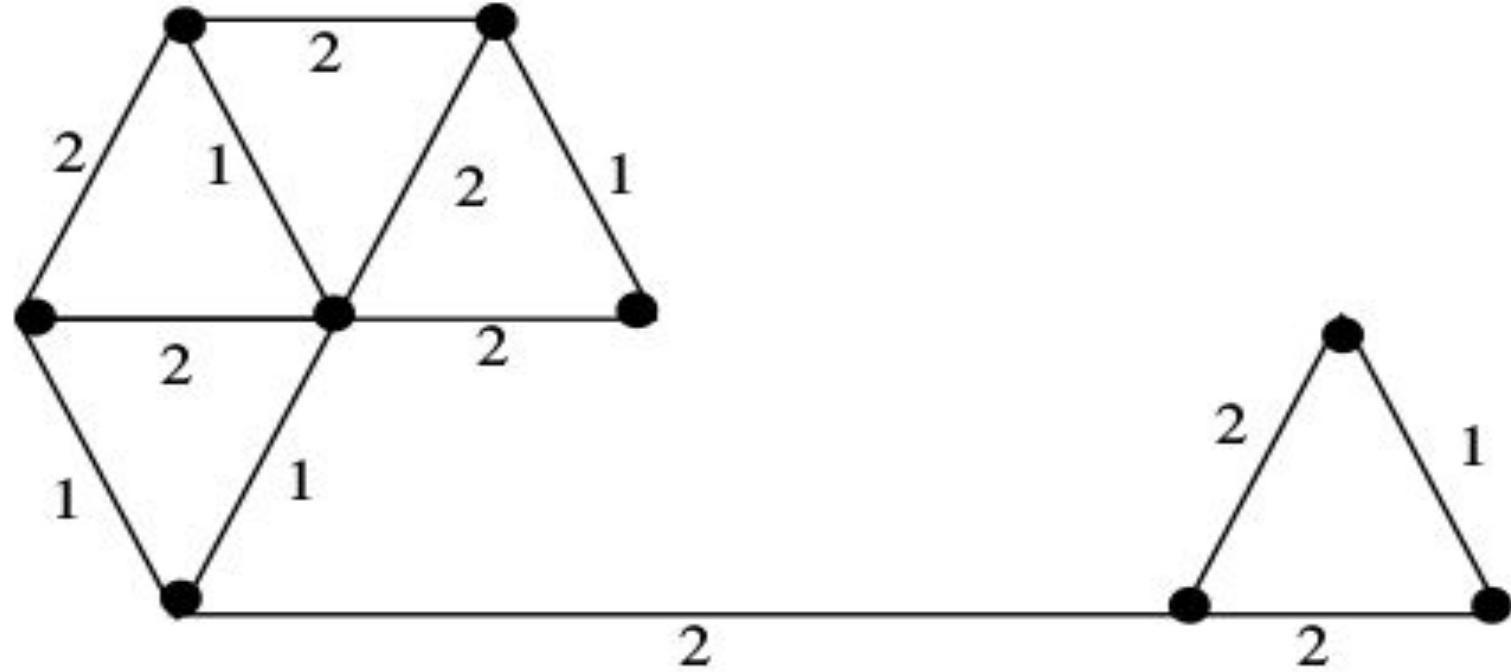
Spanning Tree  
Cost = 13



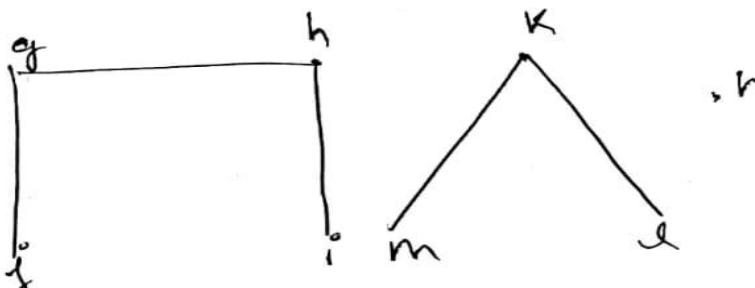
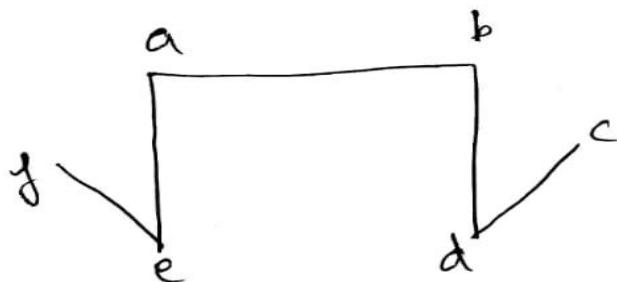
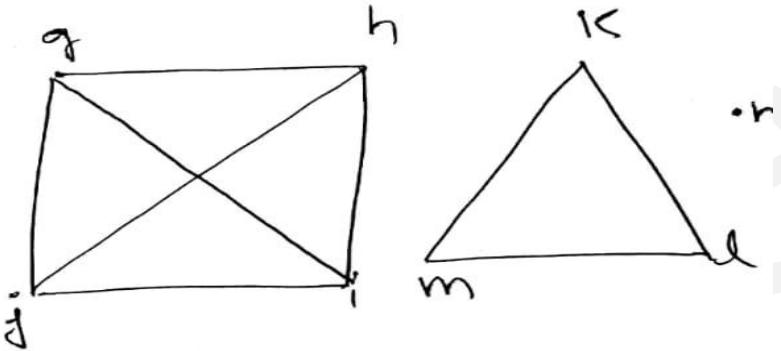
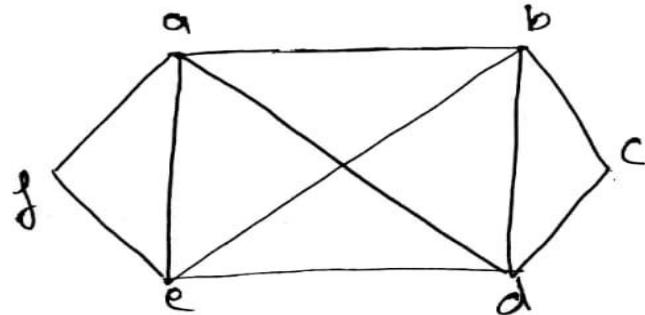
Minimum Spanning  
Tree, Cost = 7



- Minimum spanning tree (MST) can be more than one



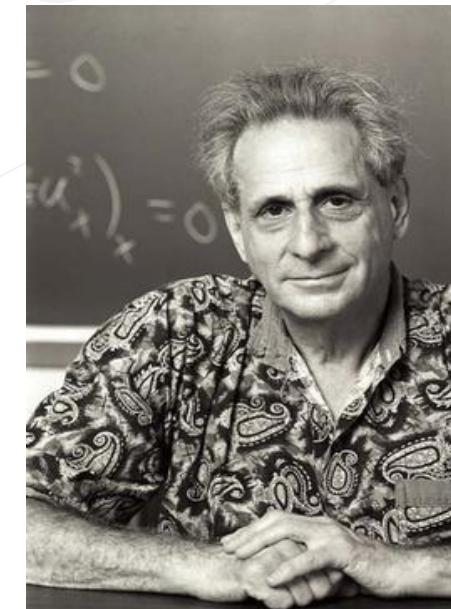
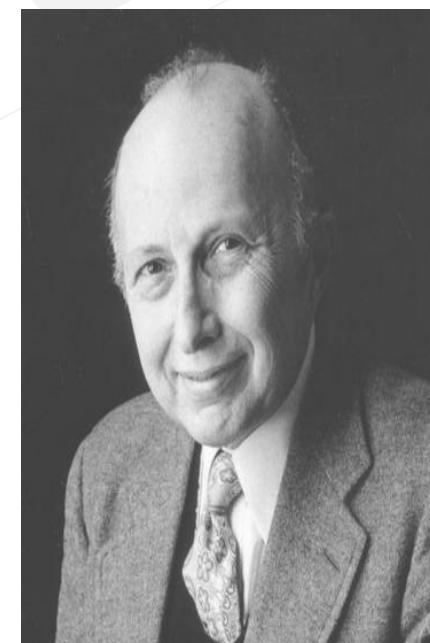
- In general, a graph may have several spanning trees, but a graph that is not connected will not contain a spanning tree (but see Spanning forests below).



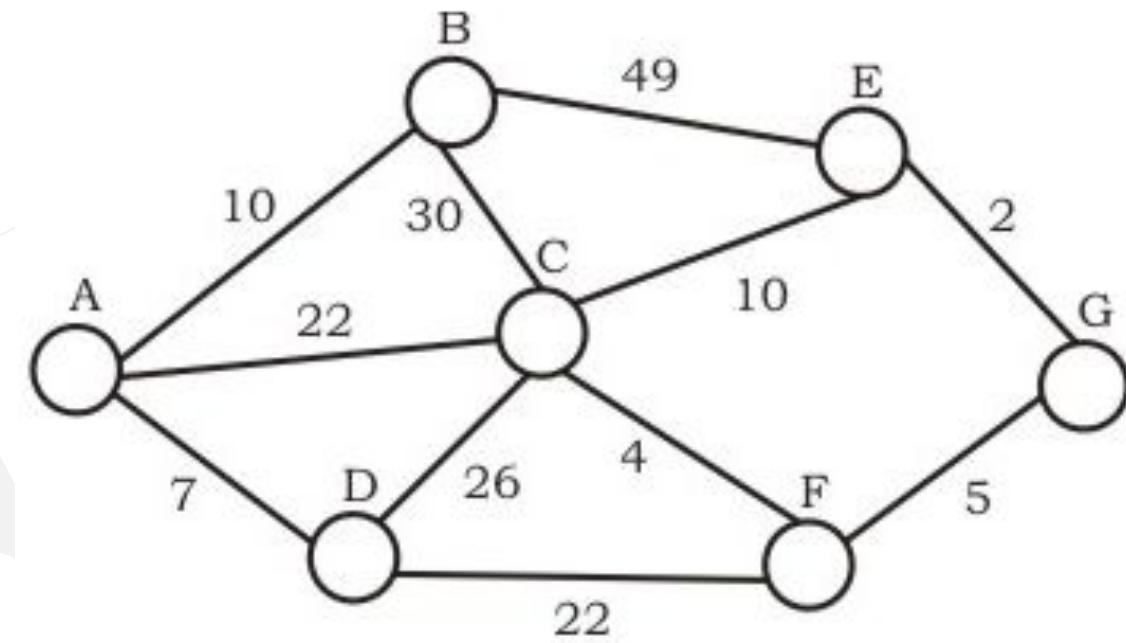
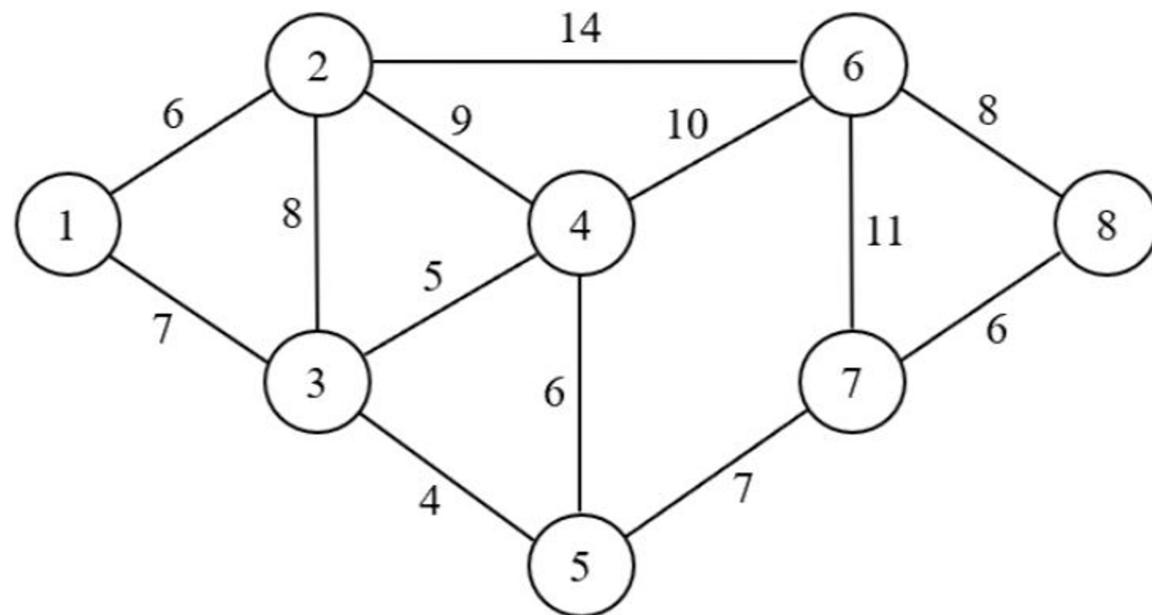
- Any edge-weighted undirected graph (not necessarily connected) has a **minimum spanning forest**, which is a union of the minimum spanning trees for its connected components.

# Kruskal Algorithm

- **Joseph Bernard Kruskal, Jr.** was an American mathematician, statistician, computer scientist and psychometrician.
- **Martin David Kruskal** (September 28, 1925 – December 26, 2006) was an American mathematician and physicist. He made fundamental contributions in many areas of mathematics and science, ranging from plasma physics to general relativity and from nonlinear analysis to asymptotic analysis. His most celebrated contribution was in the theory of solitons.
- **William Henry Kruskal** (October 10, 1919 – April 21, 2005) was an American mathematician and statistician. He is best known for having formulated the Kruskal–Wallis one-way analysis of variance (together with W. Allen Wallis), a widely used nonparametric statistical method.

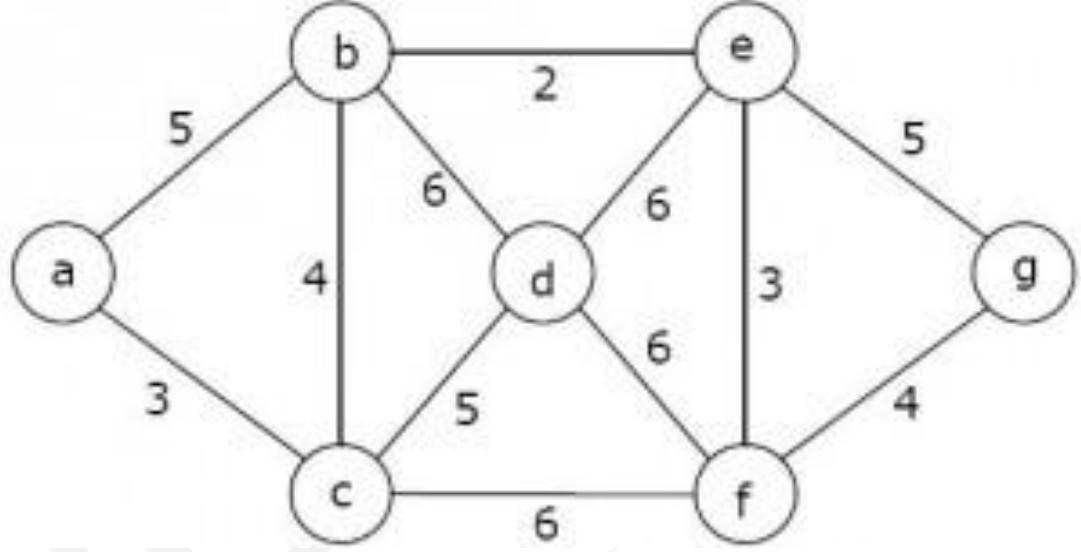


- Kruskal Algorithm Actual idea



# Kruskal

```
Minimum_Spanning_Tree (G, w)
{
    A ⊂ φ
    For each vertex v ∈ V(G)
    {
        do Make_Set(v)
    }
    Sort the edges of E into non-decreasing order by weight w
    for each edge (u, v) ∈ E, then in non-decreasing order by weights
    {
        if (Find_Set(u) != Find_Set(v))
        {
            A ⊂ A ∪ {(u, v)}
            UNION (u, v)
        }
    }
    Return A
}
```



MST-KRUSKAL( $G, w$ )

```
1   $A = \emptyset$ 
2  for each vertex  $v \in G.V$ 
3      MAKE-SET( $v$ )
4  sort the edges of  $G.E$  into nondecreasing order by weight  $w$ 
5  for each edge  $(u, v) \in G.E$ , taken in nondecreasing order by weight
6      if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
7           $A = A \cup \{(u, v)\}$ 
8          UNION( $u, v$ )
9  return  $A$ 
```

The running time of Kruskal's algorithm for a graph  $G = (V, E)$  depends on how we implement the disjoint-set data structure. We assume that we use the disjoint-set-forest implementation of Section 21.3 with the union-by-rank and path-compression heuristics, since it is the asymptotically fastest implementation known. Initializing the set  $A$  in line 1 takes  $O(1)$  time, and the time to sort the edges in line 4 is  $O(E \lg E)$ . (We will account for the cost of the  $|V|$  MAKE-SET operations in the **for** loop of lines 2–3 in a moment.) The **for** loop of lines 5–8 performs  $O(E)$  FIND-SET and UNION operations on the disjoint-set forest. Along with the  $|V|$  MAKE-SET operations, these take a total of  $O((V + E) \alpha(V))$  time, where  $\alpha$  is the very slowly growing function defined in Section 21.4. Because we assume that  $G$  is connected, we have  $|E| \geq |V| - 1$ , and so the disjoint-set operations take  $O(E \alpha(V))$  time. Moreover, since  $\alpha(|V|) = O(\lg V) = O(\lg E)$ , the total running time of Kruskal's algorithm is  $O(E \lg E)$ . Observing that  $|E| < |V|^2$ , we have  $\lg |E| = O(\lg V)$ , and so we can restate the running time of Kruskal's algorithm as  $O(E \lg V)$ .

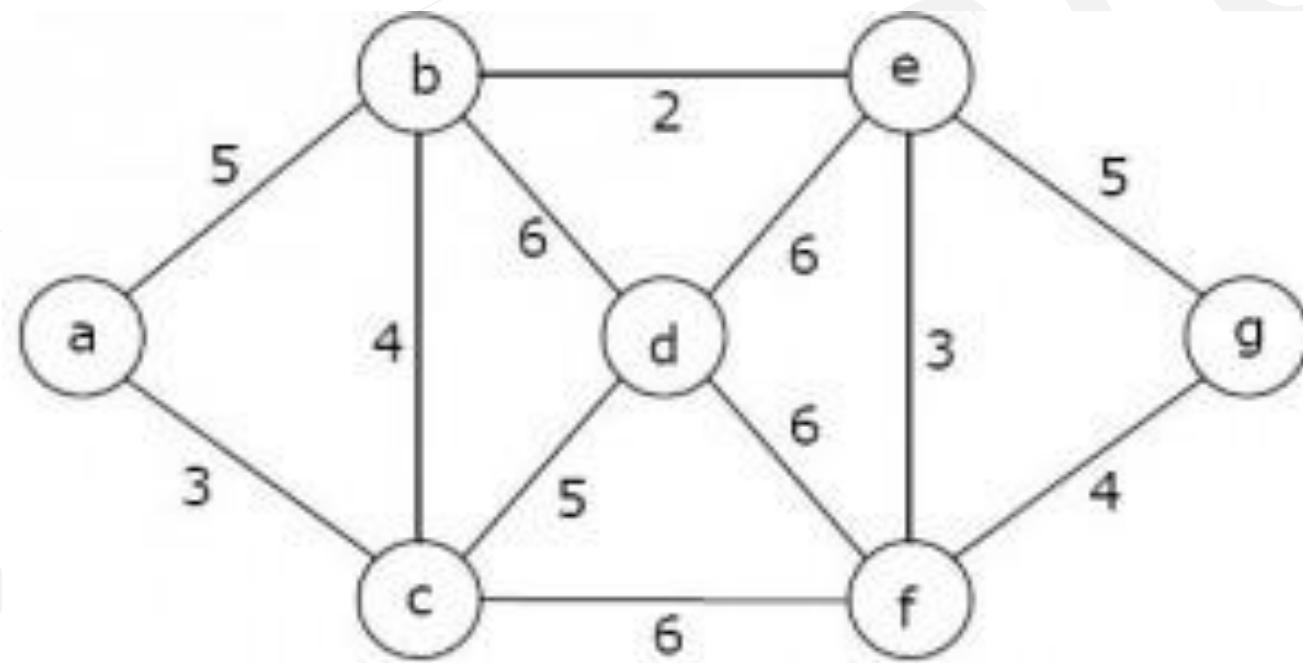
**Q** Consider the following graph: Which one of the following is NOT the sequence of edges added to the minimum spanning tree using Kruskal's algorithm? **(Gate-2009)(2 Marks)**

**(A) (b, e),(e, f),(a, c),(b, c),(f, g),(c, d)**

**(B) (b, e),(e, f),(a, c),(f, g),(b, c),(c, d)**

**(C) (b, e),(a, c),(e, f),(b, c),(f, g),(c, d)**

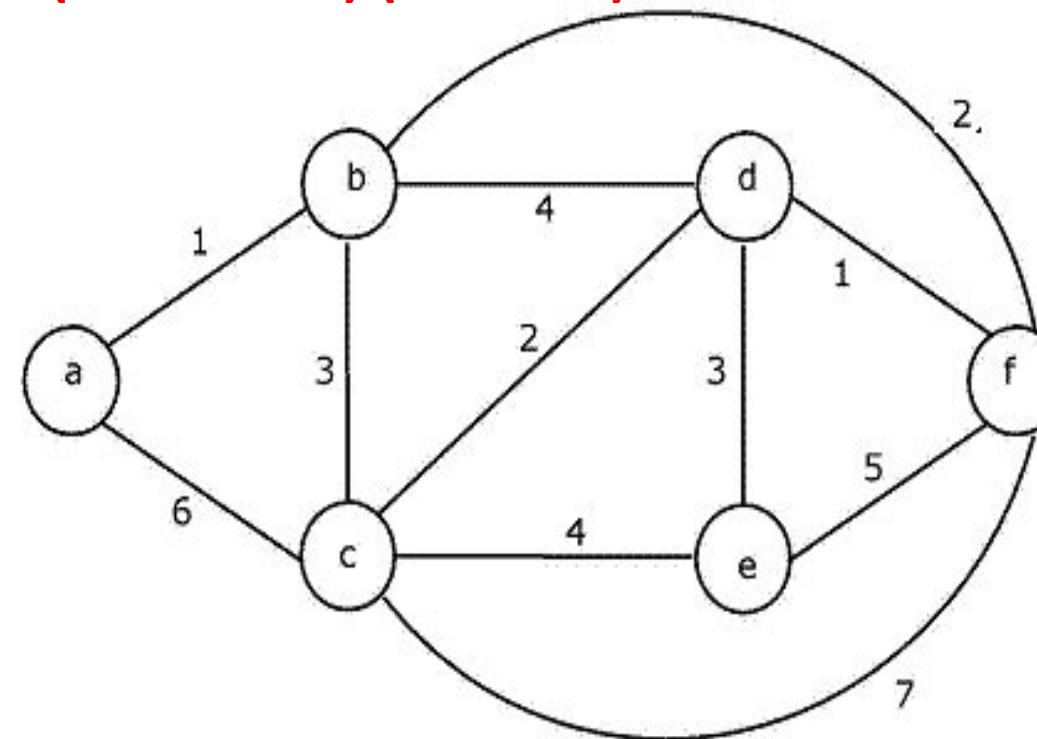
**(D) (b, e),(e, f),(b, c),(a, c),(f, g),(c, d)**



**Q** Consider the following graph:

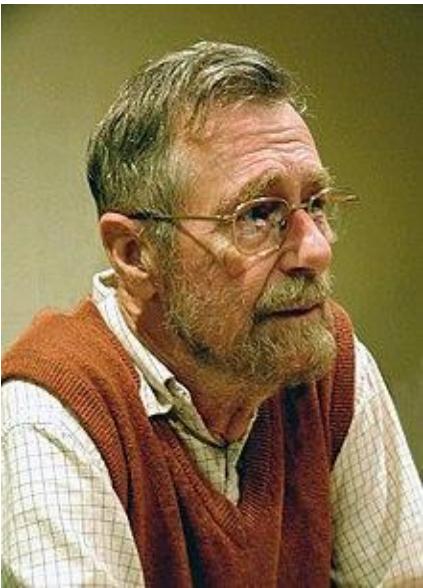
Which one of the following cannot be the sequence of edges added, in that order, to a minimum spanning tree using Kruskal's algorithm? **(Gate-2006) (2 Marks)**

- (A) (a-b),(d-f),(b-f),(d-c),(d-e)
- (B) (a-b),(d-f),(d-c),(b-f),(d-e)
- (C) (d-f),(a-b),(d-c),(b-f),(d-e)
- (D) (d-f),(a-b),(b-f),(d-e),(d-c)



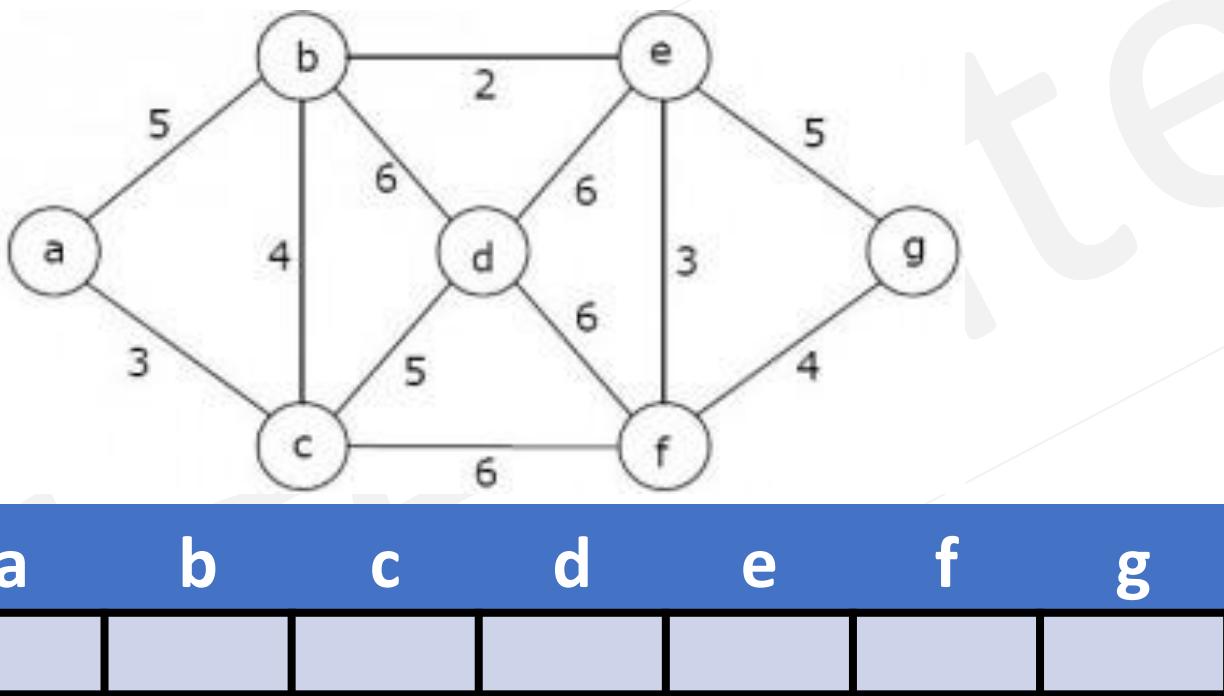
## Prim's Algorithm

- The algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník, later rediscovered and republished by computer scientists Robert C. Prim in 1957 and Edsger W. Dijkstra in 1959. Therefore, it is also sometimes called the **Jarník's algorithm**, **Prim–Jarník algorithm**, **Prim–Dijkstra algorithm** or the **DJP algorithm**.
- **Vojtěch Jarník** (1897–1970) was a Czech mathematician who worked for many years as a professor and administrator at Charles University, and helped found the Czechoslovak Academy of Sciences. He is the namesake of Jarník's algorithm for minimum spanning trees.
- **Robert Clay Prim** (born September 25, 1921 in Sweetwater, Texas) is an American mathematician and computer scientist.
- **Edsger Wybe Dijkstra** (11 May 1930 – 6 August 2002) was a Dutch computer scientist, programmer, software engineer, systems scientist, and science essayist. He received the 1972 Turing Award for fundamental contributions to developing programming languages.



## Minimum\_Spanning\_Tree (G, W, R)

```
{  
    for each  $u \in V(G)$   
    {  
        key[u]  $\leftarrow \infty$   
         $\pi[u] \leftarrow \text{NIL}$   
    }  
    Key[r]  $\leftarrow 0$   
    Q  $\leftarrow V[G]$   
    While ( $Q \neq \emptyset$ )  
    {  
        u  $\leftarrow \text{Extract-Min}(Q)$   
        For each  $v \in \text{adj}[u]$   
        {  
            if ( $v \in Q$  and  $w(u, v) < \text{key}[v]$ )  
            {  
                 $\pi[v] \leftarrow u$   
                key[v]  $\leftarrow w(u, v)$   
            }  
        }  
    }  
}
```

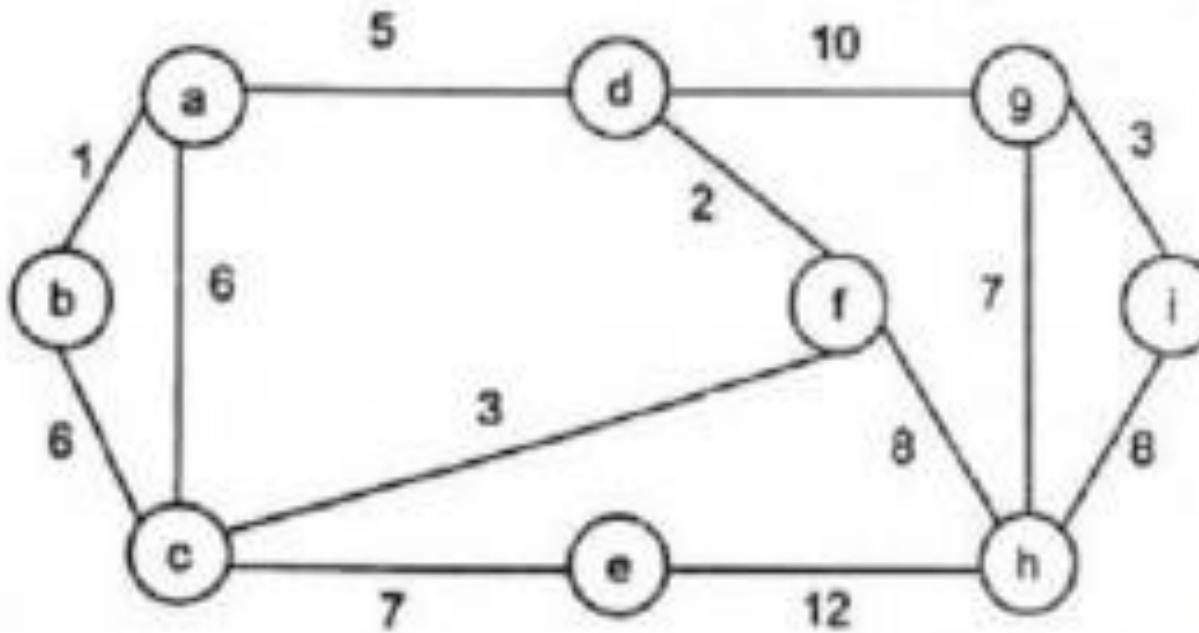


MST-PRIM( $G, w, r$ )

```
1  for each  $u \in G.V$ 
2       $u.key = \infty$ 
3       $u.\pi = \text{NIL}$ 
4   $r.key = 0$ 
5   $Q = G.V$ 
6  while  $Q \neq \emptyset$ 
7       $u = \text{EXTRACT-MIN}(Q)$ 
8      for each  $v \in G.Adj[u]$ 
9          if  $v \in Q$  and  $w(u, v) < v.key$ 
10          $v.\pi = u$ 
11          $v.key = w(u, v)$ 
```

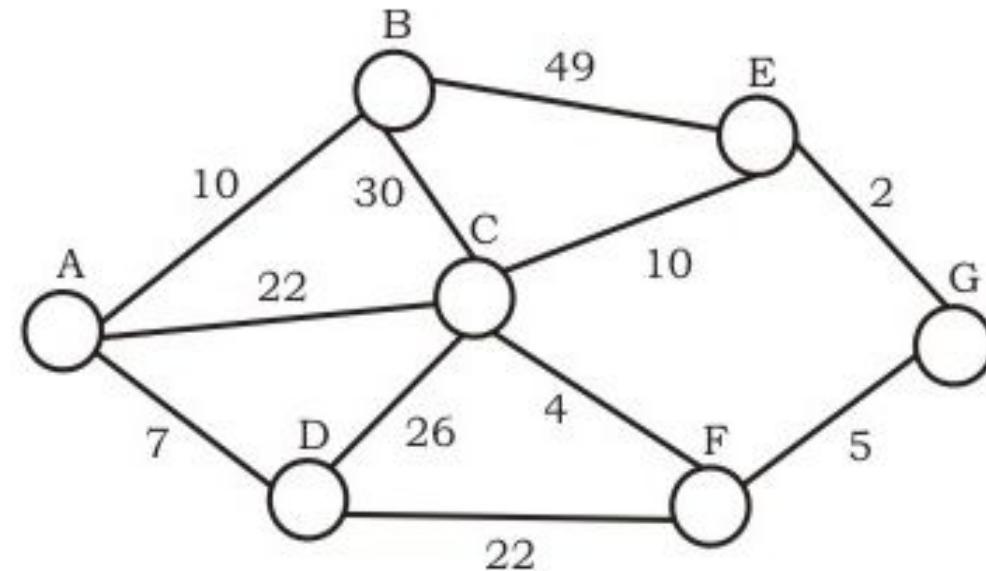
The running time of Prim's algorithm depends on how we implement the min-priority queue  $Q$ . If we implement  $Q$  as a binary min-heap (see Chapter 6), we can use the BUILD-MIN-HEAP procedure to perform lines 1–5 in  $O(V)$  time. The body of the **while** loop executes  $|V|$  times, and since each EXTRACT-MIN operation takes  $O(\lg V)$  time, the total time for all calls to EXTRACT-MIN is  $O(V \lg V)$ . The **for** loop in lines 8–11 executes  $O(E)$  times altogether, since the sum of the lengths of all adjacency lists is  $2|E|$ . Within the **for** loop, we can implement the test for membership in  $Q$  in line 9 in constant time by keeping a bit for each vertex that tells whether or not it is in  $Q$ , and updating the bit when the vertex is removed from  $Q$ . The assignment in line 11 involves an implicit DECREASE-KEY operation on the min-heap, which a binary min-heap supports in  $O(\lg V)$  time. Thus, the total time for Prim's algorithm is  $O(V \lg V + E \lg V) = O(E \lg V)$ , which is asymptotically the same as for our implementation of Kruskal's algorithm.

**Q** For the undirected, weighted graph given below, which of the following sequences of edges represents a correct execution of Prim's algorithm to construct a Minimum Spanning Tree?  
**(Gate-2008) (2 Marks)**



- (A) (a, b), (d, f), (f, c), (g, i), (d, a), (g, h), (c, e), (f, h)
- (B) (c, e), (c, f), (f, d), (d, a), (a, b), (g, h), (h, f), (g, i)
- (C) (d, f), (f, c), (d, a), (a, b), (c, e), (f, h), (g, h), (g, i)
- (D) (h, g), (g, i), (h, f), (f, c), (f, d), (d, a), (a, b), (c, e)

**Q** Consider the undirected graph below:



Using Prim's algorithm to construct a minimum spanning tree starting with node A, which one of the following sequences of edges represents a possible order in which the edges would be added to construct the minimum spanning tree? **(Gate-2004) (2 Marks)**

- (A) (E, G), (C, F), (F, G), (A, D), (A, B), (A, C)
- (B) (A, D), (A, B), (A, C), (C, F), (G, E), (F, G)
- (C) (A, B), (A, D), (D, F), (F, G), (G, E), (F, C)
- (D) (A, D), (A, B), (D, F), (F, C), (F, G), (G, E)

**Q** What is the weight of a minimum spanning tree of the following graph?

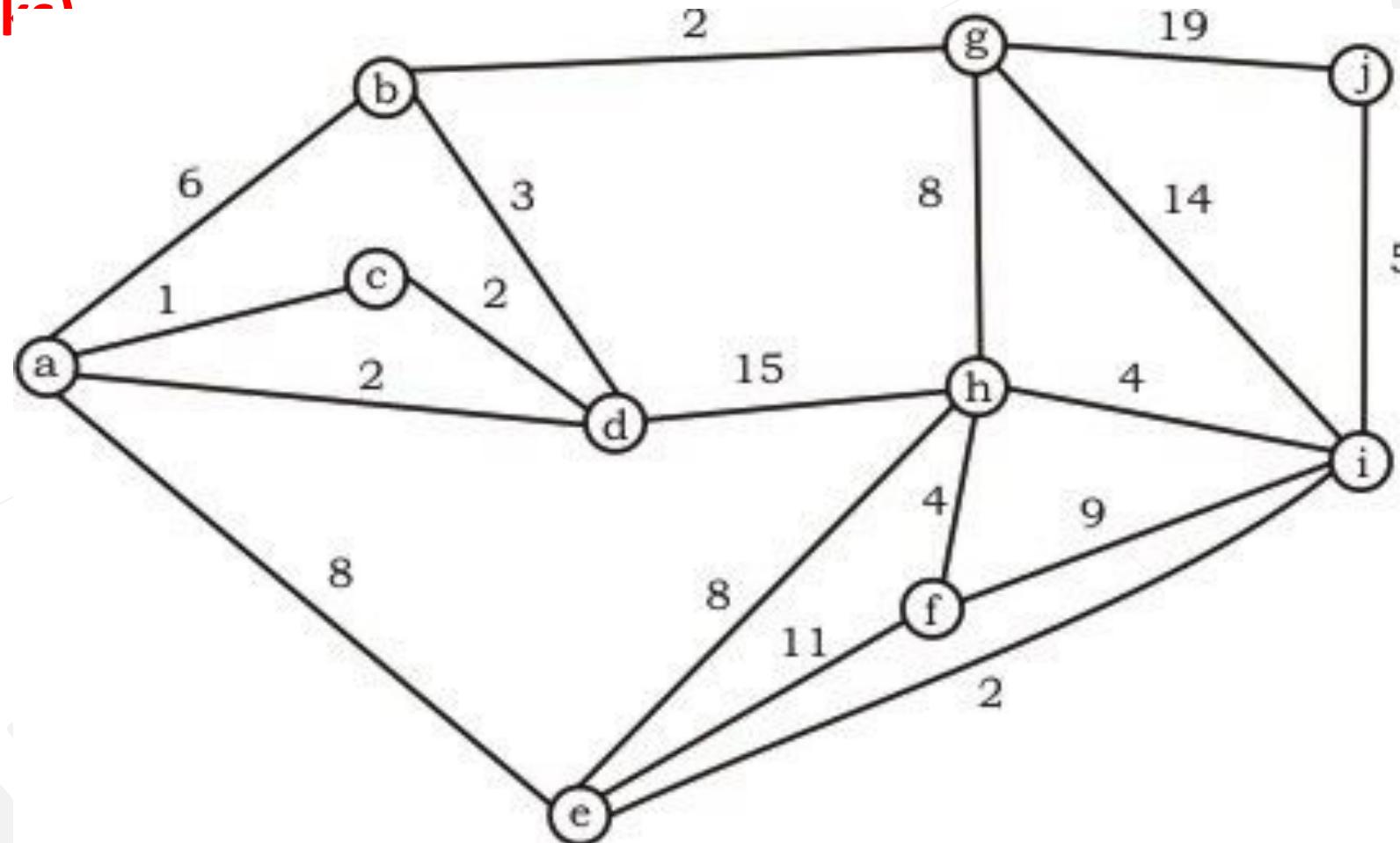
**(Gate-2003) (2 Marks)**

**(A) 29**

**(B) 31**

**(C) 38**

**(D) 41**



**Q** Let  $G(V, E)$  be a directed graph, where  $V = \{1, 2, 3, 4, 5\}$  is the set of vertices and  $E$  is the set of directed edges, as defined by the following adjacency matrix  $A$ .

$$A[i][j] = \begin{cases} 1, & 1 \leq j \leq i \leq 5 \\ 0, & \text{otherwise} \end{cases}$$

$A[i][j] = 1$  indicates a directed edge from node  $i$  to node  $j$ . A directed spanning tree of  $G$ , rooted at  $r \in V$ , is defined as a sub graph  $T$  of  $G$  such that the undirected version of  $T$  is a tree, and  $T$  contains a directed path from  $r$  to every other vertex in  $V$ . The number of such directed spanning trees rooted at vertex 5 is \_\_\_\_\_. **(GATE 2022) (2 MARKS)**

**Q** Consider a simple undirected weighted graph  $G$ , all of whose edge weights are distinct. Which of the following statements about the minimum spanning trees of  $G$  is/are TRUE? **(GATE 2022) (2 MARKS)**

- (A)** The edge with the second smallest weight is always part of any minimum spanning tree of  $G$ .
- (B)** One or both of the edges with the third smallest and the fourth smallest weights are part of any minimum spanning tree of  $G$ .
- (C)** Suppose  $S \subseteq V$  be such that  $S \neq \Phi$  and  $S \neq V$ . Consider the edge with the minimum weight such that one of its vertices is in  $S$  and the other in  $V \setminus S$ . Such an edge will always be part of any minimum spanning tree of  $G$ .
- (D)**  $G$  can have multiple minimum spanning trees.

**Q** Let G be any connected, weighted, undirected graph:

- I. G has a unique minimum spanning tree if no two edges of G have the same weight.
- II. G has a unique minimum spanning tree if, for every cut G, there is a unique minimum weight edge crossing the cut.

Which of the above two statements is/are TRUE? **(Gate-2019) (2 Marks)**

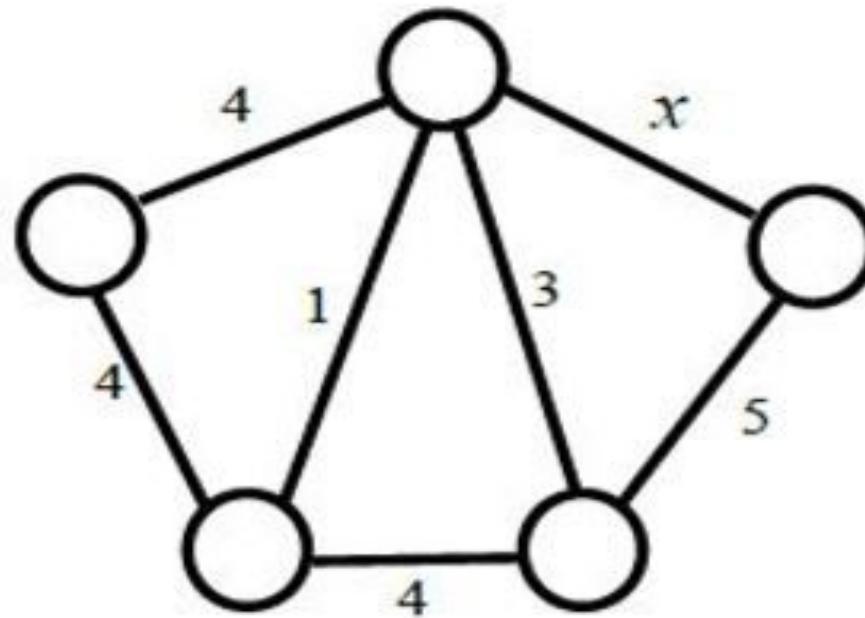
a) Neither I nor II

b) I only

c) II only

d) Both I and II

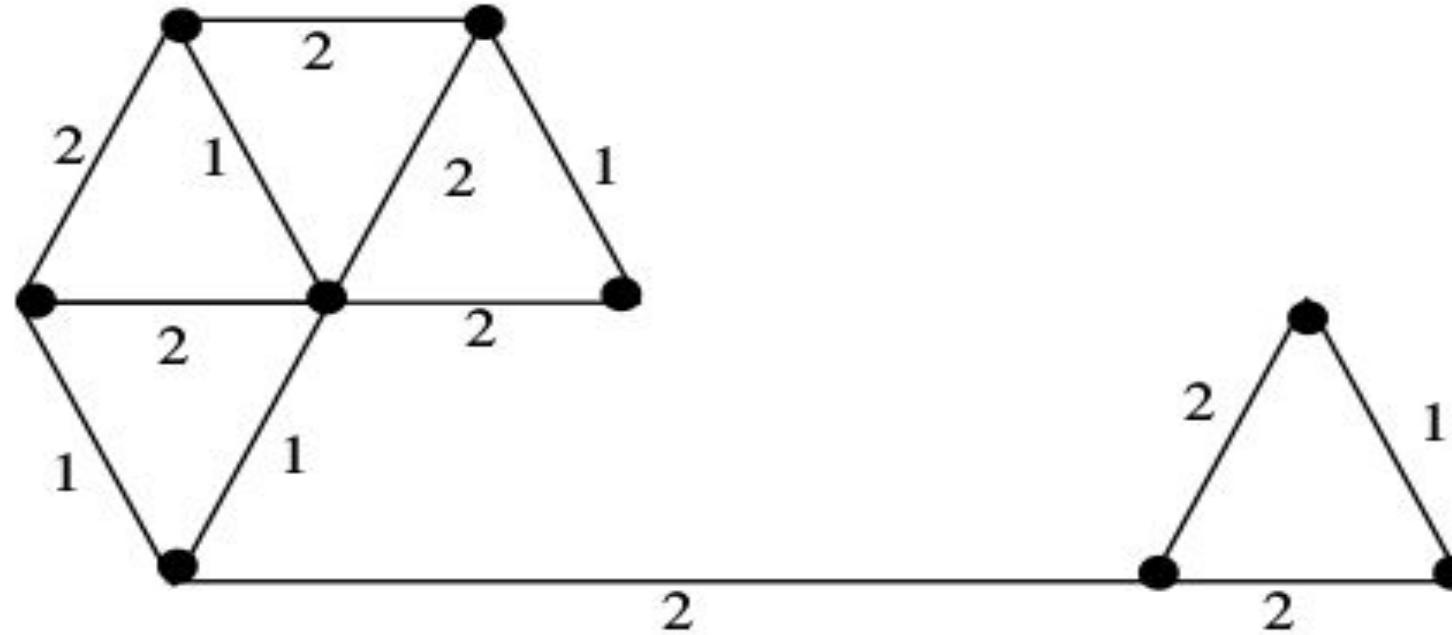
**Q** Consider the following undirected graph. Choose a value for  $x$  that will maximize the number of minimum weight spanning trees (MWSTs) of  $G$ . The number of MWSTs of  $G$  for this value of  $x$  is \_\_\_\_\_. (Gate-2018) (2 Marks)



**Q** Let  $G$  be an undirected connected graph with distinct edge weight. Let  $e_{\max}$  be the edge with maximum weight and  $e_{\min}$  the edge with minimum weight. Which of the following statements is false? **(Gate-2000) (2 Marks) (NET-NOV-2017)**

- (A) Every minimum spanning tree of  $G$  must contain  $e_{\min}$
- (B) If  $e_{\max}$  is in a minimum spanning tree, then its removal must disconnect  $G$
- (C) No minimum spanning tree contains  $e_{\max}$
- (D)  $G$  has a unique minimum spanning tree

**Q** The number of distinct minimum spanning trees for the weighted graph below is \_\_\_\_\_ (GATE-2017) (2 Marks)



**Q** Let  $G = (V, E)$  be any connected undirected edge-weighted graph. The weights of the edges in  $E$  are positive and distinct. Consider the following statements: **(Gate-2017) (2 Marks)**

- I. Minimum Spanning Tree of  $G$  is always unique.
- II. Shortest path between any two vertices of  $G$  is always unique.

Which of the above statements is/are necessarily true?

- a) I only
- b) II only
- c) both I and II
- d) neither I and II

**Q**  $G = (V, E)$  is an undirected simple graph in which each edge has a distinct weight, and  $e$  is a particular edge of  $G$ . Which of the following statements about the minimum spanning trees (MSTs) of  $G$  is/are TRUE **(Gate-2016) (2 Marks)**

- I. If  $e$  is the lightest edge of some cycle in  $G$ , then every MST of  $G$  includes  $e$
  - II. If  $e$  is the heaviest edge of some cycle in  $G$ , then every MST of  $G$  excludes  $e$
- (A) I only
- (B) II only
- (C) both I and II
- (D) neither I nor II

**Q** Let G be a weighted connected undirected graph with distinct positive edge weights. If every edge weight is increased by the same value, then which of the following statements is/are TRUE? **(Gate-2016) (2 Marks)**

**P:** Minimum spanning tree of G does not change

**Q:** Shortest path between any pair of vertices does not change

**(A)** P only

**(B)** Q only

**(C)** Neither P nor Q

**(D)** Both P and Q

**Q** Let G be a complete undirected graph on 4 vertices, having 6 edges with weights being 1, 2, 3, 4, 5, and 6. The maximum possible weight that a minimum weight spanning tree of G can have is. **(Gate-2016) (2 Marks)**

**Q** Let G be connected undirected graph of 100 vertices and 300 edges. The weight of a minimum spanning tree of G is 500. When the weight of each edge of G is increased by five, the weight of a minimum spanning tree becomes \_\_\_\_\_. (GATE-2015) (2 Marks)

**Q** Let  $G$  be a weighted graph with edge weights greater than one and  $G'$  be the graph constructed by squaring the weights of edges in  $G$ . Let  $T$  and  $T'$  be the minimum spanning trees of  $G$  and  $G'$ , respectively, with total weights  $t$  and  $t'$ . Which of the following statements is TRUE? **(Gate-2012)** **(2 Marks)**

(A)  $T' = T$  with total weight  $t' = t^2$

(B)  $T' = T$  with total weight  $t' < t^2$

(C)  $T' \neq T$  but total weight  $t' = t^2$

(D) None of the above

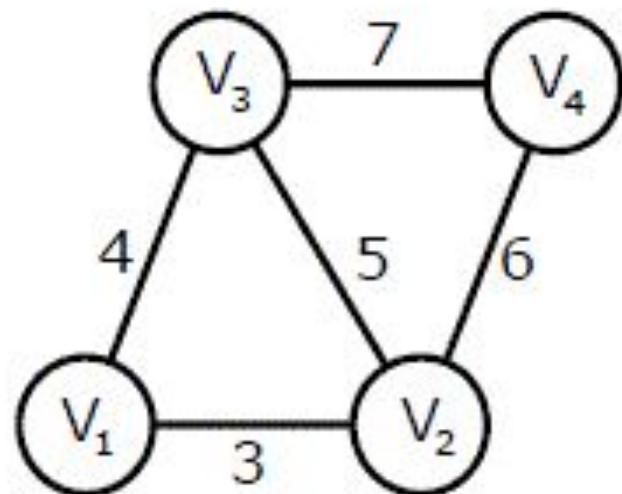
**Q** An undirected graph  $G(V, E)$  contains  $n$  ( $n > 2$ ) nodes named  $v_1, v_2, \dots, v_n$ . Two nodes  $v_i, v_j$  are connected if and only if  $0 < |i - j| \leq 2$ . Each edge  $(v_i, v_j)$  is assigned a weight  $i + j$ . A sample graph with  $n = 4$  is shown below. What will be the cost of the minimum spanning tree (MST) of such a graph with  $n$  nodes? **(GATE - 2011) (2 Marks)**

(A)  $\frac{1}{12}(11n^2 - 5n)$

(B)  $n^2 - n + 1$

(C)  $6n - 11$

(D)  $2n + 1$



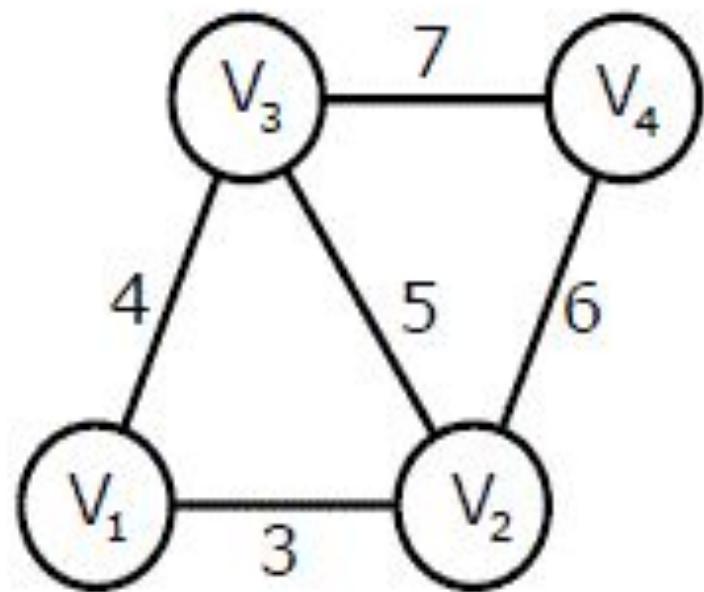
**Q** The length of the path from v<sub>5</sub> to v<sub>6</sub> in the MST of previous question with n = 10 is (GATE - 2011) (2 Marks)

(A) 11

(B) 25

(C) 31

(D) 41



**Q** Consider a complete undirected graph with vertex set  $\{0, 1, 2, 3, 4\}$ . Entry  $W_{ij}$  in the matrix  $W$  below is the weight of the edge  $\{i, j\}$ . What is the minimum possible weight of a spanning tree  $T$  in this graph such that vertex 0 is a leaf node in the tree  $T$ ? (GATE - 2010) (2 Marks)

- (A) 7
- (B) 8
- (C) 9
- (D) 10

$$W = \begin{pmatrix} 0 & 1 & 8 & 1 & 4 \\ 1 & 0 & 12 & 4 & 9 \\ 8 & 12 & 0 & 7 & 3 \\ 1 & 4 & 7 & 0 & 2 \\ 4 & 9 & 3 & 2 & 0 \end{pmatrix}$$

**Q** In the graph given in above question, what is the minimum possible weight of a path P from vertex 1 to vertex 2 in this graph such that P contains at most 3 edges?

**(GATE - 2010) (2 Marks)**

- (A) 7
- (B) 8
- (C) 9
- (D) 10

**Q** Let  $w$  be the minimum weight among all edge weights in an undirected connected graph. Let  $e$  be a specific edge of weight  $w$ . Which of the following is FALSE? **(Gate-2007) (2 Marks)**

- (A) There is a minimum spanning tree containing  $e$ .
- (B) If  $e$  is not in a minimum spanning tree  $T$ , then in the cycle formed by adding  $e$  to  $T$ , all edges have the same weight.
- (C) Every minimum spanning tree has an edge of weight  $w$ .
- (D)  $e$  is present in every minimum spanning tree.

**Q** What is the largest integer  $m$  such that every simple connected graph with  $n$  vertices and  $n$  edges contains at least  $m$  different spanning trees?

**(Gate-2007) (2 Marks)**

**(A) 1**

**(B) 2**

**(C) 3**

**(D)  $n$**

**Q** Consider a weighted complete graph  $G$  on the vertex set  $\{v_1, v_2, \dots, v_n\}$  such that the weight of the edge  $(v_i, v_j)$  is  $2|i-j|$ . The weight of a minimum spanning tree of  $G$  is: **(GATE - 2006)**

- (A)  $n - 1$
- (B)  $2n - 2$
- (C)  ${}^n C_2$
- (D) 2

**Q** An undirected graph G has n nodes. Its adjacency matrix is given by an  $n \times n$  square matrix whose (i) diagonal elements are 0's and (ii) non-diagonal elements are 1's. Which one of the following is TRUE? **(Gate-2005) (2 Marks)**

- (A) Graph G has no minimum spanning tree (MST)
- (B) Graph G has a unique MST of cost  $n-1$
- (C) Graph G has multiple distinct MSTs, each of cost  $n-1$
- (D) Graph G has multiple spanning trees of different costs

**Q** Let  $s$  and  $t$  be two vertices in a undirected graph  $G(V, E)$  having distinct positive edge weights. Let  $[X, Y]$  be a partition of  $V$  such that  $s \in X$  and  $t \in Y$ . Consider the edge  $e$  having the minimum weight amongst all those edges that have one vertex in  $X$  and one vertex in  $Y$ , The edge  $e$  must definitely belong to: **(Gate-2005) (2 Marks)**

(A) the minimum weighted spanning tree of  $G$

(B) the weighted shortest path from  $s$  to  $t$

(C) each path from  $s$  to  $t$

(D) the weighted longest path from  $s$  to  $t$

**Q** Let G be a weighted undirected graph and e be an edge with maximum weight in G. Suppose there is a minimum weight spanning tree in G containing the edge e. Which of the following statements is always TRUE? **(Gate-2005)(2 Marks)**

- (A) There exists a cut set in G having all edges of maximum weight.
- (B) There exists a cycle in G having all edges of maximum weight
- (C) Edge e cannot be contained in a cycle.
- (D) All edges in G have the same weight

**Q** Consider the following undirected graph with edge weights as shown:

The number of minimum-weight spanning trees of the graph is \_\_\_\_\_.

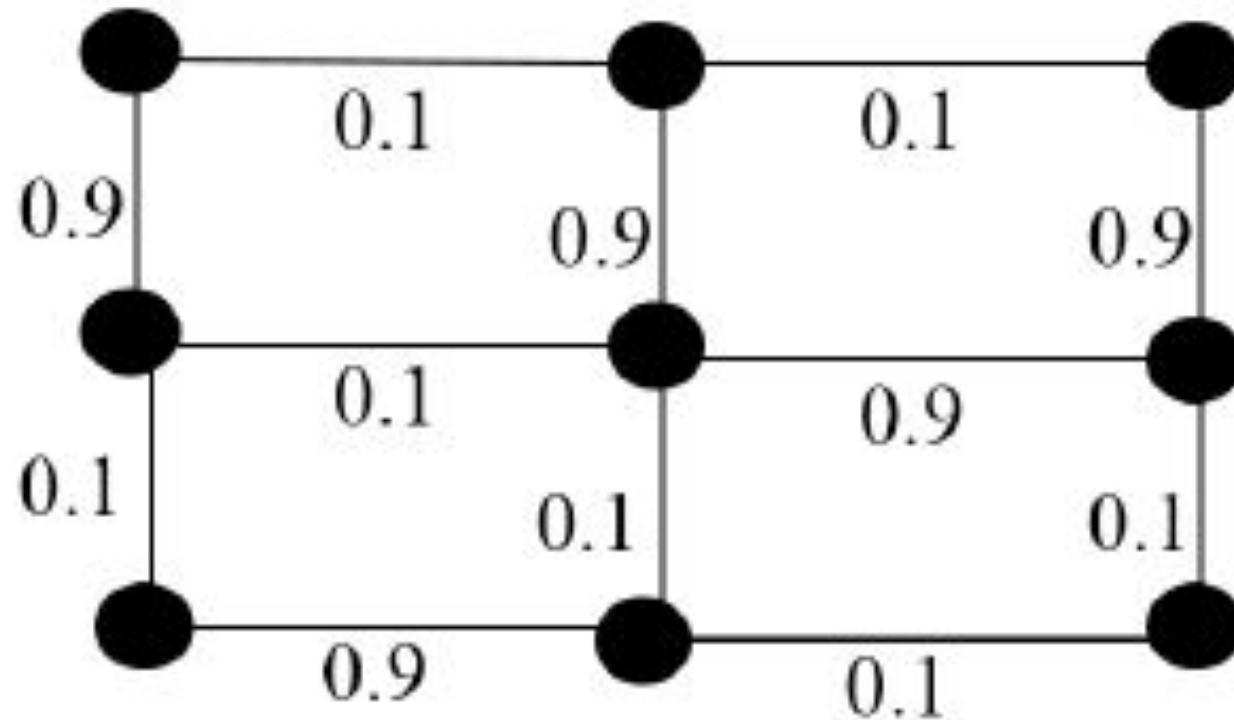
**(GATE 2021)**

**(a) 3**

**(b) 4**

**(c) 5**

**(d) 2**



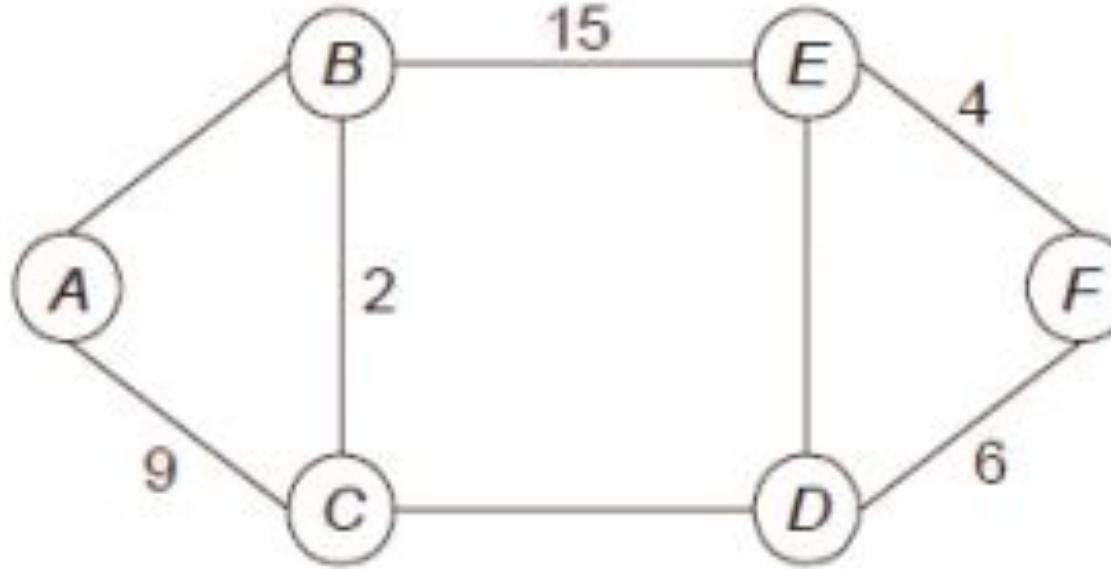
**Q** Let  $G$  be a connected undirected weighted graph. Consider the following two statements.

- $S_1$ : There exists a minimum weight edge in  $G$  which is present in every minimum spanning tree of  $G$ .
- $S_2$ : If every edge in  $G$  has distinct weight, then  $G$  has a unique minimum spanning tree.

Which one of the following options is correct? **(GATE 2021) (1 MARKS)**

- (A) Both  $S_1$  and  $S_2$  are true
- (B)  $S_1$  is true and  $S_2$  is false
- (C)  $S_1$  is false and  $S_2$  is true
- (D) Both  $S_1$  and  $S_2$  are false

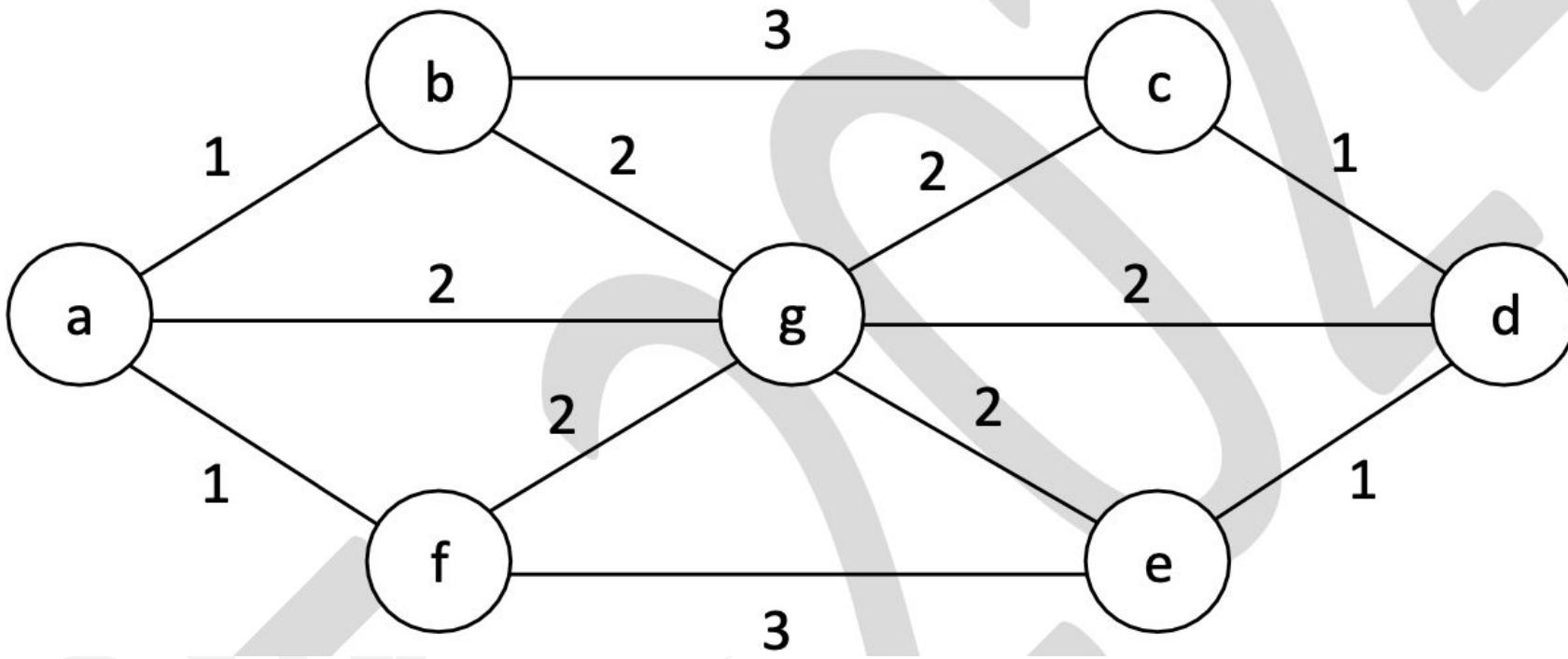
**Q** The graph shown below has 8 edges with distinct integer edge weights. The minimum spanning tree (MST) is of weight 36 and contains the edges:  $\{(A, C), (B, C), (B, E), (E, F), (D, F)\}$ . The edge weights of only those edges which are in the MST are given in the figure shown below. The minimum possible sum of weights of all 8 edges of this graph is \_\_\_\_\_ . (Gate - 2015) (2 Marks)



**Q.** Let  $G$  be an undirected connected graph in which every edge has a positive integer weight. Suppose that every spanning tree in  $G$  has even weight. Which of the following statements is/are TRUE for every such graph  $G$  ? **(Gate 2024 CS)(2 Marks) (MSQ)**

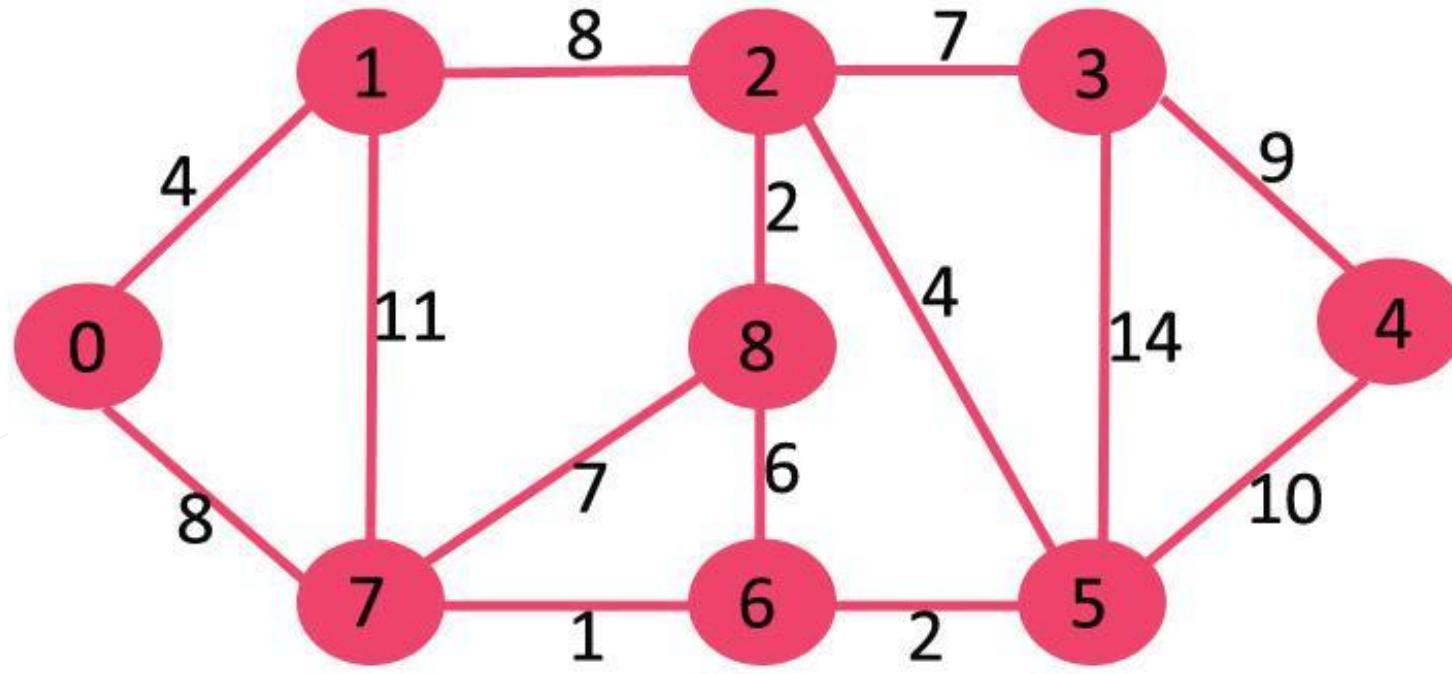
- (a)** All edges in  $G$  have even weight
- (b)** All edges in  $G$  have even weight OR all edges in  $G$  have odd weight
- (c)** In each cycle  $C$  in  $G$ , all edges in  $C$  have even weight
- (d)** In each cycle  $C$  in  $G$ , either all edges in  $C$  have even weight OR all edges in  $C$  have odd weight

**Q. The number of distinct minimum-weight spanning trees of the following graph is \_\_\_\_\_**  
**(Gate 2024 CS)(2 Marks)(NAT)**



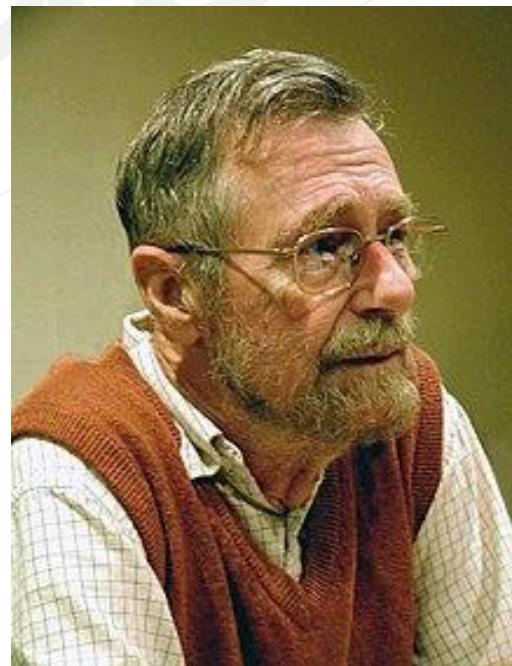
## Single Source Shortest Path

- In graph theory, the **shortest path problem** is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.



prestigious Turing Award in 1972 for his work.

- One of his most famous inventions is **Dijkstra's Algorithm**, which he developed almost by chance while sitting at a café in Amsterdam. The algorithm, created in just 20 minutes, solves the **shortest path problem**, a fundamental concept in graph theory. This algorithm, published in 1959, became widely influential and is still considered a cornerstone in computer science, particularly in network routing protocols such as IS-IS and OSPF.
- Dijkstra's algorithm is appreciated for its simplicity, which stems from his approach to designing it without the aid of pencil and paper. This forced him to avoid unnecessary complexities, making the algorithm highly efficient.



Dijkstra algorithm ( $G, W, S$ )

```
{  
    initialize-Single-source ( $G, S$ )  
     $S \leftarrow \emptyset$   
     $Q \leftarrow V[G]$   
    While ( $Q \neq \emptyset$ )  
    {  
         $u \leftarrow \text{extract-min} (Q)$   
         $S \leftarrow S \cup \{u\}$   
        for each vertex  $v \in \text{adj}(u)$   
        {  
            relax ( $u, v, w$ )  
        }  
    }  
}
```

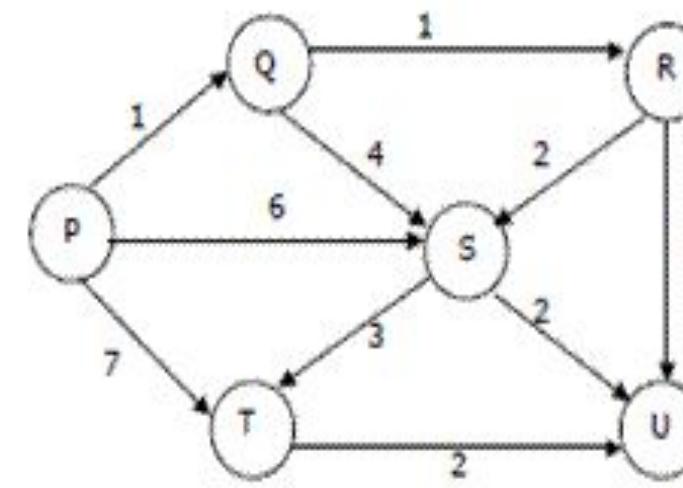
```
Relax ( $u, v, w$ )  
{
```

```
    if( $d[v] > d[u] + w(u, v)$ )  
    {  
         $d[v] \leftarrow d[u] + w(u, v)$   
         $\pi[v] \leftarrow u$   
    }
```

```
Initialize_Single_Source ( $G, S$ )  
{
```

```
    for each vertex  $v \in V[G]$   
    {  
         $d[v] \leftarrow \infty$   
         $\pi[v] \leftarrow \text{NIL}$   
    }
```

```
     $d[S] \leftarrow 0$   
}
```



P	Q	R	S	T	U

How fast is Dijkstra's algorithm? It maintains the min-priority queue  $Q$  by calling three priority-queue operations: INSERT (implicit in line 3), EXTRACT-MIN (line 5), and DECREASE-KEY (implicit in RELAX, which is called in line 8). The algorithm calls both INSERT and EXTRACT-MIN once per vertex. Because each vertex  $u \in V$  is added to set  $S$  exactly once, each edge in the adjacency list  $Adj[u]$  is examined in the **for** loop of lines 7–8 exactly once during the course of the algorithm. Since the total number of edges in all the adjacency lists is  $|E|$ , this **for** loop iterates a total of  $|E|$  times, and thus the algorithm calls DECREASE-KEY at most  $|E|$  times overall. (Observe once again that we are using aggregate analysis.)

The running time of Dijkstra's algorithm depends on how we implement the min-priority queue. Consider first the case in which we maintain the min-priority queue by taking advantage of the vertices being numbered 1 to  $|V|$ . We simply store  $v.d$  in the  $v$ th entry of an array. Each INSERT and DECREASE-KEY operation takes  $O(1)$  time, and each EXTRACT-MIN operation takes  $O(V)$  time (since we have to search through the entire array), for a total time of  $O(V^2 + E) = O(V^2)$ .

If the graph is sufficiently sparse—in particular,  $E = o(V^2 / \lg V)$ —we can improve the algorithm by implementing the min-priority queue with a binary min-heap. (As discussed in Section 6.5, the implementation should make sure that vertices and corresponding heap elements maintain handles to each other.) Each EXTRACT-MIN operation then takes time  $O(\lg V)$ . As before, there are  $|V|$  such operations. The time to build the binary min-heap is  $O(V)$ . Each DECREASE-KEY operation takes time  $O(\lg V)$ , and there are still at most  $|E|$  such operations. The total running time is therefore  $O((V + E) \lg V)$ , which is  $O(E \lg V)$  if all vertices are reachable from the source. This running time improves upon the straightforward  $O(V^2)$ -time implementation if  $E = o(V^2 / \lg V)$ .

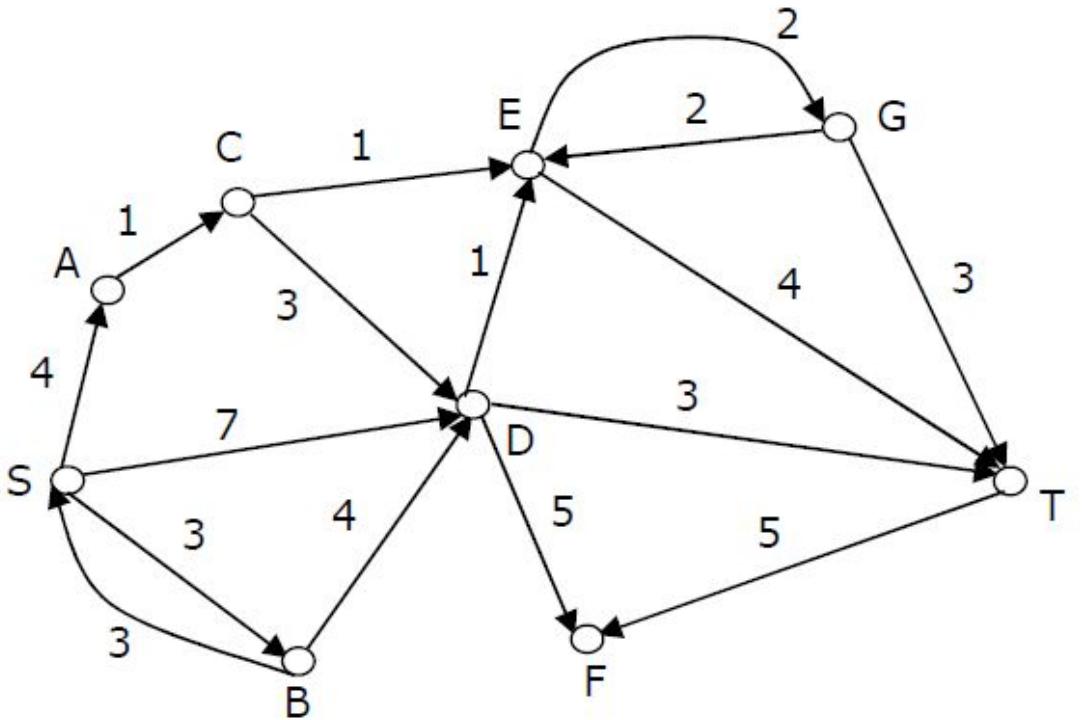
We can in fact achieve a running time of  $O(V \lg V + E)$  by implementing the min-priority queue with a Fibonacci heap (see Chapter 19). The amortized cost of each of the  $|V|$  EXTRACT-MIN operations is  $O(\lg V)$ , and each DECREASE-KEY call, of which there are at most  $|E|$ , takes only  $O(1)$  amortized time. Historically, the development of Fibonacci heaps was motivated by the observation that Dijkstra's algorithm typically makes many more DECREASE-KEY calls than EXTRACT-MIN calls, so that any method of reducing the amortized time of each DECREASE-KEY operation to  $o(\lg V)$  without increasing the amortized time of EXTRACT-MIN would yield an asymptotically faster implementation than with binary heaps.

## Dijkstra's algorithm (or Dijkstra's Shortest Path First algorithm, SPF algorithm)

- Guarantee to find optimal solution in a connected graph with positive weights.
- Can fail on graph with negative weights.

**Q** Consider the directed graph shown in the figure below. There are multiple shortest paths between vertices S and T. Which one will be reported by Dijkstra's shortest path algorithm? Assume that, in any iteration, the shortest path to a **(Gate-2012) (2 Marks)**

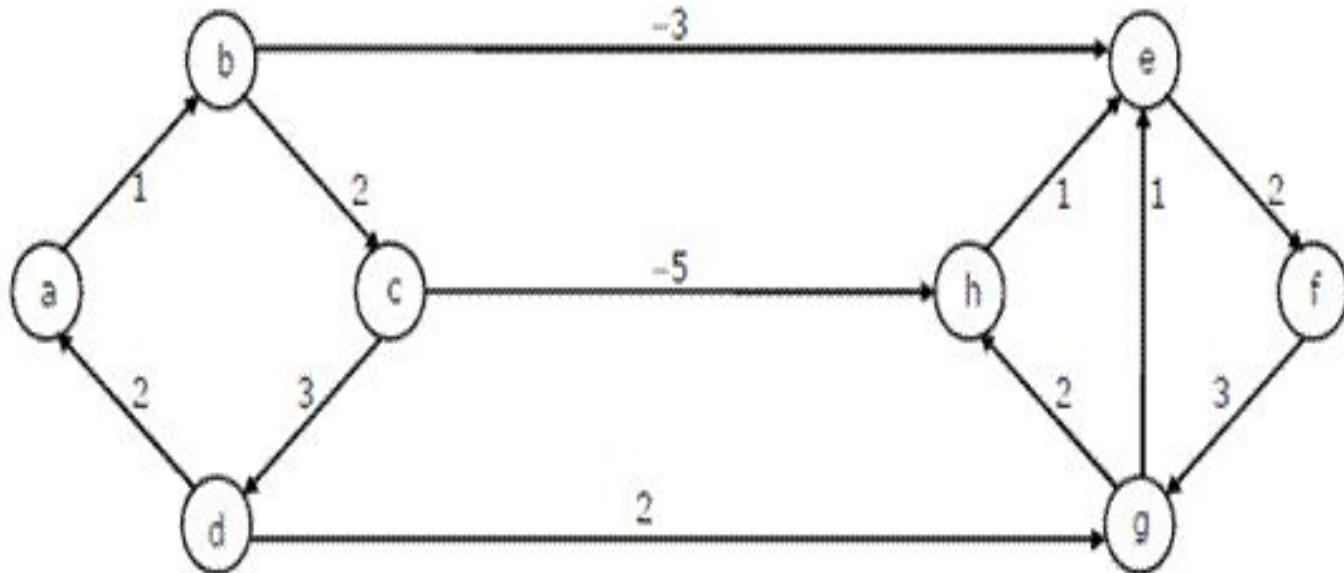
- a) SDT      b) SBDT      c) SACDT      d) SACET



A horizontal sequence of 10 boxes, each labeled with a letter from S to T. The boxes are arranged in two rows: the top row contains S, A, B, C, D, E, F, G, and T; the bottom row contains H, I, J, A, C, E, F, G, and T. The boxes in the top row are shaded light blue, while the boxes in the bottom row are white.

**Q** Dijkstra's single source shortest path algorithm when run from vertex a in the below graph, computes the correct shortest path distance to **(Gate-2008) (2 Marks)**

- (A) only vertex a
- (B) only vertices a, e, f, g, h
- (C) only vertices a, b, c, d
- (D) all the vertices



A	B	C	D	E	F	G	H

**Q** To implement Dijkstra's shortest path algorithm on unweighted graphs so that it runs in linear time, the data structure to be used is: **(Gate-2006) (1 Marks)**

**(A)** Queue

**(B)** Stack

**(C)** Heap

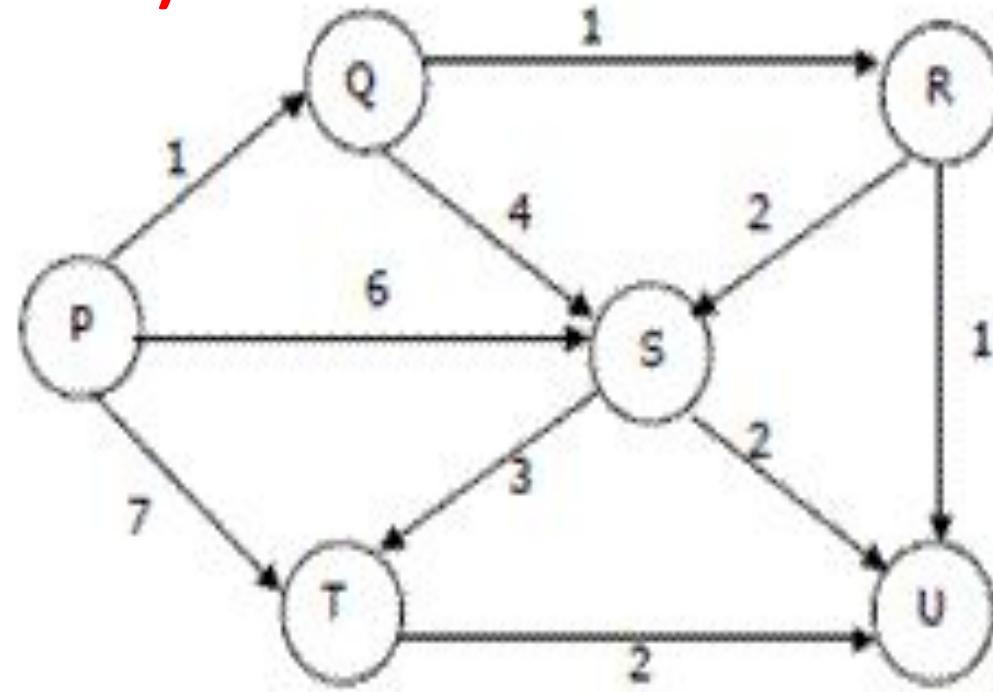
**(D)** B-Tree

**Q** Let  $G(V, E)$  an undirected graph with positive edge weights. Dijkstra's single-source shortest path algorithm can be implemented using the binary heap data structure with time complexity: **(Gate-2005) (2 Marks)**

- (A)**  $O(|V|^2)$
- (B)**  $O(|E| + |V| \log |V|)$
- (C)**  $O(|V| \log |V|)$
- (D)**  $O((|E| + |V|) \log |V|)$

**Q** Suppose we run Dijkstra's single source shortest-path algorithm on the following edge weighted directed graph with vertex P as the source. In what order do the nodes get included into the set of vertices for which the shortest path distances are finalized? **(GATE - 2004) (2 Marks)**

- (A) P, Q, R, S, T, U
- (B) P, Q, R, U, S, T
- (C) P, Q, R, U, T, S
- (D) P, Q, T, R, U, S



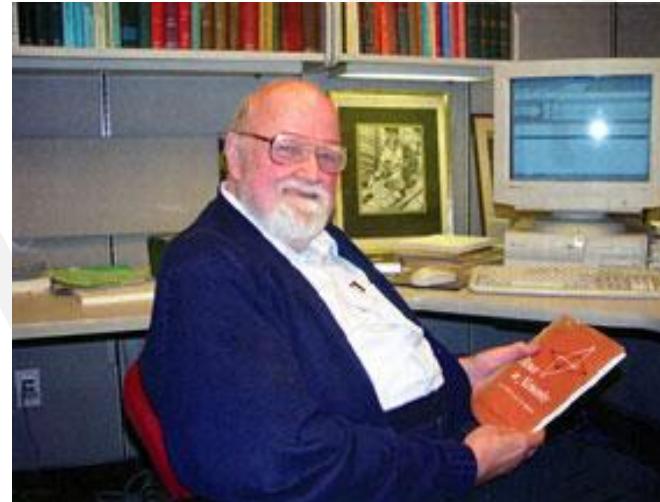
P	Q	R	S	T	U

## Bellman–Ford Algorithm

- The **Bellman–Ford algorithm** is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph.
- It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers.
- The algorithm was first proposed by Alfonso Shimbel (1955), but is instead named after Richard Bellman and Lester Ford Jr., who published it in 1958 and 1956, respectively.



Richard E. Bellman

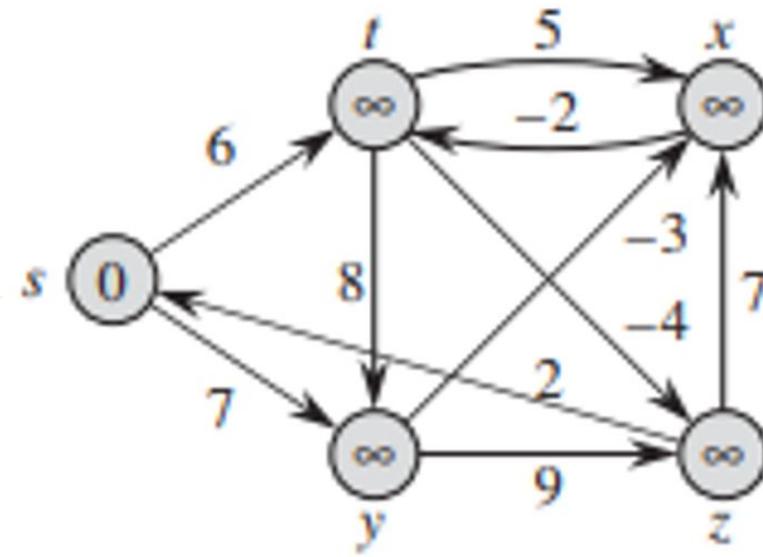


L. R. Ford Jr

```

Bellman_ford (G, W, S)
{
    initialize-Single-Source (G, S)
    for i = 1 to |V(G)| - 1
    {
        for each edge (u, v) ∈ E(G)
        {
            Relax(u, v, w)
        }
    }
    for each edge (u, v) ∈ E(G)
    {
        if(d[v] > d[u] + w (u, v))
        {
            Return false
        }
    }
}
Initialize_Single_Source (G, S)
{
    for each vertex v ∈ V[G]
    {
        d[v] = ∞
        π[v] = NIL
    }
    d[S] = 0
}
Relax (u, v, w)
{
    if(d[v] > d[u] + w (u, v))
    {
        d[v] = d[u] + w (u, v)
        π[v] = u
    }
}

```



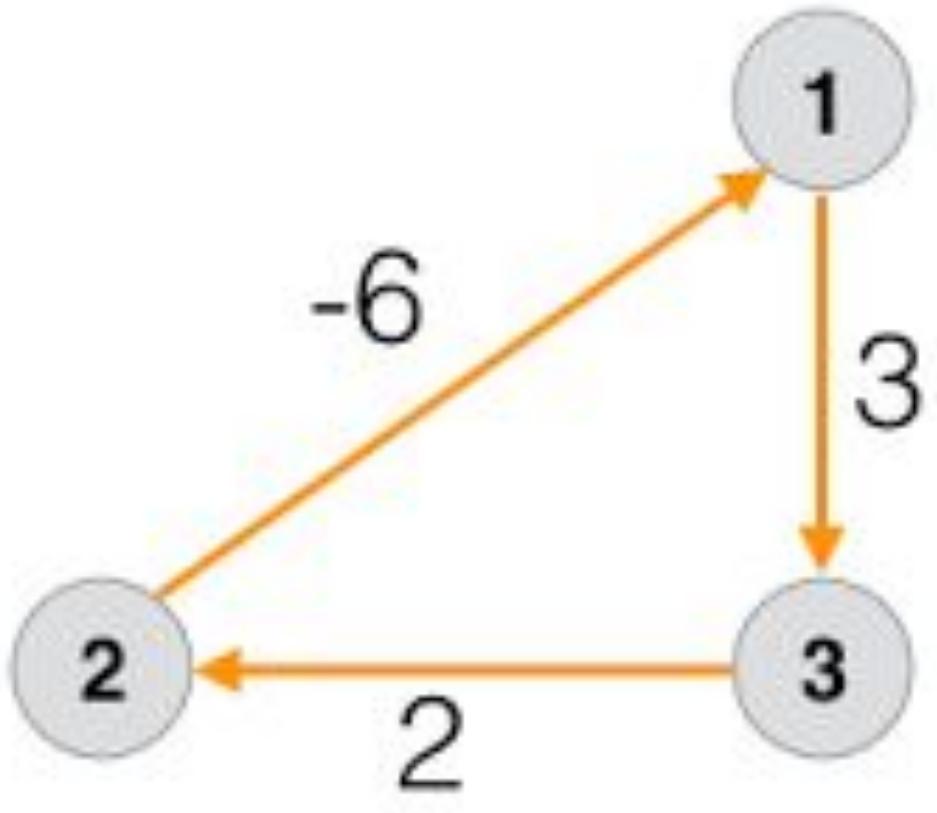
S	T	X	Y	Z

BELLMAN-FORD( $G, w, s$ )

```
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  for  $i = 1$  to  $|G.V| - 1$ 
3    for each edge  $(u, v) \in G.E$ 
4      RELAX( $u, v, w$ )
5  for each edge  $(u, v) \in G.E$ 
6    if  $v.d > u.d + w(u, v)$ 
7      return FALSE
8  return TRUE
```

The Bellman-Ford algorithm runs in time  $O(VE)$ , since the initialization in line 1 takes  $\Theta(V)$  time, each of the  $|V| - 1$  passes over the edges in lines 2–4 takes  $\Theta(E)$  time, and the **for** loop of lines 5–7 takes  $O(E)$  time.

# Bellman–Ford algorithm with negative weight cycle



**Q** Consider the following table

Algorithms	Design Paradigms
(P) Kruskal	(i) Divide and Conquer
(Q) Quicksort	(ii) Greedy
(R) Floyd-Warshall	(iii) Dynamic Programming

Match the algorithm to design paradigms they are based on: **(Gate-2017) (2 Marks)**

a) P-(ii), Q-(iii), R-(i)

b) P-(iii), Q-(i), R-(ii)

c) P-(ii), Q-(i), R-(iii)

d) P-(i), Q-(ii), R-(iii)

## Q Match the following (Gate-2015) (2 Marks)

List-I	List-II
A. Prim's algorithm for minimum spanning tree	1. Backtracking
B. Floyd-Warshall algorithm for all pairs shortest paths	2. Greed method
C. Merge sort	3. Dynamic programming
D. Hamiltonian circuit	4. Divide and conquer

	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>
a)	3	2	4	1
b)	1	2	4	3
c)	2	3	4	1
d)	2	1	3	4

**Q** What is the time complexity of Bellman-Ford single-source shortest path algorithm on a complete graph of n vertices?  
**(Gate-2013) (1 Marks)**

- a)  $\Theta(n^3)$
- b)  $\Theta(n^2)$
- c)  $\Theta(n^2 \log n)$
- d)  $\Theta(n^3 \log n)$

**Q** Which of the following statement(s) is/are correct regarding Bellman-Ford shortest path algorithm? (Gate-2009) (1 Marks)

**P:** Always finds a negative weighted cycle, if one exists.

**Q:** Finds whether any negative weighted cycle is reachable from the source.

a) P only

b) Q only

c) Both P and Q

d) Neither P nor Q

**Q** In an unweighted, undirected connected graph, the shortest path from a node S to every other node is computed most efficiently, in terms of time complexity by **(Gate-2007) (2 Marks)**

**(A)** Dijkstra's algorithm starting from S.

**(B)** Warshall's algorithm

**(C)** Performing a DFS starting from S.

**(D)** Performing a BFS starting from S.

**Q.** Let  $G$  be an edge-weighted undirected graph with positive edge weights. Suppose a positive constant  $\alpha$  is added to the weight of every edge.

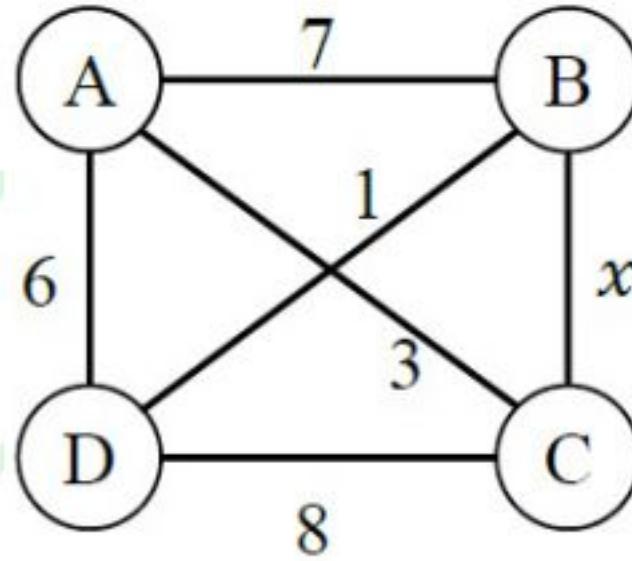
Which ONE of the following statements is TRUE about the minimum spanning trees (MSTs) and shortest paths (SPs) in  $G$  before and after the edge weight update? **(Gate 2025)**

- A) Every MST remains an MST, and every SP remains an SP.
- B) MSTs need not remain MSTs, and every SP remains an SP.
- C) Every MST remains an MST, and SPs need not remain SPs.
- D) MSTs need not remain MSTs, and SPs need not remain SPs.

**Q.** Let  $G(V, E)$  be an undirected and unweighted graph with 100 vertices. Let  $d(u, v)$  denote the number of edges in a shortest path between vertices  $u$  and  $v$  in  $V$ . Let the maximum value of  $d(u, v)$ ,  $u, v \in V$  such that  $u \neq v$ , be 30. Let  $T$  be any breadth-first-search tree of  $G$ . Which ONE of the given options is CORRECT for every such graph  $G$ ? **(Gate 2025)**

- A) The height of  $T$  is exactly 15.
- B) The height of  $T$  is exactly 30.
- C) The height of  $T$  is at least 15.
- D) The height of  $T$  is at least 30.

Q. The maximum value of  $x$  such that the edge between the nodes B and C is included in every minimum spanning tree of the given graph is \_\_\_\_\_ . (answer in integer) (Gate 2025)



**Q** Is the following statement valid?

Given a weighted graph where weights of all edges are unique (no two edge have same weights), there is always a unique shortest path from a source to destination in such a graph.

(A) True

(B) False

# Q Is the following statement valid?

Given a graph where all edges have positive weights, the shortest paths produced by Dijkstra and Bellman Ford algorithm may be different but path weight would always be same.

- (A) True** **(B) False**

**Q** Is the following statement valid about shortest paths?

Given a graph, suppose we have calculated shortest path from a source to all other vertices. If we modify the graph such that weights of all edges becomes double of the original weight, then the shortest path remains same only the total weight of path changes.

**(A)** True

**(B)** False

**Q** In a weighted graph, assume that the shortest path from a source 's' to a destination 't' is correctly calculated using a shortest path algorithm. Is the following statement true?

If we increase weight of every edge by 1, the shortest path always remains same.

(A) Yes

(B) No

**Q** Which of the following standard algorithms is not a Greedy algorithm?

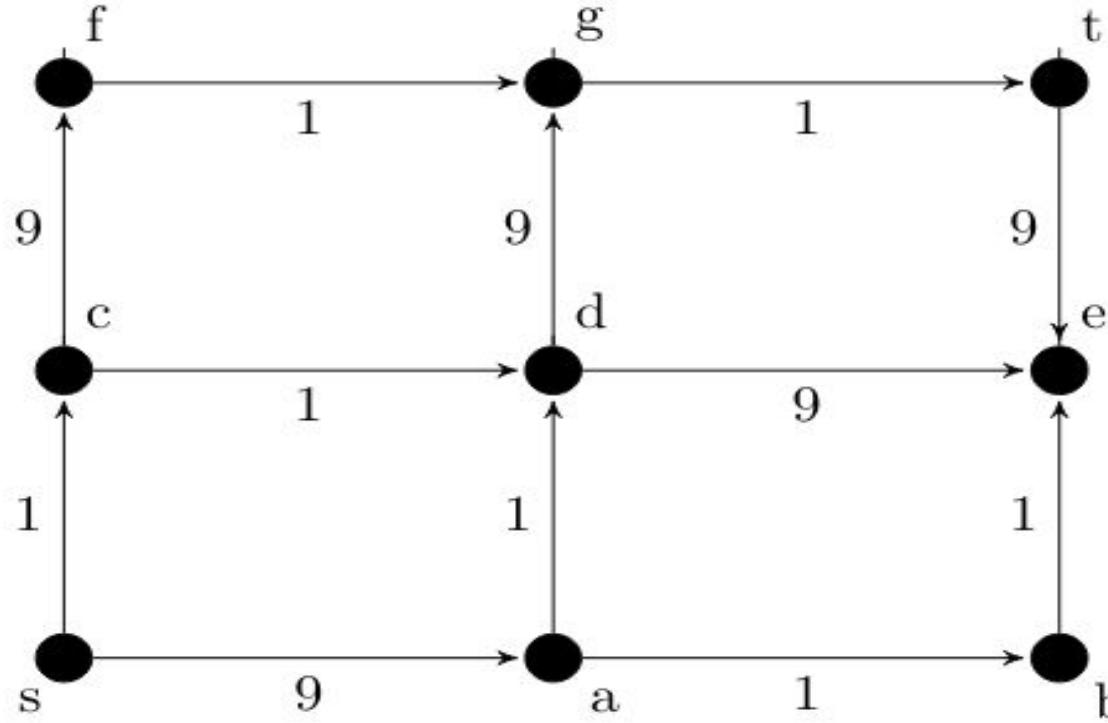
- (A) Dijkstra's shortest path algorithm
- (B) Prim's algorithm
- (C) Kruskal algorithm
- (D) Huffman Coding
- (E) Bellmen Ford Shortest path algorithm

**Q** Which of the following standard algorithms is not Dynamic Programming based.

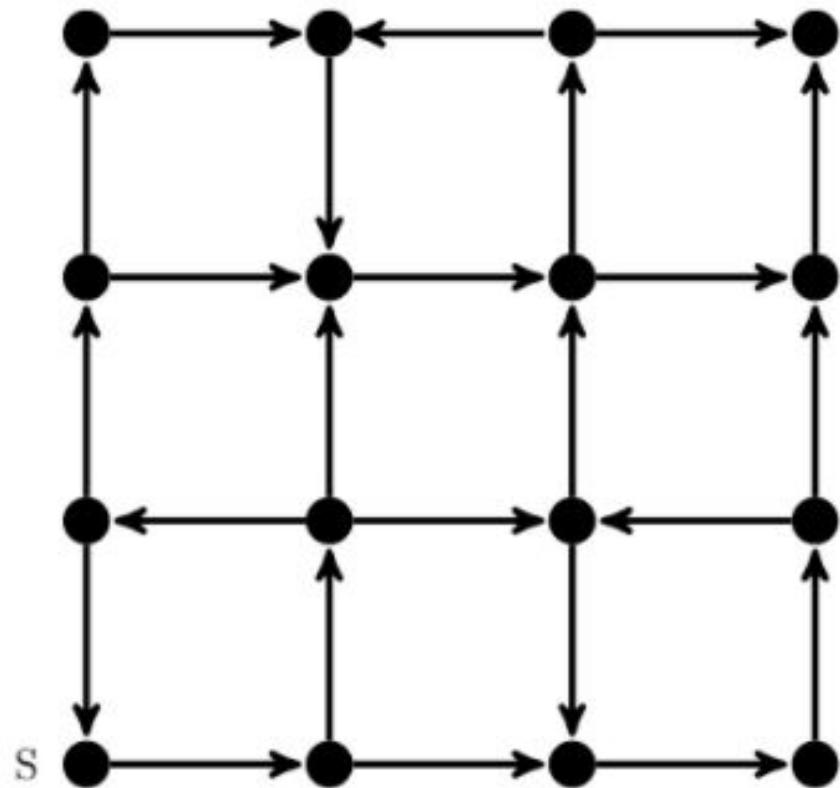
- (A) Bellman–Ford Algorithm for single source shortest path
- (B) Floyd Warshall Algorithm for all pairs shortest paths
- (C) 0-1 Knapsack problem
- (D) Prim's Minimum Spanning Tree

**Q** In a directed acyclic graph with a source vertex  $s$ , the quality-score of a directed path is defined to be the product of the weights of the edges on the path. Further, for a vertex  $v$  other than  $s$ , the quality-score of  $v$  is defined to be the maximum among the quality-scores of all the paths from  $s$  to  $v$ . The quality-score of  $s$  is assumed to be 1. The sum of the quality-scores of all vertices on the graph shown above is \_\_\_\_\_.

**(GATE 2021) (2 MARKS)**



Consider the following directed graph:



Which of the following is/are correct about the graph?

- A. The graph does not have a topological order
- B. A depth-first traversal starting at vertex  $S$  classifies three directed edges as back edges
- C. The graph does not have a strongly connected component
- D. For each pair of vertices  $u$  and  $v$ , there is a directed path from  $u$  to  $v$

**Q.25** Let  $G = (V, E)$  be an undirected unweighted connected graph. The diameter of  $G$  is defined as

$$\text{diam}(G) = \max_{u, v \in V} \{\text{the length of shortest path between } u \text{ and } v\}$$

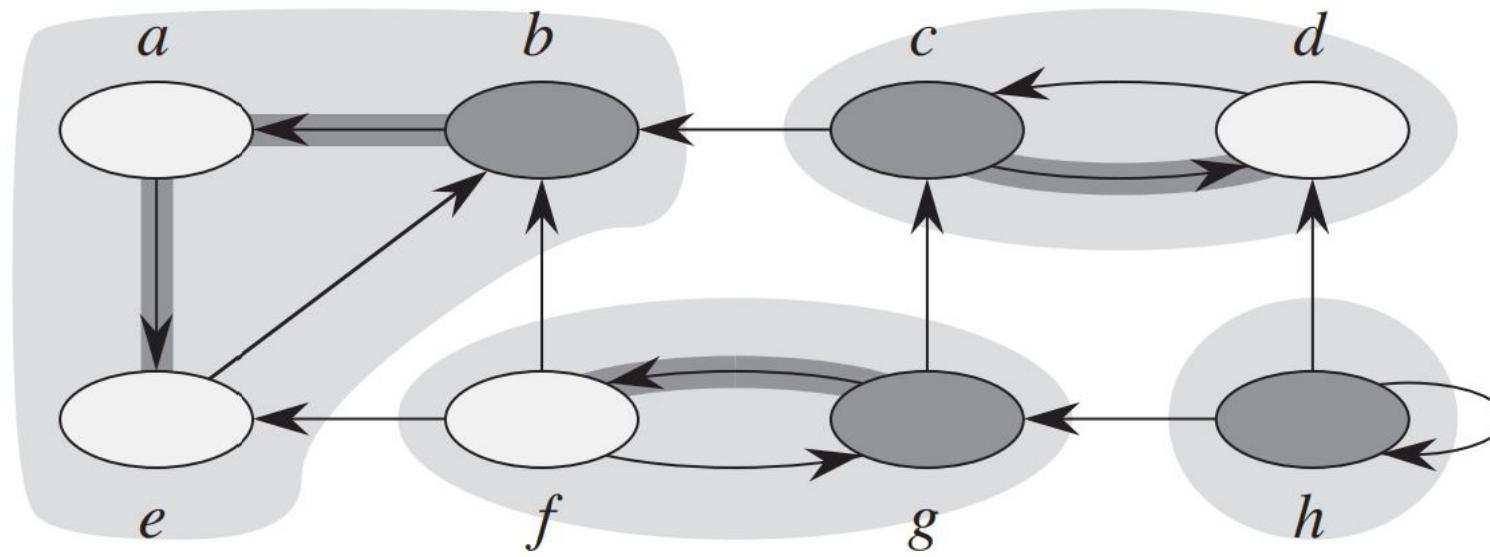
Let  $M$  be the adjacency matrix of  $G$ .

Define graph  $G_2$  on the same set of vertices with adjacency matrix  $N$ , where

$$N_{ij} = \begin{cases} 1 & \text{if } M_{ij} > 0 \text{ or } P_{ij} > 0, \text{ where } P = M^2 \\ 0 & \text{Otherwise} \end{cases}$$

Which one of the following statements is true?

- (a)  $\lceil \text{diam}(G)/2 \rceil < \text{diam}(G_2) < \text{diam}(G)$
- (b)  $\text{diam}(G_2) \leq \lceil \text{diam}(G)/2 \rceil$
- (c)  $\text{diam}(G) < \text{diam}(G_2) \leq 2 \text{ diam}(G)$

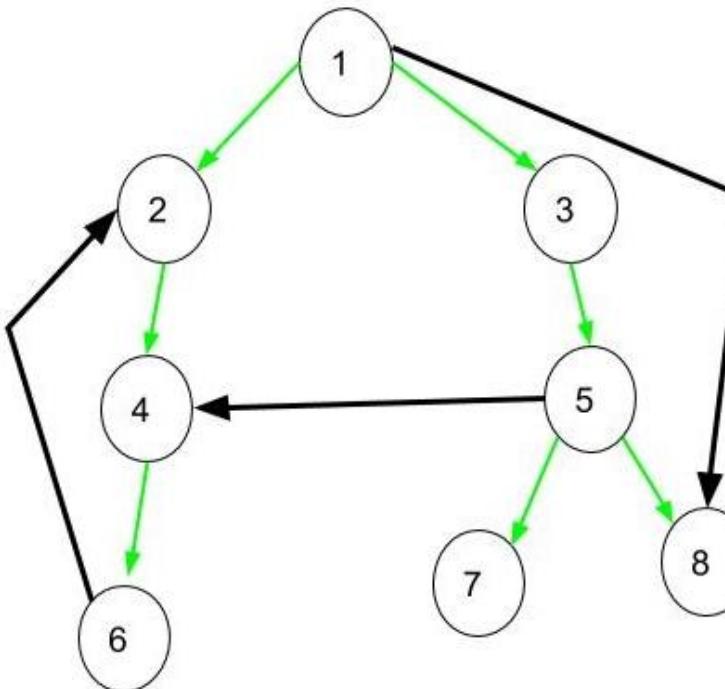


When exploring directed graphs through Depth-First Search (DFS) and Breadth-First Search (BFS), we encounter various types of edges based on the relationship between the vertices and the stage of exploration. Understanding these edges is crucial for analyzing the properties of graphs, such as detecting cycles, understanding graph connectivity, and designing algorithms for graph-related problems.

## Depth-First Search (DFS)

In DFS, you start at a vertex and explore as far as possible along each branch before backtracking. This exploration categorizes the edges into the following types:

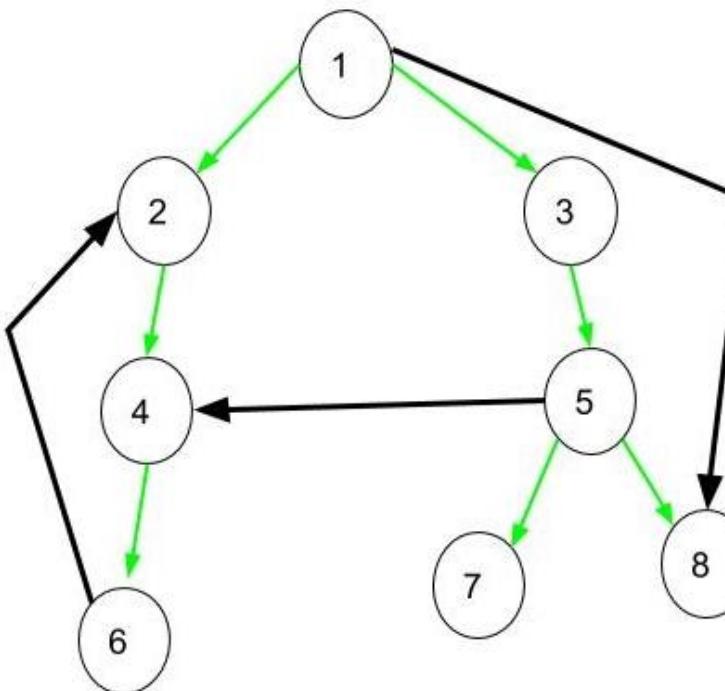
- **Tree Edges:** These are the edges that lead to a vertex that has not yet been visited. In DFS, tree edges form the DFS forest, Green color.
- **Back Edges:** These edges connect a vertex to an ancestor in the DFS tree. The presence of back edges indicates a cycle in the graph, e.g. 6 to 2.
- **Forward Edges:** These are the edges that connect a vertex to a descendant in the DFS tree, but they are not part of the tree itself. E.g. 1 to 8.
- **Cross Edges:** These connect vertices in different branches of the DFS forest. They can connect to vertices that are neither ancestors nor descendants, e.g. 5 to 4.



## Breadth-First Search (BFS)

BFS explores the graph level by level, starting from a given vertex. It explores all neighbors of a vertex before moving on to the neighbors of those neighbors. In the context of BFS in directed graphs, the concept of forward, back, and cross edges is less commonly applied because BFS primarily uses a queue to explore vertices by their distance from the start vertex. However, if we were to categorize edges in the BFS exploration context, most edges explored would fit into the following simplified categories:

- **Tree Edges:** Similar to DFS, these are the edges that lead to a vertex that has not yet been visited, forming the BFS tree.
- **Cross Edges:** In the BFS context, any edge that does not lead to an immediate next level (or is not a tree edge) can be considered a cross edge, potentially connecting vertices within the same level or to a vertex in a previously visited level. BFS does not inherently have "forward" and "back" edges as DFS does, since its exploration is not depth-oriented.



**Q** Let  $G$  be a simple undirected graph. Let  $T_D$  be a depth first search tree of  $G$ . Let  $T_B$  be a breadth first search tree of  $G$ . Consider the following statements.

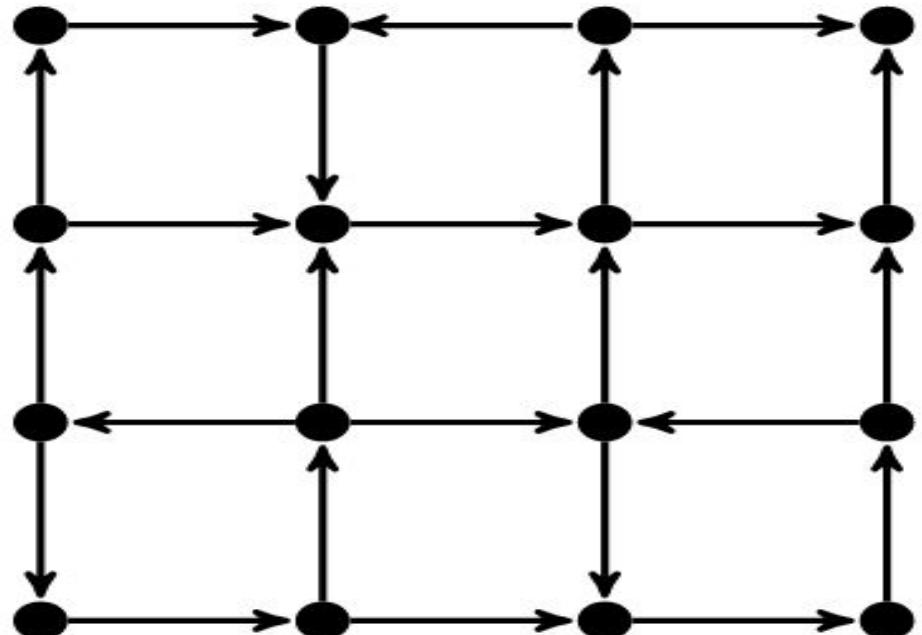
- (I) No edge of  $G$  is a cross edge with respect to  $T_D$ . (A cross edge in  $G$  is between two nodes neither of which is an ancestor of the other in  $T_D$ ).
- (II) For every edge  $(u, v)$  of  $G$ , if  $u$  is at depth  $i$  and  $v$  is at depth  $j$  in  $T_B$ , then  $|i - j| = 1$ .

Which of the statements above must necessarily be true? **(Gate-2018) (2 Marks)**

- a) I only
- b) II only
- c) Both I and II
- d) Neither I nor II

**Q** Consider the following directed graph: Which of the following is/are correct about the graph?  
**(GATE 2021) (2 MARKS)**

- (a)** The graph does not have a topological order
- (b)** A depth-first traversal starting at vertex S classifies three directed edges as back edges.
- (c)** The graph does not have a strongly connected component
- (d)** For each pair of vertices u and v, there is a directed path from u to v

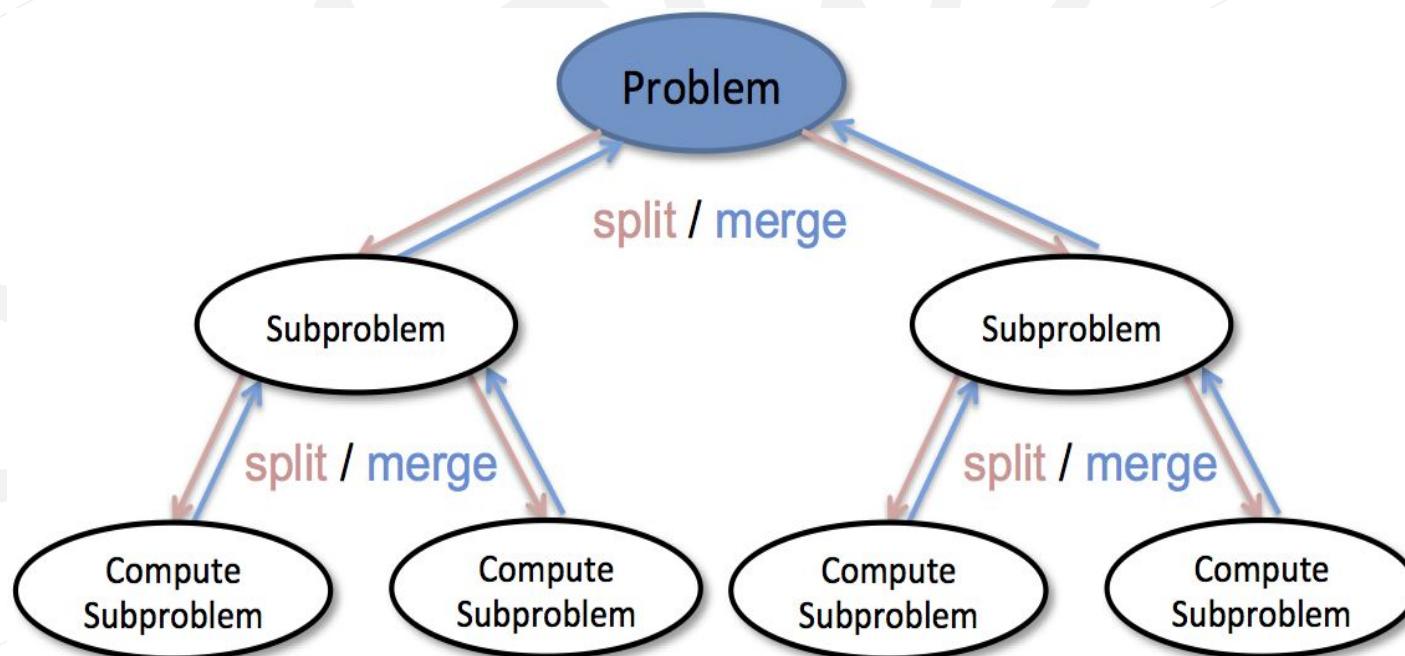


अगर आप अपने Past से कुछ सीख नहीं सकते तो  
जीवनभर छोटे काम ही करते रहेंगे ।

-Dynamic Programming

# Dynamic Programming

- Divide and conquer partition the problem into independent subproblem, solve the subproblems recursively and then combine their solutions to solve the original problems.
- Dynamic programming is like the divide and conquer method, solve problems by combining the solutions to the subproblems. In contrast, dynamic programming is applicable when the subproblems are not independent, i.e. when subproblems share subsubproblems.
- A dynamic-programming algorithm solves every subsubproblems just one and then saves its answer in a table there by avoiding the work of recomputing the answer every time the subproblem is encountered.
- Dynamic programming is typically applied to optimization problems. In such case there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal value (minimum, maximum).

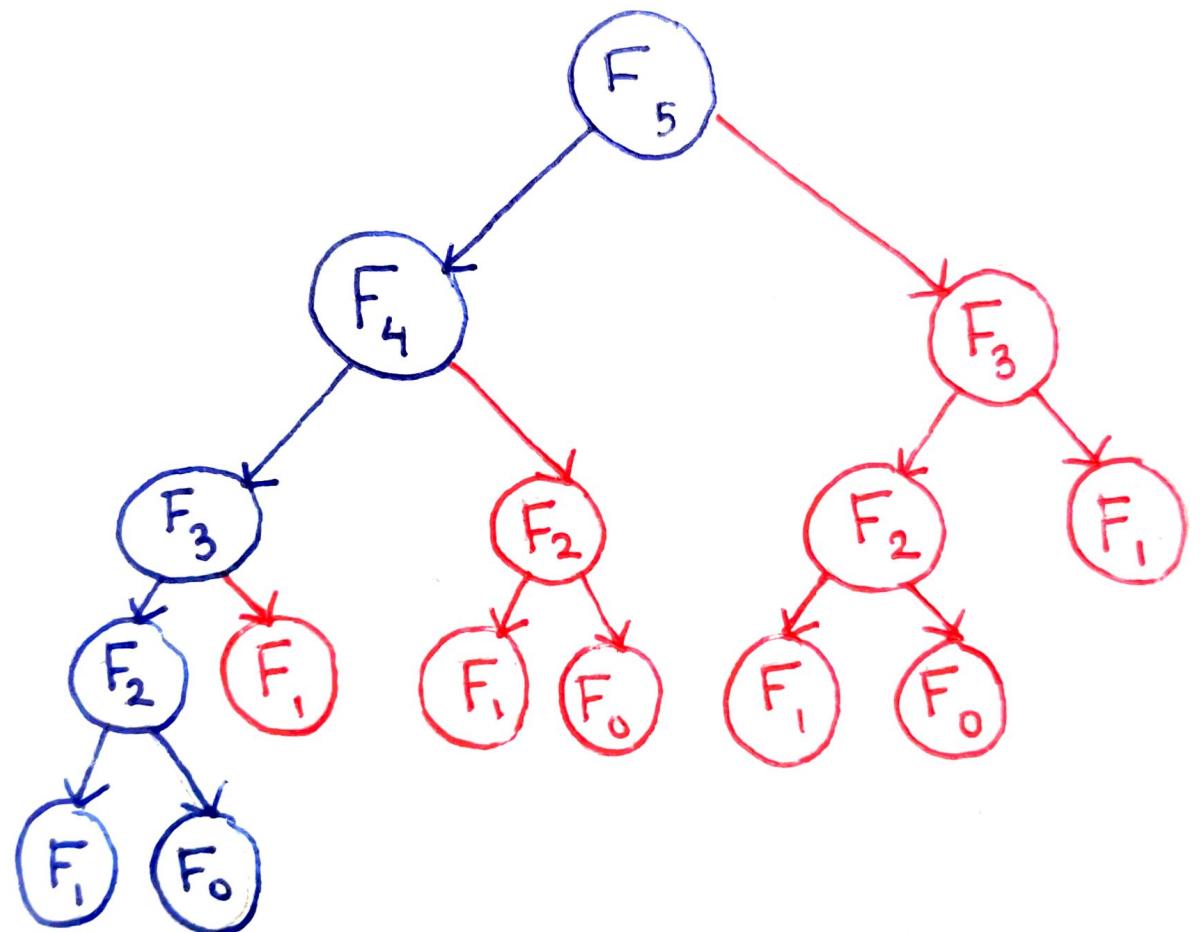


$$F(0) = 0$$

$$F(1) = 1$$

$$F(n) = F(n - 1) + F(n - 2)$$

n	0	1	2	3	4	5
F(n) )						



O(N)

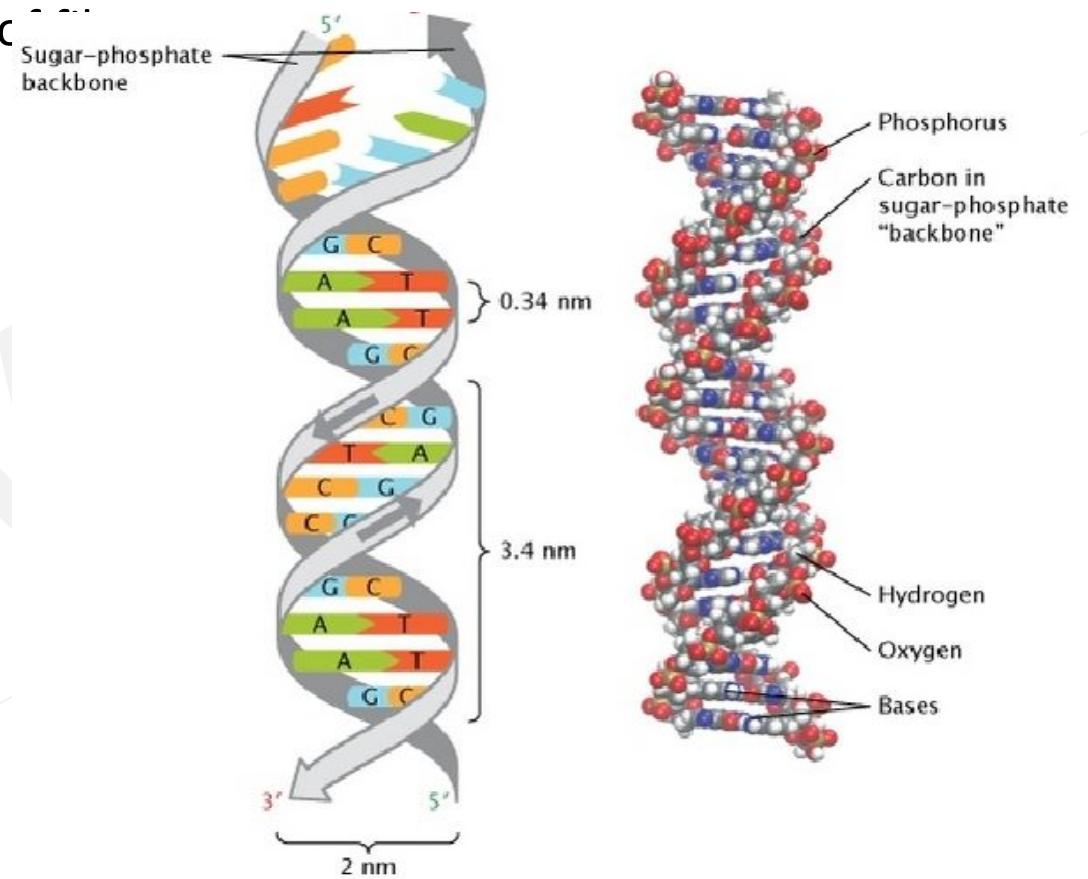
n	f(n)
0	1
1	1
2	2
3	3
4	5
5	8
6	13
7	21
8	34
9	55
10	89
11	144
12	233
13	377
14	610
15	987
16	1597
17	2584
18	4181
19	6765
20	10946

- There are four steps of dynamic programming
  - Characterize the solution of an optimal solution.
  - Recursively define the value of an optimal solution.
  - Compute the value of an optimal solution in a bottom-up-fashion.
  - Construct an optimal solution from computed information.

**माई असली बात तो example से समझ आएगी**

# Longest common subsequence

- The longest common subsequence (LCS) problem is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences).
- The longest common subsequence problem is a classic computer science problem, the basis of data comparison programs such as the diff utility, and has applications in computational linguistics and bioinformatics.
- It is also widely used by revision control systems such as Git for reconciling multiple changes made to a revision-controlled collection c



## Longest common subsequence

- Let there are two sequence X and Y, we say that a sequence Z is a common subsequence of X and Y where
- $X_m = \{x_1, x_2, x_3, \dots, x_m\}$
- $Y_n = \{y_1, y_2, y_3, \dots, y_n\}$
- we wish to find the maximum length of common of both X and Y,  
 $Z = \{z_1, z_2, z_3, \dots, z_k\}$

## STEP 1: Optimal Substructure

- $X_m = \{x_1, x_2, x_3, \dots, x_m\}$
- $Y_n = \{y_1, y_2, y_3, \dots, y_n\}$
- $Z = \{z_1, z_2, z_3, \dots, z_k\}$
- if  $x_m = y_n$ 
  - then  $z_k = x_m = y_n$  and  $Z_{k-1}$  is an LCS of  $X_{m-1}$  and  $Y_{n-1}$
- if  $x_m \neq y_n$ , then  $z_k \neq x_m$ 
  - Implies that  $Z_k$  is an LCS of  $X_{m-1}$  and  $Y_n$
- if  $x_m \neq y_n$ , then  $z_k \neq y_n$ 
  - Implies that  $Z_k$  is an LCS of  $X_m$  and  $Y_{n-1}$

## STEP 2: Recursive Solution

- Let us define  $C[i, j]$  to the length of L.C.S of sequence of  $X_i$  and  $Y_j$ , if either  $i=0$  or  $j=0$ , so LCS has length 0.
- The optimal substructure of LCS sub program gives the recursive formula
  - $C[i, j] = \{0 \quad \text{if } i=0 \text{ or } j=0\}$
  - $C[i, j] = \{C[i-1, j-1] + 1 \quad \text{if } i, j > 0 \text{ and } x_i = y_j\}$
  - $C[i, j] = \{\max(C[i, j-1], C[i-1, j]) \quad \text{if } i, j > 0 \text{ and } x_i \neq y_j\}$

## STEP 3: Computing the length of L.C.S

```
LCS-Length (x, y)
{
    m □ Length[x]
    n □ Length[y]
    for i □ 1 to m
    {
        do C[i, 0] □ 0
    }
    for j □ 0 to n
    {
        do C[0, j] □ 0
    }
    for i □ 1 to m
    {
        for j □ 1 to n
        {
            if (xi = yj)
            {
                c[i, j] □ c[i-1, j-1] + 1
                b[i, j] □ 'D_edge'
            }
            else if (C[i-1, j] >= C[i, j-1])
            {
                C[i, j] □ C[i-1, j]
                b[i, j] □ 'V_edge'
            }
            else
            {
                C[i, j] □ C[i, j-1]
                b[i, j] □ 'H_edge'
            }
        }
    }
    return b and C
}
```

The running time of the procedure is  $O(mn)$ , since each table entry takes  $O(1)$  time to compute.

```
Print_LCS (b, X, i, j)
{
    if i=0 or j=0
        return
    if (b[i, j] = 'D_edge')
    {
        then Print_LCS (b, X, i-1, j-1)
        Print Xi
    }
    Else if (b[i, j] = 'V_edge')
    {
        Print_LCS (b, X, i-1, j)
    }
    Else
    {
        Print_LCS (b, X, i, j-1)
    }
}
```

The procedure takes time  $O(m+n)$

Q Consider two strings X = "A, B, C, B, D, A, B" and Y = "B, D, C, A, B, A". Find the longest common subsequence?

B      D      C      A      B      A

	0	1	2	3	4	5	6
0							
A	1						
B	2						
C	3						
B	4						
D	5						
A	6						
B	7						

**Q** Consider two strings A = “qpqrr” and B = “pqprqrp”. Let x be the length of the longest common subsequence (not necessarily contiguous) between A and B and let y be the number of such longest common subsequences between A and B. Then  $x + 10y = \underline{\hspace{2cm}}$  (Gate-2014) (2 Marks)

**Q** A sub-sequence of a given sequence is just the given sequence with some elements (possibly none or all) left out. We are given two sequences  $X[m]$  and  $Y[n]$  of lengths  $m$  and  $n$  respectively, with indexes of  $X$  and  $Y$  starting from 0. We wish to find the length of the longest common sub-sequence(LCS) of  $X[m]$  and  $Y[n]$  as  $l(m, n)$ , where an incomplete recursive definition for the function  $l(i, j)$  to compute the length of The LCS of  $X[m]$  and  $Y[n]$  is given below **(Gate-2009) (2 Marks)**

$$l(i,j) = 0, \text{ if either } i=0 \text{ or } j=0$$

$$= \text{expr}_1, \text{ if } i, j > 0 \text{ and } X[i-1] = Y[j-1]$$

$$= \text{expr}_2, \text{ if } i, j > 0 \text{ and } X[i-1] \neq Y[j-1]$$

(A)  $\text{expr}_1 \equiv l(i-1, j) + 1$

(B)  $\text{expr}_1 \equiv l(i, j-1)$

(C)  $\text{expr}_2 \equiv \max(l(i-1, j), l(i, j-1))$

(D)  $\text{expr}_2 \equiv \max(l(i-1, j-1), l(i, j))$

	B	D	C	A	B	A
0						
A						
B						
C						
B						
D						
A						
B						

**Q** Consider the data given in the previous question. The values of  $I(i, j)$  could be obtained by dynamic programming based on the correct recursive definition of  $I(i, j)$  of the form given above, using an array  $L[M, N]$ , where  $M = m+1$  and  $N = n+1$ , such that  $L[i, j] = I(i, j)$ . Which one of the following statements would be TRUE regarding the dynamic programming solution for the recursive definition of  $I(i, j)$ ? **(Gate-2009) (2 Marks)**

- (A)** All elements  $L$  should be initialized to 0 for the values of  $I(i, j)$  to be properly computed
- (B)** The values of  $I(i, j)$  may be computed in a row major order or column major order of  $L(M, N)$
- (C)** The values of  $I(i, j)$  cannot be computed in either row major order or column major order of  $L(M, N)$
- (D)**  $L[p, q]$  needs to be computed before  $L[r, s]$  if either  $p < r$  or  $q < s$ .

	0	1	2	3	4	5	6
0							
A	1						
B	2						
C	3						
B	4						
D	5						
A	6						
B	7						

# Matrix chain Multiplication

- First lets understand what is the cost in terms of scalar multiplication for multiplying two matrix of compatible order.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} = \begin{bmatrix} (aj + bm + cp) & (ak + bn + cq) & (al + bo + cr) \\ (dj + em + fp) & (dk + en + fq) & (dl + eo + fr) \\ (gj + hm + ip) & (gk + hn + iq) & (gl + ho + ir) \end{bmatrix}$$

- In matrix-chain multiplication problem, we are not actually multiplying matrix. Our goal is only to determine an order for multiplying matrix that the lowest cost.

$$A_{2 \times 3} \times A_{3 \times 4} \times A_{4 \times 5}$$

$$A_{2 \times 3} \times A_{3 \times 4} \times A_{4 \times 5}$$

## STEP 1: The structure of an optimal parenthesization

- Suppose there are  $A_i, A_{i+1}, \dots, A_j$  matrix to be multiplied
- Now let split the product chain  $A_i, \dots, A_k, A_{k+1}, \dots, A_j$
- Let  $A_{ij}$  where  $i \leq j$  for the matrix that results from evaluation of the product  $A_i, \dots, A_k, A_{k+1}, \dots, A_j$
- if  $i \leq j$  then any parenthesization of product  $A_i, \dots, A_k, A_{k+1}, \dots, A_j$  must split the product between  $A_k$  &  $A_{k+1}$  in the range  $i \leq k < j$  for the value of k. we first compute the matrix  $A_i, \dots, A_k, A_{k+1}, \dots, A_j$ . then multiplying them to produce the final product  $A_i, \dots, A_j$ .
- The cost of this parenthesization is the cost of computing the matrix  $A_i * A_k$  and  $A_{k+1} \dots A_j$  and the cost of multiplying them together.
- Suppose that the optimal parenthesization of  $A_i, \dots, A_j$  splits the product between  $A_k$  and  $A_{k+1}$  then the parenthesization of prefix sub chain.  $A_1 \dots A_k$  with in this optimal parenthesization of  $A_i \dots A_j$  must be optimal parenthesization of  $A_1 \dots A_k$ . A similar observation holds for the parenthesization of sub chain  $A_{k+1} \dots A_j$

## STEP 2: Recursive Solution

- We define the cost of an optimal solution recursively in terms of the optimal solution to subproblems.
- Let  $m[i, j]$  be the minimum number of scalar product needed to compute the matrix  $A_{i \dots j}$ .
- Let us assume that the optimal parenthesization splits the product  $A_i, A_{i+1} \dots, A_j$  between  $A_k$  and  $A_{k+1}$ . where  $i \leq k < j$ . Then  $m[i, j]$  is equal to the minimum cost for computing the sub products  $A_i, \dots, A_k$  and  $A_{k+1}, \dots, A_j$  plus the cost of multiplying these two matrix together.
- Each matrix  $A_i$  is  $P_{i-1} * P_i$  we see that commuting the matrix product  $A_i, \dots, A_k, A_{k+1}, \dots, A_j$  takes  $P_{i-1} P_k P_j$  scalar multiplication.
- $m[i, j] = \{ 0 \text{ if } (i=j) \}$
- $m[i, j] = \{ \min[m[i, k] + m[k+1, j] + P_{i-1} P_k P_j] \text{ if } (i < j) \}$

## STEP 3: Computing the Optimal Cost

- $m[i, j] = \{ 0 \text{ if } (i=j) \}$
- $m[i, j] = \{ \min[m[i, k] + m[k+1, j] + P_{i-1} P_k P_j] \text{ if } (i < j) \}$

A simple inspection of the nested loop structure of MATRIX-CHAIN-ORDER yields a running time of  $O(n^3)$  for the algorithm

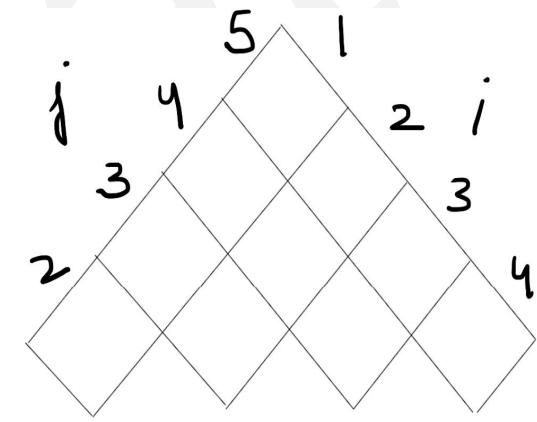
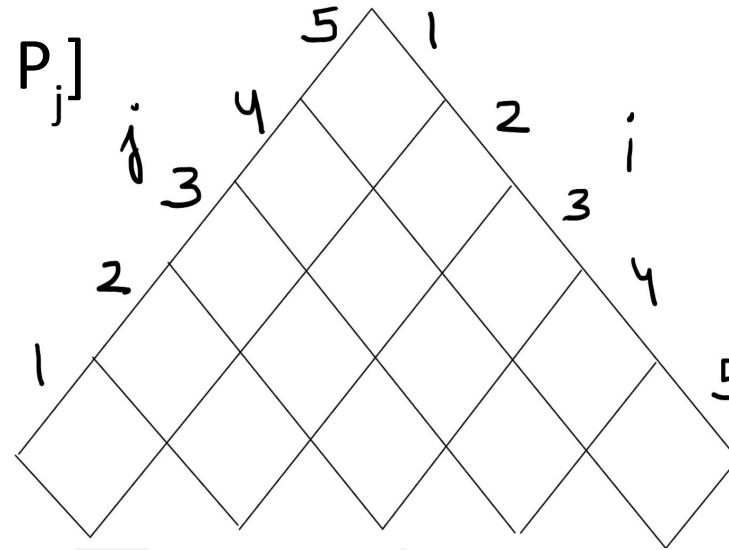
## STEP 4: Constructing an optimal Solution

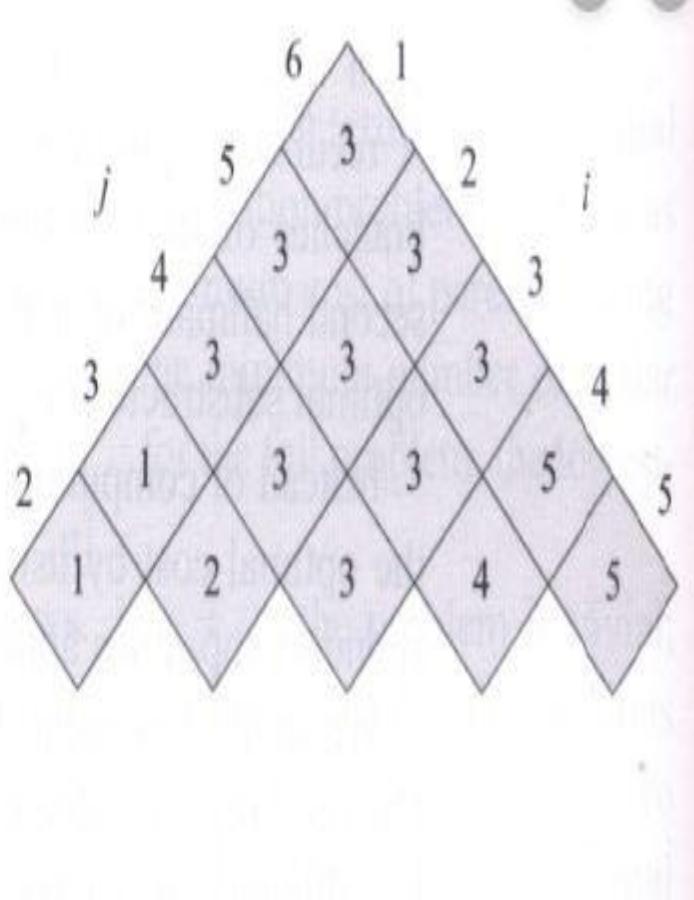
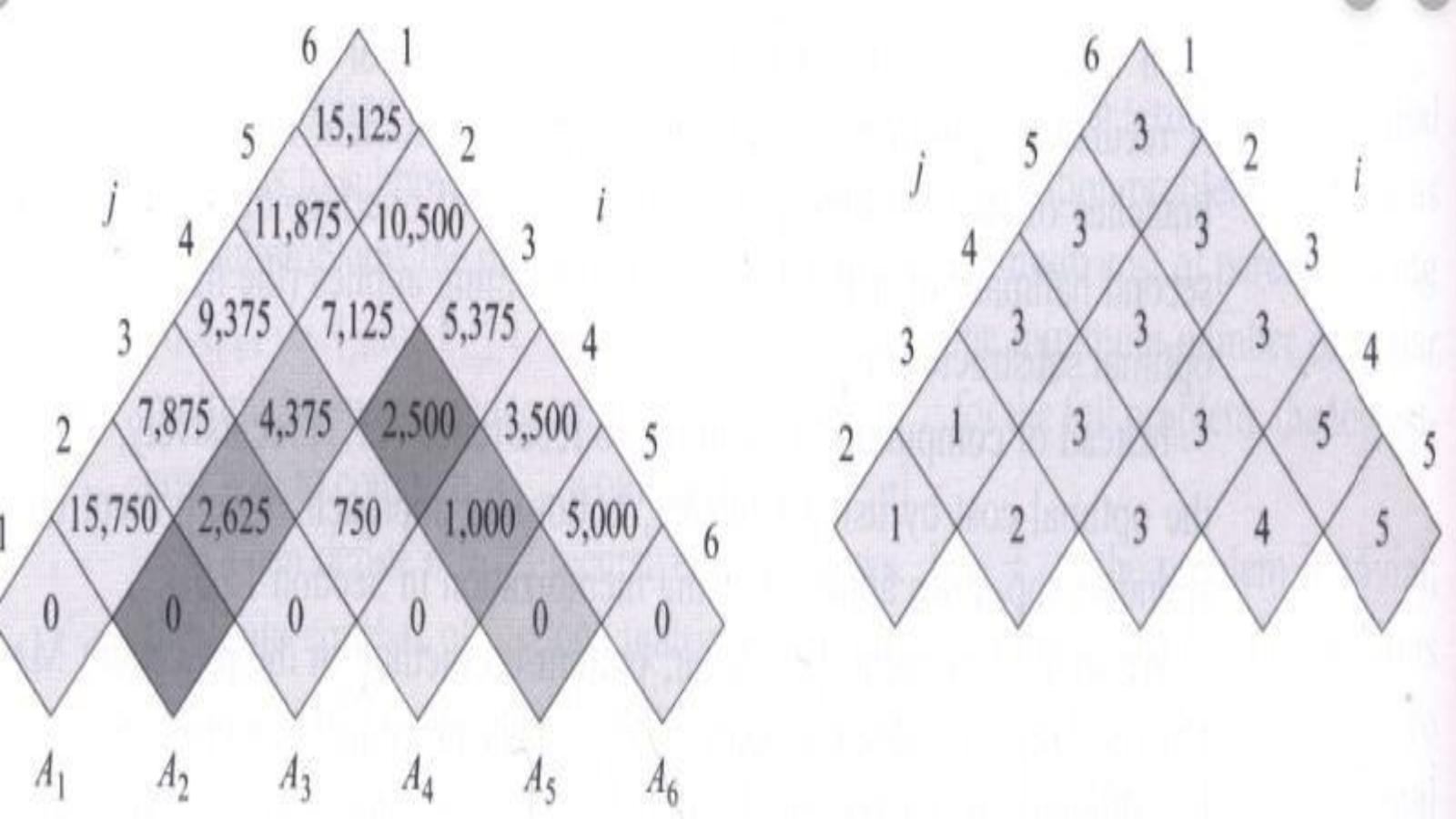
Print-Optimal Parenthesis ( $S, i, j$ )

```
{  
    if (i=j)  
    {  
        then print 'Ai'  
    }  
    Else  
    {  
        print "("  
        Print-Optimal Parenthesis (S, i, S [i, j])  
        Print-Optimal Parenthesis (S, S [i, j] + 1, j)  
        print ')'  
    }  
}
```

**Q** Let  $A_1, A_2, A_3, A_4$  and  $A_5$  be five matrices of dimensions  $30 \times 35$ ,  $35 \times 15$ ,  $15 \times 5$ ,  $5 \times 10$ , and  $10 \times 20$  respectively. The minimum number of scalar multiplications required to find the product  $A_1 A_2 A_3 A_4 A_5$  using the basic matrix multiplication method is?

$$m[i, j] = \{ \min[m[i, k] + m[k+1, j] + P_{i-1} P_k P_j] \}$$

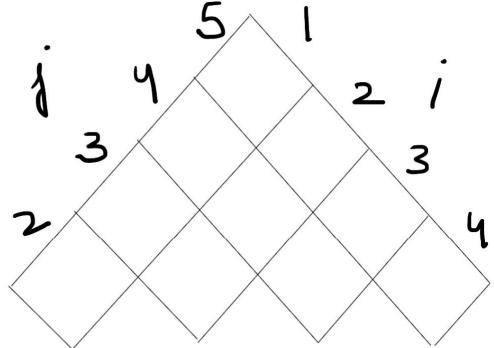
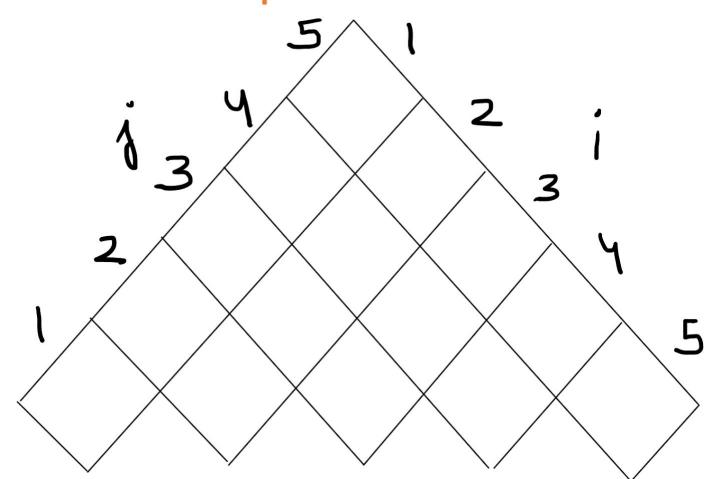




**Q** Assume that multiplying a matrix  $G_1$  of dimension  $p \times q$  with another matrix  $G_2$  of dimension  $q \times r$  requires  $pqr$  scalar multiplications. Computing the product of  $n$  matrices  $G_1 G_2 G_3 \dots G_n$  can be done by parenthesizing in different ways. Define  $G_i G_{i+1}$  as an **explicitly computed pair** for a given parenthesization if they are directly multiplied. For example, in the matrix multiplication chain  $G_1 G_2 G_3 G_4 G_5 G_6$  using parenthesization  $(G_1(G_2 G_3))(G_4(G_5 G_6))$ ,  $G_2 G_3$  and  $G_5 G_6$  are only explicitly computed pairs. Consider a matrix multiplication chain  $F_1 F_2 F_3 F_4 F_5$ , where matrices  $F_1, F_2, F_3, F_4$  and  $F_5$  are of dimensions  $2 \times 25, 25 \times 3, 3 \times 16, 16 \times 1$  and  $1 \times 1000$ , respectively. In the parenthesization of  $F_1 F_2 F_3 F_4 F_5$  that minimizes the total number of scalar multiplications, the explicitly computed pairs is/are (Gate-2018) (2 Marks)

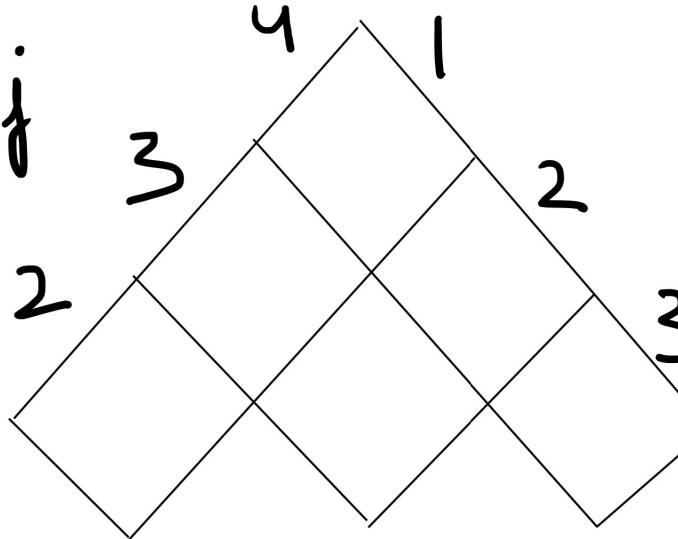
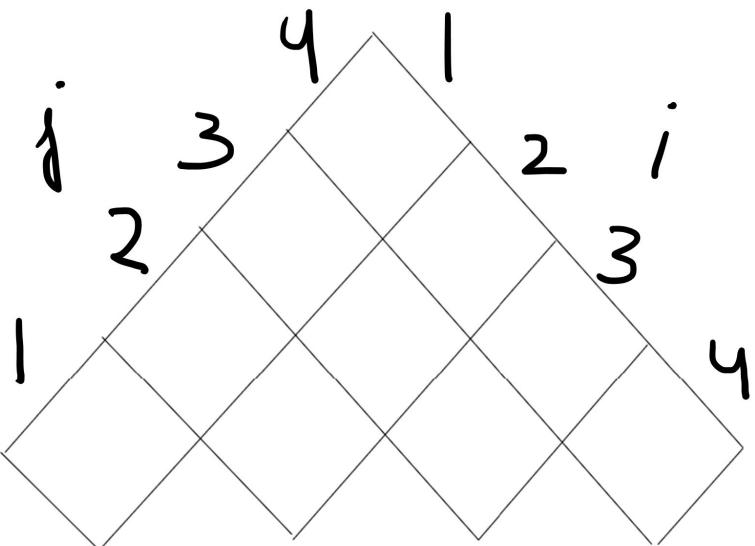
- (A)  $F_1 F_2$  and  $F_3 F_4$  only
- (C)  $F_3 F_4$  only

- (B)  $F_2 F_3$  only
- (D)  $F_1 F_2$  and  $F_4 F_5$  only



$$m[i, j] = \{ \min[m[i, k] + m[k+1, j] + P_{i-1} P_k P_j]$$

**Q** Let  $A_1, A_2, A_3$ , and  $A_4$  be four matrices of dimensions  $10 \times 5, 5 \times 20, 20 \times 10$ , and  $10 \times 5$ , respectively. The minimum number of scalar multiplications required to find the product  $A_1 A_2 A_3 A_4$  using the basic matrix multiplication method is **(Gate-2016) (2 Marks)**



$$m[i, j] = \{ \min[m[i, k] + m[k+1, j] + P_{i-1} P_k P_j]$$

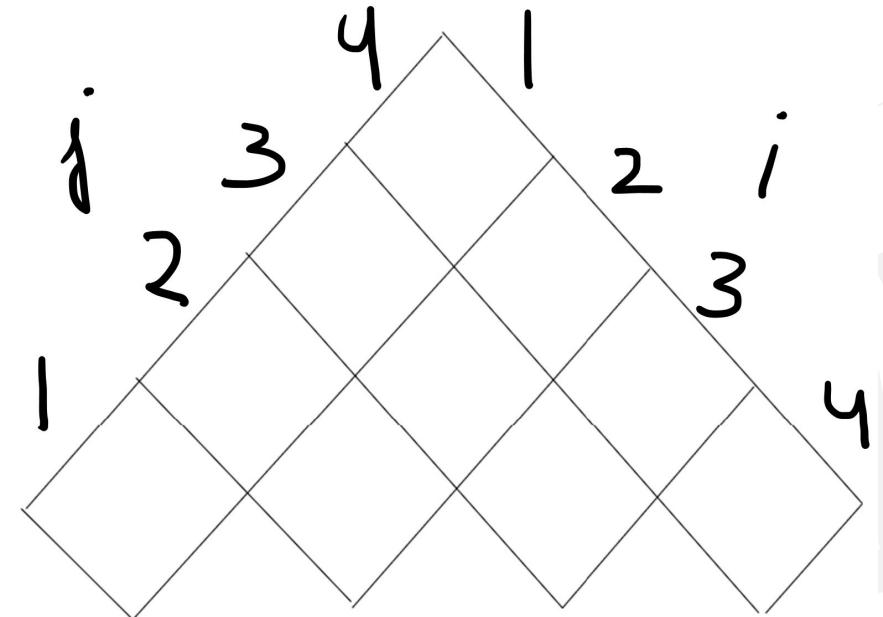
**Q** Four matrices  $M_1$ ,  $M_2$ ,  $M_3$  and  $M_4$  of dimensions  $p \times q$ ,  $q \times r$ ,  $r \times s$  and  $s \times t$  respectively can be multiplied in several ways with different number of total scalar multiplications. For example, when multiplied as  $((M_1 \times M_2) \times (M_3 \times M_4))$ , the total number of multiplications is  $pqr + rst + prt$ . When multiplied as  $((M_1 \times M_2) \times M_3) \times M_4$ , the total number of scalar multiplications is  $pqr + prs + pst$ . If  $p = 10$ ,  $q = 100$ ,  $r = 20$ ,  $s = 5$  and  $t = 80$ , then the number of scalar multiplications needed is:

a) 248000

b) 44000

c) 19000

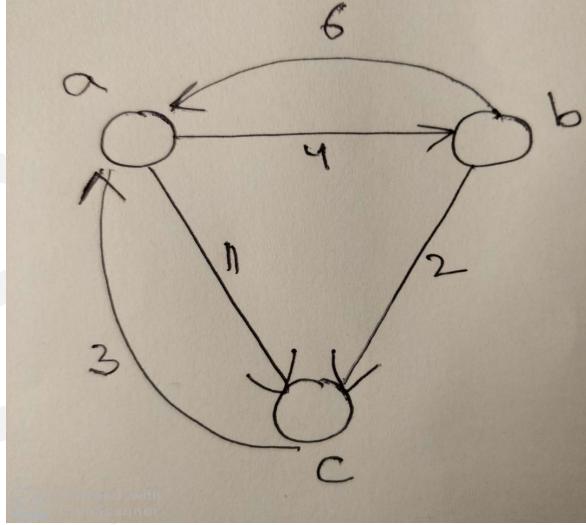
d) 25000



$$m[i, j] = \{ \min[m[i, k] + m[k+1, j] + P_{i-1} P_k P_j]$$

## All Pair Shortest Path-Floyd Warshall Algorithm

Q Consider a Directed weighted Graph and find all pair shortest path?



running time of  $O(n^3)$  for the algorithm

**Q** The Floyd-Warshall algorithm for all-pair shortest paths computation is based on:  
**(Gate-2016) (1 Marks)**

- (A)** Greedy paradigm.
- (B)** Divide-and-Conquer paradigm.
- (C)** Dynamic Programming paradigm.
- (D)** neither Greedy nor Divide-and-Conquer nor Dynamic Programming paradigm.

**Q** Consider the weighted undirected graph with 4 vertices, where the weight of edge {i, j} g is given by the entry  $W_{ij}$  in the matrix W. The largest possible integer value of x, for which at least one shortest path between some pair of vertices will contain the edge with weight x is \_\_\_\_\_

**(Gate-2016) (2 Marks)**

$$W = \begin{bmatrix} 0 & 2 & 8 & 5 \\ 2 & 0 & 5 & 8 \\ 8 & 5 & 0 & x \\ 5 & 8 & x & 0 \end{bmatrix}$$

**Q** Let  $G(V, E)$  be a directed graph with  $n$  vertices. A path from  $v_i$  to  $v_j$  in  $G$  is sequence of vertices  $(v_i, v_{i+1}, \dots, v_j)$  such that  $(v_k, v_{k+1}) \in E$  for all  $k$  in  $i$  through  $j - 1$ . A simple path is a path in which no vertex appears more than once.

Let  $A$  be an  $n \times n$  array initialized as follow

$$A[j, k] = \begin{cases} 1 & \text{if } (j, k) \in E \\ 0 & \text{otherwise} \end{cases}$$

Consider the following algorithm.

```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            A[j, k] = max (A[j, k], A[j, i] + A[i, k]);
```

Which of the following statements is necessarily true for all  $j$  and  $k$  after terminal of the above algorithm? (Gate-2003) (2 Marks)

- (A)  $A[j, k] \leq n$
- (B) If  $A[j, k] \geq n - 1$ , then  $G$  has a Hamiltonian cycle
- (C) If there exists a path from  $j$  to  $k$ ,  $A[j, k]$  contains the longest path length from  $j$  to  $k$
- (D) If there exists a path from  $j$  to  $k$ , every simple path from  $j$  to  $k$  contain most  $A[j, k]$  edges

## Sum of subset problem

- Given a set of non-negative integers, and a value  $sum$ , determine if there is a subset of the given set with sum equal to given  $sum$ .

**Q** Consider a set of non-negative integer  $S = \{2, 3, 7, 8, 10\}$ , find if there is a sub set of  $S$  with sum equal to 14?

**Q** The subset-sum problem is defined as follows. Given a set of  $n$  positive integers,  $S = \{a_1, a_2, a_3, \dots, a_n\}$  and positive integer  $W$ , is there a subset of  $S$  whose elements sum to  $W$ ? A dynamic program for solving this problem uses a 2-dimensional Boolean array  $X$ , with  $n$  rows and  $W+1$  column.  $X[i, j], 1 \leq i \leq n, 0 \leq j \leq W$ , is TRUE if and only if there is a subset of  $\{a_1, a_2, \dots, a_i\}$  whose elements sum to  $j$ . Which of the following is valid for  $2 \leq i \leq n$  and  $a_i \leq j \leq W$ ?

**(Gate-2008) (2 Marks)**

- (A)  $X[i, j] = X[i - 1, j] \vee X[i, j - a_i]$
- (B)  $X[i, j] = X[i - 1, j] \vee X[i - 1, j - a_i]$
- (C)  $X[i, j] = X[i - 1, j] \wedge X[i, j - a_i]$
- (D)  $X[i, j] = X[i - 1, j] \wedge X[i - 1, j - a_i]$

**Q** In the above question, which entry of the array X, if TRUE, implies that there is a subset whose elements sum to W?

(A)  $X[1, W]$

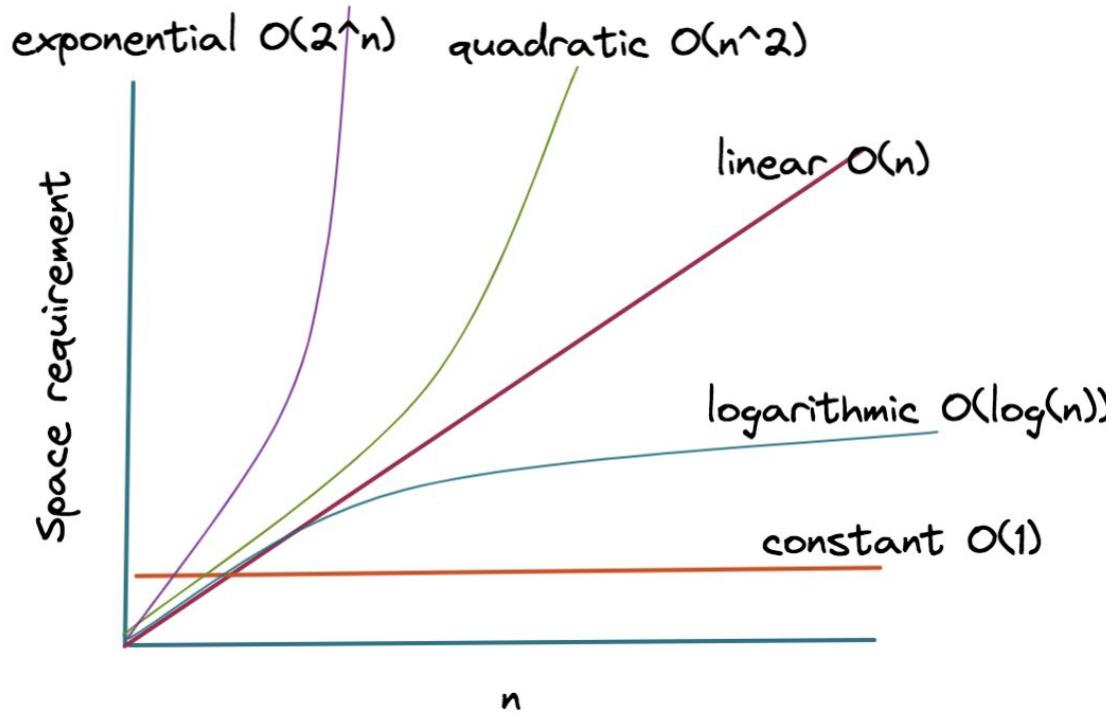
(B)  $X[n, 0]$

(C)  $X[n, W]$

(D)  $X[n - 1, n]$

# Asymptotic Notations

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



# Big O Notation

- The Big O notation defines an upper bound of an algorithm, it bounds a function only from above.
- The Big O notation is useful when we only have upper bound on time complexity of an algorithm.
- Many times, we easily find an upper bound by simply looking at the algorithm.
- $O(g(n)) = \{f(n) : \text{there exist positive constants } C \text{ and } N_0 \text{ such that } 0 \leq f(n) \leq C \cdot g(n) \text{ for all } n \geq N_0\}$



## **Ω Notation**

- Just as Big O notation provides an asymptotic upper bound on a function,  $\Omega$  notation provides an asymptotic lower bound.
- $\Omega$  Notation can be useful when we have lower bound on time complexity of an algorithm.
- For a given function  $g(n)$ , we denote by  $\Omega(g(n))$  the set of functions.
- $\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq c*g(n) \leq f(n) \text{ for all } n \geq n_0\}$ .



## Theta Notation

- **Θ Notation:** The theta notation bounds a function from above and below, so it defines exact asymptotic behaviour.
- For a given function  $g(n)$ , we denote  $\Theta(g(n))$  is following set of functions.
- $\Theta(g(n)) = \{f(n) : \text{there exist positive constants } C_1, C_2 \text{ and } n_0 \text{ such that } 0 \leq C_1 * g(n) \leq f(n) \leq C_2 * g(n) \text{ for all } n \geq n_0\}$
- The above definition means, if  $f(n)$  is theta of  $g(n)$ , then the value  $f(n)$  is always between  $C_1 * g(n)$  and  $C_2 * g(n)$  for large values of  $n$  ( $n \geq n_0$ ).
- The definition of theta also requires that  $f(n)$  must be non-negative for values of  $n$  greater than  $n_0$ .



## Small notations

- Every thing is same as big notations just, just we take strictly increasing or monotonically increasing case and equal case is not allowed.
- Analogy of asymptotic notation with real numbers
  - $f(n)$  is  $O(g(n))$        $a \leq b$
  - $f(n)$  is  $\Omega(g(n))$        $a \geq b$
  - $f(n)$  is  $\Theta(g(n))$        $a = b$
  - $f(n)$  is  $o(g(n))$        $a < b$
  - $f(n)$  is  $\omega(g(n))$        $a > b$

# Properties of Asymptotic notations

- Reflexivity:
  - $f(n) = O(f(n))$
  - $f(n) = \Omega(f(n))$
  - $f(n) = \Theta(f(n))$
- Symmetry:
  - $f(n) = \Theta(g(n))$  if and only if  $g(n) = \Theta(f(n))$
- Transitivity:
  - $f(n) = O(g(n))$  and  $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
  - $f(n) = \Theta(g(n))$  and  $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
  - $f(n) = \Omega(g(n))$  and  $g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$
  - $f(n) = o(g(n))$  and  $g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$
  - $f(n) = \omega(g(n))$  and  $g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$

- Transpose Symmetry:
  - $f(n) = O(g(n))$  if and only if  $g(n) = \Omega(f(n))$
- if  $f(n) = O(g(n))$ 
  - a  $f(n) = O(g(n))$
- if  $f(n)$  is  $O(g(n))$  and  $p(n)$  is  $O(q(n))$ 
  - $f(n) + p(n)$  is  $O(\max(g(n), q(n)))$
- $f(n) * p(n)$  is  $O(g(n) . q(n))$

## Types of Analysis

- **Worst Case Analysis:** In the worst-case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. It is that i/p class for which the algo takes maximum time. e.g. quick sort takes maximum time on a sorted i/p array.
- **Best Case Analysis:** In the best-case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. It is that i/p class for which the algo takes minimum time.
- **Average Case Analysis:** In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. Identification of all i/p class  $I_1, I_2, I_3 \dots I_k$ . Determine the probability that the algo take the i/p from the respective i/p class ( $P_1, P_2, P_3 \dots P_k$ ). Determine the time taken by the algo for each input class based on the order of magnitude ( $T_1, T_2, T_3, \dots, T_k$ ).
- $A(n) = \sum_{i=1}^k P_i * T_i$

1	2	3	4	5	6
57	12	43	68	26	35

# Conclusion

- The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.
- Most useful analysis is worst case analysis, as help designers to make reliable system even in worst case.

## Order of magnitude

- For time complexity of an algorithm will use Order of magnitude of a statement in algorithm.
- Order of magnitude of a statement is number of times the statement(fundamental operations) is executed while running.
- We have to find the order of magnitude of every statement.

- It is a two-step process
  - Find the number of fundamental operations
  - Finding the number of times, the fundamental operation executes.

```
int a = 0, b = 0; ----- O(1)
for (i = 0; i < N; i++) ----- O(n)
{
    a = a + 1; ----- O(n)
}
```

O(n)

- It is a two-step process
  - Find the number of fundamental operations
  - Finding the number of times, the fundamental operation executes.

```
int a = 0, b = 0; ----- O(1)
for (i = 0; i < N; i++) ----- O(n)
{
    a = a + 1; ----- O(n)
}
for (j = 0; j < M; j++) ----- O(m)
{
    b = b + 1; ----- O(m)
}
```

O( $n+m$ )

```
int a = 0; ----- (1)
for (i = 0; i < N; i++) ----- (n)
{
    for (j = N; j > i; j--) ----- (n2)
    {
        a = a + i + j; ----- (n2)
    }
}
```

**O(n<sup>2</sup>)**

```
int a = 0, b = 0; -----O(1)
for (i = 0; i < N; i++) -----O(n)
{
    a = a + 1; -----O(n)
}
int a = 0; -----(1)
for (i = 0; i < N; i++) -----(n)
{
    for (j = N; j > i; j--) --(n2)
    {
        a = a + i + j; -----(n2)
    }
}
```

O( $n^2$ )

```
int i = 0; -----O(1)
while (i <= n) ----O(n/2)
{
    i = i + 2; ----O(n/2)
}
```

O(n)

```
int a = 0, i = N; -----O(1)
while (i > 0) -----O(log2n)
{
    a = a + i; -----O(log2n)
    i = i / 2; -----O(log2n)
}
```

O(log<sub>2</sub>n)

```
int i, j, k = 0; -----O(1)
for (i = n / 2; i <= n; i++) -----O(n/2)
{
    for (j = 2; j <= n; j = j * 2) ---O(nlog2n/2)
    {
        k = k + n / 2; -----O(nlog2n/2)
    }
}
```

O( $n \log_2 n$ )

```
for(int i=0;i<n;i++) ---O(logkn)
```

```
{
```

```
    i = i * k; -----O(logkn)
```

```
}
```

O(log<sub>k</sub>n)

```
int i = 0; ----- O(1)
while (i * i <= n) -- O( $\sqrt{n}$ )
{
    i = i + 1; -- O( $\sqrt{n}$ )
}
```

O( $\sqrt{n}$ )

**Q** Let  $w(n)$  and  $A(n)$  denote respectively, the worst case and average case running time of an algorithm executed on an input of size  $n$ . which of the following is ALWAYS TRUE? **(Gate-2012) (1- Marks)**

(A)  $A(n) = \Omega(W(n))$

(B)  $A(n) = \Theta(W(n))$

(C)  $A(n) = O(W(n))$

(D)  $A(n) = o(W(n))$

**Q** Let  $f(n)$ ,  $g(n)$  and  $h(n)$  be functions defined for positive integers such that  $f(n) = O(g(n))$ ,  $g(n) \neq O(f(n))$ ,  $g(n) = O(h(n))$ , and  $h(n) = O(g(n))$ .

Which one of the following statements is FALSE? **(Gate-2004) (2 Marks)**

(A)  $f(n) + g(n) = O(h(n)) + h(n)$

(B)  $f(n) = O(h(n))$

(C)  $h(n) \neq O(f(n))$

(D)  $f(n)h(n) \neq O(g(n)h(n))$

Consider the following two functions:

**(Gate-1994) (2 Marks)**

$$g_1(n) = \begin{cases} n^3 & \text{for } 0 \leq n \leq 10,000 \\ n^2 & \text{for } n \geq 10,000 \end{cases}$$

$$g_2(n) = \begin{cases} n & \text{for } 0 \leq n \leq 100 \\ n^3 & \text{for } n > 100 \end{cases}$$

Which of the following is true?

- A.  $g_1(n)$  is  $O(g_2(n))$
- B.  $g_1(n)$  is  $O(n^3)$
- C.  $g_2(n)$  is  $O(g_1(n))$
- D.  $g_2(n)$  is  $O(n)$

Q Consider the following three claims

I  $(n + k)^m = \underline{(n^m)}$ , where k and m are constants

II  $2^{(n + 1)} = O(2^n)$

III  $2^{(2n + 1)} = O(2^n)$

Which of these claims are correct? (GATE-2003) (1 Marks)

**(A)** I and II

**(B)** I and III

**(C)** II and III

**(D)** I, II and III

**Q** Let  $f(n) = n$  and  $g(n) = n^{(1+\sin n)}$ , where  $n$  is a positive integer. Which of the following statements is/are correct? **(Gate-2015) (2 Marks)**

- I.  $f(n) = O(g(n))$       II.  $f(n) = \Omega(g(n))$

**(A)** Only I

**(B)** Only II

**(C)** Both I and II

**(D)** Neither I nor II

**Q** Two alternative packages A and B are available for processing a database having  $10^k$  records. Package A requires  $0.0001n^2$  time units and package B requires  $10n\log_{10}n$  time units to process n records. What is the smallest value of k for which package B will be preferred over A? **(Gate-2010) (1 Marks)**

a) 12

b) 10

c) 6

d) 5

**Q** Which one of the following statements is TRUE for all positive functions  $f(n)$ ?  
**(GATE 2022) (1 MARKS)**

- (A)  $f(n^2) = \Theta(f(n)^2)$ , when  $f(n)$  is a polynomial
- (B)  $f(n^2) = o(f(n)^2)$
- (C)  $f(n^2) = O(f(n)^2)$ , when  $f(n)$  is an exponential function
- (D)  $f(n^2) = \Omega(f(n)^2)$

**Q** Consider the following three functions.

$$f_1 = 10^n$$

$$f_2 = n^{\log n}$$

$$f_3 = n^{\sqrt{n}}$$

Which one of the following options arranges the functions in the increasing order of asymptotic growth rate? **(Gate 2021)**

(a)  $f_3, f_2, f_1$

(b)  $f_2, f_1, f_3$

(c)  $f_1, f_2, f_3$

(d)  $f_2, f_3, f_1$

**Q** Consider the following functions from positives integers to real numbers  $10, \sqrt{n}, n, \log_2 n, 100/n$ . The CORRECT arrangement of the above functions in increasing order of asymptotic complexity is: **(GATE-2017) (1 Marks)**

- (A)  $\log_2 n, 100/n, 10, \sqrt{n}, n$
- (B)  $100/n, 10, \log_2 n, \sqrt{n}, n$
- (C)  $10, 100/n, \sqrt{n}, \log_2 n, n$
- (D)  $100/n, \log_2 n, 10, \sqrt{n}, n$

**Q** Let  $f(n) = n^2 \log n$  and  $g(n) = n (\log n)^{10}$  be two positive functions of  $n$ . Which of the following statements is correct? **(Gate-2001) (1 Marks)**

**(A)**  $f(n) = O(g(n))$  and  $g(n) \neq O(f(n))$

**(B)**  $f(n) \neq O(g(n))$  and  $g(n) = O(f(n))$

**(C)**  $f(n) = O(g(n))$  but  $g(n) = O(f(n))$

**(D)**  $f(n) \neq O(g(n))$  but  $g(n) \neq O(f(n))$

**Q** The increasing order of following functions in terms of asymptotic complexity is  
**(Gate-2015) (2 Marks)**

---

$$f_1(n) = n^{0.999999} \log n$$

$$f_2(n) = 10000000n$$

$$f_3(n) = 1.000001^n$$

$$f_4(n) = n^2$$

- (A)  $f_1(n); f_4(n); f_2(n); f_3(n)$
- (B)  $f_1(n); f_2(n); f_3(n); f_4(n);$
- (C)  $f_2(n); f_1(n); f_4(n); f_3(n)$
- (D)  $f_1(n); f_2(n); f_4(n); f_3(n)$

**Q** Consider the following functions:

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

$$g(n) = n!$$

$$h(n) = n^{\log n}$$

Which of the following statements about the asymptotic behaviour of  $f(n)$ ,  $g(n)$ , and  $h(n)$  is true? **(Gate-2008) (2 Marks)**

(A)  $f(n) = O(g(n))$ ;  $g(n) = O(h(n))$

(C)  $g(n) = O(f(n))$ ;  $h(n) = O(f(n))$

(B)  $f(n) = \Omega(g(n))$ ;  $g(n) = O(h(n))$

(D)  $h(n) = O(f(n))$ ;  $g(n) = \Omega(f(n))$

**Q** Arrange the following functions in increasing asymptotic order: (GATE-2008) (1 Marks)

- a)  $n^{1/3}$     b)  $e^n$     c)  $n^{7/4}$     d)  $n \log^9 n$     e)  $1.000001^n$

(A) A, D, C, E, B

(B) D, A, C, E, B

(C) A, C, D, E, B

(D) A, C, D, B, E

**Q** Which of the given options provides the increasing order of asymptotic complexity of functions f<sub>1</sub>, f<sub>2</sub>, f<sub>3</sub> and f<sub>4</sub>? **(Gate-2011) (2 Marks)**

$$f_1(n) = 2^n \quad f_2(n) = n^{(3/2)} \quad f_3(n) = n\log n \quad f_4(n) = n^{(\log n)}$$

(A) f<sub>3</sub>, f<sub>2</sub>, f<sub>4</sub>, f<sub>1</sub>

(B) f<sub>3</sub>, f<sub>2</sub>, f<sub>1</sub>, f<sub>4</sub>

(C) f<sub>2</sub>, f<sub>3</sub>, f<sub>1</sub>, f<sub>4</sub>

(D) f<sub>2</sub>, f<sub>3</sub>, f<sub>4</sub>, f<sub>1</sub>

**Q** Consider the following functions

$$f(n) = 3n^{\sqrt{n}}$$

$$g(n) = 2^{\sqrt{n} \log_2 n}$$

$$h(n) = n!$$

Which of the following is true? **(GATE-2000) (2 Marks)**

- (a)  $h(n)$  is  $O(f(n))$
- (b)  $h(n)$  is  $O(g(n))$
- (c)  $g(n)$  is not  $O(f(n))$
- (d)  $f(n)$  is  $O(g(n))$

## (Gate-1996) (1 Marks)

Which of the following is false?

- A.  $100n \log n = O\left(\frac{n \log n}{100}\right)$
- B.  $\sqrt{\log n} = O(\log \log n)$
- C. If  $0 < x < y$  then  $n^x = O(n^y)$
- D.  $2^n \neq O(nk)$

**Q** The equality above remains correct if X is replaced by (Gate-2015) (1 Marks)

Consider the equality  $\sum_{i=0}^n i^3 = X$  and the following choices for X

- I.  $\Theta(n^4)$
- II.  $\Theta(n^5)$
- III.  $O(n^5)$
- IV.  $\Omega(n^3)$

- (A) Only I
- (B) Only II
- (C) I or III or IV but not II
- (D) II or III or IV but not I

**Q.** Given an Integer array of size N, we want to check if the array is sorted (in either ascending or descending order). An algorithm solves this problem by making a single pass through the array and comparing each element of the array only with its adjacent elements. The worst-case time complexity of this algorithm is **(Gate 2024,CS) (1 Marks)** **(MCQ)**

- (a) both  $O(N)$  and  $\Omega(N)$
- (b)  $O(N)$  but not  $\Omega(n)$
- (c)  $\Omega(N)$  but not  $O(N)$
- (d) neither  $O(N)$  nor  $\Omega(N)$

**Q** Which of the following is not  $O(n^2)$ ?

(A)  $(15^{10}) * n + 12099$

(B)  $n^{1.98}$

(C)  $n^3 / (\sqrt{n})$

(D)  $(2^{20}) * n$

- There are mainly three ways for solving recurrences.
  1. Substitution Method
  2. Master Method
  3. Recurrence Tree Method

## Master Theorem

- In the analysis of algorithms, the master theorem for divide-and-conquer recurrences provides an asymptotic analysis (using Big O notation) for recurrence relations of types that occur in the analysis of many divide and conquer algorithms.
- The approach was first presented by Jon Bentley, Dorothea Haken, and James B. Saxe in 1980, where it was described as a "unifying method" for solving such recurrences.
- The name "master theorem" was popularized by the widely used algorithms textbook *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein.

- $T(n) = a T(n/b) + f(n)$
- The above equation divides the problem into ‘a’ number of subproblems recursively,  $a \geq 1$
- Each subproblem being of size  $n/b$ . the subproblems (of size less than  $k$ ) that do not recurse. ( $b > 1$ )
- where  $f(n)$  is the time to create the subproblems and combine their results in the above procedure.

- The master theorem allows many recurrence relations of this form to be converted to  $\Theta$ -notation directly, without doing an expansion of the recursive relation.
- The master theorem often yields asymptotically tight bounds to some recurrences from divide and conquer algorithms that partition an input into smaller subproblems of equal sizes, solve the subproblems recursively, and then combine the subproblem solutions to give a solution to the original problem.

## Case 1

- $f(n) = O(n^{\log_b a - \epsilon})$  for some constant  $\epsilon > 0$ ,
- then  $T(n) = \Theta(n^{\log_b a})$

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

knowledgeGate

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

knowledgeGate

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

knowledgeGate

$$Q T(n) = 8T(n/2) + n^2$$

knowledgeGate

## Case 2

- $f(n) = \Theta(n^{\log_b a})$ ,
- then  $T(n) = \Theta(n^{\log_b a} \lg n)$

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

knowledgeGate

$$Q T(n) = T(2n/3) + 1$$

knowledgeGate

## Case 3

- $f(n) = \Omega(n^{\frac{\log_b a + \epsilon}{b}})$  for some constant  $\epsilon > 0$ ,
- and if  $f(n/b) \leq c f(n)$  for some constant  $c < 1$  and all sufficiently large  $n$ ,
- then  $T(n) = \Theta(f(n))$

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

knowledgeGate

**Q** For constants  $a \geq 1$  and  $b > 1$ , consider the following recurrence defined on the non-negative integers:

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

Which one of the following options is correct about the recurrence  $T(n)$ ? **(GATE 2021) (2 MARKS)**

- A. If  $f(n)$  is  $n \log_2(n)$ , then  $T(n)$  is  $\Theta(n \log_2(n))$
- B. If  $f(n)$  is  $\frac{n}{\log_2(n)}$ , then  $T(n)$  is  $\Theta(\log_2(n))$
- C. If  $f(n)$  is  $O(n^{\log_b(a)-\epsilon})$  for some  $\epsilon > 0$ , then  $T(n)$  is  $\Theta(n^{\log_b(a)})$
- D. If  $f(n)$  is  $\Theta(n^{\log_b(a)})$ , then  $T(n)$  is  $\Theta(n^{\log_b(a)})$

**Q** The running time of an algorithm is represented by the following recurrence relation:

**if**  $n \leq 3$  **then**  $T(n) = n$   
**else**  $T(n) = T(n/3) + cn$

Which one of the following represents the time complexity of the algorithm? **(Gate-2009)**  
**(2 Marks)**

- (A)**  $\Theta(n)$       **(B)**  $\Theta(n \log n)$       **(C)**  $\Theta(n^2)$       **(D)**  $\Theta(n^{2\log n})$

**Q** Suppose  $T(n) = 2T(n/2) + n$ ,  $T(0) = T(1) = 1$

Which one of the following is FALSE? **(Gate-2005) (2 Marks)**

(A)  $T(n) = O(n^2)$

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

(C)  $T(n) = \Omega(n^2)$

(D)  $T(n) = O(n \log n)$

**Q** Let  $T(n)$  be a function defined by the recurrence

$$T(n) = 2T(n/2) + \sqrt{n} \text{ for } n \geq 2 \text{ and } T(1) = 1$$

Which of the following statements is TRUE? **(Gate-2005) (2 Marks)**

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

(B)  $T(n) = \Theta(\sqrt{n})$

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

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

**Q** Consider the recurrence function

$$T(n) = \begin{cases} 2T(\sqrt{n}) + 1, & n > 2 \\ 2, & 0 < n \leq 2 \end{cases}$$

Then  $T(n)$  in terms of  $\Theta$  notation is **(Gate-2017) (2 Marks)**

a)  $\Theta(\log\log n)$

b)  $\Theta(\log n)$

c)  $\Theta(\sqrt{n})$

d)  $\Theta(n)$

**Q** Which one of the following correctly determines the solution of the recurrence relation with  $T(1) = 1$ ? **(Gate-2014) (1 Marks)**

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

a)  $\Theta(n)$

b)  $\Theta(n\text{Log } n)$

c)  $\Theta(n^*n)$

d)  $\Theta(\log n)$

**Q** Consider the following recurrence:

$$T(n)=2T(\sqrt{n})+1, T(1)=1$$

Which one of the following is true? **(Gate-2006) (2 Marks)**

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

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

c)  $T(n)=\Theta(\sqrt{n})$

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

**Q** The solution to the recurrence equation  $T(2^k) = 3 T(2^{k-1}) + 1$ ,  $T(1) = 1$ , is: (Gate-2002) (2 Marks)

- (A)  $2^k$
- (B)  $(3^{k+1} - 1)/2$
- (C)  $3^{\log_2 k}$
- (D)  $2^{\log_3 k}$

**Q. Consider the following recurrence relation: Which one of the following options is CORRECT? (Gate 2024,CS) (2 Marks) (MCQ)**

- (a)  $T(n)=\Theta(n \log \log n)$
- (b)  $T(n)=\Theta(n \log n)$
- (c)  $T(n)=\Theta(n^2 \log n)$
- (d)  $T(n)=\Theta(n^2 \log \log n)$

$$T(n) = \begin{cases} \sqrt{n}T(\sqrt{n}) + n & \text{for } n \geq 1, \\ 1 & \text{for } n = 1. \end{cases}$$

**Q.** Consider the following recurrence relation:

$$T(n) = 2T(n - 1) + n2^n \text{ for } n > 0, T(0) = 1.$$

Which ONE of the following options is CORRECT? **(Gate 2025)**

- A)  $T(n) = \Theta(n^2 2^n)$
- B)  $T(n) = \Theta(n 2^n)$
- C)  $T(n) = \Theta((\log n)^2 2^n)$
- D)  $T(n) = \Theta(4^n)$

$$T(n) \leq \begin{cases} c, & \text{if } n \leq 1, \\ aT(n-b) + f(n), & n > 1, \end{cases}$$

If  $f(n)$  is  $O(n^k)$ , then

1. If  $a < 1$  then  $T(n) = O(n^k)$
2. If  $a = 1$  then  $T(n) = O(n^{k+1})$
3. if  $a > 1$  then  $T(n) = O(n^k a^{n/b})$

**Q** The recurrence relation capturing the optimal execution time of the Towers of Hanoi problem with n discs is **(Gate-2012) (1 Marks)**

(A)  $T(n) = 2T(n - 2) + 2$

(B)  $T(n) = 2T(n - 1) + n$

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

(D)  $T(n) = 2T(n - 1) + 1$

**Q** Consider the following recurrence relation

$$T(1) = 1$$

$$T(n + 1) = T(n) + \lfloor \sqrt{n + 1} \rfloor \text{ for all } n \geq 1$$

The value of  $T(m^2)$  for  $m \geq 1$  is (Gate-2003) (2 Marks)

(A)  $(m/6)(21m - 39) + 4$

(B)  $(m/6)(4m^2 - 3m + 5)$

(C)  $(m/2)(m^{2.5} - 11m + 20) - 5$

(D)  $(m/6)(5m^3 - 34m^2 + 137m - 104) + (5/6)$

**Q** The recurrence equation

$$T(1) = 1$$

$$T(n) = 2T(n - 1) + n, n \geq 2$$

evaluates to **(Gate-2004) (2 Marks)**

(A)  $2^{n+1} - n - 2$

(B)  $2^n - n$

(C)  $2^{n+1} - 2n - 2$

(D)  $2^n + n$

**Q Solve the recurrence equations: (Gate-1987) (2 Marks)**

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

$$T(1)=1$$

Q. Let  $T(n)$  be the recurrence relation defined as follows:

$$T(0) = 1,$$

$$T(1) = 2, \text{ and}$$

$$T(n) = 5T(n - 1) - 6T(n - 2) \text{ for } n \geq 2$$

Which one of the following statements is TRUE? **(Gate 2024 CS) (1 Mark) (MCQ)**

- (a)  $T(n) = \Theta(2^n)$
- (b)  $T(n) = \Theta(n2^n)$
- (c)  $T(n) = \Theta(3^n)$
- (d)  $T(n) = \Theta(n3^n)$

**Q.** A meld operation on two instances of a data structure combines them into one single instance of the same data structure. Consider the following data structures:

P: Unsorted doubly linked list with pointers to the head node and tail node of the list.

Q: Min-heap implemented using an array.

R: Binary Search Tree.

Which ONE of the following options gives the worst-case time complexities for meld operation on instances of size  $n$  of these data structures? **(Gate 2025)**

A) P:  $\Theta(1)$ , Q:  $\Theta(n)$ , R:  $\Theta(n)$

B) P:  $\Theta(1)$ , Q:  $\Theta(n \log n)$ , R:  $\Theta(n)$

C) P:  $\Theta(n)$ , Q:  $\Theta(n \log n)$ , R:  $\Theta(n^2)$

D) P:  $\Theta(1)$ , Q:  $\Theta(n)$ , R:  $\Theta(n \log n)$

For parameters  $a$  and  $b$ , both of which are  $\omega(1)$ ,  $T(n) = T(n^{1/a}) + 1$ , and  $T(b) = 1$ .

Then  $T(n)$  is

**(Gate-2020) (1 Marks)**

$\Theta(\log_a \log_b n)$

$\Theta(\log_{ab} n)$

$\Theta(\log_b \log_a n)$

$\Theta(\log_2 \log_2 n)$

**Q** Consider the following recurrence:

$$f(1) = 1;$$

$$f(2n) = 2f(n)-1, \text{ for } n \geq 1;$$

$$f(2n+1) = 2f(n)+1, \text{ for } n \geq 1.$$

Then, which of the following statements is/are TRUE? **(GATE 2022) (2 MARKS)**

(A)  $f(2^n - 1) = 2^n - 1$

(B)  $f(2^n) = 1$

(C)  $f(5 \cdot 2^n) = 2^{n+1} + 1$

(D)  $f(2^n + 1) = 2^n + 1$

**Q** Consider the following recurrence relation.

$$T(n) = \begin{cases} T(n/2) + T(2n/5) + 7n & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$$

Which one of the following options is correct? **(GATE 2021) (2 MARKS)**

(a)  $T(n) = \Theta(n^{5/2})$

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

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

(d)  $T(n) = \Theta((\log n)^{5/2})$

**Q** Consider the following C function.

```
int fun(int n)
{
    int i, j;
    for (i = 1; i <= n ; i++)
    {
        for (j = 1; j < n; j += i)
        {
            printf("%d %d", i, j);
        }
    }
}
```

Time complexity of fun in terms of  $\Theta$  notation is: **(Gate-2017) (2 Marks)**

- (A)  $\Theta(n \sqrt{n})$
- (B)  $\Theta(n^2)$
- (C)  $\Theta(n \log n)$
- (D)  $\Theta(n^2 \log n)$

**Q** Consider the following function:

```
int unknown(int n){  
    int i, j, k=0;  
    for (i=n/2; i<=n; i++)  
        for (j=2; j<=n; j=j*2)  
            k = k + n/2;  
    return (k);  
}
```

The return value of the function is (Gate-2013) (2 Marks)

- a)  $\Theta(n^2)$
- b)  $\Theta(n^2 \log n)$
- c)  $\Theta(n^3)$
- d)  $\Theta(n^3 \log n)$