

Time complexities

Rabin Karp is  $O(n \cdot m + 1)$

And worst case is  $O(nm)$

Generating permutation :  $O(n \cdot n!)$

Hamiltonian cycle :  $O(n^n)$

Graph colouring:  $O(n \cdot m^n)$

N-queen :  $O(n!)$

Randomised quick sort:  $O(n \log n)$

Floyd-Warshall Algorithm :  $O(V^3)$

Subset Sum :  $O(2^N)$

Travelling Salesman Problem:  $O(n^2 \cdot 2^n)$

BFS and DFS time complexity:  $O(V+E)$

Matrix chain multiplication:  $\Theta(n^3)$

0/1 knapsack:  $O(2^n)$

Optimal BST:  $O(n^3)$

Longest common subsequence:

Worst case:  $O(n \cdot 2^m)$

Best case:  $O(nm)$

Kruskal's:  $O(E \log V)$

Prims:  $O(E \log V)$  but can be improved to  $O(E + \log V)$

Comparison of various Time Complexities

$O(c) < O(\log \log n) < O(\log n) < O(n^{1/2}) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(n^k) < O(2^n) < O(n^n) < O(2^{2^n})$

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, RAMAPURAM CAMPUS**  
**COMPUTER SCIENCE AND ENGINEERING**  
**QUESTION BANK**  
**18CSC204J DESIGN AND ANALYSIS OF ALGORITHMS**

**UNIT 1**

**Introduction-Algorithm Design, Fundamentals of Algorithms, Correctness of algorithm, Time complexity analysis, Insertion sort-Line count, Operation count, Algorithm Design paradigms, Designing an algorithm, And its analysis-Best, Worst and Average case, Asymptotic notations Based on growth functions. O,O, $\Theta$ ,  $\omega$ ,  $\Omega$  Mathematical analysis, Induction, Recurrence relations , Solution of recurrence relations, Substitution method, Solution of recurrence relations, Recursion tree, Solution of recurrence relations, Examples**

**PART A**

**1. \_\_\_\_\_ is the first step in solving the problem**

- A. Understanding the Problem
- B. Identify the Problem
- C. Evaluate the Solution
- D. Coding the Problem

Answer: - B

**2. While solving the problem with computer the most difficult step is \_\_\_\_\_.**

- A. describing the problem
- B. finding out the cost of the software
- C. writing the computer instructions
- D. testing the solution

Answer:- C

**3. \_\_\_\_\_ solution requires reasoning built on knowledge and experience**

- A. Algorithmic Solution
- B. Heuristic Solution
- C. Random Solution
- D. Brute force Solution

Answer: - B

**4. The correctness and appropriateness of \_\_\_\_\_ solution can be checked very easily.**

- A. algorithmic solution
- B. heuristic solution
- C. random solution
- D. Brute force Solution

Answer:- A

**5. When determining the efficiency of algorithm, the space factor is measured by**

- A. Counting the maximum memory needed by the algorithm
- B. Counting the minimum memory needed by the algorithm
- C. Counting the average memory needed by the algorithm
- D. Counting the maximum disk space needed by the algorithm

Answer: - A

**6. The elements of an array are stored successively in memory cells because**

- A. by this way computer can keep track only the address of the first element and the addresses of other elements can be calculated
- B. the architecture of computer memory does not allow arrays to store other than serially
- C. Either A or B
- D. Both A and B

Answer: - A

**7. The hierarchy of operations is denoted as \_\_\_\_\_.**

I. +, -      II. Power      III. \*, /      IV. \, MOD

- A. I, II, III, IV
- B. II, IV, III, I
- C. IV, I, III, II
- D. II, III, IV, I

Answer:- B

**8. What is the time complexity of following code:**

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

- A. O(N)
- B. O(Sqrt(N))
- C. O(N / 2)
- D. O(log N)

Answer: - D

**9. Two main measures for the efficiency of an algorithm are**

- A. Processor and memory
- B. Complexity and capacity
- C. Time and space
- D. Data and space

Answer: - C

**10. What does the algorithmic analysis count?**

- A. The number of arithmetic and the operations that are required to run the program
- B. The number of lines required by the program
- C. The number of seconds required by the program to execute
- D. None of these

Answer:- A

**11. An algorithm that indicates the amount of temporary storage required for running the algorithm, i.e., the amount of memory needed by the algorithm to run to completion is termed as\_\_\_\_\_.**

- A. Big Theta  $\Theta(f)$
- B. Space complexity

- C. Big Oh  $O(f)$
- D. Time Complexity

Answer B

**12. Consider a linked list of n elements. What is the time taken to insert an element after an element pointed by some pointer?**

- A. (1)
- B. (n)
- C.  $(\log_2 n)$
- D.  $(n \log_2 n)$

Answer A

**13. If the address of  $A[1][1]$  and  $A[2][1]$  are 1000 and 1010 respectively and each element occupies 2 bytes then the array has been stored in order.**

- A. row major
- B. column major
- C. matrix major
- D. none of these

Answer A

**14. The time factor when determining the efficiency of algorithm is measured by**

- A. Counting microseconds
- B. Counting the number of key operations
- C. Counting the number of statements
- D. Counting the kilobytes of algorithm

Answer B

**15. Time complexities of three algorithms are given. Which should execute the slowest for large values of N?**

- A.  $(n \log n)$
- B.  $O(n)$
- C.  $O(\log n)$
- D.  $O(n^2)$

Answer B

**16. Which one of the following is the tightest upper bound that represents the number of swaps required to sort n number using selection sort?**

- A.  $(\log n)$
- B.  $O(n)$
- C.  $(n \log n)$
- D.  $O(n^2)$

Answer B

**17. How many comparisons are needed for linear Search array when elements are in order in best case?**

- A. 1
- B. n
- C.  $n+1$
- D.  $n-1$

Answer A

**18. The complexity of Bubble sort algorithm is\_\_\_\_\_**

- A.  $O(n)$
- B.  $O(\log n)$
- C.  $O(n^2)$
- D.  $O(n \log n)$

Answer : C

**19. What is the time complexity of following code:**

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

- A.  $O(N)$
- B.  $O(\text{Sqrt}(N))$
- C.  $O(N / 2)$
- D.  $O(\log N)$

Answer D

**20. Which of the given options provides the increasing order of asymptotic complexity of functions f1, f2, f3 and f4?**

**f1(n) = 2^n**

**f2(n) = n^(3/2)**

- A. f3, f2, f1, f4
- B. f2, f3, f1, f4
- C. f2, f3, f4, f1
- D. f3, f2, f4, f1

Answer is: D

**f3(n) = nLogn**

**f4(n) = n^(Logn)**

**21. How much number of comparisons is required in insertion sort to sort a file if the file is sorted in reverse order?**

- A. N^2
- B. N
- C. N-1
- D. N/2

Answer A

**22. The worst-case occur in linear search algorithm when .....**

- A. Item is somewhere in the middle of the array
- B. Item is not in the array at all
- C. Item is the last element in the array
- D. Item is the last element in the array or item is not there at all

Answer D

**23. What is the time complexity of fun()?**

```
int fun(int n)
{
    int count = 0;
    for (int i = 0; i < n; i++)
        for (int j = i; j > 0; j--)
            count = count + 1;
    return count;
}
```

- A. Theta (n)
- B. Theta (n^2)
- C. Theta (n\*Logn)
- D. Theta (nLognLogn)

Answer : B

**24. The time complexity of the following C function is (assume n > 0 )**

```
(int recursive (mt n)
{
    if (n == 1)
        return (1);
    else
        return (recursive (n-1) + recursive (n-1));
```

}

- A.  $O(n)$
- B.  $O(n \log n)$
- C.  $O(n^2)$
- D.  $O(2^n)$

Answer D

25. **A function in which  $f(n) = \Omega(g(n))$ , if there exist positive values k and c such that  $f(n) \geq c*g(n)$ , for all  $n \geq k$ . This notation defines a lower bound for a function f(n):**

- A. Big Omega  $\Omega(f)$
- B. Big Theta  $\Theta(f)$
- C. Big Oh  $O(f)$
- D. Big Alpha  $\alpha(f)$

Answer A

26. **The concept of order Big O is important because \_\_\_\_\_**

- A. It can be used to decide the best algorithm that solves a given problem
- B. It determines the maximum size of a problem that can be solved in a given amount of time
- C. It is the lower bound of the growth rate of algorithm
- D. Both A and B

Answer A

27. **The upper bound on the time complexity of the nondeterministic sorting algorithm is**

- A.  $O(n)$
- B.  $O(n \log n)$
- C.  $O(1)$
- D.  $O(\log n)$

Answer: A

28. **In the analysis of algorithms, what plays an important role?**

- A. Text Analysis
- B. Growth factor
- C. Time
- D. Space

Answer: B

29. **Which one of the following correctly determines the solution of the recurrence relation given below with  $T(1) = 1$  and  $T(n) = 2T(n/4) + n^{1/2}$**

- A.  $O(n^2)$
- B.  $O(n)$
- C.  $O(n^{1/2} \log n)$
- D.  $O(\log n)$

Answer C

30. **What is the time complexity of recursive function given below:**

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

- |                       |                            |
|-----------------------|----------------------------|
| A. O(n <sup>2</sup> ) | C. O(n <sup>2</sup> log n) |
| B. O(n)               | D. O(n log n)              |

Answer C

## PART B

- 1 . What is an Algorithm?
- 2 . Give the notion of an algorithm.
- 3 . Design an algorithm for computing gcd(m,n) using Euclid's algorithm.
- 4 . Design an algorithm to compute the area and circumference of a circle.
- 5 . Differentiate Sequential and Parallel Algorithms.
- 6 . Write the process for design and analysis of algorithm.
- 7 . What are the fundamentals steps for design and analysis of an algorithm?
- 8 . Compare Exact and Approximation algorithm.
- 9 . What is an Algorithm Design Technique?
- 10 . Define Pseudo code.
- 11 . Define Flowchart.
- 12 . Prove the correctness of an algorithm's.
- 13 . Define algorithm validation.
- 14 . What is validation and program verification?
- 15 . Define program proving and program verification.
- 16 . Write the characteristics of an algorithm.
- 17 . What is the Efficiency of algorithm?
- 18 . What is time and space complexity?
- 19 . What is generality of an algorithm?
- 20 . What is algorithm's Optimality?
- 21 . Write an algorithm to find the number of binary digits in the binary representation of a positive decimal integer.
  
- 22 . What are the types of problems in algorithm?
- 23 . How will you measure input size of algorithms?
- 24 . What is the average case complexity of linear search algorithm?
- 25 . Differentiate searching and sorting algorithm.
- 26 . What are combinatorial problems?
- 27 . Define a graph and its type.
- 28 . Define performance analysis.
- 29 . What do you mean by Worst case-Efficiency of an algorithm?
- 30 . What do you mean by Best case-Efficiency of an algorithm?
- 31 . Define the Average-case efficiency of an algorithm.
- 32 . What do you mean by Amortized efficiency?
- 33 . How to analyze an algorithm framework?

## UNIT II: DIVIDE AND CONQUER

Introduction-Divide and Conquer, Maximum Sub array Problem
Binary Search, Complexity of binary search
Merge sort, Time complexity analysis
Quick sort and its Time complexity analysis, Best case, Worst case, Average case analysis
Strassen's Matrix multiplication and its recurrence relation, Time complexity analysis of Merge sort
Largest sub-array sum, Time complexity analysis of Largest sub-array sum
Master Theorem Proof, Master theorem examples
Finding Maximum and Minimum in an array, Time complexity analysis-Examples
Algorithm for finding closest pair problem, Convex Hull problem

### PART-A

1.) Partition and exchange sort is\_\_\_\_\_

- A. quick sort
- B. tree sort
- C. heap sort
- D. bubble sort

**ANSWER: A**

2) Which of the following is not the required condition for binary search algorithm?

- A. The list must be sorted
- B. There should be the direct access to the middle element in any sub list
- C. There must be mechanism to delete and/or insert elements in list.
- D. Number values should only be present

**ANSWER: C**

3) Which of the following sorting algorithm is of divide and conquer type?

- A. Bubble sort
- B. Insertion sort
- C. Merge sort
- D. Selection sort

**ANSWER: C**

4) \_\_\_\_\_ order is the best possible for array sorting algorithm which sorts n item.

- A.  $O(n \log n)$
- B.  $O(n^2)$
- C.  $O(n + \log n)$
- D.  $O(\log n)$

**ANSWER: C**

5) The complexity of merge sort algorithm is \_\_\_\_\_

- A.  $O(n)$
- B.  $O(\log n)$

- C.  $O(n^2)$
  - D.  $O(n \log n)$
- ANSWER: D**

- 6) Binary search algorithm cannot be applied to \_\_\_\_\_
- A. sorted linked list
  - B. sorted binary trees
  - C. sorted linear array
  - D. pointer array

**ANSWER: A**

- 7) Which of the following is not a limitation of binary search algorithm?
- A. must use a sorted array
  - B. requirement of sorted array is expensive when a lot of insertion and deletions are needed
  - C. there must be a mechanism to access middle element directly
  - D. binary search algorithm is not efficient when the data elements more than 1500.

**ANSWER: D**

- 8) Which of the following is an external sorting?
- A. Insertion Sort
  - B. Bubble Sort
  - C. Merge Sort
  - D. Tree Sort

**ANSWER: B**

- 9 ) Merging k sorted tables into a single sorted table is called \_\_\_\_\_
- A. k way merging
  - B. k th merge
  - C. k+1 merge
  - D. k-1 merge

**ANSWER: A**

- 10) The operation that combines the element is of A and B in a single sorted list C with  $n=r+s$  element is called \_\_\_\_\_
- A. Inserting
  - B. Mixing
  - C. Merging
  - D. Sharing

**ANSWER: C**

- 11) Which of the following is a stable sorting algorithm?
- a) Merge sort
  - b) typical in-place quick sort
  - c) Heap sort
  - d) Selection sort

**ANSWER: A**

12) Which of the following is not an in-place sorting algorithm?

- a) Selection sort
- b) Heap sort
- c) Quick sort
- d) Merge sort

**ANSWER: D**

13 )The time complexity of a quick sort algorithm which makes use of median, found by an  $O(n)$  algorithm, as pivot element is

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

**ANSWER: B**

14) Which of the following algorithm design technique is used in the quick sort algorithm?

- a) Dynamic programming
- b) Backtracking
- c) Divide-and-conquer
- d) Greedy method

**ANSWER: C**

15) Merge sort uses

- a) Divide-and-conquer
- b) Backtracking
- c) Heuristic approach
- d) Greedy approach

**ANSWER: A**

16 )For merging two sorted lists of size  $m$  and  $n$  into sorted list of size  $m+n$ , we require comparisons of

- a)  $O(m)$
- b)  $O(n)$
- c)  $O(m+n)$
- d)  $O(\log m + \log n)$

**ANSWER: C**

17) The running time of Strassen's algorithm for matrix multiplication is

- (A)  $\Theta(n)$
- (B)  $\Theta(n^3)$
- (C)  $\Theta(n^2)$
- (D)  $\Theta(n^{2.81})$

**ANSWER: D**

18) The Stassen's algorithm's time complexity is

- (A)  $O(n)$
- (B)  $O(n^2)$
- (C)  $O(n^{2.80})$
- (D)  $O(n^{2.81})$

**ANSWER: C**

19) Which algorithm is used for matrix multiplication?

- a. Simple algorithm
- b. Specific algorithm
- c. Strassen algorithm
- d. Addition algorithm

**ANSWER: C**

20) Which algorithm is a divided and conquer algorithm that is asymptotically faster:

- a. Simple algorithm
- b. Specific algorithm
- c. Strassen algorithm
- d. Addition algorithm

**ANSWER: C**

21) Which algorithm is named after Volker Strassen

- a. Strassen algorithm
- b. Matrix algorithm
- c. Both
- d. None of these

**ANSWER: A**

22) Which of the following algorithms is NOT a divide & conquer algorithm by nature?

- (A) Euclidean algorithm to compute the greatest common divisor
- (B) Heap Sort
- (C) Closest pair problem
- (D) Quick Sort

**Answer: B**

23). what is the average case time complexity of merge sort?

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

**ANSWER: A**

24). which of the following method is used for sorting in merge sort?

- a) Merging
- b) Partitioning
- c) Selection
- d) Exchanging

**ANSWER: A**

25) Which of the following is not a stable sorting algorithm?

- a) Quick sort

- b) Cocktail sort
- c) Bubble sort
- d) Merge sort

**ANSWER: A**

26) What is the runtime efficiency of using brute force technique for the closest pair problem?

- a)  $O(N)$
- b)  $O(N \log N)$
- c)  $O(N^2)$
- d)  $O(N^3 \log N)$

**ANSWER: C**

27) What is the basic operation of closest pair algorithm using brute force technique?

- a) Euclidean distance
- b) Radius
- c) Area
- d) Manhattan distance

**ANSWER: A**

28) \_\_\_\_\_ is a method of constructing a smallest polygon out of n given points.

- a) Closest pair problem
- b) Quick hull problem
- c) Path compression
- d) union-by-rank

**ANSWER: B**

29) Find the maximum sub-array sum for the given elements.

{-2, -1, -3, -4, -1, -2, -1, -5, -4}

- a) -3
- b) 5
- c) 3
- d) -1

**ANSWER: D**

30) Master's theorem is used for?

- a) Solving recurrences
- b) Solving iterative relations
- c) Analyzing loops
- d) Calculating the time complexity of any code

**ANSWER: A**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**  
**RAMAPURAM CAMPUS**  
**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**  
**QUESTION BANK**  
**18CSC204J DESIGN AND ANALYSIS OF ALGORITHM (REGULATION 2018)**

**UNIT III**

**Introduction-Greedy and Dynamic Programming, Examples of problems that can be solved by using greedy and dynamic approach Huffman coding using greedy approach, Comparison of brute force and Huffman method of encoding Knapsack problem using greedy approach, Complexity derivation of knapsack using greedy Tree traversals, Minimum spanning tree - greedy, Kruskal's algorithm - greedy Minimum spanning tree - Prim's algorithm, Introduction to dynamic programming 0/1 knapsack problem, Complexity calculation of knapsack problem Matrix chain multiplication using dynamic programming, Complexity of matrix chain multiplication Longest common subsequence using dynamic programming, Explanation of LCS with an example Optimal binary search tree (OBST)using dynamic programming, Explanation of OBST with an example**

**PART A**

1. ----- is a Boolean-valued function that determines whether  $x$  can be included into the solution vector
  - a) Overlapping subproblems
  - b) **Fleasible solution**
  - c) Memoization
  - d) Greedy
2. Trees with edge with weights are called -----
  - a) **weighted tree**
  - b) unweighted tree
  - c) bruteforce
  - d) Greedy
3. ----- is to determine an optimal placement of booster
  - a) Weighted tree
  - b) Vertex
  - c) **Tree Vertex Splitting Problem (TVSP)**
  - d) Greedy
4. The order in which TVS visits that computes the delay values of the nodes of the tree is called the-----.
  - a)treeorder
  - b) inorder
  - c) preorder

- d) postorder**
5. Algorithm TVS takes ----- time, where n is the number f nodes in the tree
- O(N)**
  - $\Omega(n \log n)$
  - $O(n^2 \log n)$
  - $O(n \log n)$
6. ----- is a greedy method to obtain a minimum-cost spanning tree builds this tree edge by edge
- Prism's algorithm**
  - Dynamic algorithm
  - Greedy algorithm
  - Dynamic algorithm
7. Kruskal's algorithm (choose best non-cycle edge) is better than Prim's (choose best tree edge) when the graph has relatively few edges
- True**
  - False
8. Two sorted files containing n and m records respectively could be merged together to obtain one sorted file in time -----.
- $\Omega(n \log n)$
  - O(n+m)**
  - $O(n^2 \log n)$
  - $O(n \log n)$
9. The two-way merge pattern scan be represented by-----
- Weighted tree
  - Vertex
  - binary merge tree**
  - Greedy
10. What algorithm technique is used in the implementation of Kruskal solution for the MST?
- greedy technique**
  - divide-and-conquer technique
  - dynamic programming technique
  - the algorithm combines more than one of the above techniques
11. The function Tree of Algorithm uses the ----- stated to obtain a two-way merge tree for n file
- divide-and-conquer technique
  - greedy rule**
  - dynamic programming technique
  - the algorithm combines more than one of the above techniques
12. A decode tree is a----- in which external nodes represent messages.
- minimum spanning tree
  - B tree
  - binary tree**

- d) AVL tree
13. The -----in the code word for a message determine the branching needed at each level of the decode tree to reach the correct external node.
- a) **binary bits**
  - b) decoder
  - c) encoder
  - d) binary bytes
14. The cost of decoding a -----is proportional to the number of bits in the code
- a) binary bits
  - b) code word**
  - c) data
  - d) binary bytes
15. What is the edges on the shortest paths from a vertex v to all remaining vertices in a connected undirected graph G form a spanning tree of G is called?
- a) MST
  - b) shortest-path spanning tree**
  - c) binary tree
  - d) AVL tree
16. -----is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.
- a) Dynamic Programming**
  - b) Greedy method
  - c) Huffman coding
  - d) Tree traversal
17. What is another important feature of the dynamic programming approach that optimal solutions are retained so as to avoid recomputing their values.
- a) Dynamic Programming**
  - b) Greedy method
  - c) Huffman coding
  - d) Tree traversal
18. ----- often drastically reduces the amount of enumeration by avoiding the enumeration of some decision sequences that cannot possibly be optimal.
- a) Dynamic Programming**
  - b) Greedy method
  - c) Huffman coding
  - d) Tree traversal
19. In the -----only one decision sequence is ever generated.
- a) Dynamic Programming
  - b) Greedy method**
  - c) Huffman coding
  - d) Tree traversal
20. Dynamic programming algorithms solve the----- to obtain a solution to the given problem instance

- a) optimistic
  - b) Greedy method
  - c) Huffman coding
  - d) recurrence**
21. A dynamic programming formulation for a k-stage graph problem is obtained by first noticing that every s to t path is the result of a sequence of ----- decision.
- a) k
  - b) k-1
  - c) k-2**
  - d) 2k
22. Which of the following problems is NOT solved using dynamic programming?
- a) 0/1 knapsack problem
  - b) Matrix chain multiplication problem
  - c) Edit distance problem
  - d) Fractional knapsack problem**
23. The problem of -----is to identify a minimum-cost sequence of edit operations that will transform X into Y.
- a) 0/1 knapsack problem
  - b) Matrix chain multiplication problem
  - c) Edit distance problem
  - d) string editing**
24. In Knapsack problem, the best strategy to get the optimal solution, where  $P_i$ ,  $W_i$  is the Profit, Weight associated with each of the  $X_i^{\text{th}}$  object respectively is to
- a)Arrange the values  $P_i/W_i$  in ascending order
  - b)Arrange the values  $P_i/X_i$  in ascending order
  - c)Arrange the values  $P_i/W_i$  in descending order
  - d)Arrange the values  $P_i/X_i$  in descending order**
25. Greedy job scheduling with deadlines algorithms' complexity is defined as
- a) $O(N)$
  - b) $\Omega(n \log n)$**
  - c) $O(n^2 \log n)$
  - d) $O(n \log n)$
26. In Huffman coding, data in a tree always occur?
- a) roots
  - b) leaves**
  - c) left sub trees
  - d) right sub trees
27. The multistage graph problem can also be solved using the -----
- a) backward approach

- b) forward approach**  
 c) brute force approach  
 d) right sub trees
28. The all-pairs -----problem is to determine a matrix A such that  $A(i,j)$  is the length of a shortest path from i to j.  
 a) backward approach  
 b) forward approach  
 c) brute force approach  
**d) shortest-path**
29. Which of the following methods can be used to solve the Knapsack problem?  
 a) Brute force algorithm  
 b) Recursion  
 c) Dynamic programming  
**d) Brute force, Recursion and Dynamic Programming**
30. The inorder and preorder traversal of a binary tree are d b e a f c g and a b d e c f g, respectively. The postorder traversal of the binary tree is:  
 a) **d e b f g c a**  
 b) e d b g f c a  
 c) e d b f g c a  
 d) d e f g b c a
31. Which of the following pairs of traversals is not sufficient to build a binary tree from the given traversals?  
 a) Preorder and Inorder  
 b) Preorder and Postorder  
 c) Inorder and Postorder  
**d) Inorder and levelorder**
32. Consider the following C program segment
- ```

struct CellNode
{
  struct CelINode *leftchild;
  int element;
  struct CelINode *rightChild;
}

int Dosomething(struct CelINode *ptr)
{
  int value = 0;
  if (ptr != NULL)
  {
    if (ptr->leftChild != NULL)
      value = 1 + DoSomething(ptr->leftChild);
    if (ptr->rightChild != NULL)
      value = max(value, 1 + DoSomething(ptr->rightChild));
  }
}
  
```

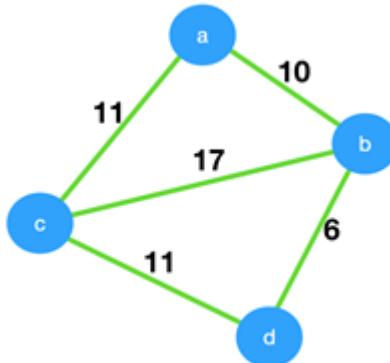
```

    }
    return (value);
}

```

The value returned by the function DoSomething when a pointer to the root of a non-empty tree is passed as argument is

- a) The number of leaf nodes in the tree
  - b) The number of nodes in the tree
  - c) The number of internal nodes in the tree
  - d) The height of the tree**
33. Given items as {value, weight} pairs  $\{\{60, 20\}, \{50, 25\}, \{20, 5\}\}$ . The capacity of knapsack=40. Find the maximum value output assuming items to be divisible and nondivisible respectively.
- a) 100,80
  - b) 110,70
  - c) 130,110
  - d) 110,80**
34. Given items as {value, weight} pairs  $\{\{40, 20\}, \{30, 10\}, \{20, 5\}\}$ . The capacity of knapsack=20. Find the maximum value output assuming items to be divisible
- a) 60**
  - b) 80
  - c) 100
  - d) 40
35. Consider the given graph.

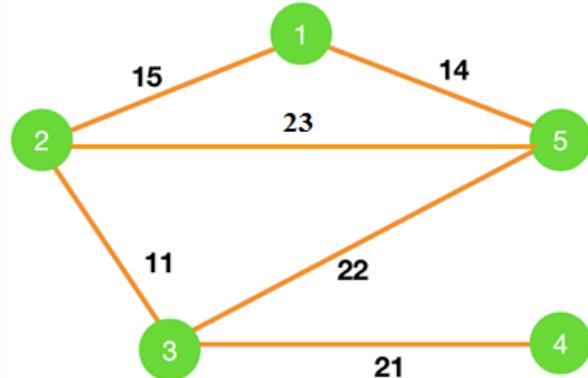


What is the weight of the minimum spanning tree using the Prim's algorithm, starting from vertex a?

- a) 23
  - b) 28
  - c) 27**
  - d) 11
36. Worst case is the worst case time complexity of Prim's algorithm if adjacency matrix is used?
- a)  $O(\log V)$

- b)  $O(V^2)$
- c)  $O(E^2)$
- d)  $O(V \log E)$

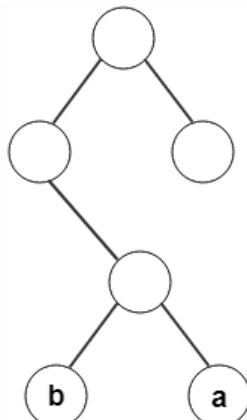
37. Consider the graph shown below.



Which of the following edges form the MST of the given graph using Prim's algorithm, starting from vertex 4.

- a) (4-3)(5-3)(2-3)(1-2)
- b) (4-3)(3-5)(5-1)(1-2)
- c) (4-3)(3-5)(5-2)(1-5)
- d) (4-3)(3-2)(2-1)(1-5)**

38. From the following given tree, what is the code word for the character 'a'?



- a) 011**
- b) 010
- c) 100
- d) 101

39. What will be the cost of the code if character  $c_i$  is at depth  $d_i$  and occurs at frequency  $f_i$ ?

- a)  $c_i f_i$
- b)  $\sum c_i f_i$
- c)  $\sum f_i d_i$**

d)  $f_i d_i$

40. What is the running time of the Huffman encoding algorithm?

- a)  $O(C)$
- b)  $O(\log C)$
- c)  **$O(C \log C)$**
- d)  $O(N \log C)$

41. The weighted array used in TVS problems for the following binary tree is \_\_\_\_\_

- a) [1,2,3,0,0,4,0,5,6]
- b) **[1,2,3,0,0,4,0,5,0,0,0,6]**
- c) [1,2,3,4,5,6]
- d) [1,2,3,0,0,4,5,6]

42. What is the time complexity of the brute force algorithm used to find the longest common subsequence?

- a)  $O(n)$
- b)  $O(n^2)$
- c)  $O(n^3)$
- d)  **$O(2^n)$**

43. Find the longest increasing subsequence for the given sequence:

{10, -10, 12, 9, 10, 15, 13, 14}

- a) {10, 12, 15}
- b) {10, 12, 13, 14}
- c) {-10, 12, 13, 14}
- d) **{-10, 9, 10, 13, 14}**

44. What is the space complexity of the following dynamic programming implementation used to find the length of the longest increasing subsequence?

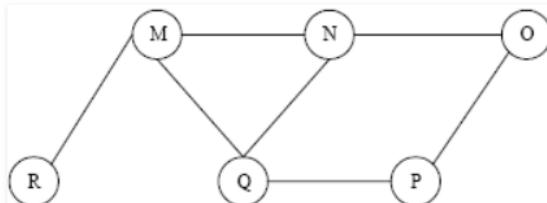
```
#include<stdio.h>
int longest_inc_sub(int *arr, int len)
{
    int i, j, tmp_max;
    int LIS[len]; // array to store the lengths of the longest increasing subsequence
    LIS[0]=1;
    for(i = 1; i < len; i++)
    {
        tmp_max = 0;
        for(j = 0; j < i; j++)
        {
            if(arr[j] < arr[i])
            {
                if(LIS[j] > tmp_max)
                    tmp_max = LIS[j];
            }
        }
        LIS[i] = tmp_max + 1;
    }
}
```

```

        }
    }
    LIS[i] = tmp_max + 1;
}
int max = LIS[0];
for(i = 0; i < len; i++)
    if(LIS[i] > max)
        max = LIS[i];
return max;
}
int main()
{
    int arr[] = {10,22,9,33,21,50,41,60,80}, len = 9;
    int ans = longest_inc_sub(arr, len);
    printf("%d",ans);
    return 0;
}

```

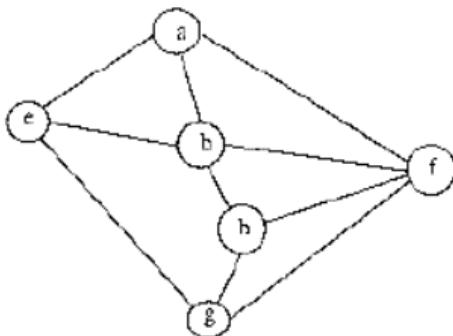
- a) O(1)  
**b) O(n)**  
c) O(n2)  
d) O(nlogn)
45. The Breadth First Search algorithm has been implemented using the queue data structure. One possible order of visiting the nodes of the following graph is
- a) O(1)  
**b) O(n)**  
c) O(n2)  
d) O(nlogn)
46. Uniform-cost search expands the node n with the \_\_\_\_\_
- a) Lowest path cost**  
b) Heuristic cost  
c) Highest path cost  
d) Average path cost
47. The Breadth First Search algorithm has been implemented using the queue data structure. One possible order of visiting the nodes of the following graph is



- a) MNOPQR  
b) NQMPOR  
**c) QMNPOR**

d) QMNPOR

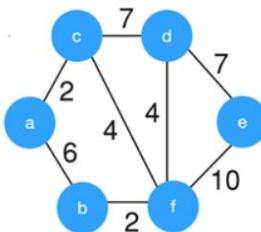
48. Consider the following graph,



Which are depth first traversals of the above graph?

- a) I,II and IV
- b) I and IV only
- c) II,III and IV only
- d) I,III and IV only

49. Consider the given graph.



What is the weight of the minimum spanning tree using the Kruskal's algorithm?

- a) 24
- b) 23
- c) 15
- d) 19

50. Which of the following is false about the Kruskal's algorithm?

- a) It is a greedy algorithm
- b) It constructs MST by selecting edges in increasing order of their weights
- c) **It can accept cycles in the MST**
- d) It uses union-find data structure

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, RAMAPURAM**

**CAMPUS COMPUTER SCIENCE AND ENGINEERING**

**QUESTION BANK**

**18CSC204J DESIGN AND ANALYSIS OF ALGORITHMS**

**UNIT 4**

**Introduction to backtracking - branch and bound, N queen's problem – backtracking, Sum of subsets using backtracking, Complexity calculation of sum of subsets, Graph introduction, Hamiltonian circuit – backtracking, Branch and bound - Knapsack problem, Example and complexity calculation. Differentiate with dynamic and greedy, Travelling salesman problem using branch and bound, Travelling salesman problem using branch and bound example, Travelling salesman problem using branch and bound example, Time complexity calculation with an example, Graph algorithms, Depth first search and Breadth first search, Shortest path introduction, Floyd-Warshall Introduction, Floyd-Warshall with sample graph, Floyd-Warshall complexity**

**PART A**

**1. Which of the following is not a backtracking algorithm?**

- (A) Knight tour problem
- (B) N queen problem
- (C) Tower of hanoi
- (D) M coloring problem

Answer: - C

**2. Backtracking algorithm is implemented by constructing a tree of choices called as?**

- A) State-space tree
- B) State-chart tree
- C) Node tree
- D) Backtracking tree

Answer: - A

**3. What happens when the backtracking algorithm reaches a complete solution?**

- A) It backtracks to the root

- B) It continues searching for other possible solutions
- C) It traverses from a different route
- D) Recursively traverses through the same route

Answer: - B

**4. In what manner is a state-space tree for a backtracking algorithm constructed?**

- A) Depth-first search
- B) Breadth-first search
- C) Twice around the tree
- D) Nearest neighbour first

Answer: - A

**5. In general, backtracking can be used to solve?**

- A) Numerical problems
- B) Exhaustive search
- C) Combinatorial problems
- D) Graph coloring problems

Answer: - C

**6. Which one of the following is an application of the backtracking algorithm?**

- A) Finding the shortest path
- B) Finding the efficient quantity to shop
- C) Ludo
- D) Crossword

Answer: - D

**7. Who coined the term ‘backtracking’?**

- A) Lehmer
- B) Donald
- C) Ross
- D) Ford

Answer: - A

**8. The problem of finding a subset of positive integers whose sum is equal to a given positive integer is called as?**

- A) n- queen problem
- B) subset sum problem
- C) knapsack problem
- D) hamiltonian circuit problem

Answer: - B

**9. The problem of placing n queens in a chessboard such that no two queens attack each other is called as?**

- A) n-queen problem
- B) eight queens puzzle
- C) four queens puzzle

D) 1-queen problem

Answer: - A

**10. In how many directions do queens attack each other?**

A) 1

B) 2

C) 3

D) 4

Answer: - C

**11. Placing n-queens so that no two queens attack each other is called?**

A) n-queen's problem

B) 8-queen's problem

C) Hamiltonian circuit problem

D) subset sum problem

Answer: - A

**12. Where is the n-queens problem implemented?**

A) carom

B) chess

C) ludo

D) cards

Answer: - B

**13. Not more than 2 queens can occur in an n-queens problem.**

A) true

B) false

Answer: - B

**14. In n-queen problem, how many values of n does not provide an optimal solution? A) 1**

B) 2

C) 3

D) 4

Answer: - B

**15. Which of the following methods can be used to solve n-queen's problem? A) greedy algorithm**

B) divide and conquer

C) iterative improvement

D) backtracking

Answer: - D

**16. Of the following given options, which one of the following is a correct option that provides an optimal solution for 4-queens problem?**

- A) (3,1,4,2)
- B) (2,3,1,4)
- C) (4,3,2,1)
- D) (4,2,3,1).

Answer: - A

**17. How many possible solutions exist for an 8-queen problem?**

- A) 100
- B) 98
- C) 92
- D) 88

Answer: - C

**18. How many possible solutions occur for a 10-queen problem?**

- A) 850
- B) 742
- C) 842
- D) 724.

Answer: - D

**19. The Knapsack problem is an example of \_\_\_\_\_**

- A) Greedy algorithm
- B) 2D dynamic programming
- C) 1D dynamic programming
- D) Divide and conquer

Answer: - B

**20. Which of the following methods can be used to solve the Knapsack problem? A) Brute force algorithm**

- B) Recursion
- C) Dynamic programming
- D) Brute force, Recursion and Dynamic Programming

Answer: - D

**21. You are given a knapsack that can carry a maximum weight of 60. There are 4 items with weights {20, 30, 40, 70} and values {70, 80, 90, 200}. What is the maximum value of the items you can carry using the knapsack?**

- A) 160
- B) 200
- C) 170
- D) 90

Answer: - A

**22. Which of the following problems is equivalent to the 0-1 Knapsack problem? A)**

You are given a bag that can carry a maximum weight of W. You are given N items which have a weight of  $\{w_1, w_2, w_3, \dots, w_n\}$  and a value of  $\{v_1, v_2, v_3, \dots, v_n\}$ . You

can break the items into smaller pieces. Choose the items in such a way that you get the maximum value

B) You are studying for an exam and you have to study N questions. The questions take  $\{t_1, t_2, t_3, \dots, t_n\}$  time(in hours) and carry  $\{m_1, m_2, m_3, \dots, m_n\}$  marks. You can study for a maximum of T hours. You can either study a question or leave it. Choose the questions in such a way that your score is maximized

C) You are given infinite coins of denominations  $\{v_1, v_2, v_3, \dots, v_n\}$  and a sum S. You have to find the minimum number of coins required to get the sum S

D) You are given a suitcase that can carry a maximum weight of 15kg. You are given 4 items which have a weight of  $\{10, 20, 15, 40\}$  and a value of  $\{1, 2, 3, 4\}$ . You can break the items into smaller pieces. Choose the items in such a way that you get the maximum value

Answer: - B

**23. What is the time complexity of the brute force algorithm used to solve the Knapsack problem?**

- A)  $O(n)$
- B)  $O(n!)$
- C)  $O(2^n)$
- D)  $O(n^3)$

Answer: - C

**24. Which of the following is/are property/properties of a dynamic programming problem?**

- A) Optimal substructure
- B) Overlapping subproblems
- C) Greedy approach
- D) Both optimal substructure and overlapping subproblems

Answer: - D

**25. If an optimal solution can be created for a problem by constructing optimal solutions for its subproblems, the problem possesses \_\_\_\_\_ property.** A) Overlapping subproblems

- B) Optimal substructure
- C) Memoization
- D) Greedy

Answer: - B

**26. If a problem can be broken into subproblems which are reused several times, the problem possesses \_\_\_\_\_ property.**

- A) Overlapping subproblems
- B) Optimal substructure
- C) Memoization
- D) Greedy

Answer: - A

**27. If a problem can be solved by combining optimal solutions to non-overlapping problems, the strategy is called \_\_\_\_\_**

- A) Dynamic programming
  - B) Greedy
  - C) Divide and conquer
  - D) Recursion
- Answer: - C

**28. In dynamic programming, the technique of storing the previously calculated values is called \_\_\_\_\_**

- A) Saving value property
- B) Storing value property
- C) Memoization
- D) Mapping

Answer: - C

**29. When a top-down approach of dynamic programming is applied to a problem, it usually \_\_\_\_\_**

- A) Decreases both, the time complexity and the space complexity
- B) Decreases the time complexity and increases the space complexity
- C) Increases the time complexity and decreases the space complexity
- D) Increases both, the time complexity and the space complexity

Answer: - B

**30. Which of the following problems is NOT solved using dynamic programming?** A) 0/1 knapsack problem

- B) Matrix chain multiplication problem
- C) Edit distance problem
- D) Fractional knapsack problem

Answer: - D

**31. Which of the following problems should be solved using dynamic programming?** A) Mergesort

- B) Binary search
- C) Longest common subsequence
- D) Quicksort

Answer: - C

**32. Time Complexity of Breadth First Search is? (V - number of vertices, E - number of edges)**

- A)  $O(V+E)$
- B)  $O(V)$
- C)  $O(E)$
- D)  $O(VE)$

Answer: - A

**33. The spanning tree of connected graph with 10 vertices contains**

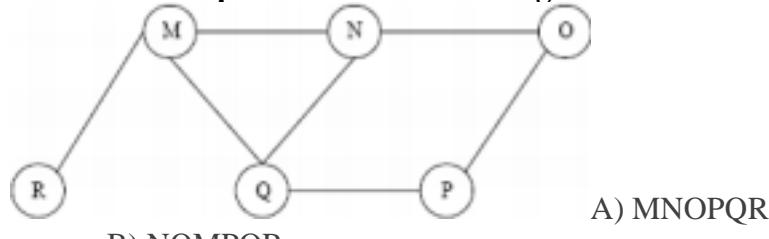
- ..... A) 9 edges
- B) 11 edges

C) 10 edges

D) 8 edges

Answer: - A

**34. The Breadth First Search algorithm has been implemented using the queue data structure. One possible order of visiting the nodes of the following graph is**



A) MNOPQR

B) NQMPOR

C) QMNPRO

D) QMNPOR

Answer: - C

**35. What is the maximum height of queue (To keep track of un-explored nodes) required to process a connected Graph G1 which contains 'N' node using BFS algorithm? A)  $(N/2)-1$**

B)  $(N/2)/2$

C) N-1

D) N

Answer: - C

## PART B

1. What is meant by knapsack problem?
2. Define fractional knapsack problem.
3. Write the running time of 0/1 knapsack problem.
4. Write recurrence relation for 0/1 knapsack problem
5. What is meant by travelling salesperson problem?
6. What is the running time of dynamic programming TSP?
7. State if backtracking always produces optimal solution.
8. Define backtracking.
9. What are the two types of constraints used in backtracking?
10. What is meant by optimization problem?
11. Define Hamiltonian circuit problem.
12. What is Hamiltonian cycle in an undirected graph?
13. Define 8queens problem. 8. List out the application of backtracking.
14. Define promising node and non-promising node.
15. Give the explicit and implicit constraint for 8-queen problem.
16. How can we represent the solution for 8-queen problem?
17. Give the categories of the problem in backtracking.

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, RAMAPURAM  
CAMPUS**  
**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**  
**QUESTION BANK**  
**18CSC204J DESIGN AND ANALYSIS OF ALGORITHMS**

**UNIT 5**

Introduction to randomization and approximation algorithm - Randomized hiring problem - Randomized quick sort, Complexity analysis - String matching algorithm, Examples - Rabin Karp algorithm for string matching, Example discussion - Approximation algorithm, Vertex covering - Introduction Complexity classes, P type problems - Introduction to NP type problems, Hamiltonian cycle problem - NP complete problem introduction, Satisfiability problem - NP hard problems - Examples

**PART A**

**1. What is a Rabin and Karp Algorithm?**

- (A) String Matching Algorithm
- (B) Shortest Path Algorithm
- (C) Minimum spanning tree Algorithm
- (D) Approximation Algorithm

**Answer: - A**

**2. What is the pre-processing time of Rabin and Karp Algorithm?**

- A) Theta( $m^2$ )
- B) Theta( $m \log n$ )
- C) Theta( $m$ )
- D) Big-Oh( $n$ )

**Answer: - C**

**3. Rabin Karp Algorithm makes use of elementary number theoretic notions.**

- A) True
- B) FALSE

**Answer: - A**

- 4. Given a pattern of length- 5 window, find the spurious hit in the given text string.**

Pattern: 3 1 4 1 5

Modulus: 13

Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Text: 2 3 5 9 0 2 3 1 4 1 5 2 6 7 3 9 9 2 1 3 9

- A) 6-10
- B) 12-16
- C) 3-7
- D) 13-17

**Answer: - D**

- 5. What is the basic principle in Rabin Karp algorithm?**

- A) Hashing
- B) Sorting
- C) Augmenting
- D) Dynamic Programming

**Answer: - A**

- 6. The worst-case efficiency of solving a problem in polynomial time is?**

- A) $O(p(n))$
- B) $O(p(n\log n))$
- C) $O(p(n^2))$
- D) $O(p(m \log n))$

**Answer: - A**

- 7. Problems that can be solved in polynomial time are known as?**

- A) Intractable
- B) Tractable
- C) Decision
- D) Complete

**Answer: - B**

- 8. \_\_\_\_\_ is the class of decision problems that can be solved by non-deterministic polynomial algorithms.**

- A) NP
- B) P
- C) Hard
- D) Complete

**Answer: - A**

**9. The Euler's circuit problem can be solved in?**

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

**Answer: - D**

**10. To which of the following class does a CNF-satisfiability problem belong?**

- A) NP class
- B) P class
- C) NP complete
- D) NP hard

**Answer: - C**

**11. Quick sort uses which of the following algorithm to implement sorting?**

- A) backtracking
- B) greedy algorithm
- C) divide and conquer
- D) dynamic programming

**Answer: - C**

**12. What is the worst case time complexity of randomized quicksort?**

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

**Answer: - C**

**13. What is the purpose of using randomized quick sort over standard quick sort?**

- A) so as to avoid worst case time complexity
- B) so as to avoid worst case space complexity
- C) to improve accuracy of output
- D) to improve average case time complexity

**Answer: - A**

**14. Which of the following is incorrect about randomized quicksort?**

- A) it has the same time complexity as standard quick sort
- B) it has the same space complexity as standard quick sort
- C) it is an in-place sorting algorithm
- D) it cannot have a time complexity of O( $n^2$ ) in any case.

**Answer: - D**

**15. Which of the following is the fastest algorithm in string matching field?**

- A) Boyer-Moore's algorithm
- B) String matching algorithm
- C) Quick search algorithm
- D) Linear search algorithm

**Answer: - C**

**16. What is vertex coloring of a graph?**

- A) A condition where any two vertices having a common edge should not have same color
- B) A condition where any two vertices having a common edge should always have same color
- C) A condition where all vertices should have a different color
- D) A condition where all vertices should have same color

**Answer: - A**

**17. How many edges will a tree consisting of N nodes have?**

- A)  $\log(N)$
- B) N
- C)  $N-1$
- D)  $N+1$

**Answer: - C**

**18. Minimum number of unique colors required for vertex coloring of a graph is called?**

- A) vertex matching
- B) chromatic index
- C) chromatic number
- D) color number.

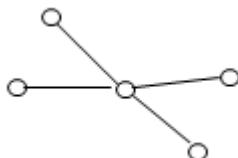
**Answer: - C**

**19. How many unique colors will be required for proper vertex coloring of an empty graph having n vertices?**

- A) 0
- B) 1
- C) n
- D)  $n!$

**Answer: - C**

**20. What will be the chromatic number of the following graph?**



- A) 1
- B) 2
- C) 3

D) 4

**Answer: - B**

**21. Assuming  $P \neq NP$ , which of the following is true ?**

- A) NP-complete = NP
- B) NP-complete  $\cap P = \emptyset$
- C) NP-hard = NP
- D)  $P = NP$ -complete

**Answer: - B**

**22. Let X be a problem that belongs to the class NP. Then which one of the following is TRUE?**

- A) There is no polynomial time algorithm for X.
- B) If X can be solved deterministically in polynomial time, then  $P = NP$ .
- C) If X is NP-hard, then it is NP-complete.
- D) X may be undecidable.

NP Complete

**Answer: - C**

**23. Which of the following statements are TRUE?**

- 1. The problem of determining whether there exists a cycle in an undirected graph is in P.
- 2. The problem of determining whether there exists a cycle in an undirected graph is in NP.
- 3. If a problem A is NP-Complete, there exists a non-deterministic polynomial time algorithm to solve A.

A) 1, 2 and 3

B) 1 and 2 only

C) 2 and 3 only

D) 1 and 3 only

**Answer: - A**

**24. Consider the following two problems on undirected graphs**

$\alpha$  : Given  $G(V, E)$ , does G have an independent set of size  $|V| - 4$ ?

$\beta$  : Given  $G(V, E)$ , does G have an independent set of size 5?

**Which one of the following is TRUE?**

- A)  $\alpha$  is in P and  $\beta$  is NP-complete
- B)  $\alpha$  is NP-complete and  $\beta$  is in P
- C) Both  $\alpha$  and  $\beta$  are NP-complete
- D) Both  $\alpha$  and  $\beta$  are in P

**Answer: - C**

**25. Which of the following algorithm can be used to solve the Hamiltonian path problem efficiently?**

- A) branch and bound

- B) iterative improvement
- C) divide and conquer
- D) greedy algorithm

**Answer: - A**

**26. Hamiltonian path problem is \_\_\_\_\_**

- A) NP problem
- B) N class problem
- C) P class problem
- D) NP complete problem

**Answer: - D**

**27. There is no existing relationship between a Hamiltonian path problem and Hamiltonian circuit problem.**

- A) true
- B) false

**Answer: - B**

**28. Which of the following problems is similar to that of a Hamiltonian path problem?**

- A) knapsack problem
- B) closest pair problem
- C) travelling salesman problem
- D) assignment problem

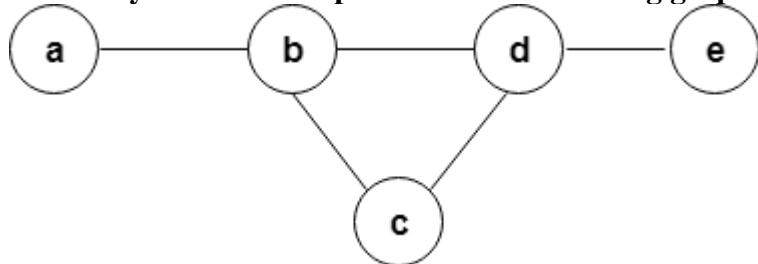
**Answer: - C**

**29. In what time can the Hamiltonian path problem can be solved using dynamic programming?**

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

**Answer: - D**

**30. How many Hamiltonian paths does the following graph have?**



- A) 1
- B) 2
- C) 3
- D) 4

**Answer: - A**

**31. A node is said to be \_\_\_\_\_ if it has a possibility of reaching a complete solution.**

- A) Non-promising
- B) Promising
- C) Succeeding
- D) Preceding

**Answer: - B**

**32. Minimum number of unique colors required for vertex coloring of a graph is called?**

- A) vertex matching
- B) chromatic index
- C) chromatic number
- D) color number

**Answer: C**

## **PART B**

1. Define NP hard and NP completeness.
2. Compare NP hard and NP completeness.
3. Write Short notes on “the class P and NP problem”.
4. How NP Hard problems are different from NP Complete?
5. Whether class P solves a problem in polynomial time? Justify.
6. An NP hard problem can be solved in deterministic polynomial time, how?
7. Give examples for NP Complete problems
8. State the property of NP complete problem.
9. Define adversary method.
10. Define lower bound.
11. What type of output yields trivial lower bound?
12. What is information theoretic lower bound?
13. Define complexity theory.
14. What is halting problem?

1) If the sequence of operations - push(1), push(2), pop, push(1), push(2), pop, pop, pop, push(2), pop are performed on a stack, the sequence of popped out values are ?

- A) 2, 2, 1, 1, 2
- B) 2, 2, 1, 2, 2
- C) 2, 1, 2, 2, 1
- D) 2, 1, 2, 2, 2

Explanation: The elements are popped from the top of the stack.

2) Queue can be used to implement ?

- A) radix sort
- B) quick sort
- C) recursion
- D) depth first search

3) The postfix equivalent of the prefix \* + ab - cd is ?

- A) ab + cd - \*
- B) abcd + - \*
- C) ab + cd \* -
- D) ab + - cd \*

4) The terms PUSH and POP are related to ?

- A) Arrays
- B) Stacks
- C) Linked List
- D) None

5) Minimum number of queues needed to implement the priority queue?

- A) four
- B) three
- C) two
- D) one

6) The data structure required to evaluate a postfix expression is

- (A) queue
- (B) stack
- (C) array
- (D) linked-list

7) What data structure would you mostly likely see in a nonrecursive implementation of a recursive algorithm?

- (A) Stack
- (B) Linked list
- (C) Queue
- (D) Trees

8) Let the following circular queue can accommodate maximum six elements with the following data  
front = 2 rear = 4  
queue = \_\_\_\_\_; L, M, N, \_\_\_, \_\_\_

What will happen after ADD O operation takes place?

- (A) front = 2 rear = 5  
queue = \_\_\_\_\_; L, M, N, O, \_\_\_
- (B) front = 3 rear = 5  
queue = L, M, N, O, \_\_\_
- (C) front = 3 rear = 4  
queue = \_\_\_\_\_; L, M, N, O, \_\_\_
- (D) front = 2 rear = 4  
queue = L, M, N, O, \_\_\_

9) A queue is a,

- (A) FIFO (First In First Out) list
- . (B) LIFO (Last In First Out) list.
- (C) Ordered array.
- (D) Linear tree

10) What is the result of the following operation

Top (Push (S, X))

- (A) X
- (B) null
- (C) S
- (D) None of these.

11) Which data structure is used for implementing recursion?

(A) Queue.

**(B) Stack.**

(C) Arrays.

(D) List.

12) The process of accessing data stored in a serial access memory is similar to manipulating data on a -----?

a) Heap

b) Binary Tree

c) Array

**d) Stack**

13) Consider the linked list implementation of a stack. Which of the following node is considered as Top of the stack?

**a) First node**

b) Last node

c) Any node

d) Middle node

14) Consider the following operation performed on a stack of size 5.

Push(1);

Pop();

Push(2);

Push(3);

Pop();

Push(4);

Pop();

Pop();

Push(5);

After the completion of all operation, the no of element present on stack are

**a) 1**

b) 2

- c) 3
- d) 4

15) Which of the following is not an inherent application of stack?

- a) Reversing a string
- b) Evaluation of postfix expression
- c) Implementation of recursion
- d) Job scheduling**

16) Consider the following array implementation of stack:

```
#define MAX 10
Struct STACK
{
    Int arr [MAX];
    Int top = -1;
}
```

If the array index starts with 0, the maximum value of top which does not cause stack overflow is?

- a) 8**
- b) 9
- c) 10
- d) 11

17) What is the minimum number of stacks of size n required to implement a queue of size n?

- a) One
- b) Two**
- c) Three
- d) Four

18) If the elements “A”, “B”, “C” and “D” are placed in a stack and are deleted one at a time, in what order will they be removed?

- a) ABCD
- b) DCBA**
- c) DCAB
- d) ABDC

19) A linear list of elements in which deletion can be done from one end (front) and insertion can take place only at the other end (rear) is known as a ?

- a) Queue**
- b) Stack
- c) Tree
- d) Linked list

20) The data structure required for Breadth First Traversal on a graph is?

- a) Stack
- b) Array
- c) Queue**
- d) Tree

21) In linked list implementation of a queue, where does a new element be inserted?

- a) At the head of link list
- b) At the tail of the link list**
- c) At the centre position in the link list
- d) None

22) In the array implementation of circular queue, which of the following operation take worst case linear time?

- a) Insertion
- b) Deletion
- c) To empty a queue
- d) None**

23) In linked list implementation of queue, if only front pointer is maintained, which of the following operation take worst case linear time?

- a) Insertion
- b) Deletion
- c) To empty a queue
- d) Both a) and c)

24) If the MAX\_SIZE is the size of the array used in the implementation of circular queue. How is rear manipulated while inserting an element in the queue?

- a)  $\text{rear}=(\text{rear}\%1)+\text{MAX\_SIZE}$
- b)  $\text{rear}=\text{rear}\%( \text{MAX\_SIZE}+1)$
- c)  $\text{rear}=(\text{rear}+1)\% \text{MAX\_SIZE}$
- d)  $\text{rear}=\text{rear}+(1\%\text{MAX\_SIZE})$

25) If the MAX\_SIZE is the size of the array used in the implementation of circular queue, array index start with 0, front point to the first element in the queue, and rear point to the last element in the queue. Which of the following condition specify that circular queue is FULL?

- a) Front=rear= -1
- b)  $\text{Front}=(\text{rear}+1)\% \text{MAX\_SIZE}$
- c) Rear=front+1
- d) Rear=(front+1)\%MAX\_SIZE

26) A circular queue is implemented using an array of size 10. The array index starts with 0, front is 6, and rear is 9. The insertion of next element takes place at the array index.

- a) 0
- b) 7
- c) 9
- d) 10

27) If the MAX\_SIZE is the size of the array used in the implementation of circular queue, array index start with 0, front point to the first element in the queue, and rear point to the last element in the queue. Which of the following condition specify that circular queue is EMPTY?

- a) Front=rear=0
- b) **Front= rear=-1**
- c) Front=rear+1
- d) Front=(rear+1)%MAX\_SIZE

28) A data structure in which elements can be inserted or deleted at/from both the ends but not in the middle is?

- a) Queue
- b) Circular queue
- c) **Dequeue**
- d) Priority queue

29) In linked list implementation of a queue, front and rear pointers are tracked. Which of these pointers will change during an insertion into a NONEMPTY queue?

- a) Only front pointer
- b) **Only rear pointer**
- c) Both front and rear pointer
- d) None of the front and rear pointer

30) A normal queue, if implemented using an array of size MAX\_SIZE, gets full when

- a) Rear=MAX\_SIZE-1
- b) Front=(rear+1)mod MAX\_SIZE
- c) Front=rear+1
- d) Rear=front

31) In linked list implementation of a queue, front and rear pointers are tracked. Which of these pointers will change during an insertion into EMPTY queue?

- a) Only front pointer
- b) Only rear pointer
- c) Both front and rear pointer
- d) None

32) An array of size MAX\_SIZE is used to implement a circular queue. Front, Rear, and count are tracked. Suppose front is 0 and rear is MAX\_SIZE -1. How many elements are present in the queue?

- a) Zero
- b) One
- c) MAX\_SIZE-1
- d) MAX\_SIZE

33) Suppose a circular queue of capacity  $(n-1)$  elements is implemented with an array of  $n$  elements. Assume that the insertion and deletion operations are carried out using REAR and FRONT as array index variables, respectively. Initially REAR=FRONT=0. The conditions to detect queue full and queue is empty are?

- a) Full:  $(REAR+1) \bmod n == FRONT$   
Empty:  $REAR == FRONT$
- b) Full:  $(REAR+1) \bmod n == FRONT$   
Empty:  $(FRONT+1) \bmod n == REAR$
- c) Full:  $REAR == FRONT$   
Empty:  $(REAR+1) \bmod n == FRONT$
- d) Full:  $(FRONT+1) \bmod n == REAR$   
Empty:  $REAR == FRONT$

34) Which of the following is an application of stack?

- A. finding factorial
- B. tower of Hanoi
- C. infix to postfix
- D. all of the above

35) Identify the data structure which allows deletions at both ends of the list but insertion at only one end.

- A) A Stack
- B) B Priority queues
- C) C Output restricted queue
- D) **Input restricted dequeue**

36) Which of the following data structure is non linear type?

- A) **Graph**
- B) B Stacks
- C) C Lists
- D) None

37) In a queue, the initial values of front pointer f rare pointer r should be ..... and ..... respectively.

- A) 0 and 1
- B) **0 and -1**
- C) -1 and 0
- D) 1 and 0

38) There is an extra element at the head of the list called a .....

- A) **Sentinel**
- B) Antinell
- C) List head
- D) List header

39) When new data are to be inserted into a data structure, but there is not available space; this situation is usually called .

- A) **overflow**
- B) Underflow
- C) housefull
- D) memoryfull

40) A data structure where elements can be added or removed at either end but not in the middle is called .....

- A) stacks
- B) queues
- C) **dequeue**
- D) linked lists

41) Which of the following is not the type of queue?

- A) Priority queue
- B) Circular queue
- C) Ordinary queue
- D) **Single ended queue**

42) Which one of the following is an application of Queue Data Structure?

- (A) When a resource is shared among multiple consumers.
- (B) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes
- (C) Load Balancing
- (D) **All of the above**

43) How many stacks are needed to implement a queue. Consider the situation where no other data structure like arrays, linked list is available to you.

- (A) 1
- (B) **2**
- (C) 3
- (D) 4

44) How many queues are needed to implement a stack. Consider the situation where no other data structure like arrays, linked list is available to you.

- (A) 1
- (B) **2**
- (C) 3
- (D) 4

45) A priority queue can efficiently implemented using which of the following data structures? Assume that the number of insert and peek (operation to see the current highest priority item) and extraction (remove the highest priority item) operations are almost same.

- (A) Array
- (B) Linked List
- (C) Heap Data Structures like Binary Heap, Fibonacci Heap**
- (D) None of the above

46) A Priority-Queue is implemented as a Max-Heap. Initially, it has 5 elements. The level-order traversal of the heap is given below:

10, 8, 5, 3, 2

Two new elements "1" and "7" are inserted in the heap in that order. The level-order traversal of the heap after the insertion of the elements is:

- (A) 10, 8, 7, 5, 3, 2, 1
- (B) 10, 8, 7, 2, 3, 1, 5
- (C) 10, 8, 7, 1, 2, 3, 5
- (D) 10, 8, 7, 3, 2, 1, 5**

1. Two main measures for the efficiency of an algorithm are
  - a. Processor and memory
  - b. Complexity and capacity
  - c. Time and space**
  - d. Data and space
  
2. The Worst case occur in linear search algorithm when
  - a. Item is somewhere in the middle of the array
  - b. Item is not in the array at all**
  - c. Item is the last element in the array
  - d. Item is the last element in the array or is not there at all
  
3. The Average case occur in linear search algorithm
  - a. When Item is somewhere in the middle of the array**
  - b. When Item is not in the array at all
  - c. When Item is the last element in the array
  - d. When Item is the last element in the array or is not there at all
  
4. The complexity of linear search algorithm is
  - a.  $O(n)$**
  - b.  $O(\log n)$
  - c.  $O(n^2)$
  - d.  $O(n \log n)$
  
5. The complexity of Bubble sort algorithm is
  - a.  $O(n)$**
  - b.  $O(\log n)$
  - c.  $O(n^2)$
  - d.  $O(n \log n)$
  
6. The complexity of Binary search algorithm is
  - a.  $O(n)$
  - b.  $O(\log n)$**
  - c.  $O(n^2)$
  - d.  $O(n \log n)$
  
7. Which of the following data structure is linear data structure?
  - a. Trees
  - b. Graphs
  - c. Arrays**
  - d. None of above
  
8. What are Abstract Data Type?
  - a) They are a set of values and operations for manipulating those values
  - b) They are a scheme for storing values in computer memory
  - c) Arrays, stacks, queues, lists, and trees are all examples of abstract data types

d) a, b

e) a, c

Answer:E

9. Data structures generally employ which of the following implementation strategies?

a) Contiguous Implementation

b) Linked Implementation

c) All of the mentioned

Answer:C

10. What is Linked Implementation?

a) Values are stored in adjacent memory cells

b) Values are not necessarily stored in adjacent memory cells and are accessed using pointers or references

c) Values are not stored in adjacent memory cells

Answer:B

11. Which one of the below is not divide and conquer approach?

**A** - Insertion Sort

**B** - Merge Sort

**C** - Shell Sort

**D** - Heap Sort

Answer:B

12. If the array is already sorted, which of these algorithms will exhibit the best performance

**A** - Merge Sort

**B** - Insertion Sort

**C** - Quick Sort

**D** - Heap Sort

Answer:B

13. How many swaps are required to sort the given array using bubble sort - { 2, 5, 1, 3, 4 }

**A** - 4

**B** - 5

**C** - 6

**D** - 7

Answer:A

14. Which of the following is not possible with an array in C programming langauge -

**A** - Declaration

**B** - Definition

**C** - Dynamic Allocation

**D** - Array of strings

Answer:C

15. Which of the following sorting methods would be most suitable for sorting a list which is almost sorted

- a. Bubble Sort
- b. Insertion Sort
- c. Selection Sort
- d. Quick Sort

Answer:A

16. Which of the following is not a limitation of binary search algorithm?

- a. must use a sorted array
- b. requirement of sorted array is expensive when a lot of insertion and deletions are needed
- c. there must be a mechanism to access middle element directly
- d. binary search algorithm is not efficient when the data elements are more than 1000.

Answer:D

**17.** An empty list is the one which has no

- a.nodes
- b.data
- c.both a and b
- d.address

Answer:c

**18. In a circular linked list**

- a) Components are all linked together in some sequential manner.
- b) There is no beginning and no end.
- c) Components are arranged hierarchically.
- d) Forward and backward traversal within the list is permitted.

**ANSWER: B**

**19. Which of the following operations is performed more efficiently by doubly linked list than by singly linked list?**

- a) Deleting a node whose location is given
- b) Searching of an unsorted list for a given item
- c) Inverting a node after the node with given location
- d) Traversing a list to process each node

**ANSWER: A**

**20. Consider an implementation of unsorted singly linked list. Suppose it has its representation with a head and tail pointer. Given the representation, which of the following operation can be implemented in O(1) time?**

- i) Insertion at the front of the linked list
  - ii) Insertion at the end of the linked list
  - iii) Deletion of the front node of the linked list
  - iv) Deletion of the last node of the linked list
- a) I and II
  - b) I and III
  - c) I,II and III
  - d) I,II and IV

**ANSWER: C**

**21. In linked list each node contain minimum of two fields. One field is data field to store the data second field is?**

- a) Pointer to character
- b) Pointer to integer
- c) Pointer to node
- d) Node

**ANSWER: C**

**22. What would be the asymptotic time complexity to add an element in the linked list?**

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) None

**ANSWER: B**

**23.** struct node  
{  
int data;  
struct node \* next;  
}  
typedef struct node NODE;  
NODE \*ptr;

Which of the following c code is used to create new node?

- a) ptr=(NODE\*)malloc(sizeof(NODE));
- b) ptr=(NODE\*)malloc(NODE);
- c) ptr=(NODE\*)malloc(sizeof(NODE\*));
- d) ptr=(NODE)malloc(sizeof(NODE));

**ANSWER: A**

**24. Which of the following points is/are true about Linked List data structure when it is compared with array**

- a. Arrays have better cache locality that can make them better in terms of performance
- b. It is easy to insert and delete elements in Linked List

- c. Random access is not allowed in a typical implementation of Linked Lists
- d. All of the above

**ANSWER: D**

**25.** Consider the following function that takes reference to head of a Doubly Linked List as parameter. Assume that a node of doubly linked list has previous pointer as *prev* and next pointer as *next*.

```
void fun(struct node **head_ref)
{
    struct node *temp = NULL;
    struct node *current = *head_ref;

    while (current != NULL)
    {
        temp = current->prev;
        current->prev = current->next;
        current->next = temp;
        current = current->prev;
    }

    if(temp != NULL )
        *head_ref = temp->prev;
}
```

Assume that reference of head of following doubly linked list is passed to above function  
 1 <--> 2 <--> 3 <--> 4 <--> 5 <--> 6. What should be the modified linked list after the function call?

- a. 2 <--> 1 <--> 4 <--> 3 <--> 6 <--> 5
- b. 5 <--> 4 <--> 3 <--> 2 <--> 1 <--> 6
- c. 6 <--> 5 <--> 4 <--> 3 <--> 2 <--> 1
- d. 6 <--> 5 <--> 4 <--> 3 <--> 1 <--> 2

**ANSWER: C**

**26.** The following function reverse() is supposed to reverse a singly linked list. There is one line missing at the end of the function.

```
/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* head_ref is a double pointer which points to head (or start) pointer
   of linked list */
static void reverse(struct node** head_ref)
```

```

{
    struct node* prev = NULL;
    struct node* current = *head_ref;
    struct node* next;
    while (current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
}

```

What should be added in place of "/\*ADD A STATEMENT HERE\*/", so that the function correctly reverses a linked list.

- a. \*head\_ref = prev;
- b. \*head\_ref = current;
- c. \*head\_ref = next
- d. \*head\_ref = NULL

Answer:A

27. What is the output of following function for start pointing to first node of following linked list? 1->2->3->4->5->6

```

void fun(struct node* start)
{
    if(start == NULL)
        return;
    printf("%d ", start->data);

    if(start->next != NULL )
        fun(start->next->next);
    printf("%d ", start->data);
}

```

- a. 1 4 6 6 4 1
- b. 1 3 5 1 3 5
- c. 1 2 3 5
- d. 1 3 5 5 3 1

Answer:D

28. In the worst case, the number of comparisons needed to search a singly linked list of length n for a given element is

- a.  $\log_2 n$
- b.  $n/2$
- c.  $\log_2 n - 1$
- d. n

Answer:D

29. To find out maximum element in a list of n numbers, one needs at least

- A n comparisons
- B n-1 comparisons
- C n(n-1) com

Answer:B

30. In a doubly linked list traversing comes to a halt at:

- A null
- B front
- C rear

Answer:A

31. The following C function takes a singly linked list as input argument. It modifies the list by moving the last element to the front of the list and returns the modified list. Some part of the code left blank.

```
typedef struct node
{
int value;
struct node* next;
}Node;
Node* move_to_front(Node* head)
{
Node* p, *q;
if(head==NULL) || (head->next==NULL))
return head;
q=NULL;
p=head;
while(p->next != NULL)
{
q=p;
p=p->next;
}
return head;
}
```

Choose the correct alternative to replace the blank line

- a) q=NULL; p->next=head; head =p ;
- b) q->next=NULL; head =p; p->next = head;
- c) head=p; p->next=q; q->next=NULL;
- d) q->next=NULL; p->next=head; head=p;

32. The following C Function takes a singly- linked list of integers as a parameter and rearranges

the elements of the lists. The function is called with the list containing the integers 1,2,3,4,5,6,7 in the given order. What will be the contents of the list after the function completes execution?

```
struct node{
int value;
struct node* next;
};
void rearrange (struct node* list)
{
struct node *p,q;
int temp;
if (!List || !list->next) return;
p->list; q=list->next;
while(q)
{
temp=p->value; p->value=q->value;
q->value=temp;p=q->next;
q=p?p->next:0;
}
}
```

- a) 1, 2, 3, 4, 5, 6, 7
- b) 2, 1, 4, 3, 6, 5, 7**
- c) 1, 3, 2, 5, 4, 7, 6
- d) 2, 3, 4, 5, 6, 7, 1

33. What does the following function do for a given Linked List with first node as *head*?

```
void fun1(struct node* head)
{
    if(head == NULL)
        return;

    fun1(head->next);
    printf("%d ", head->data);
}
```

- a. Prints all nodes of linked lists
- b. Prints all nodes of linked list in reverse order**
- c. Prints alternate nodes of list

- d. Prints alternate nodes in reverse order

34. **consider the function f defined here:**

```
struct item
{
int data;
struct item * next;
};
int f (struct item *p)
{
return((p==NULL) ||((p->next==NULL)|| (p->data<=p->next->data) && (p->next)));
}
```

For a given linked list p, the function f returns 1 if and only if

- a) the list is empty or has exactly one element
- b) the element in the list are sorted in non-decreasing order of data value**
- c) the element in the list are sorted in non-increasing order of data value
- d) not all element in the list have the same data value

35. What is the output of following function for start pointing to first node of following linked list? 1->2->3->4->5->6

```
void fun(struct node* start)
{
if(start == NULL)
    return;
printf("%d ", start->data);

if(start->next != NULL )
    fun(start->next->next);
printf("%d ", start->data);
}
```

- a. 1 4 6 6 4 1
- b. 1 3 5 1 3 5
- c. 1 2 3 5
- d. 1 3 5 5 3 1**

# 18CSC201J-DSA - Surprise Test(08.09.2020)

## Instructions

\*\*\*\*\*

1. CT-1 contains 2 sessions. Session-A contains 15 MCQs for 15 marks and Session-B comprises of 5 short answers for 10 marks. Students are requested to complete both the sessions from 9.00am to 10.00am
2. This form is only for Section-A.
3. No negative marking

The respondent's email address (**sg5429@srmist.edu.in**) was recorded on submission of this form.

Register Number \*

RA1911026010007

Name of the Student \*

Sahaj Ghatiya

Section - A Questions



✓ What are the correct intermediate steps of the following data set when it is being sorted with the bubble sort? 15, 20, 10, 18

- 15,18,10,20 --> 10,18,15,20 --> 10,15,18,20 --> 10,15,18,20
- 10, 20,15,18-->10,15,20,18-->10,15,18,20
- 15,20,10,18 --> 15,10,20,18 -->10,15,20,18 --> 10,15,18,20
- 15,10,20,18-->15,10,18,20-->10,15,18,20



✓ What will be the number of passes to sort the elements using insertion sort? 4, 22, 16, 35, 3, 10, 40

- 7
- 5
- 4
- 6



✓ Consider the following input sequence for binary searching technique “2, 4, 6, 8, 10, 11, 15, 17, 18, 20, 23, 25”. To search an element 25 in the above given sequence, What would be the middle element in the second pass?

- 23
- 17
- 18
- 20



- ✓ For the given sequence of input “1,3,4,6,8,10,15,17,19,35”, how many comparisons are required to hit the search element “1” using binary searching technique.

- 5
- 3
- 4
- 2



Identify the output for the given code:

```
int main()
{
    int n=4;
    int A[n];
    A[0]=3; A[1]=4; A[2]=8; A[3]=4;
    printf("%d %d", *(A+2), A[1]);
}
```

- 4 4
- 8 4
- 3 4
- Compiler error





Identify the output for the given code:

```
int main()
{
    int A(4)=[1,2,3,4];
    printf("%d", A(1));
}
```

- 1
- 0
- 2
- Compiler Error



✓ Given a sequence of input element, Find the worst case time complexity of best suitable algorithm to find the first duplicate copy of the given key element

- $O(n^3)$
- $O(\log n)$
- $O(n^2)$
- $O(n)$





Find the computational complexity for the following code:

```
for (cnt=0, i=1;i<=n;i++)  
{  
    for(j=i;j<=n;j++)  
        cnt++;  
}
```

- $i*j$
- $n$
- $n^2$
- $n+1$





What will be the output of the C program?

```
#include<stdio.h>
int main()
{
    struct s
    {
        int i = 20;
        char city[] = "kattankulathur";
    };
    struct s1;
    printf("%d", s1.city);
    printf("%d", s1.i);
    return 0;
}
```

- Compilation Error ✓
- Runtime Error
- kattankulathur 20
- Nothing will be displayed





What will be the output of the C program?

```
#include<stdio.h>
struct {
    int i;
    float f;
}d;

int main()
{
    d.i = 4;
    d.f = 3.678923;
    printf("%d %.2f", d.i, d.f);
    return 0;
}
```

- None of the above
- 4 3.67892
- Compilation error
- 4 3.68





What will be the output of the C program?

```
#include<stdio.h>
struct emp{
    char *empnme;
    int sal;
};

int main()
{
    struct emp e, e1;
    e.empnme = "Rahuvaran";
    e1 = e;
    printf("%s %s", e.empnme, e1.empnme);
    return 0;
}
```

- Compilation Error
- Rahuvaran Garbage value
- RahuvaranRahuvaran
- Garbage value Rahuvaran



\*

What will be the output of the C program?

```
#include<stdio.h>
struct TeamScore
{
    int wickets;
    int score;
} ts = {2, 325};

struct country
{
    char *name;
} coun = {"India"};

int main()
{
    struct TeamScore tcon = ts;
    printf("%d %d %s", tcon.score, ts.wickets, coun.name);
    return 0;
}
```

- None of the above
- 325 2 garbage value
- compilation error
- 325 2 India





What will be the output of the C program?

```
#include<stdio.h>
int main()
{
    struct str
    {
        int s1;
        char st[30];
    };
    struct str s[] = { {1, "struct1"}, {2, "struct2"}, {3, "struct3"} };
    printf("%d %s", s[2].s1, (*(s+2)).st);
}
```

- 2 struct2
- 1 struct1
- Compilation Error
- 3 struct3





What will be the output of the C program?

```
#include<stdio.h>
int main()
{
    struct play{
        char name[10];
        int playnum;
    };

    struct play p1 = {"sachin", 18};
    struct play p2 = p1;
    if(p1 == p2)
        printf("Two structure members are equal");
    return 0;
}
```

- Nothing will be display
- Two structure members are equal
- Compilation Error
- Runtime Error





What will be the output of the C program? Hint : size of int is 2 bytes

```
#include<stdio.h>
int main()
{
    struct employee
    {
        int empid[5];
        int salary;
        struct employee *s;
    }emp;
    printf("%d %d", sizeof(employee), sizeof(emp.empid));
    return 0;
}
```

- 14 10
- 8 2
- 12 10
- 6 2



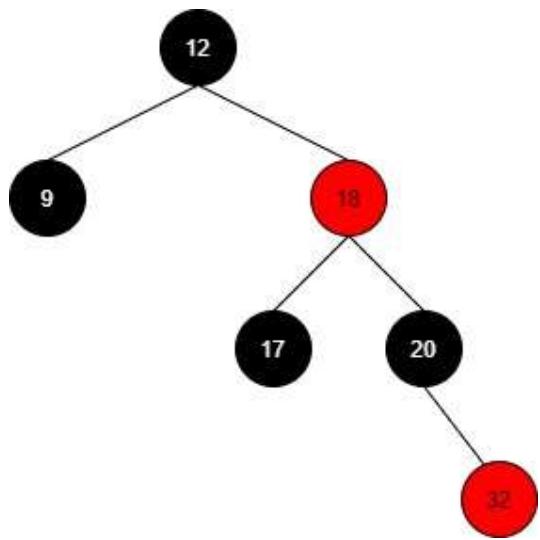
This form was created inside of SRM Institute of Science and Technology.

Google Forms

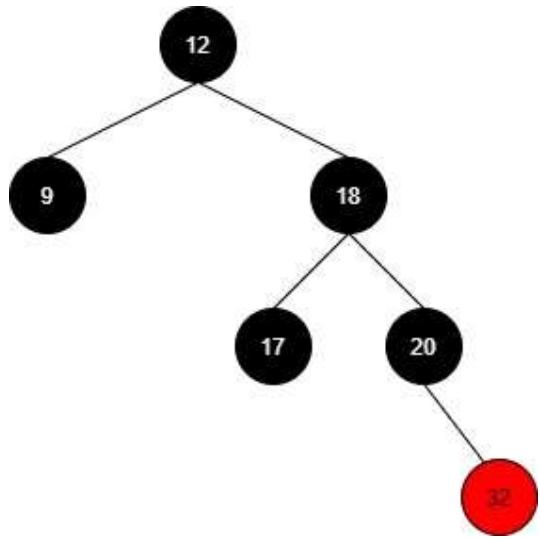


MCQ: Unit IV

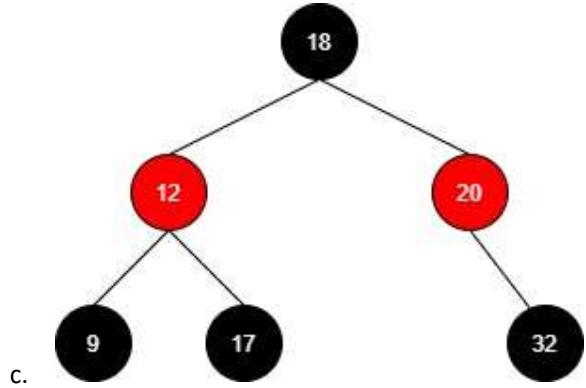
1. A binary search tree generated by inserting the sequence of nodes 10, 15, 13, 2, 5, 8, 1, 7 in order. What will be the height of the resultant tree?
  - a. 6
  - b. 5
  - c. 7
  - d. 8
2. In AVL tree, a node having two children is to be deleted, then it is replaced by its,
  - a. In-order Predecessor
  - b. In-order Successor
  - c. Pre-Order Predecessor
  - d. Pre-order Successor
3. The minimum possible depth of a binary tree with 23 nodes is
  - a. 3
  - b. 4
  - c. 5
  - d. 6
4. A binary search tree formed by inserting the sequence of nodes 47, 12, 59, 2, 17, 55, 88, 1, 5, 34, 57, 21, 35 in order. The number of nodes in the left sub-tree and right sub-tree of the root node is
  - a. (4,8)
  - b. (5,7)
  - c. (7,5)
  - d. (8,4)
5. The best situation to choose B-Tree, Reb-Black Tree and AVL Tree is,
  - a. When Managing more data items, Many Insertions, and Many searching respectively
  - b. Many Insertion, Many Searching, and when Managing more data items respectively
  - c. When Managing more data items, Many Sorting, and Many Insertion respectively
  - d. Many Insertion, Many sorting and Many searching respectively
6. An AVL tree is formed by inserting the sequence of nodes 18, 6, 22, 5, 9, 19, 8 in order. Suppose if we remove the root node by replacing it with something from the left sub tree, what will be the new root?
  - a. 8
  - b. 9
  - c. 5
  - d. 18
7. A B-tree of order 3 and of height 3 will have the maximum of \_\_\_\_ keys, when all nodes are completely filled.
  - a. 26
  - b. 28
  - c. 82
  - d. 80
8. Which of the following is the Red-Black tree structure after inserting the following nodes in sequence : 12, 20, 9, 17, 18, 32.



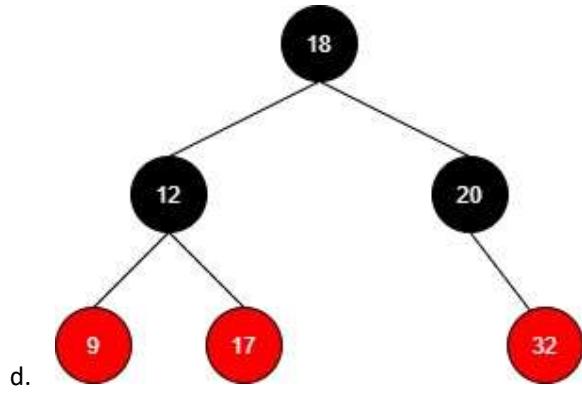
a.



b.



c.

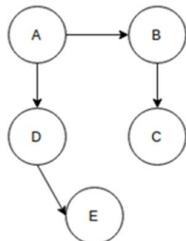


**Answer : A**

9. The infix, prefix and postfix expression is produced from an expression tree by,
  - a. In-order traversal, Post-order traversal and Pre-order traversal respectively
  - b. In-order traversal, Pre-order traversal and Post-order traversal respectively**
  - c. Level – order traversal, Pre-order traversal and Post-order traversal respectively
  - d. Level-order traversal, Post-order traversal and Pre-order traversal respectively
  
10. In a tree, for any node n, every descendant node's value in the left sub tree of n is less than the value of n and every descendant node's value in the right sub tree is greater than the value n. This property should be satisfied for,
  - a. Red-Black tree, Binary search tree and AVL tree
  - b. B-Tree, AVL tree and Heap tree
  - c. Complete binary tree, Red-Black tree and Binary search tree
  - d. Binary search tree, Extended binary tree and AVL tree

#### Unit V

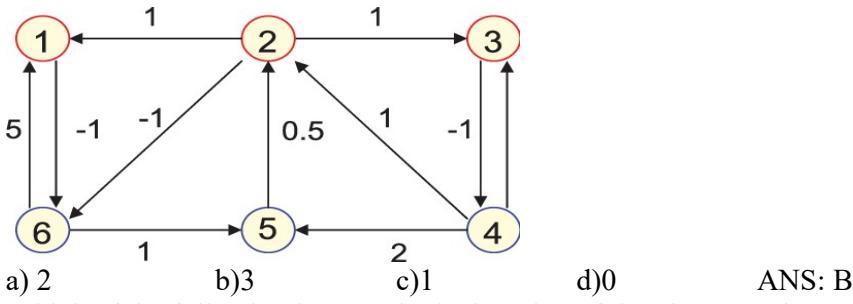
1. What graph traversal algorithm uses a queue data structure to keep track of vertices which need to be processed?
  - a) Breadth-first search.
  - b) Depth-first search.
  - c) Both BFS and DFS
  - d) Neither BFS nor DFSANS:A
  
2. What would be the BFS traversal of the given Graph?



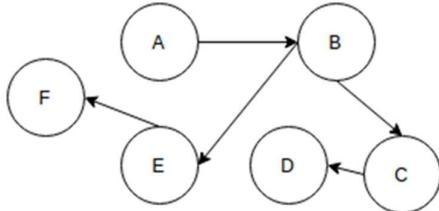
- a) ABCED
- b) AEDCB
- c) ABDCE
- d) ADECB

ANS:C

3. What is the out-degree of the node 2 in the given graph?



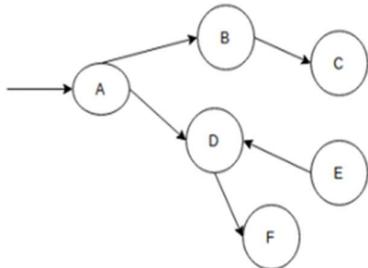
4. Which of the following is a topological sorting of the given graph?



ANS: a b c

- a) A B C D E F  
 b) A B F E D C  
 c) A B E C F D  
 d) B C D E F A

5. What sequence would the BFS traversal of the given graph yield?



- a) A F D B C E  
 b) ABCDFE  
 c) A B D C F  
 d) F D C B A

ANS: B

6. If the number of nodes in complete graph is 4, then the maximum possible number of spanning tree will be \_\_\_\_\_

- a) 16  
 b) 256  
 c) 32  
 d) 8

ANS: A

7. What is the search complexity in direct addressing/hashing?

- a) O(n)  
 b) O(logn)  
 c) O(nlogn)  
 d) O(1)

ANS: D

8. Calculate hash values of keys 1234 and 5462 while m=97

- a) 16,70  
 b) 16,16  
 c) 70,16  
 d) 17,60

Ans: C

9. What is the position of 72, 27, 36 in a hash table with size 10 using Linear probing

- a) 2,7,6

- b) 7,6,2
- c) 6,2,7
- d) 1,3,4

ANS: A

10. What is the position of 271 in the following table

|    |    |
|----|----|
| 0  | 22 |
| 1  | 34 |
| 2  |    |
| 3  |    |
| 4  |    |
| 5  |    |
| 6  |    |
| 7  |    |
| 8  | 41 |
| 9  | 18 |
| 10 |    |

- A.3
- B.4
- C.5
- D.7

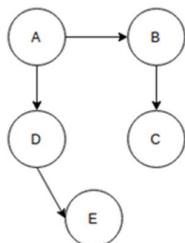
Ans:D

**PART-A**

1. What graph traversal algorithm uses a queue data structure to keep track of vertices which need to be processed?
  - a) Breadth-first search.
  - b) Depth-first search.
  - c) Both BFS and DFS
  - d) Neither BFS nor DFS

ANS:A

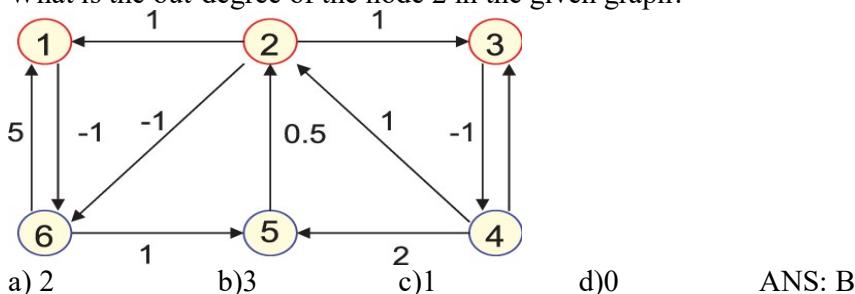
2. What would be the BFS traversal of the given Graph?



- a) ABCED      b) AEDCB      c) ABDCE      d) ADECB

ANS:C

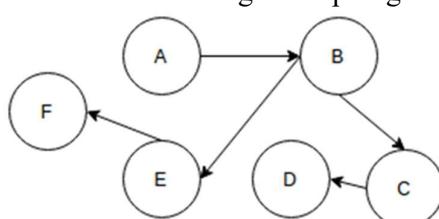
3. What is the out-degree of the node 2 in the given graph?



- a) 2      b) 3      c) 1      d) 0

ANS: B

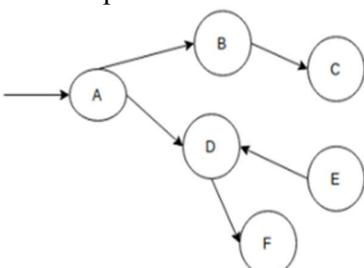
4. Which of the following is a topological sorting of the given graph?



ANS:D

- a) A B C D E F      b) A B F E D C
- c) A B E C F D      d) All of the Mentioned

5. What sequence would the BFS traversal of the given graph yield?



- a) A F D B C E      b) ABCDFE      c) A B D C F      d) F D C B A

ANS:B

6. If the number of nodes in complete graph is 4, then the maximum possible number of spanning tree will be \_\_\_\_\_

- a) 16  
 b) 256  
 c) 32  
 d) 8

ANS: A

7. What is the search complexity in direct addressing/hashing?  
 a)  $O(n)$   
 b)  $O(\log n)$   
 c)  $O(n \log n)$   
 d)  $O(1)$

ANS: D

8. Calculate hash values of keys 1234 and 5462 while  $m=97$

- a) 16,70  
 b) 16,16  
 c) 70,16  
 d) 17,60

Ans: C

9. What is the position of 72, 27, 36 in a hash table with size 10 using Linear probing  
 a) 2,7,6  
 b) 7,6,2  
 c) 6,2,7  
 d) 1,3,4

ANS: A

10. What is the position of 271 in the following table

|    |    |
|----|----|
| 0  | 22 |
| 1  | 34 |
| 2  |    |
| 3  |    |
| 4  |    |
| 5  |    |
| 6  |    |
| 7  |    |
| 8  | 41 |
| 9  | 18 |
| 10 |    |

- A.3  
 B.4  
 C.5  
 D.7  
2 MARKS

Ans:D

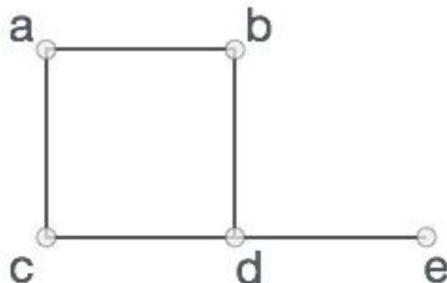
### 1. Define graph.

**Ans:**

- Graph is a non linear data structure,
  - a set of points known as nodes (or vertices)
  - set of links known as edges (or Arcs)
- Graph is a collection of nodes and edges which connects nodes in the graph  
 Generally, a graph G is represented as  $G = (V, E)$  Where V is set of vertices and E is set of edges.

2. Consider the below graph and mention it's vertices and edges.

Ans:

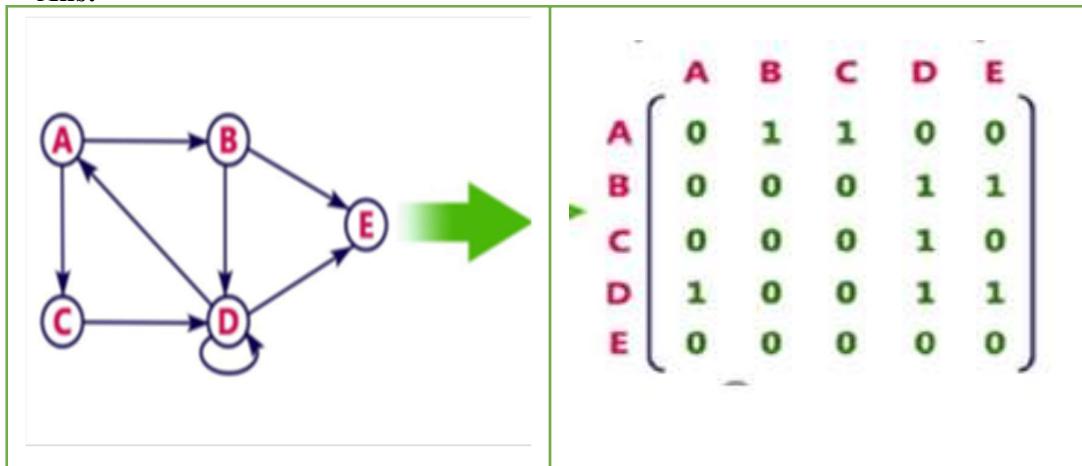


$$V = \{a, b, c, d, e\}$$

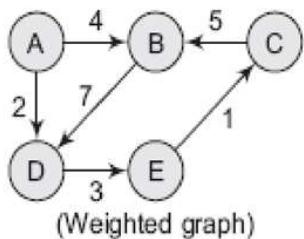
$$E = \{(a,b), (a,c), (b,d), (c,d), (d,e)\}$$

3. Draw the adjacency matrix for the following graph.

Ans:



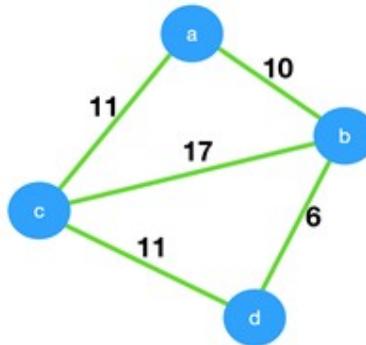
4. What is the adjacency list of following graph?



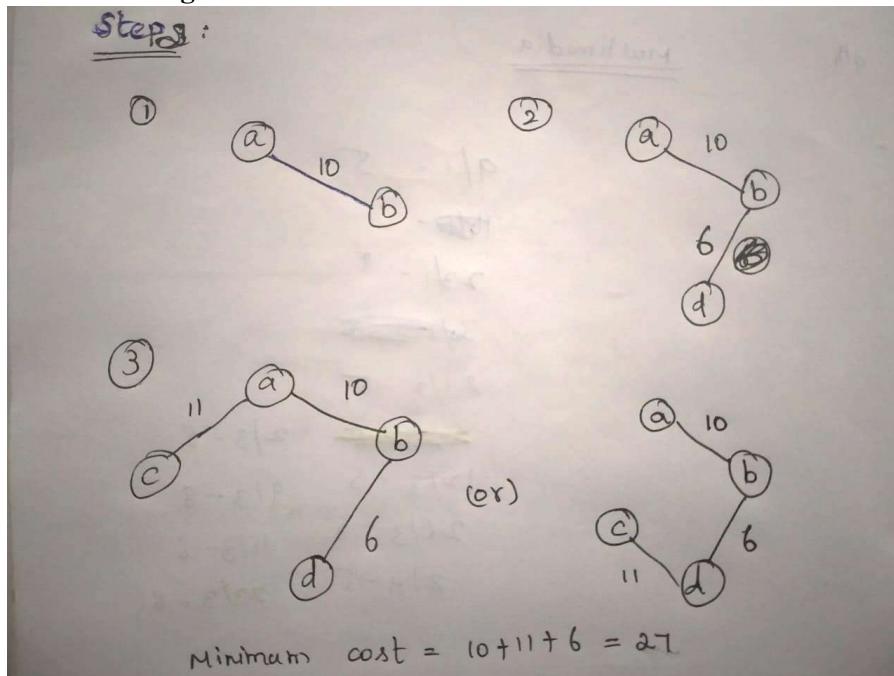
Answer:

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| A | → | B | 4 | — | → | D | 2 | X |
| B | → | D | 7 | X |   |   |   |   |
| C | → | B | 5 | X |   |   |   |   |
| D | → | E | 3 | X |   |   |   |   |
| E | → | C | 1 | X |   |   |   |   |

5. Consider the given graph and find the weight of the minimum spanning tree using the Prim's algorithm, starting vertex is a.



Ans: Starting vertex is a



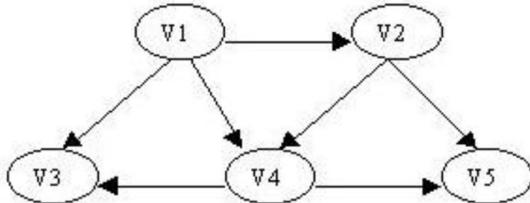
6. Define Topological sorting and explain its algorithm.

Ans:

- Topological sort is an ordering of the vertices in a directed acyclic graph, such that : If there is a path from u to v, then v appears after u in the ordering.

**Algorithm:**

1. Compute the indegrees of all vertices
2. Find a vertex U with indegree 0 and print it (store it in the ordering) If there is no such vertex then there is a cycle and the vertices cannot be ordered. Stop.
3. Remove U and all its edges(U,V)from the graph.
4. Update the indegrees of the remaining vertices.
5. Repeat steps 2 through 4 while there are vertices to be processed.
7. Sort the vertices in the given graph using topological sorting algorithm.



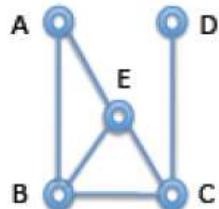
**Ans:**

|               | Indegree |           |              |                 |                    |                       |
|---------------|----------|-----------|--------------|-----------------|--------------------|-----------------------|
| <b>Sorted</b> |          | <b>V1</b> | <b>V1,V2</b> | <b>V1,V2,V4</b> | <b>V1,V2,V4,V3</b> | <b>V1,V2,V4,V3,V5</b> |
| V1            | 0        |           |              |                 |                    |                       |
| V2            | 1        | 0         |              |                 |                    |                       |
| V3            | 2        | 1         | 1            | 0               |                    |                       |
| V4            | 2        | 1         | 0            |                 |                    |                       |
| V5            | 2        | 2         | 1            | 0               | 0                  |                       |

One possible sorting: V1, V2, V4, V3, V5

Another sorting: V1, V2, V4, V5, V3

#### 8. Define Spanning tree and draw any possible two spanning tree for the given graph.

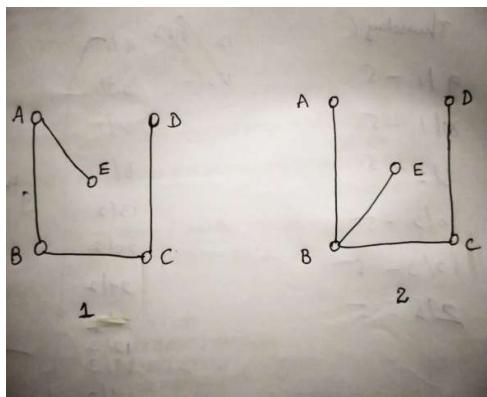


Connected graph

**Ans:**

- A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges.
- A complete undirected graph can have maximum  $n^{n-2}$  number of spanning trees, where n is number of nodes.

#### 2 Possible Spanning Tree



9. Insert the following pairs of values (Key, Value) in an array of size 11 using the hash table method. (Without Linear probing)

(54,"A"), (26,"B"), (93,"C"), (17,"D"), (31,"E"), (44,"F"), (55,"K"), (77,"I"), (20,"L").  
Find out the index at where collision occurred.

Ans:

| <u>After Hashing</u> |   |   |   |    |    |    |   |   |    |    |  |
|----------------------|---|---|---|----|----|----|---|---|----|----|--|
| 0                    | 1 | 2 | 3 | 4  | 5  | 6  | 7 | 8 | 9  | 10 |  |
| 44                   |   |   |   | 22 | 93 | 17 | 1 |   | 31 | 54 |  |

Hash function :  $h(k, i) = k \% m$

$m = 11$ ,  $k = \text{Keys}$

Collision occurred at index 0 while trying to  
insert key (55, "K"), (77, "I")

At index 9  $\rightarrow (20, "L")$

10. Insert the elements 13, 15, 24, 6 into an open addressing hash table of size 7 and apply linear probing when collision occurs. Also Insert 23 in the resultant table, by rehashing method.

Ans:

Linear hash function :

$$h(k, i) = [h'(k) + i] \bmod m$$

where  $i = \text{probe number}$ ,  $h'(k) = k \% m$   
 $(0 \text{ to } m-1)$ ,  $m = 7$

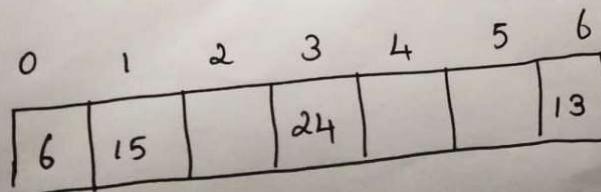
$$h(13, 0) = 6$$

$$h(15, 0) = 1$$

$$h(24, 0) = 2$$

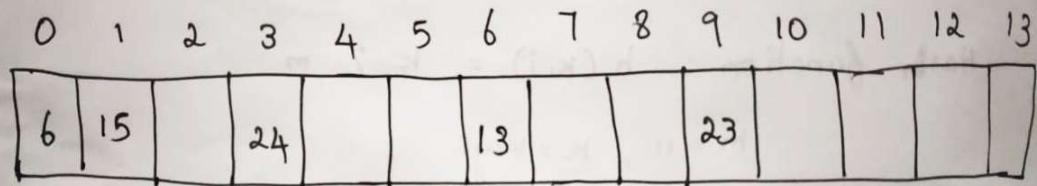
$h(6, 0) = 6 \Rightarrow$  collision occurred increment from 0 to 1

$$\Rightarrow h(6, 1) = 0$$



Insertion of 23 by rehashing:

After rehashing : Hash function  $h(k) = k \% 14$



$$h(23) = 23 \% 14 = 9$$

#### 11. Calculate hash values of keys: 1892 using division method

Ans:

Setting M= 97,

$$h(x) = x \% M$$

$$h(1892) = 1892 \% 97 = 49$$

12. Map these key values using folding method 1921, 2007, 3456.

Ans:

| key        | 1921      | 2007      | 3456      |
|------------|-----------|-----------|-----------|
| Parts      | 19 and 21 | 20 and 07 | 34 and 56 |
| Sum        | 40        | 27        | 90        |
| Hash value | 40        | 27        | 90        |

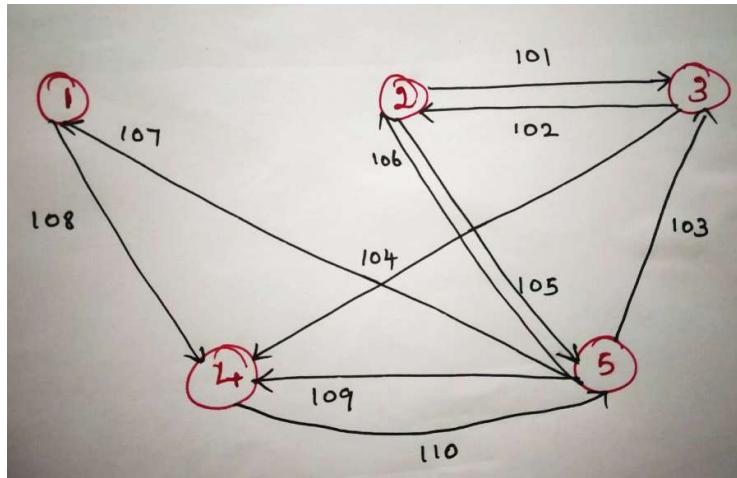
$h(1921) = 40$   
 $h(2007) = 27$   
 $h(3456) = 90$

13. Given five cities (1) New Delhi (2) Mumbai (3) Chennai (4) Bangalore (5) Kolkata and a list of flights that connect these cities as shown in the following table. Use the given information to construct a graph.

| Flight No. | Origin | Destination |
|------------|--------|-------------|
| 101        | 2      | 3           |
| 102        | 3      | 2           |
| 103        | 5      | 3           |
| 104        | 3      | 4           |
| 105        | 2      | 5           |
| 106        | 5      | 2           |
| 107        | 5      | 1           |

|     |   |   |
|-----|---|---|
| 108 | 1 | 4 |
| 109 | 5 | 4 |
| 110 | 4 | 5 |

Ans:



14. Using quadratic probing, insert the following values in a hash table of size 10. Show how many collisions occur in each technique.

99, 33, 23, 44, 56, 43, 19

15. Hash the following numbers into a table of size 11. Use double hashing techniques:  
23, 55, 10, 71, 67, 32, 100, 18, 10, 90, 44

16. Brief Open Addressing technique.

Ans:

- Open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).
- Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.
- Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.
- Delete(k): Delete operation is interesting. If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted".

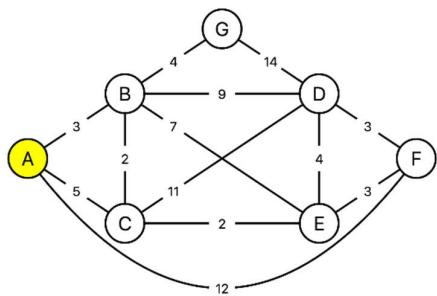
17. What is the maximum number of edges present in a simple directed graph with 7 vertices if there exists no cycles in the graph?

Ans: 6

**18. How is a graph different from a tree though both being non-linear data structures.**

| Sr. No. | Key                   | Graph                                                                                                                                                                                                                 | Tree                                                                                                                                                                                                                                                                                                                   |
|---------|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1       | Definition            | Graph is the graphical representation of nonlinear data where data is denoted by nodes and the relation between them is denoted by connecting path which is known as Edge.                                            | On other hand Tree is also used to represents the nonlinear data but in context of hierarchy where data is again denoted by node and its successive data is denoted by node just below it termed as child nodes/s.                                                                                                     |
| 2       | Implementation        | For representation of nonlinear data Graph is implemented in such a way that the nodes may or may not be connected or even there may be chances of self-loops between nodes to represent the connection between data. | On other hand Tree is implemented in such a manner that each node must have its parent except Parent or first node and must be connected to some other node i.e no node could exist without other node. Also there is no chance of loop or self-loop in case of Tree as data representation is in hierarchical nature. |
| 3       | Data Searching        | As Graph may contain self-loop hence it is difficult to search the data on traversal approach. User has to connect the dots to reach out the desired data.                                                            | On other hand in case of Tree the data is represented as nodes which are connected in hierarchical manner so traversal search could be possible for user to search the desired data at particular level of the tree.                                                                                                   |
| 4       | Parent Child relation | As Graph did not represent the data in hierarchical manner so there is no parent child relation between data representation hence no such parent node or child node do exist in case of Graph.                        | On other hand in case of Tree data is represented in hierarchical manner so parent to child relation exists between the nodes and yes there exist parent node as well as child node in case of Tree.                                                                                                                   |
| 5       | Vice e Versa          | If cases of Graph we can say that all Graphs are not Trees.                                                                                                                                                           | On other hand on case of Tree we can say that All trees are Graphs.                                                                                                                                                                                                                                                    |
| 6       | Usage                 | Main use of graphs is colouring and job scheduling.                                                                                                                                                                   | On other hand Main use of trees is for sorting and traversing.                                                                                                                                                                                                                                                         |

**19. What is the shortest path from Node A to Node F**



**Ans:** A -5-> C -2-> E -3-> F = 10

**20. Mention the table size when the value of p is 9 in multiplication method of creating hash functions?**

**Ans:**

- In multiplication method of creating hash functions the table size can be taken in integral powers of 2.

$$m = 2^p$$

$$m = 2^7$$

$$m = 128.$$

**21. What is the position of 72, 27, 36 in a hash table with size 10 using Linear probing**

Ans:

Linear probing

$$\text{hash function } h(k, i) = [h'(k) + i] \bmod m, \quad m = 10$$

$$h(72) = 2$$

$$h(27) = 7$$

$$h(36) = 6$$

**22. What is the position of 24, 63, 81 in a hash table with size 10 using Quadratic probing . Consider  $c_1 = 1$  and  $c_2 = 3$ .**

Ans:

Quadratic probing

$$h(k, i) = [h'(k) + c_1 i + c_2 i^2] \bmod m$$

$$m = 10, \quad c_1 = 1, \quad c_2 = 3, \quad i = 0 \text{ to } m-1$$

~~$$h(24) = (24 \% 10) + 0 + 0 =$$~~

$$h(24, 0) = [(24 \% 10) + 0 + 0] \% 10 = 4$$

$$h(63, 0) = 3$$

$$h(81, 0) = 1$$

**23.In which type of graph topological ordering may be done.**

**Ans:**

Conditions for Topological ordering

- The graphs should be directed
- The graphs should be a cyclic

**25. What happens in rehashing method to overcome collision.**

- When the hash table becomes nearly full, the number of collisions increases, thereby degrading the performance of insertion and search operations.
- In such cases, a better option is to create a new hash table with size double of the original hash table.
- Rehashing is the building of another table with an associated new hash function that is about twice as big as the original table and scanning down the entire original hash table computing the new hash value for each non deleted element and inserting it in the new table.

**Instructions: This question paper contains Part –A and Part-B.**

**Part A contains 20 MCQs (20 x 1 = 20Marks)**

**Part B contains 20 fill in the blank type of questions where the students should fill the missing statements in the given code snippet by understanding the logic for the specified applications. Each PART B questions carry 1.5 marks (20 x 1.5 = 30Marks). Enjoy coding.... All the best....**

### **Part A (20 x 1 = 20marks)**

#### **MCQ**

1. What is the operation of below function?

```
function(k)
value=x[k];
for(i=k;i<size;i++)
x[i]=x[i+1];
size--;
return(val)
```

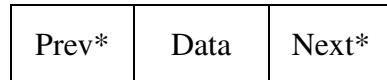
- A. Adding element ‘k’ to the 1 D Array
- B. Removing element ‘k’ from the 1 D Array
- C. Deleting element ‘k’ from 2D Array
- D. Adding element ‘k’ to the Stack ADT

**Answer: B**

2. Insertion and deletion operation in array takes O(n) time complexity, because of
  - A. Insertion and deletion operation require additional space
  - B. Both the operations required the visiting of all elements
  - C. Both the operations required the shifting of elements
  - D. All the above

**Answer: C**

3. Find the correct code for the following node structure.



```
struct node {struct node *prev; int data; struct node *next;}  
struct node { struct node prev; int data; struct node next;}  
struct node { int *prev; int data; int *next;}  
struct node { struct node *prev; struct node data; struct node *next;}
```

**Answer: A**

4. Find the concept which is right for the below condition.

“Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again”.

Circular linked list

Singly linked list

Stack

Queue

**Answer: A**

5. What is the right statement (condition) to check the last node in the circular linked list.  
(Assume temp=head)

temp->next=NULL

temp->next=head

temp=NULL

temp=head

**Answer: B**

6. Which linked list is suitable to find item at given position n?

Singly linked list

Doubly linked list

Circular linked list

Array implementation of linked list

**Answer: D**

7. Find the code used to count the nodes in the circular linked list.

- A.     public int length(Node head)  
    {  
        int length = 0;  
        if( head == null)  
            return 0;  
        Node temp = head.getNext();  
        while(temp!=head )  
        { length++; } }
- B.     public int length(Node head)  
    {  
        int length = 0;  
        if( head == null)  
            return 0;  
        Node temp = head.getNext();  
        while(temp !=NULL)  
        {length++;}
- C.     public int length(Node head)  
    {  
        int length = 0;  
        if( head == null)  
            return 0;  
        Node temp = head.getNext();  
        while(temp.getNext()!=NULL)  
        {  
            length++;  
            temp=temp.getNext();  
        }
- D.     public int length(Node head)  
    {  
        int length = 0;  
        if( head == null)  
            return 0;  
        Node temp = head.getNext();  
        while(temp!=head)  
        {  
            length++;

```
        temp=temp.getNext();
    }
```

**Answer: D**

8. If a header node is used, which of the following indicates a list L with one item?

Header->next = null

Header= null

Header->Next != null

Header!= null

**Answer: A**

9. What is the functionality of the following code?

```
public void function(int data)
{
    int flag = 0;
    if( head != null)
    {
        Node temp = head.getNext();
        while((temp != head) && (!(temp.getItem() == data)))
        {
            temp = temp.getNext();
            flag = 1;
            break;
        }
    }
    if(flag)
        System.out.println("success");
    else
        System.out.println("fail");
}
```

Print success if a particular element is not found

Print fail if a particular element is not found

Print success if a particular element is equal to 1

Print fail if the list is empty

**Answer: B**

10. The number of zero in the sparse matrix of p rows and q columns must be

>  $(p - q) / 2$

>  $(p + q) / 2$

>  $(p / q) / 2$

>  $(p * q) / 2$

**Answer: D**

11. The following sequence of operations is performed on stack:

PUSH (30),PUSH (40),POP,PUSH (30),PUSH (40),POP,POP,POP,PUSH (40),POP

The sequence of the value popped out is:

40,30,40,30,40

30,40,40,30,40

40,40,30,40,30

**40,40,30,30,40**

12 Match the following

- |                            |   |                      |
|----------------------------|---|----------------------|
| 1. top                     | - | a) insert at one end |
| 2. front                   | - | b) job scheduling    |
| 3. stack application       | - | c) delete at one end |
| 4. queue application       | - | d) stack             |
| 5. input restricted queue  | - | e) infix to postfix  |
| 6. output restricted queue | - | f) queue             |

- a) 1-f 2-e 3-d 4-b 5-c 6-a

- b) 1-d 2-f 3-e 4-b 5-a 6-c
- c) 1-d 2-f 3-b 4-e 5-c 6-a
- d) 1-f 2-e 3-d 4-b 5-a 6-c

13. In the worst case time complexity to search an element in singly linked list of length n is

- (a)  $\log_2 n$
- (b)  $n/2$
- (c)  $\log_2 n - 1$
- (d)  $n$

14. The result evaluating the postfix expression 5 5+60 6/\*10- is

- a) 90
- b) 110
- c) 80
- d) 150

15. Convert the given infix expression to postfix expression  $a + b * c - d ^ e ^ f$

- a) abc\*+def^a^-
- b) abc\*+def^
- c) ab+c\*d-e^f^
- d) -+a\*bc^^def

16. Number of moves & Function calls needed respectively to solve Tower of Hanoi Problem, contains n disks are

- a)  $2^{n-1}$  &  $2^{n-1}$
- b)  $2^{n-1}$  &  $2^n - 1$
- c)  $2n - 2$  &  $2^n - 1$
- d)  $2n - 1$  &  $2^n - 2$

17. If MAX is the size of the array used to implement the circular queue. At enqueue how the rear is calculated?

- a) rear=rear%MAX;
- b) rear=(rear+1)%MAX
- c) rear=rear%(MAX+1)
- d) rear=MAX%rear;

18. In Queues, we can insert an element at \_\_\_\_ end and can delete an element at \_\_\_\_ end.

**REAR,FRONT**  
FRONT,REAR  
TOP,BOTTOM  
BOTTOM, TOP

19. Consider E,F,G and H are the four elements in a queue. If we delete an element at a time then on which order they will get deleted?

**EFGH**  
HGFE  
EFHG  
HGEF

20. A circular queue is implemented using an array of size 10. The array index starts with 0, front is 6, and rear is 9. The insertion of next element takes place at the array index of\_\_.

- A. 0
- B. 7
- C. 9
- D. 10

## **Part B (20 x 1.5 = 30)**

### **Fill the missing code**

1. Consider the following function to sort the DLL, find the code left blank.

```
void sort()
{
    int i, j, x;
    i = h;
    j = h;
    if (t2 == NULL)
    {
        printf("\n List empty to sort");
        return;
    }
    for (i = h; _____; i = i->next)(Missing Snippet)
    {
        for (j = i->next; _____; j = j->next)(Missing Snippet)
        {
            if (_____)(Missing Snippet)
            {

```

```

        x = i->n;
        i->n = j->n;
        j->n = x;
    }
}
}

(where 'n' refers to data part of the node)

```

**Answer: (each statement carries 0.5 marks)**

```

i!=NULL,
j != NULL,
i->n > j->n

```

2. The following function modifies the list by making the last element to the start node of the list and returns the modified list. Write the missing code.

```

typedef struct node
{
int value;
struct node* next;
}Node;
Node* move_to_front(Node* head)
{
Node* p, *q;
if((head==NULL) || (head->next==NULL))
    return head;
q=NULL;
p=head;
while(p->next != NULL)
{
q=p;
p=p->next;
}

```

---

(Missing Snippet)

---

(Missing Snippet)

---

(Missing Snippet)

```
return head;  
}
```

Write the correct alternative to replace the blank line

**Answer: (each statement carries 0.5 marks)**

```
q->next=NULL;  
p->next=head;  
head=p;
```

3. Find the missing statement in the below code (inserting node at beginning)

```
if(head == NULL)  
{  
    newNode->next = NULL;
```

\_\_\_\_\_? (Missing Snippet)

```
}  
else  
{  
    newNode->next = head;
```

\_\_\_\_\_?(Missing Snippet)

```
}  
printf("\nOne node inserted!!!\n");
```

**Answer**

```
head = newNode; (0.5 mark)  
head = newNode; (1 mark)
```

4. Find the output of the function for the linked list starting from 1 to 6

```
void f (struct node1* first)  
{  
    if(first == Ø)  
        return;  
    cout<< first->data;
```

```

        if(first->next != Ø )
            f(first->next->next);
        cout<< first->data;
    }

```

**Answer: 1 3 5 5 3 1** (1.5 marks for correct answer; otherwise 0 mark)

5. Consider a scenario that students are playing a game where all the players are tied up by a rope in a circular fashion. Find the missing snippet if a player wants to be inserted

```

void insert(node *pointer, int data)
{
    node *start = pointer;
    while(pointer->next!=start)
    {
        pointer = pointer -> next;
    }
    pointer->next = (node *)malloc(sizeof(node));

    pointer->data= data;
    -----(Missing Snippet)
    -----(Missing Snippet)

}

```

**Answer :**      pointer = pointer->next; (0.5 mark)  
 pointer->next = start; (1 mark)

6. Assume that you are creating a website in which you can traverse a page before and after. Find the missing snippet to delete one of the pages

```

void delete(node *pointer, int data)
{
    while(pointer->next!=NULL && (pointer->next)->data != data)
    {
        pointer = pointer -> next;
    }
    if(pointer->next==NULL)
    {
        printf("Element %d is not present in the list\n",data);
        return;
    }
}

```

```

    }
node *temp;
temp = pointer -> next;
-----(Missing Snippet)
temp->prev = pointer;

free(temp);
return;
}

```

**Answer :**

pointer->next = temp->next; (correct answer: 1.5 marks)

7. Let us assume that you are entering the student's details in a class in a linked fashion.

Find the missing snippet to arrange the students by register numbers.

```

void sort ()
{
    snode *nxt;
    int t;
    if (first == NULL)
    {
        ISEMPY;
        printf(":No elements to sort\n");
    }
else
{
    for (ptr = first;ptr != NULL;ptr = ptr->next)
    {
        for (nxt = ptr->next;nxt != NULL;nxt = nxt->next)
        {
            if (ptr->value > nxt->value)
            {
                -----(Missing Snippet)
                -----(Missing Snippet)
                -----(Missing Snippet)

            }
        }
    }
}

```

**Answer : (each statement carries 0.5 marks)**

t = ptr->value;  
ptr->value = nxt->value;

```
nxt->value = t;
```

8. Consider a linked list in which the last node points to the first node Find the missing snippet for printing all the nodes.

```
void print(node *start, node *pointer)
{
    if(pointer==start) return;

    printf("%d ",pointer->data);
-----(Missing Snippet) // Hint: Recursive function call

}
```

**Answer : print(start,pointer->next); (it is a recursive call : 1.5 marks)**

9. Consider there are n numbers in a list. Find the missing snippet to reverse the linked list

```
struct node
{
    int data;
    struct node* next;
};

void reverse(struct node* first)
{
    struct node* prev = NULL;
    struct node* current = first;
    struct node* next;
    while (current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current;
    }
}
```

```

        current = next;
    }

-----(Missing Snippet)

}

```

**Answer:** first = prev; (1.5 marks)

10. What is the output of following function for start pointing to first node of following linked list? 4->5->6->7->8->9

```

void fun(struct node* start)
{
    if(start == NULL)
        return;
    printf("%d ", start->data);
    if(start->next != NULL )
        fun(start->next->next);
    printf("%d ", start->data);
}

```

**Ans:** 4 6 8 8 6 4 (1.5 marks)

11. Fill in the blanks such that the following code returns the bottom most element from the stack.

```

int bottom_of_stack(struct node *top)
{
    While(-----)(Missing Snippet)
    {
        -----; (Missing Snippet)
    }
    return -----; (Missing Snippet)
}

```

**Ans:** (each statement carries 0.5 marks)

```
top->next!=NULL  
top=top->next;  
return top->data;
```

12. In an array implementation of stack, the stack is filled from the end of the allocated memory. That is, if the size of array is 10, first element to be pushed into the stack will be in arr[9], next element in arr[8] and so on. Fill in the blanks to implement push and pop operations in such a scheme. Assume the edge cases are handled by the main function.

```
int stk[MAX]; //MAX macro contains the size of the stack
```

```
int top= MAX;  
void push(int n)  
{
```

----- (Missing Snippet)

```
}
```

```
void pop()  
{  
----- (Missing snippet)
```

```
}
```

Ans:

```
Stk[--top]=n; (1 mark)  
top++; (0.5 mark)
```

13. A queue is implemented using 2 stacks (Stack1 and Stack2). Initially, Enqueue operation is done by pushing into Stack1. Dequeue operation is done by popping elements from Stack1, pushing them into Stack2 and finally popping from Stack2. If enqueueer has to be done again, elements are popped from Stack2, pushed into Stack1 and finally the element is pushed into Stack1. Fill the blanks for enqueue operation using push1(), push2(), pop1() and pop2() operations which are push and pop operations of Stack1 and Stack2. Pop operations return top of stack while popping.

```
void enqueue(int n)  
{  
If (top2== -1)
```

```

{
    Push1(n);
}
else
if(top2>-1)
{
    While(!Stack2.isEmpty())
    {
        _____; (Missing Snippet)
    }
    Push1(n);
}
}

```

**Ans:**

**Push1(Pop2()); (1.5 marks)**

**(OR)**

**t=pop2(); push1(t); (0.5 and 1 marks respectively)**

14. Write a recursive function to implement division by repeated subtraction. The function should return the quotient of 2 numbers x and y when they are divided.

```

int division(int x,int y)
{
if(x>=y)
{
    return division (____, ____)+____ (Missing Snippet)
}
else
    return 0;
}

```

**Answer**

**Return division (x-y,y)+1 (1.5 marks)**

15. Consider the following code snippet. What is the maximum number of activation records (including main()) that will be held in the stack memory at any point in time during the execution of the program?

```

main()
{
    a();
    b();
}

a()
{
    b();
    c();
    d();
}

b()
{
    c();
}

c()
{
    d();
}

```

**Ans: 4 (1.5 marks)**

16. A person arranged Coid-19 check up camp. For a day only MAX number of people allowed for checkup. It is followed in First come First check order. Write the necessary conditions

```

void insert_in_Q(int queue[],int ele)
{
    if(.....)(Missing Snippet)
    {
        front=rear=0;
        queue[rear]=ele;
    }
    else if(rear==MAX-1)
    {
        _____ (Missing Snippet)
    }
}

```

\_\_\_\_\_ (Missing Snippet)

```
    }
    else
    {
        rear++;
        queue[rear]=ele;
    }
    printf("\nItem inserted..");
}
```

**Answer :** (each statement carries 0.5 marks)

```
voidinsert_in_Q(int queue[],int ele)
{
if(rear== -1)
{
front=rear=0;
queue[rear]=ele;
}
else if(rear==MAX-1)
{
printf("\nQUEUE is full.\n");
return 0;
}
else
{
rear++;
queue[rear]=ele;
}
printf("\nItem inserted..");
}
```

17. Set of students playing ball passing game. Passing ball from one person to another person. And there is MAXIMUM limit to join in group and play the game. If group is full no one can join more. If anyone left from that group then that place will be filled by outsider. Now you write the condition to check whether group of people reaches maximum limit or not.

```
void isfull(int value)
{
if(_____) (Missing Snippet)
{
    printf("\nQueue is Full");
    return;
}
```

```
}
```

**Answer (0.5 mark for condition 1 and 1 mark for the 2<sup>nd</sup> condition)**

```
((front == 0 && rear == size-1) || (rear == (front-1)%(size-1)))
```

18. Children at park playing “merry-go-round” and the operator want to see all the children sitting on this round. He wrote code for that. You too try that.

```
void display(int queue[],int size)
{
    int i;
    printf("\n");
if (front > rear)
{
    for (____)(Missing Snippet)
    {
        printf("%d ", queue[i]);
    }
    for (____)(Missing Snippet)
        printf("%d ", queue[i]);
}
else
{
    for (____)(Missing Snippet)
        printf("%d ", queue[i]);
}
```

**Answer (each statement carries 0.5 marks)**

```
for (i = front; i < size; i++)
for (i = 0; i <= rear; i++)
for (i = front; i <= rear; i++)
```

19. A watch man at ticket counter allowing person one by one to take a ticket. To easy his task Raju write a code for that. But he forgot some portions of code. Help him to fill up.

```
Void enqueue(int num)
{
    node *temp;
    temp=new node;

    if(temp==NULL)
```

```

cout<<endl<<"queue is full\n";

temp->data=num;
temp->link=NULL;
if(front==NULL)
{
    front=rear=temp;
    return;
}

_____ ; (Missing Snippet)
_____ ; (Missing Snippet)
}

```

**Answer**

**rear->link=temp;** (1 mark)  
**rear=rear->link;** (0.5 mark)

20. A group of students are asked to submit their notebooks to the teacher. The class topper submits his notebook first. The rest of the students submit their notebooks above that of the topper. The topper wants his notebook to be checked first. So he pulls his notebook out and submits it on top of the notebook stack. Complete the following code to implement the above scenario in a linked list implementation of stack?

```

struct node * keep_my_note_on_top (struct node * top)
{
    struct node * temp = top;
    if(temp==NULL)
        Return top; //there are no notebooks yet
    if(temp->next==NULL)
        return top; //do nothing since there is only 1 notebook
    while(temp->next->next!=NULL){
        temp=temp->next;
    }
    _____ (Missing Snippet)
    _____ (Missing Snippet)
    temp->next=NULL;
    return top;
}

```

**Answer**

**temp->next->next=top;** (1 mark)  
**top=temp->next;** (0.5 mark)

**18CSC201J – DATA STRUCTURES AND ALGORITHMS**  
**CYCLE TEST -1**

**Total Marks :25**

**Duration: 1 Hour**

**Instructions:**

- 1. CT-1 contains 2 sessions. Session-A contains MCQs that carries 15 marks and Session -B comprises of 5 short answers which carries 10 marks. Students are requested to complete both the sessions from 9.00AM to 10.00AM**
- 2. You can either type the answers or write it by hand and scan for session B and submit it.**
- 3. Multiple copies of the same answers will be awarded zero.**

**SECTION – A ( 15 X 1 = 15)**

**Multiple choice questions:**

1. What are the correct intermediate steps of the following data set when it is being sorted with the bubble sort? 15, 20, 10, 18
  - a) 10, 20,15,18-- 10,15,20,18 -- 10,15,18,20
  - b) 15,10,20,18 -- 15,10,18,20 -- 10,15,18,20
  - c) 15,20,10,18 -- 15,10,20,18 -- 10,15,20,18 -- 10,15,18,20
  - d) 15,18,10,20 -- 10,18,15,20 -- 10,15,18,20 -- 10,15,18,20

Ans: B
2. What will be the number of passes to sort the elements using insertion sort?  
4, 22, 16, 35, 3, 10, 40
  - a) 4
  - b) 5
  - c) 6
  - d) 7

Ans: C
3. Consider the following input sequence for binary searching technique “2, 4, 6, 8, 10, 11, 15, 17, 18, 20, 23, 25”. To search an element 25 in the above given sequence, What would be the middle element in the second pass?  
A.17  
B.20  
C.18  
D.23  
Ans: B
4. For the given sequence of input “1,3,4,6,8,10,15,17,19,35”, how many comparisons are required to hit the search element “1” using binary searching technique.  
A.4  
B.3  
C.5  
D.2

Ans: A

5. Identify the output for the given code:

```
int main()
{
    int n=4;
    int A[n];
    A[0]=3;A[1]=4;
    A[2]=8;A[3]=4;
    printf("%d %d", *(A+2), A[1]);
}
```

- A. 3 4
- B. 8 4
- C. 4 4
- D. Compiler error

Ans: B

6. Identify the output for the given code:

```
int main()
{
    int A(4)=[1,2,3,4];
    printf("%d", A(1));
}
```

- A. 2
- B. 1
- C. 0
- D. Compiler Error

Ans: D

7. Given a sequence of input element, Find the worst case time complexity of best suitable algorithm to find the first duplicate copy of the given key element

- A.O( $n^2$ )
- B.O( $n^3$ )
- C.O(n)
- D.O(logn)

Ans: C

8. Find the computational complexity for the following code:

```
for(cnt=0, i=1;i<=n;i++)
```

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

```
        cnt++;
```

- a. n<sup>2</sup>
- b. n
- c. n+1
- d. i\*j

**Ans: A**

9. What will be the output of the C program?

```
#include<stdio.h>
int main()
{
    struct s
    {
        int i = 20;
        char city[] = "kattankulathur";
    };
    struct s1;
    printf("%d",s1.city);
    printf("%d", s1.i);
    return 0;
}
```

- A. kattankulathur 20
- B. Nothing will be displayed
- C. Runtime Error
- D. Compilation Error

Ans : D

10. What will be the output of the C program?

```
#include<stdio.h>
struct {
    int i;
    float f;
}d;
int main()
{
```

```

d.i = 4;
d.f = 3.678923;
printf("%d %.2f", d.i, d.f);
return 0;
}
A. 4 3.68
B. 4 3.67892
C. Compilation error
D. None of the above

```

Ans : A

11. What will be the output of the C program?

```

#include<stdio.h>
struct emp{
    char *empnme;
    int sal;
};
int main()
{
    struct emp e, e1;
    e.empnme = "Rahuvaran";
    e1 = e;
    printf("%s %s", e.empnme, e1.empnme);
    return 0;
}

```

- A. Garbage value Rahuvaran
- B. Rahuvaran Garbage value
- C. RahuvaranRahuvaran
- D. Compilation Error

Ans : C

12. What will be the output of the C program?

```

#include<stdio.h>
struct TeamScore
{
    int wickets;
    int score;
}ts = {2, 325};
struct country
{
    char *name;
}coun = {"India"};
int main()
{
    struct TeamScore tcon = ts;
    printf("%d %d %s", tcon.score, ts.wickets, coun.name);
    return 0;
}

```

- }
- A. 325 2 India  
 B. 325 2 garbage value  
 C. compilation error  
 D. None of the above

Ans : A

13. What will be the output of the C program?

```
#include<stdio.h>
int main(){
    struct str {
        int s1;
        char st[30];
    };
    struct str s[] = { {1, "struct1"}, {2, "struct2"}, {3, "struct3"} };
    printf("%d %s", s[2].s1, (*(s+2)).st);
}
```

A. Compilation Error  
 B. 1 struct1  
 C. 2 struct2  
 D. 3 struct3

Ans : D

14. What will be the output of the C program?

```
#include<stdio.h>
int main()
{
    Struct play {
        char name[10];
        int playnum;
    };
    struct play p1 = {"sachin", 18};
    struct play p2 = p1;
    if(p1 == p2)
        printf("Two structure members are equal");
    return 0;
}
```

- A. Compilation Error  
 B. Two structure members are equal  
 C. Nothing will be display  
 D. Runtime Error

Ans : A

15. What will be the output of the C program? Hint : size of int is 2 bytes

```
#include<stdio.h>
int main(){
    struct employee
```

```

{
    int empid[5];
    int salary;
    struct employee *s;
}emp;
printf("%d %d", sizeof(employee), sizeof(emp.empid));
return 0;
}
A. 6 2
B. 14 10
C. 8 2
D. 12 10

```

Ans : B

### **SECTION – B ( 5 X 2 = 10)**

#### **Short Answers:**

1. How insertion sort is more efficient than bubble sort?

**Ans:**

- **Fast implementation**
- **Number of swaps reduced than bubble sort.**
- **After some iterations, get sorted sub array**
- **For smaller values of N, insertion sort performs efficiently like other quadratic sorting algorithms.**
- **Stable sort.**

2. Modify linear search to find the frequencies (occurrences) of the given list of elements.  
Hint: Input: [ 1, 4, 8, 1, 4, 8, 1], Output: {3, 2, 2, 3, 2, 2, 3}.

**Ans:**

**Algorithm MLINEARSEARCH(DATA,N)**

**MLINEARSEARCH finds the frequencies for the given list of elements in array DATA[1..N]**

1. Set LOC = 1
2. Repeat For LOC = 1 to N
3. Set CNT=0;
4.     Repeat For J = LOC+1 to N
  5.         if DATA[LOC] = DATA[J]
    6.             CNT++;
    7.         Set OUT[LOC] = CNT;
8. Repeat For LOC =1 to N
  9.         Print OUT[LOC]

3. Derive the time complexity for the following code

```
int a=0;
for(int i=1;i<=n;i++)
{
    if(i<=(n/2))
        print i;
}
```

**Answer: Time complexity:  $3n+n/2+3 = O(n)$**

4. Mention the steps to initialize and access 1D array using pointers.

**Ans:**

**Step1 : Declare the integer array**

```
int a[] = {11,4,3,23,2};
```

**Step 2: Declare integer pointer**

```
int *p;
```

**Step 3: Initialize pointer with base address of array of integers.**

```
p = a; // return base address of array 'a'(using array name)
```

**Step 4: To access the array element we use  $*(p + n)$**

**where n is the number of element.**

**i.e.  $*(p+0)$  returns  $a[0]$  value,  $*(p+1)$  returns  $a[1]$  value**

5. Illustrate the concept of structure with in structure with a student record as example.

- ✓ A binary search tree generated by inserting the sequence of nodes 10, 15, 13, 2, 5, 8, 1, 7 in order. What will be the height of the resultant tree? 1/1

- 6
- 5 ✓
- 7
- 8

- ✓ In AVL tree, a node having two children is to be deleted, then it is replaced by its, 1/1

- In-order Predecessor
- In-order Successor ✓
- Pre-Order Predecessor
- Pre-order Successor

- ✓ The minimum possible depth of a binary tree with 23 nodes is 1/1

- 3
- 4 ✓
- 5
- 6



✓ A binary search tree formed by inserting the sequence of nodes 47, 12, 1/1 59, 2, 17, 55, 88, 1, 5, 34, 57, 21, 35 in order. The number of nodes in the left sub-tree and right sub-tree of the root node is

- (4,8)
- (5,7)
- (7,5)
- (8,4) ✓

✗ The best situation to choose B-Tree, Reb-Black Tree and AVL Tree is, 0/1

- When Managing more data items, Many Insertions, and Many searching respectively
- Many Insertion, Many Searching, and when Managing more data items respectively ✗
- When Managing more data items, Many Sorting, and Many Insertion respectively
- Many Insertion, Many sorting and Many searching respectively

Correct answer

- When Managing more data items, Many Insertions, and Many searching respectively



- ✓ An AVL tree is formed by inserting the sequence of nodes 18, 6, 22, 5, 9, 1/1  
19, 8 in order. Suppose if we remove the root node by replacing it with  
something from the left sub tree, what will be the new root?

- 8
- 9
- 5
- 18

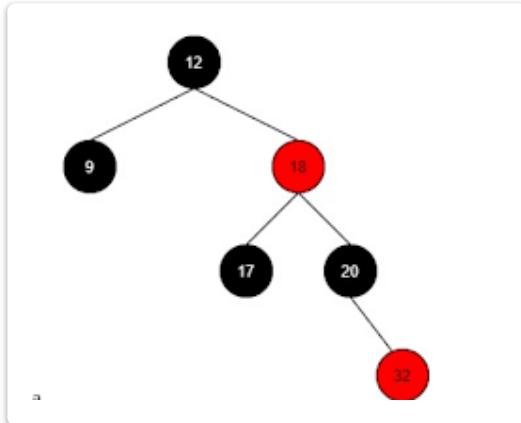


- ✓ A B-tree of order 3 and of height 3 will have the maximum of \_\_\_ keys, 1/1  
when all nodes are completely filled.

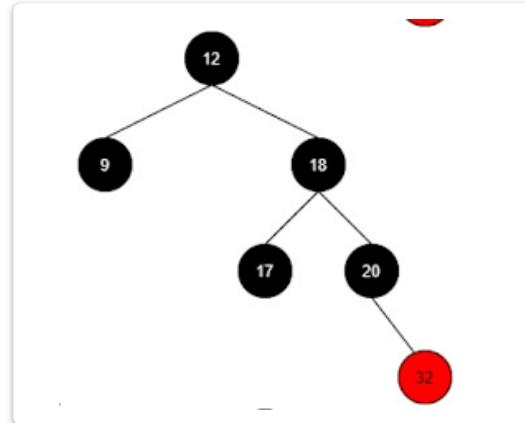
- 26
- 28
- 82
- 80



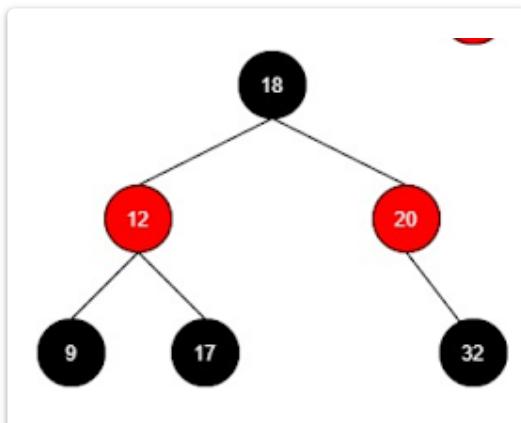
- ✓ Which of the following is the Red-Black tree structure after inserting the following nodes in sequence : 12, 20, 9, 17, 18, 32.



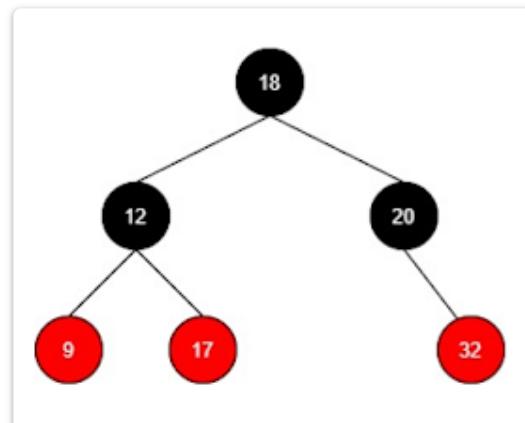
Option 1



Option 2



Option 3



Option 4



✓ The infix, prefix and postfix expression is produced from an expression tree by, 1/1

- In-order traversal, Post-order traversal and Pre-order traversal respectively
- In-order traversal, Pre-order traversal and Post-order traversal respectively ✓
- Level – order traversal, Pre-order traversal and Post-order traversal respectively
- Level-order traversal, Post-order traversal and Pre-order traversal respectively

✓ In a tree, for any node n, every descendant node's value in the left sub tree of n is less than the value of n and every descendant node's value in the right sub tree is greater than the value n. This property should be satisfied for, 1/1

- Red-Black tree, Binary search tree and AVL tree ✓
- B-Tree, AVL tree and Heap tree
- Complete binary tree, Red-Black tree and Binary search tree
- Binary search tree, Extended binary tree and AVL tree

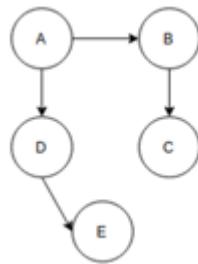
✓ What graph traversal algorithm uses a queue data structure to keep track of vertices which need to be processed? 1/1

- Breadth-first search. ✓
- Depth-first search.
- Both BFS and DFS
- Neither BFS nor DFS



X What would be the BFS traversal of the given Graph?

0/1



- ABCED
- AEDCB
- ABDCE
- ADECB

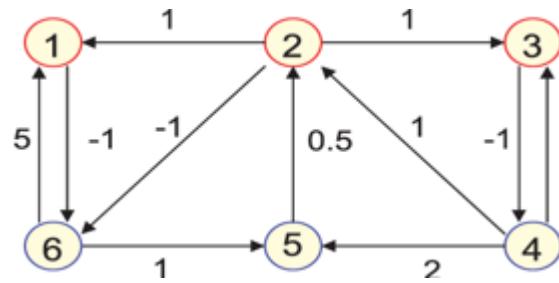
X

Correct answer

- ABDCE

✓ What is the out-degree of the node 2 in the given graph?

1/1

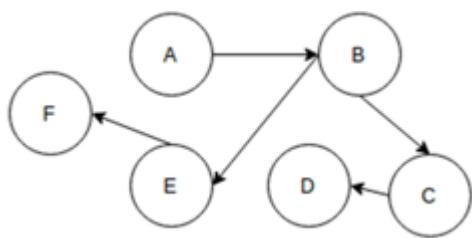


- 2
- 3
- 1
- 0

✓

✖ Which of the following is a topological sorting of the given graph?

0/1



A B C D E F ✓

A B F E D C ✗

A B E C F D ✓

B C D E F A ✓

Correct answer

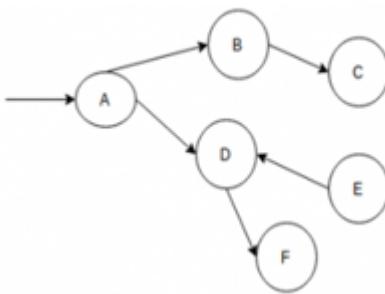
A B C D E F ✓

A B E C F D ✓



✓ What sequence would the BFS traversal of the given graph yield?

1/1



- A F D B C E
- ABCDFE
- A B D C F ✓
- F D C B A

✓ If the number of nodes in complete graph is 4, then the maximum possible number of spanning tree will be \_\_\_\_\_

1/1

- 16 ✓
- 256
- 32
- 8



✓ What is the search complexity in direct addressing/hashing?

1/1

- O(n)
- O(logn)
- O(nlogn)
- O(1)



✓ Calculate hash values of keys 1234 and 5462 while m=97

1/1

- 16,70
- 16,16
- 70,16
- 17,60



✓ What is the position of 72, 27, 36 in a hash table with size 10 using Linear probing

1/1

- 2,7,6
- 7,6,2
- 6,2,7
- 1,3,4



✓ What is the position of 271 in the following table

1/1

|    |    |
|----|----|
| 0  | 22 |
| 1  | 34 |
| 2  |    |
| 3  |    |
| 4  |    |
| 5  |    |
| 6  |    |
| 7  |    |
| 8  | 41 |
| 9  | 18 |
| 10 |    |

- 3
- 4
- 5
- 7



This form was created inside of SRM Institute of Science and Technology.

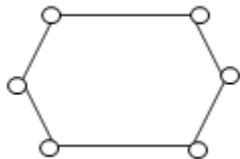
Google Forms



## UNIT 4 – Backtracking and Branch & Bound MCQs

1. What will be the chromatic number of the following graph?

- a) 1    b) 2    c) 3    d) 4



2. What is the condition for proper coloring of a graph?

**a) two vertices having a common edge should not have same color**

b) two vertices having a common edge should always have same color

c) all vertices should have a different color

d) all vertices should have same color

3. What is a chromatic number?

a) The maximum number of colors required for proper edge coloring of graph

b) The maximum number of colors required for proper vertex coloring of graph

**c) The minimum number of colors required for proper vertex coloring of graph**

d) The minimum number of colors required for proper edge coloring of graph

4. The Data structure used in standard implementation of Breadth First Search is?

a) Stack

**b) Queue**

c) Linked List

d) Tree

5. The Data structure used in standard implementation of Depth First Search is?

**a) Stack**

b) Queue

c) Linked List

d) Tree

6. Backtracking algorithm is implemented by constructing a tree of choices called as?

**a) State-space tree**    b) State-chart tree    c) Node tree    d) Backtracking tree

7. A node is said to be \_\_\_\_\_ if it has a possibility of reaching a complete solution.

a) Non-promising      b) **Promising**      c) Succeeding      d) Preceding

8. In what manner is a state-space tree for a backtracking algorithm constructed?

a) Depth-first search    b) Breadth-first search    c) Twice around the tree    d) Nearest neighbour first

9. \_\_\_\_\_ enumerates a list of promising nodes that could be computed to give the possible solutions of a given problem.

a) Exhaustive search    b) Brute force    c) **Backtracking**    d) Divide and conquer

10. A \_\_\_\_\_ is a round trip path along n edges of G that visits every vertex once and return to its starting position

a) MST    b) TSP    c) Multistage Graph    d) **Hamiltonian Cycle**

11. In general, backtracking can be used to solve?

a) Numerical problems    b) Exhaustive search    c) **Combinatorial problems**    d) Graph coloring problems

12. Which of the following is not a branch and bound strategy to generate branches?

a) LIFO branch and bound    b) FIFO branch and bound  
c) Lowest cost branch and bound    d) **Highest cost branch and bound**

13. Which of the following can traverse the state space tree only in DFS manner?

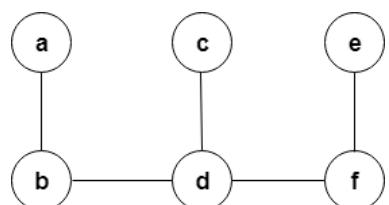
a) branch and bound    b) dynamic programming    c) greedy algorithm    d) **backtracking**

14. which of the following problems is similar to that of a Hamiltonian path problem?

a) knapsack problem    b) closest pair problem  
c) **travelling salesman problem**    d) assignment problem

15. How many Hamiltonian paths does the following graph have?

a) 1    b) 2    c) 3    d) 4



# CS8451 DESIGN AND ANALYSIS OF ALGORITHMS

**CSE - SEMESTER 4**  
**REG. 2017**

---

## **UNIT I INTRODUCTION**

1. Recursion is a method in which the solution of a problem depends on

- a) Larger instances of different problems
- b) Larger instances of the same problem
- c) Smaller instances of the same problem
- d) Smaller instances of different problems

**Answer:** c

**Explanation:** In recursion, the solution of a problem depends on the solution of smaller instances of the same problem.

2. Which of the following problems can't be solved using recursion?

- a) Factorial of a number
- b) Nth fibonacci number
- c) Length of a string
- d) Problems without base case

**Answer:** d

**Explanation:** Problems without base case leads to infinite recursion call. In general, we will assume a base case to avoid infinite recursion call. Problems like finding Factorial of a number, Nth Fibonacci number and

Length of a string can be solved using recursion.

3. Recursion is similar to which of the following?

- a) Switch Case
- b) Loop
- c) If-else
- d) if elif else

**Answer:** b

**Explanation:** Recursion is similar to a loop.

4. In recursion, the condition for which the function will stop calling itself is

- a) Best case
- b) Worst case
- c) Base case
- d) There is no such condition

**Answer:** c

**Explanation:** For recursion to end at some point, there always has to be a condition for which the function will not call itself. This condition is known as base case.

5. Consider the following code snippet:

```
void my_recursive_function()
{
    my_recursive_function();
}
int main()
{
    my_recursive_function();
    return 0;
}
```

What will happen when the above snippet is executed?

- a) The code will be executed successfully and no output will be generated
- b) The code will be executed successfully and random output will be generated
- c) The code will show a compile time error
- d) The code will run for some time and stop when the stack overflows

**Answer:** d

**Explanation:** Every function call is stored in the stack memory. In this case, there is no terminating condition(base case). So, my\_recursive\_function() will be called continuously till the stack overflows and there is no more space to store the function calls. At this point of time, the program will stop abruptly.

6. What is the output of the following code?

```
void my_recursive_function(int n)
{
    if(n == 0)
        return;
    printf("%d ",n);
    my_recursive_function(n-1);
}
int main()
{
    my_recursive_function(10);
    return 0;
}
```

- a) 10
- b) 1
- c) 10 9 8 ... 1 0
- d) 10 9 8 ... 1

**Answer:** d

**Explanation:** The program prints the numbers from 10 to 1.

7. What is the base case for the following code?

```
void my_recursive_function(int n)
{
    if(n == 0)
        return;
    printf("%d ",n);
    my_recursive_function(n-1);
}
int main()
{
    my_recursive_function(10);
    return 0;
}
```

- a) return
- b) printf("%d ", n)

- c) if(n == 0)
- d) my\_recursive\_function(n-1)

**Answer:** c

**Explanation:** For the base case, the recursive function is not called. So, “if(n == 0)” is the base case.

8. How many times is the recursive function called, when the following code is executed?

```
void my_recursive_function(int n)
{
    if(n == 0)
        return;
    printf("%d ",n);
    my_recursive_function(n-1);
}
int main()
{
    my_recursive_function(10);
    return 0;
}
```

- a) 9
- b) 10
- c) 11
- d) 12

**Answer:** c

**Explanation:** The recursive function is called 11 times.

9. What does the following recursive code do?

```
void my_recursive_function(int n)
{
    if(n == 0)
        return;
    my_recursive_function(n-1);
    printf("%d ",n);
}
int main()
{
    my_recursive_function(10);
    return 0;
}
```

- a) Prints the numbers from 10 to 1
- b) Prints the numbers from 10 to 0

- c) Prints the numbers from 1 to 10
- d) Prints the numbers from 0 to 10

**Answer:** c

**Explanation:** The above code prints the numbers from 1 to 10.

10. Which of the following statements is true?
- a) Recursion is always better than iteration
  - b) Recursion uses more memory compared to iteration
  - c) Recursion uses less memory compared to iteration
  - d) Iteration is always better and simpler than recursion

**Answer:** b

**Explanation:** Recursion uses more memory compared to iteration because every time the recursive function is called, the function call is stored in stack.

11. What will be the output of the following code?

```
int cnt=0;
void my_recursive_function(int n)
{
    if(n == 0)
        return;
    cnt++;
    my_recursive_function(n/10);
}
int main()
{
    my_recursive_function(123456789);
    printf("%d",cnt);
    return 0;
}
```

- a) 123456789
- b) 10
- c) 0
- d) 9

**Answer:** d

**Explanation:** The program prints the number of digits in the number 123456789, which is 9.

12. What will be the output of the following code?

```
void my_recursive_function(int n)
{
    if(n == 0)
    {
        printf("False");
        return;
    }
    if(n == 1)
    {
        printf("True");
        return;
    }
    if(n%2==0)
        my_recursive_function(n/2);
    else
    {
        printf("False");
        return;
    }
}
int main()
{
    my_recursive_function(100);
    return 0;
}
```

- a) True
- b) False

**Answer:** b

**Explanation:** The function checks if a number is a power of 2. Since 100 is not a power of 2, it prints false.

13. What is the output of the following code?

```
int cnt = 0;
void my_recursive_function(char *s, int i )
{
    if(s[i] == '\0')
        return;
    if(s[i] == 'a' || s[i] == 'e' || s[i]
    ] == 'i' || s[i] == 'o' || s[i] == 'u')
        cnt++;
    my_recursive_function(s,i+1);
}
int main()
{
    my_recursive_function("thisisrecursi
on",0);
```

```

    printf("%d", cnt);
    return 0;
}

```

- a) 6
- b) 9
- c) 5
- d) 10

**Answer:** a

**Explanation:** The function counts the number of vowels in a string. In this case the number of vowels is 6.

14. What is the output of the following code?

```

void my_recursive_function(int *arr, int
                           val, int idx, int len)
{
    if(idx == len)
    {
        printf("-1");
        return ;
    }
    if(arr[idx] == val)
    {
        printf("%d", idx);
        return;
    }
    my_recursive_function(arr, val, idx+1, l
en);
}
int main()
{
    int array[10] = {7, 6, 4, 3, 2, 1, 9
, 5, 0, 8};
    int value = 2;
    int len = 10;
    my_recursive_function(array, value,
0, len);
    return 0;
}

```

- a) 3
- b) 4
- c) 5
- d) 6

**Answer:** b

**Explanation:** The program searches for a value in the given array and prints the index at which the value is found. In this case, the program searches for value = 2. Since, the

index of 2 is 4(0 based indexing), the program prints 4.

1. In general, which of the following methods isn't used to find the factorial of a number?
  - a) Recursion
  - b) Iteration
  - c) Dynamic programming
  - d) Non iterative / recursive

**Answer:** d

**Explanation:** In general we use recursion, iteration and dynamic programming to find the factorial of a number. We can also implement without using iterative / recursive method by using tgammal() method. Most of us never use it generally.

2. Which of the following recursive formula can be used to find the factorial of a number?
  - a) fact(n) = n \* fact(n)
  - b) fact(n) = n \* fact(n+1)
  - c) fact(n) = n \* fact(n-1)
  - d) fact(n) = n \* fact(1)

**Answer:** c

**Explanation:**  $\text{fact}(n) = n * \text{fact}(n - 1)$  can be used to find the factorial of a number.

3. Consider the following iterative implementation to find the factorial of a number:

```

int main()
{
    int n = 6, i;
    int fact = 1;
    for(i=1;i<=n;i++)
    {
        _____;
        printf("%d", fact);
        return 0;
    }
}

```

Which of the following lines should be inserted to complete the above code?

- a) fact = fact + i
- b) fact = fact \* i
- c) i = i \* fact
- d) i = i + fact

**Answer:** b

**Explanation:** The line “fact = fact \* i” should be inserted to complete the above code.

4. Consider the following recursive implementation to find the factorial of a number:

```
int fact(int n)
{
    if(_____)
        return 1;
    return n * fact(n - 1);
}

int main()
{
    int n = 5;
    int ans = fact(n);
    printf("%d",ans);
    return 0;
}
```

Which of the following lines should be inserted to complete the above code?

- a)  $n = 0$
- b)  $n \neq 0$
- c)  $n == 0$
- d)  $n == 1$

**Answer:** c

**Explanation:** The line “ $n == 0$ ” should be inserted to complete the above code.

Note: “ $n == 1$ ” cannot be used because it does not take care of the case when  $n = 0$ , i.e when we want to find the factorial of 0.

5. The time complexity of the following recursive implementation to find the factorial of a number is \_\_\_\_\_

```
int fact(int n)
{
    if(_____)
        return 1;
    return n * fact(n - 1);
}

int main()
{
    int n = 5;
    int ans = fact(n);
    printf("%d",ans);
}
```

```
    return 0;
}
```

- a)  $O(1)$
- b)  $O(n)$
- c)  $O(n^2)$
- d)  $O(n^3)$

**Answer:** b

**Explanation:** The time complexity of the above recursive implementation to find the factorial of a number is  $O(n)$ .

6. What is the space complexity of the following recursive implementation to find the factorial of a number?

```
int fact(int n)
{
    if(_____)
        return 1;
    return n * fact(n - 1);
}

int main()
{
    int n = 5;
    int ans = fact(n);
    printf("%d",ans);
    return 0;
}
```

- a)  $O(1)$
- b)  $O(n)$
- c)  $O(n^2)$
- d)  $O(n^3)$

**Answer:** a

**Explanation:** The space complexity of the above recursive implementation to find the factorial of a number is  $O(1)$ .

7. Consider the following recursive implementation to find the factorial of a number:

```
int fact(int n)
{
    if(n == 0)
        return 1;
    return n * fact(n - 1);
}
```

```
int main()
{
    int n = 5;
    int ans = fact(n);
    printf("%d",ans);
    return 0;
}
```

Which of the following lines is the base case?

- a) return 1
- b) return n \* fact(n-1)
- c) if(n == 0)
- d) if(n == 1)

**Answer:** c

**Explanation:** The line “if(n == 0)” is the base case.

8. What is the output of the following code?

```
int fact(int n)
{
    if(n == 0)
        return 1;
    return n * fact(n - 1);
}
int main()
{
    int n = 0;
    int ans = fact(n);
    printf("%d",ans);
    return 0;
}
```

- a) 0
- b) 1
- c) 2
- d) 3

**Answer:** b

**Explanation:** The program prints 0!, which is 1.

9. What is the output of the following code?

```
int fact(int n)
{
    if(n == 0)
        return 1;
    return n * fact(n - 1);
}
int main()
{
```

```
    int n = 1;
    int ans = fact(n);
    printf("%d",ans);
    return 0;
}
```

- a) 0
- b) 1
- c) 2
- d) 3

**Answer:** b

**Explanation:** The program prints 1!, which is 1.

10. How many times will the function fact() be called when the following code is executed?

```
int fact(int n)
{
    if(n == 0)
        return 1;
    return n * fact(n - 1);
}
int main()
{
    int n = 5;
    int ans = fact(n);
    printf("%d",ans);
    return 0;
}
```

- a) 4
- b) 5
- c) 6
- d) 7

**Answer:** c

**Explanation:** The fact() function will be called 6 times with the following arguments: fact(5), fact(4), fact(3), fact(2), fact(1), fact(0).

11. What is the output of the following code?

```
int fact(int n)
{
    if(n == 0)
        return 1;
    return n * fact(n - 1);
}
```

```
int main()
{
    int n = 5;
    int ans = fact(n);
    printf("%d",ans);
    return 0;
}
```

- a) 24
- b) 120
- c) 720
- d) 1

**Answer:** b

**Explanation:** The function prints 5!, which is 120.

---

1. Suppose the first fibonnaci number is 0 and the second is 1. What is the sixth fibonnaci number?

- a) 5
- b) 6
- c) 7
- d) 8

**Answer:** a

**Explanation:** The sixth fibonnaci number is 5.

2. Which of the following is not a fibonnaci number?

- a) 8
- b) 21
- c) 55
- d) 14

**Answer:** d

**Explanation:** 14 is not a fibonnaci number.

3. Which of the following option is wrong?

- a) Fibonacci number can be calculated by using Dynamic programming
- b) Fibonacci number can be calculated by using Recursion method
- c) Fibonacci number can be calculated by using Iteration method
- d) No method is defined to calculate Fibonacci number

**Answer:** d

**Explanation:** Fibonacci number can be calculated by using Dynamic Programming, Recursion method, Iteration Method.

4. Consider the following iterative implementation to find the nth fibonacci number?

```
int main()
{
    int n = 10,i;
    if(n == 1)
        printf("0");
    else if(n == 2)
        printf("1");
    else
    {
        int a = 0, b = 1, c;
        for(i = 3; i <= n; i++)
        {
            c = a + b;
            _____;
            _____;
        }
        printf("%d",c);
    }
    return 0;
}
```

Which of the following lines should be added to complete the above code?

- a)

c = b

b = a

- b)

a = b

b = c

- c)

b = c

a = b

- d)

a = b

```
b = a
```

**Answer:** b

**Explanation:** The lines “ $a = b$ ” and “ $b = c$ ” should be added to complete the above code.

5. Which of the following recurrence relations can be used to find the nth fibonacci number?

- a)  $F(n) = F(n) + F(n - 1)$
- b)  $F(n) = F(n) + F(n + 1)$
- c)  $F(n) = F(n - 1)$
- d)  $F(n) = F(n - 1) + F(n - 2)$

**Answer:** d

**Explanation:** The relation  $F(n) = F(n - 1) + F(n - 2)$  can be used to find the nth fibonacci number.

6. Consider the following recursive implementation to find the nth fibonacci number:

```
int fibo(int n)
{
    if(n == 1)
        return 0;
    else if(n == 2)
        return 1;
    return _____;
}
int main()
{
    int n = 5;
    int ans = fibo(n);
    printf("%d",ans);
    return 0;
}
```

Which of the following lines should be inserted to complete the above code?

- a)  $fibo(n - 1)$
- b)  $fibo(n - 1) + fibo(n - 2)$
- c)  $fibo(n) + fibo(n - 1)$
- d)  $fibo(n - 2) + fibo(n - 1)$

**Answer:** b

**Explanation:** The line  $fibo(n - 1) + fibo(n -$

2) should be inserted to complete the above code.

7. Consider the following recursive implementation to find the nth fibonnaci number:

```
int fibo(int n)
{
    if(n == 1)
        return 0;
    else if(n == 2)
        return 1;
    return fibo(n - 1) + fibo(n - 2);
}
int main()
{
    int n = 5;
    int ans = fibo(n);
    printf("%d",ans);
    return 0;
}
```

Which of the following is the base case?

- a)  $if(n == 1)$
- b)  $else if(n == 2)$
- c)  $return fibo(n - 1) + fibo(n - 2)$
- d) both  $if(n == 1)$  and  $else if(n == 2)$

**Answer:** d

**Explanation:** Both  $if(n == 1)$  and  $else if(n == 2)$  are the base cases.

8. What is the time complexity of the following recursive implementation to find the nth fibonacci number?

```
int fibo(int n)
{
    if(n == 1)
        return 0;
    else if(n == 2)
        return 1;
    return fibo(n - 1) + fibo(n - 2);
}
int main()
{
    int n = 5;
    int ans = fibo(n);
    printf("%d",ans);
    return 0;
}
```

- a) O(1)
- b) O( $2^n$ )
- c) O( $n^2$ )
- d) O( $2^n$ )

**Answer:** d

**Explanation:** The time complexity of the above recursive implementation to find the nth fibonacci number is O( $2^n$ ).

9. What is the space complexity of the following recursive implementation to find the nth fibonacci number?

```
int fibo(int n)
{
    if(n == 1)
        return 0;
    else if(n == 2)
        return 1;
    return fibo(n - 1) + fibo(n - 2);
}
int main()
{
    int n = 5;
    int ans = fibo(n);
    printf("%d",ans);
    return 0;
}
```

- a) O(1)
- b) O( $2^n$ )
- c) O( $n^2$ )
- d) O( $2^n$ )

**Answer:** a

**Explanation:** The space complexity of the above recursive implementation to find the nth fibonacci number is O(1).

10. What is the output of the following code?

```
int fibo(int n)
{
    if(n == 1)
        return 0;
    else if(n == 2)
        return 1;
    return fibo(n - 1) + fibo(n - 2);
}
int main()
```

```
{
    int n = -1;
    int ans = fibo(n);
    printf("%d",ans);
    return 0;
}
```

- a) 0
- b) 1
- c) Compile time error
- d) Runtime error

**Answer:** d

**Explanation:** Since negative numbers are not handled by the program, the function fibo() will be called infinite times and the program will produce a runtime error when the stack overflows.

11. What is the output of the following code?

```
int fibo(int n)
{
    if(n == 1)
        return 0;
    else if(n == 2)
        return 1;
    return fibo(n - 1) + fibo(n - 2);
}
int main()
{
    int n = 5;
    int ans = fibo(n);
    printf("%d",ans);
    return 0;
}
```

- a) 1
- b) 2
- c) 3
- d) 5

**Answer:** c

**Explanation:** The program prints the 5th fibonacci number, which is 3.

12. How many times will the function fibo() be called when the following code is executed?

```
int fibo(int n)
{
```

```

    if(n == 1)
        return 0;
    else if(n == 2)
        return 1;
    return fibo(n - 1) + fibo(n - 2);
}
int main()
{
    int n = 5;
    int ans = fibo(n);
    printf("%d",ans);
    return 0;
}

```

- a) 5
- b) 6
- c) 8
- d) 9

**Answer:** d

**Explanation:** The function fibo() will be called 9 times, when the above code is executed.

13. What is the output of the following code?

```

int fibo(int n)
{
    if(n == 1)
        return 0;
    else if(n == 2)
        return 1;
    return fibo(n - 1) + fibo(n - 2);
}
int main()
{
    int n = 10;
    int ans = fibo(n);
    printf("%d",ans);
    return 0;
}

```

- a) 21
- b) 34
- c) 55
- d) 13

**Answer:** b

**Explanation:** The program prints the 10th fibonacci number, which is 34.

1. Which of the following option is wrong about natural numbers?

- a) Sum of first n natural numbers can be calculated by using Iteration method
- b) Sum of first n natural numbers can be calculated by using Recursion method
- c) Sum of first n natural numbers can be calculated by using Binomial coefficient method
- d) No method is prescribed to calculate sum of first n natural number

**Answer:** d

**Explanation:** All of the above mentioned methods can be used to find the sum of first n natural numbers.

2. Which of the following gives the sum of the first n natural numbers?

- a)  $nC2$
- b)  $(n-1)C2$
- c)  $(n+1)C2$
- d)  $(n+2)C2$

**Answer:** c

**Explanation:** The sum of first n natural numbers is given by  $n*(n+1)/2$ , which is equal to  $(n+1)C2$ .

3. Consider the following iterative solution to find the sum of first n natural numbers:

```

#include<stdio.h>
int get_sum(int n)
{
    int sm = 0, i;
    for(i = 1; i <= n; i++)
        _____;
    return sm;
}
int main()
{
    int n = 10;
    int ans = get_sum(n);
    printf("%d",ans);
    return 0;
}

```

Which of the following lines completes the above code?

- a)  $sm = i$

- b)  $sm += i$
- c)  $i = sm$
- d)  $i += sm$

**Answer:** b

**Explanation:** The line “ $sm += i$ ” completes the above code.

4. What is the output of the following code?

```
#include<stdio.h>
int get_sum(int n)
{
    int sm, i;
    for(i = 1; i <= n; i++)
        sm += i;
    return sm;
}

int main()
{
    int n = 10;
    int ans = get_sum(n);
    printf("%d",ans);
    return 0;
}
```

- a) 55
- b) 45
- c) 35
- d) Depends on compiler

**Answer:** d

**Explanation:** Since the variable “ $sm$ ” is not initialized to 0, it will produce a garbage value. Some compiler will automatically initialise variables to 0 if not initialised. In that case the value is 55. Hence the value depends on the compiler.

5. What is the time complexity of the following iterative method used to find the sum of the first n natural numbers?

```
#include<stdio.h>
int get_sum(int n)
{
    int sm, i;
    for(i = 1; i <= n; i++)
        sm += i;
    return sm;
}

int main()
```

```
{
    int n = 10;
    int ans = get_sum(n);
    printf("%d",ans);
    return 0;
}

a) O(1)
b) O(n)
c) O( $n^2$ )
d) O( $n^3$ )
```

**Answer:** b

**Explanation:** The time complexity of the above iterative method used to find the sum of first n natural numbers is  $O(n)$ .

6. Consider the following code:

```
#include<stdio.h>
int recursive_sum(int n)
{
    if(n == 0)
        return 0;
    return _____;
}

int main()
{
    int n = 5;
    int ans = recursive_sum(n);
    printf("%d",ans);
    return 0;
}
```

Which of the following lines is the recurrence relation for the above code?

- a)  $(n - 1) + \text{recursive\_sum}(n)$
- b)  $n + \text{recursive\_sum}(n)$
- c)  $n + \text{recursive\_sum}(n - 1)$
- d)  $(n - 1) + \text{recursive\_sum}(n - 1)$

**Answer:** c

**Explanation:** The recurrence relation for the above code is:  $n + \text{recursive\_sum}(n - 1)$ .

7. Consider the following code:

```
#include<stdio.h>
int recursive_sum(int n)
{
    if(n == 0)
        return 0;
```

```

        return n + recursive_sum(n - 1);
}
int main()
{
    int n = 5;
    int ans = recursive_sum(n);
    printf("%d",ans);
    return 0;
}

```

Which of the following is the base case for the above recursive code?

- a) if( $n == 0$ )
- b) return 0
- c) return  $n + \text{recursive\_sum}(n - 1)$
- d) if( $n == 1$ )

**Answer:** a

**Explanation:** “if( $n == 0$ )” is the base case for the above recursive code.

8. What is the time complexity of the following recursive implementation used to find the sum of the first  $n$  natural numbers?

```

#include<stdio.h>
int recursive_sum(int n)
{
    if(n == 0)
        return 0;
    return n + recursive_sum(n - 1);
}
int main()
{
    int n = 5;
    int ans = recursive_sum(n);
    printf("%d",ans);
    return 0;
}

a) O(1)
b) O( $n$ )
c) O( $n^2$ )
d) O( $n^3$ )

```

**Answer:** b

**Explanation:** The time complexity of the above recursive implementation used to find the sum of first  $n$  natural numbers is  $O(n)$ .

9. Which of the following methods used to find the sum of first  $n$  natural numbers has the

least time complexity?

- a) Recursion
- b) Iteration
- c) Binomial coefficient
- d) All have equal time complexity

**Answer:** c

**Explanation:** Recursion and iteration take  $O(n)$  time to find the sum of first  $n$  natural numbers while binomial coefficient takes  $O(1)$  time.

10. What is the output of the following code?

```

#include<stdio.h>
int recursive_sum(int n)
{
    if(n == 0)
        return 0;
    return n + recursive_sum(n - 1);
}
int main()
{
    int n = 5;
    int ans = recursive_sum(n);
    printf("%d",ans);
    return 0;
}

```

- a) 10
- b) 15
- c) 21
- d) 14

**Answer:** b

**Explanation:** The above code prints the sum of first 5 natural numbers, which is 15.

11. How many times is the function `recursive_sum()` called when the following code is executed?

```

#include<stdio.h>
int recursive_sum(int n)
{
    if(n == 0)
        return 0;
    return n + recursive_sum(n - 1);
}
int main()
{
    int n = 5;

```

```

int ans = recursive_sum(n);
printf("%d",ans);
return 0;
}

```

- a) 4
- b) 5
- c) 6
- d) 7

**Answer:** c

**Explanation:** The function recursive\_sum is called 6 times when the following code is executed.

12. What is the output of the following code?

```

#include<stdio.h>
int recursive_sum(int n)
{
    if(n == 0)
        return 0;
    return n + recursive_sum(n - 1);
}
int main()
{
    int n = 0;
    int ans = recursive_sum(n);
    printf("%d",ans);
    return 0;
}

```

- a) -1
- b) 0
- c) 1
- d) runtime error

**Answer:** b

**Explanation:** The program prints the sum of first 0 natural numbers, which is 0.

13. What is the output of the following code?

```

#include<stdio.h>
int recursive_sum(int n)
{
    if(n == 0)
        return 0;
    return n + recursive_sum(n - 1);
}
int main()
{
    int n = -4;

```

```

int ans = recursive_sum(n);
printf("%d",ans);
return 0;
}

```

- a) 0
- b) -10
- c) 1
- d) runtime error

**Answer:** d

**Explanation:** The above code doesn't handle the case of negative numbers and so the function recursive\_sum() will be called again and again till the stack overflows and the program produces a runtime error.

1. Which of the following is not another name for GCD(Greatest Common Divisor)?

- a) LCM
- b) GCM
- c) GCF
- d) HCF

**Answer:** a

**Explanation:** : LCM (Least Common Multiple) and GCD are not same. GCM (Greatest Common Measure), GCF (Greatest Common Factor), HCF (Highest Common Factor) are other names for GCD.

2. What is the GCD of 8 and 12?

- a) 8
- b) 12
- c) 2
- d) 4

**Answer:** d

**Explanation:** GCD is largest positive integer that divides each of the integer. So the GCD of 8 and 12 is 4.

3. If GCD of two number is 8 and LCM is 144, then what is the second number if first number is 72?

- a) 24
- b) 2

- c) 3  
d) 16

**Answer:** d

**Explanation:** As  $A * B = \text{GCD}(A, B) * \text{LCM}(A, B)$ . So  $B = (144 * 8)/72 = 16$ .

4. Which of the following is also known as GCD?

- a) Highest Common Divisor
- b) Highest Common Multiple
- c) Highest Common Measure
- d) Lowest Common Multiple

**Answer:** a

**Explanation:** GCM (Greatest Common Measure), GCF (Greatest Common Factor), HCF (Highest Common Factor) and HCF (Highest Common Divisor) are also known as GCD.

5. Which of the following is coprime number?

- a) 54 and 24
- b) 4 and 8
- c) 6 and 12
- d) 9 and 28

**Answer:** d

**Explanation:** Coprime numbers have GCD 1. So 9 and 28 are coprime numbers. While 54 and 24 have GCD 6, 4 and 8 have GCD 4, 6 and 12 have GCD 6.

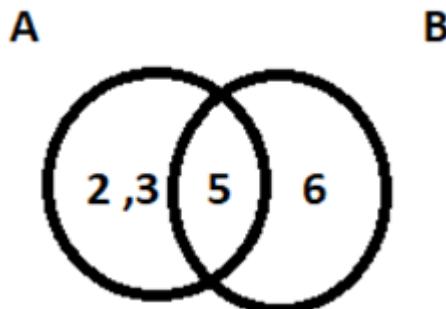
6. In terms of Venn Diagram, which of the following expression gives GCD (Given  $A \cap B \neq \emptyset$ )?

- a) Multiplication of  $A \cup B$  terms
- b) Multiplication of  $A \cap B$  terms
- c) Multiplication of  $A * B$  terms
- d) Multiplication of  $A - B$  terms

**Answer:** b

**Explanation:** In terms of Venn Diagram, the GCD is given by the intersection of two sets. So  $A \cap B$  gives the GCD. While  $A \cup B$  gives the LCM.

7. What is the GCD according to the given Venn Diagram?



- a) 2
- b) 3
- c) 5
- d) 6

**Answer:** c

**Explanation:** In terms of Venn Diagram, the GCD is given by the intersection of two sets. So  $A \cap B$  gives the GCD. While  $A \cup B$  gives the LCM. So here  $A \cap B$  is 5.

8. What is the GCD of a and b?

- a)  $a + b$
- b)  $\text{gcd}(a-b, b)$  if  $a > b$
- c)  $\text{gcd}(a+b, a-b)$
- d)  $a - b$

**Answer:** b

**Explanation:** As per Euclid's Algorithm,  $\text{gcd}(a, b) = \text{gcd}(a-b, b)$  if  $a > b$  or  $\text{gcd}(a, b) = \text{gcd}(a, b-a)$  if  $b > a$ .

9. What is the GCD of 48, 18, 0?

- a) 24
- b) 2
- c) 3
- d) 6

**Answer:** d

**Explanation:** GCD is largest positive integer that divides each of the integers. So the GCD of 48, 18, 0 is 6.

10. Is 9 and 28 coprime number?

- a) True
- b) False

**Answer:** a

**Explanation:** Coprime numbers have GCD 1. So 9 and 28 are coprime numbers.

11. If gcd (a, b) is defined by the expression,  $d=a*p + b*q$  where d, p, q are positive integers and a, b is both not zero, then what is the expression called?

- a) Bezout's Identity
- b) Multiplicative Identity
- c) Sum of Product
- d) Product of Sum

**Answer:** a

**Explanation:** If gcd (a, b) is defined by the expression,  $d=a*p + b*q$  where d, p, q are positive integers and a, b is both not zero, then the expression is called Bezout's Identity and p, q can be calculated by extended form of Euclidean algorithm.

12. Is gcd an associative function.

- a) True
- b) False

**Answer:** a

**Explanation:** The gcd function is an associative function as  $\text{gcd} (a, \text{gcd} (b, c)) = \text{gcd} (\text{gcd} (a, b), c)$ .

13. Which is the correct term of the given relation,  $\text{gcd} (a, b) * \text{lcm} (a, b) = ?$

- a)  $|a*b|$
- b)  $a + b$
- c)  $a - b$
- d)  $a / b$

**Answer:** a

**Explanation:** The gcd is closely related to lcm as  $\text{gcd} (a, b) * \text{lcm} (a, b) = |a*b|$ .

14. Who gave the expression for the probability and expected value of gcd?

- a) James E. Nymann
- b) Riemann
- c) Thomas
- d) Euler

**Answer:** a

**Explanation:** In the year 1972, James E. Nymann showed some result to show the probability and expected value of gcd.

15. What is the computational complexity of Binary GCD algorithm where a and b are integers?

- a)  $O (\log a + \log b)^2$
- b)  $O (\log (a + b))$
- c)  $O (\log ab)$
- d)  $O (\log a-b)$

**Answer:** a

**Explanation:** From the Binary GCD algorithm, it is found that the computational complexity is  $O (\log a + \log b)^2$  as the total number of steps in the execution is at most the total sum of number of bits of a and b.

1. LCM is also called as \_\_\_\_\_

- a) GCD
- b) SCM
- c) GCF
- d) HCF

**Answer:** b

**Explanation:** GCD (Greatest Common Divisor), GCF (Greatest Common Factor), HCF (Highest Common Factor) is not an alias for LCM. LCM is also called as Smallest Common Multiple(SCM).

2. What is the LCM of 8 and 13?

- a) 8
- b) 12
- c) 20
- d) 104

**Answer:** d

**Explanation:** 104 is the smallest positive integer that is divisible by both 8 and 12.

3. Which is the smallest number of 3 digits that is divisible by 2, 4, 8?

- a) 100
- b) 102

- c) 116  
d) 104

**Answer:** d

**Explanation:** LCM of 2, 4, 8 is 8. So check for the number that is divisible by 8. So 104 is the smallest number that is divisible by 2, 4, 8.

4. Which of the following is also known as LCM?

- a) Lowest Common Divisor
- b) Least Common Multiple
- c) Lowest Common Measure
- d) Highest Common Multiple

**Answer:** a

**Explanation:** Least Common Multiple is also known as LCM or Lowest Common Multiple.

5. What is the LCM of two coprime numbers?

- a) 1
- b) 0
- c) Addition of two coprime numbers
- d) Multiplication of two coprime numbers

**Answer:** d

**Explanation:** Coprime numbers have GCD 1. While LCM of coprime numbers is the product of those two coprime numbers.

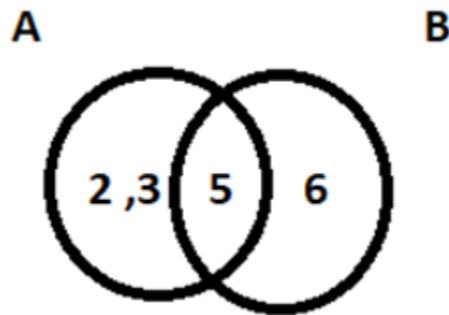
6. In terms of Venn Diagram, which of the following expression gives LCM (Given  $A \cap B \neq \emptyset$ )?

- a) Multiplication of  $A \cup B$  terms
- b) Multiplication of  $A \cap B$  terms
- c) Multiplication of  $A * B$  terms
- d) Multiplication of  $A - B$  terms

**Answer:** a

**Explanation:** In terms of Venn Diagram, the LCM is given by the Union of two sets. So  $A \cup B$  gives the LCM. While  $A \cap B$  gives the GCD.

7. What is the LCM according to the given Venn Diagram?



- a) 2
- b) 3
- c) 180
- d) 6

**Answer:** c

**Explanation:** In terms of Venn Diagram, the LCM is given by the Union of two sets. So  $A \cup B$  gives the LCM. So product of all the terms is 180.

8. What is the lcm (a, b)?

- a)  $a + b$
- b)  $\gcd(a-b, b)$  if  $a > b$
- c)  $\text{lcm}(b, a)$
- d)  $a - b$

**Answer:** c

**Explanation:** Since the LCM function is commutative, so  $\text{lcm}(a, b) = \text{lcm}(b, a)$ .

9. What is the LCM of 48, 18, 6?

- a)  $12^2$
- b)  $12 * 2$
- c) 3
- d) 6

**Answer:** a

**Explanation:** The LCM of 48, 18, 6 is 144 and  $12^2$  is 144.

10. Is 9 and 28 coprime number.

- a) True
- b) False

**Answer:** a

**Explanation:** Coprime numbers have GCD 1

and LCM is the product of the two given terms. So 9 and 28 are coprime numbers.

11. What is the following expression,  $\text{lcm}(a, \text{lcm}(b, c))$  equal to?

- a)  $\text{lcm}(a, b, c)$
- b)  $a^*b^*c$
- c)  $a + b + c$
- d)  $\text{lcm}(\text{lcm}(a, b), c)$

**Answer:** d

**Explanation:** Since LCM function follows associativity, hence  $\text{lcm}(a, \text{lcm}(b, c))$  is equal to  $\text{lcm}(\text{lcm}(a, b), c)$ .

12. Is lcm an associative function.

- a) True
- b) False

**Answer:** a

**Explanation:** The lcm function is an associative function as  $\text{lcm}(a, \text{lcm}(b, c))$  is equal to  $\text{lcm}(\text{lcm}(a, b), c)$ .

13. Which is the correct term of the given relation,  $\text{lcm}(a, b) * \text{gcd}(a, b) = ?$

- a)  $|a^*b|$
- b)  $a + b$
- c)  $a - b$
- d)  $a / b$

**Answer:** a

**Explanation:** The lcm is closely related to gcd as  $\text{lcm}(a, b) * \text{gcd}(a, b) = |a^*b|$ .

14. What is the following expression,  $\text{lcm}(a, \text{gcd}(a, b))$  equal to?

- a) a
- b) b
- c)  $a^*b$
- d)  $a + b$

**Answer:** a

**Explanation:** Since the lcm function follows absorption laws so  $\text{lcm}(a, \text{gcd}(a, b))$  equal to a.

15. Which algorithm is the most efficient numerical algorithm to obtain lcm?

- a) Euler's Algorithm
- b) Euclid's Algorithm
- c) Chebyshev Function
- d) Partial Division Algorithm

**Answer:** b

**Explanation:** The most efficient way of calculating the lcm of a given number is using Euclid's algorithm which computes the lcm in much lesser time compared to other algorithms.

1. Which of the following methods can be used to find the sum of digits of a number?

- a) Recursion
- b) Iteration
- c) Greedy algorithm
- d) Both recursion and iteration

**Answer:** d

**Explanation:** Both recursion and iteration can be used to find the sum of digits of a number.

2. What can be the maximum sum of digits for a 4 digit number?

- a) 1
- b) 16
- c) 36
- d) 26

**Answer:** c

**Explanation:** The sum of digits will be maximum when all the digits are 9. Thus, the sum will be maximum for the number 9999, which is 36.

3. What can be the minimum sum of digits for a 4 digit number?

- a) 0
- b) 1
- c) 16
- d) 36

**Answer:** b

**Explanation:** The sum of digits will be minimum for the number 1000 and the sum is 1.

4. Consider the following iterative implementation to find the sum of digits of a number:

```
#include<stdio.h>
int sum_of_digits(int n)
{
    int sm = 0;
    while(n != 0)
    {
        _____;
        n /= 10;
    }
    return sm;
}
int main()
{
    int n = 1234;
    int ans = sum_of_digits(n);
    printf("%d",ans);
    return 0;
}
```

Which of the following lines should be inserted to complete the above code?

- a)  $sm += n$
- b)  $sm += n \% 10$
- c)  $sm += n - 10$
- d)  $sm += n / 10$

**Answer:** b

**Explanation:** The line “ $sm += n \% 10$ ” adds the last digit(LSB) of the number to the current sum. Thus, the line “ $sm += n \% 10$ ” should be added to complete the above code.

5. What is the output of the following code?

```
#include<stdio.h>
int sum_of_digits(int n)
{
    int sm = 0;
    while(n != 0)
    {
        sm += n \% 10;
        n /= 10;
    }
    return sm;
```

```
}
int main()
{
    int n = 1234;
    int ans = sum_of_digits(n);
    printf("%d",ans);
    return 0;
}

a) 1
b) 3
c) 7
d) 10
```

**Answer:** d

**Explanation:** The above code prints the sum of digits of the number 1234, which is 10.

6. Consider the following recursive implementation to find the sum of digits of number:

```
#include<stdio.h>
int recursive_sum_of_digits(int n)
{
    if(n == 0)
        return 0;
    return _____;
}
int main()
{
    int n = 1201;
    int ans = recursive_sum_of_digits(n);
    printf("%d",ans);
    return 0;
}
```

Which of the following lines should be inserted to complete the above code?

- a)  $(n / 10) + \text{recursive\_sum\_of\_digits}(n \% 10)$
- b)  $(n) + \text{recursive\_sum\_of\_digits}(n \% 10)$
- c)  $(n \% 10) + \text{recursive\_sum\_of\_digits}(n / 10)$
- d)  $(n \% 10) + \text{recursive\_sum\_of\_digits}(n \% 10)$

**Answer:** c

**Explanation:** The line “ $(n \% 10) + \text{recursive\_sum\_of\_digits}(n / 10)$ ” should be inserted to complete the above code.

7. What is the time complexity of the following recursive implementation to find the sum of digits of a number n?

```
#include<stdio.h>
int recursive_sum_of_digits(int n)
{
    if(n == 0)
        return 0;
    return _____;
}
int main()
{
    int n = 1201;
    int ans = recursive_sum_of_digits(n);
    printf("%d",ans);
    return 0;
}
```

- a) O(n)
- b) O(1)
- c) O(len(n)), where len(n) is the number of digits in n
- d) O(1/2)

**Answer:** c

**Explanation:** The time complexity of the above recursive implementation to find the sum of digits of a number is O(len(n)).

8. What is the output of the following code?

```
#include<stdio.h>
int recursive_sum_of_digits(int n)
{
    if(n == 0)
        return 0;
    return n % 10 + recursive_sum_of_digits(n/10);
}
int main()
{
    int n = 1234321;
    int ans = recursive_sum_of_digits(n);
    printf("%d",ans);
    return 0;
}
```

- a) 10
- b) 16

- c) 15
- d) 14

**Answer:** b

**Explanation:** The above code prints the sum of digits of the number 1234321, which is 16.

9. How many times is the function recursive\_sum\_of\_digits() called when the following code is executed?

```
#include<stdio.h>
int recursive_sum_of_digits(int n)
{
    if(n == 0)
        return 0;
    return n % 10 + recursive_sum_of_digits(n/10);
}
int main()
{
    int n = 1201;
    int ans = recursive_sum_of_digits(n);
    printf("%d",ans);
    return 0;
}
```

- a) 6
- b) 7
- c) 5
- d) 9

**Answer:** c

**Explanation:** The function recursive\_sum\_of\_digits() is called 8 times, when the following code is executed.

10. You have to find the sum of digits of a number given that the number is always greater than 0. Which of the following base cases can replace the base case for the below code?

```
#include<stdio.h>
int recursive_sum_of_digits(int n)
{
    if(n == 0)
        return 0;
    return n % 10 + recursive_sum_of_digits(n/10);
}
```

```

int main()
{
    int n = 1201;
    int ans = recursive_sum_of_digits(n);
    printf("%d",ans);
    return 0;
}

```

- a) if(n == 0) return 1
- b) if(n == 1) return 0
- c) if(n == 1) return 1
- d) no need to modify the base case

**Answer:** d

**Explanation:** None of the above mentioned base cases can replace the base case if(n == 0) return 0.

11. What is the output of the following code?

```

#include<stdio.h>
int recursive_sum_of_digits(int n)
{
    if(n == 0)
        return 0;
    return n % 10 + recursive_sum_of_digits(n/10);
}
int main()
{
    int n = 10000;
    int ans = recursive_sum_of_digits(n);
    printf("%d",ans);
    return 0;
}

```

- a) 0
- b) 1
- c) runtime error
- d) -1

**Answer:** b

**Explanation:** The program prints the sum of digits of the number 10000, which is 1.

12. What is the output of the following code?

```

#include<stdio.h>
int cnt =0;
int my_function(int n, int sm)
{

```

```

    int i, tmp_sm;
    for(i=1;i<=n;i++)
    {
        tmp_sm = recursive_sum_of_digits(i);
        if(tmp_sm == sm)
            cnt++;
    }
    return cnt;
}

int recursive_sum_of_digits(int n)
{
    if(n == 0)
        return 0;
    return n % 10 + recursive_sum_of_digits(n/10);
}

int main()
{
    int n = 20, sum = 3;
    int ans = my_function(n,sum);
    printf("%d",ans);
    return 0;
}

a) 0
b) 1
c) 2
d) 3

```

**Answer:** c

**Explanation:** The code prints the count of numbers between 1 and 20 such that the sum of their digits is 3. There are only two such numbers: 3 and 12.

1. Consider the following iterative implementation used to reverse a string:

```

#include<stdio.h>
#include<string.h>
void reverse_string(char *s)
{
    int len = strlen(s);
    int i,j;
    i=0;
    j=len-1;
    while(_____)
    {
        char tmp = s[i];
        s[i] = s[j];
        s[j] = tmp;
        i++;
    }
}

```

```

        j--;
    }
}

```

Which of the following lines should be inserted to complete the above code?

- a)  $i > j$
- b)  $i < \text{len}$
- c)  $j > 0$
- d)  $i < j$

**Answer:** d

**Explanation:** The line “ $i < j$ ” should be inserted to complete the above code.

2. What is the output of the following code?

```

#include<stdio.h>
#include<string.h>
void reverse_string(char *s)
{
    int len = strlen(s);
    int i,j;
    i=0;
    j=len-1;
    while(i < j)
    {
        char tmp = s[i];
        s[i] = s[j];
        s[j] = tmp;
        i++;
        j--;
    }
}
int main()
{
    char s[100] = "reverse";
    reverse_string(s);
    printf("%s",s);
    return 0;
}

```

- a) ersevre
- b) esrever
- c) eserver
- d) eresevr

**Answer:** b

**Explanation:** The program reverses the string “reverse” and prints “esrever”.

3. What is the time complexity of the above code used to reverse a string?

- a)  $O(1)$
- b)  $O(n)$
- c)  $O(n^2)$
- d)  $O(n^3)$

**Answer:** b

**Explanation:** The time complexity of the above code used to reverse a string is  $O(n)$ .

4. What does the following code do?

```

#include<stdio.h>
#include<string.h>
void reverse_string(char *s)
{
    int len = strlen(s);
    int i,j;
    i=0;
    j=len-1;
    while(i < j)
    {
        char tmp = s[i];
        s[i] = s[j];
        s[j] = tmp;
        i++;
        j--;
    }
}
int main()
{
    char s[100] = "abcdefg";
    char t[100];
    strcpy(t,s);
    reverse_string(s);
    if(strcmp(t,s) == 0)
        printf("Yes");
    else
        printf("No");
    return 0;
}

```

- a) Copies a string to another string
- b) Compares two strings
- c) Reverses a string
- d) Checks if a string is a palindrome

**Answer:** d

**Explanation:** The main purpose of the above code is to check if a string is a palindrome.

5. What is the output of the following code?

```
#include<stdio.h>
#include<string.h>
void reverse_string(char *s)
{
    int len = strlen(s);
    int i,j;
    i=0;
    j=len-1;
    while(i < j)
    {
        char tmp = s[i];
        s[i] = s[j];
        s[j] = tmp;
        i++;
        j--;
    }
}
int main()
{
    char s[100] = "rotator";
    char t[100];
    strcpy(t,s);
    reverse_string(s);
    if(strcmp(t,s) == 0)
        printf("Yes");
    else
        printf("No");
    return 0;
}
```

- a) Yes
- b) No
- c) Runtime error
- d) Compile time error

**Answer:** a

**Explanation:** The program checks if a string is a palindrome. Since the string rotator is a palindrome, it prints “yes”.

6. Consider the following recursive implementation used to reverse a string:

```
void recursive_reverse_string(char *s, int left, int right)
{
    if(left < right)
    {
        char tmp = s[left];
        s[left] = s[right];
        s[right] = tmp;
        _____;
    }
}
```

Which of the following lines should be inserted to complete the above code?

- a) recursive\_reverse\_string(s, left+1, right+1)
- b) recursive\_reverse\_string(s, left-1, right-1)
- c) recursive\_reverse\_string(s, left+1, right-1)
- d) recursive\_reverse\_string(s, left-1, right+1)

**Answer:** c

**Explanation:** The line “recursive\_reverse\_string(s, left+1, right-1)” should be inserted to complete the above code.

7. What is the output of the following code?

```
#include<stdio.h>
#include<string.h>
void recursive_reverse_string(char *s, int left, int right)
{
    if(left < right)
    {
        char tmp = s[left];
        s[left] = s[right];
        s[right] = tmp;
        recursive_reverse_string(s, left+1, right-1);
    }
}
int main()
{
    char s[100] = "recursion";
    int len = strlen(s);
    recursive_reverse_string(s,0,len-1);
    printf("%s",s);
    return 0;
}
```

- a) recursion
- b) nsoirucer
- c) noisrcuer
- d) noisrucer

**Answer:** d

**Explanation:** The program prints the reversed string of “recursion”, which is “noisrucer”.

8. How many times is the function recursive\_reverse\_string() called when the following code is executed?

```
#include<stdio.h>
#include<string.h>
void recursive_reverse_string(char *s, int left, int right)
{
    if(left < right)
    {
        char tmp = s[left];
        s[left] = s[right];
        s[right] = tmp;
        recursive_reverse_string(s, left+1, right-1);
    }
}
int main()
{
    char s[100] = "madam";
    int len = strlen(s);
    recursive_reverse_string(s, 0, len-1);
    printf("%s", s);
    return 0;
}

a) 3
b) 4
c) 5
d) 6
```

**Answer:** a

**Explanation:** The function recursive\_reverse\_string() is called 3 times when the above code is executed.

9. What is the time complexity of the above recursive implementation used to reverse a string?
- O(1)
  - O(n)
  - O( $n^2$ )
  - O( $n^3$ )

**Answer:** b

**Explanation:** The time complexity of the above recursive implementation used to reverse a string is O(n).

10. In which of the following cases is the reversal of a string not equal to the original string?
- Palindromic strings
  - Strings of length 1

- c) Empty String  
d) Strings of length 2

**Answer:** d

**Explanation:** String “ab” is a string of length 2 whose reversal is not the same as the given one. Palindromic Strings, String of length 1 and Empty Strings case – the reversal is the same as the one given.

---

1. Which of the following is the binary representation of 100?

- 1010010
- 1110000
- 1100100
- 1010101

**Answer:** c

**Explanation:**  $100 = 64 + 32 + 4 = 2^6 + 2^5 + 2^2 = 1100100$ .

2. Consider the following iterative code used to convert a decimal number to its equivalent binary:

```
#include<stdio.h>
void dec_to_bin(int n)
{
    int arr[31], len = 0, i;
    if(n == 0)
    {
        arr[0] = 0;
        len = 1;
    }
    while(n != 0)
    {
        arr[len++] = n % 2;
        _____;
    }
    for(i=len-1; i>=0; i--)
        printf("%d", arr[i]);
}
int main()
{
    int n = 10;
    dec_to_bin(n);
    return 0;
}
```

Which of the following lines should be inserted to complete the above code?

- a) n-
- b) n /= 2
- c) n /= 10
- d) n++

**Answer:** b

**Explanation:** The line “n /= 2” should be inserted to complete the above code.

3. What is the output of the following code?

```
#include<stdio.h>
void dec_to_bin(int n)
{
    int arr[31],len = 0,i;
    if(n == 0)
    {
        arr[0] = 0;
        len = 1;
    }
    while(n != 0)
    {
        arr[len++] = n % 2;
        n /= 2;
    }
    for(i=len-1; i>=0; i--)
        printf("%d",arr[i]);
}
int main()
{
    int n = 63;
    dec_to_bin(n);
    return 0;
}
```

- a) 111111
- b) 111011
- c) 101101
- d) 101010

**Answer:** a

**Explanation:** The program prints the binary equivalent of 63, which is 111111.

4. What is the output of the following code?

```
#include<stdio.h>
void dec_to_bin(int n)
{
    int arr[31],len = 0,i;
    if(n == 0)
```

```
{
    arr[0] = 0;
    len = 1;
}
while(n != 0)
{
    arr[len++] = n % 2;
    n /= 2;
}
for(i=len-1; i>=0; i--)
    printf("%d",arr[i]);
}
```

```
int main()
{
    int n = 0;
    dec_to_bin(n);
    return 0;
}
```

- a) 0
- b) 1
- c) Runtime error
- d) Garbage value

**Answer:** a

**Explanation:** The program prints the binary equivalent of 0, which is 0.

5. What is the time complexity of the following code used to convert a decimal number to its binary equivalent?

```
#include<stdio.h>
void dec_to_bin(int n)
{
    int arr[31],len = 0,i;
    if(n == 0)
    {
        arr[0] = 0;
        len = 1;
    }
    while(n != 0)
    {
        arr[len++] = n % 2;
        n /= 2;
    }
    for(i=len-1; i>=0; i--)
        printf("%d",arr[i]);
}
int main()
{
    int n = 0;
    dec_to_bin(n);
    return 0;
}
```

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) O(logn)

**Answer:** d

**Explanation:** The time complexity of the above code used to convert a decimal number to its binary equivalent is O(logn).

6. Consider the following recursive implementation used to convert a decimal number to its binary equivalent:

```
#include<stdio.h>
int arr[31], len = 0;
void recursive_dec_to_bin(int n)
{
    if(n == 0 && len == 0)
    {
        arr[len++] = 0;
        return;
    }
    if(n == 0)
        return;
    _____;
    recursive_dec_to_bin(n/2);
}
int main()
{
    int n = 100,i;
    recursive_dec_to_bin(n);
    for(i=len-1; i>=0; i--)
        printf("%d",arr[i]);
    return 0;
}
```

Which of the following lines should be inserted to complete the above code?

- a) arr[len] = n
- b) arr[len] = n % 2
- c) arr[len++] = n % 2
- d) arr[len++] = n

**Answer:** c

**Explanation:** The line “arr[len++] = n % 2” should be inserted to complete the above code.

7. Consider the following code:

```
#include<stdio.h>
int arr[31], len = 0;
void recursive_dec_to_bin(int n)
{
    if(n == 0 && len == 0)
    {
        arr[len++] = 0;
        return;
    }
    if(n == 0)
        return;
    arr[len++] = n % 2;
    recursive_dec_to_bin(n/2);
}
```

Which of the following lines is the base case for the above code?

- a) if(n == 0 && len == 0)
- b) if(n == 0)
- c) if(n == 0 && len == 0) and if(n == 0)
- d) if(n == 1)

**Answer:** c

**Explanation:** Both of the above mentioned lines are the base cases for the above code.

8. What is the output of the following code?

```
#include<stdio.h>
int arr[31], len = 0;
void recursive_dec_to_bin(int n)
{
    if(n == 0 && len == 0)
    {
        arr[len++] = 0;
        return;
    }
    if(n == 0)
        return;
    arr[len++] = n % 2;
    recursive_dec_to_bin(n/2);
}
int main()
{
    int n = -100,i;
    recursive_dec_to_bin(n);
    for(i=len-1; i>=0; i--)
        printf("%d",arr[i]);
    return 0;
}
```

- a) -1100100
- b) 1100100

- c) 2's complement of 1100100  
d) Garbage value

**Answer:** d

**Explanation:** The program doesn't handle negative inputs and so produces a garbage value.

9. What is the time complexity of the recursive implementation used to convert a decimal number to its binary equivalent?

```
#include<stdio.h>
int arr[31], len = 0;
void recursive_dec_to_bin(int n)
{
    if(n == 0 && len == 0)
    {
        arr[len++] = 0;
        return;
    }
    if(n == 0)
        return;
    arr[len++] = n % 2;
    recursive_dec_to_bin(n/2);
}
```

a) O(1)  
b) O(n)  
c) O( $n^2$ )  
d) O(logn)

**Answer:** d

**Explanation:** The time complexity of the recursive implementation used to convert a decimal number to its binary equivalent is O(logn).

10. What is the space complexity of the recursive implementation used to convert a decimal number to its binary equivalent?

```
#include<stdio.h>
int arr[31], len = 0;
void recursive_dec_to_bin(int n)
{
    if(n == 0 && len == 0)
    {
        arr[len++] = 0;
        return;
    }
    if(n == 0)
```

```
        return;
    arr[len++] = n % 2;
    recursive_dec_to_bin(n/2);
}
```

- a) O(1)  
b) O(n)  
c) O( $n^2$ )  
d) O(logn)

**Answer:** d

**Explanation:** The space complexity of the recursive implementation used to convert a decimal number to its binary equivalent is O(logn).

11. What is the output of the following code?

```
#include<stdio.h>
int arr[31], len = 0;
void recursive_dec_to_bin(int n)
{
    if(n == 0 && len == 0)
    {
        arr[len++] = 0;
        return;
    }
    if(n == 0)
        return;
    arr[len++] = n % 2;
    recursive_dec_to_bin(n/2);
}
int main()
{
    int n = 111, i;
    recursive_dec_to_bin(n);
    for(i=len-1; i>=0; i--)
        printf("%d", arr[i]);
    return 0;
}
```

- a) 1110111  
b) 1001111  
c) 1101111  
d) 1010111

**Answer:** c

**Explanation:** The program prints the binary equivalent of 111, which is 1101111.

12. How many times is the function recursive\_dec\_to\_bin() called when the

following code is executed?

```
#include<stdio.h>
int arr[31], len = 0;
void recursive_dec_to_bin(int n)
{
    if(n == 0 && len == 0)
    {
        arr[len++] = 0;
        return;
    }
    if(n == 0)
        return;
    arr[len++] = n % 2;
    recursive_dec_to_bin(n/2);
}
int main()
{
    int n = 111, i;
    recursive_dec_to_bin(n);
    for(i=len-1; i>=0; i--)
        printf("%d", arr[i]);
    return 0;
}
```

- a) 7
- b) 8
- c) 9
- d) 10

**Answer:** b

**Explanation:** The function recursive\_dec\_to\_bin() is called 8 times when the above code is executed.

1. Consider the following iterative implementation used to find the length of a linked list:

```
struct Node
{
    int val;
    struct Node *next;
}*head;
int get_len()
{
    struct Node *temp = head->next;
    int len = 0;
    while(____)
    {
        len++;
        temp = temp->next;
    }
}
```

```
    return len;
}
```

Which of the following conditions should be checked to complete the above code?

- a) temp->next != 0
- b) temp == 0
- c) temp != 0
- d) temp->next == 0

**Answer:** c

**Explanation:** The condition “temp != 0” should be checked to complete the above code.

2. What is the output of the following code?

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node *next;
}*head;
int get_len()
{
    struct Node *temp = head->next;
    int len = 0;
    while(temp != 0)
    {
        len++;
        temp = temp->next;
    }
    return len;
}
int main()
{
    int arr[10] = {1,2,3,4,5}, n = 5, i
;
    struct Node *temp, *newNode;
    head = (struct Node*)malloc(sizeof(
    struct Node));
    head->next = 0;
    temp = head;
    for(i=0; i<n; i++)
    {
        newNode = (struct Node*)malloc(
        sizeof(struct Node));
        newNode->val = arr[i];
        newNode->next = 0;
        temp->next = newNode;
        temp = temp->next;
    }
    int len = get_len();
    printf("%d", len);
}
```

```
    return 0;
}
```

- a) 4
- b) 5
- c) 6
- d) 7

**Answer:** b

**Explanation:** The program prints the length of the linked list, which is 5.

3. What is the time complexity of the following iterative implementation used to find the length of a linked list?

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node *next;
}*head;
int get_len()
{
    struct Node *temp = head->next;
    int len = 0;
    while(temp != 0)
    {
        len++;
        temp = temp->next;
    }
    return len;
}
int main()
{
    int arr[10] = {1,2,3,4,5}, n = 5, i
;
    struct Node *temp, *newNode;
    head = (struct Node*)malloc(sizeof(
struct Node));
    head->next = 0;
    temp = head;
    for(i=0; i<n; i++)
    {
        newNode = (struct Node*)malloc(
sizeof(struct Node));
        newNode->val = arr[i];
        newNode->next = 0;
        temp->next = newNode;
        temp = temp->next;
    }
    int len = get_len();
    printf("%d",len);
}
```

```
    return 0;
}
```

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) O(logn)

**Answer:** b

**Explanation:** The time complexity of the above iterative implementation used to find the length of a linked list is O(n).

4. What is the output of the following code?

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node *next;
}*head;
int get_len()
{
    struct Node *temp = head->next;
    int len = 0;
    while(temp != 0)
    {
        len++;
        temp = temp->next;
    }
    return len;
}
int main()
{
    int arr[10] = {1,2,3,4,5}, n = 5, i
;
    struct Node *temp, *newNode;
    head = (struct Node*)malloc(sizeof(
struct Node));
    head->next = 0;
    int len = get_len();
    printf("%d",len);
    return 0;
}
```

- a) 0
- b) Garbage value
- c) Compile time error
- d) Runtime error

**Answer:** a

**Explanation:** The program prints the length

of the linked list, which is 0.

5. Which of the following can be the base case for the recursive implementation used to find the length of linked list?

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node *next;
}*head;
int get_len()
{
    struct Node *temp = head->next;
    int len = 0;
    while(temp != 0)
    {
        len++;
        temp = temp->next;
    }
    return len;
}
int main()
{
    int arr[10] = {1,2,3,4,5}, n = 5, i;
    struct Node *temp, *newNode;
    head = (struct Node*)malloc(sizeof(struct Node));
    head->next = 0;
    int len = get_len();
    printf("%d",len);
    return 0;
}
```

- a) if(current\_node == 0) return 1
- b) if(current\_node->next == 0) return 1
- c) if(current\_node->next == 0) return 0
- d) if(current\_node == 0) return 0

**Answer:** d

**Explanation:** The line “if(current\_node == 0) return 0” can be used as a base case in the recursive implementation used to find the length of a linked list. Note: The line “if(current\_node->next == 0) return 1” cannot be used because it won’t work when the length of the linked list is zero.

6. Which of the following lines should be inserted to complete the following recursive

implementation used to find the length of a linked list?

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node *next;
}*head;
int recursive_get_len(struct Node *current_node)
{
    if(current_node == 0)
        return 0;
    return _____;
}
int main()
{
    int arr[10] = {1,2,3,4,5}, n = 5, i;
    struct Node *temp, *newNode;
    head = (struct Node*)malloc(sizeof(struct Node));
    head->next = 0;
    temp = head;
    for(i=0; i<n; i++)
    {
        newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->val = arr[i];
        newNode->next = 0;
        temp->next = newNode;
        temp = temp->next;
    }
    int len = recursive_get_len(head->next);
    printf("%d",len);
    return 0;
}
```

- a) recursive\_get\_len(current\_node)
- b) 1 + recursive\_get\_len(current\_node)
- c) recursive\_get\_len(current\_node->next)
- d) 1 + recursive\_get\_len(current\_node->next)

**Answer:** d

**Explanation:** The line “1 + recursive\_get\_len(current\_node->next)” should be inserted to complete the above code.

7. What is the output of the following code?

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node *next;
}*head;
int recursive_get_len(struct Node *current_node)
{
    if(current_node == 0)
        return 0;
    return 1 + recursive_get_len(current_node->next);
}
int main()
{
    int arr[10] = {-1,2,3,-3,4,5,0}, n
= 7, i;
    struct Node *temp, *newNode;
    head = (struct Node*)malloc(sizeof(struct Node));
    head->next = 0;
    temp = head;
    for(i=0; i<n; i++)
    {
        newNode = (struct Node*)malloc(
sizeof(struct Node));
        newNode->val = arr[i];
        newNode->next = 0;
        temp->next = newNode;
        temp = temp->next;
    }
    int len = recursive_get_len(head->next);
    printf("%d",len);
    return 0;
}
```

- a) 6
- b) 7
- c) 8
- d) 9

**Answer:** b

**Explanation:** The program prints the length of the linked list, which is 7.

8. What is the time complexity of the following code used to find the length of a linked list?

```
#include<stdio.h>
#include<stdlib.h>
struct Node
```

```
{
    int val;
    struct Node *next;
}*head;
int recursive_get_len(struct Node *current_node)
{
    if(current_node == 0)
        return 0;
    return 1 + recursive_get_len(current_node->next);
}
int main()
{
    int arr[10] = {-1,2,3,-3,4,5,0}, n
= 7, i;
    struct Node *temp, *newNode;
    head = (struct Node*)malloc(sizeof(struct Node));
    head->next = 0;
    temp = head;
    for(i=0; i<n; i++)
    {
        newNode = (struct Node*)malloc(
sizeof(struct Node));
        newNode->val = arr[i];
        newNode->next = 0;
        temp->next = newNode;
        temp = temp->next;
    }
    int len = recursive_get_len(head->next);
    printf("%d",len);
    return 0;
}
```

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) O( $n^3$ )

**Answer:** b

**Explanation:** To find the length of the linked list, the program iterates over the linked list once. So, the time complexity of the above code is O(n).

9. What is the output of the following code?

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
```

```

        struct Node *next;
}*head;
int recursive_get_len(struct Node *current
t_node)
{
    if(current_node == 0)
        return 0;
    return 1 + recursive_get_len(current
t_node->next);
}
int main()
{
    int arr[10] = {-1,2,3,-3,4,5}, n =
6, i;
    struct Node *temp, *newNode;
    head = (struct Node*)malloc(sizeof(
struct Node));
    head->next = 0;
    temp = head;
    for(i=0; i<n; i++)
    {
        newNode = (struct Node*)malloc(
sizeof(struct Node));
        newNode->val = arr[i];
        newNode->next = 0;
        temp->next = newNode;
        temp = temp->next;
    }
    int len = recursive_get_len(head->next);
    printf("%d",len);
    return 0;
}

```

- a) 5
- b) 6
- c) 7
- d) 8

**Answer:** b

**Explanation:** The program prints the length of the linked list, which is 6.

10. How many times is the function recursive\_get\_len() called when the following code is executed?

```

#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node *next;
}*head;
int recursive_get_len(struct Node *current

```

```

t_node)
{
    if(current_node == 0)
        return 0;
    return 1 + recursive_get_len(current
t_node->next);
}
int main()
{
    int arr[10] = {-1,2,3,-3,4,5}, n =
6, i;
    struct Node *temp, *newNode;
    head = (struct Node*)malloc(sizeof(
struct Node));
    head->next = 0;
    temp = head;
    for(i=0; i<n; i++)
    {
        newNode = (struct Node*)malloc(
sizeof(struct Node));
        newNode->val = arr[i];
        newNode->next = 0;
        temp->next = newNode;
        temp = temp->next;
    }
    int len = recursive_get_len(head->next);
    printf("%d",len);
    return 0;
}

```

- a) 5
- b) 6
- c) 7
- d) 8

**Answer:** c

**Explanation:** The function is called “len + 1” times. Since the length of the linked list in the above code is 6, the function is called  $6 + 1 = 7$  times.

---

1. Consider the following iterative implementation to find the length of the string:

```

#include<stdio.h>
int get_len(char *s)
{
    int len = 0;
    while(_____)
        len++;
    return len;
}

```

```

}

int main()
{
    char *s = "harsh";
    int len = get_len(s);
    printf("%d",len);
    return 0;
}

```

Which of the following lines should be inserted to complete the above code?

- a) s[len-1] != 0
- b) s[len+1] != 0
- c) s[len] != '\0'
- d) s[len] == '\0'

**Answer:** c

**Explanation:** The line “s[len] != ‘\0’” should be inserted to complete the above code.

2. What is the output of the following code?

```

#include<stdio.h>
int get_len(char *s)
{
    int len = 0;
    while(s[len] != '\0')
        len++;
    return len;
}
int main()
{
    char *s = "lengthofstring";
    int len = get_len(s);
    printf("%d",len);
    return 0;
}

a) 14
b) 0
c) Compile time error
d) Runtime error

```

**Answer:** a

**Explanation:** The program prints the length of the string “lengthofstring”, which is 14.

3. What is the time complexity of the following code used to find the length of the string?

```

#include<stdio.h>
int get_len(char *s)
{
    int len = 0;
    while(s[len] != '\0')
        len++;
    return len;
}
int main()
{
    char *s = "lengthofstring";
    int len = get_len(s);
    printf("%d",len);
    return 0;
}

a) O(1)
b) O(n)
c) O(n2)
d) O(logn)

```

**Answer:** b

**Explanation:** The time complexity of the code used to find the length of the string is O(n).

4. What is the output of the following code?

```

#include<stdio.h>
int get_len(char *s)
{
    int len = 0;
    while(s[len] != '\0')
        len++;
    return len;
}
int main()
{
    char *s = "";
    int len = get_len(s);
    printf("%d",len);
    return 0;
}

```

- a) 0
- b) 1
- c) Runtime error
- d) Garbage value

**Answer:** a

**Explanation:** The program prints the length of an empty string, which is 0.

5. Which of the following can be the base case for the recursive implementation used to find the length of a string?

```
#include<stdio.h>
int get_len(char *s)
{
    int len = 0;
    while(s[len] != '\0')
        len++;
    return len;
}
int main()
{
    char *s = "";
    int len = get_len(s);
    printf("%d",len);
    return 0;
}
```

- a) if(string[len] == 1) return 1
- b) if(string[len+1] == 1) return 1
- c) if(string[len] == '\0') return 0
- d) if(string[len] == '\0') return 1

**Answer:** c

**Explanation:** “if(string[len] == '\0') return 0” can be used as base case in the recursive implementation used to find the length of the string.

6. Consider the following recursive implementation used to find the length of a string:

```
#include<stdio.h>
int recursive_get_len(char *s, int len)
{
    if(s[len] == 0)
        return 0;
    return _____;
}
int main()
{
    char *s = "abcdef";
    int len = recursive_get_len(s,0);
    printf("%d",len);
    return 0;
}
```

Which of the following lines should be inserted to complete the above code?

- a) 1

- b) len
- c) recursive\_get\_len(s, len+1)
- d) 1 + recursive\_get\_len(s, len+1)

**Answer:** d

**Explanation:** The line “1 + recursive\_get\_len(s, len+1)” should be inserted to complete the code.

7. What is the output of the following code?

```
#include<stdio.h>
int recursive_get_len(char *s, int len)
{
    if(s[len] == 0)
        return 0;
    return 1 + recursive_get_len(s, len
+1);
}
int main()
{
    char *s = "abcdef";
    int len = recursive_get_len(s,0);
    printf("%d",len);
    return 0;
}
```

- a) 5
- b) 6
- c) 7
- d) 8

**Answer:** b

**Explanation:** The above code prints the length of the string “abcdef”, which is 6.

8. What is the time complexity of the following recursive implementation used to find the length of the string?

```
#include<stdio.h>
int recursive_get_len(char *s, int len)
{
    if(s[len] == 0)
        return 0;
    return 1 + recursive_get_len(s, len
+1);
}
int main()
{
    char *s = "abcdef";
    int len = recursive_get_len(s,0);
    printf("%d",len);
}
```

```

        return 0;
}

```

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) O( $n^3$ )

**Answer:** b

**Explanation:** The time complexity of the above recursive implementation used to find the length of the string is O(n).

9. How many times is the function recursive\_get\_len() called when the following code is executed?

```

#include<stdio.h>
int recursive_get_len(char *s, int len)
{
    if(s[len] == 0)
        return 0;
    return 1 + recursive_get_len(s, len
+1);
}
int main()
{
    char *s = "adghjkl";
    int len = recursive_get_len(s,0);
    printf("%d",len);
    return 0;
}

```

- a) 6
- b) 7
- c) 8
- d) 9

**Answer:** c

**Explanation:** The function recursive\_get\_len() is called 8 times when the above code is executed.

10. What is the output of the following code?

```

#include<stdio.h>
int recursive_get_len(char *s, int len)
{
    if(s[len] == 0)
        return 0;
    return 1 + recursive_get_len(s, len
+1);
}

```

```

}
int main()
{
    char *s = "123-1-2-3";
    int len = recursive_get_len(s,0);
    printf("%d",len);
    return 0;
}

```

- a) 3
- b) 6
- c) 9
- d) 10

**Answer:** c

**Explanation:** The above program prints the length of the string “123-1-2-3”, which is 9.

1. If Matrix A is of order X\*Y and Matrix B is of order M\*N, then what is the order of the Matrix A\*B given that Y=M?

- a) Y\*N
- b) X\*M
- c) X\*N
- d) Y\*M

**Answer:** c

**Explanation:** The Matrix A\*B is of order X\*N as it is given that Y=M i.e. number of columns in Matrix A is equal to total number of rows in matrix B. So the Matrix A\*B must have X number of rows and N number of columns.

2. How many recursive calls are there in Recursive matrix multiplication through Simple Divide and Conquer Method?

- a) 2
- b) 6
- c) 9
- d) 8

**Answer:** d

**Explanation:** For the multiplication two square matrix recursively using Simple Divide and Conquer Method, there are 8 recursive calls performed for high time complexity.

3. What is the time complexity of matrix multiplied recursively by Divide and Conquer Method?

- a)  $O(n)$
- b)  $O(n^2)$
- c)  $O(n^3)$
- d)  $O(n!)$

**Answer:** c

**Explanation:** The time complexity of recursive multiplication of two square matrices by the Divide and Conquer method is found to be  $O(n^3)$  since there are total of 8 recursive calls.

4. What is the time complexity of matrix multiplied recursively by Strassen's Method?

- a)  $O(n^{\log 7})$
- b)  $O(n^2)$
- c)  $O(n^3)$
- d)  $O(n!)$

**Answer:** a

**Explanation** The time complexity of recursive multiplication of two square matrices by Strassen's Method is found to be  $O(n^{\log 7})$  since there are total 7 recursive calls.

5. How many recursive calls are there in Recursive matrix multiplication by Strassen's Method?

- a) 5
- b) 7
- c) 8
- d) 4

**Answer:** b

**Explanation:** For the multiplication two square matrix recursively using Strassen's Method, there are 7 recursive calls performed for high time complexity.

6. Matrix A is of order 3\*4 and Matrix B is of order 4\*5. How many elements will be there in a matrix A\*B multiplied recursively.

- a) 12
- b) 15

- c) 16
- d) 20

**Answer:** b

**Explanation:** The resultant matrix will be of order 3\*5 when multiplied recursively and therefore the matrix will have  $3*5=15$  elements.

7. If Matrix X is of order A\*B and Matrix Y is of order C\*D, and B=C then the order of the Matrix X\*Y is A\*D?

- a) True
- b) False

**Answer:** a

**Explanation:** Given that B=C, so the two matrix can be recursively multiplied. Therefore, the order of the Matrix X\*Y is A\*D.

8. What is the time complexity of the fastest known matrix multiplication algorithm?

- a)  $O(n^{\log 7})$
- b)  $O(n^{2.37})$
- c)  $O(n^3)$
- d)  $O(n!)$

**Answer:** b

**Explanation:** The Coppersmith-Winograd algorithm multiplies the matrices in  $O(n^{2.37})$  time. Several improvements have been made in the algorithm since 2010.

9. Is Coppersmith-Winograd algorithm better than Strassen's algorithm in terms of time complexity?

- a) True
- b) False

**Answer:** a

**Explanation:** Since The Coppersmith-Winograd algorithm multiplies the matrices in  $O(n^{2.37})$  time. The time complexity of recursive multiplication of two square matrices by Strassen's Method is found to be  $O(n^{2.80})$ . Therefore, Coppersmith-Winograd

algorithm better than Strassen's algorithm in terms of time complexity.

1. Which of the following statement is true about stack?

- a) Pop operation removes the top most element
- b) Pop operation removes the bottom most element
- c) Push operation adds new element at the bottom
- d) Push operation removes the top most element

**Answer:** a

**Explanation:** As stack is based on LIFO(Last In First Out) principle so the deletion takes place from the topmost element. Thus pop operator removes topmost element.

2. What is the space complexity of program to reverse stack recursively?

- a) O(1)
- b) O(log n)
- c) O(n)
- d) O(n log n)

**Answer:** c

**Explanation:** The recursive program to reverse stack uses memory of the order n to store function call stack.

3. Stack can be reversed without using extra space by \_\_\_\_\_

- a) using recursion
- b) using linked list to implement stack
- c) using an extra stack
- d) it is not possible

**Answer:** b

**Explanation:** If linked list is used for implementing stack then it can be reversed without using any extra space.

4. Which of the following is considered as the top of the stack in the linked list implementation of the stack?

- a) Last node
- b) First node
- c) Random node
- d) Middle node

**Answer:** b

**Explanation:** First node is considered as the top element when stack is implemented using linked list.

5. What is the time complexity of the program to reverse stack when linked list is used for its implementation?

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

**Answer:** a

**Explanation:** As a linked list takes O(n) time for getting reversed thus linked list version of stack will also take the same time.

6. Which of the following takes O(n) time in worst case in array implementation of stack?

- a) pop
- b) push
- c) isEmpty
- d) pop, push and isEmpty takes constant time

**Answer:** d

**Explanation:** Functions pop, push and isEmpty all are implemented in constant time in worst case.

7. What will be the time complexity of the code to reverse stack recursively?

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

**Answer:** d

**Explanation:** The recurrence relation for the recursive code to reverse stack will be given by  $T(n)=T(n-1)+n$ . This is calculated to be equal to  $O(n^2)$ .

8. Which of the following functions correctly represents the recursive approach to reverse a stack?

a)

```
int reverse()
{
    if(s.size()<0)
    {
        int x = s.top();
        s.pop();
        reverse();
        BottomInsert(x);
    }
}
```

b)

```
int reverse()
{
    if(s.size()>=0)
    {
        int x = s.top();
        s.pop();
        reverse();
        BottomInsert(x);
    }
}
```

c)

```
int reverse()
{
    if(s.size()>0)
    {
        int x = s.top();
        s.pop();
        reverse();
        BottomInsert(x);
    }
}
```

d)

```
int reverse()
{
    if(s.size()>0)
    {
        int x = s.top();
        BottomInsert(x);
        s.pop();
        reverse();
    }
}
```

**Answer:** c

**Explanation:** We keep on holding the elements in call stack until we reach the bottom of the stack. Then we insert elements at the bottom. This reverses our stack.

9. Which of the following correctly represents the function to insert elements at the bottom of stack?

a)

```
int BottomInsert(int x)
{
    if(s.size()!=0) s.push(x);
    else
    {
        int a = s.top();
        s.pop();
        BottomInsert(x);
        s.push(a);
    }
}
```

b)

```
int BottomInsert(int x)
{
    if(s.size()==0) s.push(x);
    else
    {
        int a = s.top();
        s.pop();
        s.push(a);
        BottomInsert(x);
    }
}
```

c)

```
int BottomInsert(int x)
{
    if(s.size()==0) s.push(x);
    else
    {
        int a = s.top();
        s.pop();
        BottomInsert(x);
        s.push(a);
    }
}
```

d)

```

int BottomInsert(int x)
{
    if(s.size()==0) s.push(x);
    else
    {
        s.pop();
        int a = s.top();
        BottomInsert(x);
        s.push(a);
    }
}

```

**Answer:** c

**Explanation:** We hold all the elements in the call stack until we reach the bottom of stack and then the first if statement is executed as the stack is empty at this stage. Finally we push back all the elements held in the call stack.

10. Which of the following code correctly represents the function to reverse stack without using recursion?

a)

```

#include <stack>
void reverseStack(stack<int> &input, stack<int> &extra)
{
    while(input.size()!=0)
    {
        extra.push(input.top());
        input.pop();
    }
    input.swap(extra);
}

```

b)

```

#include <stack>
void reverseStack(stack<int> &input, stack<int> &extra)
{
    while(input.size()!=0)
    {
        extra.push(input.top());
        input.pop();
    }
    extra.swap(input);
}

```

c)

```

#include <stack>
void reverseStack(stack<int> &input, stack<int> &extra)
{
    while(input.size()!=0)
    {
        input.pop();
        extra.push(input.top());
    }
    extra.swap(input);
}

```

d)

```

#include <stack>
void reverseStack(stack<int> &input, stack<int> &extra)
{
    while(input.size()==0)
    {
        input.pop();
        extra.push(input.top());
    }
    extra.swap(input);
}

```

**Answer:** b

**Explanation:** We are using one extra stack to reverse the given stack. First the elements of the original stack are pushed into the other stack which creates a reversed version of the original stack. Then we swap this stack with the original stack.

1. Which of the following sorting algorithm has best case time complexity of  $O(n^2)$ ?

- a) bubble sort
- b) selection sort
- c) insertion sort
- d) stupid sort

**Answer:** b

**Explanation:** Selection sort is not an adaptive sorting algorithm. It finds the index of minimum element in each iteration even if the given array is already sorted. Thus its best case time complexity becomes  $O(n^2)$ .

2. Which of the following is the biggest advantage of selection sort?
- its has low time complexity
  - it has low space complexity
  - it is easy to implement
  - it requires only n swaps under any condition

**Answer:** d

**Explanation:** Selection sort works by obtaining least value element in each iteration and then swapping it with the current index. So it will take n swaps under any condition which will be useful when memory write operation is expensive.

3. What will be the recurrence relation of the code of recursive selection sort?
- $T(n) = 2T(n/2) + n$
  - $T(n) = 2T(n/2) + c$
  - $T(n) = T(n-1) + n$
  - $T(n) = T(n-1) + c$

**Answer:** c

**Explanation:** Function to find the minimum element index takes n time. The recursive call is made to one less element than in the previous call so the overall recurrence relation becomes  $T(n) = T(n-1) + n$ .

4. Which of the following sorting algorithm is NOT stable?
- Selection sort
  - Brick sort
  - Bubble sort
  - Merge sort

**Answer:** a

**Explanation:** Out of the given options selection sort is the only algorithm which is not stable. It is because the order of identical elements in sorted output may be different from input array.

5. What will be the best case time complexity of recursive selection sort?
- $O(n)$
  - $O(n^2)$

- $O(\log n)$
- $O(n \log n)$

**Answer:** b

**Explanation:** Selection sort's algorithm is such that it finds the index of minimum element in each iteration even if the given array is already sorted. Thus its best case time complexity becomes  $O(n^2)$ .

6. Recursive selection sort is a comparison based sort.
- true
  - false

**Answer:** a

**Explanation:** In selection sort we need to compare elements in order to find the minimum element in each iteration. So we can say that it uses comparisons in order to sort the array. Thus it qualifies as a comparison based sort.

7. What is the average case time complexity of recursive selection sort?
- $O(n)$
  - $O(n \log n)$
  - $O(n^2)$
  - $O(\log n)$

**Answer:** c

**Explanation:** The overall recurrence relation of recursive selection sort is given by  $T(n) = T(n-1) + n$ . It is found to be equal to  $O(n^2)$ . It is unvaried throughout the three cases.

8. What is the bidirectional variant of selection sort?
- cocktail sort
  - bogo sort
  - gnome sort
  - bubble sort

**Answer:** a

**Explanation:** A bidirectional variant of selection sort is called cocktail sort. It's an algorithm which finds both the minimum and maximum values in the array in every pass.

This reduces the number of scans of the array by a factor of 2.

9. Choose correct C++ code for recursive selection sort from the following.

a)

```
#include <iostream>
using namespace std;
int minIndex(int a[], int i, int j)
{
    if (i == 0)
        return i;
    int k = minIndex(a, i + 1, j);
    return (a[i] < a[k])? i : k;
}
void recursiveSelectionSort(int a[], int n, int index = 0)
{
    if (index == n)
        return;
    int x = minIndex(a, index, n-1);
    if (x == index)
    {
        swap(a[x], a[index]);
    }
    recursiveSelectionSort(a, n, inde
x + 1);
}
```

```
int main()
{
    int arr[] = {5,3,2,4,1};
    int n = sizeof(arr)/sizeof(arr[0])
);
```

```
recursiveSelectionSort(arr, n);
return 0;
}
```

b)

```
#include <iostream>
using namespace std;
int minIndex(int a[], int i, int j)
{
    if (i == j)
        return i;
    int k = minIndex(a, i + 1, j);
    return (a[i] < a[k])? i : k;
}
void recursiveSelectionSort(int a[], int n, int index = 0)
{
    if (index == n)
        return;
```

```
int x = minIndex(a, index, n-1);
if (x != index)
{
    swap(a[x], a[index]);
}
recursiveSelectionSort(a, n, inde
```

x + 1);
}

int main()

```
{
    int arr[] = {5,3,2,4,1};
    int n = sizeof(arr)/sizeof(arr[0])
);
```

```
recursiveSelectionSort(arr, n);
return 0;
}
```

c)

```
#include <iostream>
using namespace std;
int minIndex(int a[], int i, int j)
{
    if (i == j)
        return i;
    int k = minIndex(a, i + 1, j);
    return (a[i] > a[k])? i : k;
}
void recursiveSelectionSort(int a[], int n, int index = 0)
{
    if (index == n)
        return;
```

```
int x = minIndex(a, index, n-1);
if (x != index)
{
    swap(a[x], a[index]);
}
recursiveSelectionSort(a, n, inde
```

x + 1);
}

int main()

```
{
    int arr[] = {5,3,2,4,1};
    int n = sizeof(arr)/sizeof(arr[0])
);
```

```
recursiveSelectionSort(arr, n);
return 0;
}
```

d)

```
#include <iostream>
using namespace std;
int minIndex(int a[], int i, int j)
{
```

```

        if (i == j)
            return i;
        int k = minIndex(a, i + 1, j);
        return (a[i] > a[k])? i : k;
    }
void recursiveSelectionSort(int a[], int
n, int index = 0)
{
    if (index == 0)
        return;
    int x = minIndex(a, index, n-1);
    if (x == index)
    {
        swap(a[x], a[index]);
    }
    recursiveSelectionSort(a, n, inde
x + 1);
}
int main()
{
    int arr[] = {5,3,2,4,1};
    int n = sizeof(arr)/sizeof(arr[0])
);
    recursiveSelectionSort(arr, n);
    return 0;
}

```

**Answer:** b

**Explanation:** Using the function recursiveSelectionSort() we find the element that needs to be placed at the current index. For finding the minimum element index we use another function minIndex(). After finding the minimum element index the current element is swapped with this element in the function recursiveSelectionSort().

10. What is the number of swaps required to sort the array arr={5,3,2,4,1} using recursive selection sort?

- a) 0
- b) 1
- c) 2
- d) 3

**Answer:** c

**Explanation:** The first swap takes place between 1 and 5. The second swap takes place between 3 and 2 which sorts our array.

1. Which of the following methods can be used to find the largest and smallest element in an array?
  - a) Recursion
  - b) Iteration
  - c) Both recursion and iteration
  - d) No method is suitable

**Answer:** c

**Explanation:** Both recursion and iteration can be used to find the largest and smallest element in an array.

2. Consider the following iterative code snippet to find the largest element:

```

int get_max_element(int *arr,int n)
{
    int i, max_element = arr[0];
    for(i = 1; i < n; i++)
        if(_____)
            max_element = arr[i];
    return max_element;
}

```

Which of the following lines should be inserted to complete the above code?

- a) arr[i] > max\_element
- b) arr[i] < max\_element
- c) arr[i] == max\_element
- d) arr[i] != max\_element

**Answer:** a

**Explanation:** The line “arr[i] > max\_element” should be inserted to complete the above code snippet.

3. Consider the following code snippet to find the smallest element in an array:

```

int get_min_element(int *arr, int n)
{
    int i, min_element = arr[0];
    for(i = 1; i < n; i++)
        if(_____)
            min_element = arr[i];
    return min_element;
}

```

Which of the following lines should be inserted to complete the above code?

- a)  $\text{arr}[i] > \text{min\_element}$
- b)  $\text{arr}[i] < \text{min\_element}$
- c)  $\text{arr}[i] == \text{min\_element}$
- d)  $\text{arr}[i] != \text{min\_element}$

**Answer:** b

**Explanation:** The line “ $\text{arr}[i] < \text{min\_element}$ ” should be inserted to complete the above code.

4. What is the output of the following code?

```
#include<stdio.h>
int get_max_element(int *arr,int n)
{
    int i, max_element = arr[0];
    for(i = 1; i < n; i++)
        if(arr[i] > max_element)
            max_element = arr[i];
    return max_element;
}
int get_min_element(int *arr, int n)
{
    int i, min_element = arr[0];
    for(i = 1; i < n; i++)
        if(arr[i] < min_element)
            min_element = arr[i];
    return min_element;
}
int main()
{
    int n = 7, arr[7] = {1,1,1,0,-1,-1,-1};
    int max_element = get_max_element(ar
r,n);
    int min_element = get_min_element(ar
r,n);
    printf("%d %d",max_element,min_eleme
nt);
    return 0;
}
```

- a) 5 3
- b) 3 5
- c) 8 1
- d) 1 8

**Answer:** c

**Explanation:** The program prints the values of the largest and the smallest elements in the array, which are 8 and 1 respectively.

5. What is the output of the following code?

```
#include<stdio.h>
int get_max_element(int *arr,int n)
{
    int i, max_element = arr[0];
    for(i = 1; i < n; i++)
        if(arr[i] > max_element)
            max_element = arr[i];
    return max_element;
}
int get_min_element(int *arr, int n)
{
    int i, min_element;
    for(i = 1; i < n; i++)
        if(arr[i] < min_element)
            min_element = arr[i];
    return min_element;
}
int main()
{
    int n = 7, arr[7] = {1,1,1,0,-1,-1,-1};
    int max_element = get_max_element(ar
r,n);
    int min_element = get_min_element(ar
r,n);
    printf("%d %d",max_element,min_eleme
nt);
    return 0;
}
```

- a) 1 -1
- b) -1 1
- c) 1 Garbage value
- d) Depends on the compiler

**Answer:** c

**Explanation:** Since the `min_element` variable is not initialised, some compilers will auto initialise as 0 which produces -1 as output whereas some compilers won't initialise automatically. In that case, the result will be a garbage value hence the output depends on the compiler.

6. What is the time complexity of the following iterative implementation used to find the largest and smallest element in an array?

```
#include<stdio.h>
int get_max_element(int *arr,int n)
{
    int i, max_element = arr[0];
    for(i = 1; i < n; i++)
```

```

        if(arr[i] > max_element)
            max_element = arr[i];
        return max_element;
    }
    int get_min_element(int *arr, int n)
    {
        int i, min_element;
        for(i = 1; i < n; i++)
            if(arr[i] < min_element)
                min_element = arr[i];
        return min_element;
    }
    int main()
    {
        int n = 7, arr[7] = {1,1,1,0,-1,-1,-1};
        int max_element = get_max_element(ar
r,n);
        int min_element = get_min_element(ar
r,n);
        printf("%d %d",max_element,min_eleme
nt);
        return 0;
    }

```

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) O( $n/2$ )

**Answer:** b

**Explanation:** The time complexity of the above iterative implementation used to find the largest and the smallest element in an array is O(n).

7. Consider the following recursive implementation to find the largest element in an array.

```

int max_of_two(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int recursive_max_element(int *arr, int l
en, int idx)
{
    if(idx == len - 1)
        return arr[idx];
    return _____;
}

```

Which of the following lines should be inserted to complete the above code?

- a) max\_of\_two(arr[idx], recursive\_max\_element(arr, len, idx))
- b) recursive\_max\_element(arr, len, idx)
- c) max\_of\_two(arr[idx], recursive\_max\_element(arr, len, idx + 1))
- d) recursive\_max\_element(arr, len, idx + 1)

**Answer:** c

**Explanation:** The line “max\_of\_two(arr[idx], recursive\_max\_element(arr, len, idx + 1))” should be inserted to complete the above code.

8. What is the output of the following code?

```

#include<stdio.h>
int max_of_two(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int min_of_two(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int recursive_max_element(int *arr, int l
en, int idx)
{
    if(idx == len - 1)
        return arr[idx];
    return max_of_two(arr[idx], recursi
ve_max_element(arr, len, idx + 1));
}
int recursive_min_element(int *arr, int l
en, int idx)
{
    if(idx == len - 1)
        return arr[idx];
    return min_of_two(arr[idx], recursi
ve_min_element(arr, len, idx + 1));
}
int main()
{
    int n = 10, idx = 0, arr[] = {5,2,6,7
,8,9,3,-1,1,10};
    int max_element = recursive_max_eleme
nt(arr,n,idx);
    int min_element = recursive_min_eleme
nt(arr,n,idx);

```

```

        printf("%d %d",max_element,min_element);
    }
    return 0;
}

```

- a) -1 10
- b) 10 -1
- c) 1 10
- d) 10 1

**Answer:** b

**Explanation:** The program prints the values of the largest and the smallest element in the array, which are 10 and -1 respectively.

9. What is the time complexity of the following recursive implementation used to find the largest and the smallest element in an array?

```

#include<stdio.h>
int max_of_two(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int min_of_two(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int recursive_max_element(int *arr, int len, int idx)
{
    if(idx == len - 1)
        return arr[idx];
    return max_of_two(arr[idx], recursive_max_element(arr, len, idx + 1));
}
int recursive_min_element(int *arr, int len, int idx)
{
    if(idx == len - 1)
        return arr[idx];
    return min_of_two(arr[idx], recursive_min_element(arr, len, idx + 1));
}
int main()
{
    int n = 10, idx = 0, arr[] = {5,2,6,7,8,9,3,-1,1,10};
    int max_element = recursive_max_element(arr,n,idx);
    int min_element = recursive_min_element(arr,n,idx);
    printf("%d %d",max_element,min_element);
    return 0;
}

```

```

nt(arr,n,idx);
    int min_element = recursive_min_element(arr,n,idx);
    printf("%d %d",max_element,min_element);
}
return 0;
}

```

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) O( $n^3$ )

**Answer:** b

**Explanation:** The time complexity of the above recursive implementation used to find the largest and smallest element in an array is  $O(n)$ .

10. How many times is the function recursive\_min\_element() called when the following code is executed?

```

int min_of_two(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int recursive_min_element(int *arr, int len, int idx)
{
    if(idx == len - 1)
        return arr[idx];
    return min_of_two(arr[idx], recursive_min_element(arr, len, idx + 1));
}
int main()
{
    int n = 10, idx = 0, arr[] = {5,2,6,7,8,9,3,-1,1,10};
    int min_element = recursive_min_element(arr,n,idx);
    printf("%d",min_element);
    return 0;
}

a) 9
b) 10
c) 11
d) 12

```

**Answer:** b

**Explanation:** The function recursive\_min\_element() is called 10 times when the above code is executed.

11. What is the output of the following code?

```
#include<stdio.h>
int max_of_two(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int min_of_two(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int recursive_max_element(int *arr, int len, int idx)
{
    if(idx == len - 1)
        return arr[idx];
    return max_of_two(arr[idx], recursive_max_element(arr, len, idx + 1));
}
int recursive_min_element(int *arr, int len, int idx)
{
    if(idx == len - 1)
        return arr[idx];
    return min_of_two(arr[idx], recursive_min_element(arr, len, idx + 1));
}
int main()
{
    int n = 5, idx = 0, arr[] = {1,1,1,1,1};
    int max_element = recursive_max_element(arr,n,idx);
    int min_element = recursive_min_element(arr,n,idx);
    printf("%d %d",max_element,min_element);
    return 0;
}

a) 1 1
b) 0 0
c) compile time error
d) runtime error
```

**Answer:** a

**Explanation:** The program prints the values of the largest and the smallest element in the array, which are 1 and 1.

1. Which of the following methods can be used to find the largest and smallest number in a linked list?

- a) Recursion
- b) Iteration
- c) Both Recursion and iteration
- d) Impossible to find the largest and smallest numbers

**Answer:** c

**Explanation:** Both recursion and iteration can be used to find the largest and smallest number in a linked list.

2. Consider the following code snippet to find the largest element in a linked list:

```
struct Node{
    int val;
    struct Node *next;
}*head;
int get_max()
{
    struct Node* temp = head->next;
    int max_num = temp->val;
    while(____)
    {
        if(temp->val > max_num)
            max_num = temp->val;
        temp = temp->next;
    }
    return max_num;
}
```

Which of the following lines should be inserted to complete the above code?

- a) temp->next != 0
- b) temp != 0
- c) head->next != 0
- d) head != 0

**Answer:** b

**Explanation:** The line “temp != 0” should be inserted to complete the above code.

3. Consider the following code snippet to find the smallest element in a linked list:

```
struct Node
{
    int val;
    struct Node* next;
}*head;
int get_min()
{
    struct Node* temp = head->next;
    int min_num = temp->val;
    while(temp != 0)
    {
        if(______)
            min_num = temp->val;
        temp = temp->next;
    }
    return min_num;
}
```

Which of the following lines should be inserted to complete the above code?

- a) temp > min\_num
- b) val > min\_min
- c) temp->val < min\_num
- d) temp->val > min\_num

**Answer:** c

**Explanation:** The line “temp->val = min\_num” should be inserted to complete the above code.

4. What is the output of the following code:

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node* next;
}*head;
int get_max()
{
    struct Node* temp = head->next;
    int max_num = temp->val;
    while(temp != 0)
    {
        if(temp->val > max_num)
            max_num = temp->val;
        temp = head->next;
    }
    return max_num;
}
```

```
int main()
{
    int n = 9, arr[9] ={5,1,3,4,5,2,3,3
,1},i;
    struct Node *temp, *newNode;
    head = (struct Node*)malloc(sizeof(
struct Node));
    head -> next = 0;
    temp = head;
    for(i=0;i<n;i++)
    {
        newNode =(struct Node*)malloc(s
izeof(struct Node));
        newNode->next = 0;
        newNode->val = arr[i];
        temp->next =newNode;
        temp = temp->next;
    }
    int max_num = get_max();
    printf("%d %d",max_num);
    return 0;
}
```

- a) 5
- b) 1
- c) runtime error
- d) garbage value

**Answer:** c

**Explanation:** The variable temp will always point to the first element in the linked list due to the line “temp = head->next” in the while loop. So, it will be an infinite while loop and the program will produce a runtime error.

5. What is the output of the following code?

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node* next;
}*head;
int get_max()
{
    struct Node* temp = head->next;
    int max_num = temp->val;
    while(temp != 0)
    {
        if(temp->val > max_num)
            max_num = temp->val;
        temp = temp->next;
    }
    return max_num;
}
```

```

}

int get_min()
{
    struct Node* temp = head->next;
    int min_num = temp->val;
    while(temp != 0)
    {
        if(temp->val < min_num)
            min_num = temp->val;
        temp = temp->next;
    }
    return min_num;
}

int main()
{
    int i, n = 9, arr[9] ={8,3,3,4,5,2,
5,6,7};
    struct Node *temp, *newNode;
    head = (struct Node*)malloc(sizeof(
struct Node));
    head -> next =0;
    temp = head;
    for(i=0;i<n;i++)
    {
        newNode =(struct Node*)malloc(s
izeof(struct Node));
        newNode->next = 0;
        newNode->val = arr[i];
        temp->next =newNode;
        temp = temp->next;
    }
    int max_num = get_max();
    int min_num = get_min();
    printf("%d %d",max_num,min_num);
    return 0;
}

```

- a) 2 2
- b) 8 8
- c) 2 8
- d) 8 2

**Answer:** d

**Explanation:** The program prints the largest and smallest elements in the linked list, which are 8 and 2 respectively.

6. What is the time complexity of the following iterative code used to find the smallest and largest element in a linked list?

```

#include<stdio.h>
#include<stdlib.h>
struct Node
{

```

```

        int val;
        struct Node* next;
    }*head;
int get_max()
{
    struct Node* temp = head->next;
    int max_num = temp->val;
    while(temp != 0)
    {
        if(temp->val > max_num)
            max_num = temp->val;
        temp = temp->next;
    }
    return max_num;
}

int get_min()
{
    struct Node* temp = head->next;
    int min_num = temp->val;
    while(temp != 0)
    {
        if(temp->val < min_num)
            min_num = temp->val;
        temp = temp->next;
    }
    return min_num;
}

int main()
{
    int i, n = 9, arr[9] ={8,3,3,4,5,2,
5,6,7};
    struct Node *temp, *newNode;
    head = (struct Node*)malloc(sizeof(
struct Node));
    head -> next =0;
    temp = head;
    for(i=0;i<n;i++)
    {
        newNode =(struct Node*)malloc(s
izeof(struct Node));
        newNode->next = 0;
        newNode->val = arr[i];
        temp->next =newNode;
        temp = temp->next;
    }
    int max_num = get_max();
    int min_num = get_min();
    printf("%d %d",max_num,min_num);
    return 0;
}

a) O(1)
b) O(n)
c) O(n2)
d) O(n3)

```

**Answer:** b

**Explanation:** The time complexity of the above iterative code used to find the largest and smallest element in a linked list is O(n).

7. Consider the following recursive implementation to find the largest element in a linked list:

```
struct Node
{
    int val;
    struct Node* next;
}*head;
int max_of_two(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int recursive_get_max(struct Node* temp)
{
    if(temp->next == 0)
        return temp->val;
    return max_of_two(_____, _____);
}
```

Which of the following arguments should be passed to the function max\_of\_two() to complete the above code?

- a) temp->val, recursive\_get\_max(temp->next)
- b) temp, temp->next
- c) temp->val, temp->next->val
- d) temp->next->val, temp

**Answer:** a

**Explanation:** The arguments {temp->val, recursive\_get\_max(temp->next)} should be passed to the function max\_of\_two() to complete the above code.

8. What is the output of the following code?

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node* next;
}*head;
int max_of_two(int a, int b)
{
    if(a > b)
```

```
        return a;
    return b;
}
int recursive_get_max(struct Node* temp)
{
    if(temp->next == 0)
        return temp->val;
    return max_of_two(temp->val, recursive_get_max(temp->next));
}
int min_of_two(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int recursive_get_min(struct Node* temp)
{
    if(temp->next == 0)
        return temp->val;
    return min_of_two(temp->val, recursive_get_min(temp->next));
}
int main()
{
    int n = 9, arr[9] ={1,3,2,4,5,0,5,6,7},i;
    struct Node *temp, *newNode;
    head = (struct Node*)malloc(sizeof(struct Node));
    head -> next =0;
    temp = head;
    for(i=0;i<n;i++)
    {
        newNode =(struct Node*)malloc(sizeof(struct Node));
        newNode->next = 0;
        newNode->val = arr[i];
        temp->next =newNode;
        temp = temp->next;
    }
    int max_num = recursive_get_max(head->next);
    int min_num = recursive_get_min(head->next);
    printf("%d %d",max_num,min_num);
    return 0;
}
```

- a) 7 1
- b) 0 7
- c) 7 0
- d) 1 1

**Answer:** c

**Explanation:** The program prints the largest

and the smallest elements in the linked list, which are 7 and 0 respectively.

9. What is the time complexity of the recursive implementation used to find the largest and smallest element in a linked list?

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node* next;
}*head;
int max_of_two(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int recursive_get_max(struct Node* temp)
{
    if(temp->next == 0)
        return temp->val;
    return max_of_two(temp->val,recursive_get_max(temp->next));
}
int min_of_two(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int recursive_get_min(struct Node* temp)
{
    if(temp->next == 0)
        return temp->val;
    return min_of_two(temp->val,recursive_get_min(temp->next));
}
int main()
{
    int n = 9, arr[9] ={1,3,2,4,5,0,5,6,7},i;
    struct Node *temp, *newNode;
    head = (struct Node*)malloc(sizeof(struct Node));
    head -> next =0;
    temp = head;
    for(i=0;i<n;i++)
    {
        newNode =(struct Node*)malloc(sizeof(struct Node));
        newNode->next = 0;
        newNode->val = arr[i];
        temp->next =newNode;
        temp =temp->next;
    }
}
```

```
        temp = temp->next;
    }
    int max_num = recursive_get_max(head->next);
    int min_num = recursive_get_min(head->next);
    printf("%d %d",max_num,min_num);
    return 0;
}
```

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) O( $n^3$ )

**Answer:** b

**Explanation:** The time complexity of the above recursive implementation used to find the largest and smallest element in linked list is O(n).

10. What is the output of the following code?

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node* next;
}*head;
int min_of_two(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int recursive_get_min(struct Node* temp)
{
    if(temp->next == 0)
        return temp->val;
    return min_of_two(temp->val,recursive_get_min(temp->next));
}
int main()
{
    int n = 5, arr[5] ={1,1,1,1,1},i;
    struct Node *temp, *newNode;
    head = (struct Node*)malloc(sizeof(struct Node));
    head -> next =0;
    temp = head;
    for(i=0;i<n;i++)
    {
        newNode =(struct Node*)malloc(sizeof(struct Node));
        newNode->next = 0;
        newNode->val = arr[i];
        temp->next =newNode;
        temp =temp->next;
    }
}
```

```

        sizeof(struct Node));
        newNode->next = 0;
        newNode->val = arr[i];
        temp->next =newNode;
        temp = temp->next;
    }
    int min_num = recursive_get_min(head
->next);
    printf("%d",min_num);
    return 0;
}

```

- a) 1
- b) 0
- c) compile time error
- d) runtime error

**Answer:** a

**Explanation:** The program prints the smallest element in the linked list, which is 1.

11. How many times will the function recursive\_get\_min() be called when the following code is executed?

```

#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node* next;
}*head;
int min_of_two(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int recursive_get_min(struct Node* temp)
{
    if(temp->next == 0)
        return temp->val;
    return min_of_two(temp->val,recursive_get_min(temp->next));
}
int main()
{
    int n = 5, arr[5] ={1,1,1,1,1},i;
    struct Node *temp, *newNode;
    head = (struct Node*)malloc(sizeof(struct Node));
    head -> next =0;
    temp = head;
    for(i=0;i<n;i++)
    {

```

```

        newNode =(struct Node*)malloc(
        sizeof(struct Node));
        newNode->next = 0;
        newNode->val = arr[i];
        temp->next =newNode;
        temp = temp->next;
    }
    int min_num = recursive_get_min(head
->next);
    printf("%d",min_num);
    return 0;
}

```

- a) 4
- b) 5
- c) 6
- d) 7

**Answer:** b

**Explanation:** The function recursive\_get\_min() will be called 5 times when the above code is executed.

1. Which of the following techniques can be used to search an element in an unsorted array?

- a) Iterative linear search
- b) Recursive binary search
- c) Iterative binary search
- d) Normal binary search

**Answer:** a

**Explanation:** Iterative linear search can be used to search an element in an unsorted array.

Note: Binary search can be used only when the array is sorted.

2. Consider the array {1,1,1,1,1}. Select the wrong option?

- a) Iterative linear search can be used to search for the elements in the given array
- b) Recursive linear search can be used to search for the elements in the given array
- c) Recursive binary search can be used to search for the elements in the given array
- d) No method is defined to search for an element in the given array

**Answer:** d

**Explanation:** Iterative linear search, Recursive linear search and Recursive binary search can be applied to search for an element in the above given array.

3. What does the following code do?

```
#include<stdio.h>
int search_num(int *arr, int num, int len)
{
    int i;
    for(i = 0; i < len; i++)
        if(arr[i] == num)
            return i;
        return -1;
}
int main()
{
    int arr[5] ={1,2,3,4,5},num=3,len =
5;
    int indx = search_num(arr,num,len);
    printf("Index of %d is %d",num,indx
);
    return 0;
}
```

- a) Search and returns the index of all the occurrences of the number that is searched
- b) Search and returns the index of the first occurrence of the number that is searched
- c) Search and returns of the last occurrence of the number that is searched
- d) Returns the searched element from the given array

**Answer:** b

**Explanation:** The code finds the index of the first occurrence of the number that is searched.

4. What is the output of the following code?

```
#include<stdio.h>
int search_num(int *arr, int num, int len
)
{
    int i;
    for(i = 0; i < len; i++)
        if(arr[i] == num)
            return i;
        return -1;
```

```
}
int main()
{
    int arr[5] ={1,3,3,3,5},num=3,len =
5;
    int indx = search_num(arr,num,len);
    printf("Index of %d is %d",num,indx
);
    return 0;
}
```

- a) Index of 3 is 0
- b) Index of 3 is 1
- c) Index of 3 is 2
- d) Index of 3 is 3

**Answer:** b

**Explanation:** The program prints the index of the first occurrence of 3, which is 1.

5. What is the time complexity of the following code used to search an element in an array?

```
#include<stdio.h>
int search_num(int *arr, int num, int len
)
{
    int i;
    for(i = 0; i < len; i++)
        if(arr[i] == num)
            return i;
        return -1;
}
int main()
{
    int arr[5] ={1,3,3,3,5},num=3,len =
5;
    int indx = search_num(arr,num,len);
    printf("Index of %d is %d",num,indx
);
    return 0;
}
```

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) O( $n^3$ )

**Answer:** b

**Explanation:** The time complexity of the

above code used to search an element in an array is O(n).

6. Consider the following recursive implementation of linear search:

```
#include<stdio.h>
int recursive_search_num(int *arr, int num, int idx, int len)
{
    if(idx == len)
        return -1;
    if(arr[idx] == num)
        return idx;
    return _____;
}
int main()
{
    int arr[5] ={1,3,3,3,5},num=2,len = 5;
    int indx = recursive_search_num(arr,num,0,len);
    printf("Index of %d is %d",num,indx);
    return 0;
}
```

Which of the following recursive calls should be added to complete the above code?

- a) recursive\_search\_num(arr, num+1, idx, len);
- b) recursive\_search\_num(arr, num, idx, len);
- c) recursive\_search\_num(arr, num, idx+1, len);
- d) recursive\_search\_num(arr, num+1, idx+1, len);

**Answer:** c

**Explanation:** The recursive call “recursive\_search\_num(arr, num, idx+1, len)” should be added to complete the above code.

7. What is the output of the following code?

```
#include<stdio.h>
int recursive_search_num(int *arr, int num, int idx, int len)
{
    if(idx == len)
        return -1;
    if(arr[idx] == num)
        return idx;
    return recursive_search_num(arr, num
```

```
, idx+1, len);
}
int main()
{
    int arr[8] ={1,2,3,3,3,5,6,7},num=5
    ,len = 8;
    int indx = recursive_search_num(arr
    ,num,0,len);
    printf("Index of %d is %d",num,indx
    );
    return 0;
}
```

- a) Index of 5 is 5
- b) Index of 5 is 6
- c) Index of 5 is 7
- d) Index of 5 is 8

**Answer:** a

**Explanation:** The program prints the index of 5, which is 5.

8. How many times is the function recursive\_search\_num() called when the following code is executed?

```
#include<stdio.h>
int recursive_search_num(int *arr, int num, int idx, int len)
{
    if(idx == len)
        return -1;
    if(arr[idx] == num)
        return idx;
    return recursive_search_num(arr, num
    , idx+1, len);
}
int main()
{
    int arr[8] ={1,2,3,3,3,5,6,7},num=5
    ,len = 8;
    int indx = recursive_search_num(arr
    ,num,0,len);
    printf("Index of %d is %d",num,indx
    );
    return 0;
}
```

- a) 5
- b) 6
- c) 7
- d) 8

**Answer:** b

**Explanation:** The function recursive\_search\_num() is called 6 times when the above code is executed.

9. What is the time complexity of the following recursive implementation of linear search?

```
#include<stdio.h>
int recursive_search_num(int *arr, int num, int idx, int len)
{
    if(idx == len)
        return -1;
    if(arr[idx] == num)
        return idx;
    return recursive_search_num(arr, num, idx+1, len);
}
int main()
{
    int arr[8] = {1,2,3,3,3,5,6,7},num=5
    ,len = 8;
    int indx = recursive_search_num(arr, num, 0, len);
    printf("Index of %d is %d",num,indx);
    return 0;
}
```

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) O( $n^3$ )

**Answer:** b

**Explanation:** The time complexity of the above recursive implementation of linear search is O(n).

1. What is the output of the following code?

```
#include<stdio.h>
int recursive_search_num(int *arr, int num, int idx, int len)
{
    if(idx == len)
        return -1;
    if(arr[idx] == num)
        return idx;
    return recursive_search_num(arr, num,
```

```
, idx+1, len);
}
int main()
{
    int arr[8] = {-11,2,-3,0,3,5,-6,7},num = -2,len = 8;
    int indx = recursive_search_num(arr, num, 0, len);
    printf("Index of %d is %d",num,indx);
    return 0;
}
```

- a) Index of -2 is 1
- b) Index of -2 is 0
- c) Index of -2 is -1
- d) None of the mentioned

**Answer:** c

**Explanation:** The program prints the index of the first occurrence of -2. Since, -2 doesn't exist in the array, the program prints -1.

2. What does the following code do?

```
#include<stdio.h>
int cnt = 0;
int recursive_search_num(int *arr, int num, int idx, int len)
{
    int cnt = 0;
    if(idx == len)
        return cnt;
    if(arr[idx] == num)
        cnt++;
    return cnt + recursive_search_num(arr, num, idx+1, len);
}
int main()
{
    int arr[8] = {0,0,0,0,3,5,-6,7},num = 0,len = 8;
    int ans = recursive_search_num(arr, num, 0, len);
    printf("%d",ans);
    return 0;
}
```

- a) Adds all the indexes of the number 0
- b) Finds the first last occurrence of the number 0
- c) Counts the number of occurrences of the number 0
- d) None of the mentioned

**Answer:** c

**Explanation:** The above code counts the number of occurrences of the number 0.

3. Consider the following recursive implementation of the binary search:

```
#include<stdio.h>
int recursive_binary_search(int *arr, int num, int lo, int hi)
{
    if(lo > hi)
        return -1;
    int mid = (lo + hi)/2;
    if(arr[mid] == num)
        return mid;
    else if(arr[mid] < num)
        _____;
    else
        hi = mid - 1;
    return recursive_binary_search(arr, num, lo, hi);
}
int main()
{
    int arr[8] ={0,0,0,0,3,5,6,7},num =
7,len = 8;
    int indx = recursive_binary_search(
arr,num,0,len-1);
    printf("Index of %d is %d",num,indx
);
    return 0;
}
```

Which of the following lines should be added to complete the above code?

- a)  $hi = mid - 1$
- b)  $mid = (lo + hi)/2$
- c)  $mid = lo - 1$
- d)  $lo = mid + 1$

**Answer:** d

**Explanation:** The line “ $lo = mid + 1$ ” should be added to complete the above code.

4. What is the output of the following code?

```
#include<stdio.h>
int recursive_binary_search(int *arr, int num, int lo, int hi)
{
    if(lo > hi)
        return -1;
    int mid = (lo + hi)/2;
```

```
        if(arr[mid] == num)
            return mid;
        else if(arr[mid] < num)
            lo = mid + 1;
        else
            hi = mid - 1;
    return recursive_binary_search(arr, num, lo, hi);
}
int main()
{
    int arr[8] = {1,2,3,4,5,6,7,8},num
= 7,len = 8;
    int indx = recursive_binary_search(
arr,num,0,len-1);
    printf("Index of %d is %d",num,indx
);
    return 0;
}
```

a) Index of 7 is 4

b) Index of 7 is 5

c) Index of 7 is 6

d) Index of 7 is 7

**Answer:** c

**Explanation:** The program prints the index of number 7, which is 6.

5. What is the output of the following code?

```
#include<stdio.h>
int recursive_binary_search(int *arr, int num, int lo, int hi)
{
    if(lo > hi)
        return -1;
    int mid = (lo + hi)/2;
    if(arr[mid] == num)
        return mid;
    else if(arr[mid] < num)
        lo = mid + 1;
    else
        hi = mid - 1;
    return recursive_binary_search(arr, num, lo, hi);
}
int main()
{
    int arr[8] = {0,0,0,0,3,5,6,7},num
= 0,len = 8;
    int indx = recursive_binary_search(
arr,num,0,len-1);
    printf("Index of %d is %d",num,indx
);
```

```

        return 0;
}

```

- a) Index of 0 is 0
- b) Index of 0 is 1
- c) Index of 0 is 2
- d) Index of 0 is 3

**Answer:** d

**Explanation:** In this case, when the function recursive\_binary\_search() is called for the first time we have: lo = 0 and hi = 7. So, the value of mid is:

$\text{mid} = (\text{lo} + \text{hi})/2 = (0 + 7)/2 = 3$ . Since, arr[mid] = arr[3] = 0, the function returns the value of mid, which is 3.

6. What is the time complexity of the above recursive implementation of binary search?

- a) O(n)
- b) O( $2^n$ )
- c) O(logn)
- d) O(n!)

**Answer:** c

**Explanation:** The time complexity of the above recursive implementation of binary search is O(logn).

7. In which of the below cases will the following code produce a wrong output?

```

int recursive_binary_search(int *arr, int
num, int lo, int hi)
{
    if(lo > hi)
        return -1;
    int mid = (lo + hi)/2;
    if(arr[mid] == num)
        return mid;
    else if(arr[mid] < num)
        lo = mid + 1;
    else
        hi = mid - 1;
    return recursive_binary_search(arr,
num, lo, hi);
}

```

- a) Array: {0,0,0,0,0,0} Search: -10
- b) Array: {1,2,3,4,5} Search: 0

- c) Array: {5,4,3,2,1} Search: 1
- d) Array: {-5,-4,-3,-2,-1} Search: -1

**Answer:** c

**Explanation:** Since the array {5,4,3,2,1} is sorted in descending order, it will produce a wrong output.

8. How many times is the function recursive\_binary\_search() called when the following code is executed?

```

#include<stdio.h>
int recursive_binary_search(int *arr, int
num, int lo, int hi)
{
    if(lo > hi)
        return -1;
    int mid = (lo + hi)/2;
    if(arr[mid] == num)
        return mid;
    else if(arr[mid] < num)
        lo = mid + 1;
    else
        hi = mid - 1;
    return recursive_binary_search(arr,
num, lo, hi);
}
int main()
{
    int arr[5] = {1,2,3,4,5},num = 1,len
= 5;
    int indx = recursive_binary_search(
arr,num,0,len-1);
    printf("Index of %d is %d",num,indx
);
    return 0;
}

```

- a) 0
- b) 1
- c) 2
- d) 3

**Answer:** c

**Explanation:** The function recursive\_binary\_search() is called 2 times, when the above code is executed.

9. What is the output of the following code?

```

#include<stdio.h>
int recursive_binary_search(int *arr, int

```

```

num, int lo, int hi)
{
    if(lo > hi)
        return -1;
    int mid = (lo + hi)/2;
    if(arr[mid] == num)
        return mid;
    else if(arr[mid] < num)
        lo = mid + 1;
    else
        hi = mid - 1;
    return recursive_binary_search(arr,
num, lo, hi);
}
int main()
{
    int arr[5] = {5,4,3,2,1},num = 1,length = 5;
    int indx = recursive_binary_search(
arr,num,0,length-1);
    printf("Index of %d is %d",num,indx);
    return 0;
}

```

- a) Index of 1 is 4
- b) Index of 1 is 5
- c) Index of 1 is -1
- d) Index of 1 is 0

**Answer:** c

**Explanation:** Since the array is sorted in descending order, the above implementation of binary search will not work for the given array. Even though 1 is present in the array, binary search won't be able to search it and it will produce -1 as an answer.

---

1. Which of the following methods can be used to search an element in a linked list?
- a) Iterative linear search
  - b) Iterative binary search
  - c) Recursive binary search
  - d) Normal binary search

**Answer:** a

**Explanation:** Iterative linear search can be used to search an element in a linked list. Binary search can be used only when the list is sorted.

2. Consider the following code snippet to search an element in a linked list:

```

struct Node
{
    int val;
    struct Node* next;
}*head;
int linear_search(int value)
{
    struct Node *temp = head->next;
    while(temp != 0)
    {
        if(temp->val == value)
            return 1;
        _____;
    }
    return 0;
}

```

Which of the following lines should be inserted to complete the above code?

- a) temp = next
- b) temp->next = temp
- c) temp = temp->next
- d) return 0

**Answer:** c

**Explanation:** The line “temp = temp->next” should be inserted to complete the above code.

3. What does the following code do?

```

#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node* next;
}*head;
int linear_search(int value)
{
    struct Node *temp = head->next;
    while(temp != 0)
    {
        if(temp->val == value)
            return 1;
        temp = temp->next;
    }
    return 0;
}
int main()
{

```

```

int arr[5] = {1,2,3,4,5};
int n = 5,i;
head = (struct Node*)malloc(sizeof(
struct Node));
head->next = 0;
struct Node *temp;
temp = head;
for(i=0; i<n; i++)
{
    struct Node *newNode = (struct
Node*)malloc(sizeof(struct Node));
    newNode->next = 0;
    newNode->val = arr[i];
    temp->next = newNode;
    temp = temp->next;
}
int ans = linear_search(60);
if(ans == 1)
printf("Found");
else
printf("Not found");
return 0;
}

```

- a) Finds the index of the first occurrence of a number in a linked list
- b) Finds the index of the last occurrence of a number in a linked list
- c) Checks if a number is present in a linked list
- d) Checks whether the given list is sorted or not

**Answer:** c

**Explanation:** The above code checks if a number is present in a linked list.

4. What is the output of the following code?

```

#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node* next;
}*head;
int linear_search(int value)
{
    struct Node *temp = head->next;
    while(temp != 0)
    {
        if(temp->val == value)
            return 1;
        temp = temp->next;
    }
}

```

```

    }
    return 0;
}
int main()
{
    int arr[5] = {1,2,3,4,5};
    int n = 5,i;
    head = (struct Node*)malloc(sizeof(s
truct Node));
    head->next = 0;
    struct Node *temp;
    temp = head;
    for(i=0; i<n; i++)
    {
        struct Node *newNode = (struct
Node*)malloc(sizeof(struct Node));
        newNode->next = 0;
        newNode->val = arr[i];
        temp->next = newNode;
        temp = temp->next;
    }
    int ans = linear_search(-1);
    if(ans == 1)
    printf("Found");
    else
    printf("Not found");
    return 0;
}

```

- a) Found
- b) Not found
- c) Compile time error
- d) Runtime error

**Answer:** b

**Explanation:** Since the number -1 is not present in the linked list, the program prints not found.

5. What is the time complexity of the following implementation of linear search on a linked list?

```

#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node* next;
}*head;
int linear_search(int value)
{
    struct Node *temp = head->next;
    while(temp != 0)
    {

```

```

        if(temp->val == value)
            return 1;
        temp = temp->next;
    }
    return 0;
}
int main()
{
    int arr[5] = {1,2,3,4,5};
    int n = 5,i;
    head = (struct Node*)malloc(sizeof(struct Node));
    head->next = 0;
    struct Node *temp;
    temp = head;
    for(i=0; i<n; i++)
    {
        struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->next = 0;
        newNode->val = arr[i];
        temp->next = newNode;
        temp = temp->next;
    }
    int ans = linear_search(-1);
    if(ans == 1)
        printf("Found");
    else
        printf("Not found");
    return 0;
}

```

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) O( $n^3$ )

**Answer:** b

**Explanation:** The time complexity of the above implementation of linear search on a linked list is O(n).

6. What is the output of the following code?

```

#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node* next;
}*head;
int linear_search(int value)
{
    struct Node *temp = head->next;

```

```

    while(temp->next != 0)
    {
        if(temp->val == value)
            return 1;
        temp = temp->next;
    }
    return 0;
}
int main()
{
    int arr[6] = {1,2,3,4,5,6};
    int n = 6,i;
    head = (struct Node*)malloc(sizeof(struct Node));
    head->next = 0;
    struct Node *temp;
    temp = head;
    for(i=0; i<n; i++)
    {
        struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->next = 0;
        newNode->val = arr[i];
        temp->next = newNode;
        temp = temp->next;
    }
    int ans = linear_search(60);
    if(ans == 1)
        printf("Found");
    else
        printf("Not found");
    return 0;
}

```

- a) Found
- b) Not found
- c) Compile time error
- d) Runtime error

**Answer:** b

**Explanation:** The condition in the while loop “temp->next == 0”, checks if the current element is the last element. If the current element is the last element, the value of the current element is not compared with the value to be searched. So, even though the number 6 is present in the linked list, it will print not found.

7. Can binary search be applied on a sorted linked list in O(Logn) time?

- a) No
- b) Yes

**Answer:** a

**Explanation:** Since linked list doesn't allow random access, binary search cannot be applied on a sorted linked list in O(Logn) time.

8. What will be time complexity when binary search is applied on a linked list?

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) O( $n^3$ )

**Answer:** b

**Explanation:** The time complexity will be O(n) when binary search is applied on a linked list.

9. Consider the following recursive implementation of linear search on a linked list:

```
struct Node
{
    int val;
    struct Node* next;
}*head;
int linear_search(struct Node *temp,int value)
{
    if(temp == 0)
        return 0;
    if(temp->val == value)
        return 1;
    return _____;
}
```

Which of the following lines should be inserted to complete the above code?

- a) 1
- b) 0
- c) linear\_search(temp, value)
- d) linear\_search(temp->next, value)

**Answer:** d

**Explanation:** The line “linear\_search(temp->next, value)”, should be inserted to complete the above code.

10. What is the output of the following code?

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node* next;
}*head;
int linear_search(struct Node *temp,int value)
{
    if(temp == 0)
        return 0;
    if(temp->val == value)
        return 1;
    return linear_search(temp->next, value);
}
int main()
{
    int arr[6] = {1,2,3,4,5,6};
    int n = 6,i;
    head = (struct Node*)malloc(sizeof(struct Node));
    head->next = 0;
    struct Node *temp;
    temp = head;
    for(i=0; i<n; i++)
    {
        struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->next = 0;
        newNode->val = arr[i];
        temp->next = newNode;
        temp = temp->next;
    }
    int ans = linear_search(head->next,6);
    if(ans == 1)
        printf("Found");
    else
        printf("Not found");
    return 0;
}
```

- a) Found
- b) Not found
- c) Compile time error
- d) Runtime error

**Answer:** a

**Explanation:** Since the element 6 is present in the linked list, the program prints “Found”.

11. How many times is the function linear\_search() called when the following

code is executed?

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node* next;
}*head;
int linear_search(struct Node *temp,int value)
{
    if(temp == 0)
        return 0;
    if(temp->val == value)
        return 1;
    return linear_search(temp->next, value);
}
int main()
{
    int arr[6] = {1,2,3,4,5,6};
    int n = 6,i;
    head = (struct Node*)malloc(sizeof(struct Node));
    head->next = 0;
    struct Node *temp;
    temp = head;
    for(i=0; i<n; i++)
    {
        struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->next = 0;
        newNode->val = arr[i];
        temp->next = newNode;
        temp = temp->next;
    }
    int ans = linear_search(head->next,6);
    if(ans == 1)
        printf("Found");
    else
        printf("Not found");
    return 0;
}
```

- a) 5
- b) 6
- c) 7
- d) 8

**Answer:** b

**Explanation:** The function linear\_search() is called 6 times when the above code is executed.

12. What is the time complexity of the following recursive implementation of linear search?

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int val;
    struct Node* next;
}*head;
int linear_search(struct Node *temp,int value)
{
    if(temp == 0)
        return 0;
    if(temp->val == value)
        return 1;
    return linear_search(temp->next, value);
}
int main()
{
    int arr[6] = {1,2,3,4,5,6};
    int n = 6,i;
    head = (struct Node*)malloc(sizeof(struct Node));
    head->next = 0;
    struct Node *temp;
    temp = head;
    for(i=0; i<n; i++)
    {
        struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->next = 0;
        newNode->val = arr[i];
        temp->next = newNode;
        temp = temp->next;
    }
    int ans = linear_search(head->next,6);
    if(ans == 1)
        printf("Found");
    else
        printf("Not found");
    return 0;
}

a) O(1)
b) O(n)
c) O(n2)
d) O(n3)
```

**Answer:** b

**Explanation:** The time complexity of the above recursive implementation of linear search is  $O(n)$ .

1. What will be the output for following code?

```
#include<stdio.h>
int func(int x, int y)
{
    if (y == 0)
        return 1;
    else if (y%2 == 0)
        return func(x, y/2)*func(
x, y/2);
    else
        return x*func(x, y/2)*fun
c(x, y/2);
}
int main()
{
    int x = 2;
    int y = 3;

    printf("%d", func(x, y));
    return 0;
}
```

- a) 9
- b) 6
- c) 8
- d) 5

**Answer:** c

**Explanation:** The given program calculates the value of  $x$  raised to power  $y$ . Thus  $2^3 = 8$ .

2. What will be the time complexity of the following code which raises an integer  $x$  to the power  $y$ ?

```
#include<stdio.h>
int power(int x, int y)
{
    if (y == 0)
        return 1;
    else if (y%2 == 0)
        return power(x, y/2)*powe
r(x, y/2);
    else
        return x*power(x, y/2)*po
```

```
wer(x, y/2);
}
int main()
{
    int x = 2;
    int y = 3;

    printf("%d", power(x, y));
    return 0;
}
```

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

**Answer:** a

**Explanation:** The recurrence relation for the above code is given by  $T(n)=2T(n/2)+c$ . By using master theorem we can calculate the result for this relation. It is found to be equal to  $O(n)$ .

3. What is the space complexity of the given code?

```
#include<stdio.h>
int power(int x, int y)
{
    if (y == 0)
        return 1;
    else if (y%2 == 0)
        return power(x, y/2)*powe
r(x, y/2);
    else
        return x*power(x, y/2)*po
wer(x, y/2);
}
int main()
{
    int x = 2;
    int y = 3;

    printf("%d", power(x, y));
    return 0;
}
```

- a)  $O(1)$
- b)  $O(n)$
- c)  $O(\log n)$
- d)  $O(n \log n)$

**Answer:** a

**Explanation:** The space complexity of the given code will be equal to O(1) as it uses only constant space in the memory.

4. Recursive program to raise an integer x to power y uses which of the following algorithm?
- Dynamic programming
  - Backtracking
  - Divide and conquer
  - Greedy algorithm

**Answer:** c

**Explanation:** The recursive approach uses divide and conquer algorithm as we break the problem into smaller parts and then solve the smaller parts and finally combine their results to get the overall solution.

5. What is the least time in which we can raise a number x to power y?
- $O(x)$
  - $O(y)$
  - $O(\log x)$
  - $O(\log y)$

**Answer:** d

**Explanation:** We can optimize the code for finding power of a number by calculating x raised to power  $y/2$  only once and using it depending on whether y is even or odd.

6. What will be the time complexity of the following code?

```
#include<stdio.h>
int power(int x, int y)
{
    int temp;
    if (y == 0)
        return 1;
    temp = power(x, y/2);
    if (y%2 == 0)
        return temp*temp;
    else
        return x*temp*temp;
}
int main()
{
    int x = 2;
```

```
    int y = 3;
    printf("%d", power(x, y));
    return 0;
}
```

- $O(1)$
- $O(n)$
- $O(\log n)$
- $O(n \log n)$

**Answer:** c

**Explanation:** The given code is the optimized version for finding the power of a number. It forms a recurrence relation given by  $T(n)=T(n/2)+c$  which can be solved using master theorem. It is calculated to be equal to  $O(\log n)$ .

7. What is the advantage of iterative code for finding power of number over recursive code?

- Iterative code requires less time
- Iterative code requires less space
- Iterative code is more compiler friendly
- It has no advantage

**Answer:** b

**Explanation:** Both iterative and recursive approach can be implemented in  $\log n$  time but the recursive code requires memory in call stack which makes it less preferable.

8. Which of the following correctly implements iterative code for finding power of a number?

a)

```
#include <stdio.h>
int power(int x,  int y)
{
    int res = 1;
    while (y > 0)
    {
        if (y & 1)
            res = res * x;
        y = y >> 1;
        x = x * x;
    }
    return res;
```

```

}
int main()
{
    int x = 3;
    unsigned int y = 5;
    printf("%d", power(x, y));
    return 0;
}

```

b)

```

#include <stdio.h>
int power(int x, int y)
{
    int res = 1;
    while (y > 0)
    {
        if (y && 1)
            res = res * x;
        y = y >> 1;
        x = x * x;
    }
    return res;
}
int main()
{
    int x = 3;
    unsigned int y = 5;
    printf("%d", power(x, y));
    return 0;
}

```

c)

```

#include <stdio.h>
int power(int x, int y)
{
    int res = 1;
    while (y > 0)
    {
        if (y && 1)
            res = x * x;
        y = y >> 1;
        x = x * x;
    }
    return res;
}
int main()
{
    int x = 3;
    unsigned int y = 5;
    printf("%d", power(x, y));
    return 0;
}

```

d)

```

#include <stdio.h>
int power(int x, int y)
{
    int res = 1;
    while (y > 0)
    {
        if (y & 1)
            res = x * x;
        y = y >> 1;
        x = x * x;
    }
    return res;
}
int main()
{
    int x = 3;
    unsigned int y = 5;
    printf("%d", power(x, y));
    return 0;
}

```

**Answer:** a

**Explanation:** It represents the iterative version of required code. It has a time and space complexity of O(log n) and O(1) respectively.

9. Recursive approach to find power of a number is preferred over iterative approach.

- a) True
- b) False

**Answer:** b

**Explanation:** The recursive code requires memory in call stack which makes it less preferable as compared to iterative approach.

10. What will be the output for following code?

```

float power(float x, int y)
{
    float temp;
    if( y==0)
        return 1;
    temp = power(x, y/2);
    if (y%2 == 0)
        return temp*temp;
    else

```

```

    {
        if(y > 0)
            return x*temp*temp
    p;
        else
            return (temp*temp
    )/x;
    }
int main()
{
    float x = 2;
    int y = -3;
    printf("%f", power(x, y));
    return 0;
}

```

- a) Error
- b) 1/4
- c) 4
- d) 0.25

**Answer:** d

**Explanation:** The given code is capable of handling negative powers too. Thus, the output will be  $2^{-2} = 0.25$ .

### Sanfoundry Global Education & Learning Series – Data Structures & Algorithms.

---

1. What is the objective of tower of hanoi puzzle?
  - a) To move all disks to some other rod by following rules
  - b) To divide the disks equally among the three rods by following rules
  - c) To move all disks to some other rod in random order
  - d) To divide the disks equally among three rods in random order

**Answer:** a

**Explanation:** Objective of tower of hanoi problem is to move all disks to some other rod by following the following rules-1) Only one disk can be moved at a time. 2) Disk can only be moved if it is the uppermost disk of the stack. 3) No disk should be placed over a smaller disk.

2. Which of the following is NOT a rule of tower of hanoi puzzle?
  - a) No disk should be placed over a smaller disk
  - b) Disk can only be moved if it is the uppermost disk of the stack
  - c) No disk should be placed over a larger disk
  - d) Only one disk can be moved at a time

**Answer:** c

**Explanation:** The rule is to not put a disk over a smaller one. Putting a smaller disk over larger one is allowed.

3. The time complexity of the solution tower of hanoi problem using recursion is

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

**Answer:** b

**Explanation:** Time complexity of the problem can be found out by solving the recurrence relation:  $T(n)=2T(n-1)+c$ . Result of this relation is found to be equal to  $2^n$ . It can be solved using substitution.

4. Recurrence equation formed for the tower of hanoi problem is given by \_\_\_\_\_
  - a)  $T(n) = 2T(n-1)+n$
  - b)  $T(n) = 2T(n/2)+c$
  - c)  $T(n) = 2T(n-1)+c$
  - d)  $T(n) = 2T(n/2)+n$

**Answer:** c

**Explanation:** As there are 2 recursive calls to  $n-1$  disks and one constant time operation so the recurrence relation will be given by  $T(n) = 2T(n-1)+c$ .

5. Minimum number of moves required to solve a tower of hanoi problem with  $n$  disks is

- 
- a)  $2^n$
  - b)  $2^{n-1}$

- c)  $n^2$   
d)  $n^2-1$

**Answer:** b

**Explanation:** Minimum number of moves can be calculated by solving the recurrence relation –  $T(n)=2T(n-1)+c$ . Alternatively we can observe the pattern formed by the series of number of moves 1,3,7,15.....Either way it turns out to be equal to  $2^n-1$ .

6. Space complexity of recursive solution of tower of hanoi puzzle is \_\_\_\_\_

- a)  $O(1)$   
b)  $O(n)$   
c)  $O(\log n)$   
d)  $O(n \log n)$

**Answer:** b

**Explanation:** Space complexity of tower of hanoi problem can be found out by solving the recurrence relation  $T(n)=T(n-1)+c$ . Result of this relation is found out to be  $n$ . It can be solved using substitution.

7. Which of the following functions correctly represent the solution to tower of hanoi puzzle?

a)

```
void ToH(int n,int a,int b,int c)
{
    If(n>0)
    {
        ToH(n-1,a,c,b);
        cout<<"move a disk from" <<a<<" to"
        <<c;
        ToH(n-1,b,a,c);
    }
}
```

b)

```
void ToH(int n,int a,int b,int c
{
    If(n>0)
    {
        ToH(n-1,a,b,c);
        cout<<"move a disk from" <<a<<" to"
        <<c;
        ToH(n-1,b,a,c);
```

}  
}

c)

```
void ToH(int n,int a,int b,int c)
{
    If(n>0)
    {
        ToH(n-1,a,c,b);
        cout<<"move a disk from" <<a<<" to"
        <<c;
        ToH(n-1,a,b,c);
    }
}
```

d)

```
void ToH(int n,int a,int b,int c)
{
    If(n>0)
    {
        ToH(n-1,b,a,c);
        cout<<"move a disk from" <<a<<" to"
        <<c;
        ToH(n-1,a,c,b);
    }
}
```

**Answer:** a

**Explanation:** The first recursive call moves  $n-1$  disks from a to b using c. Then we move a disk from a to c. Finally the second recursive call moves  $n-1$  disks from b to c using a.

8. Recursive solution of tower of hanoi problem is an example of which of the following algorithm?

- a) Dynamic programming  
b) Backtracking  
c) Greedy algorithm  
d) Divide and conquer

**Answer:** d

**Explanation:** The recursive approach uses divide and conquer algorithm as we break the problem into smaller parts and then solve the smaller parts and finally combine their results to get the overall solution.

9. Tower of hanoi problem can be solved iteratively.

- a) True
- b) False

**Answer:** a

**Explanation:** Iterative solution to tower of hanoi puzzle also exists. Its approach depends on whether the total numbers of disks are even or odd.

10. Minimum time required to solve tower of hanoi puzzle with 4 disks assuming one move takes 2 seconds, will be \_\_\_\_\_

- a) 15 seconds
- b) 30 seconds
- c) 16 seconds
- d) 32 seconds

**Answer:** b

**Explanation:** Number of moves =  $2^4 - 1 = 16 - 1 = 15$

Time for 1 move = 2 seconds

Time for 15 moves =  $15 \times 2 = 30$  seconds.

### Sanfoundry Global Education & Learning Series – Data Structures & Algorithms.

---

1. Master's theorem is used for?

- a) solving recurrences
- b) solving iterative relations
- c) analysing loops
- d) calculating the time complexity of any code

**Answer:** a

**Explanation:** Master's theorem is a direct method for solving recurrences. We can solve any recurrence that falls under any one of the three cases of master's theorem.

2. How many cases are there under Master's theorem?

- a) 2
- b) 3
- c) 4
- d) 5

**Answer:** b

**Explanation:** There are primarily 3 cases under master's theorem. We can solve any recurrence that falls under any one of these three cases.

3. What is the result of the recurrences which fall under first case of Master's theorem (let the recurrence be given by  $T(n)=aT(n/b)+f(n)$  and  $f(n)=n^c$ )?

- a)  $T(n) = O(n^{\log_b a})$
- b)  $T(n) = O(n^c \log n)$
- c)  $T(n) = O(f(n))$
- d)  $T(n) = O(n^2)$

**Answer:** a

**Explanation:** In first case of master's theorem the necessary condition is that  $c < \log_b a$ . If this condition is true then  $T(n) = O(n^{\log_b a})$ .

4. What is the result of the recurrences which fall under second case of Master's theorem (let the recurrence be given by  $T(n)=aT(n/b)+f(n)$  and  $f(n)=n^c$ )?

- a)  $T(n) = O(n \log_b a)$
- b)  $T(n) = O(n^c \log n)$
- c)  $T(n) = O(f(n))$
- d)  $T(n) = O(n^2)$

**Answer:** b

**Explanation:** In second case of master's theorem the necessary condition is that  $c = \log_b a$ . If this condition is true then  $T(n) = O(n^c \log n)$

5. What is the result of the recurrences which fall under third case of Master's theorem (let the recurrence be given by  $T(n)=aT(n/b)+f(n)$  and  $f(n)=n^c$ )?

- b)  $T(n) = O(n \log_b a)$
- b)  $T(n) = O(n^c \log n)$
- c)  $T(n) = O(f(n))$
- d)  $T(n) = O(n^2)$

**Answer:** c

**Explanation:** In third case of master's theorem the necessary condition is that  $c > \log_b a$ . If this condition is true then  $T(n) = O(f(n))$ .

6. We can solve any recurrence by using Master's theorem.

- a) true
- b) false

**Answer:** b

**Explanation:** No we cannot solve all the recurrences by only using master's theorem. We can solve only those which fall under the three cases prescribed in the theorem.

7. Under what case of Master's theorem will the recurrence relation of merge sort fall?

- a) 1
- b) 2
- c) 3
- d) It cannot be solved using master's theorem

**Answer:** b

**Explanation:** The recurrence relation of merge sort is given by  $T(n) = 2T(n/2) + O(n)$ . So we can observe that  $c = \log_b a$  so it will fall under case 2 of master's theorem.

8. Under what case of Master's theorem will the recurrence relation of stooge sort fall?

- a) 1
- b) 2
- c) 3
- d) It cannot be solved using master's theorem

**Answer:** a

**Explanation:** The recurrence relation of stooge sort is given as  $T(n) = 3T(2/3n) + O(1)$ . It is found to be equal to  $O(n^{2.7})$  using master's theorem first case.

9. Which case of master's theorem can be extended further?

- a) 1
- b) 2

c) 3

d) No case can be extended

**Answer:** b

**Explanation:** The second case of master's theorem can be extended for a case where  $f(n) = n^c (\log n)^k$  and the resulting recurrence becomes  $T(n) = O(n^c (\log n)^{k+1})$ .

10. What is the result of the recurrences which fall under the extended second case of Master's theorem (let the recurrence be given by  $T(n) = aT(n/b) + f(n)$  and  $f(n) = n^c (\log n)^k$ )?

- a)  $T(n) = O(n \log_b a)$
- b)  $T(n) = O(n^c \log n)$
- c)  $T(n) = O(n^c (\log n)^{k+1})$
- d)  $T(n) = O(n^2)$

**Answer:** c

**Explanation:** In the extended second case of master's theorem the necessary condition is that  $c = \log_b a$ . If this condition is true then  $T(n) = O(n^c (\log n)^{k+1})$ .

11. Under what case of Master's theorem will the recurrence relation of binary search fall?

- a) 1
- b) 2
- c) 3
- d) It cannot be solved using master's theorem

**Answer:** b

**Explanation:** The recurrence relation of binary search is given by  $T(n) = T(n/2) + O(1)$ . So we can observe that  $c = \log_b a$  so it will fall under case 2 of master's theorem.

1. Solve the following recurrence using Master's theorem.

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

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

**Answer:** c

**Explanation:** The given recurrence can be solved by using the second case of Master's theorem.

$$T(n) = O(nc \log n) = \text{Here } nc = n^2$$

So the solution becomes  $T(n) = O(n^2 \log n)$ .

2. Solve the following recurrence using Master's theorem.

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

- a)  $T(n) = O(n^2)$
- b)  $T(n) = O(n^2 \log n)$
- c)  $T(n) = O(2^n)$
- d) cannot be solved

**Answer:** c

**Explanation:** The given recurrence can be solved by using the third case (as  $f(n) > \log 21$ ) of Master's theorem.

$$T(n) = O(f(n)) = \text{Here } f(n) = 2n.$$

So the solution becomes  $T(n) = O(2n)$ .

3. Solve the following recurrence using Master's theorem.

$$T(n) = 16T(n/4) + n$$

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

**Answer:** d

**Explanation:** The given recurrence can be solved by using the first case of Master's theorem. So the solution becomes  $T(n) = O(n^2)$ .

4. Solve the following recurrence using Master's theorem.

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

- a)  $T(n) = O(n)$
- b)  $T(n) = O(\log n)$
- c)  $T(n) = O(n^2 \log n)$
- d) cannot be solved using master's theorem

**Answer:** d

**Explanation:** The given recurrence cannot be

solved by using the Master's theorem. It is because this recurrence relation does not fit into any case of Master's theorem.

5. Solve the following recurrence using Master's theorem.

$$T(n) = 0.7 T(n/2) + 1/n$$

- a)  $T(n) = O(n)$
- b)  $T(n) = O(\log n)$
- c)  $T(n) = O(n^2 \log n)$
- d) cannot be solved using master's theorem

**Answer:** d

**Explanation:** The given recurrence cannot be solved by using the Master's theorem. It is because in this recurrence relation  $a < 1$  so master's theorem cannot be applied.

6. Solve the following recurrence using Master's theorem.

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

- a)  $T(n) = O(n!)$
- b)  $T(n) = O(n! \log n)$
- c)  $T(n) = O(n^2 \log n)$
- d) cannot be solved using master's theorem

**Answer:** a

**Explanation:** The given recurrence can be solved by using the third case of Master's theorem. So the solution becomes  $T(n) = O(n!)$ .

7. Solve the following recurrence using Master's theorem.

$$T(n) = 4T(n/4) + n \log n$$

- a)  $T(n) = O(n (\log n)^2)$
- b)  $T(n) = O(n \log n)$
- c)  $T(n) = O(n^2 \log n)$
- d) cannot be solved using master's theorem

**Answer:** a

**Explanation:** The given recurrence can be solved by using the extended second case of Master's theorem. So the solution becomes  $T(n) = O(n (\log n)^2)$ .

8. What will be the recurrence relation of the following code?

```
Int sum(int n)
{
    If(n==1)
        return 1;
    else
        return n+sum(n-1);
}
```

- a)  $T(n) = T(n/2) + n$
- b)  $T(n) = T(n-1) + n$
- c)  $T(n) = T(n-1) + O(1)$
- d)  $T(n) = T(n/2) + O(1)$

**Answer:** c

**Explanation:** As after every recursive call the integer up to which the sum is to be calculated decreases by 1. So the recurrence relation for the given code will be  $T(n) = T(n-1) + O(1)$ .

9. What will be the recurrence relation of the following code?

```
int xpowy(int x, int n)
if (n==0) return 1;
if (n==1) return x;
if ((n % 2) == 0)
return xpowy(x*x, n/2);
else
return xpowy(x*x, n/2) * x;
```

- a)  $T(n) = T(n/2) + n$
- b)  $T(n) = T(n-1) + n$
- c)  $T(n) = T(n-1) + O(1)$
- d)  $T(n) = T(n/2) + O(1)$

**Answer:** d

**Explanation:** As after every recursive call the integer up to which the power is to be calculated decreases by half. So the recurrence relation for the given code will be  $T(n) = T(n/2) + O(1)$ . It can be solved by using master's theorem.

10. What will be the time complexity of the following code?

```
int xpowy(int x, int n)
{
    if (n==0)
        return 1;
    if (n==1)
        return x;
    if ((n % 2) == 0)
        return xpowy(x*x, n/2);
    else
        return xpowy(x*x, n/2) * x;
}
```

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

**Answer:** a

**Explanation:** As the recurrence relation of the code is given by  $T(n) = T(n/2) + O(1)$  so it can be solved by using master's theorem second case.

---



---

## UNIT II BRUTE FORCE AND DIVIDE-AND- CONQUER

1. Which of the following is a sub string of “SANFOUNDRY”?

- a) SANO
- b) FOUND
- c) SAND
- d) FOND

**Answer:** b

**Explanation:** A sub string is a subset of another string. So “FOUND” is the only possible sub string out of the given options.

2. What will be the output of the following code?

```
#include<bits/stdc++.h>
using namespace std;

void func(char* str2, char* str1)
```

```

{
    int m = strlen(str2);
    int n = strlen(str1);
    for (int i = 0; i <= n - m; i++)
    {
        int j;
        for (j = 0; j < m; j++)
            if (str1[i + j] != str2[j])
                break;
        if (j == m)
            cout << i << endl;
    }
}
int main()
{
    char str1[] = "1253234";
    char str2[] = "323";
    func(str2, str1);
    return 0;
}

a) O(n)
b) O(m)
c) O(m * n)
d) O(m + n)

```

**Answer:** c

**Explanation:** The given code describes the naive method of finding a pattern in a string. So the output will be 3 as the given sub string begins at that index in the pattern.

3. What will be the worst case time complexity of the following code?

```
#include<bits/stdc++.h>
using namespace std;

void func(char* str2, char* str1)
{
    int m = strlen(str2);
    int n = strlen(str1);
    for (int i = 0; i <= n - m; i++)
    {
        int j;
        for (j = 0; j < m; j++)
            if (str1[i + j] != str2[j])
                break;
    }
}
```

```

= str2[j])                                break;
  if (j == m)
  cout << i << endl;
  ;
  }
}
int main()
{
    char str1[] = "1253234";
    char str2[] = "323";
    func(str2, str1);
    return 0;
}
```

- a) O(n)
- b) O(m)
- c) O(m \* n)
- d) O(m + n)

**Answer:** c

**Explanation:** The given code describes the naive method of pattern searching. By observing the nested loop in the code we can say that the time complexity of the loop is  $O(m*n)$ .

4. What will be the auxiliary space complexity of the following code?

```
#include<bits/stdc++.h>
using namespace std;

void func(char* str2, char* str1)
{
    int m = strlen(str2);
    int n = strlen(str1);
    for (int i = 0; i <= n - m; i++)
    {
        int j;
        for (j = 0; j < m; j++)
            if (str1[i + j] != str2[j])
                break;
        if (j == m)
            cout << i << endl;
    }
}
```

```

int main()
{
    char str1[] = "1253234";
    char str2[] = "323";
    func(str2, str1);
    return 0;
}

```

- a) O(n)
- b) O(1)
- c) O(log n)
- d) O(m)

**Answer:** b

**Explanation:** The given code describes the naive method of pattern searching. Its auxiliary space requirement is O(1).

5. What is the worst case time complexity of KMP algorithm for pattern searching ( $m$  = length of text,  $n$  = length of pattern)?

- a) O(n)
- b) O( $n^*m$ )
- c) O(m)
- d) O( $\log n$ )

**Answer:** c

**Explanation:** KMP algorithm is an efficient pattern searching algorithm. It has a time complexity of O(m) where m is the length of text.

6. What will be the best case time complexity of the following code?

```

#include<bits/stdc++.h>
using namespace std;
void func(char* str2, char* str1)
{
    int m = strlen(str2);
    int n = strlen(str1);

    for (int i = 0; i <= n - m; i++)
    {
        int j;

        for (j = 0; j < m; j++)
            if (str1[i + j] != str2[j])
                break;
    }
}

```

```

        if (j == m)
            cout << i << endl
        ;
    }
}

int main()
{
    char str1[] = "1253234";
    char str2[] = "323";
    func(str2, str1);
    return 0;
}

```

- a) O(n)
- b) O(m)
- c) O( $m * n$ )
- d) O( $m + n$ )

**Answer:** b

**Explanation:** The given code describes the naive method of pattern searching. The best case of the code occurs when the first character of the pattern does not appear in the text at all. So in such a case, only one iteration is required thus time complexity will be O(m).

7. What is the time complexity of Z algorithm for pattern searching ( $m$  = length of text,  $n$  = length of pattern)?

- a) O( $n + m$ )
- b) O(m)
- c) O(n)
- d) O( $m * n$ )

**Answer:** a

**Explanation:** Z algorithm is an efficient pattern searching algorithm as it searches the pattern in linear time. It has a time complexity of O( $m + n$ ) where m is the length of text and n is the length of the pattern.

8. What is the auxiliary space complexity of Z algorithm for pattern searching ( $m$  = length of text,  $n$  = length of pattern)?

- a) O( $n + m$ )
- b) O(m)
- c) O(n)
- d) O( $m * n$ )

**Answer:** b

**Explanation:** Z algorithm is an efficient pattern searching algorithm as it searches the pattern in linear time. It uses auxiliary space of  $O(m)$  for maintaining Z array.

9. The naive pattern searching algorithm is an in place algorithm.  
 a) true  
 b) false

**Answer:** a

**Explanation:** The auxiliary space complexity required by naive pattern searching algorithm is  $O(1)$ . So it qualifies as an in place algorithm.

10. Rabin Karp algorithm and naive pattern searching algorithm have the same worst case time complexity.  
 a) true  
 b) false

**Answer:** a

**Explanation:** The worst case time complexity of Rabin Karp algorithm is  $O(m*n)$  but it has a linear average case time complexity. So Rabin Karp and naive pattern searching algorithm have the same worst case time complexity.

1. Which of the following sorting algorithms is the fastest?  
 a) Merge sort  
 b) Quick sort  
 c) Insertion sort  
 d) Shell sort

**Answer:** b

**Explanation:** Quick sort is the fastest known sorting algorithm because of its highly optimized inner loop.

2. Quick sort follows Divide-and-Conquer strategy.  
 a) True  
 b) False

**Answer:** a

**Explanation:** In quick sort, the array is divided into sub-arrays and then it is sorted (divide-and-conquer strategy).

3. What is the worst case time complexity of a quick sort algorithm?  
 a)  $O(N)$   
 b)  $O(N \log N)$   
 c)  $O(N^2)$   
 d)  $O(\log N)$

**Answer:** c

**Explanation:** The worst case performance of a quick sort algorithm is mathematically found to be  $O(N^2)$ .

4. Which of the following methods is the most effective for picking the pivot element?  
 a) first element  
 b) last element  
 c) median-of-three partitioning  
 d) random element

**Answer:** c

**Explanation:** Median-of-three partitioning is the best method for choosing an appropriate pivot element. Picking a first, last or random element as a pivot is not much effective.

5. Find the pivot element from the given input using median-of-three partitioning method.  
 8, 1, 4, 9, 6, 3, 5, 2, 7, 0.  
 a) 8  
 b) 7  
 c) 9  
 d) 6

**Answer:** d

**Explanation:** Left element=8, right element=0,  
 Centre=[position(left+right)/2]=6.

6. Which is the safest method to choose a pivot element?  
 a) choosing a random element as pivot  
 b) choosing the first element as pivot

- c) choosing the last element as pivot
- d) median-of-three partitioning method

**Answer:** a

**Explanation:** This is the safest method to choose the pivot element since it is very unlikely that a random pivot would consistently provide a poor partition.

7. What is the average running time of a quick sort algorithm?

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

**Answer:** c

**Explanation:** The best case and average case analysis of a quick sort algorithm are mathematically found to be  $O(N \log N)$ .

8. Which of the following sorting algorithms is used along with quick sort to sort the sub arrays?

- a) Merge sort
- b) Shell sort
- c) Insertion sort
- d) Bubble sort

**Answer:** c

**Explanation:** Insertion sort is used along with quick sort to sort the sub arrays. It is used only at the end.

9. Quick sort uses join operation rather than merge operation.

- a) true
- b) false

**Answer:** a

**Explanation:** Quick sort uses join operation since join is a faster operation than merge.

10. How many sub arrays does the quick sort algorithm divide the entire array into?

- a) one
- b) two

- c) three
- d) four

**Answer:** b

**Explanation:** The entire array is divided into two partitions, 1st sub array containing elements less than the pivot element and 2nd sub array containing elements greater than the pivot element.

11. Which is the worst method of choosing a pivot element?

- a) first element as pivot
- b) last element as pivot
- c) median-of-three partitioning
- d) random element as pivot

**Answer:** a

**Explanation:** Choosing the first element as pivot is the worst method because if the input is pre-sorted or in reverse order, then the pivot provides a poor partition.

12. Which among the following is the best cut-off range to perform insertion sort within a quick sort?

- a)  $N=0-5$
- b)  $N=5-20$
- c)  $N=20-30$
- d)  $N>30$

**Answer:** b

**Explanation:** A good cut-off range is anywhere between  $N=5$  and  $N=20$  to avoid nasty degenerate cases.

1. The shortest distance between a line and a point is achieved when?

- a) a line is drawn at 90 degrees to the given line from the given point
- b) a line is drawn at 180 degrees to the given line from the given point
- c) a line is drawn at 60 degrees to the given line from the given point
- d) a line is drawn at 270 degrees to the given line from the given point

**Answer:** a

**Explanation:** The shortest distance between a line and a point is achieved when a line is drawn at 90 degrees to the given line from the given point.

2. What is the shortest distance between the line given by  $ax + by + c = 0$  and the point  $(x_1, y_1)$ ?

$$\frac{|ax_1+by_1+c|}{\sqrt{a^2+b^2}}$$

- a)  $\frac{\sqrt{a^2+b^2}}{|ay_1+bx_1+c|}$
- b)  $\frac{|ay_1+bx_1+c|}{\sqrt{a^2+b^2}}$
- c)  $\frac{|ax_1+by_1|}{\sqrt{a^2+b^2}}$
- d)  $ax_1+by_1+c$

**Answer:** a

**Explanation:** The shortest distance between a line and a point is given by the formula  $(ax_1+by_1+c)/(\sqrt{a^2+b^2})$ . This formula can be derived using the formula of area of a triangle.

3. What is the shortest distance between the line given by  $-2x + 3y + 4 = 0$  and the point  $(5,6)$ ?

- a) 4.5 units
- b) 5.4 units
- c) 4.3 units
- d) 3.3 units

**Answer:** d

**Explanation:** The shortest distance between a line and a point is given by the formula  $(ax_1+by_1+c)/(\sqrt{a^2+b^2})$ . Using this formula we get the answer 3.3 units.

4. What is the general formula for finding the shortest distance between two parallel lines given by  $ax+by+c_1=0$  and  $ax+by+c_2=0$ ?

$$\frac{|ax_1+by_1+c|}{\sqrt{a^2+b^2}}$$

- a)  $\frac{\sqrt{a^2+b^2}}{|a_1x_1+b_1y_1+c_1|}$

b)  $\frac{|c_1 - c_2|}{\sqrt{a^2+b^2}}$

c)  $\frac{|c_1 + c_2|}{\sqrt{a^2+b^2}}$

- d)  $c_1 + c_2$

**Answer:** b

**Explanation:** The general formula for finding the shortest distance between two parallel lines given by  $ax+by+c_1$  and  $ax+by+c_2$  is  $(c_1-c_2)/(\sqrt{a^2+b^2})$ . We can find this by considering the distance of any one point on one of the line to the other line.

5. What is the distance between the lines  $3x-4y+7=0$  and  $3x-4y+5=0$ ?

- a) 1 unit
- b) 0.5 unit
- c) 0.8 unit
- d) 0.4 unit

**Answer:** d

**Explanation:** As the given lines are parallel so the distance between them can be calculated by using the formula  $(c_1 - c_2)/(\sqrt{a^2+b^2})$ . So we get the distance as 0.4 unit.

6. What will be the slope of the line given by  $ax + by + c = 0$ ?

- a)  $-a/b$
- b)  $-b/a$
- c)  $-c/a$
- d)  $a/c$

**Answer:** a

**Explanation:** The slope of a line given by the equation  $ax + by + c = 0$  has the slope of  $-a/b$ . So two lines having the same ratio of the coefficient of x and y will be parallel to each other.

7. What will be the slope of the line given by  $10x + 5y + 8 = 0$ ?

- a) -5

- b) -2
- c) -1.25
- d) 5

**Answer:** b

**Explanation:** The slope of a line given by the equation  $ax + by + c = 0$  has the slope of  $-a/b$ . So the slope of the given line will be -2.

8. What will be the co-ordinates of foot of perpendicular line drawn from the point (-1,3) to the line  $3x-4y-16=0$ ?

- a) (1/5,2/5)
- b) (2/25,5/25)
- c) (68/25,-49/25)
- d) (-49/25,68/25)

**Answer:** c

**Explanation:** The foot of perpendicular can be found by equating the distance between the two points and the distance between point and line. This is found to be (68/25,-49/25).

9. Which of the following is used to find the absolute value of the argument in C++?

- a) abs()
- b) fabs()
- c) mod()
- d) ab()

**Answer:** b

**Explanation:** In C++ the absolute value of an argument can be found by using the function fabs(). It is available under the header file math.h.

10. What will be the slope of the line perpendicular to the line  $6x-3y-16=0$ ?

- a) 1/2
- b) -1/2
- c) 2
- d) -2

**Answer:** b

**Explanation:** For two lines to be perpendicular the product of their slopes should be equal to -1. So as the slope of given

line is 2 so the slope of line perpendicular to it will be  $-1/2$ .

11. Find the output of the following code.

```
#include<math.h>
#include<iostream>
using namespace std;
void distance(float x, float y, float a,
float b, float c)
{
    float d = fabs((a * x + b * y + c
)) / (sqrt(a * a + b * b));
    cout<<d;
    return;
}
int main()
{
    float x = -2;
    float y = -3;
    float a = 5;
    float b = -2;
    float c = - 4;
    distance(x, y, a, b, c);
    return 0;
}
```

- a) 2.8
- b) 1.8
- c) 1.4
- d) 2.4

**Answer:** c

**Explanation:** The given code calculates the shortest distance between line and a point. So the output will be 1.4.

1. Which of the following areas do closest pair problem arise?

- a) computational geometry
- b) graph colouring problems
- c) numerical problems
- d) string matching

**Answer:** a

**Explanation:** Closest pair problem arises in two most important areas- computational geometry and operational research.

2. Which approach is based on computing the distance between each pair of distinct points

- and finding a pair with the smallest distance?
- Brute force
  - Exhaustive search
  - Divide and conquer
  - Branch and bound

**Answer:** a

**Explanation:** Brute force is a straight forward approach that solves closest pair problem using that algorithm.

3. What is the runtime efficiency of using brute force technique for the closest pair problem?
- $O(N)$
  - $O(N \log N)$
  - $O(N^2)$
  - $O(N^3 \log N)$

**Answer:** c

**Explanation:** The efficiency of closest pair algorithm by brute force technique is mathematically found to be  $O(N^2)$ .

4. The most important condition for which closest pair is calculated for the points  $(p_i, p_j)$  is?
- $i > j$
  - $i \neq j$
  - $i = j$
  - $i < j$

**Answer:** d

**Explanation:** To avoid computing the distance between the same pair of points twice, we consider only the pair of points  $(p_i, p_j)$  for which  $i < j$ .

5. What is the basic operation of closest pair algorithm using brute force technique?
- Euclidean distance
  - Radius
  - Area
  - Manhattan distance

**Answer:** a

**Explanation:** The basic operation of closest

pair algorithm is Euclidean distance and its formula is given by  $d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ .

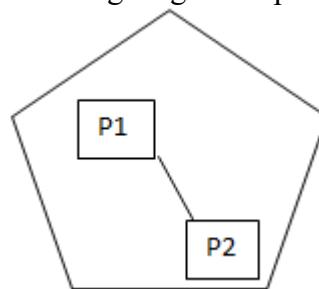
6. Which of the following is similar to Euclidean distance?

- Manhattan distance
- Pythagoras metric
- Chebyshev distance
- Heuristic distance

**Answer:** b

**Explanation:** In older times, Euclidean distance metric is also called a Pythagoras metric which is the length of the line segment connecting two points.

7. Which of the following strategies does the following diagram depict?



- Divide and conquer strategy
- Brute force
- Exhaustive search
- Backtracking

**Answer:** b

**Explanation:** Brute force is a straight forward approach to solve critical problems. Here, we use brute force technique to find the closest distance between p1 and p2.

8. Manhattan distance is an alternative way to define a distance between two points.

- true
- false

**Answer:** a

**Explanation:** Manhattan distance is an alternative way to calculate distance. It is the distance between two points measured along axes at right angles.

9. What is the optimal time required for solving the closest pair problem using divide and conquer approach?

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

**Answer:** c

**Explanation:** The optimal time for solving using a divide and conquer approach is mathematically found to be  $O(N \log N)$ .

10. In divide and conquer, the time is taken for merging the subproblems is?

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

**Answer:** b

**Explanation:** The time taken for merging the smaller subproblems in a divide and conquer approach is mathematically found to be  $O(N \log N)$ .

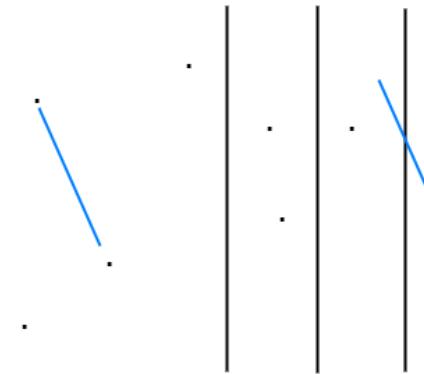
11. The optimal time obtained through divide and conquer approach using merge sort is the best case efficiency.

- a) true
- b) false

**Answer:** a

**Explanation:** The optimal time obtained through divide and conquer approach is the best class efficiency and it is given by  $\Omega(N \log N)$ .

12. Which of the following strategies does the following diagram depict?

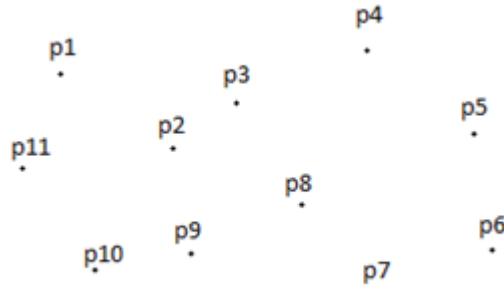


- a) Brute force
- b) Divide and conquer
- c) Exhaustive search
- d) Branch and bound

**Answer:** b

**Explanation:** The above diagram depicts the implementation of divide and conquer. The problem is divided into sub problems and are separated by a line.

13. Which of the points are closer to each other?



- a) p1 and p11
- b) p3 and p8
- c) p2 and p3
- d) p9 and p10

**Answer:** c

**Explanation:** From symmetry, we determine that the closest pair is p2 and p3. But the exact calculations have to be done using Euclid's algorithm.

1. Cross product is a mathematical operation performed between \_\_\_\_\_
- a) 2 scalar numbers

- b) a scalar and a vector
- c) 2 vectors
- d) any 2 numbers

**Answer:** c

**Explanation:** Cross product is a mathematical operation that is performed on 2 vectors in a 3D plane. It has many applications in computer programming and physics.

2. Cross product is also known as?

- a) scalar product
- b) vector product
- c) dot product
- d) multiplication

**Answer:** b

**Explanation:** Cross product is also known as a vector product. It is a mathematical operation that is performed on 2 vectors in 3D plane.

3. What is the magnitude of resultant of cross product of two parallel vectors a and b?

- a)  $|a| \cdot |b|$
- b)  $|a| \cdot |b| \cos(180)$
- c)  $|a| \cdot |b| \sin(180)$
- d) 1

**Answer:** c

**Explanation:** The resultant of cross product of 2 parallel vectors is always 0 as the angle between them is 0 or 180 degrees. So the answer is  $|a| \cdot |b| \sin(180)$ .

4. What is the general formula for finding the magnitude of the cross product of two vectors a and b with angle  $\theta$  between them?

- a)  $|a| \cdot |b|$
- b)  $|a| \cdot |b| \cos(\theta)$
- c)  $|a| \cdot |b| \sin(\theta)$
- d)  $|a| \cdot |b| \tan(\theta)$

**Answer:** c

**Explanation:** The general formula for finding the magnitude of cross product of two vectors

is  $|a| \cdot |b| \sin(\theta)$ . Its direction is perpendicular to the plane containing a and b.

5. The concept of cross product is applied in the field of computer graphics.

- a) true
- b) false

**Answer:** a

**Explanation:** The concept of cross product find its application in the field of computer graphics. It can be used to find the winding of polygon about a point.

6. Which of the following equals the  $a \times b$  ( a and b are two vectors)?

- a)  $-(a \times b)$
- b)  $a \cdot b$
- c)  $b \times a$
- d)  $-(b \times a)$

**Answer:** d

**Explanation:** The vector product  $a \times b$  is equal to  $-(b \times a)$ . The minus sign shows that these vectors have opposite directions.

7. Cross product of two vectors can be used to find?

- a) area of rectangle
- b) area of square
- c) area of parallelogram
- d) perimeter of rectangle

**Answer:** c

**Explanation:** Cross product of two vectors can be used to find the area of parallelogram. For this, we need to consider the vectors as the adjacent sides of the parallelogram.

8. The resultant vector from the cross product of two vectors is \_\_\_\_\_

- a) perpendicular to any one of the two vectors involved in cross product
- b) perpendicular to the plane containing both vectors
- c) parallel to any one of the two vectors involved in cross product

- d) parallel to the plane containing both vectors

**Answer:** b

**Explanation:** The resultant vector from the cross product of two vectors is perpendicular to the plane containing both vectors. In other words, it should be perpendicular to both the vectors involved in the cross product.

9. What will be the cross product of the vectors  $2\mathbf{i} + 3\mathbf{j} + \mathbf{k}$  and  $3\mathbf{i} + 2\mathbf{j} + \mathbf{k}$ ?

- a)  $\mathbf{i} + 2\mathbf{j} + \mathbf{k}$
- b)  $2\mathbf{i} + 3\mathbf{j} + \mathbf{k}$
- c)  $\mathbf{i} + \mathbf{j} - 5\mathbf{k}$
- d)  $2\mathbf{i} - \mathbf{j} - 5\mathbf{k}$

**Answer:** c

**Explanation:** We can find the cross product of the given vectors by solving the determinant.

$$\begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ 2 & 3 & 1 \\ 3 & 2 & 1 \end{vmatrix}$$

10. What will be the cross product of the vectors  $2\mathbf{i} + 3\mathbf{j} + \mathbf{k}$  and  $6\mathbf{i} + 9\mathbf{j} + 3\mathbf{k}$ ?

- a)  $\mathbf{i} + 2\mathbf{j} + \mathbf{k}$
- b)  $\mathbf{i} - \mathbf{j} - 5\mathbf{k}$
- c) 0
- d)  $2\mathbf{i} - \mathbf{j} - 5\mathbf{k}$

**Answer:** c

**Explanation:** The given vectors are parallel to each other. The cross product of parallel vectors is 0 because  $\sin(0)$  is 0.

11. Find the output of the following code.

```
#include <bits/stdc++.h>
using namespace std;
void crossP(int A[], int B[], int cross[])
{
    cross[0] = A[1] * B[2] - A[2] * B[1];
    cross[1] = A[0] * B[2] - A[2] * B[0];
}
```

```
cross[2] = A[0] * B[1] - A[1] * B[0];
}
int main()
{
    int A[] = { 1, 2, 4 };
    int B[] = { 2, 3, 2 };
    int cross[3];
    crossP(A, B, cross);
    for (int i = 0; i < 3; i++)
        cout << cross[i] << " ";
    return 0;
}

a) 1 2 5
b) -1 -5 -3
c) -6 -8 -1
d) -8 -6 -1
```

**Answer:** d

**Explanation:** The given code calculates the cross product of the vectors stored in arrays A and B respectively. So the output will be -8 -6 -1.

12. Which of the following operation will give a vector that is perpendicular to both vectors a and b?

- a)  $\mathbf{a} \times \mathbf{b}$
- b)  $\mathbf{a} \cdot \mathbf{b}$
- c)  $\mathbf{b} \times \mathbf{a}$
- d) both  $\mathbf{a} \times \mathbf{b}$  and  $\mathbf{b} \times \mathbf{a}$

**Answer:** d

**Explanation:** The resultant vector from the cross product of two vectors is perpendicular to the plane containing both vectors. So both  $\mathbf{a} \times \mathbf{b}$  and  $\mathbf{b} \times \mathbf{a}$  will give a vector that is perpendicular to both vectors a and b.

1. \_\_\_\_\_ is a method of constructing a smallest polygon out of n given points.
- a) closest pair problem
  - b) quick hull problem
  - c) path compression
  - d) union-by-rank

**Answer:** b

**Explanation:** Quick hull is a method of

constructing a smallest convex polygon out of  $n$  given points in a plane.

2. What is the other name for quick hull problem?

- a) convex hull
- b) concave hull
- c) closest pair
- d) path compression

**Answer:** a

**Explanation:** The other name for quick hull problem is convex hull problem whereas the closest pair problem is the problem of finding the closest distance between two points.

3. How many approaches can be applied to solve quick hull problem?

- a) 1
- b) 2
- c) 3
- d) 4

**Answer:** b

**Explanation:** Most commonly, two approaches are adopted to solve quick hull problem- brute force approach and divide and conquer approach.

4. What is the average case complexity of a quick hull algorithm?

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

**Answer:** b

**Explanation:** The average case complexity of quickhull algorithm using divide and conquer approach is mathematically found to be  $O(N \log N)$ .

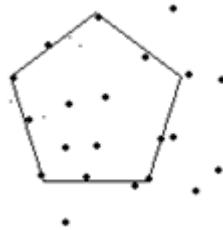
5. What is the worst case complexity of quick hull?

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

**Answer:** c

**Explanation:** The worst case complexity of quickhull algorithm using divide and conquer approach is mathematically found to be  $O(N^2)$ .

6. What does the following diagram depict?



- a) closest pair
- b) convex hull
- c) concave hull
- d) path compression

**Answer:** b

**Explanation:** The above diagram is a depiction of convex hull, also known as quick hull, since it encloses  $n$  points into a convex polygon.

7. Which of the following statement is not related to quickhull algorithm?

- a) finding points with minimum and maximum coordinates
- b) dividing the subset of points by a line
- c) eliminating points within a formed triangle
- d) finding the shortest distance between two points

**Answer:** d

**Explanation:** Finding the shortest distance between two points belongs to closest pair algorithm while the rest is quickhull.

8. The quick hull algorithm runs faster if the input uses non-extreme points.

- a) true
- b) false

**Answer:** a

**Explanation:** It is proved that the quick hull algorithm runs faster if the input uses non-

extreme points and also, if it uses less memory.

9. To which type of problems does quick hull belong to?

- a) numerical problems
- b) computational geometry
- c) graph problems
- d) string problems

**Answer:** b

**Explanation:** Quick hull problem and closest pair algorithms are some of the examples of computational problems.

10. Which of the following algorithms is similar to a quickhull algorithm?

- a) merge sort
- b) shell sort
- c) selection sort
- d) quick sort

**Answer:** d

**Explanation:** Quickhull algorithm is similar to a quick sort algorithm with respect to the run time average case and worst case efficiencies.

11. Who formulated quick hull algorithm?

- a) Eddy
- b) Andrew
- c) Chan
- d) Graham

**Answer:** a

**Explanation:** Eddy formulated quick hull algorithm. Graham invented graham scan. Andrew formulated Andrew's algorithm and Chan invented Chan's algorithm.

12. The time is taken to find the 'n' points that lie in a convex quadrilateral is?

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

**Answer:** a

**Explanation:** The time taken to find the 'n' points that lie in a convex quadrilateral is mathematically found to be  $O(N)$ .

1. Chan's algorithm is used for computing

- a) Closest distance between two points
- b) Convex hull
- c) Area of a polygon
- d) Shortest path between two points

**Answer:** b

**Explanation:** Chan's algorithm is an output-sensitive algorithm used to compute the convex hull set of n points in a 2D or 3D space. Closest pair algorithm is used to compute the closest distance between two points.

2. What is the running time of Chan's algorithm?

- a)  $O(\log n)$
- b)  $O(n \log n)$
- c)  $O(n \log h)$
- d)  $O(\log h)$

**Answer:** c

**Explanation:** The running time of Chan's algorithm is calculated to be  $O(n \log h)$  where h is the number of vertices of the convex hull.

3. Who formulated Chan's algorithm?

- a) Timothy
- b) Kirkpatrick
- c) Frank Nielsen
- d) Seidel

**Answer:** a

**Explanation:** Chan's algorithm was formulated by Timothy Chan. Kirkpatrick and Seidel formulated the Kirkpatrick-Seidel algorithm. Frank Nielsen developed a paradigm relating to Chan's algorithm.

4. The running time of Chan's algorithm is obtained from combining two algorithms.

- a) True
- b) False

**Answer:** a

**Explanation:** The  $O(n \log h)$  running time of Chan's algorithm is obtained by combining the running time of Graham's scan [ $O(n \log n)$ ] and Jarvis match [ $O(nh)$ ].

5. Which of the following is called the “ultimate planar convex hull algorithm”?

- a) Chan's algorithm
- b) Kirkpatrick-Seidel algorithm
- c) Gift wrapping algorithm
- d) Jarvis algorithm

**Answer:** b

**Explanation:** Kirkpatrick-Seidel algorithm is called as the ultimate planar convex hull algorithm. Its running time is the same as that of Chan's algorithm (i.e.)  $O(n \log h)$ .

6. Which of the following algorithms is the simplest?

- a) Chan's algorithm
- b) Kirkpatrick-Seidel algorithm
- c) Gift wrapping algorithm
- d) Jarvis algorithm

**Answer:** a

**Explanation:** Chan's algorithm is very practical for moderate sized problems whereas Kirkpatrick-Seidel algorithm is not. Although, they both have the same running time. Gift wrapping algorithm is a non-output sensitive algorithm and has a longer running time.

7. What is the running time of Hershberger algorithm?

- a)  $O(\log n)$
- b)  $O(n \log n)$
- c)  $O(n \log h)$
- d)  $O(\log h)$

**Answer:** b

**Explanation:** Hershberger's algorithm is an output sensitive algorithm whose running

time was originally  $O(n \log n)$ . He used Chan's algorithm to speed up to  $O(n \log h)$  where  $h$  is the number of edges.

8. Which of the following statements is not a part of Chan's algorithm?

- a) eliminate points not in the hull
- b) recompute convex hull from scratch
- c) merge previously calculated convex hull
- d) reuse convex hull from the previous iteration

**Answer:** b

**Explanation:** Chan's algorithm implies that the convex hulls of larger points can be arrived at by merging previously calculated convex hulls. It makes the algorithm simpler instead of recomputing every time from scratch.

9. Which of the following factors account more to the cost of Chan's algorithm?

- a) computing a single convex hull
- b) locating points that constitute a hull
- c) computing convex hull in groups
- d) merging convex hulls

**Answer:** c

**Explanation:** The majority of the cost of the algorithm lies in the pre-processing (i.e.) computing convex hull in groups. To reduce cost, we reuse convex hulls from previous iterations.

10. Chan's algorithm can be used to compute the lower envelope of a trapezoid.

- a) true
- b) false

**Answer:** a

**Explanation:** An extension of Chan's algorithm can be used for proving solutions to complex problems like computing the lower envelope  $L(S)$  where  $S$  is a set of ' $n$ ' line segments in a trapezoid.

1. Which of the following is false in the case of a spanning tree of a graph G?
- It is tree that spans G
  - It is a subgraph of the G
  - It includes every vertex of the G
  - It can be either cyclic or acyclic

**Answer:** d

**Explanation:** A graph can have many spanning trees. Each spanning tree of a graph G is a subgraph of the graph G, and spanning trees include every vertex of the graph. Spanning trees are always acyclic.

2. Every graph has only one minimum spanning tree.
- True
  - False

**Answer:** b

**Explanation:** Minimum spanning tree is a spanning tree with the lowest cost among all the spanning trees. Sum of all of the edges in the spanning tree is the cost of the spanning tree. There can be many minimum spanning trees for a given graph.

3. Consider a complete graph G with 4 vertices. The graph G has \_\_\_\_\_ spanning trees.
- 15
  - 8
  - 16
  - 13

**Answer:** c

**Explanation:** A graph can have many spanning trees. And a complete graph with n vertices has  $n^{(n-2)}$  spanning trees. So, the complete graph with 4 vertices has  $4^{(4-2)} = 16$  spanning trees.

4. The travelling salesman problem can be solved using \_\_\_\_\_
- A spanning tree
  - A minimum spanning tree
  - Bellman – Ford algorithm
  - DFS traversal

**Answer:** b

**Explanation:** In the travelling salesman problem we have to find the shortest possible route that visits every city exactly once and returns to the starting point for the given a set of cities. So, travelling salesman problem can be solved by contracting the minimum spanning tree.

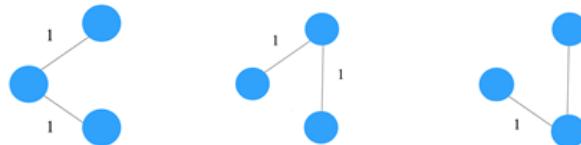
5. Consider the graph M with 3 vertices. Its adjacency matrix is shown below. Which of the following is true?

$$M = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

- Graph M has no minimum spanning tree
- Graph M has a unique minimum spanning tree of cost 2
- Graph M has 3 distinct minimum spanning trees, each of cost 2
- Graph M has 3 spanning trees of different costs

**Answer:** c

**Explanation:** Here all non-diagonal elements in the adjacency matrix are 1. So, every vertex is connected every other vertex of the graph. And, so graph M has 3 distinct minimum spanning trees.



6. Consider a undirected graph G with vertices { A, B, C, D, E }. In graph G, every edge has distinct weight. Edge CD is edge with minimum weight and edge AB is edge with maximum weight. Then, which of the following is false?

- Every minimum spanning tree of G must contain CD
- If AB is in a minimum spanning tree, then its removal must disconnect G

- c) No minimum spanning tree contains AB  
d) G has a unique minimum spanning tree

**Answer:** c

**Explanation:** Every MST will contain CD as it is smallest edge. So, Every minimum spanning tree of G must contain CD is true. And G has a unique minimum spanning tree is also true because the graph has edges with distinct weights. So, no minimum spanning tree contains AB is false.

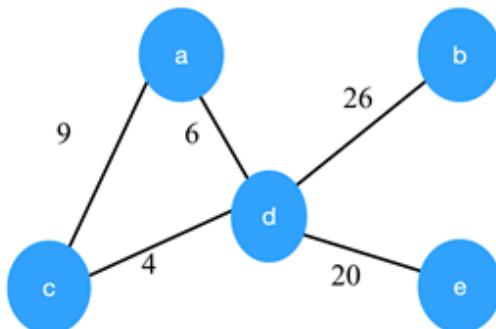
7. If all the weights of the graph are positive, then the minimum spanning tree of the graph is a minimum cost subgraph.

- a) True  
b) False

**Answer:** a

**Explanation:** A subgraph is a graph formed from a subset of the vertices and edges of the original graph. And the subset of vertices includes all endpoints of the subset of the edges. So, we can say MST of a graph is a subgraph when all weights in the original graph are positive.

8. Consider the graph shown below. Which of the following are the edges in the MST of the given graph?

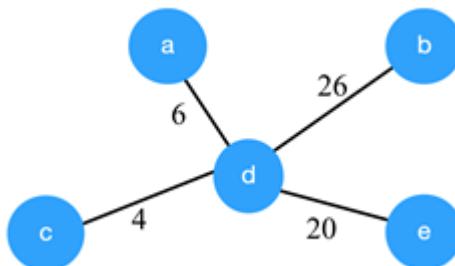


- a) (a-c)(c-d)(d-b)(d-b)  
b) (c-a)(a-d)(d-b)(d-e)  
c) (a-d)(d-c)(d-b)(d-e)  
d) (c-a)(a-d)(d-c)(d-b)(d-e)

**Answer:** c

**Explanation:** The minimum spanning tree of the given graph is shown below. It has cost

56.



9. Which of the following is not the algorithm to find the minimum spanning tree of the given graph?

- a) Boruvka's algorithm  
b) Prim's algorithm  
c) Kruskal's algorithm  
d) Bellman-Ford algorithm

**Answer:** d

**Explanation:** The Boruvka's algorithm, Prim's algorithm and Kruskal's algorithm are the algorithms that can be used to find the minimum spanning tree of the given graph. The Bellman-Ford algorithm is used to find the shortest path from the single source to all other vertices.

10. Which of the following is false?

- a) The spanning trees do not have any cycles  
b) MST have  $n - 1$  edges if the graph has n edges  
c) Edge e belonging to a cut of the graph if has the weight smaller than any other edge in the same cut, then the edge e is present in all the MSTs of the graph  
d) Removing one edge from the spanning tree will not make the graph disconnected

**Answer:** d

**Explanation:** Every spanning tree has  $n - 1$  edges if the graph has n edges and has no cycles. The MST follows the cut property, Edge e belonging to a cut of the graph if has the weight smaller than any other edge in the same cut, then the edge e is present in all the MSTs of the graph.

1. Kruskal's algorithm is used to \_\_\_\_\_
- find minimum spanning tree
  - find single source shortest path
  - find all pair shortest path algorithm
  - traverse the graph

**Answer:** a

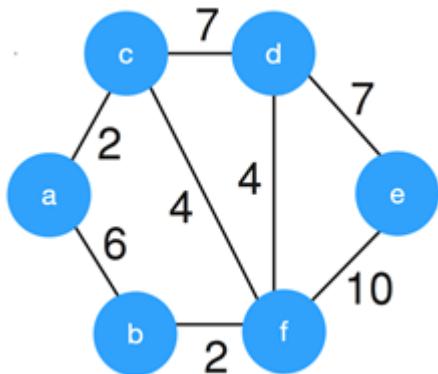
**Explanation:** The Kruskal's algorithm is used to find the minimum spanning tree of the connected graph. It constructs the MST by finding the edge having the least possible weight that connects two trees in the forest.

2. Kruskal's algorithm is a \_\_\_\_\_
- divide and conquer algorithm
  - dynamic programming algorithm
  - greedy algorithm
  - approximation algorithm

**Answer:** c

**Explanation:** Kruskal's algorithm uses a greedy algorithm approach to find the MST of the connected weighted graph. In the greedy method, we attempt to find an optimal solution in stages.

3. Consider the given graph.



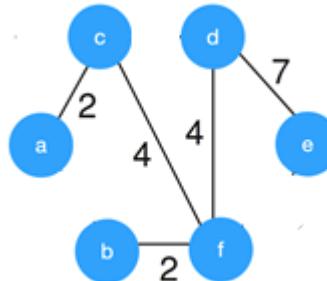
What is the weight of the minimum spanning tree using the Kruskal's algorithm?

- 24
- 23
- 15
- 19

**Answer:** d

**Explanation:** Kruskal's algorithm constructs the minimum spanning tree by constructing

by adding the edges to spanning tree one-by-one. The MST for the given graph is,



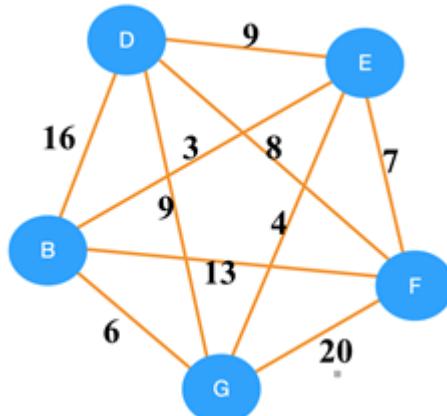
So, the weight of the MST is 19.

4. What is the time complexity of Kruskal's algorithm?
- $O(\log V)$
  - $O(E \log V)$
  - $O(E^2)$
  - $O(V \log E)$

**Answer:** b

**Explanation:** Kruskal's algorithm involves sorting of the edges, which takes  $O(E \log E)$  time, where  $E$  is a number of edges in graph and  $V$  is the number of vertices. After sorting, all edges are iterated and union-find algorithm is applied. union-find algorithm requires  $O(\log V)$  time. So, overall Kruskal's algorithm requires  $O(E \log V)$  time.

5. Consider the following graph. Using Kruskal's algorithm, which edge will be selected first?



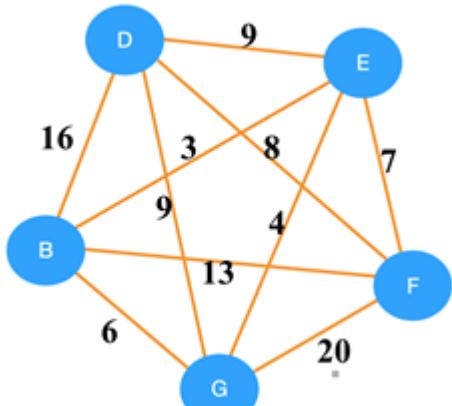
- GF
- DE

- c) BE  
d) BG

**Answer:** c

**Explanation:** In Kruskal's algorithm the edges are selected and added to the spanning tree in increasing order of their weights. Therefore, the first edge selected will be the minimal one. So, correct option is BE.

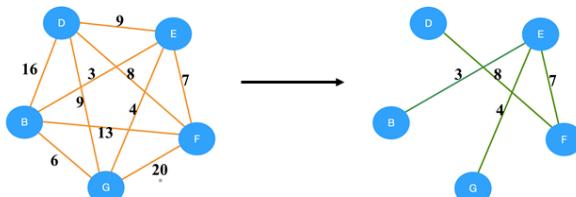
6. Which of the following edges form minimum spanning tree on the graph using kruskals algorithm?



- a) (B-E)(G-E)(E-F)(D-F)  
b) (B-E)(G-E)(E-F)(B-G)(D-F)  
c) (B-E)(G-E)(E-F)(D-E)  
d) (B-E)(G-E)(E-F)(D-F)(D-G)

**Answer:** a

**Explanation:** Using Krushkal's algorithm on the given graph, the generated minimum spanning tree is shown below.



So, the edges in the MST are, (B-E)(G-E)(E-F)(D-F).

7. Which of the following is true?

- a) Prim's algorithm can also be used for disconnected graphs  
b) Kruskal's algorithm can also run on the

- disconnected graphs  
c) Prim's algorithm is simpler than Kruskal's algorithm  
d) In Kruskal's sort edges are added to MST in decreasing order of their weights

**Answer:** b

**Explanation:** Prim's algorithm iterates from one node to another, so it can not be applied for disconnected graph. Kruskal's algorithm can be applied to the disconnected graphs to construct the minimum cost forest. Kruskal's algorithm is comparatively easier and simpler than prim's algorithm.

8. Which of the following is false about the Kruskal's algorithm?

- a) It is a greedy algorithm  
b) It constructs MST by selecting edges in increasing order of their weights  
c) It can accept cycles in the MST  
d) It uses union-find data structure

**Answer:** c

**Explanation:** Kruskal's algorithm is a greedy algorithm to construct the MST of the given graph. It constructs the MST by selecting edges in increasing order of their weights and rejects an edge if it may form the cycle. So, using Kruskal's algorithm is never formed.

9. Kruskal's algorithm is best suited for the dense graphs than the prim's algorithm.

- a) True  
b) False

**Answer:** b

**Explanation:** Prim's algorithm outperforms the Kruskal's algorithm in case of the dense graphs. It is significantly faster if graph has more edges than the Kruskal's algorithm.

10. Consider the following statements.

- S1. Kruskal's algorithm might produce a non-minimal spanning tree.  
S2. Kruskal's algorithm can efficiently implemented using the disjoint-set data structure.

- a) S1 is true but S2 is false
- b) Both S1 and S2 are false
- c) Both S1 and S2 are true
- d) S2 is true but S1 is false

**Answer:** d

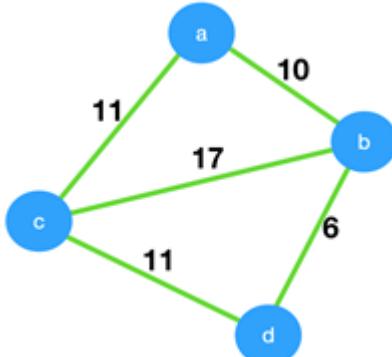
**Explanation:** In Kruskal's algorithm, the disjoint-set data structure efficiently identifies the components containing a vertex and adds the new edges. And Kruskal's algorithm always finds the MST for the connected graph.

1. Which of the following is true?
- a) Prim's algorithm initialises with a vertex
  - b) Prim's algorithm initialises with a edge
  - c) Prim's algorithm initialises with a vertex which has smallest edge
  - d) Prim's algorithm initialises with a forest

**Answer:** a

**Explanation:** Steps in Prim's algorithm: (I) Select any vertex of given graph and add it to MST (II) Add the edge of minimum weight from a vertex not in MST to the vertex in MST; (III) If MST is complete the stop, otherwise go to step (II).

2. Consider the given graph.

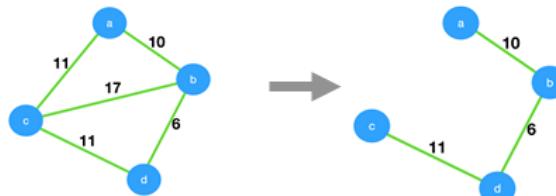


What is the weight of the minimum spanning tree using the Prim's algorithm, starting from vertex a?

- a) 23
- b) 28
- c) 27
- d) 11

**Answer:** c

**Explanation:** In Prim's algorithm, we select a vertex and add it to the MST. Then we add the minimum edge from the vertex in MST to vertex not in MST. From, figure shown below weight of MST = 27.



3. Worst case is the worst case time complexity of Prim's algorithm if adjacency matrix is used?

- a)  $O(\log V)$
- b)  $O(V^2)$
- c)  $O(E^2)$
- d)  $O(V \log E)$

**Answer:** b

**Explanation:** Use of adjacency matrix provides the simple implementation of the Prim's algorithm. In Prim's algorithm, we need to search for the edge with a minimum weight for that vertex. So, worst case time complexity will be  $O(V^2)$ , where V is the number of vertices.

4. Prim's algorithm is a \_\_\_\_\_

- a) Divide and conquer algorithm
- b) Greedy algorithm
- c) Dynamic Programming
- d) Approximation algorithm

**Answer:** b

**Explanation:** Prim's algorithm uses a greedy algorithm approach to find the MST of the connected weighted graph. In greedy method, we attempt to find an optimal solution in stages.

5. Prim's algorithm resembles Dijkstra's algorithm.

- a) True
- b) False

**Answer:** a

**Explanation:** In Prim's algorithm, the MST is constructed starting from a single vertex and adding in new edges to the MST that link the partial tree to a new vertex outside of the MST. And Dijkstra's algorithm also rely on the similar approach of finding the next closest vertex. So, Prim's algorithm resembles Dijkstra's algorithm.

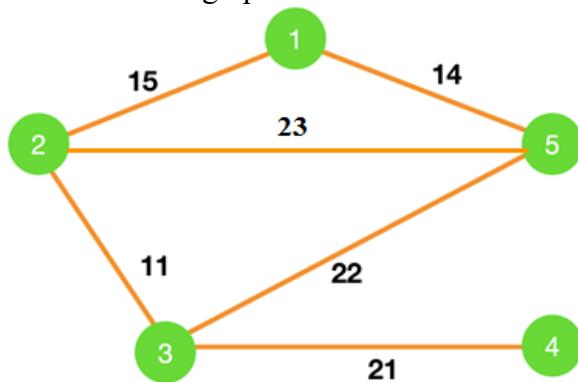
6. Kruskal's algorithm is best suited for the sparse graphs than the prim's algorithm.

- a) True
- b) False

**Answer:** a

**Explanation:** Prim's algorithm and Kruskal's algorithm perform equally in case of the sparse graphs. But Kruskal's algorithm is simpler and easy to work with. So, it is best suited for sparse graphs.

7. Consider the graph shown below.

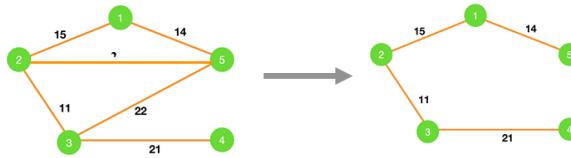


Which of the following edges form the MST of the given graph using Prim'a algorithm, starting from vertex 4.

- a) (4-3)(5-3)(2-3)(1-2)
- b) (4-3)(3-5)(5-1)(1-2)
- c) (4-3)(3-5)(5-2)(1-5)
- d) (4-3)(3-2)(2-1)(1-5)

**Answer:** d

**Explanation:** The MST for the given graph using Prim's algorithm starting from vertex 4 is,



So, the MST contains edges (4-3)(3-2)(2-1)(1-5).

8. Prim's algorithm is also known as

- a) Dijkstra–Scholten algorithm
- b) Boruvka's algorithm
- c) Floyd–Warshall algorithm
- d) DJP Algorithm

**Answer:** d

**Explanation:** The Prim's algorithm was developed by Vojtěch Jarník and it was latter discovered by the duo Prim and Dijkstra. Therefore, Prim's algorithm is also known as DJP Algorithm.

9. Prim's algorithm can be efficiently implemented using \_\_\_\_\_ for graphs with greater density.

- a) d-ary heap
- b) linear search
- c) fibonacci heap
- d) binary search

**Answer:** a

**Explanation:** In Prim's algorithm, we add the minimum weight edge for the chosen vertex which requires searching on the array of weights. This searching can be efficiently implemented using binary heap for dense graphs. And for graphs with greater density, Prim's algorithm can be made to run in linear time using d-ary heap(generalization of binary heap).

10. Which of the following is false about Prim's algorithm?

- a) It is a greedy algorithm
- b) It constructs MST by selecting edges in increasing order of their weights
- c) It never accepts cycles in the MST
- d) It can be implemented using the Fibonacci heap

**Answer:** b

**Explanation:** Prim's algorithm can be implemented using Fibonacci heap and it never accepts cycles. And Prim's algorithm follows greedy approach. Prim's algorithms span from one vertex to another.

1. Dijkstra's Algorithm is used to solve \_\_\_\_\_ problems.

- a) All pair shortest path
- b) Single source shortest path
- c) Network flow
- d) Sorting

**Answer:** b

**Explanation:** Dijkstra's Algorithm is used for solving single source shortest path problems. In this algorithm, a single node is fixed as a source node and shortest paths from this node to all other nodes in graph is found.

2. Which of the following is the most commonly used data structure for implementing Dijkstra's Algorithm?

- a) Max priority queue
- b) Stack
- c) Circular queue
- d) Min priority queue

**Answer:** d

**Explanation:** Minimum priority queue is the most commonly used data structure for implementing Dijkstra's Algorithm because the required operations to be performed in Dijksta's Algorithm match with specialty of a minimum priority queue.

3. What is the time complexity of Dijikstra's algorithm?

- a) O(N)
- b) O( $N^3$ )
- c) O( $N^2$ )
- d) O(logN)

**Answer:** c

**Explanation:** Time complexity of Dijksta's

algorithm is  $O(N^2)$  because of the use of doubly nested for loops. It depends on how the table is manipulated.

4. Dijkstra's Algorithm cannot be applied on

- a) Directed and weighted graphs
- b) Graphs having negative weight function
- c) Unweighted graphs
- d) Undirected and unweighted graphs

**Answer:** b

**Explanation:** Dijksta's Algorithm cannot be applied on graphs having negative weight function because calculation of cost to reach a destination node from the source node becomes complex.

5. What is the pseudo code to compute the shortest path in Dijksta's algorithm?

a)

```
if(!T[w].Known)
    if(T[v].Dist + C(v,w) < T[w].Dist
) {
    Decrease(T[w].Dist to T[
v].Dist +C(v,w));
    T[w].path=v; }
```

b)

```
if(T[w].Known)
    if(T[v].Dist + C(v,w) < T[w].Dist
) {
        Increase (T[w].Dist to T
[v].Dist +C(v,w));
        T[w].path=v; }
```

c)

```
if(!T[w].Known)
    if(T[v].Dist + C(v,w) > T[w].Dist
) {
        Decrease(T[w].Dist to T[
v].Dist +C(v,w));
        T[w].path=v; }
```

d)

```
if(T[w].Known)
    if(T[v].Dist + C(v,w) < T[w].Dist
) {
```

```

    Increase(T[w].Dist to T[
v].Dist);
T[w].path=v; }

```

**Answer:** a

**Explanation:** If the known value of the adjacent vertex(w) is not set then check whether the sum of distance from source vertex(v) and cost to travel from source to adjacent vertex is less than the existing distance of the adjacent node. If so, perform decrease key operation.

6. How many priority queue operations are involved in Dijkstra's Algorithm?

- a) 1
- b) 3
- c) 2
- d) 4

**Answer:** b

**Explanation:** The number of priority queue operations involved is 3. They are insert, extract-min and decrease key.

7. How many times the insert and extract min operations are invoked per vertex?

- a) 1
- b) 2
- c) 3
- d) 0

**Answer:** a

**Explanation:** Insert and extract min operations are invoked only once per vertex because each vertex is added only once to the set and each edge in the adjacency list is examined only once during the course of algorithm.

8. The maximum number of times the decrease key operation performed in Dijkstra's algorithm will be equal to

- 
- a) Total number of vertices
  - b) Total number of edges

- c) Number of vertices – 1
- d) Number of edges – 1

**Answer:** b

**Explanation:** If the total number of edges in all adjacency list is E, then there will be a total of E number of iterations, hence there will be a total of at most E decrease key operations.

9. What is running time of Dijkstra's algorithm using Binary min- heap method?

- a) O(V)
- b) O(VlogV)
- c) O(E)
- d) O(ElogV)

**Answer:** d

**Explanation:** Time required to build a binary min heap is O(V). Each decrease key operation takes O(logV) and there are still at most E such operations. Hence total running time is O(ElogV).

10. The running time of Bellmann Ford algorithm is lower than that of Dijkstra's Algorithm.

- a) True
- b) False

**Answer:** b

**Explanation:** The number of iterations involved in Bellmann Ford Algorithm is more than that of Dijkstra's Algorithm.

11. Dijkstra's Algorithm run on a weighted, directed graph  $G=\{V,E\}$  with non-negative weight function w and source s, terminates with  $d[u]=\delta(s,u)$  for all vertices u in V.

- a) True
- b) False

**Answer:** a

**Explanation:** The equality  $d[u]=\delta(s,u)$  holds good when vertex u is added to set S and this equality is maintained thereafter by the upper bound property.

12. Given pseudo code of Dijkstra's Algorithm.

```

1. //Initialise single source(G,s)
2. S=0
3. Q=V[G]
4. While Q != 0
5.   Do u=extract-min(Q)
6.     S=S union {u}
7.     For each vertex v in adj[u]
8.       Do relax(u,v,w)
  
```

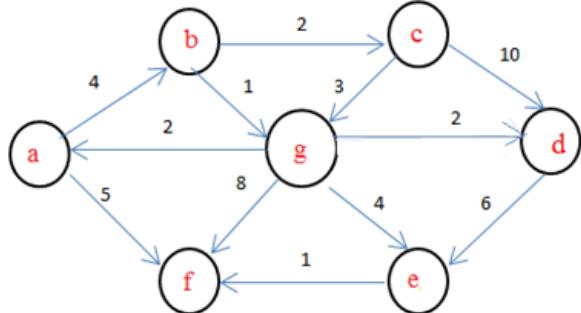
What happens when while loop in line 4 is changed to while  $Q > 1$ ?

- a) While loop gets executed for  $v$  times
- b) While loop gets executed for  $v-1$  times
- c) While loop gets executed only once
- d) While loop does not get executed

**Answer:** b

**Explanation:** In the normal execution of Dijkstra's Algorithm, the while loop gets executed  $V$  times. The change in the while loop statement causes it to execute only  $V - 1$  times.

13. Consider the following graph.



If b is the source vertex, what is the minimum cost to reach f vertex?

- a) 8
- b) 9
- c) 4
- d) 6

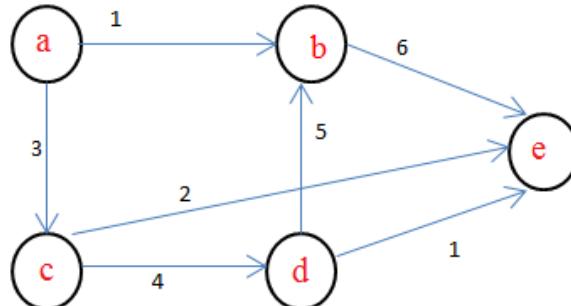
**Answer:** d

**Explanation:** The minimum cost to reach f vertex from b vertex is 6 by having vertices g and e as intermediates.

b to g, cost is 1

g to e, cost is 4  
e to f, cost is 1  
hence total cost  $1+4+1=6$ .

14. In the given graph:



Identify the shortest path having minimum cost to reach vertex E if A is the source vertex

- a) a-b-e
- b) a-c-e
- c) a-c-d-e
- d) a-c-d-b-e

**Answer:** b

**Explanation:** The minimum cost required to travel from vertex A to E is via vertex C  
A to C, cost= 3  
C to E, cost= 2  
Hence the total cost is 5.

15. Dijkstra's Algorithm is the prime example for \_\_\_\_\_

- a) Greedy algorithm
- b) Branch and bound
- c) Back tracking
- d) Dynamic programming

**Answer:** a

**Explanation:** Dijkstra's Algorithm is the prime example for greedy algorithms because greedy algorithms generally solve a problem in stages by doing what appears to be the best thing at each stage.

1. The Bellmann Ford algorithm returns \_\_\_\_\_ value.

- a) Boolean
- b) Integer

- c) String
- d) Double

**Answer:** a

**Explanation:** The Bellmann Ford algorithm returns Boolean value whether there is a negative weight cycle that is reachable from the source.

2. Bellmann ford algorithm provides solution for \_\_\_\_\_ problems.

- a) All pair shortest path
- b) Sorting
- c) Network flow
- d) Single source shortest path

**Answer:** d

**Explanation:** Bellmann ford algorithm is used for finding solutions for single source shortest path problems. If the graph has no negative cycles that are reachable from the source then the algorithm produces the shortest paths and their weights.

3. Bellmann Ford algorithm is used to indicate whether the graph has negative weight cycles or not.

- a) True
- b) False

**Answer:** a

**Explanation:** Bellmann Ford algorithm returns true if the graph does not have any negative weight cycles and returns false when the graph has negative weight cycles.

4. How many solution/solutions are available for a graph having negative weight cycle?

- a) One solution
- b) Two solutions
- c) No solution
- d) Infinite solutions

**Answer:** c

**Explanation:** If the graph has any negative weight cycle then the algorithm indicates that no solution exists for that graph.

5. What is the running time of Bellmann Ford Algorithm?

- a)  $O(V)$
- b)  $O(V^2)$
- c)  $O(E \log V)$
- d)  $O(VE)$

**Answer:** d

**Explanation:** Bellmann Ford algorithm runs in time  $O(VE)$ , since the initialization takes  $O(V)$  for each of  $V-1$  passes and the for loop in the algorithm takes  $O(E)$  time. Hence the total time taken by the algorithm is  $O(VE)$ .

6. How many times the for loop in the Bellmann Ford Algorithm gets executed?

- a)  $V$  times
- b)  $V-1$
- c)  $E$
- d)  $E-1$

**Answer:** b

**Explanation:** The for loop in the Bellmann Ford Algorithm gets executed for  $V-1$  times. After making  $V-1$  passes, the algorithm checks for a negative weight cycle and returns appropriate boolean value.

7. Dijikstra's Algorithm is more efficient than Bellmann Ford Algorithm.

- a) True
- b) False

**Answer:** a

**Explanation:** The running time of Bellmann Ford Algorithm is  $O(VE)$  whereas Dijikstra's Algorithm has running time of only  $O(V^2)$ .

8. Identify the correct Bellmann Ford Algorithm.

- a)

```

for i=1 to V[g]-1
    do for each edge (u,v) in E[g]
        do Relax(u,v,w)
    for each edge (u,v) in E[g]
        do if d[v]>d[u]+w(u,v)
            then return False
return True
  
```

b)

```

for i=1 to V[g]-1
    for each edge (u,v) in E[g]
        do if d[v]>d[u]+w(u,v)
            then return False
    return True

```

c)

```

for i=1 to V[g]-1
    do for each edge (u,v) in E[g]
        do Relax(u,v,w)
    for each edge (u,v) in E[g]
        do if d[v]<d[u]+w(u,v)
            then return true
    return True

```

d)

```

for i=1 to V[g]-1
    do for each edge (u,v) in E[g]
        do Relax(u,v,w)
    return True

```

**Answer:** a

**Explanation:** After initialization, the algorithm makes  $V-1$  passes over the edges of the graph. Each pass is one iteration of the for loop and consists of relaxing each edge of the graph once. Then it checks for the negative weight cycle and returns an appropriate Boolean value.

9. What is the basic principle behind Bellmann Ford Algorithm?

- a) Interpolation
- b) Extrapolation
- c) Regression
- d) Relaxation

**Answer:** d

**Explanation:** Relaxation methods which are also called as iterative methods in which an approximation to the correct distance is replaced progressively by more accurate values till an optimum solution is found.

10. Bellmann Ford Algorithm can be applied for \_\_\_\_\_

- a) Undirected and weighted graphs
- b) Undirected and unweighted graphs
- c) Directed and weighted graphs
- d) All directed graphs

**Answer:** c

**Explanation:** Bellmann Ford Algorithm can be applied for all directed and weighted graphs. The weight function in the graph may either be positive or negative.

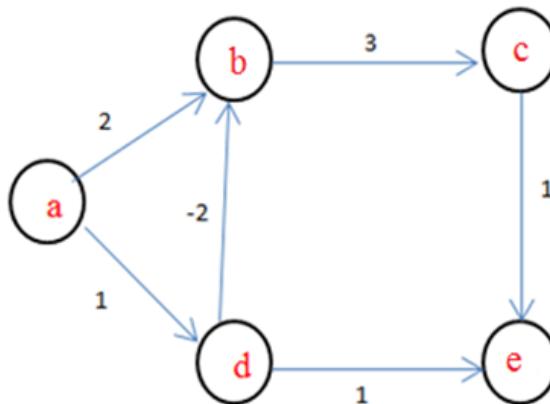
11. Bellmann Ford algorithm was first proposed by \_\_\_\_\_

- a) Richard Bellmann
- b) Alfonso Shimbe
- c) Lester Ford Jr
- d) Edward F. Moore

**Answer:** b

**Explanation:** Alfonso Shimbe proposed Bellmann Ford algorithm in the year 1955. Later it was published by Richard Bellmann in 1957 and Lester Ford Jr in the year 1956. Hence it is called Bellmann Ford Algorithm.

12. Consider the following graph:



What is the minimum cost to travel from node A to node C

- a) 5
- b) 2
- c) 1
- d) 3

**Answer:** b

**Explanation:** The minimum cost to travel from node A to node C is 2.

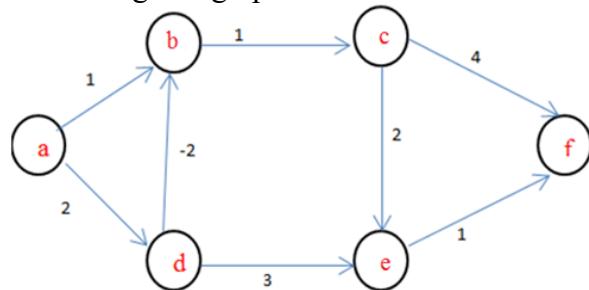
A-D, cost=1

D-B, cost=-2

B-C, cost=3

Hence the total cost is 2.

13. In the given graph:



Identify the path that has minimum cost to travel from node a to node f

- a) a-b-c-f
- b) a-d-e-f
- c) a-d-b-c-f
- d) a-d-b-c-e-f

**Answer:** d

**Explanation:** The minimum cost taken by the path a-d-b-c-e-f is 4.

a-d, cost=2

d-b, cost=-2

b-c, cost=1

c-e, cost= 2

e-f, cost=1

Hence the total cost is 4.

14. Bellmann Ford Algorithm is an example for

- a) Dynamic Programming
- b) Greedy Algorithms
- c) Linear Programming
- d) Branch and Bound

**Answer:** a

**Explanation:** In Bellmann Ford Algorithm the shortest paths are calculated in bottom up manner which is similar to other dynamic programming problems.

15. A graph is said to have a negative weight cycle when?

- a) The graph has 1 negative weighted edge
- b) The graph has a cycle
- c) The total weight of the graph is negative
- d) The graph has 1 or more negative weighted edges

**Answer:** c

**Explanation:** When the total weight of the graph sums up to a negative number then the graph is said to have a negative weight cycle. Bellmann Ford Algorithm provides no solution for such graphs.

1. Floyd Warshall's Algorithm is used for solving \_\_\_\_\_

- a) All pair shortest path problems
- b) Single Source shortest path problems
- c) Network flow problems
- d) Sorting problems

**Answer:** a

**Explanation:** Floyd Warshall's Algorithm is used for solving all pair shortest path problems. It means the algorithm is used for finding the shortest paths between all pairs of vertices in a graph.

2. Floyd Warshall's Algorithm can be applied on \_\_\_\_\_

- a) Undirected and unweighted graphs
- b) Undirected graphs
- c) Directed graphs
- d) Acyclic graphs

**Answer:** c

**Explanation:** Floyd Warshall Algorithm can be applied in directed graphs. From a given directed graph, an adjacency matrix is framed and then all pair shortest path is computed by the Floyd Warshall Algorithm.

3. What is the running time of the Floyd Warshall Algorithm?

- a) Big-oh( $V$ )
- b) Theta( $V^2$ )

- c) Big-Oh(VE)
- d) Theta( $V^3$ )

**Answer:** d

**Explanation:** The running time of the Floyd Warshall algorithm is determined by the triply nested for loops. Since each execution of the for loop takes  $O(1)$  time, the algorithm runs in time  $\Theta(V^3)$ .

4. What approach is being followed in Floyd Warshall Algorithm?
- a) Greedy technique
  - b) Dynamic Programming
  - c) Linear Programming
  - d) Backtracking

**Answer:** b

**Explanation:** Floyd Warshall Algorithm follows dynamic programming approach because all pair shortest paths are computed in bottom up manner.

5. Floyd Warshall Algorithm can be used for finding \_\_\_\_\_
- a) Single source shortest path
  - b) Topological sort
  - c) Minimum spanning tree
  - d) Transitive closure

**Answer:** d

**Explanation:** One of the ways to compute the transitive closure of a graph in  $\Theta(N^3)$  time is to assign a weight of 1 to each edge of E and then run the Floyd Warshall Algorithm.

6. What procedure is being followed in Floyd Warshall Algorithm?
- a) Top down
  - b) Bottom up
  - c) Big bang
  - d) Sandwich

**Answer:** b

**Explanation:** Bottom up procedure is being used to compute the values of the matrix elements  $dij(k)$ . The input is an  $n \times n$  matrix.

The procedure returns the matrix  $D(n)$  of the shortest path weights.

7. Floyd- Warshall algorithm was proposed by \_\_\_\_\_
- a) Robert Floyd and Stephen Warshall
  - b) Stephen Floyd and Robert Warshall
  - c) Bernad Floyd and Robert Warshall
  - d) Robert Floyd and Bernad Warshall

**Answer:** a

**Explanation:** Floyd- Warshall Algorithm was proposed by Robert Floyd in the year 1962. The same algorithm was proposed by Stephen Warshall during the same year for finding the transitive closure of the graph.

8. Who proposed the modern formulation of Floyd-Warshall Algorithm as three nested loops?
- a) Robert Floyd
  - b) Stephen Warshall
  - c) Bernard Roy
  - d) Peter Ingerman

**Answer:** d

**Explanation:** The modern formulation of Floyd-Warshall Algorithm as three nested for-loops was proposed by Peter Ingerman in the year 1962.

9. Complete the program.

```

n=rows[W]
D(0)=W
for k=1 to n
    do for i=1 to n
        do for j=1 to n
            do _____
_____
return D(n)

```

- a)  $dij(k) = \min(dij(k-1), dik(k-1) - dkj(k-1))$
- b)  $dij(k) = \max(dij(k-1), dik(k-1) - dkj(k-1))$
- c)  $dij(k) = \min(dij(k-1), dik(k-1) + dkj(k-1))$
- d)  $dij(k) = \max(dij(k-1), dik(k-1) + dkj(k-1))$

**Answer:** c

**Explanation:** In order to compute the shortest path from vertex i to vertex j, we need to find

the minimum of 2 values which are  $dij(k-1)$  and sum of  $dik(k-1)$  and  $dkj(k-1)$ .

10. What happens when the value of k is 0 in the Floyd Warshall Algorithm?

- a) 1 intermediate vertex
- b) 0 intermediate vertex
- c) N intermediate vertices
- d) N-1 intermediate vertices

**Answer:** b

**Explanation:** When  $k=0$ , a path from vertex i to vertex j has no intermediate vertices at all. Such a path has at most one edge and hence  $dij(0) = w_{ij}$ .

11. Using logical operator's instead arithmetic operators saves time and space.

- a) True
- b) False

**Answer:** a

**Explanation:** In computers, logical operations on single bit values execute faster than arithmetic operations on integer words of data.

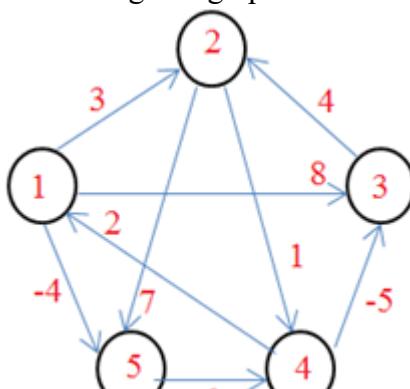
12. The time taken to compute the transitive closure of a graph is  $\Theta(n^2)$ .

- a) True
- b) False

**Answer:** b

**Explanation:** The time taken to compute the transitive closure of a graph is  $\Theta(n^3)$ . Transitive closure can be computed by assigning weight of 1 to each edge and by running Floyd Warshall Algorithm.

13. In the given graph



What is the minimum cost to travel from vertex 1 to vertex 3?

- a) 3
- b) 2
- c) 10
- d) -3

**Answer:** d

**Explanation:** The minimum cost required to travel from node 1 to node 5 is -3.

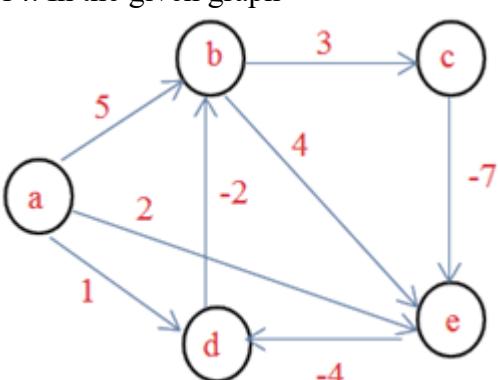
1-5, cost is -4

5-4, cost is 6

4-3, cost is -5

Hence total cost is  $-4 + 6 + -5 = -3$ .

14. In the given graph



How many intermediate vertices are required to travel from node a to node e at a minimum cost?

- a) 2
- b) 0
- c) 1
- d) 3

**Answer:** c

**Explanation:** The minimum cost to travel from node a to node e is 1 by passing via nodes b and c.

a-b, cost 5

b-c, cost 3

c-e, cost -7

Hence the total cost is 1.

15. What is the formula to compute the transitive closure of a graph?

- a)  $tij(k) = tij(k-1) \text{ AND } (tik(k-1) \text{ OR } tkj(k-1))$
- b)  $tij(k) = tij(k-1) \text{ OR } (tik(k-1) \text{ AND } tkj(k-1))$
- c)  $tij(k) = tij(k-1) \text{ AND } (tik(k-1) \text{ AND } tkj(k-1))$
- d)  $tij(k) = tij(k-1) \text{ OR } (tik(k-1) \text{ OR } tkj(k-1))$

**Answer:** b

**Explanation:** Transitive closure of a graph can be computed by using Floyd Warshall algorithm. This method involves substitution of logical operations (logical OR and logical AND) for arithmetic operations min and + in Floyd Warshall Algorithm.

Floyd Warshall Algorithm:  $dij(k) = \min(dij(k-1), dik(k-1) + dkj(k-1))$

Transitive closure:  $tij(k) = tij(k-1) \text{ OR } (tik(k-1) \text{ AND } tkj(k-1))$ .

### UNIT III DYNAMIC PROGRAMMING AND GREEDY TECHNIQUE

1. Which of the following is/are property/properties of a dynamic programming problem?
  - a) Optimal substructure
  - b) Overlapping subproblems
  - c) Greedy approach
  - d) Both optimal substructure and overlapping subproblems

**Answer:** d

**Explanation:** A problem that can be solved using dynamic programming possesses overlapping subproblems as well as optimal substructure properties.

2. If an optimal solution can be created for a problem by constructing optimal solutions for its subproblems, the problem possesses \_\_\_\_\_ property.

- a) Overlapping subproblems
- b) Optimal substructure
- c) Memoization
- d) Greedy

**Answer:** b

**Explanation:** Optimal substructure is the property in which an optimal solution is found for the problem by constructing optimal solutions for the subproblems.

3. If a problem can be broken into subproblems which are reused several times, the problem possesses \_\_\_\_\_ property.

- a) Overlapping subproblems
- b) Optimal substructure
- c) Memoization
- d) Greedy

**Answer:** a

**Explanation:** Overlapping subproblems is the property in which value of a subproblem is used several times.

4. If a problem can be solved by combining optimal solutions to non-overlapping problems, the strategy is called

- a) Dynamic programming
- b) Greedy
- c) Divide and conquer
- d) Recursion

**Answer:** c

**Explanation:** In divide and conquer, the problem is divided into smaller non-overlapping subproblems and an optimal

solution for each of the subproblems is found. The optimal solutions are then combined to get a global optimal solution. For example, mergesort uses divide and conquer strategy.

5. When dynamic programming is applied to a problem, it takes far less time as compared to other methods that don't take advantage of overlapping subproblems.

- a) True
- b) False

**Answer:** a

**Explanation:** Dynamic programming calculates the value of a subproblem only once, while other methods that don't take advantage of the overlapping subproblems property may calculate the value of the same subproblem several times. So, dynamic programming saves the time of recalculation and takes far less time as compared to other methods that don't take advantage of the overlapping subproblems property.

6. A greedy algorithm can be used to solve all the dynamic programming problems.

- a) True
- b) False

**Answer:** b

**Explanation:** A greedy algorithm gives optimal solution for all subproblems, but when these locally optimal solutions are combined it may NOT result into a globally optimal solution. Hence, a greedy algorithm CANNOT be used to solve all the dynamic programming problems.

7. In dynamic programming, the technique of storing the previously calculated values is called \_\_\_\_\_

- a) Saving value property
- b) Storing value property
- c) Memoization
- d) Mapping

**Answer:** c

**Explanation:** Memoization is the technique

in which previously calculated values are stored, so that, these values can be used to solve other subproblems.

8. When a top-down approach of dynamic programming is applied to a problem, it usually \_\_\_\_\_

- a) Decreases both, the time complexity and the space complexity
- b) Decreases the time complexity and increases the space complexity
- c) Increases the time complexity and decreases the space complexity
- d) Increases both, the time complexity and the space complexity

**Answer:** b

**Explanation:** The top-down approach uses the memoization technique which stores the previously calculated values. Due to this, the time complexity is decreased but the space complexity is increased.

9. Which of the following problems is NOT solved using dynamic programming?

- a) 0/1 knapsack problem
- b) Matrix chain multiplication problem
- c) Edit distance problem
- d) Fractional knapsack problem

**Answer:** d

**Explanation:** The fractional knapsack problem is solved using a greedy algorithm.

10. Which of the following problems should be solved using dynamic programming?

- a) Mergesort
- b) Binary search
- c) Longest common subsequence
- d) Quicksort

**Answer:** c

**Explanation:** The longest common subsequence problem has both, optimal substructure and overlapping subproblems. Hence, dynamic programming should be used to solve this problem.

1. The following sequence is a fibonacci sequence:  
 0, 1, 1, 2, 3, 5, 8, 13, 21,.....  
 Which technique can be used to get the nth fibonacci term?  
 a) Recursion  
 b) Dynamic programming  
 c) A single for loop  
 d) Recursion, Dynamic Programming, For loops

**Answer:** d

**Explanation:** Each of the above mentioned methods can be used to find the nth fibonacci term.

2. Consider the recursive implementation to find the nth fibonacci number:

```
int fibo(int n)
if n <= 1
    return n
return _____
```

Which line would make the implementation complete?

- a) fibo(n) + fibo(n)
- b) fibo(n) + fibo(n - 1)
- c) fibo(n - 1) + fibo(n + 1)
- d) fibo(n - 1) + fibo(n - 2)

**Answer:** d

**Explanation:** Consider the first five terms of the fibonacci sequence: 0,1,1,2,3. The 6th term can be found by adding the two previous terms, i.e.  $fibo(6) = fibo(5) + fibo(4) = 3 + 2 = 5$ . Therefore, the nth term of a fibonacci sequence would be given by:  
 $fibo(n) = fibo(n-1) + fibo(n-2)$ .

3. What is the time complexity of the recursive implementation used to find the nth fibonacci term?
- a) O(1)
  - b)  $O(n^2)$
  - c)  $O(n!)$
  - d) Exponential

**Answer:** d

**Explanation:** The recurrence relation is given by  $fibo(n) = fibo(n - 1) + fibo(n - 2)$ . So, the time complexity is given by:

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

Approximately,

$$T(n) = 2 * T(n - 1)$$

$$= 4 * T(n - 2)$$

$$= 8 * T(n - 3)$$

:

:

$$= 2^k * T(n - k)$$

This recurrence will stop when  $n - k = 0$

i.e.  $n = k$

$$\text{Therefore, } T(n) = 2^n * O(0) = 2^n$$

Hence, it takes exponential time.

It can also be proved by drawing the recursion tree and counting the number of leaves.

4. Suppose we find the 8th term using the recursive implementation. The arguments passed to the function calls will be as follows:

`fibonacci(8)`

`fibonacci(7) + fibonacci(6)`

`fibonacci(6) + fibonacci(5) + fibonacci(5) + fibonacci(4)`

`fibonacci(5) + fibonacci(4) + fibonacci(4) + fibonacci(3) + fibonacci(4) + fibonacci(3) + fibonacci(3) + fibonacci(2)`

:

:

:

Which property is shown by the above function calls?

- a) Memoization
- b) Optimal substructure
- c) Overlapping subproblems
- d) Greedy

**Answer:** c

**Explanation:** From the function calls, we can see that fibonacci(4) is calculated twice and fibonacci(3) is calculated thrice. Thus, the same subproblem is solved many times and hence the function calls show the overlapping subproblems property.

5. What is the output of the following program?

```
#include<stdio.h>
int fibo(int n)
{
    if(n<=1)
        return n;
    return fibo(n-1) + fibo(n-2);
}
int main()
{
    int r = fibo(50000);
    printf("%d",r);
    return 0;
}
```

- a) 1253556389
- b) 5635632456
- c) Garbage value
- d) Runtime error

**Answer:** d

**Explanation:** The value of n is 50000. The function is recursive and it's time complexity is exponential. So, the function will be called almost  $2^{50000}$  times. Now, even though NO variables are stored by the function, the space required to store the addresses of these function calls will be enormous. Stack memory is utilized to store these addresses and only a particular amount of stack memory can be used by any program. So, after a certain function call, no more stack space will be available and it will lead to stack overflow causing runtime error.

6. What is the space complexity of the recursive implementation used to find the nth fibonacci term?

- a) O(1)
- b) O(n)

c)  $O(n^2)$

d)  $O(n^3)$

**Answer:** a

**Explanation:** The recursive implementation doesn't store any values and calculates every value from scratch. So, the space complexity is  $O(1)$ .

7. Consider the following code to find the nth fibonacci term:

```
int fibo(int n)
    if n == 0
        return 0
    else
        prevFib = 0
        curFib = 1
        for i : 1 to n-1
            nextFib = prevFib + curFib
            _____
            _____
return curFib
```

Complete the above code.

a)

prevFib = curFib

curFib = curFib

b)

prevFib = nextFib

curFib = prevFib

c)

prevFib = curFib

curFib = nextFib

d)

prevFib = nextFib

nextFib = curFib

**Answer:** c

**Explanation:** The lines, prevFib = curFib and curFib = nextFib, make the code complete.

8. What is the time complexity of the following for loop method used to compute the nth fibonacci term?

```
int fibo(int n)
{
    if n == 0
        return 0
    else
        prevFib = 0
        curFib = 1
        for i : 1 to n-1
            nextFib = prevFib + curFib
            prevFib = curFib
            curFib = nextFib
        return curFib
```

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) Exponential

**Answer:** b

**Explanation:** To calculate the nth term, the for loop runs  $(n - 1)$  times and each time a for loop is run, it takes a constant time.

Therefore, the time complexity is of the order of n.

9. What is the space complexity of the following for loop method used to compute the nth fibonacci term?

```
int fibo(int n)
{
    if n == 0
        return 0
    else
        prevFib = 0
        curFib = 1
        for i : 1 to n-1
            nextFib = prevFib + curFib
            prevFib = curFib
            curFib = nextFib
        return curFib
```

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) Exponential

**Answer:** a

**Explanation:** To calculate the nth term, we just store the previous term and the current term and then calculate the next term using these two terms. It takes a constant space to store these two terms and hence O(1) is the answer.

10. What will be the output when the following code is executed?

```
#include<stdio.h>
int fibo(int n)
{
    if(n==0)
        return 0;
    int i;
    int prevFib=0,curFib=1;
    for(i=1;i<=n-1;i++)
    {
        int nextFib = prevFib + curFib
    ;
        prevFib = curFib;
        curFib = nextFib;
    }
    return curFib;
}
int main()
{
    int r = fibo(10);
    printf("%d",r);
    return 0;
}
```

- a) 34
- b) 55
- c) Compile error
- d) Runtime error

**Answer:** b

**Explanation:** The output is the 10th fibonacci number, which is 55.

11. Consider the following code to find the nth fibonacci term using dynamic programming:

1. int fibo(int n)
2. int fibo\_terms[100000] //arr to store the fibonacci numbers
3. fibo\_terms[0] = 0
4. fibo\_terms[1] = 1
- 5.

```

6.   for i: 2 to n
7.       fibo_terms[i] = fibo_terms[i - 1]
] + fibo_terms[i - 2]
8.
9.   return fibo_terms[n]

```

Which property is shown by line 7 of the above code?

- a) Optimal substructure
- b) Overlapping subproblems
- c) Both overlapping subproblems and optimal substructure
- d) Greedy substructure

**Answer:** a

**Explanation:** We find the nth fibonacci term by finding previous fibonacci terms, i.e. by solving subproblems. Hence, line 7 shows the optimal substructure property.

12. Consider the following code to find the nth fibonacci term using dynamic programming:

```

1. int fibo(int n)
2.     int fibo_terms[100000] //arr to
    store the fibonacci numbers
3.     fibo_terms[0] = 0
4.     fibo_terms[1] = 1
5.
6.     for i: 2 to n
7.         fibo_terms[i] = fibo_terms[i - 1]
] + fibo_terms[i - 2]
8.
9.     return fibo_terms[n]

```

Which technique is used by line 7 of the above code?

- a) Greedy
- b) Recursion
- c) Memoization
- d) Overlapping subproblems

**Answer:** c

**Explanation:** Line 7 stores the current value that is calculated, so that the value can be used later directly without calculating it from scratch. This is memoization.

13. What is the time complexity of the following dynamic programming

implementation used to compute the nth fibonacci term?

```

1. int fibo(int n)
2.     int fibo_terms[100000] //arr to
    store the fibonacci numbers
3.     fibo_terms[0] = 0
4.     fibo_terms[1] = 1
5.
6.     for i: 2 to n
7.         fibo_terms[i] = fibo_terms[i - 1]
] + fibo_terms[i - 2]
8.
9.     return fibo_terms[n]

```

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) Exponential

**Answer:** b

**Explanation:** To calculate the nth term, the for loop runs ( $n - 1$ ) times and each time a for loop is run, it takes a constant time.

Therefore, the time complexity is of the order of  $n$ .

14. What is the space complexity of the following dynamic programming implementation used to compute the nth fibonacci term?

```

1. int fibo(int n)
2.     int fibo_terms[100000] //arr to
    store the fibonacci numbers
3.     fibo_terms[0] = 0
4.     fibo_terms[1] = 1
5.
6.     for i: 2 to n
7.         fibo_terms[i] = fibo_terms[i - 1]
] + fibo_terms[i - 2]
8.
9.     return fibo_terms[n]

```

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) Exponential

**Answer:** b

**Explanation:** To calculate the nth term, we

store all the terms from 0 to  $n - 1$ . So, it takes  $O(n)$  space.

15. What will be the output when the following code is executed?

```
#include<stdio.
int fibo(int n)
{
    int i;
    int fibo_terms[100];
    fibo_terms[0]=0;
    fibo_terms[1]=1;
    for(i=2;i<=n;i++)
        fibo_terms[i] = fibo_terms[i-2]
+ fibo_terms[i-1];
    return fibo_terms[n];
}
int main()
{
    int r = fibo(8);
    printf("%d",r);
    return 0;
}
```

- a) 34
- b) 55
- c) Compile error
- d) 21

**Answer:** d

**Explanation:** The program prints the 8th fibonacci term, which is 21.

1. You are given infinite coins of denominations  $v_1, v_2, v_3, \dots, v_n$  and a sum  $S$ . The coin change problem is to find the minimum number of coins required to get the sum  $S$ . This problem can be solved using

- a) Greedy algorithm
- b) Dynamic programming
- c) Divide and conquer
- d) Backtracking

**Answer:** b

**Explanation:** The coin change problem has overlapping subproblems(same subproblems are solved multiple times) and optimal substructure(the solution to the problem can

be found by finding optimal solutions for subproblems). So, dynamic programming can be used to solve the coin change problem.

2. Suppose you have coins of denominations 1, 3 and 4. You use a greedy algorithm, in which you choose the largest denomination coin which is not greater than the remaining sum. For which of the following sums, will the algorithm NOT produce an optimal answer?

- a) 20
- b) 12
- c) 6
- d) 5

**Answer:** c

**Explanation:** Using the greedy algorithm, three coins {4,1,1} will be selected to make a sum of 6. But, the optimal answer is two coins {3,3}.

3. Suppose you have coins of denominations 1,3 and 4. You use a greedy algorithm, in which you choose the largest denomination coin which is not greater than the remaining sum. For which of the following sums, will the algorithm produce an optimal answer?

- a) 14
- b) 10
- c) 6
- d) 100

**Answer:** d

**Explanation:** Using the greedy algorithm, three coins {4,1,1} will be selected to make a sum of 6. But, the optimal answer is two coins {3,3}. Similarly, four coins {4,4,1,1} will be selected to make a sum of 10. But, the optimal answer is three coins {4,3,3}. Also, five coins {4,4,4,1,1} will be selected to make a sum of 14. But, the optimal answer is four coins {4,4,3,3}. For a sum of 100, twenty-five coins {all 4's} will be selected and the optimal answer is also twenty-five coins {all 4's}.

4. Fill in the blank to complete the code.

```
#include<stdio.h>
int main()
{
    int coins[10]={1,3,4}, lookup[100000];
    int i,j,tmp,num_coins = 3,sum=100;
    lookup[0]=0;
    for(i = 1; i <= sum; i++)
    {
        int min_coins = i;
        for(j = 0;j < num_coins; j++)
        {
            tmp = i - coins[j];
            if(tmp < 0)
                continue;
            if(lookup[tmp] < min_coins)
                min_coins = lookup[tmp];
        }
        lookup[i] = min_coins + 1;
    }
    printf("%d",lookup[sum]);
    return 0;
}
```

- a)  $\text{lookup}[\text{tmp}] = \text{min\_coins}$
- b)  $\text{min\_coins} = \text{lookup}[\text{tmp}]$
- c) break
- d) continue

**Answer:** b

**Explanation:**  $\text{min\_coins} = \text{lookup}[\text{tmp}]$  will complete the code.

5. You are given infinite coins of N denominations v<sub>1</sub>, v<sub>2</sub>, v<sub>3</sub>, ..., v<sub>n</sub> and a sum S. The coin change problem is to find the minimum number of coins required to get the sum S. What is the time complexity of a dynamic programming implementation used to solve the coin change problem?

- a) O(N)
- b) O(S)
- c) O(N<sup>2</sup>)
- d) O(S\*N)

**Answer:** d

**Explanation:** The time complexity is O(S\*N).

6. Suppose you are given infinite coins of N denominations v<sub>1</sub>, v<sub>2</sub>, v<sub>3</sub>, ..., v<sub>n</sub> and a sum S.

The coin change problem is to find the minimum number of coins required to get the sum S. What is the space complexity of a dynamic programming implementation used to solve the coin change problem?

- a) O(N)
- b) O(S)
- c) O(N<sup>2</sup>)
- d) O(S\*N)

**Answer:** b

**Explanation:** To get the optimal solution for a sum S, the optimal solution is found for each sum less than equal to S and each solution is stored. So, the space complexity is O(S).

7. You are given infinite coins of denominations 1, 3, 4. What is the total number of ways in which a sum of 7 can be achieved using these coins if the order of the coins is not important?

- a) 4
- b) 3
- c) 5
- d) 6

**Answer:** c

**Explanation:** A sum of 7 can be achieved in the following ways:

{1,1,1,1,1,1}, {1,1,1,1,3}, {1,3,3}, {1,1,1,4}, {3,4}.

Therefore, the sum can be achieved in 5 ways.

8. You are given infinite coins of denominations 1, 3, 4. What is the minimum number of coins required to achieve a sum of 7?

- a) 1
- b) 2
- c) 3
- d) 4

**Answer:** b

**Explanation:** A sum of 7 can be achieved by using a minimum of two coins {3,4}.

9. You are given infinite coins of denominations 5, 7, 9. Which of the following sum CANNOT be achieved using these coins?

- a) 50
- b) 21
- c) 13
- d) 23

**Answer:** c

**Explanation:** One way to achieve a sum of 50 is to use ten coins of 5. A sum of 21 can be achieved by using three coins of 7. One way to achieve a sum of 23 is to use two coins of 7 and one coin of 9. A sum of 13 cannot be achieved.

10. You are given infinite coins of denominations 3, 5, 7. Which of the following sum CANNOT be achieved using these coins?

- a) 15
- b) 16
- c) 17
- d) 4

**Answer:** d

**Explanation:** Sums can be achieved as follows:

$$\begin{aligned}15 &= \{5,5,5\} \\16 &= \{3,3,5,5\} \\17 &= \{3,7,7\}\end{aligned}$$

we can't achieve for sum=4 because our available denominations are 3,5,7 and sum of any two denominations is greater than 4.

11. What is the output of the following program?

```
#include<stdio.h>
int main()
{
    int coins[10]={1,3,4},lookup[100];
    int i,j,tmp,num_coins = 3,sum=10;
    lookup[0]=0;
    for(i=1;i<=sum;i++)
    {
        int min_coins = i;
        for(j=0;j<num_coins;j++)
        {
```

```
            tmp=i-coins[j];
            if(tmp<0)
                continue;
            if(lookup[tmp] < min_coins)
                min_coins=lookup[tmp];
        }
        lookup[i] = min_coins + 1;
    }
    printf("%d",lookup[sum]);
    return 0;
}
```

- a) 2
- b) 3
- c) 4
- d) 5

**Answer:** b

**Explanation:** The program prints the minimum number of coins required to get a sum of 10, which is 3.

12. What is the output of the following program?

```
#include<stdio.h>
int main()
{
    int coins[10]={1,3,4},lookup[100];
    int i,j,tmp,num_coins = 3,sum=14;
    lookup[0]=0;
    for(i=1;i<=sum;i++)
    {
        int min_coins = i;
        for(j=0;j<num_coins;j++)
        {
            tmp=i-coins[j];
            if(tmp<0)
                continue;
            if(lookup[tmp] < min_coins)
                min_coins=lookup[tmp];
        }
        lookup[i] = min_coins + 1;
    }
    printf("%d",lookup[sum]);
    return 0;
}
```

- a) 2
- b) 3
- c) 4
- d) 5

**Answer:** c

**Explanation:** The program prints the minimum number of coins required to get a sum of 14, which is 4.

1. Given a one-dimensional array of integers, you have to find a sub-array with maximum sum. This is the maximum sub-array sum problem. Which of these methods can be used to solve the problem?

- a) Dynamic programming
- b) Two for loops (naive method)
- c) Divide and conquer
- d) Dynamic programming, naïve method and Divide and conquer methods

**Answer:** d

**Explanation:** Dynamic programming, naïve method and Divide and conquer methods can be used to solve the maximum sub array sum problem.

2. Find the maximum sub-array sum for the given elements.

{2, -1, 3, -4, 1, -2, -1, 5, -4}

- a) 3
- b) 5
- c) 8
- d) 6

**Answer:** b

**Explanation:** The maximum sub-array sum is 5.

3. Find the maximum sub-array sum for the given elements.

{-2, -1, -3, -4, -1, -2, -1, -5, -4}

- a) -3
- b) 5
- c) 3
- d) -1

**Answer:** d

**Explanation:** All the elements are negative. So, the maximum sub-array sum will be equal to the largest element. The largest element is

-1 and therefore, the maximum sub-array sum is -1.

4. Consider the following naive method to find the maximum sub-array sum:

```
#include<stdio.h>
int main()
{
    int arr[1000]={2, -1, 3, -4, 1, -2,
-1, 5, -4}, len=9;
    int cur_max, tmp_max, strt_idx, sub_
arr_idx;
    cur_max = arr[0];
    for(strt_idx = 0; strt_idx < len; st
rt_idx++)
    {
        tmp_max=0;
        for(sub_arr_idx = strt_idx; sub_
arr_idx < len; sub_arr_idx++)
        {
            tmp_max +=arr[sub_arr_idx]
;
            if(tmp_max > cur_max)
                _____;
        }
    }
    printf("%d",cur_max);
    return 0;
}
```

Which line should be inserted to complete the above code?

- a) tmp\_max = cur\_max
- b) break
- c) continue
- d) cur\_max = tmp\_max

**Answer:** d

**Explanation:** If the tmp\_max element is greater than the cur\_max element, we make cur\_max equal to tmp\_max, i.e. cur\_max = tmp\_max.

5. What is the time complexity of the following naive method used to find the maximum sub-array sum in an array containing n elements?

```
#include<stdio.h>
int main()
{
    int arr[1000]={2, -1, 3, -4, 1, -2,
```

```

-1, 5, -4}, len=9;
    int cur_max, tmp_max, strt_idx, sub_
arr_idx;
    cur_max = arr[0];
    for(strt_idx = 0; strt_idx < len; st
rt_idx++)
    {
        tmp_max=0;
        for(sub_arr_idx = strt_idx; sub
_arr_idx < len; sub_arr_idx++)
        {
            tmp_max +=arr[sub_arr_idx]
;
            if(tmp_max > cur_max)
                _____;
        }
    printf("%d",cur_max);
    return 0;
}

```

- a) O( $n^2$ )
- b) O(n)
- c) O( $n^3$ )
- d) O(1)

**Answer:** a

**Explanation:** The naive method uses two for loops. The outer loop runs from 0 to n, i.e.  $i = 0 : n$ .

The inner loop runs from  $i$  to n, i.e.  $j = i : n$ . Therefore, the inner loop runs:

n times when the outer loop runs the first time.

(n-1) times when the outer loop runs the second time.

:

:

:

2 times when the outer loop runs (n-1)th time.  
1 time when the outer loop runs nth time.

Therefore, time complexity =  $n + (n-1) + (n-2) + \dots + 2 + 1 = n * (n + 1) / 2 = O(n^2)$ .

6. What is the space complexity of the following naive method used to find the maximum sub-array sum in an array containing n elements?

```
#include<stdio.h>
int main()
```

```

{
    int arr[1000]={2, -1, 3, -4, 1, -2,
-1, 5, -4}, len=9;
    int cur_max, tmp_max, strt_idx, sub_
arr_idx;
    cur_max = arr[0];
    for(strt_idx = 0; strt_idx < len; st
rt_idx++)
    {
        tmp_max=0;
        for(sub_arr_idx = strt_idx; sub
_arr_idx < len; sub_arr_idx++)
        {
            tmp_max +=arr[sub_arr_idx]
;
            if(tmp_max > cur_max)
                _____;
        }
    printf("%d",cur_max);
    return 0;
}

```

- a) O( $n^2$ )
- b) O(1)
- c) O( $n^3$ )
- d) O(n)

**Answer:** b

**Explanation:** The naive method uses only a constant space. So, the space complexity is O(1).

7. What is the output of the following naive method used to find the maximum sub-array sum?

```
#include<stdio.h>
int main()
{
    int arr[1000] = {-2, -5, 6, -2, 3, -1,
0,-5, 6}, len = 9;
    int cur_max, tmp_max, strt_idx, sub_
arr_idx;
    cur_max = arr[0];
    for(strt_idx = 0; strt_idx < len; st
rt_idx++)
    {
        tmp_max = 0;
        for(sub_arr_idx = strt_idx; su
b_arr_idx < len; sub_arr_idx++)
        {
            tmp_max += arr[sub_arr_id
x];
        }
    }
}
```

```

        if(tmp_max > cur_max)
            cur_max = tmp_max;
    }
    printf("%d",cur_max);
    return 0;
}

```

- a) 6
- b) 9
- c) 7
- d) 4

**Answer:** c

**Explanation:** The naive method prints the maximum sub-array sum, which is 7.

8. What is the time complexity of the divide and conquer algorithm used to find the maximum sub-array sum?
- a)  $O(n)$
  - b)  $O(\log n)$
  - c)  $O(n \log n)$
  - d)  $O(n^2)$

**Answer:** c

**Explanation:** The time complexity of the divide and conquer algorithm used to find the maximum sub-array sum is  $O(n \log n)$ .

9. What is the space complexity of the divide and conquer algorithm used to find the maximum sub-array sum?
- a)  $O(n)$
  - b)  $O(1)$
  - c)  $O(n!)$
  - d)  $O(n^2)$

**Answer:** b

**Explanation:** The divide and conquer algorithm uses a constant space. So, the space complexity is  $O(1)$ .

1. Which line should be inserted in the blank to complete the following dynamic programming implementation of the maximum sub-array sum problem?

```

#include<stdio.h>
int max_num(int a,int b)
{
    if(a> b)
        return a;
    return b;
}
int maximum_subarray_sum(int *arr, int len)
{
    int sum[len], idx;
    sum[0] = arr[0];
    for(idx = 1; idx < len; idx++)
        sum[idx] = _____
    ;
    int mx = sum[0];
    for(idx = 0; idx < len; idx++)
        if(sum[idx] > mx)
            mx =sum[idx];
    return mx;
}
int main()
{
    int arr[] = {-2, -5, 6, -2, 3, -1, 0, -5, 6}, len = 9;
    int ans = maximum_subarray_sum(arr, len);
    printf("%d",ans);
    return 0;
}

a) max_num(sum[idx - 1] + arr[idx], arr[idx])
b) sum[idx - 1] + arr[idx].
c) min_num(sum[idx - 1] + arr[idx], arr[idx])
d) arr[idx].

```

**Answer:** a

**Explanation:** The array “sum” is used to store the maximum sub-array sum. The appropriate way to do this is by using:  $sum[idx] = max\_num(sum[idx - 1] + arr[idx], arr[idx])$ .

2. What is the time complexity of the following dynamic programming algorithm used to find the maximum sub-array sum?

```

#include<stdio.h>
int max_num(int a,int b)
{
    if(a> b)
        return a;
    return b;
}

```

```

int maximum_subarray_sum(int *arr, int len)
{
    int sum[len], idx;
    sum[0] = arr[0];
    for(idx = 1; idx < len; idx++)
        sum[idx] = max_num(sum[idx - 1]
+ arr[idx], arr[idx]);
    int mx = sum[0];
    for(idx = 0; idx < len; idx++)
        if(sum[idx] > mx)
            mx = sum[idx];
    return mx;
}
int main()
{
    int arr[] = {-2, -5, 6, -2, 3, -1,
0, -5, 6}, len = 9;
    int ans = maximum_subarray_sum(arr,
len);
    printf("%d", ans);
    return 0;
}

```

- a) O(n)
- b) O(logn)
- c) O(nlogn)
- d) O( $n^2$ )

**Answer:** a

**Explanation:** The time complexity of the above dynamic programming algorithm used to solve maximum sub-array sum is O(n).

3. What is the space complexity of the following dynamic programming algorithm used to find the maximum sub-array sum?

```

#include<stdio.h>
int max_num(int a,int b)
{
    if(a > b)
        return a;
    return b;
}
int maximum_subarray_sum(int *arr, int len)
{
    int sum[len], idx;
    sum[0] = arr[0];
    for(idx = 1; idx < len; idx++)
        sum[idx] = max_num(sum[idx - 1]
+ arr[idx], arr[idx]);
    int mx = sum[0];

```

```

        for(idx = 0; idx < len; idx++)
            if(sum[idx] > mx)
                mx = sum[idx];
    return mx;
}
int main()
{
    int arr[] = {-2, -5, 6, -2, 3, -1,
0, -5, 6}, len = 9;
    int ans = maximum_subarray_sum(arr,
len);
    printf("%d", ans);
    return 0;
}

a) O(n)
b) O(1)
c) O(n!)
d) O( $n^2$ )

```

**Answer:** a

**Explanation:** The above dynamic programming algorithm uses space equal to the length of the array to store the sum values. So, the space complexity is O(n).

4. Consider the following code snippet:

```

1. int sum[len], idx;
2. sum[0] = arr[0];
3. for(idx = 1; idx < len; idx++)
4.     sum[idx] = max(sum[idx - 1] + a
rr[idx], arr[idx]);
5. int mx = sum[0];
6. for(idx = 0; idx < len; idx++)
7.     if(sum[idx] > mx)
8.         mx = sum[idx];
9. return mx;

```

Which property is shown by line 4 of the above code snippet?

- a) Optimal substructure
- b) Overlapping subproblems
- c) Both overlapping subproblems and optimal substructure
- d) Greedy substructure

**Answer:** a

**Explanation:** The current sum (i.e. sum[idx]) uses the previous sum (i.e. sum[idx - 1]) to get an optimal value. So, line 4 shows the optimal substructure property.

5. Consider the following code snippet:

```

1. int sum[len], idx;
2. sum[0] = arr[0];
3. for(idx = 1; idx < len; idx++)
4.     sum[idx] = max(sum[idx - 1] + a
rr[idx], arr[idx]);
5. int mx = sum[0];
6. for(idx = 0; idx < len; idx++)
7.     if(sum[idx] > mx)
8.         mx =sum[idx];
9. return mx;

```

Which method is used by line 4 of the above code snippet?

- a) Divide and conquer
- b) Recursion
- c) Both memoization and divide and conquer
- d) Memoization

**Answer:** d

**Explanation:** The array “sum” is used to store the previously calculated values, so that they aren’t recalculated. So, line 4 uses the memoization technique.

6. Find the maximum sub-array sum for the following array:

{3, 6, 7, 9, 3, 8}

- a) 33
- b) 36
- c) 23
- d) 26

**Answer:** b

**Explanation:** All the elements of the array are positive. So, the maximum sub-array sum is equal to the sum of all the elements, which is 36.

7. What is the output of the following program?

```
#include<stdio.h>
int max_num(int a,int b)
{
    if(a> b)
        return a;
    return b;
}
int maximum_subarray_sum(int *arr, int le
```

```

n)
{
    int sum[len], idx;
    sum[0] = arr[0];
    for(idx = 1; idx < len; idx++)
        sum[idx] = max_num(sum[idx - 1] +
arr[idx], arr[idx]);
    int mx = sum[0];
    for(idx = 0; idx < len; idx++)
        if(sum[idx] > mx)
            mx =sum[idx];
    return mx;
}
int main()
{
    int arr[] = {-20, 23, 10, 3, -10, 11
, -5},len = 7;
    int ans = maximum_subarray_sum(arr,
len);
    printf("%d",ans);
    return 0;
}

a) 27
b) 37
c) 36
d) 26

```

**Answer:** b

**Explanation:** The program prints the value of maximum sub-array sum, which is 37.

8. What is the value stored in sum[4] after the following program is executed?

```
#include<stdio.h>
int max_num(int a,int b)
{
    if(a> b)
        return a;
    return b;
}
int maximum_subarray_sum(int *arr, int le
n)
{
    int sum[len], idx;
    sum[0] = arr[0];
    for(idx = 1; idx < len; idx++)
        sum[idx] = max_num(sum[idx - 1]
+ arr[idx], arr[idx]);
    int mx = sum[0];
    for(idx = 0; idx < len; idx++)
        if(sum[idx] > mx)
            mx =sum[idx];
    return mx;
}
```

```

}
int main()
{
    int arr[] = {-2, 14, 11, -13, 10, -5, 11, -6, 3, -5}, len = 10;
    int ans = maximum_subarray_sum(arr, len);
    printf("%d", ans);
    return 0;
}

```

- a) 28
- b) 25
- c) 22
- d) 12

**Answer:** c

**Explanation:** After the program is executed the value stored in sum[4] is 22.

Note: You are asked to find the value stored in sum[4] and NOT the output of the program.

---

1. Kadane's algorithm is used to find

- a) Longest increasing subsequence
- b) Longest palindrome subsequence
- c) Maximum sub-array sum
- d) Longest decreasing subsequence

**Answer:** c

**Explanation:** Kadane's algorithm is used to find the maximum sub-array sum for a given array.

2. Kadane's algorithm uses which of the following techniques?

- a) Divide and conquer
- b) Dynamic programming
- c) Recursion
- d) Greedy algorithm

**Answer:** b

**Explanation:** Kadane's algorithm uses dynamic programming.

3. For which of the following inputs would Kadane's algorithm produce the INCORRECT output?

- a) {0,1,2,3}
- b) {-1,0,1}
- c) {-1,-2,-3,0}
- d) {-4,-3,-2,-1}

**Answer:** d

**Explanation:** Kadane's algorithm works if the input array contains at least one non-negative element. Every element in the array {-4,-3,-2,-1} is negative. Hence Kadane's algorithm won't work.

4. For which of the following inputs would Kadane's algorithm produce a WRONG output?

- a) {1,0,-1}
- b) {-1,-2,-3}
- c) {1,2,3}
- d) {0,0,0}

**Answer:** b

**Explanation:** Kadane's algorithm doesn't work for all negative numbers. So, the answer is {-1,-2,-3}.

5. Complete the following code for Kadane's algorithm:

```

#include<stdio.h>
int max_num(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int kadane_algo(int *arr, int len)
{
    int ans, sum, idx;
    ans = 0;
    sum = 0;
    for(idx = 0; idx < len; idx++)
    {
        sum = max_num(0, sum + arr[idx]);
        ans = _____;
    }
    return ans;
}
int main()
{
    int arr[] = {-2, -3, 4, -1, -2, 1, 5, -3}, len=7;
    int ans = kadane_algo(arr, len);

```

```

    printf("%d",ans);
    return 0;
}

a) max_num(sum, sum + arr[idx])
b) sum
c) sum + arr[idx]
d) max_num(sum,ans)

```

**Answer:** d

**Explanation:** The maximum of sum and ans, is stored in ans. So, the answer is max\_num(sum, ans).

6. What is the time complexity of Kadane's algorithm?

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) O(5)

**Answer:** b

**Explanation:** The time complexity of Kadane's algorithm is O(n) because there is only one for loop which scans the entire array exactly once.

7. What is the space complexity of Kadane's algorithm?

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) None of the mentioned

**Answer:** a

**Explanation:** Kadane's algorithm uses a constant space. So, the space complexity is O(1).

8. What is the output of the following implementation of Kadane's algorithm?

```

#include<stdio.h>
int max_num(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int kadane_algo(int *arr, int len)

```

```

    {
        int ans, sum, idx;
        ans = 0;
        sum = 0;
        for(idx = 0; idx < len; idx++)
        {
            sum = max_num(0,sum + arr[idx]);
            ans = max_num(sum,ans);
        }
        return ans;
    }
    int main()
    {
        int arr[] = {2, 3, -3, -1, 2, 1, 5,
-3}, len = 8;
        int ans = kadane_algo(arr,len);
        printf("%d",ans);
        return 0;
    }

a) 6
b) 7
c) 8
d) 9

```

**Answer:** d

**Explanation:** Kadane's algorithm produces the maximum sub-array sum, which is equal to 9.

9. What is the output of the following implementation of Kadane's algorithm?

```

#include<stdio.h>
int max_num(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int kadane_algo(int *arr, int len)
{
    int ans, sum, idx;
    ans = 0;
    sum = 0;
    for(idx = 0; idx < len; idx++)
    {
        sum = max_num(0,sum + arr[idx]);
    }
    ans = max_num(sum,ans);
    return ans;
}
int main()

```

```

{
    int arr[] = {-2, -3, -3, -1, -2, -1
, -5, -3},len = 8;
    int ans = kadane_algo(arr,len);
    printf("%d",ans);
    return 0;
}

```

- a) 1
- b) -1
- c) -2
- d) 0

**Answer:** d

**Explanation:** Kadane's algorithm produces a wrong output when all the elements are negative. The output produced is 0.

10. Consider the following implementation of Kadane's algorithm:

```

1. #include<stdio.h>
2. int max_num(int a, int b)
3. {
4.     if(a > b)
5.         return a;
6.     return b;
7. }
8. int kadane_algo(int *arr, int len)
9. {
10.     int ans = 0, sum = 0, idx;
11.     for(idx = 0; idx < len; idx++)
12.     {
13.         sum = max_num(0,sum + arr
[idx]);
14.         ans = max_num(sum,ans);
15.     }
16.     return ans;
17. }
18. int main()
19. {
20.     int arr[] = {-2, -3, -3, -1, -2
, -1, -5, -3},len = 8;
21.     int ans = kadane_algo(arr,len);
22.     printf("%d",ans);
23.     return 0;
24. }

```

What changes should be made to the Kadane's algorithm so that it produces the right output even when all array elements are negative?

Change 1 = Line 10: int sum = arr[0], ans = arr[0]  
Change 2 = Line 13: sum = max\_num(arr[idx],sum+arr[idx])

- a) Only Change 1 is sufficient
- b) Only Change 2 is sufficient
- c) Both Change 1 and Change 2 are necessary
- d) No change is required

**Answer:** c

**Explanation:** Both change 1 and change 2 should be made to Kadane's algorithm so that it produces the right output even when all the array elements are negative.

1. The longest increasing subsequence problem is a problem to find the length of a subsequence from a sequence of array elements such that the subsequence is sorted in increasing order and its length is maximum. This problem can be solved using

- a) Recursion
- b) Dynamic programming
- c) Brute force
- d) Recursion, Dynamic programming, Brute force

**Answer:** d

**Explanation:** The longest increasing subsequence problem can be solved using all of the mentioned methods.

2. Find the longest increasing subsequence for the given sequence:

- {10, -10, 12, 9, 10, 15, 13, 14}
- a) {10, 12, 15}
  - b) {10, 12, 13, 14}
  - c) {-10, 12, 13, 14}
  - d) {-10, 9, 10, 13, 14}

**Answer:** d

**Explanation:** The longest increasing subsequence is {-10, 9, 10, 13, 14}.

3. Find the length of the longest increasing subsequence for the given sequence:

- {-10, 24, -9, 35, -21, 55, -41, 76, 84};  
 a) 5  
 b) 4  
 c) 3  
 d) 6

**Answer:** d

**Explanation:** The longest increasing subsequence is {-10, 24, 35, 55, 76, 84} and it's length is 6.

4. For any given sequence, there will ALWAYS be a unique increasing subsequence with the longest length.  
 a) True  
 b) False

**Answer:** b

**Explanation:** For a given sequence, it is possible that there is more than one subsequence with the longest length. Consider, the following sequence:

{10,11,12,1,2,3}:

There are two longest increasing subsequences: {1,2,3} and {10,11,12}.

5. The number of increasing subsequences with the longest length for the given sequence are:

- {10, 9, 8, 7, 6, 5}  
 a) 3  
 b) 4  
 c) 5  
 d) 6

**Answer:** d

**Explanation:** Each array element individually forms a longest increasing subsequence and so, the length of the longest increasing subsequence is 1. So, the number of increasing subsequences with the longest length is 6.

6. In the brute force implementation to find the longest increasing subsequence, all the subsequences of a given sequence are found. All the increasing subsequences are then selected and the length of the longest

subsequence is found. What is the time complexity of this brute force implementation?

- a) O(n)  
 b) O( $n^2$ )  
 c) O(n!)  
 d) O( $2^n$ )

**Answer:** d

**Explanation:** The time required to find all the subsequences of a given sequence is  $2^n$ , where 'n' is the number of elements in the sequence. So, the time complexity is O( $2^n$ ).

7. Complete the following dynamic programming implementation of the longest increasing subsequence problem:

```
#include<stdio.h>
int longest_inc_sub(int *arr, int len)
{
    int i, j, tmp_max;
    int LIS[len]; // array to store the lengths of the longest increasing subsequence
    LIS[0]=1;
    for(i = 1; i < len; i++)
    {
        tmp_max = 0;
        for(j = 0; j < i; j++)
        {
            if(arr[j] < arr[i])
            {
                if(LIS[j] > tmp_max)
                    _____;
            }
        }
        LIS[i] = tmp_max + 1;
    }
    int max = LIS[0];
    for(i = 0; i < len; i++)
        if(LIS[i] > max)
            max = LIS[i];
    return max;
}
int main()
{
    int arr[] = {10,22,9,33,21,50,41,60,80}, len = 9;
    int ans = longest_inc_sub(arr, len);
    printf("%d",ans);
```

```
        return 0;
}
```

- a) tmp\_max = LIS[j]
- b) LIS[i] = LIS[j]
- c) LIS[j] = tmp\_max
- d) tmp\_max = LIS[i]

**Answer:** a

**Explanation:** tmp\_max is used to store the maximum length of an increasing subsequence for any 'j' such that: arr[j] < arr[i] and  $0 < j < i$ . So, tmp\_max = LIS[j] completes the code.

8. What is the time complexity of the following dynamic programming implementation used to find the length of the longest increasing subsequence?

```
#include<stdio.h>
int longest_inc_sub(int *arr, int len)
{
    int i, j, tmp_max;
    int LIS[len]; // array to store the lengths of the longest increasing subsequence
    LIS[0]=1;
    for(i = 1; i < len; i++)
    {
        tmp_max = 0;
        for(j = 0; j < i; j++)
        {
            if(arr[j] < arr[i])
            {
                if(LIS[j] > tmp_max)
                    tmp_max = LIS[j];
            }
        }
        LIS[i] = tmp_max + 1;
    }
    int max = LIS[0];
    for(i = 0; i < len; i++)
        if(LIS[i] > max)
            max = LIS[i];
    return max;
}
int main()
{
    int arr[] = {10,22,9,33,21,50,41,60,80}, len = 9;
    int ans = longest_inc_sub(arr, len);
    printf("%d",ans);
}
```

```
        return 0;
}
```

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) O(nlogn)

**Answer:** c

**Explanation:** The time complexity of the above dynamic programming implementation used to find the length of the longest increasing subsequence is  $O(n^2)$ .

9. What is the space complexity of the following dynamic programming implementation used to find the length of the longest increasing subsequence?

```
#include<stdio.h>
int longest_inc_sub(int *arr, int len)
{
    int i, j, tmp_max;
    int LIS[len]; // array to store the lengths of the longest increasing subsequence
    LIS[0]=1;
    for(i = 1; i < len; i++)
    {
        tmp_max = 0;
        for(j = 0; j < i; j++)
        {
            if(arr[j] < arr[i])
            {
                if(LIS[j] > tmp_max)
                    tmp_max = LIS[j];
            }
        }
        LIS[i] = tmp_max + 1;
    }
    int max = LIS[0];
    for(i = 0; i < len; i++)
        if(LIS[i] > max)
            max = LIS[i];
    return max;
}
int main()
{
    int arr[] = {10,22,9,33,21,50,41,60,80}, len = 9;
    int ans = longest_inc_sub(arr, len);
    printf("%d",ans);
}
```

```

        return 0;
}

```

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) O(nlogn)

**Answer:** b

**Explanation:** The above dynamic programming implementation uses space equal to the length of the sequence. So, the space complexity of the above dynamic programming implementation used to find the length of the longest increasing subsequence is O(n).

10. What is the output of the following program?

```

#include<stdio.h>
int longest_inc_sub(int *arr, int len)
{
    int i, j, tmp_max;
    int LIS[len]; // array to store th
e lengths of the longest increasing subse
quence
    LIS[0]=1;
    for(i = 1; i < len; i++)
    {
        tmp_max = 0;
        for(j = 0; j < i; j++)
        {
            if(arr[j] < arr[i])
            {
                if(LIS[j] > tmp_max)
                    tmp_max = LIS[j];
            }
            LIS[i] = tmp_max + 1;
        }
        int max = LIS[0];
        for(i = 0; i < len; i++)
            if(LIS[i] > max)
                max = LIS[i];
        return max;
    }
    int main()
    {
        int arr[] = {10,22,9,33,21,50,41,60
,80}, len = 9;
        int ans = longest_inc_sub(arr, len);
        printf("%d",ans);
    }
}

```

```

        return 0;
}

```

- a) 3
- b) 4
- c) 5
- d) 6

**Answer:** d

**Explanation:** The program prints the length of the longest increasing subsequence, which is 6.

11. What is the value stored in LIS[5] after the following program is executed?

```

#include<stdio.h>
int longest_inc_sub(int *arr, int len)
{
    int i, j, tmp_max;
    int LIS[len]; // array to store th
e lengths of the longest increasing subse
quence
    LIS[0]=1;
    for(i = 1; i < len; i++)
    {
        tmp_max = 0;
        for(j = 0; j < i; j++)
        {
            if(arr[j] < arr[i])
            {
                if(LIS[j] > tmp_max)
                    tmp_max = LIS[j];
            }
            LIS[i] = tmp_max + 1;
        }
        int max = LIS[0];
        for(i = 0; i < len; i++)
            if(LIS[i] > max)
                max = LIS[i];
        return max;
    }
    int main()
    {
        int arr[] = {10,22,9,33,21,50,41,60
,80}, len = 9;
        int ans = longest_inc_sub(arr, len);
        printf("%d",ans);
        return 0;
    }
}

```

- a) 2
- b) 3
- c) 4
- d) 5

**Answer:** c

**Explanation:** The value stored in LIS[5] after the program is executed is 4.

1. Given a rod of length n and the selling prices of all pieces smaller than equal to n, find the most beneficial way of cutting the rod into smaller pieces. This problem is called the rod cutting problem. Which of these methods can be used to solve the rod cutting problem?

- a) Brute force
- b) Dynamic programming
- c) Recursion
- d) Brute force, Dynamic programming and Recursion

**Answer:** d

**Explanation:** Brute force, Dynamic programming and Recursion can be used to solve the rod cutting problem.

2. You are given a rod of length 5 and the prices of each length are as follows:

length

price

1

2

2

5

3

6

4

9

5

9

**Answer:** c

**Explanation:** The pieces {1,2,2} give the maximum value of 12.

3. Consider the brute force implementation of the rod cutting problem in which all the possible cuts are found and the maximum value is calculated. What is the time complexity of this brute force implementation?

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

**Answer:** d

**Explanation:** The brute force implementation finds all the possible cuts. This takes  $O(2^n)$  time.

4. You are given a rod of length 10 and the following prices.

| length | price |
|--------|-------|
| 1      | 2     |
| 2      | 5     |
| 3      | 6     |
| 4      | 9     |
| 5      | 9     |
| 6      | 17    |
| 7      | 17    |
| 8      | 18    |
| 9      | 20    |
| 10     | 22    |

What is the maximum value that you can get after cutting the rod and selling the pieces?

- a) 10
- b) 11
- c) 12
- d) 13

Which of these pieces give the maximum price?

- a) {1,2,7}
- b) {10}
- c) {2,2,6}
- d) {1,4,5}

**Answer:** c

**Explanation:** The pieces {2,2,6} give the maximum value of 27.

5. Consider the following recursive implementation of the rod cutting problem:

```
#include<stdio.h>
#include<limits.h>
int max_of_two(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int rod_cut(int *prices, int len)
{
    int max_price = INT_MIN; // INT_MIN is the min value an integer can take
    int i;
    if(len <= 0 )
        return 0;
    for(i = 0; i < len; i++)
        max_price = max_of_two(max_price,
                               prices[i] + rod_cut(prices,len - i - 1));
    // subtract 1 because index starts from 0
    return max_price;
}
int main()
{
    int prices[]={2, 5, 6, 9, 9, 17, 17, 18, 20, 22},len_of_rod = 10;
    int ans = rod_cut(prices, len_of_rod);
    printf("%d",ans);
    return 0;
}
```

Complete the above code.

- a) max\_price, prices[i] + rod\_cut(prices,len - i - 1)
- b) max\_price, prices[i - 1].
- c) max\_price, rod\_cut(prices, len - i - 1)
- d) max\_price, prices[i - 1] + rod\_cut(prices,len - i - 1)

**Answer:** a

**Explanation:** max\_price, prices[i] + rod\_cut(prices,len - i - 1) completes the above code.

6. What is the time complexity of the following recursive implementation?

```
#include<stdio.h>
#include<limits.h>
int max_of_two(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int rod_cut(int *prices, int len)
{
    int max_price = INT_MIN; // INT_MIN is the min value an integer can take
    int i;
    if(len <= 0 )
        return 0;
    for(i = 0; i < len; i++)
        max_price = max_of_two(max_price,
                               prices[i] + rod_cut(prices,len - i - 1));
    // subtract 1 because index starts from 0
    return max_price;
}
int main()
{
    int prices[]={2, 5, 6, 9, 9, 17, 17, 18, 20, 22},len_of_rod = 10;
    int ans = rod_cut(prices, len_of_rod);
    printf("%d",ans);
    return 0;
}
```

a) O(n)

b) O( $n^2$ )

c) O( $n^3$ )

d) O( $2^n$ )

**Answer:** d

**Explanation:** The time complexity of the above recursive implementation is O( $2^n$ ).

7. What is the space complexity of the following recursive implementation?

```
#include<stdio.h>
#include<limits.h>
int max_of_two(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int rod_cut(int *prices, int len)
{
```

```

        int max_price = INT_MIN; // INT_MIN
is the min value an integer can take
        int i;
        if(len <= 0 )
            return 0;
        for(i = 0; i < len; i++)
            max_price = max_of_two(max_price
, prices[i] + rod_cut(prices,len - i - 1)
); // subtract 1 because index starts fro
m 0
        return max_price;
    }
int main()
{
    int prices[]={2, 5, 6, 9, 9, 17, 1
, 18, 20, 22},len_of_rod = 10;
    int ans = rod_cut(prices, len_of_rod);
    printf("%d",ans);
    return 0;
}

```

- a) O(1)
- b) O(logn)
- c) O(nlogn)
- d) O(n!)

**Answer:** a

**Explanation:** The space complexity of the above recursive implementation is O(1) because it uses a constant space.

8. What will be the value stored in max\_value when the following code is executed?

```

#include<stdio.h>
#include<limits.h>
int max_of_two(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int rod_cut(int *prices, int len)
{
    int max_price = INT_MIN; // INT_MI
N is the min value an integer can take
    int i;
    if(len <= 0 )
        return 0;
    for(i = 0; i < len; i++)
        max_price = max_of_two(prices[i]
] + rod_cut(prices,len - i - 1), max_price
); // subtract 1 because index starts fr
om 0

```

```

        return max_price;
    }
int main()
{
    int prices[]={2, 5, 6, 9, 9, 17, 1
, 18, 20, 22},len_of_rod = 3;
    int ans = rod_cut(prices, len_of_rod);
    printf("%d",ans);
    return 0;
}

```

- a) 5
- b) 6
- c) 7
- d) 8

**Answer:** c

**Explanation:** The value stored in max\_value after the code is executed is equal to 7.

9. For every rod cutting problem there will be a unique set of pieces that give the maximum price.

- a) True
- b) False

**Answer:** b

**Explanation:** Consider a rod of length 3. The prices are {2,3,6} for lengths {1,2,3} respectively. The pieces {1,1,1} and {3} both give the maximum value of 6.

10. Consider the following dynamic programming implementation of the rod cutting problem:

```

#include<stdio.h>
#include<limits.h>
int rod_cut(int *prices, int len)
{
    int max_val[len + 1];
    int i,j,tmp_price,tmp_idx;
    max_val[0] = 0;
    for(i = 1; i <= len; i++)
    {
        int tmp_max = INT_MIN; // mini
mum value an integer can hold
        for(j = 1; j <= i; j++)
        {
            tmp_idx = i - j;
            tmp_price =
        }; //subtract 1 because index of prices s

```

```

        tarts from 0
        if(tmp_price > tmp_max)
            tmp_max = tmp_price;
    }
    max_val[i] = tmp_max;
}
return max_val[len];
}
int main()
{
    int prices[]={2, 5, 6, 9, 9, 17, 1
7, 18, 20, 22},len_of_rod = 5;
    int ans = rod_cut(prices, len_of_r
od);
    printf("%d",ans);
    return 0;
}

```

Which line will complete the ABOVE code?

- a) prices[j-1] + max\_val[tmp\_idx]
- b) prices[j] + max\_val[tmp\_idx]
- c) prices[j-1] + max\_val[tmp\_idx - 1]
- d) prices[j] + max\_val[tmp\_idx - 1]

**Answer:** a

**Explanation:** prices[j-1] + max\_val[tmp\_idx] completes the code.

11. What is the time complexity of the following dynamic programming implementation of the rod cutting problem?

```

#include<stdio.h>
#include<limits.h>
int rod_cut(int *prices, int len)
{
    int max_val[len + 1];
    int i,j,tmp_price,tmp_idx;
    max_val[0] = 0;
    for(i = 1; i <= len; i++)
    {
        int tmp_max = INT_MIN; // mini
        mum value an integer can hold
        for(j = 1; j <= i; j++)
        {
            tmp_idx = i - j;
            tmp_price = prices[j-1]
+ max_val[tmp_idx]; //subtract 1 because
index of prices starts from 0
            if(tmp_price > tmp_max)
                tmp_max = tmp_price;
        }
        max_val[i] = tmp_max;
    }
    return max_val[len];
}

```

```

    }
int main()
{
    int prices[]={2, 5, 6, 9, 9, 17, 1
7, 18, 20, 22},len_of_rod = 5;
    int ans = rod_cut(prices, len_of_r
od);
    printf("%d",ans);
    return 0;
}

a) O(1)
b) O(n)
c) O(n2)
d) O(2n)

```

**Answer:** c

**Explanation:** The time complexity of the above dynamic programming implementation of the rod cutting problem is O(n<sup>2</sup>).

12. What is the space complexity of the following dynamic programming implementation of the rod cutting problem?

```

#include<stdio.h>
#include<limits.h>
int rod_cut(int *prices, int len)
{
    int max_val[len + 1];
    int i,j,tmp_price,tmp_idx;
    max_val[0] = 0;
    for(i = 1; i <= len; i++)
    {
        int tmp_max = INT_MIN; // mini
        mum value an integer can hold
        for(j = 1; j <= i; j++)
        {
            tmp_idx = i - j;
            tmp_price = prices[j-1]
+ max_val[tmp_idx]; //subtract 1 because
index of prices starts from 0
            if(tmp_price > tmp_max)
                tmp_max = tmp_price;
        }
        max_val[i] = tmp_max;
    }
    return max_val[len];
}
int main()
{
    int prices[]={2, 5, 6, 9, 9, 17, 1
7, 18, 20, 22},len_of_rod = 5;
    int ans = rod_cut(prices, len_of_r
od);
    printf("%d",ans);
    return 0;
}

```

```

od);
    printf("%d",ans);
    return 0;
}

```

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) O( $2^n$ )

**Answer:** b

**Explanation:** The space complexity of the above dynamic programming implementation of the rod cutting problem is O(n) because it uses a space equal to the length of the rod.

13. What is the output of the following program?

```

#include<stdio.h>
#include<limits.h>
int rod_cut(int *prices, int len)
{
    int max_val[len + 1];
    int i,j,tmp_price,tmp_idx;
    max_val[0] = 0;
    for(i = 1; i <= len; i++)
    {
        int tmp_max = INT_MIN; // minimum value an integer can hold
        for(j = 1; j <= i; j++)
        {
            tmp_idx = i - j;
            tmp_price = prices[j-1] +
max_val[tmp_idx];//subtract 1 because index of prices starts from 0
            if(tmp_price > tmp_max)
                tmp_max = tmp_price;
        }
        max_val[i] = tmp_max;
    }
    return max_val[len];
}
int main()
{
    int prices[]={2, 5, 6, 9, 9, 17, 17
, 18, 20, 22},len_of_rod = 5;
    int ans = rod_cut(prices, len_of_rod);
    printf("%d",ans);
    return 0;
}

```

- a) 9
- b) 10
- c) 11
- d) 12

**Answer:** d

**Explanation:** The program prints the maximum price that can be achieved by cutting the rod into pieces, which is equal to 27.

14. What is the value stored in max\_val[5] after the following program is executed?

```

#include<stdio.h>
#include<limits.h>
int rod_cut(int *prices, int len)
{
    int max_val[len + 1];
    int i,j,tmp_price,tmp_idx;
    max_val[0] = 0;
    for(i = 1; i <= len; i++)
    {
        int tmp_max = INT_MIN; // minimum value an integer can hold
        for(j = 1; j <= i; j++)
        {
            tmp_idx = i - j;
            tmp_price = prices[j-1] +
max_val[tmp_idx];//subtract 1 because index of prices starts from 0
            if(tmp_price > tmp_max)
                tmp_max = tmp_price;
        }
        max_val[i] = tmp_max;
    }
    return max_val[len];
}
int main()
{
    int prices[]={2, 5, 6, 9, 9, 17, 17
, 18, 20, 22},len_of_rod = 5;
    int ans = rod_cut(prices, len_of_rod);
    printf("%d",ans);
    return 0;
}
a) 12
b) 27
c) 10
d) 17

```

**Answer:** a

**Explanation:** The value stored in max\_val[5] after the program is executed is 12.

---

1. You are given an array of elements where each array element represents the MAXIMUM number of jumps that can be made in the forward direction from that element. You have to find the minimum number of jumps that are required to reach the end of the array. Which of these methods can be used to solve the problem?
  - a) Dynamic Programming
  - b) Greedy Algorithm
  - c) Recursion
  - d) Recursion and Dynamic Programming

**Answer:** d

**Explanation:** Both recursion and dynamic programming can be used to solve minimum number of jumps problem.

2. Consider the following array:

{1, 3, 5, 8, 9, 2, 6, 7, 6}

What is the minimum number of jumps required to reach the end of the array?

- a) 1
- b) 2
- c) 3
- d) 4

**Answer:** c

**Explanation:** The jumps made will be: {1 -> 2 -> 4 -> 9}. So, the number of jumps is three.

3. Consider the following recursive implementation:

```
#include<stdio.h>
#include<limits.h>
int min_jumps(int *arr, int strt, int end)
{
    int idx;
    if(strt == end)
        return 0;
    if(arr[strt] == 0) // jump cannot be
made
```

```
        return INT_MAX;
    int min = INT_MAX;
    for(idx = 1; idx <= arr[strt] && str
t + idx <= end; idx++)
    {
        int jumps = min_jumps(____,____
,____) + 1;
        if(jumps < min)
            min = jumps;
    }
    return min;
}
int main()
{
    int arr[] ={1, 3, 5, 8, 9, 2, 6, 7,
6},len = 9;
    int ans = min_jumps(arr, 0, len-1);
    printf("%d\n",ans);
    return 0;
}
```

Which of these arguments should be passed by the min\_jumps function represented by the blanks?

- a) arr, strt + idx, end
- b) arr + idx, strt, end
- c) arr, strt, end
- d) arr, strt, end + idx

**Answer:** a

**Explanation:** arr, strt + idx, end should be passed as arguments.

4. For a given array, there can be multiple ways to reach the end of the array using minimum number of jumps.

- a) True
- b) False

**Answer:** a

**Explanation:** Consider the array {1,2,3,4,5}. It is possible to reach the end in the following ways: {1 -> 2 -> 3 -> 5} or {1 -> 2 -> 4 -> 5}. In both the cases the number of jumps is 3, which is minimum. Hence, it is possible to reach the end of the array in multiple ways using minimum number of jumps.

5. What is the output of the following program?

```
#include<stdio.h>
#include<limits.h>
int min_jumps(int *arr, int strt, int end)
{
    int idx;
    if(strt == end)
        return 0;
    if(arr[strt] == 0) // jump cannot be made
        return INT_MAX;
    int min = INT_MAX;
    for(idx = 1; idx <= arr[strt] && start + idx <= end; idx++)
    {
        int jumps = min_jumps(arr, start + idx, end) + 1;
        if(jumps < min)
            min = jumps;
    }
    return min;
}
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 4, 3, 2, 1}, len = 9;
    int ans = min_jumps(arr, 0, len-1);
    printf("%d\n", ans);
    return 0;
}
```

a) 4  
b) 5  
c) 6  
d) 7

**Answer:** a

**Explanation:** The program prints the minimum number of jumps required to reach the end of the array. One way to reach the end using minimum number of jumps is {1 -> 2 -> 4 -> 8 -> 9}. So, the number of jumps is 4.

6. For any array, given that at most one element is non-zero, it is ALWAYS possible to reach the end of the array using minimum jumps.

- a) True
- b) False

**Answer:** b

**Explanation:** Consider the array {1,0,2,3,4}.

In this case, only one element is 0 but it is not possible to reach the end of the array.

7. Consider the following dynamic programming implementation of the minimum jumps problem:

```
#include<stdio.h>
#include<limits.h>
int min_jump(int *arr, int len)
{
    int j, idx, jumps[len];
    jumps[len - 1] = 0;
    for(idx = len - 2; idx >= 0; idx--)
    {
        int tmp_min = INT_MAX;
        for(j = 1; j <= arr[idx] && idx + j < len; j++)
        {
            if(jumps[idx + j] + 1 < tmp_min)
                tmp_min = jumps[idx + j] + 1;
        }
        jumps[idx] = tmp_min;
    }
    return jumps[0];
}
int main()
{
    int arr[] = {1, 1, 1, 1, 1, 1, 1, 1, 1}, len = 9;
    int ans = min_jump(arr, len);
    printf("%d\n", ans);
    return 0;
}
```

Which of the following “for” loops can be used instead of the inner for loop so that the output doesn’t change?

- a) for(j = 1; j < arr[idx] + len; j++)
- b) for(j = 0; j < arr[idx] - len; j++)
- c) for(j = idx + 1; j < len && j <= arr[idx] + idx; j++)
- d) No change is required

**Answer:** d

**Explanation:** None of the above mentioned “for” loops can be used instead of the inner for loop. Note, for(j = idx + 1; j < len && j <= arr[idx] + idx; j++) covers the same range as the inner for loop but it produces the wrong output because the indexing inside the

loops changes as “j” takes different values in the two “for” loops.

8. What is the time complexity of the following dynamic programming implementation used to find the minimum number of jumps?

```
#include<stdio.h>
#include<limits.h>
int min_jump(int *arr, int len)
{
    int j, idx, jumps[len];
    jumps[len - 1] = 0;
    for(idx = len - 2; idx >= 0; idx--)
    {
        int tmp_min = INT_MAX;
        for(j = 1; j <= arr[idx] && idx + j < len; j++)
        {
            if(jumps[idx + j] + 1 < tmp_min)
                tmp_min = jumps[idx + j] + 1;
        }
        jumps[idx] = tmp_min;
    }
    return jumps[0];
}
int main()
{
    int arr[] = {1, 1, 1, 1, 1, 1, 1, 1, 1}, len = 9;
    int ans = min_jump(arr, len);
    printf("%d\n", ans);
    return 0;
}
```

a) O(1)  
b) O(n)  
c) O( $n^2$ )  
d) None of the mentioned

**Answer:** c

**Explanation:** The time complexity of the above dynamic programming implementation is  $O(n^2)$ .

9. What is the space complexity of the following dynamic programming implementation used to find the minimum number of jumps?

```
#include<stdio.h>
#include<limits.h>
int min_jump(int *arr, int len)
{
    int j, idx, jumps[len];
    jumps[len - 1] = 0;
    for(idx = len - 2; idx >= 0; idx--)
    {
        int tmp_min = INT_MAX;
        for(j = 1; j <= arr[idx] && idx + j < len; j++)
        {
            if(jumps[idx + j] + 1 < tmp_min)
                tmp_min = jumps[idx + j] + 1;
        }
        jumps[idx] = tmp_min;
    }
    return jumps[0];
}
int main()
{
    int arr[] = {1, 1, 1, 1, 1, 1, 1, 1, 1}, len = 9;
    int ans = min_jump(arr, len);
    printf("%d\n", ans);
    return 0;
}
```

- a) O(1)  
b) O(n)  
c) O( $n^2$ )  
d) O(5)

**Answer:** b

**Explanation:** The space complexity of the above dynamic programming implementation is O(n).

10. What is the output of the following program?

```
#include<stdio.h>
#include<limits.h>
int min_jump(int *arr, int len)
{
    int j, idx, jumps[len];
    jumps[len - 1] = 0;
    for(idx = len - 2; idx >= 0; idx--)
    {
        int tmp_min = INT_MAX;
        for(j = 1; j <= arr[idx] && idx + j < len; j++)
```

```

    {
        if(jumps[idx + j] + 1
< tmp_min)
            tmp_min = jumps[idx
+ j] + 1;
        }
        jumps[idx] = tmp_min;
    }
    return jumps[0];
}
int main()
{
    int arr[] ={1, 1, 1, 1, 1, 1, 1, 1,
1},len = 9;
    int ans = min_jump(arr,len);
    printf("%d\n",ans);
    return 0;
}

```

- a) 7
- b) 8
- c) 9
- d) 10

**Answer:** b

**Explanation:** The program prints the minimum jumps required to reach the end of the array, which is 8 and so the output is 8.

11. What is the output of the following program?

```

#include<stdio.h>
#include<limits.h>
int min_jump(int *arr, int len)
{
    int j, idx, jumps[len];
    jumps[len - 1] = 0;
    for(idx = len - 2; idx >= 0; idx--)
    {
        int tmp_min = INT_MAX;
        for(j = 1; j <= arr[idx] && idx
+ j < len; j++)
        {
            if(jumps[idx + j] + 1 < t
mp_min)
                tmp_min = jumps[idx + j
] + 1;
        }
        jumps[idx] = tmp_min;
    }
    return jumps[0];
}
int main()
{

```

```

        int arr[] ={9, 9, 9, 9, 9, 9, 9,
9},len = 9;
        int ans = min_jump(arr,len);
        printf("%d\n",ans);
        return 0;
    }

```

- a) 1
- b) 6
- c) 2
- d) 7

**Answer:** a

**Explanation:** The program prints the minimum jumps required to reach the end of the array, which is 1 and so the output is 1.

1. The Knapsack problem is an example of

- a) Greedy algorithm
- b) 2D dynamic programming
- c) 1D dynamic programming
- d) Divide and conquer

**Answer:** b

**Explanation:** Knapsack problem is an example of 2D dynamic programming.

2. Which of the following methods can be used to solve the Knapsack problem?

- a) Brute force algorithm
- b) Recursion
- c) Dynamic programming
- d) Brute force, Recursion and Dynamic Programming

**Answer:** d

**Explanation:** Brute force, Recursion and Dynamic Programming can be used to solve the knapsack problem.

3. You are given a knapsack that can carry a maximum weight of 60. There are 4 items with weights {20, 30, 40, 70} and values {70, 80, 90, 200}. What is the maximum value of the items you can carry using the knapsack?

- a) 160
- b) 200

- c) 170  
d) 90

**Answer:** a

**Explanation:** The maximum value you can get is 160. This can be achieved by choosing the items 1 and 3 that have a total weight of 60.

4. Which of the following problems is equivalent to the 0-1 Knapsack problem?
- You are given a bag that can carry a maximum weight of W. You are given N items which have a weight of  $\{w_1, w_2, w_3, \dots, w_n\}$  and a value of  $\{v_1, v_2, v_3, \dots, v_n\}$ . You can break the items into smaller pieces. Choose the items in such a way that you get the maximum value
  - You are studying for an exam and you have to study N questions. The questions take  $\{t_1, t_2, t_3, \dots, t_n\}$  time(in hours) and carry  $\{m_1, m_2, m_3, \dots, m_n\}$  marks. You can study for a maximum of T hours. You can either study a question or leave it. Choose the questions in such a way that your score is maximized
  - You are given infinite coins of denominations  $\{v_1, v_2, v_3, \dots, v_n\}$  and a sum S. You have to find the minimum number of coins required to get the sum S
  - You are given a suitcase that can carry a maximum weight of 15kg. You are given 4 items which have a weight of  $\{10, 20, 15, 40\}$  and a value of  $\{1, 2, 3, 4\}$ . You can break the items into smaller pieces. Choose the items in such a way that you get the maximum value

**Answer:** b

**Explanation:** In this case, questions are used instead of items. Each question has a score which is same as each item having a value. Also, each question takes a time t which is same as each item having a weight w. You have to maximize the score in time T which is same as maximizing the value using a bag of weight W.

5. What is the time complexity of the brute force algorithm used to solve the Knapsack

- problem?  
a)  $O(n)$   
b)  $O(n!)$   
c)  $O(2^n)$   
d)  $O(n^3)$

**Answer:** c

**Explanation:** In the brute force algorithm all the subsets of the items are found and the value of each subset is calculated. The subset of items with the maximum value and a weight less than equal to the maximum allowed weight gives the answer. The time taken to calculate all the subsets is  $O(2^n)$ .

6. The 0-1 Knapsack problem can be solved using Greedy algorithm.

- True
- False

**Answer:** b

**Explanation:** The Knapsack problem cannot be solved using the greedy algorithm.

7. Consider the following dynamic programming implementation of the Knapsack problem:

```
#include<stdio.h>
int find_max(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int knapsack(int W, int *wt, int *val,int n)
{
    int ans[n + 1][W + 1];
    int itm,w;
    for(itm = 0; itm <= n; itm++)
        ans[itm][0] = 0;
    for(w = 0;w <= W; w++)
        ans[0][w] = 0;
    for(itm = 1; itm <= n; itm++)
    {
        for(w = 1; w <= W; w++)
        {
            if(wt[itm - 1] <= w)
                ans[itm][w] = _____;
            else
                ans[itm][w] = _____;
        }
    }
}
```

```

        else
            ans[item][w] = ans[item -
1][w];
    }
    return ans[n][W];
}
int main()
{
    int w[] = {10,20,30}, v[] = {60, 100
, 120}, W = 50;
    int ans = knapsack(W, w, v, 3);
    printf("%d",ans);
    return 0;
}

```

Which of the following lines completes the above code?

- a) find\_max(ans[item - 1][w - wt[item - 1]] + val[item - 1], ans[item - 1][w])
- b) find\_max(ans[item - 1][w - wt[item - 1]], ans[item - 1][w])
- c) ans[item][w] = ans[item - 1][w];
- d) ans[item+1][w] = ans[item - 1][w];

**Answer:** a

**Explanation:** find\_max(ans[item - 1][w - wt[item - 1]] + val[item - 1], ans[item - 1][w]) completes the above code.

8. What is the time complexity of the following dynamic programming implementation of the Knapsack problem with n items and a maximum weight of W?

```

#include<stdio.h>
int find_max(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int knapsack(int W, int *wt, int *val,int n)
{
    int ans[n + 1][W + 1];
    int item,w;
    for(item = 0; item <= n; item++)
        ans[item][0] = 0;
    for(w = 0;w <= W; w++)
        ans[0][w] = 0;
    for(item = 1; item <= n; item++)
    {
        for(w = 1; w <= W; w++)

```

```

        {
            if(wt[item - 1] <= w)
                ans[item][w] = find_max(
ans[item - 1][w - wt[item - 1]] + val[item -
1], ans[item - 1][w]);
            else
                ans[item][w] = ans[item -
1][w];
        }
    }
    return ans[n][W];
}
int main()
{
    int w[] = {10,20,30}, v[] = {60, 100
, 120}, W = 50;
    int ans = knapsack(W, w, v, 3);
    printf("%d",ans);
    return 0;
}

a) O(n)
b) O(n + w)
c) O(nW)
d) O(n2)

```

**Answer:** c

**Explanation:** The time complexity of the above dynamic programming implementation of the Knapsack problem is O(nW).

9. What is the space complexity of the following dynamic programming implementation of the Knapsack problem?

```

#include<stdio.h>
int find_max(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int knapsack(int W, int *wt, int *val,int n)
{
    int ans[n + 1][W + 1];
    int item,w;
    for(item = 0; item <= n; item++)
        ans[item][0] = 0;
    for(w = 0;w <= W; w++)
        ans[0][w] = 0;
    for(item = 1; item <= n; item++)
    {
        for(w = 1; w <= W; w++)

```

```

    {
        if(wt[item - 1] <= w)
            ans[item][w] = find_max(
ans[item - 1][w - wt[item - 1]] + val[item -
1], ans[item - 1][w]);
        else
            ans[item][w] = ans[item -
1][w];
    }
    return ans[n][W];
}
int main()
{
    int w[] = {10,20,30}, v[] = {60, 100
, 120}, W = 50;
    int ans = knapsack(W, w, v, 3);
    printf("%d",ans);
    return 0;
}

```

- a) O(n)
- b) O(n + w)
- c) O(nW)
- d) O(n<sup>2</sup>)

**Answer:** c

**Explanation:** The space complexity of the above dynamic programming implementation of the Knapsack problem is O(nW).

10. What is the output of the following code?

```

#include<stdio.h>
int find_max(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int knapsack(int W, int *wt, int *val,int n)
{
    int ans[n + 1][W + 1];
    int item,w;
    for(item = 0; item <= n; item++)
        ans[item][0] = 0;
    for(w = 0;w <= W; w++)
        ans[0][w] = 0;
    for(item = 1; item <= n; item++)
    {
        for(w = 1; w <= W; w++)
        {
            if(wt[item - 1] <= w)
                ans[item][w] = find_max(

```

```

ans[item - 1][w - wt[item - 1]] + val[item -
1], ans[item - 1][w]);
            else
                ans[item][w] = ans[item -
1][w];
        }
    }
    return ans[n][W];
}
int main()
{
    int w[] = {10,20,30}, v[] = {60, 100
, 120}, W = 50;
    int ans = knapsack(W, w, v, 3);
    printf("%d",ans);
    return 0;
}

```

- a) 120
- b) 100
- c) 180
- d) 220

**Answer:** d

**Explanation:** The output of the above code is 220.

1. Which of the following methods can be used to solve the matrix chain multiplication problem?

- a) Dynamic programming
- b) Brute force
- c) Recursion
- d) Dynamic Programming, Brute force, Recursion

**Answer:** d

**Explanation:** Dynamic Programming, Brute force, Recursion methods can be used to solve the matrix chain multiplication problem.

2. Which of the following is the recurrence relation for the matrix chain multiplication problem where mat[i-1] \* mat[i] gives the dimension of the ith matrix?

- a) dp[i,j] = 1 if i=j  
dp[i,j] = min{dp[i,k] + dp[k+1,j]}
- b) dp[i,j] = 0 if i=j  
dp[i,j] = min{dp[i,k] + dp[k+1,j]}

- c)  $dp[i,j] = 1$  if  $i=j$   
 $dp[i,j] = \min\{dp[i,k] + dp[k+1,j]\} + mat[i-1]*mat[k]*mat[j].$
- d)  $dp[i,j] = 0$  if  $i=j$   
 $dp[i,j] = \min\{dp[i,k] + dp[k+1,j]\} + mat[i-1]*mat[k]*mat[j].$

**Answer:** d

**Explanation:** The recurrence relation is given by:

- $dp[i,j] = 0$  if  $i=j$
- $dp[i,j] = \min\{dp[i,k] + dp[k+1,j]\} + mat[i-1]*mat[k]*mat[j].$

3. Consider the two matrices P and Q which are  $10 \times 20$  and  $20 \times 30$  matrices respectively. What is the number of multiplications required to multiply the two matrices?

- a)  $10*20$
- b)  $20*30$
- c)  $10*30$
- d)  $10*20*30$

**Answer:** d

**Explanation:** The number of multiplications required is  $10*20*30$ .

4. Consider the matrices P, Q and R which are  $10 \times 20$ ,  $20 \times 30$  and  $30 \times 40$  matrices respectively. What is the minimum number of multiplications required to multiply the three matrices?

- a) 18000
- b) 12000
- c) 24000
- d) 32000

**Answer:** a

**Explanation:** The minimum number of multiplications are 18000. This is the case when the matrices are parenthesized as  $(P*Q)*R$ .

5. Consider the matrices P, Q, R and S which are  $20 \times 15$ ,  $15 \times 30$ ,  $30 \times 5$  and  $5 \times 40$  matrices respectively. What is the minimum number of multiplications required to multiply the four matrices?

- a) 6050
- b) 7500
- c) 7750
- d) 12000

**Answer:** c

**Explanation:** The minimum number of multiplications required is 7750.

6. Consider the brute force implementation in which we find all the possible ways of multiplying the given set of n matrices. What is the time complexity of this implementation?

- a)  $O(n!)$
- b)  $O(n^3)$
- c)  $O(n^2)$
- d) Exponential

**Answer:** d

**Explanation:** The time complexity of finding all the possible ways of multiplying a set of n matrices is given by  $(n-1)^{\text{th}}$  Catalan number which is exponential.

7. Consider the following dynamic programming implementation of the matrix chain problem:

```
#include<stdio.h>
#include<limits.h>
int mat_chain_multiplication(int *mat, int n)
{
    int arr[n][n];
    int i,k,row,col,len;
    for(i=1;i<n;i++)
        arr[i][i] = 0;
    for(len = 2; len < n; len++)
    {
        for(row = 1; row <= n - len + 1; row++)
        {
            col = row + len - 1;
            arr[row][col] = INT_MAX;
            for(k = row; k <= col - 1; k++)
            {
                int tmp = _____;
                if(tmp < arr[row][col]
```

```

])           arr[row][col] = tmp;
        }
    }
    return arr[1][n - 1];
}
int main()
{
    int mat[6] = {20,5,30,10,40};
    int ans = mat_chain_multiplication(m
at,5);
    printf("%d",ans);
    return 0;
}

```

Which of the following lines should be inserted to complete the above code?

- a) arr[row][k] – arr[k + 1][col] + mat[row – 1] \* mat[k] \* mat[col];
- b) arr[row][k] + arr[k + 1][col] – mat[row – 1] \* mat[k] \* mat[col];
- c) arr[row][k] + arr[k + 1][col] + mat[row – 1] \* mat[k] \* mat[col];
- d) arr[row][k] – arr[k + 1][col] – mat[row – 1] \* mat[k] \* mat[col];

**Answer:** c

**Explanation:** The line arr[row][k] + arr[k + 1][col] + mat[row – 1] \* mat[k] \* mat[col] should be inserted to complete the above code.

8. What is the time complexity of the following dynamic programming implementation of the matrix chain problem?

```

#include<stdio.h>
#include<limits.h>
int mat_chain_multiplication(int *mat, in
t n)
{
    int arr[n][n];
    int i,k,row,col,len;
    for(i=1;i<n;i++)
        arr[i][i] = 0;
    for(len = 2; len < n; len++)
    {
        for(row = 1; row <= n - len +
1; row++)
        {
            col = row + len - 1;
            arr[row][col] = INT_MAX;
        }
    }
}

```

```

for(k = row; k <= col - 1;
{
    int tmp = arr[row][k]
+ arr[k + 1][col] + mat[row - 1] * mat[k]
* mat[col];
    if(tmp < arr[row][col])
        arr[row][col] = tmp;
}
return arr[1][n - 1];
}
int main()
{
    int mat[6] = {20,5,30,10,40};
    int ans = mat_chain_multiplication(m
at,5);
    printf("%d",ans);
    return 0;
}

a) O(1)
b) O(n)
c) O(n2)
d) O(n3)

```

**Answer:** d

**Explanation:** The time complexity of the above dynamic programming implementation of the matrix chain multiplication is O(n<sup>3</sup>).

9. What is the space complexity of the following dynamic programming implementation of the matrix chain problem?

```

#include<stdio.h>
#include<limits.h>
int mat_chain_multiplication(int *mat, in
t n)
{
    int arr[n][n];
    int i,k,row,col,len;
    for(i=1;i<n;i++)
        arr[i][i] = 0;
    for(len = 2; len < n; len++)
    {
        for(row = 1; row <= n - len +
1; row++)
        {
            col = row + len - 1;
            arr[row][col] = INT_MAX;
            for(k = row; k <= col - 1;

```

```

k++)
{
    int tmp = arr[row][k]
+ arr[k + 1][col] + mat[row - 1] * mat[k]
* mat[col];
    if(tmp < arr[row][col])
        arr[row][col] = tmp;
}
return arr[1][n - 1];
}
int main()
{
    int mat[6] = {20,5,30,10,40};
    int ans = mat_chain_multiplication(m
at,5);
    printf("%d",ans);
    return 0;
}

a) O(1)
b) O(n)
c) O( $n^2$ )
d) O( $n^3$ )

```

**Answer:** c

**Explanation:** The space complexity of the above dynamic programming implementation of the matrix chain multiplication is  $O(n^2)$ .

10. What is the output of the following code?

```
#include<stdio.h>
#include<limits.h>
int mat_chain_multiplication(int *mat, in
t n)
{
    int arr[n][n];
    int i,k,row,col,len;
    for(i=1;i<n;i++)
        arr[i][i] = 0;
    for(len = 2; len < n; len++)
    {
        for(row = 1; row <= n - len + 1
; row++)
        {
            col = row + len - 1;
            arr[row][col] = INT_MAX;
            for(k = row; k <= col - 1;
k++)
            {
                int tmp = arr[row][k]
+ arr[k + 1][col] + mat[row - 1] * mat[k]
* mat[col];
                if(tmp < arr[row][col])
                    arr[row][col] = tmp;
            }
        }
    }
}
```

```

+ arr[k + 1][col] + mat[row - 1] * mat[k]
* mat[col];
if(tmp < arr[row][col])
arr[row][col] = tmp;
}
return arr[1][n - 1];
}
int main()
{
    int mat[6] = {20,30,40,50};
    int ans = mat_chain_multiplication(m
at,4);
    printf("%d",ans);
    return 0;
}
```

- a) 64000
- b) 70000
- c) 120000
- d) 150000

**Answer:** a

**Explanation:** The program prints the minimum number of multiplications required, which is 64000.

11. What is the value stored in arr[2][3] when the following code is executed?

```
#include<stdio.h>
#include<limits.h>
int mat_chain_multiplication(int *mat, in
t n)
{
    int arr[n][n];
    int i,k,row,col,len;
    for(i=1;i<n;i++)
        arr[i][i] = 0;
    for(len = 2; len < n; len++)
    {
        for(row = 1; row <= n - len + 1
; row++)
        {
            col = row + len - 1;
            arr[row][col] = INT_MAX;
            for(k = row; k <= col - 1;
k++)
            {
                int tmp = arr[row][k]
+ arr[k + 1][col] + mat[row - 1] * mat[k]
* mat[col];
                if(tmp < arr[row][co
```

```

    1])
        arr[row][col] = tmp;
    }
}
return arr[1][n - 1];
}
int main()
{
    int mat[6] = {20,30,40,50};
    int ans = mat_chain_multiplication(m
at,4);
    printf("%d",ans);
    return 0;
}

a) 64000
b) 60000
c) 24000
d) 12000

```

**Answer:** b

**Explanation:** The value stored in arr[2][3] when the above code is executed is 60000.

12. What is the output of the following code?

```

#include<stdio.h>
#include<limits.h>
int mat_chain_multiplication(int *mat, in
t n)
{
    int arr[n][n];
    int i,k,row,col,len;
    for(i=1;i<n;i++)
        arr[i][i] = 0;
    for(len = 2; len < n; len++)
    {
        for(row = 1; row <= n - len + 1
; row++)
        {
            col = row + len - 1;
            arr[row][col] = INT_MAX;
            for(k = row; k <= col - 1;
k++)
            {
                int tmp = arr[row][k]
+ arr[k + 1][col] + mat[row - 1] * mat[k]
* mat[col];
                if(tmp < arr[row][col
])
                    arr[row][col] = tmp;
            }
        }
    }
}

```

```

    return arr[1][n-1];
}
int main()
{
    int mat[6] = {10,10,10,10,10,10};
    int ans = mat_chain_multiplication(m
at,6);
    printf("%d",ans);
    return 0;
}

a) 2000
b) 3000
c) 4000
d) 5000

```

**Answer:** c

**Explanation:** The program prints the minimum number of multiplications required to multiply the given matrices, which is 4000.

13. What is the output of the following code?

```

#include<stdio.h>
#include<limits.h>
int mat_chain_multiplication(int *mat, in
t n)
{
    int arr[n][n];
    int i,k,row,col,len;
    for(i=1;i<n;i++)
        arr[i][i] = 0;
    for(len = 2; len < n; len++)
    {
        for(row = 1; row <= n - len + 1
; row++)
        {
            col = row + len - 1;
            arr[row][col] = INT_MAX;
            for(k = row; k <= col - 1;
k++)
            {
                int tmp = arr[row][k]
+ arr[k + 1][col] + mat[row - 1] * mat[k]
* mat[col];
                if(tmp < arr[row][col
])
                    arr[row][col] = tmp;
            }
        }
    }
    return arr[1][n-1];
}
int main()
{

```

```

int mat[6] = {20,25,30,35,40};
int ans = mat_chain_multiplication(m
at,5);
printf("%d",ans);
return 0;
}

```

- a) 32000
- b) 28000
- c) 64000
- d) 70000

**Answer:** c

**Explanation:** The output of the program is 64000.

---

1. Which of the following methods can be used to solve the longest common subsequence problem?
  - a) Recursion
  - b) Dynamic programming
  - c) Both recursion and dynamic programming
  - d) Greedy algorithm

**Answer:** c

**Explanation:** Both recursion and dynamic programming can be used to solve the longest subsequence problem.

2. Consider the strings “PQRSTPQRS” and “PRATPBRQRPS”. What is the length of the longest common subsequence?

- a) 9
- b) 8
- c) 7
- d) 6

**Answer:** c

**Explanation:** The longest common subsequence is “PRTPQRS” and its length is 7.

3. Which of the following problems can be solved using the longest subsequence problem?

- a) Longest increasing subsequence
- b) Longest palindromic subsequence

- c) Longest bitonic subsequence
- d) Longest decreasing subsequence

**Answer:** b

**Explanation:** To find the longest palindromic subsequence in a given string, reverse the given string and then find the longest common subsequence in the given string and the reversed string.

4. Longest common subsequence is an example of \_\_\_\_\_
  - a) Greedy algorithm
  - b) 2D dynamic programming
  - c) 1D dynamic programming
  - d) Divide and conquer

**Answer:** b

**Explanation:** Longest common subsequence is an example of 2D dynamic programming.

5. What is the time complexity of the brute force algorithm used to find the longest common subsequence?

- a)  $O(n)$
- b)  $O(n^2)$
- c)  $O(n^3)$
- d)  $O(2^n)$

**Answer:** d

**Explanation:** The time complexity of the brute force algorithm used to find the longest common subsequence is  $O(2^n)$ .

6. Consider the following dynamic programming implementation of the longest common subsequence problem:

```

#include<stdio.h>
#include<string.h>
int max_num(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int lcs(char *str1, char *str2)
{
    int i,j,len1,len2;

```

```

len1 = strlen(str1);
len2 = strlen(str2);
int arr[len1 + 1][len2 + 1];
for(i = 0; i <= len1; i++)
    arr[i][0] = 0;
for(i = 0; i <= len2; i++)
    arr[0][i] = 0;
for(i = 1; i <= len1; i++)
{
    for(j = 1; j <= len2; j++)
    {
        if(str1[i-1] == str2[j - 1])
            _____;
        else
            arr[i][j] = max_num(ar
r[i - 1][j], arr[i][j - 1]);
    }
    return arr[len1][len2];
}
int main()
{
    char str1[] = " abcedfg", str2[] =
"bcdfh";
    int ans = lcs(str1,str2);
    printf("%d",ans);
    return 0;
}

```

Which of the following lines completes the above code?

- a)  $\text{arr}[i][j] = 1 + \text{arr}[i][j]$ .
- b)  $\text{arr}[i][j] = 1 + \text{arr}[i - 1][j - 1]$ .
- c)  $\text{arr}[i][j] = \text{arr}[i - 1][j - 1]$ .
- d)  $\text{arr}[i][j] = \text{arr}[i][j]$ .

**Answer:** b

**Explanation:** The line,  $\text{arr}[i][j] = 1 + \text{arr}[i - 1][j - 1]$  completes the above code.

7. What is the time complexity of the following dynamic programming implementation of the longest common subsequence problem where length of one string is “m” and the length of the other string is “n”?

```

#include<stdio.h>
#include<string.h>
int max_num(int a, int b)
{
    if(a > b)
        return a;
}

```

```

        return b;
}
int lcs(char *str1, char *str2)
{
    int i,j,len1,len2;
    len1 = strlen(str1);
    len2 = strlen(str2);
    int arr[len1 + 1][len2 + 1];
    for(i = 0; i <= len1; i++)
        arr[i][0] = 0;
    for(i = 0; i <= len2; i++)
        arr[0][i] = 0;
    for(i = 1; i <= len1; i++)
    {
        for(j = 1; j <= len2; j++)
        {
            if(str1[i-1] == str2[j - 1])
                arr[i][j] = 1 + arr[i - 1][j - 1];
            else
                arr[i][j] = max_num(ar
r[i - 1][j], arr[i][j - 1]);
        }
    }
    return arr[len1][len2];
}
int main()
{
    char str1[] = " abcedfg", str2[] =
"bcdfh";
    int ans = lcs(str1,str2);
    printf("%d",ans);
    return 0;
}

```

- a) O(n)
- b) O(m)
- c) O(m + n)
- d) O(mn)

**Answer:** d

**Explanation:** The time complexity of the above dynamic programming implementation of the longest common subsequence is O(mn).

8. What is the space complexity of the following dynamic programming implementation of the longest common subsequence problem where length of one string is “m” and the length of the other string is “n”?

```

#include<stdio.h>
#include<string.h>
int max_num(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int lcs(char *str1, char *str2)
{
    int i,j,len1,len2;
    len1 = strlen(str1);
    len2 = strlen(str2);
    int arr[len1 + 1][len2 + 1];
    for(i = 0; i <= len1; i++)
        arr[i][0] = 0;
    for(i = 0; i <= len2; i++)
        arr[0][i] = 0;
    for(i = 1; i <= len1; i++)
    {
        for(j = 1; j <= len2; j++)
        {
            if(str1[i-1] == str2[j - 1])
                arr[i][j] = 1 + arr[i - 1][j - 1];
            else
                arr[i][j] = max_num(arr[i - 1][j], arr[i][j - 1]);
        }
    }
    return arr[len1][len2];
}
int main()
{
    char str1[] = " abcdedfg", str2[] = "bcdfh";
    int ans = lcs(str1,str2);
    printf("%d",ans);
    return 0;
}

```

- a) O(n)
- b) O(m)
- c) O(m + n)
- d) O(mn)

**Answer:** d

**Explanation:** The space complexity of the above dynamic programming implementation of the longest common subsequence is O(mn).

9. What is the output of the following code?

```

#include<stdio.h>
#include<string.h>
int max_num(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int lcs(char *str1, char *str2)
{
    int i,j,len1,len2;
    len1 = strlen(str1);
    len2 = strlen(str2);
    int arr[len1 + 1][len2 + 1];
    for(i = 0; i <= len1; i++)
        arr[i][0] = 0;
    for(i = 0; i <= len2; i++)
        arr[0][i] = 0;
    for(i = 1; i <= len1; i++)
    {
        for(j = 1; j <= len2; j++)
        {
            if(str1[i-1] == str2[j - 1])
                arr[i][j] = 1 + arr[i - 1][j - 1];
            else
                arr[i][j] = max_num(arr[i - 1][j], arr[i][j - 1]);
        }
    }
    return arr[len1][len2];
}
int main()
{
    char str1[] = "hbcfgmnapq", str2[] = "cbhgrsfnmq";
    int ans = lcs(str1,str2);
    printf("%d",ans);
    return 0;
}

```

- a) 3
- b) 4
- c) 5
- d) 6

**Answer:** b

**Explanation:** The program prints the length of the longest common subsequence, which is 4.

10. Which of the following is the longest common subsequence between the strings “hbcfgmnapq” and “cbhgrsfnmq” ?

- a) hgmq
- b) cfnq
- c) bfmq
- d) fgmna

**Answer:** d

**Explanation:** The length of the longest common subsequence is 4. But 'fgmna' is not the longest common subsequence as its length is 5.

11. What is the value stored in arr[2][3] when the following code is executed?

```
#include<stdio.h>
#include<string.h>
int max_num(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int lcs(char *str1, char *str2)
{
    int i,j,len1,len2;
    len1 = strlen(str1);
    len2 = strlen(str2);
    int arr[len1 + 1][len2 + 1];
    for(i = 0; i <= len1; i++)
        arr[i][0] = 0;
    for(i = 0; i <= len2; i++)
        arr[0][i] = 0;
    for(i = 1; i <= len1; i++)
    {
        for(j = 1; j <= len2; j++)
        {
            if(str1[i-1] == str2[j - 1])
                arr[i][j] = 1 + arr[i - 1][j - 1];
            else
                arr[i][j] = max_num(arr[i - 1][j], arr[i][j - 1]);
        }
    }
    return arr[len1][len2];
}
int main()
{
    char str1[] = "hbcfgmnapq", str2[]
= "cbhgrsfnmq";
    int ans = lcs(str1,str2);
    printf("%d",ans);
    return 0;
}
```

- a) 1
- b) 2
- c) 3
- d) 4

**Answer:** a

**Explanation:** The value stored in arr[2][3] is 1.

12. What is the output of the following code?

```
#include<stdio.h>
#include<string.h>
int max_num(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int lcs(char *str1, char *str2)
{
    int i,j,len1,len2;
    len1 = strlen(str1);
    len2 = strlen(str2);
    int arr[len1 + 1][len2 + 1];
    for(i = 0; i <= len1; i++)
        arr[i][0] = 0;
    for(i = 0; i <= len2; i++)
        arr[0][i] = 0;
    for(i = 1; i <= len1; i++)
    {
        for(j = 1; j <= len2; j++)
        {
            if(str1[i-1] == str2[j - 1])
                arr[i][j] = 1 + arr[i - 1][j - 1];
            else
                arr[i][j] = max_num(ar
r[i - 1][j], arr[i][j - 1]);
        }
    }
    return arr[len1][len2];
}
int main()
{
    char str1[] = "abcd", str2[] = "efg
h";
    int ans = lcs(str1,str2);
    printf("%d",ans);
    return 0;
}
```

- a) 3
- b) 2

- c) 1  
d) 0

**Answer:** d

**Explanation:** The program prints the length of the longest common subsequence, which is 0.

1. Which of the following methods can be used to solve the longest palindromic subsequence problem?
  - a) Dynamic programming
  - b) Recursion
  - c) Brute force
  - d) Dynamic programming, Recursion, Brute force

**Answer:** d

**Explanation:** Dynamic programming, Recursion, Brute force can be used to solve the longest palindromic subsequence problem.

2. Which of the following is not a palindromic subsequence of the string "ababcdabba"?
  - a) abcba
  - b) abba
  - c) abbbba
  - d) adba

**Answer:** d

**Explanation:** 'adba' is not a palindromic sequence.

3. For which of the following, the length of the string is not equal to the length of the longest palindromic subsequence?
  - a) A string that is a palindrome
  - b) A string of length one
  - c) A string that has all the same letters(e.g. aaaaaa)
  - d) Some strings of length two

**Answer:** d

**Explanation:** A string of length 2 for eg: ab is not a palindrome.

4. What is the length of the longest palindromic subsequence for the string "ababcdabba"?
  - a) 6
  - b) 7
  - c) 8
  - d) 9

**Answer:** b

**Explanation:** The longest palindromic subsequence is "abbabba" and its length is 7.

5. What is the time complexity of the brute force algorithm used to find the length of the longest palindromic subsequence?
  - a)  $O(1)$
  - b)  $O(2^n)$
  - c)  $O(n)$
  - d)  $O(n^2)$

**Answer:** b

**Explanation:** In the brute force algorithm, all the subsequences are found and the length of the longest palindromic subsequence is calculated. This takes exponential time.

6. For every non-empty string, the length of the longest palindromic subsequence is at least one.
  - a) True
  - b) False

**Answer:** a

**Explanation:** A single character of any string can always be considered as a palindrome and its length is one.

7. Longest palindromic subsequence is an example of \_\_\_\_\_
  - a) Greedy algorithm
  - b) 2D dynamic programming
  - c) 1D dynamic programming
  - d) Divide and conquer

**Answer:** b

**Explanation:** Longest palindromic subsequence is an example of 2D dynamic programming.

8. Consider the following code:

```
#include<stdio.h>
#include<string.h>
int max_num(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int lps(char *str1)
{
    int i,j,len;
    len = strlen(str1);
    char str2[len + 1];
    strcpy(str2, str1);
    _____;
    int arr[len + 1][len + 1];
    for(i = 0; i <= len; i++)
        arr[i][0] = 0;
    for(i = 0; i <= len; i++)
        arr[0][i] = 0;
    for(i = 1; i <= len; i++)
    {
        for(j = 1; j <= len; j++)
        {
            if(str1[i-1] == str2[j - 1])
                arr[i][j] = 1 + arr[i - 1][j - 1];
            else
                arr[i][j] = max_num(arr[i - 1][j], arr[i][j - 1]);
        }
    }
    return arr[len][len];
}
int main()
{
    char str1[] = "ababcdabba";
    int ans = lps(str1);
    printf("%d",ans);
    return 0;
}
```

Which of the following lines completes the above code?

- a) strrev(str2);
- b) str2 = str1
- c) len2 = strlen(str2)
- d) strlen(str2)

*Answer:* a

*Explanation:* To find the longest palindromic

subsequence, we need to reverse the copy of the string, which is done by strrev.

9. What is the time complexity of the following dynamic programming implementation to find the longest palindromic subsequence where the length of the string is n?

```
#include<stdio.h>
#include<string.h>
int max_num(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int lps(char *str1)
{
    int i,j,len;
    len = strlen(str1);
    char str2[len + 1];
    strcpy(str2, str1);
    strrev(str2);
    int arr[len + 1][len + 1];
    for(i = 0; i <= len; i++)
        arr[i][0] = 0;
    for(i = 0; i <= len; i++)
        arr[0][i] = 0;
    for(i = 1; i <= len; i++)
    {
        for(j = 1; j <= len; j++)
        {
            if(str1[i-1] == str2[j - 1])
                arr[i][j] = 1 + arr[i - 1][j - 1];
            else
                arr[i][j] = max_num(arr[i - 1][j], arr[i][j - 1]);
        }
    }
    return arr[len][len];
}
int main()
{
    char str1[] = "ababcdabba";
    int ans = lps(str1);
    printf("%d",ans);
    return 0;
}
```

a) O(n)

b) O(1)

- c)  $O(n^2)$   
 d)  $O(2)$

**Answer:** c

**Explanation:** The time complexity of the above dynamic programming implementation to find the longest palindromic subsequence is  $O(n^2)$ .

10. What is the space complexity of the following dynamic programming implementation to find the longest palindromic subsequence where the length of the string is n?

```
#include<stdio.h>
#include<string.h>
int max_num(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int lps(char *str1)
{
    int i,j,len;
    len = strlen(str1);
    char str2[len + 1];
    strcpy(str2, str1);
    strrev(str2);
    int arr[len + 1][len + 1];
    for(i = 0; i <= len; i++)
        arr[i][0] = 0;
    for(i = 0; i <= len; i++)
        arr[0][i] = 0;
    for(i = 1; i <= len; i++)
    {
        for(j = 1; j <= len; j++)
        {
            if(str1[i-1] == str2[j - 1])
                arr[i][j] = 1 + arr[i - 1][j - 1];
            else
                arr[i][j] = max_num(arr[i - 1][j], arr[i][j - 1]);
        }
    }
    return arr[len][len];
}
int main()
{
    char str1[] = "ababcdabba";
    int ans = lps(str1);
```

```
    printf("%d",ans);
    return 0;
}
```

- a)  $O(n)$   
 b)  $O(1)$   
 c)  $O(n^2)$   
 d)  $O(2)$

**Answer:** c

**Explanation:** The space complexity of the above dynamic programming implementation to find the longest palindromic subsequence is  $O(n^2)$ .

11. What is the value stored in arr[3][3] when the following code is executed?

```
#include<stdio.h>
#include<string.h>
int max_num(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int lps(char *str1)
{
    int i,j,len;
    len = strlen(str1);
    char str2[len + 1];
    strcpy(str2, str1);
    strrev(str2);
    int arr[len + 1][len + 1];
    for(i = 0; i <= len; i++)
        arr[i][0] = 0;
    for(i = 0; i <= len; i++)
        arr[0][i] = 0;
    for(i = 1; i <= len; i++)
    {
        for(j = 1; j <= len; j++)
        {
            if(str1[i-1] == str2[j - 1])
                arr[i][j] = 1 + arr[i - 1][j - 1];
            else
                arr[i][j] = max_num(arr[i - 1][j], arr[i][j - 1]);
        }
    }
    return arr[len][len];
}
int main()
```

```

{
    char str1[] = "ababcdabba";
    int ans = lps(str1);
    printf("%d",ans);
    return 0;
}

a) 2
b) 3
c) 4
d) 5

```

**Answer:** a

**Explanation:** The value stored in arr[3][3] when the above code is executed is 2.

12. What is the output of the following code?

```

#include<stdio.h>
#include<string.h>
int max_num(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int lps(char *str1)
{
    int i,j,len;
    len = strlen(str1);
    char str2[len + 1];
    strcpy(str2, str1);
    strrev(str2);
    int arr[len + 1][len + 1];
    for(i = 0; i <= len; i++)
        arr[i][0] = 0;
    for(i = 0; i <= len; i++)
        arr[0][i] = 0;
    for(i = 1; i <= len; i++)
    {
        for(j = 1; j <= len; j++)
        {
            if(str1[i-1] == str2[j - 1])
                arr[i][j] = 1 + arr[i - 1][j - 1];
            else
                arr[i][j] = max_num(arr[i - 1][j], arr[i][j - 1]);
        }
        return arr[len][len];
}
int main()
{

```

```

    char str1[] = "abcd";
    int ans = lps(str1);
    printf("%d",ans);
    return 0;
}

a) 0
b) 1
c) 2
d) 3

```

**Answer:** b

**Explanation:** The program prints the length of the longest palindromic subsequence, which is 1.

13. What is the output of the following code?

```

#include<stdio.h>
#include<string.h>
int max_num(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int lps(char *str1)
{
    int i,j,len;
    len = strlen(str1);
    char str2[len + 1];
    strcpy(str2, str1);
    strrev(str2);
    int arr[len + 1][len + 1];
    for(i = 0; i <= len; i++)
        arr[i][0] = 0;
    for(i = 0; i <= len; i++)
        arr[0][i] = 0;
    for(i = 1; i <= len; i++)
    {
        for(j = 1; j <= len; j++)
        {
            if(str1[i-1] == str2[j - 1])
                arr[i][j] = 1 + arr[i - 1][j - 1];
            else
                arr[i][j] = max_num(arr[i - 1][j], arr[i][j - 1]);
        }
        return arr[len][len];
}
int main()
{

```

```

char str1[] = "abdkgkagdjbjccbbba";
int ans = lps(str1);
printf("%d",ans);
return 0;
}

a) 5
b) 7
c) 9
d) 11

```

**Answer:** c

**Explanation:** The program prints the length of the longest palindromic subsequence, which is 9.

---

1. Which of the following methods can be used to solve the edit distance problem?
  - a) Recursion
  - b) Dynamic programming
  - c) Both dynamic programming and recursion
  - d) Greedy Algorithm

**Answer:** c

**Explanation:** Both dynamic programming and recursion can be used to solve the edit distance problem.

2. The edit distance satisfies the axioms of a metric when the costs are non-negative.
  - a) True
  - b) False

**Answer:** a

**Explanation:**  $d(s,s) = 0$ , since each string can be transformed into itself without any change.  $d(s_1, s_2) > 0$  when  $s_1 \neq s_2$ , since the transformation would require at least one operation.

$$\begin{aligned} d(s_1, s_2) &= d(s_2, s_1) \\ d(s_1, s_3) &\leq d(s_1, s_2) + d(s_2, s_3) \end{aligned}$$

Thus, the edit distance satisfies the axioms of a metric.

3. Which of the following is an application of the edit distance problem?
  - a) Approximate string matching
  - b) Spelling correction

- c) Similarity of DNA
- d) Approximate string matching, Spelling Correction and Similarity of DNA

**Answer:** d

**Explanation:** All of the mentioned are the applications of the edit distance problem.

4. In which of the following cases will the edit distance between two strings be zero?
  - a) When one string is a substring of another
  - b) When the lengths of the two strings are equal
  - c) When the two strings are equal
  - d) The edit distance can never be zero

**Answer:** c

**Explanation:** The edit distance will be zero only when the two strings are equal.

5. Suppose each edit (insert, delete, replace) has a cost of one. Then, the maximum edit distance cost between the two strings is equal to the length of the larger string.
  - a) True
  - b) False

**Answer:** a

**Explanation:** Consider the strings “abcd” and “efghi”. The string “efghi” can be converted to “abcd” by deleting “i” and converting “efgh” to “abcd”. The cost of transformation is 5, which is equal to the length of the larger string.

6. Consider the strings “monday” and “tuesday”. What is the edit distance between the two strings?
  - a) 3
  - b) 4
  - c) 5
  - d) 6

**Answer:** b

**Explanation:** “monday” can be converted to “tuesday” by replacing “m” with “t”, “o” with “u”, “n” with “e” and inserting “s” at the appropriate position. So, the edit distance is 4.

7. Consider the two strings “”(empty string) and “abcd”. What is the edit distance between the two strings?

- a) 0
- b) 4
- c) 2
- d) 3

**Answer:** b

**Explanation:** The empty string can be transformed into “abcd” by inserting “a”, “b”, “c” and “d” at appropriate positions. Thus, the edit distance is 4.

8. Consider the following dynamic programming implementation of the edit distance problem:

```
#include<stdio.h>
#include<string.h>
int get_min(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int edit_distance(char *s1, char *s2)
{
    int len1, len2, i, j, min;
    len1 = strlen(s1);
    len2 = strlen(s2);
    int arr[len1 + 1][len2 + 1];
    for(i = 0; i <= len1; i++)
        arr[i][0] = i;
    for(i = 0; i <= len2; i++)
        arr[0][i] = i;
    for(i = 1; i <= len1; i++)
    {
        for(j = 1; j <= len2; j++)
        {
            min = get_min(arr[i-1][j],
arr[i][j-1]) + 1;
            if(s1[i - 1] == s2[j - 1])
            {
                if(arr[i-1][j-1] < mi
n)
                    min = arr[i-1][j-1]
                else
                {
                    if(arr[i-1][j-1] + 1
< min)
                        min = arr[i-1][j]
```

```
-1] + 1;
            }
        }
    }
    return arr[len1][len2];
}
int main()
{
    char s1[] = "abcd", s2[] = "defg";
    int ans = edit_distance(s1, s2);
    printf("%d", ans);
    return 0;
}
```

Which of the following lines should be added to complete the above code?

- a) arr[i-1][j] = min
- b) arr[i][j-1] = min
- c) arr[i-1][j-1] = min
- d) arr[i][j] = min

**Answer:** d

**Explanation:** The line arr[i][j] = min completes the above code.

9. What is the time complexity of the following dynamic programming implementation of the edit distance problem where “m” and “n” are the lengths of two strings?

```
#include<stdio.h>
#include<string.h>
int get_min(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int edit_distance(char *s1, char *s2)
{
    int len1, len2, i, j, min;
    len1 = strlen(s1);
    len2 = strlen(s2);
    int arr[len1 + 1][len2 + 1];
    for(i = 0; i <= len1; i++)
        arr[i][0] = i;
    for(i = 0; i <= len2; i++)
        arr[0][i] = i;
    for(i = 1; i <= len1; i++)
    {
        for(j = 1; j <= len2; j++)
        {
            min = get_min(arr[i-1][j],
```

```

        min = get_min(arr[i-1][j],
arr[i][j-1]) + 1;
        if(s1[i - 1] == s2[j - 1])
        {
            if(arr[i-1][j-1] < mi
n)
                min = arr[i-1][j-1
];
            else
            {
                if(arr[i-1][j-1] + 1
< min)
                    min = arr[i-1][j
-1] + 1;
            }
            arr[i][j] = min;
        }
    return arr[len1][len2];
}
int main()
{
    char s1[] = "abcd", s2[] = "defg";
    int ans = edit_distance(s1, s2);
    printf("%d",ans);
    return 0;
}

```

- a) O(1)
- b) O(m + n)
- c) O(mn)
- d) O(n)

**Answer:** c

**Explanation:** The time complexity of the above dynamic programming implementation of the edit distance problem is O(mn).

10. What is the space complexity of the following dynamic programming implementation of the edit distance problem where “m” and “n” are the lengths of the two strings?

```

#include<stdio.h>
#include<string.h>
int get_min(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int edit_distance(char *s1, char *s2)

```

```

int len1,len2,i,j,min;
len1 = strlen(s1);
len2 = strlen(s2);
int arr[len1 + 1][len2 + 1];
for(i = 0;i <= len1; i++)
    arr[i][0] = i;
for(i = 0; i <= len2; i++)
    arr[0][i] = i;
for(i = 1; i <= len1; i++)
{
    for(j = 1; j <= len2; j++)
    {
        min = get_min(arr[i-1][j],
arr[i][j-1]) + 1;
        if(s1[i - 1] == s2[j - 1])
        {
            if(arr[i-1][j-1] < mi
n)
                min = arr[i-1][j-1
];
            else
            {
                if(arr[i-1][j-1] + 1
< min)
                    min = arr[i-1][j
-1] + 1;
            }
            arr[i][j] = min;
        }
    }
}
return arr[len1][len2];
}
int main()
{
    char s1[] = "abcd", s2[] = "defg";
    int ans = edit_distance(s1, s2);
    printf("%d",ans);
    return 0;
}

```

- a) O(1)
- b) O(m + n)
- c) O(mn)
- d) O(n)

**Answer:** c

**Explanation:** The space complexity of the above dynamic programming implementation of the edit distance problem is O(mn).

11. What is the output of the following code?

```

#include<stdio.h>
#include<string.h>
int get_min(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int edit_distance(char *s1, char *s2)
{
    int len1,len2,i,j,min;
    len1 = strlen(s1);
    len2 = strlen(s2);
    int arr[len1 + 1][len2 + 1];
    for(i = 0;i <= len1; i++)
        arr[i][0] = i;
    for(i = 0; i <= len2; i++)
        arr[0][i] = i;
    for(i = 1; i <= len1; i++)
    {
        for(j = 1; j <= len2; j++)
        {
            min = get_min(arr[i-1][j],arr[i][j-1]) + 1;
            if(s1[i - 1] == s2[j - 1])
            {
                if(arr[i-1][j-1] < min)
                    min = arr[i-1][j-1];
            }
            else
            {
                if(arr[i-1][j-1] + 1 < min)
                    min = arr[i-1][j-1] + 1;
            }
            arr[i][j] = min;
        }
    }
    return arr[len1][len2];
}
int main()
{
    char s1[] = "abcd", s2[] = "defg";
    int ans = edit_distance(s1, s2);
    printf("%d",ans);
    return 0;
}

a) 1
b) 2
c) 3
d) 4

```

**Answer:** d

**Explanation:** The program prints the edit distance between the strings “abcd” and “defg”, which is 4.

12. What is the value stored in arr[2][2] when the following code is executed?

```

#include<stdio.h>
#include<string.h>
int get_min(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int edit_distance(char *s1, char *s2)
{
    int len1,len2,i,j,min;
    len1 = strlen(s1);
    len2 = strlen(s2);
    int arr[len1 + 1][len2 + 1];
    for(i = 0;i <= len1; i++)
        arr[i][0] = i;
    for(i = 0; i <= len2; i++)
        arr[0][i] = i;
    for(i = 1; i <= len1; i++)
    {
        for(j = 1; j <= len2; j++)
        {
            min = get_min(arr[i-1][j],arr[i][j-1]) + 1;
            if(s1[i - 1] == s2[j - 1])
            {
                if(arr[i-1][j-1] < min)
                    min = arr[i-1][j-1];
            }
            else
            {
                if(arr[i-1][j-1] + 1 < min)
                    min = arr[i-1][j-1] + 1;
            }
            arr[i][j] = min;
        }
    }
    return arr[len1][len2];
}
int main()
{
    char s1[] = "abcd", s2[] = "defg";
    int ans = edit_distance(s1, s2);
    printf("%d",ans);
    return 0;
}

a) 1
b) 2
c) 3
d) 4

```

```

        printf("%d",ans);
        return 0;
    }
}

```

- a) 1
- b) 2
- c) 3
- d) 4

**Answer:** b

**Explanation:** The value stored in arr[2][2] when the above code is executed is 2.

13. What is the output of the following code?

```

#include<stdio.h>
#include<string.h>
int get_min(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int edit_distance(char *s1, char *s2)
{
    int len1,len2,i,j,min;
    len1 = strlen(s1);
    len2 = strlen(s2);
    int arr[len1 + 1][len2 + 1];
    for(i = 0;i <= len1; i++)
        arr[i][0] = i;
    for(i = 0; i <= len2; i++)
        arr[0][i] = i;
    for(i = 1; i <= len1; i++)
    {
        for(j = 1; j <= len2; j++)
        {
            min = get_min(arr[i-1][j],a
rr[i][j-1]) + 1;
            if(s1[i - 1] == s2[j - 1])
            {
                if(arr[i-1][j-1] < min)
                    min = arr[i-1][j-1];
            }
            else
            {
                if(arr[i-1][j-1] + 1 <
min)
                    min = arr[i-1][j-1]
+ 1;
            }
            arr[i][j] = min;
        }
    }
    return arr[len1][len2];
}

```

```

    }
int main()
{
    char s1[] = "pqrstuv", s2[] = "prstu
v";
    int ans = edit_distance(s1, s2);
    printf("%d",ans);
    return 0;
}

```

- a) 1
- b) 2
- c) 3
- d) 4

**Answer:** a

**Explanation:** The code prints the edit distance between the two strings, which is 1.

1. Wagner–Fischer is a \_\_\_\_\_ algorithm.

- a) Brute force
- b) Greedy
- c) Dynamic programming
- d) Recursive

**Answer:** c

**Explanation:** Wagner–Fischer belongs to the dynamic programming type of algorithms.

2. Wagner–Fischer algorithm is used to find

- 
- a) Longest common subsequence
  - b) Longest increasing subsequence
  - c) Edit distance between two strings
  - d) Longest decreasing subsequence

**Answer:** c

**Explanation:** Wagner–Fischer algorithm is used to find the edit distance between two strings.

3. What is the edit distance between the strings “abcd” and “acbd” when the allowed operations are insertion, deletion and substitution?

- a) 1
- b) 2

- c) 3  
d) 4

**Answer:** b

**Explanation:** The string “abcd” can be changed to “acbd” by substituting “b” with “c” and “c” with “b”. Thus, the edit distance is 2.

4. For which of the following pairs of strings is the edit distance maximum?

- a) sunday & monday  
b) monday & tuesday  
c) tuesday & wednesday  
d) wednesday & thursday

**Answer:** d

**Explanation:** The edit distances are 2, 4, 4 and 5 respectively. Hence, the maximum edit distance is between the strings wednesday and thursday.

5. Consider the following implementation of the Wagner–Fischer algorithm:

```
int get_min(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int edit_distance(char *s1, char *s2)
{
    int len1, len2, i, j, min;
    len1 = strlen(s1);
    len2 = strlen(s2);
    int arr[len1 + 1][len2 + 1];
    for(i = 0; i <= len1; i++)
        arr[i][0] = i;
    for(i = 0; i <= len2; i++)
        arr[0][i] = i;
    for(i = 1; i <= len1; i++)
    {
        for(j = 1; j <= len2; j++)
        {
            min = get_min(arr[i-1][j], a
rr[i][j-1]) + 1;
            if(s1[i - 1] == s2[j - 1])
            {
                if(arr[i-1][j-1] < min)
                    _____;
            }
            else
```

```
{           if(arr[i-1][j-1] + 1 <
min)
            min = arr[i-1][j-1]
+ 1;
        }
        arr[i][j] = min;
    }
    return arr[len1][len2];
}
```

Which of the following lines should be inserted to complete the above code?

- a) arr[i][j] = min  
b) min = arr[i-1][j-1] – 1;  
c) min = arr[i-1][j-1].  
d) min = arr[i-1][j-1] + 1;

**Answer:** c

**Explanation:** The line min = arr[i-1][j-1] completes the above code.

6. What is the time complexity of the Wagner–Fischer algorithm where “m” and “n” are the lengths of the two strings?

- a) O(1)  
b) O(n+m)  
c) O(mn)  
d) O(nlogm)

**Answer:** c

**Explanation:** The time complexity of the Wagner–Fischer algorithm is O(mn).

7. What is the space complexity of the above implementation of Wagner–Fischer algorithm where “m” and “n” are the lengths of the two strings?

- a) O(1)  
b) O(n+m)  
c) O(mn)  
d) O(nlogm)

**Answer:** c

**Explanation:** The space complexity of the above Wagner–Fischer algorithm is O(mn).

8. What is the output of the following code?

```
#include<stdio.h>
#include<string.h>
int get_min(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int edit_distance(char *s1, char *s2)
{
    int len1,len2,i,j,min;
    len1 = strlen(s1);
    len2 = strlen(s2);
    int arr[len1 + 1][len2 + 1];
    for(i = 0;i <= len1; i++)
        arr[i][0] = i;
    for(i = 0; i <= len2; i++)
        arr[0][i] = i;
    for(i = 1; i <= len1; i++)
    {
        for(j = 1; j <= len2; j++)
        {
            min = get_min(arr[i-1][j],
arr[i][j-1]) + 1;
            if(s1[i - 1] == s2[j - 1])
            {
                if(arr[i-1][j-1] < min)
                    min = arr[i-1][j-1];
            }
            else
            {
                if(arr[i-1][j-1] + 1 < min)
                    min = arr[i-1][j-1] + 1;
            }
            arr[i][j] = min;
        }
    }
    return arr[len1][len2];
}
int main()
{
    char s1[] = "somestring", s2[] = "anotherthing";
    int ans = edit_distance(s1, s2);
    printf("%d",ans);
    return 0;
}

a) 6
b) 7
c) 8
d) 9
```

**Answer:** a

**Explanation:** The program prints the edit distance between the strings “somestring” and “anotherthing”, which is 6.

9. What is the value stored in arr[3][3] when the below code is executed?

```
#include<stdio.h>
#include<string.h>
int get_min(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int edit_distance(char *s1, char *s2)
{
    int len1,len2,i,j,min;
    len1 = strlen(s1);
    len2 = strlen(s2);
    int arr[len1 + 1][len2 + 1];
    for(i = 0;i <= len1; i++)
        arr[i][0] = i;
    for(i = 0; i <= len2; i++)
        arr[0][i] = i;
    for(i = 1; i <= len1; i++)
    {
        for(j = 1; j <= len2; j++)
        {
            min = get_min(arr[i-1][j],
arr[i][j-1]) + 1;
            if(s1[i - 1] == s2[j - 1])
            {
                if(arr[i-1][j-1] < min)
                    min = arr[i-1][j-1];
            }
            else
            {
                if(arr[i-1][j-1] + 1 < min)
                    min = arr[i-1][j-1] + 1;
            }
            arr[i][j] = min;
        }
    }
    return arr[len1][len2];
}
int main()
{
    char s1[] = "somestring", s2[] = "anotherthing";
```

```

    int ans = edit_distance(s1, s2);
    printf("%d",ans);
    return 0;
}

a) 1
b) 2
c) 3
d) 4

```

**Answer:** c

**Explanation:** The value stored in arr[3][3] is 3.

10. What is the output of the following code?

```

#include<stdio.h>
#include<string.h>
int get_min(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int edit_distance(char *s1, char *s2)
{
    int len1,len2,i,j,min;
    len1 = strlen(s1);
    len2 = strlen(s2);
    int arr[len1 + 1][len2 + 1];
    for(i = 0;i <= len1; i++)
        arr[i][0] = i;
    for(i = 0; i <= len2; i++)
        arr[0][i] = i;
    for(i = 1; i <= len1; i++)
    {
        for(j = 1; j <= len2; j++)
        {
            min = get_min(arr[i-1][j],a
rr[i][j-1]) + 1;
            if(s1[i - 1] == s2[j - 1])
            {
                if(arr[i-1][j-1] < min)
                    min = arr[i-1][j-1];
            }
            else
            {
                if(arr[i-1][j-1] + 1 <
min)
                    min = arr[i-1][j-1];
            }
            arr[i][j] = min;
        }
    }
}

```

```

    return arr[len1][len2];
}
int main()
{
    char s1[] = "abcd", s2[] = "dcba";
    int ans = edit_distance(s1, s2);
    printf("%d",ans);
    return 0;
}
```

a) 1

b) 2

c) 3

d) 4

**Answer:** d

**Explanation:** The program prints the edit distance between the strings “abcd” and “dcba”, which is 4.

1. Which of the following is NOT a Catalan number?

a) 1

b) 5

c) 14

d) 43

**Answer:** d

**Explanation:** Catalan numbers are given by:  $(2n!)/((n+1)!n!)$ .

For  $n = 0$ , we get  $C_0 = 1$ .

For  $n = 3$ , we get  $C_3 = 5$ .

For  $n = 4$ , we get  $C_4 = 14$ .

For  $n = 5$ , we get  $C_5 = 42$ .

2. Which of the following numbers is the 6th Catalan number?

a) 14

b) 429

c) 132

d) 42

**Answer:** d

**Explanation:** Catalan numbers are given by:  $(2n!)/((n+1)!n!)$ .

First Catalan number is given by  $n = 0$ .

So the 6th Catalan number will be given by  $n = 5$ , which is 42.

3. Which of the following is not an application of Catalan Numbers?
- Counting the number of Dyck words
  - Counting the number of expressions containing n pairs of parenthesis
  - Counting the number of ways in which a convex polygon can be cut into triangles by connecting vertices with straight lines
  - Creation of head and tail for a given number of tosses

**Answer:** d

**Explanation:** Counting the number of Dyck words, Counting the number of expressions containing n pairs of parenthesis, Counting the number of ways in which a convex polygon can be cut into triangles by connecting vertices with straight lines are the applications of Catalan numbers where as creation of head and tails for a given number of tosses is an application of Pascal's triangle.

4. Which of the following methods can be used to find the nth Catalan number?
- Recursion
  - Binomial coefficients
  - Dynamic programming
  - Recursion, Binomial Coefficients, Dynamic programming

**Answer:** d

**Explanation:** All of the mentioned methods can be used to find the nth Catalan number.

5. The recursive formula for Catalan number is given by  $C_n = \sum C_i * C_{n-i}$ . Consider the following dynamic programming implementation for Catalan numbers:

```
#include<stdio.h>
int cat_number(int n)
{
    int i,j,arr[n],k;
    arr[0] = 1;
    for(i = 1; i < n; i++)
    {
        arr[i] = 0;
        for(j = 0,k = i - 1; j < i; j++,
k--)

```

```
    }
    return arr[n-1];
}

int main()
{
    int ans, n = 8;
    ans = cat_number(n);
    printf("%d\n",ans);
    return 0;
}
```

Which of the following lines completes the above code?

- $arr[i] = arr[j] * arr[k];$
- $arr[j] += arr[i] * arr[k];$
- $arr[i] += arr[j] * arr[k].$
- $arr[j] = arr[i] * arr[k];$

**Answer:** c

**Explanation:** The line  $arr[i] += arr[j] * arr[k]$  reflects the recursive formula  $C_n = \sum C_i * C_{n-i}$ .

6. Which of the following implementations of Catalan numbers has the smallest time complexity?
- Dynamic programming
  - Binomial coefficients
  - Recursion
  - All have equal time complexity

**Answer:** b

**Explanation:** The time complexities are as follows:

Dynamic programming:  $O(n^2)$

Recursion: Exponential

Binomial coefficients:  $O(n)$ .

7. What is the output of the following code?

```
#include<stdio.h>
int cat_number(int n)
{
    int i,j,arr[n],k;
    arr[0] = 1;
    for(i = 1; i < n; i++)
    {
        arr[i] = 0;
        for(j = 0,k = i - 1; j < i; j++,
k--)
```

```

        arr[i] += arr[j] * arr[k];
    }
    return arr[n-1];
}

```

```

int main()
{
    int ans, n = 8;
    ans = cat_number(n);
    printf("%d\n", ans);
    return 0;
}

```

- a) 42
- b) 132
- c) 429
- d) 1430

**Answer:** c

**Explanation:** The program prints the 8th Catalan number, which is 429.

8. Which of the following implementations of Catalan numbers has the largest space complexity(Don't consider the stack space)?

- a) Dynamic programming
- b) Binomial coefficients
- c) Recursion
- d) All have equal space complexities

**Answer:** a

**Explanation:** The space complexities are as follows:

Dynamic programming: O(n)

Recursion: O(1)

Binomial coefficients: O(1).

9. What will be the value stored in arr[5] when the following code is executed?

```

#include<stdio.h>
int cat_number(int n)
{
    int i,j,arr[n],k;
    arr[0] = 1;
    for(i = 1; i < n; i++)
    {
        arr[i] = 0;
        for(j = 0,k = i - 1; j < i; j++, k--)
            arr[i] += arr[j] * arr[k];
    }
    return arr[n-1];
}

```

```

    }
    int main()
    {
        int ans, n = 10;
        ans = cat_number(n);
        printf("%d\n",ans);
        return 0;
    }

```

- a) 14
- b) 42
- c) 132
- d) 429

**Answer:** b

**Explanation:** The 6th Catalan number will be stored in arr[5], which is 42.

10. Which of the following errors will occur when the below code is executed?

```

#include<stdio.h>
int cat_number(int n)
{
    int i,j,arr[n],k;
    arr[0] = 1;
    for(i = 1; i < n; i++)
    {
        arr[i] = 0;
        for(j = 0,k = i - 1; j < i; j++, k--)
            arr[i] += arr[j] * arr[k];
    }
    return arr[n-1];
}
int main()
{
    int ans, n = 100;
    ans = cat_number(n);
    printf("%d\n",ans);
    return 0;
}

```

- a) Segmentation fault
- b) Array size too large
- c) Integer value out of range
- d) Array index out of range

**Answer:** c

**Explanation:** The 100th Catalan number is too large to be stored in an integer. So, the error produced will be integer value out of

range.(It will be a logical error and the compiler wont show it).

1. Which of the following methods can be used to solve the assembly line scheduling problem?
  - a) Recursion
  - b) Brute force
  - c) Dynamic programming
  - d) All of the mentioned

**Answer:** d

**Explanation:** All of the above mentioned methods can be used to solve the assembly line scheduling problem.

2. What is the time complexity of the brute force algorithm used to solve the assembly line scheduling problem?
  - a) O(1)
  - b) O(n)
  - c) O( $n^2$ )
  - d) O( $2^n$ )

**Answer:** d

**Explanation:** In the brute force algorithm, all the possible ways are calculated which are of the order of  $2^n$ .

3. In the dynamic programming implementation of the assembly line scheduling problem, how many lookup tables are required?
  - a) 0
  - b) 1
  - c) 2
  - d) 3

**Answer:** c

**Explanation:** In the dynamic programming implementation of the assembly line scheduling problem, 2 lookup tables are required one for storing the minimum time and the other for storing the assembly line number.

4. Consider the following assembly line problem:

```
time_to_reach[2][3] = {{17, 2, 7}, {19,
4, 9}}
```

```
time_spent[2][4] = {{6, 5, 15, 7}, {5, 1
0, 11, 4}}
```

```
entry_time[2] = {8, 10}
```

```
exit_time[2] = {10, 7}
```

```
num_of_stations = 4
```

For the optimal solution which should be the starting assembly line?

- a) Line 1
- b) Line 2
- c) All of the mentioned
- d) None of the mentioned

**Answer:** b

**Explanation:** For the optimal solution, the starting assembly line is line 2.

5. Consider the following assembly line problem:

```
time_to_reach[2][3] = {{17, 2, 7}, {19,
4, 9}}
```

```
time_spent[2][4] = {{6, 5, 15, 7}, {5, 1
0, 11, 4}}
```

```
entry_time[2] = {8, 10}
```

```
exit_time[2] = {10, 7}
```

```
num_of_stations = 4
```

For the optimal solution, which should be the exit assembly line?

- a) Line 1
- b) Line 2
- c) All of the mentioned
- d) None of the mentioned

**Answer:** b

**Explanation:** For the optimal solution, the exit assembly line is line 2.

6. Consider the following assembly line problem:

```
time_to_reach[2][3] = {{17, 2, 7}, {19, 4, 9}}
```

```
time_spent[2][4] = {{6, 5, 15, 7}, {5, 10, 11, 4}}
```

```
entry_time[2] = {8, 10}
```

```
exit_time[2] = {10, 7}
```

```
num_of_stations = 4
```

What is the minimum time required to build the car chassis?

- a) 40
- b) 41
- c) 42
- d) 43

**Answer:** d

**Explanation:** The minimum time required is 43.

The path is S<sub>2,1</sub> -> S<sub>1,2</sub> -> S<sub>2,3</sub> -> S<sub>2,4</sub>, where S<sub>i,j</sub> : i = line number, j = station number

7. Consider the following code:

```
#include<stdio.h>
int get_min(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int minimum_time_required(int reach[][3],
int spent[][4], int *entry, int *exit, int n)
{
    int t1[n], t2[n], i;
    t1[0] = entry[0] + spent[0][0];
    t2[0] = entry[1] + spent[1][0];
    for(i = 1; i < n; i++)
    {
        t1[i] = get_min(t1[i-1]+spent[0][i],
        t2[i-1]+reach[1][i-1]+spent[0][i]);
        _____;
    }
    return get_min(t1[n-1]+exit[0], t2[n-1]+exit[1]);
}
```

Which of the following lines should be inserted to complete the above code?

- a) t2[i] = get\_min(t2[i-1]+spent[1][i], t1[i-1]+reach[0][i-1]+spent[1][i])
- b) t2[i] = get\_min(t2[i-1]+spent[1][i], t1[i-1]+spent[1][i])
- c) t2[i] = get\_min(t2[i-1]+spent[1][i], t1[i-1]+reach[0][i-1])
- d) none of the mentioned

**Answer:** a

**Explanation:** The line t2[i] = get\_min(t2[i-1]+spent[1][i], t1[i-1]+reach[0][i-1]+spent[1][i]) should be added to complete the above code.

8. What is the time complexity of the above dynamic programming implementation of the assembly line scheduling problem?

- a) O(1)
- b) O(n)
- c) O(n<sup>2</sup>)
- d) O(n<sup>3</sup>)

**Answer:** b

**Explanation:** The time complexity of the above dynamic programming implementation of the assembly line scheduling problem is O(n).

9. What is the space complexity of the above dynamic programming implementation of the assembly line scheduling problem?

- a) O(1)
- b) O(n)
- c) O(n<sup>2</sup>)
- d) O(n<sup>3</sup>)

**Answer:** b

**Explanation:** The space complexity of the above dynamic programming implementation of the assembly line scheduling problem is O(n).

10. What is the output of the following code?

```
#include<stdio.h>
int get_min(int a, int b)
```

```

{
    if(a<b)
        return a;
    return b;
}
int minimum_time_required(int reach[][3],
int spent[][4], int *entry, int *exit, in
t n)
{
    int t1[n], t2[n], i;
    t1[0] = entry[0] + spent[0][0];
    t2[0] = entry[1] + spent[1][0];
    for(i = 1; i < n; i++)
    {
        t1[i] = get_min(t1[i-1]+spent[0]
[i], t2[i-1]+reach[1][i-1]+spent[0][i]);
        t2[i] = get_min(t2[i-1]+spent[1]
[i], t1[i-1]+reach[0][i-1]+spent[1][i]);
    }
    return get_min(t1[n-1]+exit[0], t2[n-
1]+exit[1]);
}
int main()
{
    int time_to_reach[][3] = {{6, 1, 5},
                               {2, 4, 7}};
    int time_spent[][4] = {{6, 5, 4, 7},
                           {5, 10, 2, 6}};
    int entry_time[2] = {5, 6};
    int exit_time[2] = {8, 9};
    int num_of_stations = 4;
    int ans = minimum_time_required(time_
to_reach, time_spent, entry_time, exit_t
ime, num_of_stations);
    printf("%d",ans);
    return 0;
}

a) 32
b) 33
c) 34
d) 35

```

**Answer:** c

**Explanation:** The program prints the optimal time required to build the car chassis, which is 34.

11. What is the value stored in t1[2] when the following code is executed?

```
#include<stdio.h>
int get_min(int a, int b)
{
    if(a<b)
        return a;
    return b;
}
```

```

        return a;
    return b;
}
int minimum_time_required(int reach[][3],
int spent[][4], int *entry, int *exit, in
t n)
{
    int t1[n], t2[n], i;
    t1[0] = entry[0] + spent[0][0];
    t2[0] = entry[1] + spent[1][0];
    for(i = 1; i < n; i++)
    {
        t1[i] = get_min(t1[i-1]+spent[0]
[i], t2[i-1]+reach[1][i-1]+spent[0][i]);
        t2[i] = get_min(t2[i-1]+spent[1]
[i], t1[i-1]+reach[0][i-1]+spent[1][i]);
    }
    return get_min(t1[n-1]+exit[0], t2[n-
1]+exit[1]);
}
int main()
{
    int time_to_reach[][3] = {{6, 1, 5},
                               {2, 4, 7}};
    int time_spent[][4] = {{6, 5, 4, 7},
                           {5, 10, 2, 6}};
    int entry_time[2] = {5, 6};
    int exit_time[2] = {8, 9};
    int num_of_stations = 4;
    int ans = minimum_time_required(time_
to_reach, time_spent, entry_time, exit_t
ime, num_of_stations);
    printf("%d",ans);
    return 0;
}
```

- a) 16
- b) 18
- c) 20
- d) 22

**Answer:** c

**Explanation:** The value stored in t1[2] when the above code is executed is 20.

12. What is the value stored in t2[3] when the following code is executed?

```
#include<stdio.h>
int get_min(int a, int b)
{
    if(a<b)
        return a;
    return b;
}
```

```

int minimum_time_required(int reach[][3],
int spent[][4], int *entry, int *exit, int n)
{
    int t1[n], t2[n];
    t1[0] = entry[0] + spent[0][0];
    t2[0] = entry[1] + spent[1][0];
    for(i = 1; i < n; i++)
    {
        t1[i] = get_min(t1[i-1]+spent[0]
[i], t2[i-1]+reach[1][i-1]+spent[0][i]);
        t2[i] = get_min(t2[i-1]+spent[1]
[i], t1[i-1]+reach[0][i-1]+spent[1][i]);
    }
    return get_min(t1[n-1]+exit[0], t2[n-1]+exit[1]);
}
int main()
{
    int time_to_reach[][3] = {{6, 1, 5},
                               {2, 4, 7}};
    int time_spent[][4] = {{6, 5, 4, 7},
                           {5, 10, 2, 6}};
    int entry_time[2] = {5, 6};
    int exit_time[2] = {8, 9};
    int num_of_stations = 4;
    int ans = minimum_time_required(time_
to_reach, time_spent, entry_time, exit_time,
num_of_stations);
    printf("%d", ans);
    return 0;
}

```

- a) 19
- b) 23
- c) 25
- d) 27

**Answer:** c

**Explanation:** The value stored in t2[3] when the above code is executed is 25.

13. What is the output of the following code?

```

#include<stdio.h>
int get_min(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
int minimum_time_required(int reach[][4],
int spent[][5], int *entry, int *exit, int n)
{

```

```

    int t1[n], t2[n];
    t1[0] = entry[0] + spent[0][0];
    t2[0] = entry[1] + spent[1][0];
    for(i = 1; i < n; i++)
    {
        t1[i] = get_min(t1[i-1]+spent[0]
[i], t2[i-1]+reach[1][i-1]+spent[0][i]);
        t2[i] = get_min(t2[i-1]+spent[1]
[i], t1[i-1]+reach[0][i-1]+spent[1][i]);
    }
    return get_min(t1[n-1]+exit[0], t2[n-1]+exit[1]);
}
int main()
{
    int time_to_reach[][4] = {{16, 10, 5,
                               12}, {12, 4, 17, 8}};
    int time_spent[][5] = {{13, 5, 20, 1,
                           9}, {15, 10, 12, 16,
                           13}};
    int entry_time[2] = {12, 9};
    int exit_time[2] = {10, 13};
    int num_of_stations = 5;
    int ans = minimum_time_required(time_
to_reach, time_spent, entry_time, exit_time,
num_of_stations);
    printf("%d", ans);
    return 0;
}

```

- a) 62
- b) 69
- c) 75
- d) 88

**Answer:** d

**Explanation:** The program prints the optimal time required to build the car chassis, which is 88.

1. Given a string, you have to find the minimum number of characters to be inserted in the string so that the string becomes a palindrome. Which of the following methods can be used to solve the problem?
  - a) Greedy algorithm
  - b) Recursion
  - c) Dynamic programming
  - d) Both recursion and dynamic programming

**Answer:** d

**Explanation:** Dynamic programming and recursion can be used to solve the problem.

2. In which of the following cases the minimum no of insertions to form palindrome is maximum?
- String of length one
  - String with all same characters
  - Palindromic string
  - Non palindromic string

**Answer:** d

**Explanation:** In string of length one, string with all same characters and a palindromic string the no of insertions is zero since the strings are already palindromes. To convert a non-palindromic string to a palindromic string, the minimum length of string to be added is 1 which is greater than all the other above cases. Hence the minimum no of insertions to form palindrome is maximum in non-palindromic strings.

3. In the worst case, the minimum number of insertions to be made to convert the string into a palindrome is equal to the length of the string.

- True
- False

**Answer:** b

**Explanation:** In the worst case, the minimum number of insertions to be made to convert the string into a palindrome is equal to length of the string minus one. For example, consider the string “abc”. The string can be converted to “abcba” by inserting “a” and “b”. The number of insertions is two, which is equal to length minus one.

4. Consider the string “efge”. What is the minimum number of insertions required to make the string a palindrome?

- 0
- 1
- 2
- 3

**Answer:** b

**Explanation:** The string can be converted to “efgfe” by inserting “f” or to “egfge” by inserting “g”. Thus, only one insertion is required.

5. Consider the string “abbccbba”. What is the minimum number of insertions required to make the string a palindrome?

- 0
- 1
- 2
- 3

**Answer:** a

**Explanation:** The given string is already a palindrome. So, no insertions are required.

6. Which of the following problems can be used to solve the minimum number of insertions to form a palindrome problem?

- Minimum number of jumps problem
- Longest common subsequence problem
- Coin change problem
- Knapsack problems

**Answer:** b

**Explanation:** A variation of longest common subsequence can be used to solve the minimum number of insertions to form a palindrome problem.

7. Consider the following dynamic programming implementation:

```
#include<stdio.h>
#include<string.h>
int max(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int min_ins(char *s)
{
    int len = strlen(s), i, j;
    int arr[len + 1][len + 1];
    char rev[len + 1];
    strcpy(rev, s);
    strrev(rev);
    for(i = 0; i <= len; i++)
        for(j = 0; j <= len; j++)
            if(s[i] == rev[j])
                arr[i][j] = arr[i - 1][j - 1];
            else
                arr[i][j] = max(arr[i - 1][j], arr[i][j - 1]) + 1;
}
```

```

        arr[i][0] = 0;
        for(i = 0; i <= len; i++)
            arr[0][i] = 0;
        for(i = 1; i <= len; i++)
        {
            for(j = 1; j <= len; j++)
            {
                if(s[i - 1] == rev[j - 1])
                    arr[i][j] = arr[i - 1][j
- 1] + 1;
                else
                    arr[i][j] = max(arr[i -
1][j], arr[i][j - 1]);
            }
            return _____;
}
int main()
{
    char s[] = "abcda";
    int ans = min_ins(s);
    printf("%d",ans);
    return 0;
}

```

Which of the following lines should be added to complete the code?

- a) arr[len][len]
- b) len + arr[len][len]
- c) len
- d) len – arr[len][len]

**Answer:** d

**Explanation:** arr[len][len] contains the length of the longest palindromic subsequence. So, len – arr[len][len] gives the minimum number of insertions required to form a palindrome.

8. What is the time complexity of the following dynamic programming implementation of the minimum number of insertions to form a palindrome problem?

```

#include<stdio.h>
#include<string.h>
int max(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int min_ins(char *s)
{
    int len = strlen(s), i, j;

```

```

        int arr[len + 1][len + 1];
        char rev[len + 1];
        strcpy(rev, s);
        strrev(rev);
        for(i = 0; i <= len; i++)
            arr[i][0] = 0;
        for(i = 0; i <= len; i++)
            arr[0][i] = 0;
        for(i = 1; i <= len; i++)
        {
            for(j = 1; j <= len; j++)
            {
                if(s[i - 1] == rev[j - 1])
                    arr[i][j] = arr[i - 1][j
- 1] + 1;
                else
                    arr[i][j] = max(arr[i -
1][j], arr[i][j - 1]);
            }
            return len - arr[len][len];
}
int main()
{
    char s[] = "abcda";
    int ans = min_ins(s);
    printf("%d",ans);
    return 0;
}

a) O(1)
b) O(n)
c) O(n2)
d) O(mn)

```

**Answer:** c

**Explanation:** The time complexity of the above dynamic programming implementation is O(n<sup>2</sup>).

9. What is the space complexity of the following dynamic programming implementation of the minimum number of insertions to form a palindrome problem?

```

#include<stdio.h>
#include<string.h>
int max(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int min_ins(char *s)

```

```

{
    int len = strlen(s), i, j;
    int arr[len + 1][len + 1];
    char rev[len + 1];
    strcpy(rev, s);
    strrev(rev);
    for(i = 0; i <= len; i++)
        arr[i][0] = 0;
    for(i = 0; i <= len; i++)
        arr[0][i] = 0;
    for(i = 1; i <= len; i++)
    {
        for(j = 1; j <= len; j++)
        {
            if(s[i - 1] == rev[j - 1])
                arr[i][j] = arr[i - 1][j
- 1] + 1;
            else
                arr[i][j] = max(arr[i -
1][j], arr[i][j - 1]);
        }
        return len - arr[len][len];
    }
}

int main()
{
    char s[] = "abcda";
    int ans = min_ins(s);
    printf("%d", ans);
    return 0;
}

a) O(1)
b) O(n)
c) O(n2)
d) O(mn)

```

**Answer:** c

**Explanation:** The space complexity of the above dynamic programming implementation is  $O(n^2)$ .

10. What is the output of the following code?

```
#include<stdio.h>
#include<string.h>
int max(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int min_ins(char *s)
{
```

```

    int len = strlen(s), i, j;
    int arr[len + 1][len + 1];
    char rev[len + 1];
    strcpy(rev, s);
    strrev(rev);
    for(i = 0; i <= len; i++)
        arr[i][0] = 0;
    for(i = 0; i <= len; i++)
        arr[0][i] = 0;
    for(i = 1; i <= len; i++)
    {
        for(j = 1; j <= len; j++)
        {
            if(s[i - 1] == rev[j - 1])
                arr[i][j] = arr[i - 1][j
- 1] + 1;
            else
                arr[i][j] = max(arr[i -
1][j], arr[i][j - 1]);
        }
        return len - arr[len][len];
    }
}

int main()
{
    char s[] = "abcda";
    int ans = min_ins(s);
    printf("%d", ans);
    return 0;
}
```

a) 1

b) 2

c) 3

d) 4

**Answer:** b

**Explanation:** The length of the longest palindromic subsequence is 3. So, the output will be  $5 - 3 = 2$ .

11. What is the value stored in  $\text{arr}[2][4]$  when the following code is executed?

```
#include<stdio.h>
#include<string.h>
int max(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int min_ins(char *s)
{
    int len = strlen(s), i, j;
```

```

int arr[len + 1][len + 1];
char rev[len + 1];
strcpy(rev, s);
strrev(rev);
for(i = 0; i <= len; i++)
    arr[i][0] = 0;
for(i = 0; i <= len; i++)
    arr[0][i] = 0;
for(i = 1; i <= len; i++)
{
    for(j = 1; j <= len; j++)
    {
        if(s[i - 1] == rev[j - 1])
            arr[i][j] = arr[i - 1][j
- 1] + 1;
        else
            arr[i][j] = max(arr[i -
1][j], arr[i][j - 1]);
    }
    return len - arr[len][len];
}
int main()
{
    char s[] = "abcda";
    int ans = min_ins(s);
    printf("%d", ans);
    return 0;
}

```

- a) 2
- b) 3
- c) 4
- d) 5

**Answer:** a

**Explanation:** The value stored in arr[2][4] when the above code is executed is 2.

12. What is the output of the following code?

```

#include<stdio.h>
#include<string.h>
int max(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int min_ins(char *s)
{
    int len = strlen(s), i, j;
    int arr[len + 1][len + 1];
    char rev[len + 1];
    strcpy(rev, s);

```

```

    strrev(rev);
    for(i = 0; i <= len; i++)
        arr[i][0] = 0;
    for(i = 0; i <= len; i++)
        arr[0][i] = 0;
    for(i = 1; i <= len; i++)
    {
        for(j = 1; j <= len; j++)
        {
            if(s[i - 1] == rev[j - 1])
                arr[i][j] = arr[i - 1][j
- 1] + 1;
            else
                arr[i][j] = max(arr[i -
1][j], arr[i][j - 1]);
        }
    }
    return len - arr[len][len];
}
int main()
{
    char s[] = "efgfe";
    int ans = min_ins(s);
    printf("%d", ans);
    return 0;
}

```

- a) 0
- b) 2
- c) 4
- d) 6

**Answer:** a

**Explanation:** Since the string “efgfe” is already a palindrome, the number of insertions required is 0.

1. Given a 2D matrix, find a submatrix that has the maximum sum. Which of the following methods can be used to solve this problem?

- a) Brute force
- b) Recursion
- c) Dynamic programming
- d) Brute force, Recursion, Dynamic programming

**Answer:** d

**Explanation:** Brute force, Recursion and Dynamic programming can be used to find the submatrix that has the maximum sum.

2. In which of the following cases, the maximum sum rectangle is the 2D matrix itself?
- When all the elements are negative
  - When all the elements are positive
  - When some elements are positive and some negative
  - When diagonal elements are positive and rest are negative

**Answer:** a

**Explanation:** When all the elements of a matrix are positive, the maximum sum rectangle is the 2D matrix itself.

3. Consider the following statements and select which of the following statement are true.

Statement 1: The maximum sum rectangle can be 1X1 matrix containing the largest element If the matrix size is 1X1

Statement 2: The maximum sum rectangle can be 1X1 matrix containing the largest element If all the elements are zero

Statement 3: The maximum sum rectangle can be 1X1 matrix containing the largest element If all the elements are negative

- Only statement 1 is correct
- Only statement 1 and Statement 2 are correct
- Only statement 1 and Statement 3 are correct
- Statement 1, Statement 2 and Statement 3 are correct

**Answer:** d

**Explanation:** If the matrix size is  $1 \times 1$  then the element itself is the maximum sum of that  $1 \times 1$  matrix. If all elements are zero, then the sum of any submatrix of the given matrix is zero. If all elements are negative, then the maximum element in that matrix is the highest sum in that matrix which is again  $1 \times 1$  submatrix of the given matrix. Hence all three statements are correct.

4. Consider a matrix in which all the elements are non-zero(at least one positive and at least

one negative element). In this case, the sum of the elements of the maximum sum rectangle cannot be zero.

- True
- False

**Answer:** a

**Explanation:** If a matrix contains all non-zero elements with at least one positive and at least one negative element, then the sum of elements of the maximum sum rectangle cannot be zero.

5. Consider the  $2 \times 3$  matrix  $\{\{1,2,3\},\{1,2,3\}\}$ . What is the sum of elements of the maximum sum rectangle?

- 3
- 6
- 12
- 18

**Answer:** c

**Explanation:** Since all the elements of the  $2 \times 3$  matrix are positive, the maximum sum rectangle is the matrix itself and the sum of elements is 12.

6. Consider the  $2 \times 2$  matrix  $\{\{-1,-2\},\{-3,-4\}\}$ . What is the sum of elements of the maximum sum rectangle?

- 0
- 1
- 7
- 12

**Answer:** b

**Explanation:** Since all the elements of the  $2 \times 2$  matrix are negative, the maximum sum rectangle is  $\{-1\}$ , a  $1 \times 1$  matrix containing the largest element. The sum of elements of the maximum sum rectangle is -1.

7. Consider the  $3 \times 3$  matrix  $\{\{2,1,-3\},\{6,3,4\},\{-2,3,0\}\}$ . What is the sum of the elements of the maximum sum rectangle?

- 13
- 16

- c) 14  
d) 19

**Answer:** c

**Explanation:** The complete matrix represents the maximum sum rectangle and its sum is 14.

8. What is the time complexity of the brute force implementation of the maximum sum rectangle problem?

- a)  $O(n)$   
b)  $O(n^2)$   
c)  $O(n^3)$   
d)  $O(n^4)$

**Answer:** d

**Explanation:** The time complexity of the brute force implementation of the maximum sum rectangle problem is  $O(n^4)$ .

9. The dynamic programming implementation of the maximum sum rectangle problem uses which of the following algorithm?

- a) Hirschberg's algorithm  
b) Needleman-Wunsch algorithm  
c) Kadane's algorithm  
d) Wagner Fischer algorithm

**Answer:** c

**Explanation:** The dynamic programming implementation of the maximum sum rectangle problem uses Kadane's algorithm.

10. Consider the following code snippet:

```
int max_sum_rectangle(int arr[][3],int ro
w,int col)
{
    int left, right, tmp[row], mx_sm =
    INT_MIN, idx, val;
    for(left = 0; left < col; left++)
    {
        for(right = left; right < col;
right++)
        {
            if(right == left)
            {
                for(idx = 0; idx < row
; idx++)

```

```
        tmp[idx] = arr[idx][
right];
    }
    else
    {
        for(idx = 0; idx < row
; idx++)
            tmp[idx] += arr[idx
][right];
    }
    val = kadane_algo(tmp, row)
;
    if(val > mx_sm)
        _____;
}
return mx_sm;
}
```

Which of the following lines should be inserted to complete the above code?

- a) val = mx\_sm  
b) return val  
c) mx\_sm = val  
d) return mx\_sm

**Answer:** c

**Explanation:** The line "mx\_sm = val" should be inserted to complete the above code.

11. What is the time complexity of the following dynamic programming implementation of the maximum sum rectangle problem?

```
int max_sum_rectangle(int arr[][3],int ro
w,int col)
{
    int left, right, tmp[row], mx_sm =
    INT_MIN, idx, val;
    for(left = 0; left < col; left++)
    {
        for(right = left; right < col;
right++)
        {
            if(right == left)
            {
                for(idx = 0; idx < row
; idx++)
                    tmp[idx] = arr[idx][
right];
            }
            else
            {
```

```

        for(idx = 0; idx < row ; val = kadane_algo(tmp, row)
; idx++)
            tmp[idx] += arr[idx] ;
        }
    val = kadane_algo(tmp, row) if(val > mx_sm)
; if(val > mx_sm) mx_sm = val;
}
return mx_sm;
}

a) O(row*col)
b) O(row)
c) O(col)
d) O(row*col*col)

```

**Answer:** d

**Explanation:** The time complexity of the above dynamic programming implementation of the maximum sum rectangle is  $O(\text{row}^*\text{col}^*\text{col})$ .

12. What is the space complexity of the following dynamic programming implementation of the maximum sum rectangle problem?

```

int max_sum_rectangle(int arr[][3],int ro
w,int col)
{
    int left, right, tmp[row], mx_sm =
INT_MIN, idx, val;
    for(left = 0; left < col; left++)
    {
        for(right = left; right < col;
right++)
        {
            if(right == left)
            {
                for(idx = 0; idx < row
; idx++)
                    tmp[idx] = arr[idx][
right];
            }
            else
            {
                for(idx = 0; idx < row
; idx++)
                    tmp[idx] += arr[idx][
right];
            }
        }
    }
}
```

- a) O( $\text{row}^*\text{col}$ )
- b) O( $\text{row}$ )
- c) O( $\text{col}$ )
- d) O( $\text{row}^*\text{col}^*\text{col}$ )

**Answer:** b

**Explanation:** The space complexity of the above dynamic programming implementation of the maximum sum rectangle problem is  $O(\text{row})$ .

13. What is the output of the following code?

```

#include<stdio.h>
#include<limits.h>
int max_num(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int kadane_algo(int *arr, int len)
{
    int ans, sum, idx;
    ans = 0;
    sum = 0;
    for(idx = 0; idx < len; idx++)
    {
        sum = max_num(0,sum + arr[idx])
;
        ans = max_num(sum,ans);
    }
    return ans;
}
int max_sum_rectangle(int arr[][5],int ro
w,int col)
{
    int left, right, tmp[row], mx_sm =
INT_MIN, idx, val;
    for(left = 0; left < col; left++)
    {
        for(right = left; right < col;
right++)
        {
            if(right == left)
            {

```

```

w; idx++)
        for(idx = 0; idx < ro
            tmp[idx] = arr[idx][
right];
    }
    else
    {
        for(idx = 0; idx < r
            tmp[idx] += arr[id
x][right];
    }
    val = kadane_algo(tmp,ro
);
    if(val > mx_sm)
        mx_sm = val;
}
return mx_sm;
}
int main()
{
    int arr[2][5] ={{1, 2, -1, -4, -20
}, {-4, -1, 1, 7, -6}};
    int row = 2, col = 5;
    int ans = max_sum_rectangle(arr,ro
w,col);
    printf("%d",ans);
    return 0;
}

a) 7
b) 8
c) 9
d) 10

```

**Answer:** b

**Explanation:** The program prints the sum of elements of the maximum sum rectangle, which is 8.

14. What is the output of the following code?

```

#include<stdio.h>
#include<limits.h>
int max_num(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int kadane_algo(int *arr, int len)
{
    int ans, sum, idx;
    ans =0;
        sum =0;
        for(idx =0; idx < len; idx++)
    {
        sum = max_num(0,sum + arr[idx])
    }
    ans = max_num(sum,ans);
}
return ans;
}
int max_sum_rectangle(int arr[][5],int ro
w,int col)
{
    int left, right, tmp[row], mx_sm =
INT_MIN, idx, val=0;
    for(left = 0; left < col; left++)
    {
        for(right = left; right < col;
right++)
        {
            if(right == left)
            {
                for(idx = 0; idx < row
; idx++)
                    tmp[idx] = arr[idx][
right];
            }
            else
            {
                for(idx = 0; idx < row
; idx++)
                    tmp[idx] += arr[idx][
right];
            }
            val = kadane_algo(tmp,row)
;
            if(val > mx_sm)
                mx_sm = val;
        }
    }
    return mx_sm;
}
int main()
{
    int arr[4][5] ={{ 7,  1, -3, -6, -1
5},
                    { 10, -6,  3,  -4,
11},
                    { -2, -3, -1,  2, -5}
,
                    { 3,  0,  1,  0,  3}}
;
    int row = 4, col = 5;
    int ans = max_sum_rectangle(arr,ro
w,col);
    printf("%d",ans);
    return 0;
}

```

- a) 17
- b) 18
- c) 19
- d) 20

**Answer:** b

**Explanation:** The program prints the sum of elements of the maximum sum rectangle, which is 18.

1. Given an array, check if the array can be divided into two subsets such that the sum of elements of the two subsets is equal. This is the balanced partition problem. Which of the following methods can be used to solve the balanced partition problem?

- a) Dynamic programming
- b) Recursion
- c) Brute force
- d) Dynamic programming, Recursion, Brute force

**Answer:** d

**Explanation:** All of the mentioned methods can be used to solve the balanced partition problem.

2. In which of the following cases, it is not possible to have two subsets with equal sum?

- a) When the number of elements is odd
- b) When the number of elements is even
- c) When the sum of elements is odd
- d) When the sum of elements is even

**Answer:** c

**Explanation:** When the sum of all the elements is odd, it is not possible to have two subsets with equal sum.

3. What is the time complexity of the brute force algorithm used to solve the balanced partition problem?

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) O( $2^n$ )

**Answer:** d

**Explanation:** In the brute force implementation, all the possible subsets will be formed. This takes exponential time.

4. Consider a variation of the balanced partition problem in which we find two subsets such that  $|S_1 - S_2|$  is minimum. Consider the array {1, 2, 3, 4, 5}. Which of the following pairs of subsets is an optimal solution for the above problem?

- a) {5, 4} & {3, 2, 1}
- b) {5} & {4, 3, 2, 1}
- c) {4, 2} & {5, 3, 1}
- d) {5, 3} & {4, 2, 1}

**Answer:** d

**Explanation:** For  $S_1 = \{5, 3\}$  and  $S_2 = \{4, 2, 1\}$ ,  $\text{sum}(S_1) - \text{sum}(S_2) = 1$ , which is the optimal solution.

5. Consider the following code:

```
#include<stdio.h>
int balanced_partition(int *arr, int len)
{
    int sm = 0, i, j;
    for(i = 0; i < len; i++)
        sm += arr[i];
    if(sm % 2 != 0)
        return 0;
    int ans[sm/2 + 1][len + 1];
    for(i = 0; i <= len; i++)
        ans[0][i] = 1;
    for(i = 1; i <= sm/2; i++)
        ans[i][0] = 0;
    for(i = 1; i <= sm/2; i++)
    {
        for(j = 1; j <= len; j++)
        {
            ans[i][j] = ans[i][j-1];
            if(i >= arr[j - 1])
                ans[i][j] = _____
        }
    }
    return ans[sm/2][len];
}
int main()
{
    int arr[] = {3, 4, 5, 6, 7, 1}, len
    = 6;
    int ans = balanced_partition(arr, len)
```

```

);
if(ans == 0)
    printf("false");
else
    printf("true");
return 0;
}

```

Which of the following lines should be inserted to complete the above code?

- a) ans[i - arr[j - 1]][j - 1]
- b) ans[i][j]
- c) ans[i][j] || ans[i - arr[j - 1]][j - 1]
- d) ans[i][j] && ans[i - arr[j - 1]][j - 1]

**Answer:** c

**Explanation:** The line “ans[i][j] || ans[i - arr[j - 1]][j - 1]” completes the above code.

6. What is the time complexity of the following dynamic programming implementation of the balanced partition problem where “n” is the number of elements and “sum” is their sum?

```

#include<stdio.h>
int balanced_partition(int *arr, int len)
{
    int sm = 0, i, j;
    for(i = 0; i < len; i++)
        sm += arr[i];
    if(sm % 2 != 0)
        return 0;
    int ans[sm/2 + 1][len + 1];
    for(i = 0; i <= len; i++)
        ans[0][i] = 1;
    for(i = 1; i <= sm/2; i++)
        ans[i][0] = 0;
    for(i = 1; i <= sm/2; i++)
    {
        for(j = 1; j <= len; j++)
        {
            ans[i][j] = ans[i][j-1];
            if(i >= arr[j - 1])
                ans[i][j] = ans[i][j] ||
ans[i - arr[j - 1]][j - 1];
        }
    }
    return ans[sm/2][len];
}
int main()
{
    int arr[] = {3, 4, 5, 6, 7, 1}, len
= 6;
}

```

```

int ans = balanced_partition(arr, len
);
if(ans == 0)
    printf("false");
else
    printf("true");
return 0;
}

```

- a) O(sum)
- b) O(n)
- c) O(sum \* n)
- d) O(sum + n)

**Answer:** c

**Explanation:** The time complexity of the above dynamic programming implementation of the balanced partition problem is O(sum \* n).

7. What is the space complexity of the following dynamic programming implementation of the balanced partition problem?

```

#include<stdio.h>
int balanced_partition(int *arr, int len)
{
    int sm = 0, i, j;
    for(i = 0; i < len; i++)
        sm += arr[i];
    if(sm % 2 != 0)
        return 0;
    int ans[sm/2 + 1][len + 1];
    for(i = 0; i <= len; i++)
        ans[0][i] = 1;
    for(i = 1; i <= sm/2; i++)
        ans[i][0] = 0;
    for(i = 1; i <= sm/2; i++)
    {
        for(j = 1; j <= len; j++)
        {
            ans[i][j] = ans[i][j-1];
            if(i >= arr[j - 1])
                ans[i][j] = ans[i][j] ||
ans[i - arr[j - 1]][j - 1];
        }
    }
    return ans[sm/2][len];
}
int main()
{
    int arr[] = {3, 4, 5, 6, 7, 1}, len
= 6;
}

```

```

        int ans = balanced_partition(arr, len
);
        if(ans == 0)
            printf("false");
        else
            printf("true");
        return 0;
    }
}

```

- a) O(sum)
- b) O(n)
- c) O(sum \* n)
- d) O(sum + n)

**Answer:** c

**Explanation:** The space complexity of the above dynamic programming implementation of the balanced partition problem is  $O(\text{sum} * n)$ .

8. What is the output of the following code?

```

#include<stdio.h>
int balanced_partition(int *arr, int len)
{
    int sm = 0, i, j;
    for(i = 0; i < len; i++)
        sm += arr[i];
    if(sm % 2 != 0)
        return 0;
    int ans[sm/2 + 1][len + 1];
    for(i = 0; i <= len; i++)
        ans[0][i] = 1;
    for(i = 1; i <= sm/2; i++)
        ans[i][0] = 0;
    for(i = 1; i <= sm/2; i++)
    {
        for(j = 1; j <= len; j++)
        {
            ans[i][j] = ans[i][j-1];
            if(i >= arr[j - 1])
                ans[i][j] = ans[i][j] ||
ans[i - arr[j - 1]][j - 1];
        }
        return ans[sm/2][len];
    }
    int main()
    {
        int arr[] = {3, 4, 5, 6, 7, 1}, len
= 6;
        int ans = balanced_partition(arr,le
n);
        if(ans == 0)
            printf("false");
    }
}

```

```

        else
            printf("true");
        return 0;
    }
}

```

- a) True
- b) False

**Answer:** a

**Explanation:** The partitions are  $S_1 = \{6, 7\}$  and  $S_2 = \{1, 3, 4, 5\}$  and the sum of each partition is 13. So, the array can be divided into balanced partitions.

9. What is the value stored in  $\text{ans}[3][3]$  when the following code is executed?

```

#include<stdio.h>
int balanced_partition(int *arr, int len)
{
    int sm = 0, i, j;
    for(i = 0; i < len; i++)
        sm += arr[i];
    if(sm % 2 != 0)
        return 0;
    int ans[sm/2 + 1][len + 1];
    for(i = 0; i <= len; i++)
        ans[0][i] = 1;
    for(i = 1; i <= sm/2; i++)
        ans[i][0] = 0;
    for(i = 1; i <= sm/2; i++)
    {
        for(j = 1; j <= len; j++)
        {
            ans[i][j] = ans[i][j-1];
            if(i >= arr[j - 1])
                ans[i][j] = ans[i][j] ||
ans[i - arr[j - 1]][j - 1];
        }
    }
    return ans[sm/2][len];
}
int main()
{
    int arr[] = {3, 4, 5, 6, 7, 1}, len
= 6;
    int ans = balanced_partition(arr, len
);
    if(ans == 0)
        printf("false");
    else
        printf("true");
    return 0;
}

```

- a) 0
- b) 1
- c) -1
- d) -2

**Answer:** b

**Explanation:** The value stored in  $\text{ans}[3][3]$  indicates if a sum of 3 can be obtained using a subset of the first 3 elements. Since the sum can be obtained the value stored is 1.

10. What is the sum of each of the balanced partitions for the array {5, 6, 7, 10, 3, 1}?

- a) 16
- b) 32
- c) 0
- d) 64

**Answer:** a

**Explanation:** The sum of all the elements of the array is 32. So, the sum of all the elements of each partition should be 16.

11. What is the output of the following code?

```
#include<stdio.h>
int balanced_partition(int *arr, int len)
{
    int sm = 0, i, j;
    for(i = 0; i < len; i++)
        sm += arr[i];
    if(sm % 2 != 0)
        return 0;
    int ans[sm/2 + 1][len + 1];
    for(i = 0; i <= len; i++)
        ans[0][i] = 1;
    for(i = 1; i <= sm/2; i++)
        ans[i][0] = 0;
    for(i = 1; i <= sm/2; i++)
    {
        for(j = 1; j <= len; j++)
        {
            ans[i][j] = ans[i][j-1];
            if(i >= arr[j - 1])
                ans[i][j] = ans[i][j] ||
                ans[i - arr[j - 1]][j - 1];
        }
    }
    return ans[sm/2][len];
}
int main()
{
    int arr[] = {5, 6, 7, 10, 3, 1}, len
```

```
= 6;
    int ans = balanced_partition(arr, len
);
    if(ans == 0)
        printf("false");
    else
        printf("true");
    return 0;
}
```

- a) True
- b) False

**Answer:** a

**Explanation:** The array can be divided into two partitions  $S1 = \{10, 6\}$  and  $S2 = \{5, 7, 3, 1\}$  and the sum of all the elements of each partition is 16. So, the answer is true.

12. What is the output of the following code?

```
#include<stdio.h>
int balanced_partition(int *arr, int len)
{
    int sm = 0, i, j;
    for(i = 0; i < len; i++)
        sm += arr[i];
    if(sm % 2 != 0)
        return 0;
    int ans[sm/2 + 1][len + 1];
    for(i = 0; i <= len; i++)
        ans[0][i] = 1;
    for(i = 1; i <= sm/2; i++)
        ans[i][0] = 0;
    for(i = 1; i <= sm/2; i++)
    {
        for(j = 1; j <= len; j++)
        {
            ans[i][j] = ans[i][j-1];
            if(i >= arr[j - 1])
                ans[i][j] = ans[i][j] ||
                ans[i - arr[j - 1]][j - 1];
        }
    }
    return ans[sm/2][len];
}
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8,
9}, len = 9;
    int ans = balanced_partition(arr, len
);
    if(ans == 0)
        printf("false");
    else
```

```

    printf("true");
    return 0;
}

```

- a) True  
b) False

**Answer:** b

**Explanation:** Since the sum of all the elements of the array is 45, the array cannot be divided into two partitions of equal sum and the answer is false.

---

1. You are given n dice each having f faces. You have to find the number of ways in which a sum of S can be achieved. This is the dice throw problem. Which of the following methods can be used to solve the dice throw problem?

- a) Brute force  
b) Recursion  
c) Dynamic programming  
d) Brute force, Recursion and Dynamic Programming

**Answer:** d

**Explanation:** Brute force, Recursion and Dynamic Programming can be used to solve the dice throw problem.

2. You have n dice each having f faces. What is the number of permutations that can be obtained when you roll the n dice together?

- a)  $n \cdot n \cdot n \dots f$  times  
b)  $f \cdot f \cdot f \dots n$  times  
c)  $n \cdot n \cdot n \dots n$  times  
d)  $f \cdot f \cdot f \dots f$  times

**Answer:** b

**Explanation:** Each die can take f values and there are n dice. So, the total number of permutations is  $f \cdot f \cdot f \dots n$  times.

3. You have 3 dice each having 6 faces. What is the number of permutations that can be obtained when you roll the 3 dice together?

- a) 27  
b) 36

- c) 216  
d) 81

**Answer:** c

**Explanation:** The total number of permutations that can be obtained is  $6 \cdot 6 \cdot 6 = 216$ .

4. You have 2 dice each of them having 6 faces numbered from 1 to 6. What is the number of ways in which a sum of 11 can be achieved?

- a) 0  
b) 1  
c) 2  
d) 3

**Answer:** c

**Explanation:** The sum of 11 can be achieved when the dice show {6, 5} or {5, 6}.

5. There are n dice with f faces. The faces are numbered from 1 to f. What is the minimum possible sum that can be obtained when the n dice are rolled together?

- a) 1  
b) f  
c) n  
d)  $n \cdot f$

**Answer:** c

**Explanation:** The sum will be minimum when all the faces show a value 1. The sum in this case will be n.

6. There are n dice with f faces. The faces are numbered from 1 to f. What is the maximum possible sum that can be obtained when the n dice are rolled together?

- a) 1  
b)  $f \cdot f$   
c)  $n \cdot n$   
d)  $n \cdot f$

**Answer:** d

**Explanation:** The sum will be maximum when all the faces show a value f. The sum in this case will be  $n \cdot f$ .

7. There are 10 dice having 5 faces. The faces are numbered from 1 to 5. What is the number of ways in which a sum of 4 can be achieved?

- a) 0
- b) 2
- c) 4
- d) 8

**Answer:** a

**Explanation:** Since there are 10 dice and the minimum value each die can take is 1, the minimum possible sum is 10. Hence, a sum of 4 cannot be achieved.

8. Consider the following dynamic programming implementation of the dice throw problem:

```
#include<stdio.h>
int get_ways(int num_of_dice, int num_of_faces, int S)
{
    int arr[num_of_dice + 1][S + 1];
    int dice, face, sm;
    for(dice = 0; dice <= num_of_dice; dice++)
        for(sm = 0; sm <= S; sm++)
            arr[dice][sm] = 0;
    for(sm = 1; sm <= S; sm++)
        arr[1][sm] = 1;
    for(dice = 2; dice <= num_of_dice; dice++)
    {
        for(sm = 1; sm <= S; sm++)
        {
            for(face = 1; face <= num_of_faces && face < sm; face++)
                arr[dice][sm] += arr[dice - 1][sm - face];
        }
        return _____;
    }
    int main()
    {
        int num_of_dice = 3, num_of_faces = 4, sum = 6;
        int ans = get_ways(num_of_dice, num_of_faces, sum);
        printf("%d",ans);
        return 0;
    }
}
```

Which of the following lines should be added to complete the above code?

- a) arr[num\_of\_dice][S]
- b) arr[dice][sm]
- c) arr[dice][S]
- d) arr[S][dice]

**Answer:** a

**Explanation:** The line arr[num\_of\_dice][S] completes the above code.

9. What is time complexity of the following dynamic programming implementation of the dice throw problem where f is the number of faces, n is the number of dice and s is the sum to be found?

```
#include<stdio.h>
int get_ways(int num_of_dice, int num_of_faces, int S)
{
    int arr[num_of_dice + 1][S + 1];
    int dice, face, sm;
    for(dice = 0; dice <= num_of_dice; dice++)
        for(sm = 0; sm <= S; sm++)
            arr[dice][sm] = 0;
    for(sm = 1; sm <= S; sm++)
        arr[1][sm] = 1;
    for(dice = 2; dice <= num_of_dice; dice++)
    {
        for(sm = 1; sm <= S; sm++)
        {
            for(face = 1; face <= num_of_faces && face < sm; face++)
                arr[dice][sm] += arr[dice - 1][sm - face];
        }
        return arr[num_of_dice][S];
    }
    int main()
    {
        int num_of_dice = 3, num_of_faces = 4, sum = 6;
        int ans = get_ways(num_of_dice, num_of_faces, sum);
        printf("%d",ans);
        return 0;
    }
}
```

- a) O(n\*f)
- b) O(f\*s)

- c)  $O(n^*s)$
- d)  $O(n^*f^*s)$

**Answer:** d

**Explanation:** The time complexity of the above dynamic programming implementation is  $O(n^*f^*s)$ .

10. What is space complexity of the following dynamic programming implementation of the dice throw problem where f is the number of faces, n is the number of dice and s is the sum to be found?

```
#include<stdio.h>
int get_ways(int num_of_dice, int num_of_faces, int S)
{
    int arr[num_of_dice + 1][S + 1];
    int dice, face, sm;
    for(dice = 0; dice <= num_of_dice; dice++)
        for(sm = 0; sm <= S; sm++)
            arr[dice][sm] = 0;
    for(sm = 1; sm <= S; sm++)
        arr[1][sm] = 1;
    for(dice = 2; dice <= num_of_dice; dice++)
    {
        for(sm = 1; sm <= S; sm++)
        {
            for(face = 1; face <= num_of_faces && face < sm; face++)
                arr[dice][sm] += arr[dice - 1][sm - face];
        }
        return arr[num_of_dice][S];
    }
    int main()
    {
        int num_of_dice = 3, num_of_faces = 4, sum = 6;
        int ans = get_ways(num_of_dice, num_of_faces, sum);
        printf("%d", ans);
        return 0;
    }
}
a) O(n*f)
b) O(f*s)
c) O(n*s)
d) O(n*f*s)
```

**Answer:** c

**Explanation:** The space complexity of the above dynamic programming implementation is  $O(n^*s)$ .

11. What is the output of the following code?

```
#include<stdio.h>
int get_ways(int num_of_dice, int num_of_faces, int S)
{
    int arr[num_of_dice + 1][S + 1];
    int dice, face, sm;
    for(dice = 0; dice <= num_of_dice; dice++)
        for(sm = 0; sm <= S; sm++)
            arr[dice][sm] = 0;
    for(sm = 1; sm <= S; sm++)
        arr[1][sm] = 1;
    for(dice = 2; dice <= num_of_dice; dice++)
    {
        for(sm = 1; sm <= S; sm++)
        {
            for(face = 1; face <= num_of_faces && face < sm; face++)
                arr[dice][sm] += arr[dice - 1][sm - face];
        }
        return arr[num_of_dice][S];
    }
    int main()
    {
        int num_of_dice = 3, num_of_faces = 4, sum = 6;
        int ans = get_ways(num_of_dice, num_of_faces, sum);
        printf("%d", ans);
        return 0;
    }
}
```

- a) 10
- b) 12
- c) 14
- d) 16

**Answer:** a

**Explanation:** The output of the above code is 10.

12. What will be the value stored in  $arr[2][2]$  when the following code is executed?

```
#include<stdio.h>
int get_ways(int num_of_dice, int num_of_faces, int S)
{
    int arr[num_of_dice + 1][S + 1];
    int dice, face, sm;
    for(dice = 0; dice <= num_of_dice; dice++)
        for(sm = 0; sm <= S; sm++)
            arr[dice][sm] = 0;
    for(sm = 1; sm <= S; sm++)
        arr[1][sm] = 1;
    for(dice = 2; dice <= num_of_dice; dice++)
    {
        for(sm = 1; sm <= S; sm++)
        {
            for(face = 1; face <= num_of_faces && face < sm; face++)
                arr[dice][sm] += arr[dice - 1][sm - face];
        }
        return arr[num_of_dice][S];
    }
int main()
{
    int num_of_dice = 3, num_of_faces = 4, sum = 6;
    int ans = get_ways(num_of_dice, num_of_faces, sum);
    printf("%d", ans);
    return 0;
}
```

- a) 0
- b) 1
- c) 2
- d) 3

**Answer:** b

**Explanation:** The value stored in arr[2][2] is 1.

13. What is the output of the following code?

```
#include<stdio.h>
int get_ways(int num_of_dice, int num_of_faces, int S)
{
    int arr[num_of_dice + 1][S + 1];
    int dice, face, sm;
    for(dice = 0; dice <= num_of_dice; dice++)
        for(sm = 0; sm <= S; sm++)
            arr[dice][sm] = 0;
```

```
for(sm = 1; sm <= S; sm++)
    arr[1][sm] = 1;
for(dice = 2; dice <= num_of_dice; dice++)
{
    for(sm = 1; sm <= S; sm++)
    {
        for(face = 1; face <= num_of_faces && face < sm; face++)
            arr[dice][sm] += arr[dice - 1][sm - face];
    }
}
return arr[num_of_dice][S];
}

int main()
{
    int num_of_dice = 4, num_of_faces = 6, sum = 3;
    int ans = get_ways(num_of_dice, num_of_faces, sum);
    printf("%d", ans);
    return 0;
}
```

- a) 0
- b) 1
- c) 2
- d) 3

**Answer:** a

**Explanation:** The minimum possible sum is 4. So, the output for sum = 3 is 0.

14. What is the output of the following code?

```
#include<stdio.h>
int get_ways(int num_of_dice, int num_of_faces, int S)
{
    int arr[num_of_dice + 1][S + 1];
    int dice, face, sm;
    for(dice = 0; dice <= num_of_dice; dice++)
        for(sm = 0; sm <= S; sm++)
            arr[dice][sm] = 0;
    for(sm = 1; sm <= S; sm++)
        arr[1][sm] = 1;
    for(dice = 2; dice <= num_of_dice; dice++)
    {
        for(sm = 1; sm <= S; sm++)
        {
            for(face = 1; face <= num_of_faces && face < sm; face++)
                arr[dice][sm] += arr[dice - 1][sm - face];
        }
    }
}
```

```

f_faces && face < sm; face++)
    arr[dice][sm] += arr[dic
e - 1][sm - face];
}
return arr[num_of_dice][S];
}
int main()
{
    int num_of_dice = 2, num_of_faces =
6, sum = 5;
    int ans = get_ways(num_of_dice, num
_of_faces, sum);
    printf("%d",ans);
    return 0;
}

a) 2
b) 3
c) 4
d) 5

```

**Answer:** c**Explanation:** The output of the above code is 4.

1. You are given a boolean expression which consists of operators  $\&$ ,  $|$  and  $\wedge$  (AND, OR and XOR) and symbols T or F (true or false). You have to find the number of ways in which the symbols can be parenthesized so that the expression evaluates to true. This is the boolean parenthesization problem. Which of the following methods can be used to solve the problem?

- a) Dynamic programming
- b) Recursion
- c) Brute force
- d) Dynamic programming, Recursion and Brute force

**Answer:** d**Explanation:** Dynamic programming, Recursion and Brute force can be used to solve the Boolean parenthesization problem.

2. Consider the expression  $T \& F | T$ . What is the number of ways in which the expression can be parenthesized so that the output is T (true)?

- a) 0
- b) 1
- c) 2
- d) 3

**Answer:** c**Explanation:** The expression can be parenthesized as  $T \& (F | T)$  and  $(T \& F) | T$  so that the output is T.

3. Consider the expression  $T \& F \wedge T$ . What is the number of ways in which the expression can be parenthesized so that the output is T (true)?

- a) 0
- b) 1
- c) 2
- d) 3

**Answer:** c**Explanation:** The expression can be parenthesized as  $(T \& F) \wedge T$  or  $T \& (F \wedge T)$ , so that the output is T.

4. Consider the expression  $T | F \wedge T$ . In how many ways can the expression be parenthesized so that the output is F (false)?

- a) 0
- b) 1
- c) 2
- d) 3

**Answer:** b**Explanation:** The expression can be parenthesized as  $(T | F) \wedge T$ , so that the output is F (false).

5. Which of the following gives the total number of ways of parenthesizing an expression with  $n + 1$  terms?

- a)  $n$  factorial
- b)  $n$  square
- c)  $n$  cube
- d)  $n$ th catalan number

**Answer:** d**Explanation:** The  $n$ th Catalan number gives

the total number of ways of parenthesizing an expression with  $n + 1$  terms.

6. What is the maximum number of ways in which a boolean expression with  $n + 1$  terms can be parenthesized, such that the output is true?

- a) nth catalan number
- b) n factorial
- c) n cube
- d) n square

**Answer:** a

**Explanation:** The number of ways will be maximum when all the possible parenthesizations result in a true value. The number of possible parenthesizations is given by the nth catalan number.

7. Consider the following dynamic programming implementation of the boolean parenthesization problem:

```
int count_bool_parenthesization(char *sym
, char *op)
{
    int str_len = strlen(sym);
    int True[str_len][str_len], False[str_len][str_len];
    int row, col, length, l;
    for(row = 0, col = 0; row < str_len
; row++, col++)
    {
        if(sym[row] == 'T')
        {
            True[row][col] = 1;
            False[row][col] = 0;
        }
        else
        {
            True[row][col] = 0;
            False[row][col] = 1;
        }
    }
    for(length = 1; length < str_len; l
length++)
    {
        for(row = 0, col = length; col
< str_len; col++, row++)
        {
            True[row][col] = 0;
            False[row][col] = 0;
            for(l = 0; l < length; l++)
            {
                if(sym[l] == '|')
                {
                    True[row][col] += True[row][l] * False[l][col];
                    False[row][col] += False[row][l] * True[l][col];
                }
                else if(sym[l] == '&')
                {
                    True[row][col] += True[row][l] * True[l][col];
                    False[row][col] += False[row][l] * False[l][col];
                }
                else if(sym[l] == '^')
                {
                    True[row][col] += True[row][l] * True[l][col];
                    False[row][col] += False[row][l] * False[l][col];
                }
            }
        }
    }
}
```

```
{
    int pos = row + 1;
    int t_row_pos = True[ro
w][pos] + False[row][pos];
    int t_pos_col = True[po
s+1][col] + False[pos+1][col];
    if(op[pos] == '|')
    {
        _____;
    }
    if(op[pos] == '&')
    {
        _____;
    }
    if(op[pos] == '^')
    {
        _____;
    }
}
return True[0][str_len-1];
}
```

Which of the following lines should be added to complete the “if(op[pos] == '|’)” part of the code?

- a) False[row][col] += True[row][pos] \* False[pos+1][col];  
True[row][col] += t\_row\_pos \* t\_pos\_col + False[row][pos] \* False[pos+1][col];
- b) False[row][col] += False[row][pos] \* True[pos+1][col];  
True[row][col] += t\_row\_pos \* t\_pos\_col - True[row][pos] \* True[pos+1][col];
- c) False[row][col] += True[row][pos] \* True[pos+1][col];  
True[row][col] += t\_row\_pos \* t\_pos\_col + True[row][pos] \* True[pos+1][col];
- d) False[row][col] += False[row][pos] \* False[pos+1][col];  
True[row][col] += t\_row\_pos \* t\_pos\_col - False[row][pos] \* False[pos+1][col];

**Answer:** d

**Explanation:** The following lines should be added:

```
False[row][col] += False[row][pos] *
False[pos+1][col];
True[row][col] += t_row_pos * t_pos_col +
False[row][pos] * False[pos+1][col];
```

8. Which of the following lines should be added to complete the “if( $op[k] == '&'$ )” part of the following code?

```
int count_bool_parenthesization(char *sym
, char *op)
{
    int str_len = strlen(sym);
    int True[str_len][str_len], False[st
r_len][str_len];
    int row,col,length,l;
    for(row = 0, col = 0; row < str_len
; row++,col++)
    {
        if(sym[row] == 'T')
        {
            True[row][col] = 1;
            False[row][col] = 0;
        }
        else
        {
            True[row][col] = 0;
            False[row][col] = 1;
        }
    }
    for(length = 1; length < str_len; l
ength++)
    {
        for(row = 0, col = length; col
< str_len; col++, row++)
        {
            True[row][col] = 0;
            False[row][col] = 0;
            for(l = 0; l < length; l++)
            {
                int pos = row + l;
                int t_row_pos = True[ro
w][pos] + False[row][pos];
                int t_pos_col = True[po
s+1][col] + False[pos+1][col];
                if(op[pos] == '|')
                {
                    _____;
                }
                if(op[pos] == '&')
                {
                    _____;
                }
                if(op[pos] == '^')
                {
                    _____;
                }
            }
        }
    }
}
```

```
return True[0][str_len-1];
}

a) True[row][col] += False[row][pos] *
False[pos+1][col];
False[row][col] += t_row_pos * t_pos_col -
True[row][pos] * True[pos+1][col];
b) True[row][col] += True[row][pos] *
True[pos+1][col];
False[row][col] += t_row_pos * t_pos_col -
True[row][pos] * True[pos+1][col];
c) True[row][col] += True[row][pos] *
False[pos+1][col];
False[row][col] += t_row_pos * t_pos_col -
False[row][pos] * True[pos+1][col];
d) True[row][col] += False[row][pos] *
True[pos+1][col];
False[row][col] += t_row_pos * t_pos_col -
False[row][pos] * True[pos+1][col];
```

*Answer:* b

*Explanation:* The following lines should be added:

```
True[row][col] += True[row][pos] *
True[pos+1][col];
False[row][col] += t_row_pos * t_pos_col -
True[row][pos] * True[pos+1][col];
```

9. What is the time complexity of the following dynamic programming implementation of the boolean parenthesization problem?

```
int count_bool_parenthesization(char *sym
, char *op)
{
    int str_len = strlen(sym);
    int True[str_len][str_len], False[st
r_len][str_len];
    int row,col,length,l;
    for(row = 0, col = 0; row < str_len
; row++,col++)
    {
        if(sym[row] == 'T')
        {
            True[row][col] = 1;
            False[row][col] = 0;
        }
        else
        {
            True[row][col] = 0;
```

```

        False[row][col] = 1;
    }
}
for(length = 1; length < str_len; length++)
{
    for(row = 0, col = length; col < str_len; col++, row++)
    {
        True[row][col] = 0;
        False[row][col] = 0;
        for(l = 0; l < length; l++)
        {
            int pos = row + l;
            int t_row_pos = True[row][pos] + False[row][pos];
            int t_pos_col = True[pos+1][col] + False[pos+1][col];
            if(op[pos] == '|')
            {
                False[row][col] += False[row][pos] * False[pos+1][col];
                True[row][col] += t_row_pos * t_pos_col - False[row][pos] * False[pos+1][col];
            }
            if(op[pos] == '&')
            {
                True[row][col] += True[row][pos] * True[pos+1][col];
                False[row][col] += t_row_pos * t_pos_col - True[row][pos] * True[pos+1][col];
            }
            if(op[pos] == '^')
            {
                True[row][col] += True[row][pos] * False[pos+1][col] + False[row][pos] * True[pos+1][col];
                False[row][col] += True[row][pos] * True[pos+1][col] + False[row][pos] * False[pos+1][col];
            }
        }
    }
    return True[0][str_len-1];
}

```

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) O( $n^3$ )

**Answer:** d

**Explanation:** The time complexity of the above dynamic programming implementation is  $O(n^3)$ .

10. What is the space complexity of the following dynamic programming implementation of the boolean parenthesization problem?

```

int count_bool_parenthesization(char *sym
, char *op)
{
    int str_len = strlen(sym);
    int True[str_len][str_len], False[str_len][str_len];
    int row,col,length,l;
    for(row = 0, col = 0; row < str_len ; row++,col++)
    {
        if(sym[row] == 'T')
        {
            True[row][col] = 1;
            False[row][col] = 0;
        }
        else
        {
            True[row][col] = 0;
            False[row][col] = 1;
        }
    }
    for(length = 1; length < str_len; length++)
    {
        for(row = 0, col = length; col < str_len; col++, row++)
        {
            True[row][col] = 0;
            False[row][col] = 0;
            for(l = 0; l < length; l++)
            {
                int pos = row + l;
                int t_row_pos = True[row][pos] + False[row][pos];
                int t_pos_col = True[pos+1][col] + False[pos+1][col];
                if(op[pos] == '|')
                {
                    False[row][col] += False[row][pos] * False[pos+1][col];
                    True[row][col] += t_row_pos * t_pos_col - False[row][pos] * False[pos+1][col];
                }
                if(op[pos] == '&')
                {
                    True[row][col] += True[row][pos] * True[pos+1][col];
                    False[row][col] += t_row_pos * t_pos_col - True[row][pos] * True[pos+1][col];
                }
            }
        }
    }
}

```

```

    {
        True[row][col] += True[row][pos] * True[pos+1][col];
        False[row][col] += t_row_pos * t_pos_col - True[row][pos] * True[pos+1][col];
    }
    if(op[pos] == '^')
    {
        True[row][col] += True[row][pos] * False[pos+1][col] + False[row][pos] * True[pos+1][col];
        False[row][col] += True[row][pos] * True[pos+1][col] + False[row][pos] * False[pos+1][col];
    }
}
return True[0][str_len-1];
}

```

- a) O(1)
- b) O(n)
- c) O( $n^2$ )
- d) O( $n^3$ )

**Answer:** c

**Explanation:** The space complexity of the above dynamic programming implementation is  $O(n^2)$ .

11. What is the output of the following code?

```

#include<stdio.h>
#include<string.h>
int count_bool_parenthesization(char *sym, char *op)
{
    int str_len = strlen(sym);
    int True[str_len][str_len], False[str_len][str_len];
    int row, col, length, l;
    for(row = 0, col = 0; row < str_len; row++, col++)
    {
        if(sym[row] == 'T')
        {
            True[row][col] = 1;
            False[row][col] = 0;
        }
        else
        {
            True[row][col] = 0;
        }
    }
    for(length = 1; length < str_len; length++)
    {
        for(row = 0, col = length; col < str_len; col++, row++)
        {
            True[row][col] = 0;
            False[row][col] = 0;
            for(l = 0; l < length; l++)
            {
                int pos = row + l;
                int t_row_pos = True[row][pos] + False[row][pos];
                int t_pos_col = True[pos+1][col] + False[pos+1][col];
                if(op[pos] == '|')
                {
                    False[row][col] += False[row][pos] * False[pos+1][col];
                    True[row][col] += t_row_pos * t_pos_col - False[row][pos] * False[pos+1][col];
                }
                if(op[pos] == '&')
                {
                    True[row][col] += True[row][pos] * True[pos+1][col];
                    False[row][col] += t_row_pos * t_pos_col - True[row][pos] * True[pos+1][col];
                }
                if(op[pos] == '^')
                {
                    True[row][col] += True[row][pos] * False[pos+1][col] + False[row][pos] * True[pos+1][col];
                    False[row][col] += True[row][pos] * True[pos+1][col] + False[row][pos] * False[pos+1][col];
                }
            }
        }
    }
    return True[0][str_len-1];
}
int main()
{
    char sym[] = "TTTT";
    char op[] = "|^^";
    int ans = count_bool_parenthesization(sym, op);
    printf("%d", ans);
    return 0;
}

```

- a) 1
- b) 2
- c) 3
- d) 4

**Answer:** d

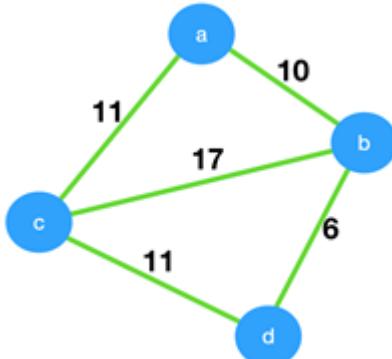
**Explanation:** The output of the above program is 4.

1. Which of the following is true?
- a) Prim's algorithm initialises with a vertex
  - b) Prim's algorithm initialises with a edge
  - c) Prim's algorithm initialises with a vertex which has smallest edge
  - d) Prim's algorithm initialises with a forest

**Answer:** a

**Explanation:** Steps in Prim's algorithm: (I) Select any vertex of given graph and add it to MST (II) Add the edge of minimum weight from a vertex not in MST to the vertex in MST; (III) If MST is complete the stop, otherwise go to step (II).

2. Consider the given graph.



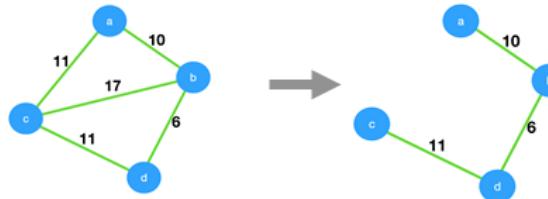
What is the weight of the minimum spanning tree using the Prim's algorithm, starting from vertex a?

- a) 23
- b) 28
- c) 27
- d) 11

**Answer:** c

**Explanation:** In Prim's algorithm, we select a vertex and add it to the MST. Then we add

the minimum edge from the vertex in MST to vertex not in MST. From, figure shown below weight of MST = 27.



3. Worst case is the worst case time complexity of Prim's algorithm if adjacency matrix is used?
- a)  $O(\log V)$
  - b)  $O(V^2)$
  - c)  $O(E^2)$
  - d)  $O(V \log E)$

**Answer:** b

**Explanation:** Use of adjacency matrix provides the simple implementation of the Prim's algorithm. In Prim's algorithm, we need to search for the edge with a minimum for that vertex. So, worst case time complexity will be  $O(V^2)$ , where V is the number of vertices.

4. Prim's algorithm is a \_\_\_\_\_
- a) Divide and conquer algorithm
  - b) Greedy algorithm
  - c) Dynamic Programming
  - d) Approximation algorithm

**Answer:** b

**Explanation:** Prim's algorithm uses a greedy algorithm approach to find the MST of the connected weighted graph. In greedy method, we attempt to find an optimal solution in stages.

5. Prim's algorithm resembles Dijkstra's algorithm.
- a) True
  - b) False

**Answer:** a

**Explanation:** In Prim's algorithm, the MST is

constructed starting from a single vertex and adding in new edges to the MST that link the partial tree to a new vertex outside of the MST. And Dijkstra's algorithm also rely on the similar approach of finding the next closest vertex. So, Prim's algorithm resembles Dijkstra's algorithm.

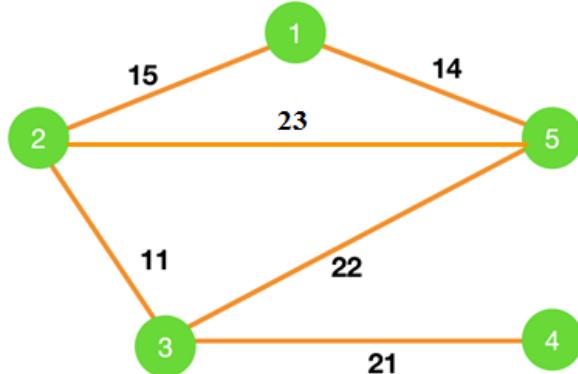
6. Kruskal's algorithm is best suited for the sparse graphs than the prim's algorithm.

- a) True
- b) False

**Answer:** a

**Explanation:** Prim's algorithm and Kruskal's algorithm perform equally in case of the sparse graphs. But Kruskal's algorithm is simpler and easy to work with. So, it is best suited for sparse graphs.

7. Consider the graph shown below.



Which of the following edges form the MST of the given graph using Prim's algorithm, starting from vertex 4.

- a) (4-3)(5-3)(2-3)(1-2)
- b) (4-3)(3-5)(5-1)(1-2)
- c) (4-3)(3-5)(5-2)(1-5)
- d) (4-3)(3-2)(2-1)(1-5)

**Answer:** d

**Explanation:** The MST for the given graph using Prim's algorithm starting from vertex 4 is,



So, the MST contains edges (4-3)(3-2)(2-1)(1-5).

8. Prim's algorithm is also known as

- a) Dijkstra–Scholten algorithm
- b) Boruvka's algorithm
- c) Floyd–Warshall algorithm
- d) DJP Algorithm

**Answer:** d

**Explanation:** The Prim's algorithm was developed by Vojtěch Jarník and it was latter discovered by the duo Prim and Dijkstra. Therefore, Prim's algorithm is also known as DJP Algorithm.

9. Prim's algorithm can be efficiently implemented using \_\_\_\_\_ for graphs with greater density.

- a) d-ary heap
- b) linear search
- c) fibonacci heap
- d) binary search

**Answer:** a

**Explanation:** In Prim's algorithm, we add the minimum weight edge for the chosen vertex which requires searching on the array of weights. This searching can be efficiently implemented using binary heap for dense graphs. And for graphs with greater density, Prim's algorithm can be made to run in linear time using d-ary heap(generalization of binary heap).

10. Which of the following is false about Prim's algorithm?

- a) It is a greedy algorithm
- b) It constructs MST by selecting edges in increasing order of their weights
- c) It never accepts cycles in the MST
- d) It can be implemented using the Fibonacci heap

**Answer:** b

**Explanation:** Prim's algorithm can be implemented using Fibonacci heap and it

never accepts cycles. And Prim's algorithm follows greedy approach. Prim's algorithms span from one vertex to another.

1. Kruskal's algorithm is used to \_\_\_\_\_
- find minimum spanning tree
  - find single source shortest path
  - find all pair shortest path algorithm
  - traverse the graph

**Answer:** a

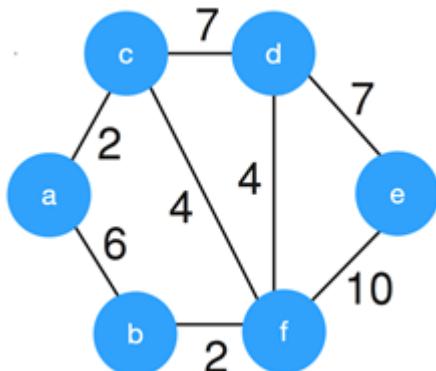
**Explanation:** The Kruskal's algorithm is used to find the minimum spanning tree of the connected graph. It constructs the MST by finding the edge having the least possible weight that connects two trees in the forest.

2. Kruskal's algorithm is a \_\_\_\_\_
- divide and conquer algorithm
  - dynamic programming algorithm
  - greedy algorithm
  - approximation algorithm

**Answer:** c

**Explanation:** Kruskal's algorithm uses a greedy algorithm approach to find the MST of the connected weighted graph. In the greedy method, we attempt to find an optimal solution in stages.

3. Consider the given graph.



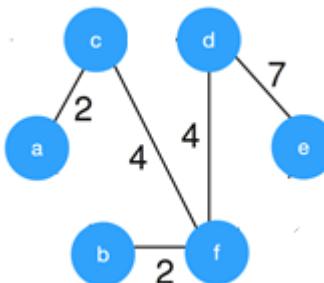
What is the weight of the minimum spanning tree using the Kruskal's algorithm?

- 24
- 23

- 15
- 19

**Answer:** d

**Explanation:** Kruskal's algorithm constructs the minimum spanning tree by constructing by adding the edges to spanning tree one-one by one. The MST for the given graph is,



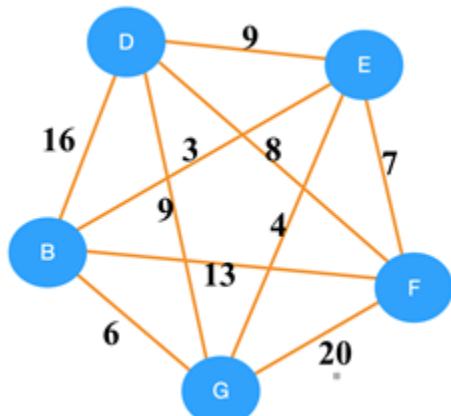
So, the weight of the MST is 19.

4. What is the time complexity of Kruskal's algorithm?
- $O(\log V)$
  - $O(E \log V)$
  - $O(E^2)$
  - $O(V \log E)$

**Answer:** b

**Explanation:** Kruskal's algorithm involves sorting of the edges, which takes  $O(E \log E)$  time, where  $E$  is a number of edges in graph and  $V$  is the number of vertices. After sorting, all edges are iterated and union-find algorithm is applied. union-find algorithm requires  $O(\log V)$  time. So, overall Kruskal's algorithm requires  $O(E \log V)$  time.

5. Consider the following graph. Using Kruskal's algorithm, which edge will be selected first?

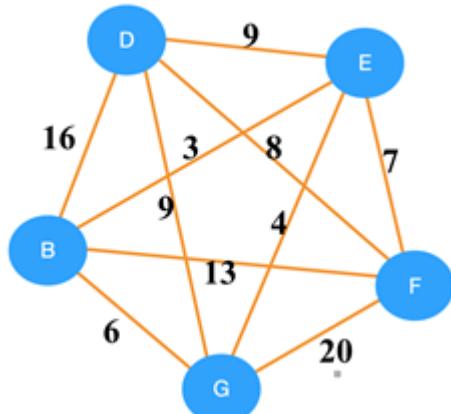


- a) GF
- b) DE
- c) BE
- d) BG

**Answer:** c

**Explanation:** In Kruskal's algorithm the edges are selected and added to the spanning tree in increasing order of their weights. Therefore, the first edge selected will be the minimal one. So, correct option is BE.

6. Which of the following edges form minimum spanning tree on the graph using kruskals algorithm?

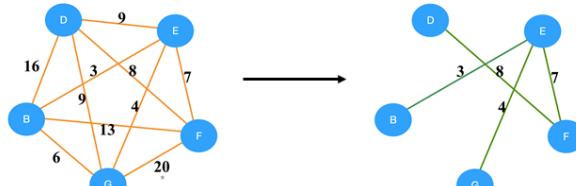


- a) (B-E)(G-E)(E-F)(D-F)
- b) (B-E)(G-E)(E-F)(B-G)(D-F)
- c) (B-E)(G-E)(E-F)(D-E)
- d) (B-E)(G-E)(E-F)(D-F)(D-G)

**Answer:** a

**Explanation:** Using Krushkal's algorithm on the given graph, the generated minimum

spanning tree is shown below.



So, the edges in the MST are, (B-E)(G-E)(E-F)(D-F).

7. Which of the following is true?

- a) Prim's algorithm can also be used for disconnected graphs
- b) Kruskal's algorithm can also run on the disconnected graphs
- c) Prim's algorithm is simpler than Kruskal's algorithm
- d) In Kruskal's sort edges are added to MST in decreasing order of their weights

**Answer:** b

**Explanation:** Prim's algorithm iterates from one node to another, so it can not be applied for disconnected graph. Kruskal's algorithm can be applied to the disconnected graphs to construct the minimum cost forest. Kruskal's algorithm is comparatively easier and simpler than prim's algorithm.

8. Which of the following is false about the Kruskal's algorithm?

- a) It is a greedy algorithm
- b) It constructs MST by selecting edges in increasing order of their weights
- c) It can accept cycles in the MST
- d) It uses union-find data structure

**Answer:** c

**Explanation:** Kruskal's algorithm is a greedy algorithm to construct the MST of the given graph. It constructs the MST by selecting edges in increasing order of their weights and rejects an edge if it may form the cycle. So, using Kruskal's algorithm is never formed.

9. Kruskal's algorithm is best suited for the dense graphs than the prim's algorithm.

- a) True
- b) False

**Answer:** b

**Explanation:** Prim's algorithm outperforms the Kruskal's algorithm in case of the dense graphs. It is significantly faster if graph has more edges than the Kruskal's algorithm.

10. Consider the following statements.
- S1. Kruskal's algorithm might produce a non-minimal spanning tree.
  - S2. Kruskal's algorithm can efficiently implemented using the disjoint-set data structure.
  - a) S1 is true but S2 is false
  - b) Both S1 and S2 are false
  - c) Both S1 and S2 are true
  - d) S2 is true but S1 is false

**Answer:** d

**Explanation:** In Kruskal's algorithm, the disjoint-set data structure efficiently identifies the components containing a vertex and adds the new edges. And Kruskal's algorithm always finds the MST for the connected graph.

---

1. Which of the following is false in the case of a spanning tree of a graph G?
- a) It is tree that spans G
  - b) It is a subgraph of the G
  - c) It includes every vertex of the G
  - d) It can be either cyclic or acyclic

**Answer:** d

**Explanation:** A graph can have many spanning trees. Each spanning tree of a graph G is a subgraph of the graph G, and spanning trees include every vertex of the graph. Spanning trees are always acyclic.

2. Every graph has only one minimum spanning tree.
- a) True
  - b) False

**Answer:** b

**Explanation:** Minimum spanning tree is a spanning tree with the lowest cost among all the spanning trees. Sum of all of the edges in the spanning tree is the cost of the spanning tree. There can be many minimum spanning trees for a given graph.

3. Consider a complete graph G with 4 vertices. The graph G has \_\_\_\_\_ spanning trees.
- a) 15
  - b) 8
  - c) 16
  - d) 13

**Answer:** c

**Explanation:** A graph can have many spanning trees. And a complete graph with n vertices has  $n^{(n-2)}$  spanning trees. So, the complete graph with 4 vertices has  $4^{(4-2)} = 16$  spanning trees.

4. The travelling salesman problem can be solved using \_\_\_\_\_
- a) A spanning tree
  - b) A minimum spanning tree
  - c) Bellman – Ford algorithm
  - d) DFS traversal

**Answer:** b

**Explanation:** In the travelling salesman problem we have to find the shortest possible route that visits every city exactly once and returns to the starting point for the given a set of cities. So, travelling salesman problem can be solved by contracting the minimum spanning tree.

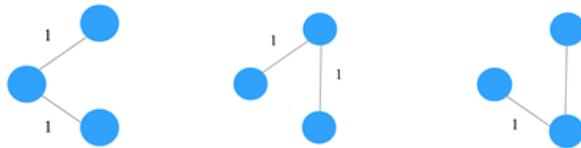
5. Consider the graph M with 3 vertices. Its adjacency matrix is shown below. Which of the following is true?

$$M = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

- a) Graph M has no minimum spanning tree
- b) Graph M has a unique minimum spanning trees of cost 2
- c) Graph M has 3 distinct minimum spanning trees, each of cost 2
- d) Graph M has 3 spanning trees of different costs

**Answer:** c

**Explanation:** Here all non-diagonal elements in the adjacency matrix are 1. So, every vertex is connected every other vertex of the graph. And, so graph M has 3 distinct minimum spanning trees.



6. Consider a undirected graph G with vertices { A, B, C, D, E}. In graph G, every edge has distinct weight. Edge CD is edge with minimum weight and edge AB is edge with maximum weight. Then, which of the following is false?

- a) Every minimum spanning tree of G must contain CD
- b) If AB is in a minimum spanning tree, then its removal must disconnect G
- c) No minimum spanning tree contains AB
- d) G has a unique minimum spanning tree

**Answer:** c

**Explanation:** Every MST will contain CD as it is smallest edge. So, Every minimum spanning tree of G must contain CD is true. And G has a unique minimum spanning tree is also true because the graph has edges with distinct weights. So, no minimum spanning tree contains AB is false.

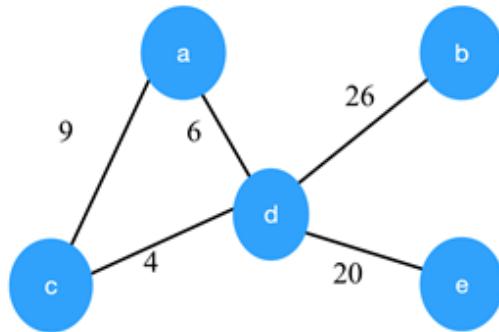
7. If all the weights of the graph are positive, then the minimum spanning tree of the graph is a minimum cost subgraph.

- a) True
- b) False

**Answer:** a

**Explanation:** A subgraph is a graph formed from a subset of the vertices and edges of the original graph. And the subset of vertices includes all endpoints of the subset of the edges. So, we can say MST of a graph is a subgraph when all weights in the original graph are positive.

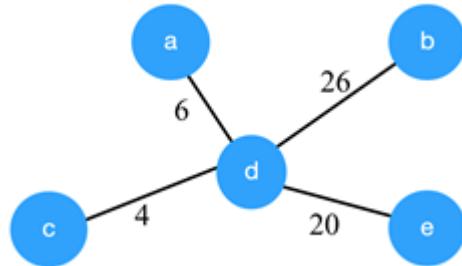
8. Consider the graph shown below. Which of the following are the edges in the MST of the given graph?



- a) (a-c)(c-d)(d-b)(d-b)
- b) (c-a)(a-d)(d-b)(d-e)
- c) (a-d)(d-c)(d-b)(d-e)
- d) (c-a)(a-d)(d-c)(d-b)(d-e)

**Answer:** c

**Explanation:** The minimum spanning tree of the given graph is shown below. It has cost 56.



9. Which of the following is not the algorithm to find the minimum spanning tree of the given graph?

- a) Boruvka's algorithm
- b) Prim's algorithm
- c) Kruskal's algorithm
- d) Bellman–Ford algorithm

**Answer:** d

**Explanation:** The Boruvka's algorithm, Prim's algorithm and Kruskal's algorithm are the algorithms that can be used to find the minimum spanning tree of the given graph. The Bellman-Ford algorithm is used to find the shortest path from the single source to all other vertices.

10. Which of the following is false?
  - a) The spanning trees do not have any cycles
  - b) MST have  $n - 1$  edges if the graph has n edges
  - c) Edge e belonging to a cut of the graph if has the weight smaller than any other edge in the same cut, then the edge e is present in all the MSTs of the graph
  - d) Removing one edge from the spanning tree will not make the graph disconnected

**Answer:** d

**Explanation:** Every spanning tree has  $n - 1$  edges if the graph has n edges and has no cycles. The MST follows the cut property, Edge e belonging to a cut of the graph if has the weight smaller than any other edge in the same cut, then the edge e is present in all the MSTs of the graph.

1. Fractional knapsack problem is also known as \_\_\_\_\_
  - a) 0/1 knapsack problem
  - b) Continuous knapsack problem
  - c) Divisible knapsack problem
  - d) Non continuous knapsack problem

**Answer:** b

**Explanation:** Fractional knapsack problem is also called continuous knapsack problem. Fractional knapsack is solved using dynamic programming.

2. Fractional knapsack problem is solved most efficiently by which of the following algorithm?
  - a) Divide and conquer
  - b) Dynamic programming

- c) Greedy algorithm
- d) Backtracking

**Answer:** c

**Explanation:** Greedy algorithm is used to solve this problem. We first sort items according to their value/weight ratio and then add item with highest ratio until we cannot add the next item as a whole. At the end, we add the next item as much as we can.

3. What is the objective of the knapsack problem?
  - a) To get maximum total value in the knapsack
  - b) To get minimum total value in the knapsack
  - c) To get maximum weight in the knapsack
  - d) To get minimum weight in the knapsack

**Answer:** a

**Explanation:** The objective is to fill the knapsack of some given volume with different materials such that the value of selected items is maximized.

4. Which of the following statement about 0/1 knapsack and fractional knapsack problem is correct?
  - a) In 0/1 knapsack problem items are divisible and in fractional knapsack items are indivisible
  - b) Both are the same
  - c) 0/1 knapsack is solved using a greedy algorithm and fractional knapsack is solved using dynamic programming
  - d) In 0/1 knapsack problem items are indivisible and in fractional knapsack items are divisible

**Answer:** d

**Explanation:** In fractional knapsack problem we can partially include an item into the knapsack whereas in 0/1 knapsack we have to either include or exclude the item wholly.

5. Time complexity of fractional knapsack problem is \_\_\_\_\_

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

**Answer:** a

**Explanation:** As the main time taking a step is of sorting so it defines the time complexity of our code. So the time complexity will be  $O(n \log n)$  if we use quick sort for sorting.

6. Fractional knapsack problem can be solved in time  $O(n)$ .

- a) True
- b) False

**Answer:** a

**Explanation:** It is possible to solve the problem in  $O(n)$  time by adapting the algorithm for finding weighted medians.

7. Given items as {value,weight} pairs  $\{\{40,20\}, \{30,10\}, \{20,5\}\}$ . The capacity of knapsack=20. Find the maximum value output assuming items to be divisible.

- a) 60
- b) 80
- c) 100
- d) 40

**Answer:** a

**Explanation:** The value/weight ratio are  $\{2,3,4\}$ . So we include the second and third items wholly into the knapsack. This leaves only 5 units of volume for the first item. So we include the first item partially.  
Final value =  $20+30+(40/4)=60$ .

8. The result of the fractional knapsack is greater than or equal to 0/1 knapsack.

- a) True
- b) False

**Answer:** a

**Explanation:** As fractional knapsack gives extra liberty to include the object partially which is not possible with 0/1 knapsack, thus

we get better results with a fractional knapsack.

9. The main time taking step in fractional knapsack problem is \_\_\_\_\_

- a) Breaking items into fraction
- b) Adding items into knapsack
- c) Sorting
- d) Looping through sorted items

**Answer:** c

**Explanation:** The main time taking step is to sort the items according to their value/weight ratio. It defines the time complexity of the code.

10. Given items as {value,weight} pairs  $\{\{60,20\}, \{50,25\}, \{20,5\}\}$ . The capacity of knapsack=40. Find the maximum value output assuming items to be divisible and nondivisible respectively.

- a) 100, 80
- b) 110, 70
- c) 130, 110
- d) 110, 80

**Answer:** d

**Explanation:** Assuming items to be divisible- The value/weight ratio are  $\{3, 2, 4\}$ . So we include third and first items wholly. So, now only 15 units of volume are left for second item. So we include it partially.

Final volume =

$$20+60+50x(15/25)=80+30=110$$

Assuming items to be indivisible- In this case we will have to leave one item due to insufficient capacity.

Final volume =  $60 + 20 = 80$ .

**Sanfoundry Global Education & Learning Series – Data Structures & Algorithms.**

## **UNIT IV ITERATIVE IMPROVEMENT**

1. Given G is a bipartite graph and the bipartitions of this graphs are U and V respectively. What is the relation between them?

- a) Number of vertices in U = Number of vertices in V
- b) Sum of degrees of vertices in U = Sum of degrees of vertices in V
- c) Number of vertices in U > Number of vertices in V
- d) Nothing can be said

**Answer:** b

**Explanation:** We can prove this by induction. By adding one edge, the degree of vertices in U is equal to 1 as well as in V. Let us assume that this is true for  $n-1$  edges and add one more edge. Since the given edge adds exactly once to both U and V we can tell that this statement is true for all n vertices.

2. A k-regular bipartite graph is the one in which degree of each vertices is k for all the vertices in the graph. Given that the bipartitions of this graph are U and V respectively. What is the relation between them?

- a) Number of vertices in U=Number of vertices in V
- b) Number of vertices in U not equal to number of vertices in V
- c) Number of vertices in U always greater than the number of vertices in V
- d) Nothing can be said

**Answer:** a

**Explanation:** We know that in a bipartite graph sum of degrees of vertices in U= sum of degrees of vertices in V. Given that the graph is a k-regular bipartite graph, we have  $k^*(\text{number of vertices in U})=k^*(\text{number of vertices in V})$ .

3. There are four students in a class namely A, B, C and D. A tells that a triangle is a bipartite graph. B tells pentagon is a bipartite graph. C tells square is a bipartite graph. D tells heptagon is a bipartite graph. Who

among the following is correct?

- a) A
- b) B
- c) C
- d) D

**Answer:** c

**Explanation:** We can prove it in this following way. Let '1' be a vertex in bipartite set X and let '2' be a vertex in the bipartite set Y. Therefore the bipartite set X contains all odd numbers and the bipartite set Y contains all even numbers. Now let us consider a graph of odd cycle (a triangle). There exists an edge from '1' to '2', '2' to '3' and '3' to '1'. The latter case ('3' to '1') makes an edge to exist in a bipartite set X itself. Therefore telling us that graphs with odd cycles are not bipartite.

4. A complete bipartite graph is a one in which each vertex in set X has an edge with set Y. Let n be the total number of vertices. For maximum number of edges, the total number of vertices hat should be present on set X is?

- a) n
- b)  $n/2$
- c)  $n/4$
- d) data insufficient

**Answer:** b

**Explanation:** We can prove this by calculus. Let x be the total number of vertices on set X. Therefore set Y will have  $n-x$ . We have to maximize  $x*(n-x)$ . This is true when  $x=n/2$ .

5. When is a graph said to be bipartite?

- a) If it can be divided into two independent sets A and B such that each edge connects a vertex from to A to B
- b) If the graph is connected and it has odd number of vertices
- c) If the graph is disconnected
- d) If the graph has at least  $n/2$  vertices whose degree is greater than  $n/2$

**Answer:** a

**Explanation:** A graph is said to be bipartite if it can be divided into two independent sets A and B such that each edge connects a vertex from A to B.

6. Are trees bipartite?

- a) Yes
- b) No
- c) Yes if it has even number of vertices
- d) No if it has odd number of vertices

**Answer:** a

**Explanation:** Condition needed is that there should not be an odd cycle. But in a tree there are no cycles at all. Hence it is bipartite.

7. A graph has 20 vertices. The maximum number of edges it can have is? (Given it is bipartite)

- a) 100
- b) 140
- c) 80
- d) 20

**Answer:** a

**Explanation:** Let the given bipartition X have x vertices, then Y will have  $20-x$  vertices. We need to maximize  $x*(20-x)$ . This will be maxed when  $x=10$ .

8. Given that a graph contains no odd cycle. Is it enough to tell that it is bipartite?

- a) Yes
- b) No

**Answer:** a

**Explanation:** It is required that the graph is connected also. If it is not then it cannot be called a bipartite graph.

9. Can there exist a graph which is both eulerian and is bipartite?

- a) Yes
- b) No
- c) Yes if it has even number of edges
- d) Nothing can be said

**Answer:** a

**Explanation:** If a graph is such that there exists a path which visits every edge atleast once, then it is said to be Eulerian. Taking an example of a square, the given question evaluates to yes.

10. A graph is found to be 2 colorable. What can be said about that graph?

- a) The given graph is eulerian
- b) The given graph is bipartite
- c) The given graph is hamiltonian
- d) The given graph is planar

**Answer:** b

**Explanation:** A graph is said to be colorable if two vertices connected by an edge are never of the same color. 2 colorable mean that this can be achieved with just 2 colors.

1. Which type of graph has no odd cycle in it?

- a) Bipartite
- b) Histogram
- c) Cartesian
- d) Pie

**Answer:** a

**Explanation:** The graph is known as Bipartite if the graph does not contain any odd length cycle in it. Odd length cycle means a cycle with the odd number of vertices in it.

2. What type of graph has chromatic number less than or equal to 2?

- a) Histogram
- b) Bipartite
- c) Cartesian
- d) Tree

**Answer:** b

**Explanation:** A graph is known as bipartite graph if and only if it has the total chromatic number less than or equal to 2. The smallest number of graphs needed to color the graph is chromatic number.

3. Which of the following is the correct type of spectrum of the bipartite graph?

- a) Symmetric
- b) Anti – Symmetric
- c) Circular
- d) Exponential

**Answer:** a

**Explanation:** The spectrum of the bipartite graph is symmetric in nature. The spectrum is the property of graph that are related to polynomial, Eigen values, Eigen vectors of the matrix related to graph.

4. Which of the following is not a property of the bipartite graph?

- a) No Odd Cycle
- b) Symmetric spectrum
- c) Chromatic Number Is Less Than or Equal to 2
- d) Asymmetric spectrum

**Answer:** d

**Explanation:** A graph is known to be bipartite if it has odd length cycle number. It also has symmetric spectrum and the bipartite graph contains the total chromatic number less than or equal to 2.

5. Which one of the following is the chromatic number of bipartite graph?

- a) 1
- b) 4
- c) 3
- d) 5

**Answer:** a

**Explanation:** A graph is known as bipartite graph if and only if it has the total chromatic number less than or equal to 2. The smallest number of graphs needed to color the graph is the chromatic number.

6. Which graph has a size of minimum vertex cover equal to maximum matching?

- a) Cartesian
- b) Tree

- c) Heap
- d) Bipartite

**Answer:** d

**Explanation:** The Konig's theorem given the equivalence relation between the minimum vertex cover and the maximum matching in graph theory. Bipartite graph has a size of minimum vertex cover equal to maximum matching.

7. Which theorem gives the relation between the minimum vertex cover and maximum matching?

- a) Konig's Theorem
- b) Kirchhoff's Theorem
- c) Kuratowski's Theorem
- d) Kelmans Theorem

**Answer:** a

**Explanation:** The Konig's theorem given the equivalence relation between the minimum vertex cover and the maximum matching in graph theory. Bipartite graph has a size of minimum vertex cover equal to maximum matching.

8. Which of the following is not a property of perfect graph?

- a) Compliment of Line Graph of Bipartite Graph
- b) Compliment of Bipartite Graph
- c) Line Graph of Bipartite Graph
- d) Line Graph

**Answer:** d

**Explanation:** The Compliment of Line Graph of Bipartite Graph, Compliment of Bipartite Graph, Line Graph of Bipartite Graph and every Bipartite Graph is known as a perfect graph in graph theory. Normal line graph is not a perfect graph whereas line perfect graph is a graph whose line graph is a perfect graph.

9. Which of the following graphs don't have chromatic number less than or equal to 2?

- a) Compliment of Line Graph of Bipartite

- Graph  
 b) Compliment of Bipartite Graph  
 c) Line Graph of Bipartite Graph  
 d) Wheel graph

**Answer:** d

**Explanation:** The perfect bipartite graph has chromatic number 2. Also, the Compliment of Line Graph of Bipartite Graph, Compliment of Bipartite Graph, Line Graph of Bipartite Graph and every Bipartite Graph is known as perfect graph in graph theory. Wheel graph  $W_n$  has chromatic number 3 if  $n$  is odd and 4 if  $n$  is even.

10. Which of the following has maximum clique size 2?

- a) Perfect graph
- b) Tree
- c) Histogram
- d) Cartesian

**Answer:** a

**Explanation:** The perfect bipartite graph has clique size 2. Also, the clique size of Compliment of Line Graph of Bipartite Graph, Compliment of Bipartite Graph, Line Graph of Bipartite Graph and every Bipartite Graph is 2.

11. What is the chromatic number of compliment of line graph of bipartite graph?

- a) 0
- b) 1
- c) 2
- d) 3

**Answer:** c

**Explanation:** The perfect bipartite graph has chromatic number 2. So the Compliment of Line Graph of Bipartite Graph, Compliment of Bipartite Graph, Line Graph of Bipartite Graph and every Bipartite Graph has chromatic number 2.

12. What is the clique size of the line graph of bipartite graph?

- a) 0

- b) 1
- c) 2
- d) 3

**Answer:** c

**Explanation:** The perfect bipartite graph has clique size 2. So the clique size of Compliment of Line Graph of Bipartite Graph, Compliment of Bipartite Graph, Line Graph of Bipartite Graph and every Bipartite Graph is 2.

13. It is possible to have a negative chromatic number of bipartite graph.

- a) True
- b) False

**Answer:** b

**Explanation:** A graph is known as bipartite graph if and only if it has the total chromatic number less than or equal to 2. The smallest number of graphs needed to color the graph is the chromatic number. But the chromatic number cannot be negative.

14. Every Perfect graph has forbidden graph characterization.

- a) True
- b) False

**Answer:** a

**Explanation:** Berge theorem proves the forbidden graph characterization of every perfect graphs. Because of that reason every bipartite graph is perfect graph.

15. Which structure can be modelled by using Bipartite graph?

- a) Hypergraph
- b) Perfect Graph
- c) Hetero Graph
- d) Directed Graph

**Answer:** a

**Explanation:** A combinatorial structure such as Hypergraph can be made using the bipartite graphs. A hypergraph in graph

theory is a type of graph in which edge can join any number of vertices.

1. Which type of graph has all the vertex of the first set connected to all the vertex of the second set?
  - a) Bipartite
  - b) Complete Bipartite
  - c) Cartesian
  - d) Pie

**Answer:** b

**Explanation:** The graph is known as Bipartite if the graph does not contain any odd length cycle in it. The complete bipartite graph has all the vertex of first set connected to all the vertex of second set.

2. Which graph is also known as biclique?
  - a) Histogram
  - b) Complete Bipartite
  - c) Cartesian
  - d) Tree

**Answer:** b

**Explanation:** A graph is known as complete bipartite graph if and only if it has all the vertex of first set connected to all the vertex of second set. Complete Bipartite graph is also known as Biclique.

3. Which term defines all the complete bipartite graph that are trees?
  - a) Symmetric
  - b) Anti – Symmetric
  - c) Circular
  - d) Stars

**Answer:** d

**Explanation:** Star is a complete bipartite graph with one internal node and k leaves. Therefore, all complete bipartite graph which is trees are known as stars in graph theory.

4. How many edges does a n vertex triangle free graph contains?
  - a)  $n^2$

- b)  $n^2 + 2$
- c)  $n^2 / 4$
- d)  $n^3$

**Answer:** c

**Explanation:** A n vertex triangle free graph contains a total of  $n^2 / 4$  number of edges. This is stated by Mantel's Theorem which is a special case in Turan's theorem for r=2.

5. Which graph is used to define the claw free graph?
  - a) Bipartite Graph
  - b) Claw Graph
  - c) Star Graph
  - d) Cartesian Graph

**Answer:** b

**Explanation:** Star is a complete bipartite graph with one internal node and k leaves. Star with three edges is called a claw. Hence this graph is used to define claw free graph.

6. What is testing of a complete bipartite subgraph in a bipartite graph problem called?
  - a) P Problem
  - b) P-Complete Problem
  - c) NP Problem
  - d) NP-Complete Problem

**Answer:** d

**Explanation:** NP stands for nondeterministic polynomial time. In a bipartite graph, the testing of a complete bipartite subgraph in a bipartite graph is an NP-Complete Problem.

7. Which graph cannot contain K3, 3 as a minor of graph?
  - a) Planar Graph
  - b) Outer Planar Graph
  - c) Non Planar Graph
  - d) Inner Planar Graph

**Answer:** a

**Explanation:** Minor graph is formed by deleting certain number of edges from a graph or by deleting certain number off

vertices from a graph. Hence Planar graph cannot contain  $K_3, 3$  as a minor graph.

8. Which of the following is not an Eigen value of the adjacency matrix of the complete bipartite graph?

- a)  $(nm)^{1/2}$
- b)  $(-nm)^{1/2}$
- c) 0
- d)  $nm$

**Answer:** d

**Explanation:** The adjacency matrix is a square matrix that is used to represent a finite graph. Therefore, the Eigen values for the complete bipartite graph is found to be  $(nm)^{1/2}$ ,  $(-nm)^{1/2}$ , 0.

9. Which complete graph is not present in minor of Outer Planar Graph?

- a)  $K_3, 3$
- b)  $K_3, 1$
- c)  $K_3, 2$
- d)  $K_1, 1$

**Answer:** c

**Explanation:** Minor graph is formed by deleting certain number of edges from a graph or by deleting certain number off vertices from a graph. Hence Outer Planar graph cannot contain  $K_3, 2$  as a minor graph.

10. Is every complete bipartite graph a Moore Graph.

- a) True
- b) False

**Answer:** a

**Explanation:** In graph theory, Moore graph is defined as a regular graph that has a degree  $d$  and diameter  $k$ . therefore, every complete bipartite graph is a Moore Graph.

11. What is the multiplicity for the adjacency matrix of complete bipartite graph for 0 Eigen value?

- a) 1
- b)  $n + m - 2$

- c) 0
- d) 2

**Answer:** b

**Explanation:** The adjacency matrix is a square matrix that is used to represent a finite graph. The multiplicity of the adjacency matrix off complete bipartite graph with Eigen Value 0 is  $n + m - 2$ .

12. Which of the following is not an Eigen value of the Laplacian matrix of the complete bipartite graph?

- a)  $n + m$
- b) n
- c) 0
- d)  $n * m$

**Answer:** d

**Explanation:** The laplacian matrix is used to represent a finite graph in the mathematical field of Graph Theory. Therefore, the Eigen values for the complete bipartite graph is found to be  $n + m$ , n, m, 0.

13. What is the multiplicity for the laplacian matrix of the complete bipartite graph for n Eigen value?

- a) 1
- b)  $m-1$
- c)  $n-1$
- d) 0

**Answer:** b

**Explanation:** The laplacian matrix is used to represent a finite graph in the mathematical field of Graph Theory. The multiplicity of the laplacian matrix of complete bipartite graph with Eigen Value n is  $m-1$ .

14. Is it true that every complete bipartite graph is a modular graph.

- a) True
- b) False

**Answer:** a

**Explanation:** Yes, the modular graph in graph theory is defined as an undirected

graph in which all three vertices have at least one median vertex. So all complete bipartite graph is called modular graph.

15. How many spanning trees does a complete bipartite graph contain?

- a)  $n^m$
- b)  $m^{n-1} * n^{n-1}$
- c) 1
- d) 0

**Answer:** b

**Explanation:** Spanning tree of a given graph is defined as the subgraph or the tree with all the given vertices but having minimum number of edges. So, there are a total of  $m^{n-1} * n^{n-1}$  spanning trees for a complete bipartite graph.

---

1. What does Maximum flow problem involve?

- a) finding a flow between source and sink that is maximum
- b) finding a flow between source and sink that is minimum
- c) finding the shortest path between source and sink
- d) computing a minimum spanning tree

**Answer:** a

**Explanation:** The maximum flow problem involves finding a feasible flow between a source and a sink in a network that is maximum and not minimum.

2. A network can have only one source and one sink.

- a) False
- b) True

**Answer:** b

**Explanation:** A network can have only one source and one sink inorder to find the feasible flow in a weighted connected graph.

3. What is the source?
  - a) Vertex with no incoming edges
  - b) Vertex with no leaving edges
  - c) Centre vertex
  - d) Vertex with the least weight

**Answer:** a

**Explanation:** Vertex with no incoming edges is called as a source. Vertex with no leaving edges is called as a sink.

4. Which algorithm is used to solve a maximum flow problem?

- a) Prim's algorithm
- b) Kruskal's algorithm
- c) Dijkstra's algorithm
- d) Ford-Fulkerson algorithm

**Answer:** d

**Explanation:** Ford-fulkerson algorithm is used to compute the maximum feasible flow between a source and a sink in a network.

5. Does Ford- Fulkerson algorithm use the idea of?

- a) Naïve greedy algorithm approach
- b) Residual graphs
- c) Minimum cut
- d) Minimum spanning tree

**Answer:** b

**Explanation:** Ford-Fulkerson algorithm uses the idea of residual graphs which is an extension of naïve greedy approach allowing undo operations.

6. The first step in the naïve greedy algorithm is?

- a) analysing the zero flow
- b) calculating the maximum flow using trial and error
- c) adding flows with higher values
- d) reversing flow if required

**Answer:** a

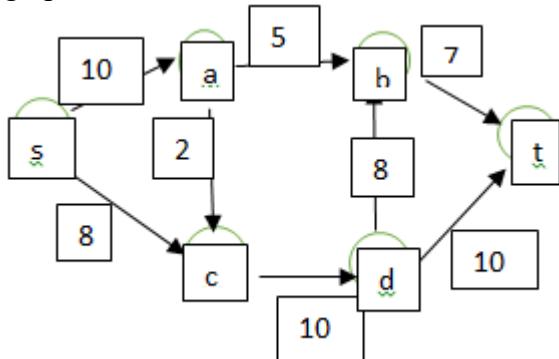
**Explanation:** The first step in the naïve greedy algorithm is to start with the zero flow followed by adding edges with higher values.

7. Under what condition can a vertex combine and distribute flow in any manner?
- It may violate edge capacities
  - It should maintain flow conservation
  - The vertex should be a source vertex
  - The vertex should be a sink vertex

**Answer:** b

**Explanation:** A vertex can combine and distribute flow in any manner but it should not violate edge capacities and it should maintain flow conservation.

8. Find the maximum flow from the following graph.



- 22
- 17
- 15
- 20

**Answer:** c

**Explanation:** Initially, zero flow is computed. Then, computing flow=  $7+1+5+2=15$ . Hence, maximum flow=15.

9. A simple acyclic path between source and sink which pass through only positive weighted edges is called?

- augmenting path
- critical path
- residual path
- maximum path

**Answer:** a

**Explanation:** Augmenting path between source and sink is a simple path without cycles. Path consisting of zero slack edges is called critical path.

10. In what time can an augmented path be found?
- $O(|E| \log |V|)$
  - $O(|E|)$
  - $O(|E|^2)$
  - $O(|E|^2 \log |V|)$

**Answer:** b

**Explanation:** An augmenting path can be found in  $O(|E|)$  mathematically by an unweighted shortest path algorithm.

11. Dinic's algorithm runs faster than the Ford-Fulkerson algorithm.

- true
- false

**Answer:** a

**Explanation:** Dinic's algorithm includes construction of level graphs and residual graphs and finding of augmenting paths along with blocking flow and is faster than the Ford-Fulkerson algorithm.

12. What is the running time of an unweighted shortest path algorithm whose augmenting path is the path with the least number of edges?

- $O(|E|)$
- $O(|E||V|)$
- $O(|E|^2|V|)$
- $O(|E| \log |V|)$

**Answer:** c

**Explanation:** Each augmenting step takes  $O(|E|)$  using an unweighted shortest path algorithm yielding a  $O(|E|2|V|)$  bound on the running time.

13. Who is the formulator of Maximum flow problem?

- Lester R. Ford and Delbert R. Fulkerson
- T.E. Harris and F.S. Ross
- Y.A. Dinitz
- Kruskal

**Answer:** b

**Explanation:** The first ever people to

formulate Maximum flow problem were T.E. Harris and F.S. Ross. Lester R. Ford and Delbert R. Fulkerson formulated Ford-Fulkerson algorithm.

14. What is the running time of Dinic's blocking flow algorithm?
- $O(V^2E)$
  - $O(VE^2)$
  - $O(V^3)$
  - $O(E \max |f|)$

**Answer:** a

**Explanation:** The running time of Dinic's blocking flow algorithm is  $O(V^2E)$ . The running of Ford-Fulkerson algorithm is  $O(E \max |f|)$ .

15. How many constraints does flow have?
- one
  - three
  - two
  - four

**Answer:** c

**Explanation:** A flow is a mapping which follows two constraints- conservation of flows and capacity constraints.

1. Stable marriage problem is an example of?
- Branch and bound algorithm
  - Backtracking algorithm
  - Greedy algorithm
  - Divide and conquer algorithm

**Answer:** b

**Explanation:** Stable marriage problem is an example for recursive algorithm because it recursively uses backtracking algorithm to find an optimal solution.

2. Which of the following algorithms does Stable marriage problem uses?
- Gale-Shapley algorithm
  - Dijkstra's algorithm

- Ford-Fulkerson algorithm
- Prim's algorithm

**Answer:** a

**Explanation:** Stable marriage problem uses Gale-Shapley algorithm. Maximum flow problem uses Ford-Fulkerson algorithm. Prim's algorithm involves minimum spanning tree.

3. An optimal solution satisfying men's preferences is said to be?
- Man optimal
  - Woman optimal
  - Pair optimal
  - Best optimal

**Answer:** a

**Explanation:** An optimal solution satisfying men's preferences are said to be man optimal. An optimal solution satisfying woman's preferences are said to be woman optimal.

4. When a free man proposes to an available woman, which of the following happens?
- She will think and decide
  - She will reject
  - She will replace her current mate
  - She will accept

**Answer:** d

**Explanation:** When a man proposes to an available woman, she will accept his proposal irrespective of his position on his preference list.

5. If there are n couples who would prefer each other to their actual marriage partners, then the assignment is said to be unstable.
- True
  - False

**Answer:** a

**Explanation:** If there are n couples such that a man and a woman are not married, and if they prefer each other to their actual partners, the assignment is unstable.

6. How many  $2 \times 2$  matrices are used in this problem?

- a) 1
- b) 2
- c) 3
- d) 4

**Answer:** b

**Explanation:** Two  $2 \times 2$  matrices are used. One for men representing corresponding woman and ranking and the other for women.

7. What happens when a free man approaches a married woman?

- a) She simply rejects him
- b) She simply replaces her mate with him
- c) She goes through her preference list and accordingly, she replaces her current mate with him
- d) She accepts his proposal

**Answer:** c

**Explanation:** If the preference of the man is greater, she replaces her current mate with him, leaving her current mate free.

8. In case of stability, how many symmetric possibilities of trouble can occur?

- a) 1
- b) 2
- c) 4
- d) 3

**Answer:** b

**Explanation:** Possibilities- There might be a woman pw, preferred to w by m, who herself prefers m to be her husband and the same applies to man as well.

9. Consider the following ranking matrix.

|    | W1   | W2   | W3   |
|----|------|------|------|
| M1 | 1, 2 | 3, 3 | 2, 1 |
| M2 | 3, 1 | 2, 1 | 1, 2 |
| M3 | 2, 3 | 3, 2 | 1, 3 |

Assume that M1 and W2 are married. Now, M2 approaches W2. Which of the following happens?

- a) W2 replaces M1 with M2
- b) W2 rejects M2
- c) W2 accepts both M1 and M2
- d) W2 rejects both M1 and M2

**Answer:** a

**Explanation:** W2 is married to M1. But the preference of W2 has M2 before M1. Hence, W2 replaces M1 with M2.

10. Consider the following ranking matrix.

|    | W1   | W2   | W3   |
|----|------|------|------|
| M1 | 1, 2 | 3, 3 | 2, 1 |
| M2 | 3, 1 | 2, 1 | 1, 2 |
| M3 | 2, 3 | 3, 2 | 1, 3 |

Assume that M1 and W1 are married and M2 and W3 are married. Now, whom will M3 approach first?

- a) W1
- b) W2
- c) W3
- d) All three

**Answer:** c

**Explanation:** M3 will approach W3 first. Since W3 is married and since her preference

list has her current mate before M3, she rejects his proposal.

11. Who formulated a straight forward backtracking scheme for stable marriage problem?

- a) McVitie and Wilson
- b) Gale
- c) Ford and Fulkerson
- d) Dinitz

**Answer:** a

**Explanation:** McVitie and Wilson formulated a much faster straight forward backtracking scheme for stable marriage problem. Ford and Fulkerson formulated Maximum flow problem.

12. Can stable marriage cannot be solved using branch and bound algorithm.

- a) True
- b) False

**Answer:** b

**Explanation:** Stable marriage problem can be solved using branch and bound approach because branch and bound follows backtracking scheme with a limitation factor.

13. What is the prime task of the stable marriage problem?

- a) To provide man optimal solution
- b) To provide woman optimal solution
- c) To determine stability of marriage
- d) To use backtracking approach

**Answer:** c

**Explanation:** The prime task of stable marriage problem is to determine stability of marriage (i.e.) finding a man and a woman who prefer each other to others.

14. Which of the following problems is related to stable marriage problem?

- a) Choice of school by students
- b) N-queen problem
- c) Arranging data in a database
- d) Knapsack problem

**Answer:** a

**Explanation:** Choice of school by students is the most related example in the given set of options since both school and students will have a preference list.

15. What is the efficiency of Gale-Shapley algorithm used in stable marriage problem?

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

**Answer:** c

**Explanation:** The time efficiency of Gale-Shapley algorithm is mathematically found to be  $O(N^2)$  where N denotes stable marriage problem.

1. \_\_\_\_\_ is a matching with the largest number of edges.

- a) Maximum bipartite matching
- b) Non-bipartite matching
- c) Stable marriage
- d) Simplex

**Answer:** a

**Explanation:** Maximum bipartite matching matches two elements with a property that no two edges share a vertex.

2. Maximum matching is also called as maximum cardinality matching.

- a) True
- b) False

**Answer:** a

**Explanation:** Maximum matching is also called as maximum cardinality matching (i.e.) matching with the largest number of edges.

3. How many colours are used in a bipartite graph?

- a) 1
- b) 2
- c) 3
- d) 4

**Answer:** b

**Explanation:** A bipartite graph is said to be two-colourable so that every edge has its vertices coloured in different colours.

4. What is the simplest method to prove that a graph is bipartite?
  - a) It has a cycle of an odd length
  - b) It does not have cycles
  - c) It does not have a cycle of an odd length
  - d) Both odd and even cycles are formed

**Answer:** c

**Explanation:** It is not difficult to prove that a graph is bipartite if and only if it does not have a cycle of an odd length.

5. A matching that matches all the vertices of a graph is called?
  - a) Perfect matching
  - b) Cardinality matching
  - c) Good matching
  - d) Simplex matching

**Answer:** a

**Explanation:** A matching that matches all the vertices of a graph is called perfect matching.

6. What is the length of an augmenting path?
  - a) Even
  - b) Odd
  - c) Depends on graph
  - d) 1

**Answer:** b

**Explanation:** The length of an augmenting path in a bipartite graph is always said to be always odd.

7. In a bipartite graph  $G=(V,U,E)$ , the matching of a free vertex in  $V$  to a free vertex in  $U$  is called?
  - a) Bipartite matching
  - b) Cardinality matching
  - c) Augmenting
  - d) Weight matching

**Answer:** c

**Explanation:** A simple path from a free vertex in  $V$  to a free vertex in  $U$  whose edges alternate between edges not in  $M$  and edges in  $M$  is called a augmenting path.

8. A matching  $M$  is maximal if and only if there exists no augmenting path with respect to  $M$ .
  - a) True
  - b) False

**Answer:** a

**Explanation:** According to the theorem discovered by the French mathematician Claude Berge, it means that the current matching is maximal if there is no augmenting path.

9. Which one of the following is an application for matching?
  - a) Proposal of marriage
  - b) Pairing boys and girls for a dance
  - c) Arranging elements in a set
  - d) Finding the shortest traversal path

**Answer:** b

**Explanation:** Pairing boys and girls for a dance is a traditional example for matching. Proposal of marriage is an application of stable marriage problem.

10. Which is the correct technique for finding a maximum matching in a graph?
  - a) DFS traversal
  - b) BFS traversal
  - c) Shortest path traversal
  - d) Heap order traversal

**Answer:** b

**Explanation:** The correct technique for finding a maximum matching in a bipartite graph is by using a Breadth First Search(BFS).

11. The problem of maximizing the sum of weights on edges connecting matched pairs of vertices is?

- a) Maximum- mass matching
- b) Maximum bipartite matching
- c) Maximum weight matching
- d) Maximum node matching

**Answer:** c

**Explanation:** The problem is called as maximum weight matching which is similar to a bipartite matching. It is also called as assignment problem.

12. What is the total number of iterations used in a maximum- matching algorithm?
- a)  $[n/2]$
  - b)  $[n/3]$
  - c)  $[n/2]+n$
  - d)  $[n/2]+1$

**Answer:** d

**Explanation:** The total number of iterations cannot exceed  $[n/2]+1$  where  $n=|V|+|U|$  denoting the number of vertices in the graph.

13. What is the efficiency of algorithm designed by Hopcroft and Karp?
- a)  $O(n+m)$
  - b)  $O(n(n+m))$
  - c)  $O(\sqrt{n(n+m)})$
  - d)  $O(n^2)$

**Answer:** c

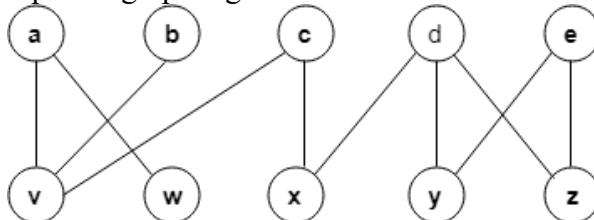
**Explanation:** The efficiency of algorithm designed by Hopcroft and Karp is mathematically found to be  $O(\sqrt{n(n+m)})$ .

14. Who was the first person to solve the maximum matching problem?
- a) Jack Edmonds
  - b) Hopcroft
  - c) Karp
  - d) Claude Berge

**Answer:** a

**Explanation:** Jack Edmonds was the first person to solve the maximum matching problem in 1965.

15. From the given graph, how many vertices can be matched using maximum matching in bipartite graph algorithm?



- a) 5
- b) 4
- c) 3
- d) 2

**Answer:** a

**Explanation:** One of the solutions of the matching problem is given by a-w,b-v,c-x,d-y,e-z. Hence the answer is 5.

---



---

## UNIT V COPING WITH THE LIMITATIONS OF ALGORITHM POWER

1. The worst-case efficiency of solving a problem in polynomial time is?
- a)  $O(p(n))$
  - b)  $O(p( n \log n))$
  - c)  $O(p(n^2))$
  - d)  $O(p(m \log n))$

**Answer:** a

**Explanation:** The worst-case efficiency of solving an problem in polynomial time is  $O(p(n))$  where  $p(n)$  is the polynomial time of input size.

2. Problems that can be solved in polynomial time are known as?
- a) intractable
  - b) tractable
  - c) decision
  - d) complete

**Answer:** b

**Explanation:** Problems that can be solved in polynomial time are known as tractable. Problems that cannot be solved in polynomial time are intractable.

3. The sum and composition of two polynomials are always polynomials.

- a) true
- b) false

**Answer:** a

**Explanation:** One of the properties of polynomial functions states that the sum and composition of two polynomials are always polynomials.

4. \_\_\_\_\_ is the class of decision problems that can be solved by non-deterministic polynomial algorithms?

- a) NP
- b) P
- c) Hard
- d) Complete

**Answer:** a

**Explanation:** NP problems are called as non-deterministic polynomial problems. They are a class of decision problems that can be solved using NP algorithms.

5. Problems that cannot be solved by any algorithm are called?

- a) tractable problems
- b) intractable problems
- c) undecidable problems
- d) decidable problems

**Answer:** c

**Explanation:** Problems that cannot be solved by any algorithm are called undecidable problems. Problems that can be solved in polynomial time are called Tractable problems.

6. The Euler's circuit problem can be solved in?

- a)  $O(N)$

- b)  $O(N \log N)$
- c)  $O(\log N)$
- d)  $O(N^2)$

**Answer:** d

**Explanation:** Mathematically, the run time of Euler's circuit problem is determined to be  $O(N^2)$ .

7. To which class does the Euler's circuit problem belong?

- a) P class
- b) NP class
- c) Partition class
- d) Complete class

**Answer:** a

**Explanation:** Euler's circuit problem can be solved in polynomial time. It can be solved in  $O(N^2)$ .

8. Halting problem is an example for?

- a) decidable problem
- b) undecidable problem
- c) complete problem
- d) trackable problem

**Answer:** b

**Explanation:** Halting problem by Alan Turing cannot be solved by any algorithm. Hence, it is undecidable.

9. How many stages of procedure does a non-deterministic algorithm consist of?

- a) 1
- b) 2
- c) 3
- d) 4

**Answer:** b

**Explanation:** A non-deterministic algorithm is a two-stage procedure- guessing stage and verification stage.

10. A non-deterministic algorithm is said to be non-deterministic polynomial if the time-efficiency of its verification stage is polynomial.

- a) true
- b) false

**Answer:** a

**Explanation:** One of the properties of NP class problems states that A non-deterministic algorithm is said to be non-deterministic polynomial if the time-efficiency of its verification stage is polynomial.

11. How many conditions have to be met if an NP- complete problem is polynomially reducible?

- a) 1
- b) 2
- c) 3
- d) 4

**Answer:** b

**Explanation:** A function  $t$  that maps all yes instances of decision problems  $D_1$  and  $D_2$  and  $t$  should be computed in polynomial time are the two conditions.

12. To which of the following class does a CNF-satisfiability problem belong?

- a) NP class
- b) P class
- c) NP complete
- d) NP hard

**Answer:** c

**Explanation:** The CNF satisfiability problem belongs to NP complete class. It deals with Boolean expressions.

13. How many steps are required to prove that a decision problem is NP complete?

- a) 1
- b) 2
- c) 3
- d) 4

**Answer:** b

**Explanation:** First, the problem should be NP. Next, it should be proved that every problem in NP is reducible to the problem in question in polynomial time.

14. Which of the following problems is not NP complete?
- a) Hamiltonian circuit
  - b) Bin packing
  - c) Partition problem
  - d) Halting problem

**Answer:** d

**Explanation:** Hamiltonian circuit, bin packing, partition problems are NP complete problems. Halting problem is an undecidable problem.

15. The choice of polynomial class has led to the development of an extensive theory called

- 
- a) computational complexity
  - b) time complexity
  - c) problem complexity
  - d) decision complexity

**Answer:** a

**Explanation:** An extensive theory called computational complexity seeks to classify problems according to their inherent difficulty.

---

1. Which of the following algorithm can be used to solve the Hamiltonian path problem efficiently?

- a) branch and bound
- b) iterative improvement
- c) divide and conquer
- d) greedy algorithm

**Answer:** a

**Explanation:** The Hamiltonian path problem can be solved efficiently using branch and bound approach. It can also be solved using a backtracking approach.

2. The problem of finding a path in a graph that visits every vertex exactly once is called?

- a) Hamiltonian path problem
- b) Hamiltonian cycle problem
- c) Subset sum problem
- d) Turnpike reconstruction problem

**Answer:** a

**Explanation:** Hamiltonian path problem is a problem of finding a path in a graph that visits every node exactly once whereas Hamiltonian cycle problem is finding a cycle in a graph.

3. Hamiltonian path problem is \_\_\_\_\_

- a) NP problem
- b) N class problem
- c) P class problem
- d) NP complete problem

**Answer:** d

**Explanation:** Hamiltonian path problem is found to be NP complete. Hamiltonian cycle problem is also an NP- complete problem.

4. There is no existing relationship between a Hamiltonian path problem and Hamiltonian circuit problem.

- a) true
- b) false

**Answer:** b

**Explanation:** There is a relationship between Hamiltonian path problem and Hamiltonian circuit problem. The Hamiltonian path in graph G is equal to Hamiltonian cycle in graph H under certain conditions.

5. Which of the following problems is similar to that of a Hamiltonian path problem?

- a) knapsack problem
- b) closest pair problem
- c) travelling salesman problem
- d) assignment problem

**Answer:** c

**Explanation:** Hamiltonian path problem is similar to that of a travelling salesman problem since both the problem traverses all the nodes in a graph exactly once.

6. Who formulated the first ever algorithm for solving the Hamiltonian path problem?

- a) Martello
- b) Monte Carlo

- c) Leonard
- d) Bellman

**Answer:** a

**Explanation:** The first ever problem to solve the Hamiltonian path was the enumerative algorithm formulated by Martello.

7. In what time can the Hamiltonian path problem can be solved using dynamic programming?

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

**Answer:** d

**Explanation:** Using dynamic programming, the time taken to solve the Hamiltonian path problem is mathematically found to be  $O(N^2 2^N)$ .

8. In graphs, in which all vertices have an odd degree, the number of Hamiltonian cycles through any fixed edge is always even.

- a) true
- b) false

**Answer:** a

**Explanation:** According to a handshaking lemma, in graphs, in which all vertices have an odd degree, the number of Hamiltonian cycles through any fixed edge is always even.

9. Who invented the inclusion-exclusion principle to solve the Hamiltonian path problem?

- a) Karp
- b) Leonard Adleman
- c) Andreas Björklund
- d) Martello

**Answer:** c

**Explanation:** Andreas Björklund came up with the inclusion-exclusion principle to reduce the counting of number of Hamiltonian cycles.

10. For a graph of degree three, in what time can a Hamiltonian path be found?

- a)  $O(0.251^n)$
- b)  $O(0.401^n)$
- c)  $O(0.167^n)$
- d)  $O(0.151^n)$

**Answer:** a

**Explanation:** For a graph of maximum degree three, a Hamiltonian path can be found in time  $O(0.251^n)$ .

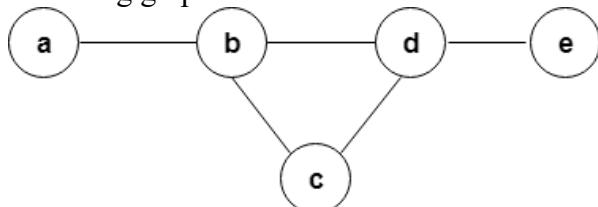
11. What is the time complexity for finding a Hamiltonian path for a graph having N vertices (using permutation)?

- a)  $O(N!)$
- b)  $O(N! * N)$
- c)  $O(\log N)$
- d)  $O(N)$

**Answer:** b

**Explanation:** For a graph having N vertices traverse the permutations in  $N!$  iterations and it traverses the permutations to see if adjacent vertices are connected or not takes N iterations (i.e.)  $O(N! * N)$ .

12. How many Hamiltonian paths does the following graph have?

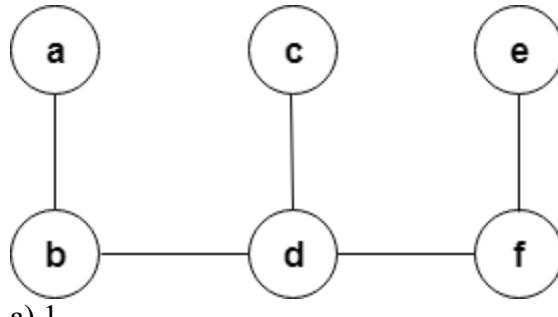


- a) 1
- b) 2
- c) 3
- d) 4

**Answer:** a

**Explanation:** The above graph has only one Hamiltonian path that is from a-b-c-d-e.

13. How many Hamiltonian paths does the following graph have?



- a) 1
- b) 2
- c) 0
- d) 3

**Answer:** c

**Explanation:** The above graph has no Hamiltonian paths. That is, we cannot traverse the graph with meeting vertices exactly once.

1. Under what condition any set A will be a subset of B?

- a) if all elements of set B are also present in set A
- b) if all elements of set A are also present in set B
- c) if A contains more elements than B
- d) if B contains more elements than A

**Answer:** b

**Explanation:** Any set A will be called a subset of set B if all elements of set A are also present in set B. So in such a case set A will be a part of set B.

2. What is a subset sum problem?

- a) finding a subset of a set that has sum of elements equal to a given number
- b) checking for the presence of a subset that has sum of elements equal to a given number and printing true or false based on the result
- c) finding the sum of elements present in a set
- d) finding the sum of all the subsets of a set

**Answer:** b

**Explanation:** In subset sum problem check for the presence of a subset that has sum of elements equal to a given number. If such a

subset is present then we print true otherwise false.

3. Which of the following is true about the time complexity of the recursive solution of the subset sum problem?

- a) It has an exponential time complexity
- b) It has a linear time complexity
- c) It has a logarithmic time complexity
- d) it has a time complexity of  $O(n^2)$

**Answer:** a

**Explanation:** Subset sum problem has both recursive as well as dynamic programming solution. The recursive solution has an exponential time complexity as it will require to check for all subsets in worst case.

4. What is the worst case time complexity of dynamic programming solution of the subset sum problem(sum=given subset sum)?

- a)  $O(n)$
- b)  $O(\text{sum})$
- c)  $O(n^2)$
- d)  $O(\text{sum} * n)$

**Answer:** d

**Explanation** Subset sum problem has both recursive as well as dynamic programming solution. The dynamic programming solution has a time complexity of  $O(n * \text{sum})$  as it is a nested loop with limits from 1 to n and 1 to sum respectively.

5. Subset sum problem is an example of NP-complete problem.

- a) true
- b) false

**Answer:** a

**Explanation:** Subset sum problem takes exponential time when we implement a recursive solution. Subset sum problem is known to be a part of NP complete problems.

6. Recursive solution of subset sum problem is faster than dynamic problem solution in terms of time complexity.

- a) true
- b) false

**Answer:** b

**Explanation:** The recursive solution to subset sum problem takes exponential time complexity whereas the dynamic programming solution takes polynomial time complexity. So dynamic programming solution is faster in terms of time complexity.

7. Which of the following is not true about subset sum problem?

- a) the recursive solution has a time complexity of  $O(2n)$
- b) there is no known solution that takes polynomial time
- c) the recursive solution is slower than dynamic programming solution
- d) the dynamic programming solution has a time complexity of  $O(n \log n)$

**Answer:** d

**Explanation:** Recursive solution of subset sum problem is slower than dynamic problem solution in terms of time complexity.

Dynamic programming solution has a time complexity of  $O(n * \text{sum})$ .

8. Which of the following should be the base case for the recursive solution of subset sum problem?

- a)

```
if(sum==0)
```

```
return true;
```

- b)

```
if(sum==0)
```

```
return true;
```

```
if (n ==0 && sum!= 0)
```

```
return false;
```

- c)

```

if (n == 0 && sum != 0)
    return false;

d)

if(sum<0)
    return true;

if (n == 0 && sum!= 0)
    return false;

```

**Answer: b**

**Explanation:** The base case condition defines the point at which the program should stop recursion. In this case we need to make sure that, the sum does not become 0 and there should be elements left in our array for recursion to happen.

9. What will be the output for the following code?

```

#include <stdio.h>
bool func(int arr[], int n, int sum)
{
    if (sum == 0)
        return true;
    if (n == 0 && sum != 0)
        return false;
    if (arr[n-1] > sum)
        return func(arr, n-1, sum);

    return func(arr, n-1, sum) || func(ar
    r, n-1, sum-arr[n-1]);
}
int main()
{
    int arr[] = {4,6, 12, 2};
    int sum = 12;
    int n = sizeof(arr)/sizeof(arr[0]);
    if (func(arr, n, sum) == true)
        printf("true");
    else
        printf("false");
    return 0;
}

```

- a) 12
- b) 4 6 2

- c) True
- d) False

**Answer: c**

**Explanation:** The given code represents the recursive approach of solving the subset sum problem. The output for the code will be true if any subset is found to have sum equal to the desired sum, otherwise false will be printed.

10. What will be the output for the following code?

```

#include <stdio.h>
bool func(int arr[], int n, int sum)
{
    bool subarr[n+1][sum+1];
    for (int i = 0; i <= n; i++)
        subarr[i][0] = true;
    for (int i = 1; i <= sum; i++)
        subarr[0][i] = false;
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= sum; j++)
        {
            if(j<arr[i-1])
                subarr[i][j] = subarr[i-1
                ][j];
            if (j >= arr[i-1])
                subarr[i][j] = subarr[i-1
                ][j] ||
                subarr[i - 1][j-arr[i-1]];
        }
    }
    return subarr[n][sum];
}

int main()
{
    int arr[] = {3, 3, 4, 4, 7};
    int sum = 5;
    int n = sizeof(arr)/sizeof(arr[0]);
    if (func(arr, n, sum) == true)
        printf("true");
    else
        printf("false");
    return 0;
}

```

- a) true
- b) false

- c) 0
- d) error in code

**Answer:** b

**Explanation:** The given code represents the dynamic programming approach of solving the subset sum problem. The output for the code will be true if any subset is found to have sum equal to the desired sum, otherwise false will be printed.

11. What will be the worst case time complexity for the following code?

```
#include <stdio.h>
bool func(int arr[], int n, int sum)
{
    if (sum == 0)
        return true;
    if (n == 0 && sum != 0)
        return false;
    if (arr[n-1] > sum)
        return func(arr, n-1, sum);
    return func(arr, n-1, sum) || func(ar
r, n-1, sum-arr[n-1]);
}
int main()
{
    int arr[] = {4,6, 12, 2};
    int sum = 12;
    int n = sizeof(arr)/sizeof(arr[0]);
    if (func(arr, n, sum) == true)
        printf("true");
    else
        printf("false");
    return 0;
}
```

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

**Answer:** c

**Explanation:** The given code represents the recursive approach solution of the subset sum problem. It has an exponential time complexity as it will require to check for all subsets in the worst case. It is equal to  $O(2^n)$ .

1. What is meant by the power set of a set?
- a) subset of all sets
- b) set of all subsets
- c) set of particular subsets
- d) an empty set

**Answer:** b

**Explanation:** Power set of a set is defined as the set of all subsets. Ex- if there is a set  $S= \{1,3\}$  then power set of set  $S$  will be  $P=\{\{\}, \{1\}, \{3\}, \{1,3\}\}$ .

2. What is the set partition problem?
- a) finding a subset of a set that has sum of elements equal to a given number
- b) checking for the presence of a subset that has sum of elements equal to a given number
- c) checking whether the set can be divided into two subsets of with equal sum of elements and printing true or false based on the result
- d) finding subsets with equal sum of elements

**Answer:** c

**Explanation:** In set partition problem we check whether a set can be divided into 2 subsets such that the sum of elements in each subset is equal. If such subsets are present then we print true otherwise false.

3. Which of the following is true about the time complexity of the recursive solution of set partition problem?
- a) It has an exponential time complexity
- b) It has a linear time complexity
- c) It has a logarithmic time complexity
- d) it has a time complexity of  $O(n^2)$

**Answer:** a

**Explanation:** Set partition problem has both recursive as well as dynamic programming solution. The recursive solution has an exponential time complexity as it will require to check for all subsets in the worst case.

4. What is the worst case time complexity of dynamic programming solution of set partition problem( $\text{sum}=\text{sum of set elements}$ )?

- a)  $O(n)$
- b)  $O(\text{sum})$
- c)  $O(n^2)$
- d)  $O(\text{sum} \cdot n)$

**Answer:** d

**Explanation:** Set partition problem has both recursive as well as dynamic programming solution. The dynamic programming solution has a time complexity of  $O(n \cdot \text{sum})$  as it has a nested loop with limits from 1 to n and 1 to sum respectively.

5. Set partition problem is an example of NP complete problem.

- a) true
- b) false

**Answer:** a

**Explanation:** Set partition problem takes exponential time when we implement a recursive solution. Set partition problem is known to be a part of NP complete problems.

6. Recursive solution of Set partition problem is faster than dynamic problem solution in terms of time complexity.

- a) true
- b) false

**Answer:** b

**Explanation:** The recursive solution to set partition problem takes exponential time complexity whereas the dynamic programming solution takes polynomial time complexity. So dynamic programming solution is faster in terms of time complexity.

7. Which of the following is not true about set partition problem?

- a) the recursive solution has a time complexity of  $O(2n)$
- b) there is no known solution that takes polynomial time
- c) the recursive solution is slower than dynamic programming solution
- d) the dynamic programming solution has a time complexity of  $O(n \log n)$

**Answer:** d

**Explanation:** Recursive solution of set partition problem is slower than dynamic problem solution in terms of time complexity. Dynamic programming solution has a time complexity of  $O(n \cdot \text{sum})$ .

8. Which of the following should be the base case for the recursive solution of a set partition problem?

a)

```
If(sum%2!=0)
    return false;
if(sum==0)
    return true;
```

b)

```
If(sum%2!=0)
    return false;
if(sum==0)
    return true;
if (n ==0 && sum!= 0)
    return false;
```

c)

```
if (n ==0 && sum!= 0)
    return false;
```

d)

```
if(sum<0)
    return true;
if (n ==0 && sum!= 0)
    return false;
```

**Answer:** b

**Explanation:** In this case, we need to make sure that, the sum does not become 0 and

there should be elements left in our array for recursion to happen. Also if the sum of elements of the set is an odd number then that set cannot be partitioned into two subsets with an equal sum so under such a condition false should be returned.

9. What will be the output for the given code?

```
#include <stdio.h>
#include <stdbool.h>
bool func1(int arr[], int n, int sum)
{
    if (sum == 0)
        return true;
    if (n == 0 && sum != 0)
        return false;
    if (arr[n-1] > sum)
        return func1(arr, n-1, sum);
    return func1(arr, n-1, sum) || func1(
        arr, n-1, sum-arr[n-1]);
}
bool func (int arr[], int n)
{
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += arr[i];
    if (sum%2 != 0)
        return false;
    return func1 (arr, n, sum/2);
}
int main()
{
    int arr[] = {4,6, 12, 2};
    int n = sizeof(arr)/sizeof(arr[0]);
    if (func(arr, n) == true)
        printf("true");
    else
        printf("false");
    return 0;
}
```

- a) true
- b) false
- c) 4 6 2
- d) 12

**Answer:** a

**Explanation:** The given code represents the recursive approach of solving the set partition problem. The code checks whether a set can

be divided into 2 subsets such that the sum of elements in each subset is equal. If such a partition is possible then we print true otherwise false. In this case true should be printed.

10. What will be the output for the given code?

```
#include <stdio.h>
bool func (int arr[], int n)
{
    int sum = 0;
    int i, j;
    for (i = 0; i < n; i++)
        sum += arr[i];
    if (sum%2 != 0)
        return false;
    bool partition[sum/2+1][n+1];
    for (i = 0; i <= n; i++)
        partition[0][i] = true;
    for (i = 1; i <= sum/2; i++)
        partition[i][0] = false;
    for (i = 1; i <= sum/2; i++)
    {
        for (j = 1; j <= n; j++)
        {
            partition[i][j] = partition[i][j-1];
            if (i >= arr[j-1])
                partition[i][j] = partition[i][j-1] ||
                    partition[i - arr[j-1]][j-1];
        }
    }
    return partition[sum/2][n];
}
int main()
{
    int arr[] = {3, 3, 4, 4, 7};
    int n = sizeof(arr)/sizeof(arr[0]);
    if (func(arr, n) == true)
        printf("true");
    else
        printf("false");
    return 0;
}
```

- a) true
- b) false
- c) 0
- d) error

**Answer:** b

**Explanation:** The given code represents the

dynamic programming approach of solving set partition problem. The code checks whether a set can be divided into 2 subsets such that the sum of elements in each subset is equal. If such a partition is possible then we print true otherwise false. In this case, false should be printed.

11. What will be the auxiliary space complexity of dynamic programming solution of set partition problem(sum=sum of set elements)?

- a)  $O(n \log n)$

- b)  $O(n^2)$
- c)  $O(2^n)$
- d)  $O(\text{sum} * n)$

*Answer:* d

*Explanation:* The auxiliary space complexity of set partition problem is required in order to store the partition table. It takes up a space of  $n * \text{sum}$ , so its auxiliary space requirement becomes  $O(n * \text{sum})$ .

---

| <b>PART – B (LONGANSWER QUESTIONS)</b> |                                                                                                        |            |    |
|----------------------------------------|--------------------------------------------------------------------------------------------------------|------------|----|
| <b>1</b>                               | <b>Explain</b> the principle of FIFO branch and bound                                                  | Apply      | 11 |
| <b>2</b>                               | <b>Explain</b> the method of reduction to solve travelling sales person problem using branch and bound | Apply      | 11 |
| <b>3</b>                               | <b>Write</b> non deterministic algorithm for sorting and searching                                     | Understand | 12 |
| <b>5</b>                               | <b>What is</b> chromatic number decision problem and clique decision problem                           | Apply      | 12 |

## **XI. OBJECTIVE QUESTIONS: JNTUH**

### **UNIT-I**

1. In analysis of algorithm, approximate relationship between the size of the job and the amount of work required to do is expressed by using \_\_\_\_\_

- (a) Central tendency (b) Differential equation (c) Order of execution (d) Order of magnitude (e) Order of Storage.

**Ans :Order of execution**

2. Worst case efficiency of binary search is

- (a)  $\log_2 n + 1$  (b)  $n$  (c)  $N^2$  (d)  $2n$  (e)  $\log n$ .

**Ans : $\log_2 n + 1$**

3. For analyzing an algorithm, which is better computing time?

- (a)  $O(100 \log N)$  (b)  $O(N)$  (c)  $O(2N)$  (d)  $O(N \log N)$  (e)  $O(N^2)$ .

**Ans : $O(100 \log N)$**

4. Consider the usual algorithm for determining whether a sequence of parentheses is balanced. What is the maximum number of parentheses that will appear on the stack AT ANY ONE TIME when the algorithm analyzes: ((())(()))

- (a) 1 (b) 2 (c) 3 (d) 4

**Ans :3**

5. Breadth first search \_\_\_\_\_

- (a) Scans each incident node along with its children. (b) Scans all incident edges before moving to other node. (c) Is same as backtracking (d) Scans all the nodes in random order.

**Ans :Scans all incident edges before moving to other node.**

6. Which method of traversal does not use stack to hold nodes that are waiting to be processed?

- (a) Dept First (b) D-search (c) Breadth first (d) Back-tracking

**Ans :Breadth first**

7. The Knapsack problem where the objective function is to minimize the profit is \_\_\_\_\_

- (a) Greedy (b) Dynamic 0 / 1 (c) Back tracking (d) Branch & Bound 0/1

**Ans :Branch & Bound 0/1**

8. Choose the correct answer for the following statements:

I. The theory of NP-completeness provides a method of obtaining a polynomial time for NP algorithms.

II. All NP-complete problems are NP-Hard.

- (a) I is FALSE and II is TRUE (b) I is TRUE and II is FALSE (c) Both are TRUE (d) Both are FALSE

**Ans :I is FALSE and II is TRUE**

9. If all  $c(i, j)$ 's and  $r(i, j)$ 's are calculated, then OBST algorithm in worst case takes one of the following time.

- (a)  $O(n \log n)$  (b)  $O(n^3)$  (c)  $O(n^2)$  (d)  $O(\log n)$  (e)  $O(n^4)$ .

**Ans : $O(n^3)$**

10. The upper bound on the time complexity of the nondeterministic sorting algorithm is

- (a)  $O(n)$  (b)  $O(n \log n)$  (c)  $O(1)$  (d)  $O(\log n)$  (e)  $O(n^2)$ .

**Ans:  $O(n)$**

11. The worst case time complexity of the nondeterministic dynamic knapsack algorithm is  
(a)  $O(n \log n)$  (b)  $O(\log n)$  (c)  $O(n^2)$  (d)  $O(n)$  (e)  $O(1)$ .

**Ans :O(n)**

12. Recursive algorithms are based on

- (a) Divideand conquer approach (b) Top-down approach (c) Bottom-up approach (d)  
Hierarchical approach

**Ans :Bottom-up approach**

13. What do you call the selected keys in the quick sort method?

- (a) Outer key (b)Inner Key (c) Partition key(d) Pivot key (e) Recombine key.

**Ans :c**

14. How do you determine the cost of a spanning tree?

- (a) By the sum of the costs of the edges of the tree (b) By the sum of the costs of the edges and vertices of the tree  
(c) By the sum of the costs of the vertices of the tree (d) By the sum of the costs of the edges of the graph  
(e) By the sum of thecosts of the edges and vertices of the graph.

**Ans :By the sum of the costs of the edges of the tree8.**

15. The time complexity of the normal quick sort, randomized quick sort algorithms in the worst case is

- (a)  $O(n^2)$ ,  $O(n \log n)$  (b)  $O(n^2)$ ,  $O(n^2)$  (c)  $O(n \log n)$ ,  $O(n^2)$  (d)  $O(n \log n)$ ,  $O(n \log n)$  (e)  
 $O(n \log n)$ ,  $O(n^2 \log n)$ .

**Ans :O(n2), O(n2)**

16. Let there be an array of length 'N', and the selection sort algorithm is used to sort it, how many times a swap function is called to complete the execution?

- (a)  $N \log N$  times (b)  $\log N$  times (c)  $N^2$  times (d)  $N-1$  times (e)  $N$  times.

**Ans :N-1 times**

17. The Sorting methodwhich is used for external sort is

- (a) Bubble sort (b) Quick sort (c) Merge sort (d) Radix sort (e) Selection sort.

**Ans :Radix sort**

18. The graph colouringalgorithm's time can be bounded by \_\_\_\_\_

- (a)  $O(mnm)$  (b)  $O(nm)$  (c)  $O(nm \cdot 2n)$  (d)  $O(nmn)$ .

**Ans :O(nmn).**

19. Sorting is not possible by using which of the following methods?

- (a) Insertion (b) Selection (c) Deletion (d) Exchange

**Ans :Deletion**

20. What is the type of the algorithm used in solving the 8 Queens problem?

- (a)Backtracking (b) Dynamic (c) Branch and Bound (d) DandC

**Ans :Backtracking**

## UNIT-II

1. Name the node which has been generated but none of its children nodes have been generated in state space tree of backtracking method.

- (a) Dead node (b) Live node (c) E-Node (d) State Node

**Ans: Livenode**

2. How many nodes are there in a full state space tree with  $n = 6$ ?

- (a) 65 (b) 64 (c) 63 (d) 32

**Ans : 63**

3. This algorithm scans the list by swapping the entries whenever pair of adjacent keys are out of desired order.

- (a) Insertion sort. (b) Bubble sort. (c) Shell sort. (d) Quick sort.

**Ans: Bubble sort.**

5. From the following chose the one which belongs to the algorithm paradigm other than to which others from the following belongs to.

- (a) Minimum & Maximum problem. (b) Knapsack problem. (c) Selection problem.(d) Merge sort.

**Ans: Knapsack problem.**

6. To calculate  $c(i, j)$ 's,  $w(i, j)$ 's and  $r(i, j)$ 's; the OBST algorithm in worst case takes the following time.

- (a)  $O(\log n)$  (b)  $O(n^4)$  (c)  $O(n^3)$  (d)  $O(n \log n)$

**Ans: O (n3)**

7. What is the type of the algorithm used in solving the 4 Queens problem?

- (a) Greedy (b) Dynamic (c) Branch and Bound (d) Backtracking.

**Ans: Backtracking.**

8. In Knapsack problem, the best strategy to get the optimal solution, where  $P_i$ ,  $W_i$  is the Profit, Weight associated with each of the  $X_i$  object respectively is to

- (a) Arrange the values  $P_i/W_i$  in ascending order (b) Arrange the values  $P_i/X_i$  in ascending order  
(c) Arrange the values  $P_i/W_i$  in descending order (d) Arrange the values  $P_i/X_i$  in descending order

**Ans: Arrange the values  $P_i/X_i$  in descending order**

9. Greedy job scheduling with deadlines algorithms' complexity is defined as

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

**Ans: O(N)**

12. From the following choose the one which belongs to the algorithm paradigm other than to which others from the following belongs to.

- (a) Minimum & Maximum problem (b) Knapsack problem (c) Selection problem (d) Merge sort

**Ans : Knapsack problem**

14. Identify the name of the sorting in which time is not proportional to  $n^2$ .

- (a) Selection sort (b) Bubble sort (c) Quicik sort (d) Insertion sort.

**Ans : Insertion sort**

15. The optimal solution to a problem is a combination of optimal solutions to its subproblems. This is known as

- (a) Principle of Duality (b) Principle of Feasibility (c) Principle of Optimality (d) Principle of Dynamicity.

**Ans : Principle of Optimality**

16. Which of the following versions of merge sort algorithm does uses space efficiently?

- (a) Contiguous version (b) Array version (c) Linked version (d) Structure version (e) Heap version.

**Ans : Linked version**

17. Identify the correct problem for multistage graph from the list given below.

- (a) Resource allocation problem (b) Traveling salesperson problem  
(c) Producer consumer problem (d) Barber's problem

**Ans : Resource allocation problem**

18. How many edges are there in a Hamiltonian cycle if the edge cost is 'c' and the cost of cycle is ' $cn$ '

- (a)c (b)  $cn$  (c) $n$  (d)  $2c$

**Ans :n.**

19. A problem L is NP-complete iff L is NP-hard and

- (a)  $L \approx NP$  (b)  $L \alpha NP$  (c)  $L \in NP$  (d)  $L = NP$

**Ans :  $L \in NP$**

20. What would be the cost value for any answering node of a sub tree with root 'r' using branch-bound algorithm?

- (a) Maximum (b) Minimum (c) Optimal (d) Average

**Ans: Minimum**

UNIT-III

1. From the following pick the one which does not belongs to the same paradigm to which others belongs to.

(a) Minimum & Maximum problem                (b) Knapsack problem  
(c) Selection problem                (d) Merge sort

**Ans:Knapsack problem**

2. Prims algorithm is based on \_\_\_\_\_ method

  - a. Divide and conquer method c. Dynamic programming
  - b. Greedy method d. Branch and bound

3. Graph Method

Ans. Greedy Method

3. The amount of memory needs to run to completion is known as \_\_\_\_\_

  - a. Space complexity c. Worst case
  - b. Time complexity d. Best case

**Ans: Space complexity**

4. The amount of time needs to run to completion is known as \_\_\_\_\_

  - a. Space complexity c. Worst case
  - b. Time complexity d. Best case

Ans: Time complexity

- Ans. Time complexity

5. \_\_\_\_\_ is the minimum number of steps that can be executed for the given parameters

  - a. Average case c. Worst case
  - b. Time complexity d. Best case

**Ans: Best case**

6. \_\_\_\_\_ is the maximum number of steps that can be executed for the given parameters

  - a. Average case c. Worst case
  - b. Time complexity d. Best case

Ans: Worst case

- Ans: Worst case

7. \_\_\_\_\_ is the average number of steps that can be executed for the given parameters  
a. Average case c. Worst case  
b. Time complexity d. Best case

**Ans: Average Case**

8. Testing of a program consists of 2 phases which are \_\_\_\_\_ and \_\_\_\_\_

- a. Average case & Worst case b. Time complexity & Space complexity
  - c. Validation and checking errors d. Debugging and profiling

**Ans: Debugging and profiling**

9. Worst case time complexity of binary search is \_\_\_\_\_  
a. O(n) b. O(logn)c.  $\Theta(n\log n)$  d.  $\Theta(\log n)$

Ans:  $\Theta(\log n)$

10. Best case time complexity of binary search is \_\_\_\_\_

  - a. O(n) c.  $\Theta(n \log n)$
  - b. O( $\log n$ ) d.  $\Theta(\log n)$

Ans:  $\Theta(\log n)$

11. Average case time complexity of binary search is \_\_\_\_\_

  - a. O(n) c.  $\Theta(n \log n)$
  - b.  $O(\log n)$  d.  $\Theta(\log n)$

b.  $\Theta(\log n)$  u.

12. Merge sort invented by \_\_\_\_\_

- a. CARHOARE c. HAMILTON
- b. JOHN VON NEUMANN d. STRASSEN

**Ans : JOHN VON NEUMANN**

13. Quick sort invented by \_\_\_\_\_

- a. CARHOARE c. HAMILTON
- b. JOHN VON NEUMANN d. STRASSEN

**Ans : CARHOARE**

14. Worst case time complexity of Quick sort is \_\_\_\_\_

- a.  $O(n^2 \log 7)$  c.  $O(n \log n)$
- b.  $O(n^2)$  d.  $O(\log n)$

**Ans :  $O(n^2)$**

15. Best case time complexity of Quick sort is \_\_\_\_\_

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

**Ans :  $O(n \log n)$**

16. Average case time complexity of Quick sort is \_\_\_\_\_

- a.  $\Theta(n \log n)$  b.  $O(\log n)$  c.  $O(n \log n)$  d.  $\Theta(\log n)$

17. Which design strategy stops the execution when it finds the solution otherwise starts the problem from top

- a. Back tracking c. Divide and conquer
- b. Branch and Bound d. Dynamic programming

**Ans: Back Tracking**

18. Graphical representation of algorithm is \_\_\_\_\_

- a. Pseudo-code c. Graph Coloring
- b. Flow Chart d. Dynamic programming

**Ans: Flow Chart**

19. In pseudo-code conventions input express as \_\_\_\_\_

- a. input c. Read
- b. Write d. Return

**Ans : Write**

20. In pseudo-code conventions output express as \_\_\_\_\_

- a. input c. Read
- b. Write d. Return

**Ans : Read**

#### **UNIT-IV**

1. Tight bound is denoted as \_\_\_\_\_

- a.  $\Omega$  c.  $\Theta$
- b.  $\Omega$  d.  $O$

**Ans :  $\Theta$**

2. Upper bound is denoted as \_\_\_\_\_

- a.  $\Omega$  c.  $\Theta$
- b.  $\omega$  d.  $O$

**Ans :  $O$**

3. lower bound is denoted as \_\_\_\_\_

- a.  $\Omega$  c.  $\Theta$
- b.  $\omega$  d.  $O$

**Ans :  $\Omega$**

4. The function  $f(n)=o(g(n))$  if and only if Limit  $f(n)/g(n)=0$  as  $n \rightarrow \infty$

- a. Little oh b. Little omega

b. Big oh d. Omega

**Ans : Little oh**

5. The function  $f(n)=o(g(n))$  if and only if  $\lim_{n \rightarrow \infty} g(n)/f(n)=0$

a. Little oh b. Little omega

b. Big oh d. Omega

**Ans : Little omega**

6. The general criteria of algorithm; zero or more quantities are externally supplied is \_\_\_\_\_

a. Output b. Finiteness

b. Effectiveness d. Input

**Ans : Input**

7. The general criteria of algorithm; at least one quantity is produced \_\_\_\_\_

a. Output b. Finiteness

b. Effectiveness d. Input

**Ans : Output**

8. The general criteria of algorithm; Each instruction is clear and unambiguous \_\_\_\_\_

a. Output b. Definiteness

b. Effectiveness d. Input

**Ans : Definiteness**

9. The general criteria of algorithm; algorithm must terminates after a finite number of steps \_\_\_\_\_

a. Output b. Finiteness

b. Effectiveness d. Input

**Ans : Finiteness**

10. Which is not a criteria of algorithm

a. Input b. Output

b. Time complexity d. Best case

**Ans : Best case**

11. Which is not in general criteria of algorithm

a. Input b. Output

b. Time complexity d. Effectiveness

**Ans : Time complexity**

12. Time complexity of given algorithm

Algorithm Display(A)

{

S:=0.0;

For i:=0 to n-1

{

S:=S+A[i];

Return S;

}

}

a.  $4n+4$  c.  $4n^2+4$

b.  $2n^2+2n+2$  d.  $4n+4$

**Ans : 4n+4**

13. Time complexity of given algorithm

AlgorithmSum(A,S)

{

for i:=1 to n-1

{

for j:=2 to n-1

```

{
S:=S+i+j;
return S;
}
}
}

a.  $6n^2 - 14n + 4$  c.  $4n^2 + 6n + 12$   

b.  $6n^2 + 14n + 10$  d.  $6n^2 - 14n + 10$ 

```

**Ans : $6n^2 - 14n + 10$**

14. Kruskal algorithm is based on \_\_\_\_\_ method  
 a. Divide and conquer method b. Greedy method c. Dynamic programming d. Branch and bound

**Ans. Greedy method**

15. Prims algorithm is based on \_\_\_\_\_ method  
 a. Divide and conquer method c. Dynamic programming  
 b. Greedy method d. Branch and bound

**Ans. Greedy Method**

16. The output of Kruskal and Prims algorithm is \_\_\_\_\_  
 a. Maximum spanning tree c. Spanning tree  
 b. Minimum spanning tree d. None of these

**UNIT-V**

1. Job sequencing with deadline is based on \_\_\_\_\_ method  
 a. greedy method c. branch and bound  
 b. dynamic programming d. divide and conquer

**Ans. Greedy method**

2. Fractional knapsack is based on \_\_\_\_\_ method  
 a. greedy method c. branch and bound  
 b. dynamic programming d. divide and conquer

**Ans. Greedy method**

3. 0/1 knapsack is based on \_\_\_\_\_ method  
 a. greedy method c. branch and bound  
 b. dynamic programming d. divide and conquer

**Ans. Dynamic programming**

4. The files  $x_1, x_2, x_3$  are 3 files of length 30, 20, 10 records each. What is the optimal merge pattern value?  
 a. 110 c. 60  
 b. 90 d. 50

**Ans. 90**

5. The optimal merge pattern is based on \_\_\_\_\_ method  
 a. Greedy method b. Dynamic programming  
 c. Knapsack method d. Branch and bound

**Ans. Greedy method**

6. Who invented the word Algorithm  
 a. Abu Ja'far Mohammed ibn Musa c. Abu Mohammed Khan  
 b. Abu Jafar Mohammed Kasim d. Abu Ja'far Mohammed Ali Khan

**Ans. Abu Ja'far Mohammed ibn Musa**

7. In Algorithm comments begin with \_\_\_\_\_  
 a. /\* c. /  
 b. \*/ d. //

**Ans : //**

8. The \_\_\_\_\_ of an algorithm is the amount of memory it needs to run to completion.

- a. Space Complexity c. Best Case
- b. Time Complexity d. Worst Case

**Ans : Space Complexity**

9. \_\_\_\_\_ is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results.

- a. Debugging c. Combining
- b. Profiling d. Conquer

**Ans : Profiling**

10. In Algorithm Specification the blocks are indicated with matching \_\_\_\_\_

- a. Braces c. Square Brackets
- b. Parenthesis d. Slashes

**Ans : Braces**

11. Huffmancode are the applications of \_\_\_\_\_ with minimal weighted external path length obtained by an optimal set.

- a. BST b. MST
- c. Binary tree d. Weighted Graph

**Ans : Binary tree**

12. From the following which is not return optimal solution

- a. Dynamic programming c. Backtracking
- b. Branch and bound d. Greedy method

**Ans . Backtracking**

13. \_\_\_\_\_ is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions

- a. Dynamic programming c. Backtracking
- b. Branch and bound d. Greedy method

**Ans : Dynamic programming**

14. The name backtrack was first coined by \_\_\_\_\_

- a. D.H.Lehmer c. L.Baumert
- b. R.J.Walker d. S. Golomb

**Ans : D.H.Lehmer**

15. The term \_\_\_\_\_ refers to all state space search methods in which all children of the – nodes are generated before any other live node can become the E-node.

- a. Backtacking c. Depth First Search
- b. Branch and Bound d. Breadth First Search

**Ans ; Branch and Bound**

16. A \_\_\_\_\_ is a round trip path along n edges of G that visits every vertex once and returns to its starting position.

- a. MST c. TSP
- b. Multistage Graph d. Hamiltonian Cycle

**Ans :Hamiltonian Cycle**

17. Graph Coloring is which type of algorithm design strategy

- a. Backtacking c. Greedy
- b. Branch and Bound d. Dynamic programming

**Ans : Backtracking**

18. Which of the following is not a limitation of binary search algorithm?

- a. must use a sorted array
- b. requirement of sorted array is expensive when a lot of insertion and deletions are needed

- c. there must be a mechanism to access middle element directly
- d. binary search algorithm is not efficient when the data elements are more than 1000.

**Ans : binary search algorithm is not efficient when the data elements are more than 1000.**

- 19. Binary Search Algorithm cannot be applied to
  - a. Sorted linked list c. Sorted linear array
  - b. Sorted binary tree d. Pointer array

**Ans :Sorted linked list**

- 20. Two main measures for the efficiency of an algorithm are
  - a. Processor and memory c. Time and space
  - b. Complexity and capacity d. Data and space

**Ans : Time and Space**

## XII. GATE QUESTIONS:

1 The order of an internal node in a B+ tree index is the maximum number of children it can have. Suppose that a child pointer takes 6 bytes, the search field value takes 14 bytes, and the block size is 512 bytes. What is the order of the internal node?

- A) 24 B) 25 C) 26 D) 27

Answer : (C)

2 The best data structure to check whether an arithmetic expression has balanced parentheses is a

- A) queue B) stack C) tree D) list

Answer : (B)

3. A Priority-Queue is implemented as a Max-Heap. Initially, it has 5 elements. The level-order traversal of the heap is given below: 10, 8,5,3,2 Two new elements 1 and 7 are inserted in the heap in that order. The level-order traversal of the heap after the insertion of the elements is

- A) 10,8,7,5,3,2,1 B) 10,8,7,2,3,1,5 C) 10,8,7,1,2,3,5 D) 10,8,7,3,2,1,5

Answer : (D)

4 The following numbers are inserted into an empty binary search tree in the given order: 10, 1, 3, 5, 15, 12, 16. What is the height of the binary search tree (the height is the maximum distance of a leaf node from the root)?

- A) 2 B) 3 C) 4 D) 6

Answer : (B)

5 The goal of structured programming is to

- A) have well indented programs B) be able to infer the flow of control from the compiled code C) be able to infer the flow of control from the program text D) avoid the use of GOTO statements

Answer : (C)

6 The tightest lower bound on the number of comparisons, in the worst case, for comparison-based sorting is of the order of

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

Answer : (B)

7 Let  $G$  be a simple graph with 20 vertices and 100 edges. The size of the minimum vertex cover of  $G$  is 8. Then, the size of the maximum independent set of  $G$  is

- A) 12 B) 8 C) Less than 8 D) More than 12

Answer : (A)

8 Let  $A$  be a sequence of 8 distinct integers sorted in ascending order. How many distinct pairs of sequences,  $B$  and  $C$  are there such that (i) each is sorted in ascending order, (ii)  $B$  has 5 and  $C$  has 3 elements, and (iii) the result of merging  $B$  and  $C$  gives  $A$ ?

A) 2 B) 30 C) 56 D) 256

Answer : (D)

9 A Priority-Queue is implemented as a Max-Heap. Initially, it has 5 elements. The level-order traversal of the heap is given below: 10, 8,5,3,2 Two new elements 1 and 7 are inserted in the heap in that order. The level-order traversal of the heap after the insertion of the elements is

A) 10,8,7,5,3,2,1 B) 10,8,7,2,3,1,5 C) 10,8,7,1,2,3,5 D) 10,8,7,3,2,1,5

Answer : (D)

10 The S-N curve for steel becomes asymptotic nearly at

A)  $10^3$  cycles B)  $10^4$  cycles C)  $10^6$  cycles D)  $10^9$  cycles

Answer : (C)

### XIII. WEBSITES:

- [http://www.ki.inf.tu-dresden.de/~hans/www-adr/alg\\_course.html](http://www.ki.inf.tu-dresden.de/~hans/www-adr/alg_course.html) --String Matching, Sorting, Linear Programming
- [http://www.algorithmist.com/index.php/Main\\_Page](http://www.algorithmist.com/index.php/Main_Page) it contains dynamic programming, greedy, ,Graph Theory, sorting, Data Structures

<http://www-2.cs.cmu.edu/~guyb/realworld.html>- it contains Data Compression, Indexing and Search engines, linear Programming, Pattern matching

### XIV. EXPERT DETAILS:

Professor Sartaj Kumar Sahni is an Indian computer scientist, now based in the USA, and is one of the pioneers in the field of data structures. He is a distinguished professor in the Department of Computer and Information Science and Engineering at the University of Florida.

<http://www.cise.ufl.edu/~sahni/>

### XV. JOURNALS:

1. Journal of Graph Algorithms and Applications  
[url: http://www.emis.de/journals/JGAA/home.html](http://www.emis.de/journals/JGAA/home.html)
2. Algorithmica –A journal about the design of algorithms in many applied and fundamental areas <http://www.springerlink.com>

### XVI. LIST OF TOPICS FOR STUDENT SEMINARS:

- Randomized Algorithms, Binary search, Connected components and spanning Trees
- Hamiltonian cycles, Single source shortest path problem, Non-deterministic algorithms

### XVII. CASE STUDIES / SMALL PROJECTS:

#### **1. Techniques for Algorithm Design and Analysis: Case Study of a Greedy Algorithm.**

Six different implementations of a greedy dominating set algorithm are presented and analyzed. The implementations and analysis illustrate many of the important techniques in the design and analysis of algorithms, as well as some interesting graph theory.

#### **2. Scheduling Two Salesmen in a Network.**

The two-server problem is concerned with the movement of two servers to request points in a metric space. We consider an offline version of the problem in a graph in which the requests may be served in any order. A family of approximations algorithms is developed for this NP-complete problem.

# **UNIT IV**

## **INTRODUCTION TO BACKTRACKING- BRANCH AND BOUND**

### **Multiple Choice Questions with Answers:**

## **Multiple Choice Questions with Answers:**

- a) O(n)
  - b) O(sum)
  - c) O(n<sup>2</sup>)
  - d) O(sum\*n)

**CLO-3 Net Source**

- a) the recursive solution has a time complexity of  $O(2n)$
  - b) there is no known solution that takes polynomial time
  - c) the recursive solution is slower than dynamic programming solution
  - d) the dynamic programming solution has a time complexity of  $O(n \log n)$**

11. Which of the following algorithm can be used to solve the Hamiltonian path problem efficiently? **CLO-3 Pg No-531**

- a) Branch and Bound
  - b) Iterative improvement
  - c) Greedy algorithm
  - d) Divide and Conquer

12. The problem of finding a path in a graph that visits every vertex exactly once is called? **CLO-3 Pg No-466**

- a) Hamiltonian path problem**      b) Hamiltonian cycle problem  
**c) Subset sum problem**      d) NP problem

13. In what time can the Hamiltonian path problem can be solved using dynamic programming? **CLQ-3 Pg No-531**



14 Which of the following problems is similar to that of a Hamiltonian path problem?

### **CLO-3 Net Source**



15. What is the time complexity for finding a Hamiltonian path for a graph having N vertices (using permutation)? **CLO-3 Pg No- 531**



16. What is the time complexity of the brute force algorithm used to solve the Knapsack problem? CLO-3 Pg No- 496

- a) O(n) b) O(n!)
  - c) O(2n) d) O(n<sup>3</sup>)

**CLO-3 Pg No-497**



18. Which of the following methods can be used to solve the Knapsack problem? CLO-3  
Pg No- 496

- a) Brute force algorithm

- b) Recursion
  - c) Dynamic programming
  - d) Brute force, Recursion and Dynamic Programming**

19. Travelling salesman problem is an example of      **CLO-4 Pg No- 486**  
a.Dynamic Algorithm  
c.Recursive Approach  
b.Greedy Algorithm  
d.Divide & Conquer

20. A graph in which all nodes are of equal degree is called      **CLO-3 Net Source**  
a) Multi graph  
c) **Regular graph**  
b) Non regular graph  
d) Complete graph

21. Which of the following algorithms can be used to most efficiently determine the presence of a cycle in a given graph ? **CLO-3 Pg No-468**  
**a) Depth First Search**  
b) Breadth First Search  
c) Prim's Minimum Spanning Tree Algorithm  
d) Kruskal' Minimum Spanning Tree Algorithm

22. Traversal of a graph is different from tree because      **CLO-4 Pg No-469**  
**a) There can be a loop in graph so we must maintain a visited flag for every vertex**  
b) DFS of a graph uses stack, but in-order traversal of a tree is recursive  
c) BFS of a graph uses queue, but a time efficient BFS of a tree is recursive  
d) All of the above

23. Given two vertices in a graph s and t, which of the two traversals (BFS and DFS) can be used to find if there is path from **s to t?**      **CLO-4 Net Source**  
a) Only BFS  
c) **Both BFS and DFS**  
b) Only DFS  
d) Neither BFS nor DFS

24. A complete graph can have      **CLO-3 Pg No-466**  
a. $n^2$  spanning trees  
c. $n^{n+1}$  spanning trees  
b. $n^{(n-2)}$  spanning trees  
d. $n^n$  spanning trees

25. Graphs are represented using      **CLO-3 Net source**  
a.Adjacency tree  
c.Adjacency graph  
b.Adjacency linked list  
d.Adjacency queue

26. The spanning tree of connected graph with 10 vertices contains      **CLO-3 Net source**  
a.9 edges  
c.10 edges  
b.11 edges  
d. **9 vertices**

27. Which of the following algorithms solves the all-pair shortest path problem? **CLO-4 Pg No-478**  
**a.Floyd's algorithm**  
c.Dijkstra's algorithm  
b.Prim's algorithm  
d.Warshall's algorithm

28. The minimum number of colors needed to color a graph having n ( $>3$ ) vertices and 2 edges is **CLO-3 Net Source**

- a.1
- b.2**
- c.3
- d.4

29. Which of the following is useful in traversing a given graph by breadth first search?

**CLO-3 Net Source**

- a.set
- b.List**
- c.stacks
- d.Queue**

30. The minimum number of edges in a connected cyclic graph on n vertices is **CLO-3 Pg No- 466**

- a.n**
- b. $n+1$
- c. $n-1$
- d.none of the above

31. Floyd Warshall's Algorithm can be applied on \_\_\_\_\_ **CLO-3 Pg No-478**

- a) Undirected and unweighted graphs
- b) Undirected graphs
- c) Directed graphs**
- d) Acyclic graphs

32. What is the running time of the Floyd Warshall Algorithm? **CLO-3 Pg No-479**

- a) Big-oh( $V$ )
- b) Theta( $V^2$ )
- c) Big-Oh( $VE$ )**
- d) Theta( $V^3$ )

34. What procedure is being followed in Floyd Warshall Algorithm? **CLO-3 Pg No-479**

- a) Top down
- b) Bottom up**
- c) Big bang
- d) Sandwich

35. Floyd Warshall Algorithm can be used for finding \_\_\_\_\_ **CLO-3 Pg No-480**

- a) Single source shortest path
- b) Topological sort
- c) Minimum spanning tree
- d) Transitive closure**

36. What approach is being followed in Floyd Warshall Algorithm? **CLO-3 Pg No-480**

- a) Greedy technique
- b) Dynamic Programming**
- c) Linear Programming
- d) Backtracking

37. What happens when the value of k is 0 in the Floyd Warshall Algorithm? **Pg No-481**

- a) 1 intermediate vertex
- b) 0 intermediate vertex**
- c) N intermediate vertices
- d) N-1 intermediate vertices

38. The time required to find shortest path in a graph with n vertices and e edges is

**CLO-3 Net Source**

- a.  $O(e)$
- b.  $O(n)$**
- c.  $O(n^2)$**
- d.  $O(e^2)$

39. For 0/1 KNAPSACK problem, the algorithm takes \_\_\_\_\_ amount of time for memory table, and \_\_\_\_\_ time to determine the optimal load, for N objects and W as the capacity of KNAPSACK **CLO-4 Pg No-497**

- a. O(NW), O(N+W)
- b. O(N), O(NW)
- c. O(N+W), O(NW)
- d. O(NW), O(N)

40. Given a directed graph where weight of every edge is same, we can efficiently find shortest path from a given source to destination using? **CLO-3 Pg No-478**

**(A) Breadth First Traversal**

(B) Dijkstra's Shortest Path Algorithm

(C) Neither Breadth First Traversal nor Dijkstra's algorithm can be used

(D) Depth First Search

41. What can be the applications of Breadth First Search? **CLO-3 Net Source**

A. Finding shortest path between two nodes

B. Finding bipartiteness of a graph

C. GPS navigation system

**D. All of the mentioned**

42. What can be the applications of Depth First Search? **CLO-3 Net Source**

A. For generating topological sort of a graph

B. For generating Strongly Connected Components of a directed graph

C. Detecting cycles in the graph

**D. All of the mentioned**

43. When the Depth First Search of a graph is unique? **CLO-3 Net Source**

A. When the graph is a Binary Tree

**B. When the graph is a Linked List**

C. When the graph is a n-ary Tree

D. None of the mentioned

44. For a directed graph with edge lengths, the floyd warshall algorithm can compute the ..... between each pair of nodes in  $O(n^3)$ . **CLO-3 Pg No- 480**

A) Transitive Hull

**B) Minimax Distance**

C) Max Min Distance

D) Safest Path

45. Dijkstra algorithm is also called the ..... shortest path problem. **CLO-3 Pg No- 470**

A) multiple source

**B) single source**

C) single destination

D) multiple destination

46. The floyd-warshall all pairs shortest path algorithm computes the shortest paths between each pair of nodes in ..... **CLO-3 Pg No- 479**

A)  $O(\log n)$

B)  $O(n^2)$

C)  $O(mn)$

**D)  $O(n^3)$**

Reg. No. \_\_\_\_\_

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY  
CYCLE TEST – III – APRIL- 2020**

Fourth Semester – Computer Science and Engineering

**18CSC204J- Design and Analysis of Algorithms**

Duration: 90 Minutes

Max. Marks: 50

**PART – A (10 X 1 = 10 Marks)**

Answer ALL Questions

1. Backtracking algorithm is implemented by constructing a tree of choices called as?
  - a) State-space tree
  - b) State-chart tree
  - c) Node tree
  - d) Backtracking tree
2. Which data structure is most suitable for implementing best first branch and bound strategy?
  - a) stack
  - b) queue
  - c) priority queue
  - d) linked list
3. In what time can the Hamiltonian path problem can be solved using dynamic programming?
  - a)  $O(n)$
  - b)  $O(n \log n)$
  - c)  $O(n^2)$
  - d)  $O(n^2 2^n)$
4. In what manner is a state-space tree for a backtracking algorithm constructed?
  - a) Depth-first search
  - b) Breadth-first search
  - c) Twice around the tree
  - d) Nearest neighbour first
5. What is the worst-case running time of Rabin Karp Algorithm?
  - a)  $\theta(n)$
  - b)  $\theta(n - m)$
  - c)  $\theta((n - m + 1)m)$
  - d)  $\theta(n \log m)$
6. What approach is being followed in Floyd Warshall Algorithm?
  - a) Greedy technique
  - b) Dynamic Programming
  - c) Linear Programming
  - d) Backtracking
7. Which of the following options match the given statement:  
Statement: The algorithms that use the random input to reduce the expected running time or memory usage, but always terminate with a correct result in a bounded amount of time.
  - a) Las Vegas Algorithm
  - b) Monte Carlo Algorithm
  - c) Atlantic City Algorithm
  - d) None of the mentioned
8. What is vertex coloring of a graph?
  - a) A condition where any two vertices having a common edge should not have same color
  - b) A condition where any two vertices having a common edge should always have same color
  - c) A condition where all vertices should have a different color
  - d) A condition where all vertices should have same color

9. To which of the following class does a CNF-satisfiability problem belong?
  - a) NP class
  - b) P class
  - c) NP complete
  - d) NP hard
10. The time complexity of the normal quick sort, randomized quick sort algorithms in the worst case is
  - a)  $O(n^2), O(n \log n)$
  - b)  $O(n^2), O(n^2)$
  - c)  $O(n \log n), O(n^2)$
  - d)  $O(n \log n), O(n \log n)$

**PART – B (4 X 4 = 16 Marks)**

Answer ANY FOUR questions

11. Solve the following 0/1 Knapsack problem using Branch and Bound.  
(W(capacity of knapsack= 15)

|        |    |    |    |    |
|--------|----|----|----|----|
| Profit | 10 | 10 | 12 | 18 |
| Weight | 2  | 4  | 6  | 9  |

12. What do you mean by Indicator Random Variable? What is its significance in randomized algorithms?
13. What do you understand by string matching algorithms? List any three such algorithms.
14. Discuss the travelling salesman problem and show how it is an instance of the Hamiltonian Circuit problem
15. Differentiate Backtracking and Branch and Bound.

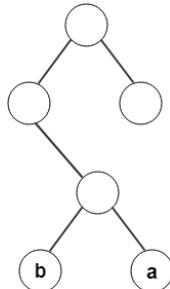
**PART – C (2 X 12 = 24 Marks)**

Answer ALL questions

16. a) What is backtracking? What kind of search technique is used for backtracking? Show how the technique can be applied to a 4x4 board using state space tree with algorithm.  
**(OR)**  
b) For given set w = {5, 10, 12, 13, 15, 18} and target sum=30, find all possible combination of subsets using state space tree with algorithm.
17. a) What do you understand by spurious hits? For string matching, how many spurious hits does the Rabin-Karp matcher encounters in Text T = 31415926535..... and Pattern P = 26. Calculate hash function as (P modulo (length of text string))  
**(OR)**  
b) Differentiate between deterministic algorithms and non-deterministic algorithms and accordingly explain in detail NP, Np Hard and NP Complete.

## UNIT 3 - Greedy and Dynamic Programming MCQs

1. From the following given tree, what is the code word for the character 'a'?



- a) 011      b) 010      c) 100      d) 101

2. The type of encoding where no character code is the prefix of another character code is called?  
a) optimal encoding    b) **prefix encoding**    c) frequency encoding    d) tree encoding

3. We use dynamic programming approach when  
a) We need an optimal solution                          b) **The solution has optimal substructure**  
c) The given problem can be reduced to the 3-SAT problem    d) It's faster than Greedy

4. Four matrices M1, M2, M3 and M4 of dimensions pxq, qxr, rxs and sxt respectively .**The minimum number of scalar multiplications required to find the product M1M2M3M4 using the basic matrix multiplication method is**

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

5. Consider the following two sequences :

X = < B, C, D, C, A, B, C >, and

Y = < C, A, D, B, C, B >

The length of longest common subsequence of X and Y is :

- a) 5    b) 3    c) **4**    d) 2

6. If a problem can be broken into subproblems which are reused several times, the problem possesses \_\_\_\_\_ property.

- a) **Overlapping subproblems**    b)Optimal substructure    c) Memoization    d)Greedy

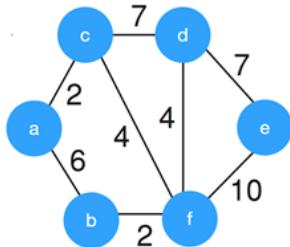
7. In dynamic programming, the technique of storing the previously calculated values is called

- a) Saving value property    b) Storing value property    c) **Memoization**    d) Mapping

8. Which of the following problems is NOT solved using dynamic programming?

- a) 0/1 knapsack problem      b) Matrix chain multiplication problem  
c) Longest Common Subsequence problem      d) **Fractional knapsack problem**

9. What is the time complexity of the above dynamic programming implementation of the longest common subsequence problem where length of one string is "m" and the length of the other string is "n"?
- a) O(n)      b) O(m)      c) O(m + n)      d) **O(mn)**
10. Which of the following standard algorithms is not a Greedy algorithm?
- a) **Matrix Chain Multiplication**      b) Prim's algorithm  
 c) Kruskal algorithm      d) Huffman Coding
11. Which of the following is true about Huffman Coding?
- a) Huffman coding may become lossy in some cases  
 b) Huffman Codes may not be optimal lossless codes in some cases  
**c) In Huffman coding, no code is prefix of any other code.**  
 d) BOTH A and B
12. The worst case time complexity of Prim's algorithm if adjacency matrix is used?
- a) O(log V)      b) **O(V<sup>2</sup>)**      c) O(E<sup>2</sup>)      d) O(V log E)
13. Kruskal's algorithm is used to \_\_\_\_\_
- a) **find minimum spanning tree**      b) find single source shortest path  
 c) find all pair shortest path algorithm      d) traverse the graph
14. What is the time complexity of Kruskal's algorithm?
- a) O(log V)      b) O(E log V)      c) O(E<sup>2</sup>)      d) **O(V log E)**
15. Consider the given graph.



What is the weight of the minimum spanning tree using the Kruskal's algorithm?

a) 24      b) 23      c) 15      d) **19**



**AALIM MUHAMMED SALEGH COLLEGE OF ENGINEERING**  
**DEPARTMENT OF INFORMATION TECHNOLOGY**  
**CS8451- DESIGN AND ANALYSIS OF ALGORITHMS**  
**UNIT I INTRODUCTION**  
**PART-A**

- 1. State the transpose symmetry property of O and  $\Omega$ . [Nov/Dec 2019]**  
 $f(n) = O(g(n))$  if and only if  $g(n) = \Omega(f(n))$

- 2. Define recursion. [Nov/Dec 2019]**

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily

- 3. How do you measure the efficiency of an Algorithm? [Apr/May 2019]**

- Size of the Input
- Running Time

- 4. Prove that  $f(n)=O(g(n))$  and  $g(n)=O(f(g(n)))$  then  $f(n)=\Theta(g(n))$  [Apr/May 2019]**

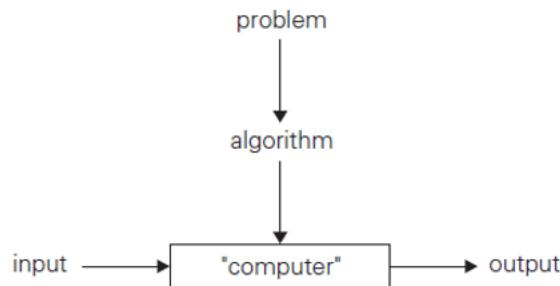
Prove by contradiction. Assume  $f(n) = O(g(n))$ , by the definition, there exist constants  $c, n_0 > 0$  such that  $0 \leq f(n) \leq cg(n)$  or  $0 \leq n \leq cn^1 + \sin n$  for all  $n \geq n_0$ . It implies  $(0.6) 0 \leq 1 \leq c \sin n$  for all  $n \geq n_0$ . Can it be true? To show that the answer is No, it suffices to show: For any  $n_0 > 0$ , we can always pick an  $n \geq n_0$  such that  $c \sin n < 1$ .

- 5. What is basic operation ?[ Apr/May 2018]**

The operation that contributes most towards the running time of the algorithm. The running time of an algorithm is the function defined by the number of steps (or amount of memory) required to solve input instances of size  $n$ .

- 6. What is an Algorithm? [Apr/May 2017]**

An algorithm is a sequence of unambiguous instructions for solving a problem. i.e., for obtaining a required output for any legitimate input in a finite amount of time



- 7. How to measure algorithm's running time? [Nov/Dec 2017]**

Time efficiency indicates how fast the algorithm runs. An algorithm's time efficiency is measured as a function of its input size by counting the number of times its basic operation (running time) is executed. Basic operation is the most time consuming operation in the algorithm's innermostloop.

**8. Compare the order of growth  $n(n-1)/2$  and  $n^2$ . [May/June2016]**

| $n$        | $n(n-1)/2$ | $n^2$     |
|------------|------------|-----------|
| Polynomial | Quadratic  | Quadratic |
| 1          | 0          | 1         |
| 2          | 1          | 4         |
| 4          | 6          | 16        |
| 8          | 28         | 64        |
| 10         | 45         | $10^2$    |
| $10^2$     | 4950       | $10^4$    |
| Complexity | Low        | High      |
| Growth     | Low        | high      |

$n(n-1)/2$  is lesser than the half of  $n^2$

**9. Define recurrence relation. [Nov/Dec2016]**

A recurrence relation is an equation that defines a sequence based on a rule that gives the next term as a function of the previous term(s). The simplest form of a recurrence relation is the case where the next term depends only on the immediately previous term.

**10. Write down the properties of asymptotic notations. [April/May2015]**

Asymptotic notation is a notation, which is used to take meaningful statement about the efficiency of a program. To compare and rank such orders of growth, computer scientists use three notations, they are:

- O - Big ohnotation
- $\Omega$  - Big omeganotation
- $\Theta$  - Big theta notation

**11. Give the Euclid's Algorithm for Computing gcd(m,n) [Apr/May '16, '18]**

**Algorithm Euclid\_gcd( $m, n$ )**

//Computes gcd( $m, n$ ) by Euclid's algorithm

//Input: Two nonnegative, not-both-zero integers  $m$  and  $n$

//Output: Greatest common divisor of  $m$  and  $n$

**while**  $n \neq 0$  **do**

$r \leftarrow m \bmod n$

$m \leftarrow n$

$n \leftarrow r$

**return**  $m$

Example:  $\text{gcd}(60, 24) = \text{gcd}(24, 12) = \text{gcd}(12, 0) = 12$ .

**12. Write an algorithm to find the number of binary digits in the binary representation of a positive decimal integer. [Apr May 2015]**

**Algorithm Binary( $n$ )**

//Input: A positive decimal integer  $n$

//Output: The number of binary digits in  $n$ 's binary representation

```

count ← 1
while  $n > 1$  do
    count ← count + 1
     $n \leftarrow \lfloor n/2 \rfloor$ 
return count

```

**13. Define Little “oh”. [April/May2014]**

The function  $f(n) =$

$O(g(n))$  iff  $\lim f(n)$

$=0$

$n \rightarrow \infty g(n)$

**14. Define Little Omega. [April/May2014]**

The function  $f(n) =$

$\omega(g(n))$  iff  $\lim f(n)$

$=0$

$n \rightarrow \infty g(n)$

**15. Define Big Theta Notations [Nov/Dec2014]**

A function  $t(n)$  is said to be in  $\Theta(g(n))$ , denoted  $t(n) \in \Theta(g(n))$ , if  $t(n)$  is bounded both above and below by some positive constant multiple of  $g(n)$  for all large  $n$ , i.e., if there exist some positive constants  $c_1$  and  $c_2$  and some nonnegative integer  $n_0$  such that  $c_1$

$g(n) \leq t(n) \leq c_2 g(n)$  for all  $n \geq n_0$

**16. Define Big Omega Notations. [May/June2013]**

A function  $t(n)$  is said to be in  $\Omega(g(n))$ , denoted  $t(n) \in \Omega(g(n))$ , if  $t(n)$  is bounded below by some positive constant multiple of  $g(n)$  for all large  $n$ , i.e., if there exist some positive constant  $c$  and some nonnegative integer  $n_0$  such that

$t(n) \geq c g(n)$  for all for all  $n \geq n_0$

**17. What is Big ‘Oh’ notation? [May/June2012]**

A function  $t(n)$  is said to be in  $O(g(n))$ , denoted  $t(n) O(g(n))$ , if  $t(n)$  is bounded above by some constant multiple of  $g(n)$  for all large  $n$ , i.e., if there exist some positive constant  $c$  and some nonnegative integers  $n_0$  such that

$t(n) \leq c g(n)$  for all  $n \geq n_0$

**18. Give the two major phases of performance evaluation.**

Performance evaluation can be loosely divided into two major phases:

- a prior estimates (performance analysis)
- a Posterior testing (performance measurement)

**19. What are six steps processes in algorithmic problem solving? [Nov/Dec2009]**

- Understanding theproblem
- Decision making on - Capabilities of computational devices, Choice of exact or approximate problem solving, Datastructures
- Algorithmicstrategies
- Specification ofalgorithm
- Algorithmicverification
- Analysis of algorithms

**20. What are the basic asymptotic efficiencyclasses?**

The various basic efficiency classes are

- Constant:1
- Logarithmic: logn
- Linear:n
- N-log-n: n logn
- Quadratic:n<sup>2</sup>
- Cubic: n<sup>3</sup>
- Exponential:2<sup>n</sup>
- Factorial:n!

**21. List the factors which affects the running time of thealgorithm.**

- A. Computer
- B. Compiler
- C. Input to thealgorithm
  - i. The content of the input affects the runningtime
  - ii. Typically, the input size is the mainconsideration.

**22. Give an non-recursive algorithm to find out the largest element in a list of nnumbers.**

**ALGORITHMMaxElement(A[0..n-1])**

//Determines the value of the largest element in a given array Input:An array A[0..n-1] of real numbers

//Output: The value of the largest element in A  
maxvalif $\neg a[0]$  for I if $\neg 1$  to n-1 do

```
if A[I] >maxval
return maxvalif $\neg$ 
A[I] return maxval
```

**23. Write a recursive algorithm for computing the nth fibonacci number.**

**ALGORITHMF(n)**

```
// Computes the nth Fibonacci number recursively by using the definition
// Input A non-negative integer n
```

```
// Output The nth Fibonacci number
if n = 1 return n
else return F(n-1)+F(n-2)
```

#### **24. What is algorithm visualization?**

Algorithm visualization can be defined as the use of images to convey some useful information about algorithms. Two principal variations are Static algorithm visualization Dynamic Algorithm visualization(also called algorithm animation)

#### **25. What is the order of growth?**

The Order of growth is the scheme for analyzing an algorithm's efficiency for different input sizes which ignores the multiplicative constant used in calculating the algorithm's running time. Measuring the performance of an algorithm in relation with the input size  $n$  is called the order of growth.

### **PART-B & C**

#### **1. Explain about algorithm with suitable example (Notion of algorithm).**

An algorithm is a sequence of unambiguous instructions for solving a computational problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

**Algorithms – Computing the Greatest Common Divisor of Two Integers**(gcd( $m, n$ ): the largest integer that divides both  $m$  and  $n$ .)

**Euclid's algorithm:** gcd( $m, n$ ) = gcd( $n, m \bmod n$ )

Step1: If  $n = 0$ , return the value of  $m$  as the answer and stop; otherwise, proceed to Step 2.

Step2: Divide  $m$  by  $n$  and assign the value of the remainder to  $r$ .

Step 3: Assign the value of  $n$  to  $m$  and the value of  $r$  to  $n$ . Go to Step 1.

**Algorithm Euclid( $m, n$ )**

```
//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n while n ≠ 0 do
    r ← m mod n
    m ← n
    n ← r
return m
```

#### **About This algorithm**

Finiteness: how do we know that Euclid's algorithm actually comes to a stop

Definiteness: nonambiguity

Effectiveness: effectively computable.

#### **Consecutive Integer Algorithm**

Step1: Assign the value of  $\min\{m, n\}$  to  $t$ .

Step2: Divide  $m$  by  $t$ . If the remainder of this division is 0, go to Step3;otherwise, go to Step 4.

Step3: Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to Step4.

Step4: Decrease the value of t by 1. Go to Step2.

### **About This algorithm**

- Finiteness
- Definiteness
- Effectiveness

### **Middle-school procedure**

Step1: Find the prime factors of m.

Step2: Find the prime factors of n.

Step3: Identify all the common factors in the two prime expansions found in Step1 and Step2. (If p is a common factor occurring  $P_m$  and  $P_n$  times in m and n, respectively, it should be repeated in  $\min\{P_m, P_n\}$  times.)

Step4: Compute the product of all the common factors and return it as the gcd of the numbers given.

## **2. Write short note on Fundamentals of Algorithmic Problem Solving**

Understanding the problem

- Asking questions, do a few examples by hand, think about special cases, etc.

Deciding on

- Exact vs. approximate problem solving
- Appropriate data structure

Design an algorithm

Proving correctness

Analyzing an algorithm

Time efficiency : how fast the algorithm runs

Space efficiency: how much extra memory the algorithm needs.

Coding an algorithm

## **3. Discuss important problem types that you face during Algorithm Analysis.**

Sorting

Rearrange the items of a given list in ascending order.

Input: A sequence of n numbers  $\langle a_1, a_2, \dots, a_n \rangle$

Output: A reordering  $\langle a'_1, a'_2, \dots, a'_n \rangle$  of the input sequence such that  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

A specially chosen piece of information used to guide sorting. I.e., sort student records by names.

### **Examples of sorting algorithms**

- Selection sort
- Bubble sort
- Insertion sort
- Merge sort
- Heap sort

### **Evaluate sorting algorithm complexity: the number of key comparisons.**

Two properties

Stability: A sorting algorithm is called stable if it preserves the relative order of any two equal elements in its input.

In place: A sorting algorithm is in place if it does not require extra memory, except, possibly for a few memory units.

- ✓ searching
- Find a given value, called a search key, in a given set.
- Examples of searching algorithms
  - Sequential searching
  - Binary searching...
- ✓ string processing
  - A string is a sequence of characters from an alphabet.
  - Text strings: letters, numbers, and special characters.
  - String matching: searching for a given word/pattern in a text.
- ✓ graph problems
  - Informal definition
    - A graph is a collection of points called vertices, some of which are connected by line segments called edges.
  - Modeling real-life problems
  - Modeling WWW
  - communication networks
  - Project scheduling ...

#### **Examples of graph algorithms**

- Graph traversal algorithms
- Shortest-path algorithms
- Topological sorting
- ✓ combinatorial problems
- ✓ geometric problems
- ✓ Numerical problems

#### **4. Discuss Fundamentals of the analysis of algorithm efficiency elaborately. [Nov/Dec 2019, Apr/May 2019]**

Algorithm's efficiency

Three notations

- Analyze of efficiency of Mathematical Analysis of Recursive Algorithms
- Analyze of efficiency of Mathematical Analysis of non-Recursive Algorithms
- Analysis of algorithms means to investigate an algorithm's efficiency with respect to resources: running time and memory space.
- Time efficiency: how fast an algorithm runs.
- Space efficiency: the space an algorithm requires.
- Measuring an input's size
- Measuring running time
- Orders of growth (of the algorithm's efficiency function)
- Worst-case, best-case and average efficiency

Measuring Input Sizes

Efficiency is defined as a function of input size.

Input size depends on the problem.

Example 1, what is the input size of the problem of sorting n numbers?

Example 2, what is the input size of adding two n by n matrices?

### Units for Measuring Running Time

Measure the running time using standard unit of time measurements, such as seconds, minutes?

Depends on the speed of the computer. count the number of times each of an algorithm's operations is executed.

Difficult and unnecessary count the number of times an algorithm's basic operation is executed.

Basic operation: the most important operation of the algorithm, the operation contributing the most to the total running time.

For example, the basic operation is usually the most time-consuming operation in the algorithm's innermost loop.

### Orders of Growth

consider only the leading term of a formula

Ignore the constant coefficient.

### Worst-Case, Best-Case, and Average-Case Efficiency

Algorithm efficiency depends on the input size n

For some algorithms efficiency depends on type of input.

Example: Sequential Search

**Problem:** Given a list of n elements and a search key K, find an element equal to K, if any.

**Algorithm:** Scan the list and compare its successive elements with K until either a matching element is found (successful search) or the list is exhausted (unsuccessful search)

#### Worst case Efficiency

**Efficiency** (# of times the basic operation will be executed) for the worst case input of size n.

The algorithm runs the longest among all possible inputs of size n.

**Best case Efficiency** (# of times the basic operation will be executed) for the best case input of size n.

The algorithm runs the fastest among all possible inputs of size n.

**Average case: Efficiency** (#of times the basic operation will be executed) for a typical/random input of size n.

NOT the average of worst and best case

## 5. Elaborate Asymptotic analysis of an algorithm with an example.

Three notations used to compare orders of growth of an algorithm's basic operation count

$O(g(n))$ : class of functions  $f(n)$  that grow no faster than  $g(n)$

A function  $t(n)$  is said to be in  $O(g(n))$ , denoted  $t(n) \in O(g(n))$ , if  $t(n)$  is bounded above by some constant multiple of  $g(n)$  for all large  $n$ , i.e., if there exist some positive constant c and some nonnegative integer  $n_0$  such that  $t(n) \leq cg(n)$  for all  $n \geq n_0$

$\Omega(g(n))$ : class of functions  $f(n)$  that grow at least as fast as  $g(n)$

A function  $t(n)$  is said to be in  $\Omega(g(n))$ , denoted  $t(n) \in \Omega(g(n))$ , if  $t(n)$  is bounded below by some

constant multiple of  $g(n)$  for all large  $n$ , i.e., if there exist some positive constant c and some nonnegative integer  $n_0$  such that  $t(n) \geq cg(n)$  for all  $n \geq n_0$

$\Theta(g(n))$ : class of functions  $f(n)$  that grow at same rate as  $g(n)$

A function  $t(n)$  is said to be in  $\Theta(g(n))$ , denoted  $t(n) \in \Theta(g(n))$ , if  $t(n)$  is bounded both above and below by some positive constant multiples of  $g(n)$  for all large  $n$ , i.e., if there exist some positive constant  $c_1$  and  $c_2$  and some nonnegative integer  $n_0$  such that  $c_2 g(n) \leq t(n) \leq c_1 g(n)$  for all  $n \geq n_0$

## 6. List out the Steps in Mathematical Analysis of non recursive Algorithms.

**Steps in mathematical analysis of nonrecursive algorithms:**

- Decide on parameter  $n$  indicating input size
- Identify algorithm's basic operation
- Check whether the number of times the basic operation is executed depends only on the input size  $n$ . If it also depends on the type of input, investigate worst, average, and best case efficiency separately.
- Set up summation for  $C(n)$  reflecting the number of times the algorithm's basic operation is executed.

Example: Finding the largest element in a given array

**Algorithm MaxElement ( $A[0..n-1]$ )**

```
//Determines the value of the largest element in a given array
//Input: An array A[0..n-1] of real numbers
//Output: The value of the largest element in A maxval ← A[0]
for i ← 1 to n-1 do
if A[i] > maxval
maxval ← A[i]
return maxval
```

## 7. List out the Steps in Mathematical Analysis of Recursive Algorithms.

Decide on parameter  $n$  indicating input size

Identify algorithm's basic operation

Determine worst, average, and best case for input of size  $n$

- ✓ Set up a recurrence relation and initial condition(s) for  $C(n)$ -the number of times the basic operation will be executed for an input of size  $n$  (alternatively count recursive calls).
- ✓ Solve the recurrence or estimate the order of magnitude of

$$\begin{array}{ll} \text{the solution } F(n) = 1 & \text{if } n = 0 \\ n * (n-1) * (n-2) \dots 3 * 2 * 1 & \text{if } n > 0 \end{array}$$

- ✓ Recursive definition

$$\begin{array}{ll} F(n) = 1 & \text{if } n = 0 \\ n * F(n-1) & \text{if } n > 0 \end{array}$$

**Algorithm  $F(n)$**

```
if n=0
else
    return 1          //base case

    return F (n -1) * n      //general case
```

**Example Recursive evaluation of  $n!$  (2)**

✓ Two Recurrences

The one for the factorial function

value:  $F(n) = F(n - 1) * n$  for every  $n > 0$

$F(0) = 1$

The one for number of multiplications to compute  $n!$ ,  $M(n) = M(n - 1) + 1$  for every  $n > 0$

$M(0) = 0$

$M(n) \in \Theta(n)$

**8. Explain in detail about linear search.**

**Sequential Search** searches for the key value in the given set of items sequentially and returns the position of the key value else returns -1.

**ALGORITHM** *SequentialSearch( $A[0..n - 1]$ ,  $K$ )*

```
//Searches for a given value in a given array by sequential search
//Input: An array  $A[0..n - 1]$  and a search key  $K$ 
//Output: The index of the first element of  $A$  that matches  $K$ 
//          or -1 if there are no matching elements
i  $\leftarrow 0$ 
while  $i < n$  and  $A[i] \neq K$  do
     $i \leftarrow i + 1$ 
if  $i < n$  return  $i$ 
else return -1
```

Analysis:

For sequential search, best-case inputs are lists of size  $n$  with their first elements equal to a search key; accordingly,

$$C_{bw}(n) = 1.$$

Average Case Analysis:

(a) The standard assumptions are that the probability of a successful search is equal  $p$  and  $1-p$  for failure.

(b) the probability of the first match occurring in the  $i$ th position of the list is the same for every  $i$ . Under these assumptions- the average number of key comparisons  $C_{avg}(n)$  is found as follows.

In the case of a successful search, the probability of the first match occurring in the  $i$ th position of the list is  $p/n$  for every  $i$ , and the number of comparisons made by the algorithm in such a situation is obviously  $i$ . In the case of an unsuccessful search, the number of comparisons is  $n$  with the probability of such a search being  $(1-p)$ . Therefore

$$\begin{aligned} C_{avg}(n) &= [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \dots + i \cdot \frac{p}{n} + \dots + n \cdot \frac{p}{n}] + n \cdot (1-p) \\ &= \frac{p}{n}[1 + 2 + \dots + i + \dots + n] + n(1-p) \\ &= \frac{p}{n} \frac{n(n+1)}{2} + n(1-p) = \frac{p(n+1)}{2} + n(1-p). \end{aligned}$$

For example, if  $p = 1$  (i.e., the search must be successful), the average number of key comparisons made by sequential search is  $(n + 1) / 2$ ; i.e., the algorithm will inspect, on average, about half of the list's elements. If  $p = 0$  (i.e., the search must be unsuccessful), the average number of key comparisons will be  $n$  because the algorithm will inspect all  $n$  elements on all such inputs.

**8. Write an Algorithm using recursion that determines the GCD of two numbers. Determine the time and space complexity. [Nov/Dec 2019]**

Extended Euclidean Algorithm:

Extended Euclidean algorithm also finds integer coefficients  $x$  and  $y$  such that:

$$ax + by = \gcd(a, b)$$

Examples:

Input:  $a = 30, b = 20$

Output:  $\gcd = 10$

$$x = 1, y = -1$$

(Note that  $30*1 + 20*(-1) = 10$ )

Input:  $a = 35, b = 15$

Output:  $\gcd = 5$

$$x = 1, y = -2$$

(Note that  $35*1 + 15*(-2) = 5$ )

The extended Euclidean algorithm updates results of  $\gcd(a, b)$  using the results calculated by recursive call  $\gcd(b \% a, a)$ . Let values of  $x$  and  $y$  calculated by the recursive call be  $x_1$  and  $y_1$ .  $x$  and  $y$  are updated using the below expressions.

$$x = y_1 - \lfloor b/a \rfloor * x_1$$

$$y = x_1$$

Time complexity is  $\log_2(\max(a,b))$  and in good case, if  $a \mid b$  or  $b \mid a$  then time complexity is  $O(1)$

## **UNIT II BRUTE FORCE AND DIVIDE AND CONQUER**

## **QUESTION BANK**

## PART - A

- ## **1. State the Convex Hull Problem. [Nov/Dec 2019]**

The **convex hull** of a set of points is defined as the smallest **convex polygon**, that encloses all of the points in the set. **Convex** means that the **polygon** has no corner that is bent inwards.

2. Write the Brute force algorithm to string matching.[Apr/May 2019]

### Algorithm NAÏVE(Text, Pattern)

n=length[Text]

n=length[Pattern]

for  $s=1$  to  $n-m$

if pattern[1...m]==Text[s+1...s+m]

Print "location of pattern is found with shift s"

3. What is the time and space complexity of Merge Sort? [Apr/May 2019]

| Time Complexity                  | Space Complexity              |
|----------------------------------|-------------------------------|
| Best Case: $\Theta(n \log n)$    | Best Case: $\Theta(n \log n)$ |
| Average Case: $\Theta(n \log n)$ | Average Case: $n \log n$      |
| Worst Case: $\Theta(n \log n)$   | Worst Case: $n \log n$        |

- #### **4. What is exhaustive search? [Apr/May 2018]**

An Exhaustive Search, also known as generate and test, is a very general problem solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.

- ## 5. State Master's Theorem. [Apr/May 2018]

Let  $T(n)$  be a monotonically increasing function that satisfies

$$T(n) = aT(n/b)$$

$$+ f(n) T(1) = c$$

Where  $a > 1$ ,  $b >$

Where  $a \geq 1, b \geq 2, c > 0$ . If  $f(n) \in \Theta(n^c)$  where  $d \leq c, \dots$

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

- ## 6. What is Closest Pair Problem? [May/June 2016, Apr/May 2017]

The closest-pair problem finds the two closest points in a set of  $n$  points. It is the simplest of a variety of problems in computational geometry that deals with proximity of points in the plane or higher-dimensional spaces. The distance between two Cartesian coordinates is calculated by Euclidean distance formula

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

- ## 7. Give the General Plan for Divide and Conquer Algorithms [Nov/Dec 2017]

A **divide and conquer algorithm** works by recursively breaking down a problem into two or more sub-problems of the same (or related) type (**divide**), until these become simple enough to be solved directly (**conquer**).

Divide-and-conquer algorithms work according to the following general plan:

- A problem is divided into several subproblems of the same type, ideally of about equal size.
- The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
- If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

**Example:** Merge sort, Quick sort, Binary search, Multiplication of Large Integers and Strassen's Matrix Multiplication.

#### 8. Write the advantages of insertion sort. [ Nov/Dec 2017]

- Simple implementations
- Efficient for Small Data Sets
- Stable
- More efficient
- Online

#### 9. Derive the Complexity of Binary Search [Apr/May 2015]

In conclusion we are now able completely describe the computing time of binary search by giving formulas that describe the best, average and worst cases.

| Successful searches                                                                             | Unsuccessful searches                                    |
|-------------------------------------------------------------------------------------------------|----------------------------------------------------------|
| Best case - $\Theta(1)$<br>Average case - $\Theta(\log_2 n)$<br>Worst case - $\Theta(\log_2 n)$ | Best case, Average case, Worst case - $\Theta(\log_2 n)$ |

#### 10. Write about traveling salespersonproblem.

Let  $g = (V, E)$  be a directed. The tour of  $G$  is a directed simple cycle that includes every vertex in  $V$ . The cost of a tour is the sum of the cost of the edges on the tour. The traveling salesperson problem to find a tour of minimum cost.

#### 11. What is binarysearch?

Binary search is a remarkably efficient algorithm for searching in a sorted array. It works by comparing a search key  $K$  with the arrays middle element  $A[m]$ . if they match the algorithm stops; otherwise the same operation is repeated recursively for the first half of the array if  $K < A[m]$  and the second half if  $K > A[m]$ .

$K > A[0] \dots A[m-1] A[m] A[m+1] \dots A[n-1]$   
search here if  $K \neq A[m]$

#### 12. What is Knapsack problem? [Nov/Dec 2019, Nov/Dec2014]

A bag or sack is given capacity and  $n$  objects are given. Each object has weight  $w_i$  and profit  $p_i$ . Fraction of object is considered as  $x_i$  (i.e)  $0 <= x_i <= 1$ . If fraction is 1 then entire object is put into sack. When we place this fraction into the sack, we get  $w_i x_i$  and  $p_i x_i$ .

#### 13. What is convexhull?

Convex Hull is defined as: If  $S$  is a set of points then the Convex Hull of  $S$  is the smallest convex set containing

**14. Write the algorithm for Iterative binarysearch.**

```

Algorithm BinSearch(a,n,x)
    //Given an array a[1:n] of elements in nondecreasing
    // order, n>0, determine whether x is present
    {
        low := 1;
        high := n;
        while (low < high) do
        {
            mid := [(low+high)/2];
            if(x < a[mid]) then high:= mid-1;
            else if (x >a[mid]) then low:=mid + 1;
            else return mid;
        }
        return 0;
    }

```

**15. Define internal path length and external pathlength.**

The internal path length 'I' is the sum of the distances of all internal nodes from the root.  
The external path length E, is defines analogously as sum of the distance of all external nodes from the root.

**16. Write an algorithm for brute force closest-pair problem. [Nov/Dec2016]**

```

Algorithm BruteForceClosestPair(P )
    //Finds distance between two closest points in the plane by brute force
    //Input: A list P of n ( $n \geq 2$ ) points  $p_1(x_1, y_1), \dots, p_n(x_n, y_n)$ 
    //Output: The distance between the closest pair of points
     $d \leftarrow \infty$ 
    for  $i \leftarrow 1$  to  $n - 1$  do
        for  $j \leftarrow i + 1$  to  $n$  do
             $d \leftarrow \min(d, \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2})$  //sqrt is square root
    return  $d$ 

```

**17. Design a brute-force algorithm for computing the value of a polynomial**

$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$  at a given point  $x_0$  and determine its worst case efficiency class.

```

Algorithm BetterBruteForcePolynomialEvaluation(P[0..n], x)
    //The algorithm computes the value of polynomial P at a given point x by the “lowest-to-
    //highest
    //term” algorithm
    //Input: Array P[0..n] of the coefficients of a polynomial of degree n, from the lowest to the
    //highest, and a number x
    //Output: The value of the polynomial at
        the point x  $p \leftarrow P[0]; power \leftarrow 1$ 
        for  $i \leftarrow 1$  to  $n$  do
             $power \leftarrow power * x$ 
             $p \leftarrow p + P[i] * power$ 
        return  $p$ 

```

**18. Show the recurrence relation of divide-and-conquer?**

The recurrence relation is

$$T(n) = g(n) \\ T(n_1) + T(n_2) + \dots + T(n_{BTL}) + f(n)$$

**19. What is the Quick sort and Write the Analysis for the Quick sort?**

In quick sort, the division into sub arrays is made so that the sorted sub arrays do not need to be merged later. In analyzing QUICKSORT, we can only make the number of element comparisons  $c(n)$ . It is easy to see that the frequency count of other operations is of the same order as  $C(n)$ .

**20. List out the Advantages in Quick Sort**

It is in-place since it uses only a small auxiliary stack

- It requires only  $n \log(n)$  time to sort  $n$  items
- It has an extremely short inner loop

This algorithm has been subjected to a thorough mathematical analysis, a very precise statement can be made about performance issues

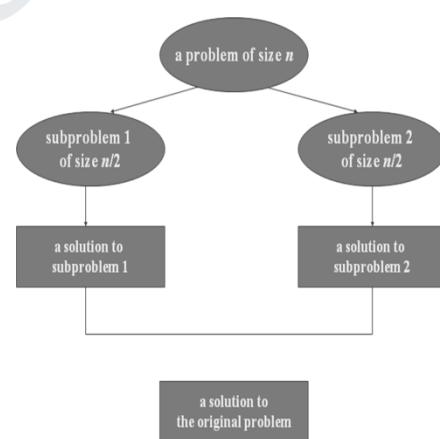
**21. What is Divide and Conquer Algorithm? [MAY/JUNE 2016][NOV/DEC 2017]**

It is a general algorithm design techniques that solved a problem's instance by dividing it into several smaller instances, solving each of them recursively, and then combining their solutions to the original instance of the problem.

**PART B & C****1. Explain Divide and Conquer Method**

The most well-known algorithm design strategy is Divide and Conquer Method. It

1. Divide the problem into two or more smaller subproblems.
2. Conquer the subproblems by solving them recursively.
3. Combine the solutions to the subproblems into the solutions for the original problem.



- ✓ Divide and Conquer Examples
- Sorting: mergesort and quicksort
  - Tree traversals
  - Binary search

- Matrix multiplication-Strassen's algorithm

**2. Explain Merge Sort with suitable example.**

- ✓ Merge sort definition.

Mergesort sorts a given array A[0..n-1] by dividing it into two halves a[0..(n/2)-1] and A[(n/2)..n-1] sorting each of them recursively and then merging the two smaller sorted arrays into a single sorted one.

- ✓ Steps in Merge Sort

1. Divide Step

If given array A has zero or one element, return S; it is already sorted. Otherwise, divide A into two arrays, A1 and A2, each containing about half of the elements of A.

2. Recursion Step

Recursively sort array A1 and A2.

3. Conquer Step

Combine the elements back in A by merging the sorted arrays A1 and A2 into a sorted sequence

- ✓ Algorithm for merge sort.

**ALGORITHM**

*Mergesort(A[0..n-1])*

//Sorts an array A[0..n-1] by recursive mergesort

//Input: An array A[0..n-1] of orderable elements

//Output: Array A[0..n-1] sorted in

nondecreasing order if n > 1

copy A[0..(n/2)-1] to B[0..(n/2)-1]

copy A[(n/2)..n-1] to C[0..(n/2)-1]

*Mergesort(B[0..(n/2)-1])*

*Mergesort(C[0*

*..(n/2)-1])*

*Merge(B,C,A)*

- ✓ Algorithm to merge two sorted arrays into one.

**ALGORITHM Merge (B [0..p-1], C[0..q-1],**

**A[0..p+q-1])**

//Merges two sorted arrays into one sorted array

//Input: arrays B[0..p-1] and C[0..q-1] both sorted

//Output: sorted array A [0..p+q-1] of the

elements of B & C I 0; j 0; k 0

while I < p

and j < q do

if B[I] <=

C[j]

A[k] B [I]; I I+1

else

A[k]

C[j]

]; j j+1 k

k+

1

if i = p

copy C[j..q-1] to A

[k..p+q-1] else  
copy B[i..p-1] to A [k..p+q-1]

### 3. Discuss Quick Sort Algorithm and Explain it with example. Derive Worst case and Average Case Complexity. [Apr/May 2019]

Quick Sort definition

Quick sort is an algorithm of choice in many situations because it is not difficult to implement, it is a good "general purpose" sort and it consumes relatively fewer resources during execution.

Quick Sort and divide and conquer

- Divide: Partition array  $A[l..r]$  into 2 subarrays,  $A[l..s-1]$  and  $A[s+1..r]$  such that each element of the first array is  $\leq A[s]$  and each element of the second array is  $\geq A[s]$ . (Computing the index of s is part of partition.)
- Implication:  $A[s]$  will be in its final position in the sorted array.
- Conquer: Sort the two subarrays  $A[l..s-1]$  and  $A[s+1..r]$  by recursive calls to quicksort
- Combine: No work is needed, because  $A[s]$  is already in its correct place after the partition is done, and the two subarrays have been sorted.

✓ Steps in Quicksort

- Select a pivot w.r.t. whose value we are going to divide the sublist. (e.g.,  $p = A[l]$ )
- Rearrange the list so that it starts with the pivot followed by a  $\leq$  sublist (a sublist whose elements are all smaller than or equal to the pivot) and a  $\geq$  sublist (a sublist whose elements are all greater than or equal to the pivot) Exchange the pivot with the last element in the first sublist(i.e.,  $\leq$  sublist) – the pivot is now in its final position
- Sort the two sublists recursively using quicksort.



✓ The Quicksort Algorithm

**ALGORITHM Quicksort( $A[l..r]$ )**

//Sorts a subarray by quicksort

//Input: A subarray  $A[l..r]$  of  $A[0..n-1]$ , defined by its left and right indices l and r

//Output: The subarray  $A[l..r]$  sorted in nondecreasing order if  $l < r$

s  $\square$  Partition ( $A[l..r]$ ) // s is a split position  
Quicksort( $A[l..s-1]$ )

Quicksort( $A[s+1..r]$ )

**ALGORITHM Partition ( $A[l ..r]$ )**

//Partitions a subarray by using its first element as a pivot

//Input: A subarray  $A[l..r]$  of  $A[0..n-1]$ , defined by its left and right indices l and r ( $l < r$ )

//Output: A partition of  $A[l..r]$ , with the split position returned as this function's value P  $\square A[l]$

i  $\square l$ ; j  $\square r + 1$ ;

Repeat

repeat i  $\square i + 1$  until  $A[i] \geq p$  //left-right scan  
repeat j  $\square j - 1$  until  $A[j] \leq p$  //right-left scan

```

if (i < j)           //need to continue
    with the scan swap(A[i], a[j])
until i >= j        //no
need to scan swap(A[l], A[j])
return j

```

✓ Advantages in Quick Sort

- It is in-place since it uses only a small auxiliary stack.
- It requires only  $n \log(n)$  time to sort  $n$  items.
- It has an extremely short inner loop

- This algorithm has been subjected to a thorough mathematical analysis, a very precise statement can be made about performance issues.

✓ Disadvantages in Quick Sort

- It is recursive. Especially if recursion is not available, the implementation is extremely complicated.
- It requires quadratic (i.e.,  $n^2$ ) time in the worst-case.
- It is fragile i.e., a simple mistake in the implementation can go unnoticed and cause it to perform badly.

✓ Efficiency of Quicksort

Based on whether the partitioning is balanced.

Best case: split in the middle —  $\Theta(n \log n)$

$C(n) = 2C(n/2) + \Theta(n)$  //2 subproblems of size  $n/2$  each

Worst case: sorted array! —  $\Theta(n^2)$

$C(n) = C(n-1) + n+1$  //2 subproblems of size 0 and  $n-1$  respectively

Average case: random arrays —  $\Theta(n \log n)$

**4. Explain in detail about Travelling Salesman Problem using exhaustive search. [Nov/Dec 2019]**

Given  $n$  cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city

Alternatively: Find shortest *Hamiltonian circuit* in a weighted connected graph

**Efficiency:  $\Theta((n-1)!)$**

Refer Anany Levitin's "Introduction to Design and Analysis of Algorithms" for examples.

**5. Explain in detail about Knapsack Problem. [Apr/May 2019]**

Given  $n$  items:

weights:  $w_1 \ w_2 \ \dots \ w_n$

values:  $v_1 \ v_2 \ \dots \ v_n$  a

knapsack of capacity  $W$

Find most valuable subset of the items that fit into the knapsack

**Example: Knapsack capacity W=16**

| item | weight | value |
|------|--------|-------|
| 1    | 2      | \$20  |
| 2    | 5      | \$30  |

|                                                      |    |              |
|------------------------------------------------------|----|--------------|
| 3                                                    | 10 | \$50         |
| 4                                                    | 5  | \$10         |
| <u>Subset</u> <u>Total weight</u> <u>Total value</u> |    |              |
| {1}                                                  | 2  | \$20         |
| {2}                                                  | 5  | \$30         |
| {3}                                                  | 10 | \$50         |
| {4}                                                  | 5  | \$10         |
| {1,2}                                                | 7  | \$50         |
| {1,3}                                                | 12 | \$70         |
| {1,4}                                                | 7  | \$30         |
| {2,3}                                                | 15 | \$80         |
| {2,4}                                                | 10 | \$40         |
| {3,4}                                                | 15 | \$60         |
| {1,2,3}                                              | 17 | not feasible |
| {1,2,4}                                              | 12 | \$60         |

**6. Write algorithm to find closest pair of points using divide and conquer and explain it with example. Derive the worst case and average case time complexity. [Nov/Dec 2019]**

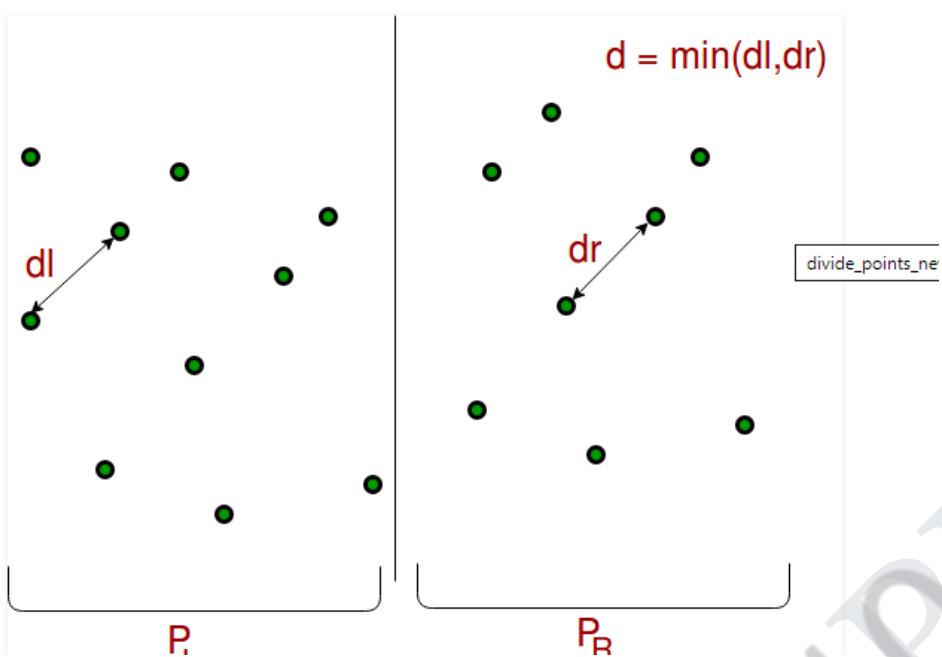
Following are the detailed steps of a  $O(n (\log n)^2)$  algorithm.

Input: An array of  $n$  points  $P[]$

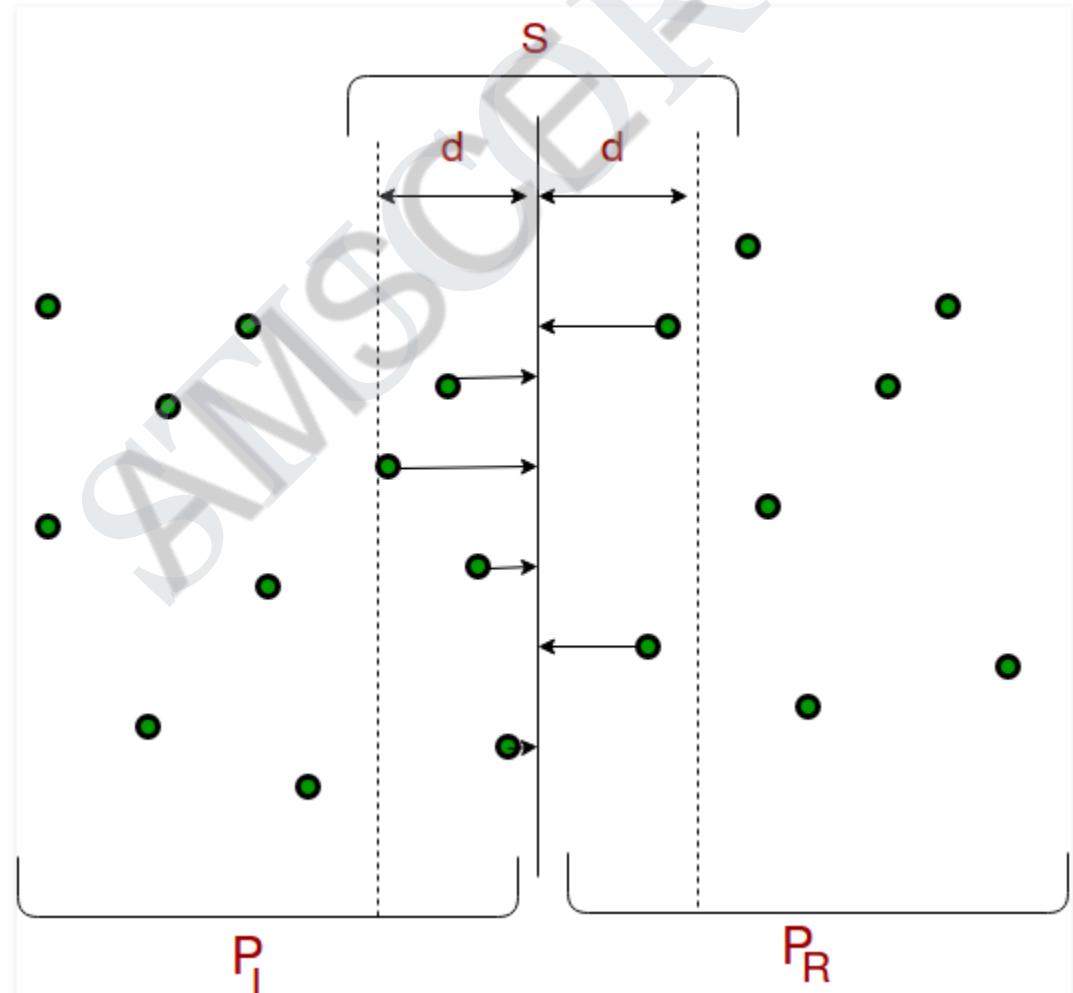
Output: The smallest distance between two points in the given array.

As a pre-processing step, the input array is sorted according to x coordinates.

- 1) Find the middle point in the sorted array, we can take  $P[n/2]$  as middle point.
- 2) Divide the given array in two halves. The first subarray contains points from  $P[0]$  to  $P[n/2]$ . The second subarray contains points from  $P[n/2+1]$  to  $P[n-1]$ .
- 3) Recursively find the smallest distances in both subarrays. Let the distances be  $dl$  and  $dr$ . Find the minimum of  $dl$  and  $dr$ . Let the minimum be  $d$ .



From the above 3 steps, we have an upper bound  $d$  of minimum distance. Now we need to consider the pairs such that one point in pair is from the left half and the other is from the right half. Consider the vertical line passing through  $P[n/2]$  and find all points whose  $x$  coordinate is closer than  $d$  to the middle vertical line. Build an array  $\text{strip}[]$  of all such points.



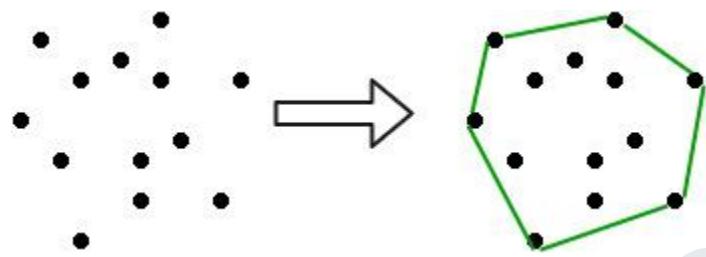
5) Sort the array strip[] according to y coordinates. This step is  $O(n\log n)$ . It can be optimized to  $O(n)$  by recursively sorting and merging.

6) Find the smallest distance in strip[]. This is tricky. From the first look, it seems to be a  $O(n^2)$  step, but it is actually  $O(n)$ . It can be proved geometrically that for every point in the strip, we only need to check at most 7 points after it (note that strip is sorted according to Y coordinate). See this for more analysis.

7) Finally return the minimum of d and distance calculated in the above step (step 6)

## **7. What is Convex hull problem? Explain the brute force approach to solve convex-hull with an example. Derive time complexity. [Apr/May 2019]**

A convex hull is the smallest convex polygon containing all the given points.



Input is an array of points specified by their x and y coordinates. The output is the convex hull of this set of points.

- The brute-force method expresses the fundamental solution, which gives you the basic building blocks and understanding to approach more complex solutions
- It's faster to implement
- It's still a viable solution when n is small, and n is usually small.

### **Brute-force construction**

Iterate over every pair of points (p,q)

If all the other points are to the right (or left, depending on implementation) of the line formed by (p,q), the segment (p,q) is part of our result set (i.e. it's part of the convex hull)

Here's the top-level code that handles the iteration and construction of resulting line segments:

```
/**
 * Compute convex hull
 */
var computeConvexHull = function() {
    console.log("--- ");

    for(var i=0; i<points.length; i++) {
        for(var j=0; j<points.length; j++) {
            if(i === j) {

```

```
        continue;  
    }  
  
    var ptI = points[i];  
    var ptJ = points[j];  
  
    // Do all other points lie within the half-plane to the right  
    var allPointsOnTheRight = true;  
    for(var k=0; k<points.length; k++) {  
        if(k === i || k === j) {  
            continue;  
        }  
  
        var d = whichSideOfLine(ptI, ptJ, points[k]);  
        if(d < 0) {  
            allPointsOnTheRight = false;  
            break;  
        }  
    }  
  
    if(allPointsOnTheRight) {  
        console.log("segment " + i + " to " + j);  
        var pointAScreen = cartToScreen(ptI, getDocumentWidth(), getDocumentHeight());  
        var pointBScreen = cartToScreen(ptJ, getDocumentWidth(), getDocumentHeight());  
        drawLineSegment(pointAScreen, pointBScreen);  
    }  
};
```

**UNIT-III DYNAMIC PROGRAMMING AND GREEDY TECHNIQUE****QUESTION BANK****PART - A****1. State the Principle of Optimality [Apr/May 2019, Nov/Dec 2017, Nov/Dec 2016]**

The principle of optimality is the basic principle of dynamic programming. It states that an optimal sequence of decisions or choices, each subsequence must also be optimal.

**2. What is the Constraint for binary search tree for insertion? [Apr/May 2019]**

When inserting or searching for an element in a binary search tree, the key of each visited node has to be compared with the key of the element to be inserted or found. The shape of the binary search tree depends entirely on the order of insertions and deletions, and can become degenerate.

**3. Define multistage graph. Give Example. [Nov/Dec 2018]**

A Multistage graph is a directed graph in which the nodes can be divided into a set of stages such that all edges are from a stage to next stage only (In other words there is no edge between vertices of same stage and from a vertex of current stage to previous stage).

**4. How Dynamic Programming is used to solve Knapsack Problem? [Nov/Dec 2018]**

An example of dynamic programming is Knapsack problem. The solution to the Knapsack problem can be viewed as a result of sequence of decisions. We have to decide the value of  $x_i$  for  $1 \leq i \leq n$ . First we make a decision on  $x_1$  and then on  $x_2$  and so on. An optimal sequence of decisions maximizes the object function  $\sum p_i x_i$ .

**5. Define transitive closure of directive graph. [Apr/May 2018]**

The transitive closure of a directed graph with 'n' vertices is defined as the n-by-n Boolean matrix  $T = \{t_{ij}\}$ , in which the elements in the  $i$ th row ( $1 \leq i \leq n$ ) and the  $j$ th column ( $1 \leq j \leq n$ ) is 1 if there exists a non trivial directed path from the  $i$ th vertex to the  $j$ th vertex otherwise,  $t_{ij}$  is 0 .

**6. Define the Minimum Spanning tree problem. [Apr/May 2018]**

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

**7. What does Floyd's Algorithm do? [Nov/Dec 2017]**

Floyd's algorithm is an application, which is used to find the entire pairs shortest paths problem. Floyd's algorithm is applicable to both directed and undirected weighted graph, but they do not contain a cycle of a negative length.

**8. State the Assignment Problem [May/June 2016]**

There are  $n$  people who need to be assigned to execute  $n$  jobs as one person per job. Each person is assigned to exactly one job and each job is assigned to exactly one person.

**9. Define the Single Source Shortest Path Problem. [May/June 2016]**

Single source shortest path problem can be used to find the shortest path from single source to all other vertices.

Example:Dijkstras algorithm

**10. List out the memory function under dynamic programming. [Apr/May 2015]**

- Top-Down Approach
- Bottom –Up Approach

**11. What is Huffman trees?**

A Huffman tree is binary tree that minimizes the weighted path length from the root to the leaves containing a set of predefined weights. The most important application of Huffman trees are Huffman code.

**12. List the advantage of Huffman's encoding?**

- Huffman's encoding is one of the most important file compression methods.
- It is simple
- It is versatility
- It provides optimal and minimum length encoding

**13. What do you mean by Huffman code?**

A Huffman code is a optimal prefix tree variable length encoding scheme that assigns bit strings to characters based on their frequencies in a given text.

**14. What is greedy method?**

The greedy method is the most straight forward design, which is applied for change making problem.

The greedy technique suggests constructing a solution to an optimization problem through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step, the choice made must be feasible, locally optimal and irrevocable.

**15. What do you mean by row major and column major?**

In a given matrix, the maximum elements in a particular row is called row major.

In a given matrix, the maximum elements in a particular column is called column major.

**16. Compare Greedy method and Dynamic Programming**

| Greedy method                                               | Dynamic programming                               |
|-------------------------------------------------------------|---------------------------------------------------|
| 1.Only one sequence of decisionis generated.                | 1.Many number of decisions are generated.         |
| 2.It does not guarantee to give an optimal solution always. | 2.It definitely gives an optimal solution always. |

**17. Show the general procedure of dynamic programming. [APR/MAY 2017]**

The development of dynamic programming algorithm can be broken into a sequence of 4 steps.

- Characterize the structure of an optimalsolution.
- Recursively define the value of the optimalsolution.
- Compute the value of an optimal solution in the bottom-up fashion.
- Construct an optimal solution from the computed information

**18.Define Kruskal Algorithm.**

Kruskal's algorithm is another greedy algorithm for the minimum spanning tree problem.

Kruskal's algorithm constructs a minimum spanning tree by selecting edges in increasing order of their weights provided that the inclusion does not create a cycle. Kruskal's algorithm provides

a optimal solution.

### **19. List the features of dynamic programming?**

Optimal solutions to sub problems are retained so as to avoid recomputing their values. Decision sequences containing subsequences that are sub optimal are not considered. It definitely gives the optimal solution always.

### **20. Write the method for Computing a Binomial Coefficient**

Computing binomial coefficients is non optimization problem but can be solved using dynamic programming.

Binomial coefficients are represented by  $C(n, k)$  or  $(nk)$  and can be used to represent the coefficients of a binomial:

$(a + b)^n = C(n, 0)a^n + \dots + C(n, k)a^{n-k}b^k + \dots + C(n, n)b^n$  The recursive relation is defined by the prior power

$$C(n, k) = C(n-1, k-1) + C(n-1, k) \text{ for } n > k > 0$$

$$\text{IC } C(n, 0) = C(n, n) = 1$$

## PART B & C

### **1. Explain Kruskal's Algorithm**

Greedy Algorithm for MST: Kruskal

Edges are initially sorted by increasing weight

Start with an empty forest

-grow MST one edge at a time

intermediate stages usually have forest of trees (not connected)

at each stage add minimum weight edge among those not yet used that does not create a cycle

at each stage the edge may:

expand an existing tree

combine two existing trees into a single tree

create a new tree

need efficient way of detecting/avoiding cycles

algorithm stops when all vertices are included

ALGORITHM Krusal(G)

//Input: A weighted connected graph  $G = \langle V, E \rangle$

//Output:  $E_T$ , the set of edges composing a minimum spanning tree of G.

*Sort E in nondecreasing order of the edge weights*

$$w(e_{i1}) \leq \dots \leq w(e_{i|E|})$$

$E_T \leftarrow \emptyset$ ;  $e_{\text{counter}} \leftarrow 0$  //initialize the set of tree edges and its size

$$k \leftarrow 0$$

while  $e_{\text{counter}} < |V| - 1$  do

$$k \leftarrow k + 1$$

if  $E_T \cup \{e_{ik}\}$  is acyclic

$$E_T \leftarrow E_T \cup \{e_{ik}\}; e_{\text{counter}} \leftarrow e_{\text{counter}} + 1$$

return  $E$

## 2. Discuss Prim's Algorithm in detail.

- ✓ Minimum Spanning Tree (MST)
  - Spanning tree of a connected graph  $G$ : a connected acyclic subgraph (tree) of  $G$  that includes all of  $G$ 's vertices.
  - Minimum Spanning Tree of a weighted, connected graph  $G$ : a spanning tree of  $G$  of minimum total weight.
- ✓ Prim's MST algorithm
- ✓ Start with a tree,  $T_0$ , consisting of one vertex
- ✓ -Grow|| tree one vertex/edge at a time
  - Construct a series of expanding subtrees  $T_1, T_2, \dots, T_{n-1}$ . At each stage construct  $T_{i+1}$  from  $T_i$  by adding the minimum weight edge connecting a vertex in tree ( $T_i$ ) to one not yet in tree
    - choose from -fringe|| edges
- ✓ (this is the -greedy|| step!) Or (another way to understand it)
- ✓ expanding each tree ( $T_i$ ) in a greedy manner by attaching to it the nearest vertex not in that tree. (a vertex not in the tree connected to a vertex in the tree by an edge of the smallest weight)
- ✓ Algorithm stops when all vertices are included

**Algorithm:** ALGORITHM Prim( $G$ )

//Prim's algorithm for constructing a minimum spanning tree

//Input A weighted connected graph  $G = V, E$

//Output  $E_T$ , the set of edges composing a minimum spanning tree of  $G$   $V_T \{v_0\}$

$\leftarrow$   
 $E_T \leftarrow F$

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

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

$V_T \leftarrow V_T \cup \{u^*\}$   
 $\leftarrow$   
 $E_T \quad E_T \cup \{e^*\}$

## 3. Write short note on Greedy Method. [Nov/Dec 2019]

- ✓ A greedy algorithm makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
  - The choice made at each step must be:
    - Satisfy the problem's constraints
  - locally optimal
    - Be the best local choice among all feasible choices
  - Irrevocable
    - Once made, the choice can't be changed on subsequent steps.
- ✓ Applications of the Greedy Strategy
  - Optimal solutions:
    - change making

- Minimum Spanning Tree (MST)
- Single-source shortest paths
- Huffman codes
- Approximations:
  - Traveling Salesman Problem (TSP)
  - Knapsack problem
  - other optimization problems

#### 4. What does dynamic programming have in common with divide-and-Conquer?

- ✓ **Dynamic Programming**
- Dynamic Programming is a general algorithm design technique. -Programming here means -planning.
- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems
- Main idea:
  - solve several smaller (overlapping) subproblems
  - record solutions in a table so that each subproblem is only solved once
  - final state of the table will be (or contain) solution

Dynamic programming vs. divide-and-conquer

- ✓ partition a problem into overlapping subproblems and independent ones
- ✓ store and not store solutions to subproblems

#### Example: Fibonacci numbers

Recall definition of Fibonacci numbers:

$$f(0) = 0$$

$$f(1) = 1$$

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

#### 5. Explain how to Floyd's Algorithm works. [Apr/May 2019]

- All pairs shortest paths problem: In a weighted graph, find shortest paths between every pair of vertices.
- Applicable to: undirected and directed weighted graphs; no negative weight.

Same idea as the Warshall's algorithm : construct solution through series of matrices  $D(0), D(1), \dots, D(n)$

Refer Examples from Anany Levitin. "Introduction to Design and Analysis of Algorithms".

#### 6. Construct optimal binary search tree for the following 5 keys with probabilities as indicated.

| i | 0    | 1   | 2   | 3   | 4   | 5   |
|---|------|-----|-----|-----|-----|-----|
| p |      | .15 | .10 | .05 | .10 | .20 |
| q | 0.05 | .10 | .05 | .05 | .05 | .10 |

[Nov/Dec 2019]

Definition about Optimal Binary search trees.

Refer Examples from Anany Levitin. "Introduction to Design and Analysis of Algorithms".

## **7. Write Huffman code algorithm and derive its complexity. [Apr/May 2019]**

Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

**There are mainly two major parts in Huffman Coding**

- 1) Build a Huffman Tree from input characters.
- 2) Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Refer Examples from Anany Levitin. “Introduction to Design and Analysis of Algorithms”.

## **10. Outline the Dynamic Programming approach to solve the Optimal Binary Search Tree problem and analyze it time complexity. [Nov/Dec 2019]**

Given a sorted array  $\text{keys}[0.. n-1]$  of search keys and an array  $\text{freq}[0.. n-1]$  of frequency counts, where  $\text{freq}[i]$  is the number of searches to  $\text{keys}[i]$ . Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.

Let us first define the cost of a BST. The cost of a BST node is level of that node multiplied by its frequency. Level of root is 1.

Input:  $\text{keys}[] = \{10, 12\}$ ,  $\text{freq}[] = \{34, 50\}$

There can be following two possible BSTs

10

12

\

/

12

10

I

II

Frequency of searches of 10 and 12 are 34 and 50 respectively.

The cost of tree I is  $34*1 + 50*2 = 134$

The cost of tree II is  $50*1 + 34*2 = 118$

Input:  $\text{keys}[] = \{10, 12, 20\}$ ,  $\text{freq}[] = \{34, 8, 50\}$

There can be following possible BSTs

10

12

20 10 20

20

\

/

\

/

/

12

10

20

12

20

10

\

/

/

/

\

20

10

12

12

12

I

II

III

IV

V

Among all possible BSTs, cost of the fifth BST is minimum.

Cost of the fifth BST is  $1*50 + 2*34 + 3*8 = 142$

### 1) Optimal Substructure:

The optimal cost for  $\text{freq}[i..j]$  can be recursively calculated using following formula.

$$\text{optcost}(i, j) = \sum_{k=i}^{j-1} \text{freq}[k] + \min_{r=i}^{j-1} (\text{optcost}(i, r-1) + \text{optcost}(r+1, j))$$

We need to calculate  $\text{optCost}(0, n-1)$  to find the result.

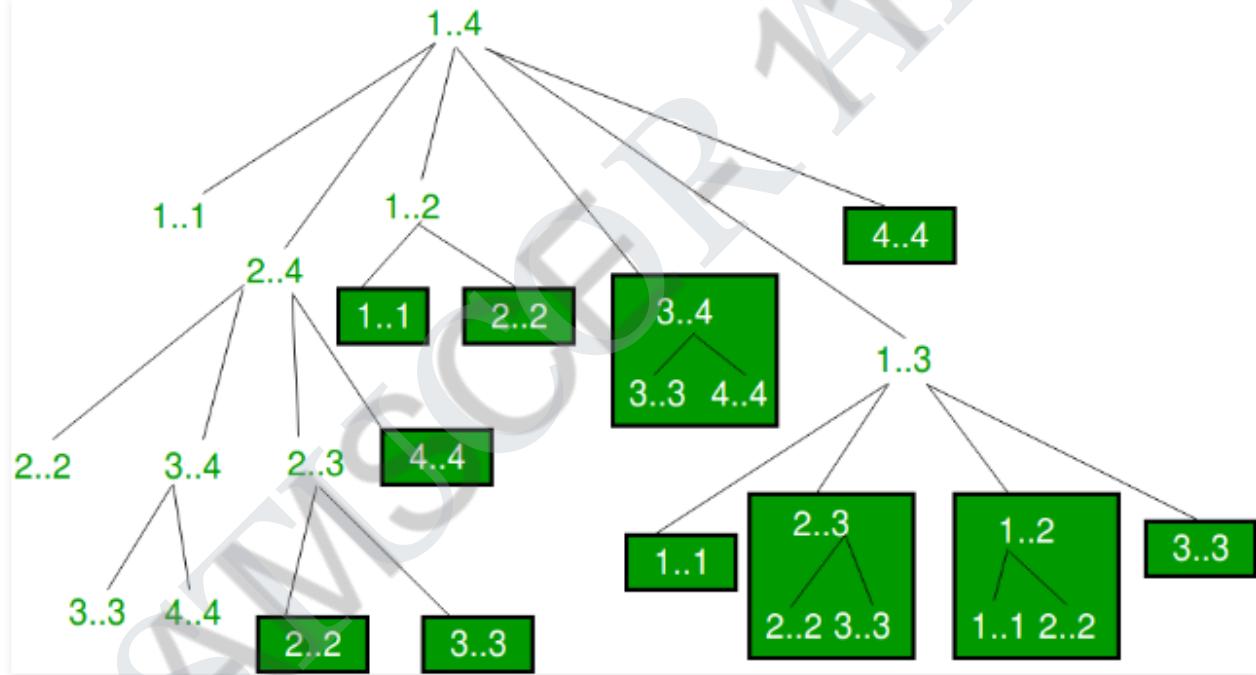
The idea of above formula is simple, we one by one try all nodes as root (r varies from i to j in second term). When we make rth node as root, we recursively calculate optimal cost from i to r-1 and r+1 to j.

We add sum of frequencies from  $i$  to  $j$  (see first term in the above formula), this is added because every search will go through root and one comparison will be done for every search.

## 2) Overlapping Subproblems

Following is recursive implementation that simply follows the recursive structure mentioned above.

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. We can see many subproblems being repeated in the following recursion tree for  $\text{freq}[1..4]$ .



## UNIT- IV ITERATIVE IMPROVEMENT

### PART A

**1. State the Principle of Duality. [Apr/May 2019]**

The **principle of duality** in Boolean algebra **states** that if you have a true Boolean statement (equation) then the **dual** of this statement (equation) is true. The **dual** of a boolean statement is found by replacing the statement's symbols with their counterparts.

**2. Define the Capacity Constraint in the context of Maximum flow problem. [Apr/May 2019]**

It represents the **maximum** amount of **flow** that can pass through an edge. for each (**capacity constraint**: the **flow** of an edge cannot exceed its **capacity**) for each (**conservation of flows**: the sum of the **flows** entering a node must equal the sum of the **flows** exiting a node, except for the source and the sink nodes)

**3. Define iterative improvement technique [Nov/Dec 2018]**

This is a computational technique in which with the help of initial feasible solution the optimal solution is obtained iteratively until no improvement is found.

**4. What is solution space? Give an example [Nov/Dec 2018]**

In mathematical optimization, a feasible region, feasible set, search space, or solution space is the set of all possible points (sets of values of the choice variables) of an optimization problem that satisfy the problem's constraints, potentially including inequalities, equalities, and integer constraints. In linear programming problems, the feasible set is a convex polytope.

**5. What is articulation point in graph? [Apr/May 2017]**

A vertex in an undirected connected **graph** is an **articulation point** (or cut vertex) iff removing it (and edges through it) disconnects the **graph**. It can be thought of as a single **point** of failure.

**6. What is state space graph? [May/June 2016]**

Graph organization of the solution space is state space tree.

**7. What do you mean by ‘perfect matching’ in bipartite graph? [Apr/May 2015]**

A perfect matching of a graph is a matching (i.e., an independent edge set) in which every vertex of the graph is incident to exactly one edge of the matching. A perfect matching is therefore a matching containing  $n/2$  edges (the largest possible), meaning perfect matching are only possible on graphs with an even number of vertices.

**8. Define Flow ‘cut’. [Apr/May 2015]**

It contains exactly one vertex with no entering edges; this vertex is called the **source** and assumed to be numbered 1. It contains exactly one vertex with no leaving edges; this vertex is called the **sink** and assumed to be numbered  $n$ . The weight  $uij$  of each directed edge  $(i, j)$  is a positive integer, called the edge **capacity**. (This number represents the upper bound on the amount of the material that can be sent from  $i$  to  $j$  through a link represented by this edge.) A digraph satisfying these properties is called a **flownetwork** or simply a **network**.

Cut is a collection of arcs such that if they are removed there is no path from source to sink

**9. What is maximum cardinality matching? [Nov/Dec 2016]**

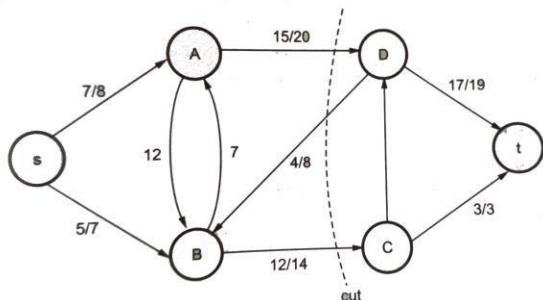
A **maximum matching** (also known as **maximum-cardinality matching**) is a **matching** that contains the largest possible number of edges. There may be many **maximum matchings**. The

matching number of a graph is the size of a **maximum matching**.

#### 10. Define Network Flow and Cut[Apr/May 2015, Nov/Dec 2015]

A network flow graph  $G = (V, E)$  is a directed graph with two special vertices: the source vertex  $s$ , and the sink (destination) vertex  $t$ . A flow network (also known as a transportation network) is a directed graph where each edge has a capacity and each edge receives a flow. The amount of flow on an edge cannot exceed the capacity of the edge.

A cut is a collection of arcs such that if they are removed there is no path from  $s$  to  $t$



A cut is said to be minimum in a network whose capacity is minimum over all cuts of the network.

#### 11. What is meant by Bipartite Graph?[Nov/Dec 2017]

A Bipartite Graph  $G = (V, E)$  is a graph in which the vertex set  $V$  can be divided into two disjoint subsets  $X$  and  $Y$  such that every edge  $e \in E$  has one end point in  $X$  and the other end point in  $Y$ . A matching  $M$  is a subset of edges such that each node in  $V$  appears in at most one edge in  $M$ .

#### 12. Give the Floyd's algorithm

```

ALGORITHM Floyd(
    W[1..n,1..n])
    //Implements Floyd's algorithm for the all-pair shortest-path problem
    //Input The weight matrix W of a graph
    //Output The distance matrix of the shortest paths' lengths
    D  $\downarrow$  W
    for k  $\downarrow$  1 to n do
        for i  $\downarrow$  1 to n do
            for j  $\downarrow$  1 to n do
                D[i,j]  $\downarrow$  min{D[i,j], D[i,k] + D[k,j]}
    return D
  
```

#### 13. Define Ford – Fulkerson Method.

1. Start with the zero flow ( $x_{ij} = 0$  for every edge)
2. On each iteration, try to find a *flow-augmenting path* from source to sink, which a path along which some additional flow can be sent
3. If a flow-augmenting path is found, adjust the flow along the edges of this path to get a flow of increased value and try again
4. If no flow-augmenting path is found, the current flow is maximum
5. How to find flow augmenting path in Network flow problem

#### 14. Define Stable Marriage Problem.

SMP (Stable Marriage Problem) is the problem of finding a stable matching between two sets of elements given a set of preferences for each element.

**15. State the extremepoints**

Any LP Problem with a nonempty bounded feasible region has an optimal solution; moreover an optimal solution can always be founded at an extreme point of the problem's feasible region.

This theorem implies that to solve a linear programming problem. at least in the case of a bounded feasible region. We can ignore all but a finite number of points in its feasible region.

**16. Write the three important things in Ford-Fulkerson method.**

- 1.Residual network
- 2.Augmenting path
- 3.Cuts

**17.How is a transportation network represented? [APR/MAY 2018 ]**

Transportation networks generally refer to a set of links, nodes, and lines that represent the infrastructure or supply side of the transportation. The links have characteristics such as speed and capacity for roadways; frequency and travel time data are defined on transit links or lines for the transit system.

**18. When a linear programming is said to be unbounded? [Nov/Dec 2019]**

An unbounded solution of a linear programming problem is a situation where objective function is infinite. A linear programming problem is said to have unbounded solution if its solution can be made infinitely large without violating any of its constraints in the problem. Since there is no real applied problem which has infinite return, hence an unbounded solution always represents a problem that has been incorrectly formulated.

**19. What is residual network in the context of flow networks? [Nov/Dec 2019]**

Residual Graph of a flow network is a graph which indicates additional possible flow. If there is a path from source to sink in residual graph, then it is possible to add flow. Every edge of a residual graph has a value called residual capacity which is equal to original capacity of the edge minus current flow.

**PART B & C****1. Explain in detail about Simplex Method.**

Every LP Problem can be represented in such form

Maximize  $3x+5y$

Subject to  $x+y \leq 4$

$x \geq 0, y \geq 0$

Maximize  $3x+5y+0u+0v$

Subject to  $x+y+u=4$

$x+3y+v=6$

**There is a 1-1 correspondence between extreme points of LP's feasible region and its basic feasible solutions.**

$$\begin{array}{ll} \text{maximize} & z = 3x + 5y + 0u + 0v \\ \text{subject to} & x + y + u = 4 \\ & x + 3y + v = 6 \\ & x \geq 0, y \geq 0, u \geq 0, v \geq 0 \text{ basic feasible solution} \\ & (0, 0, 4, 6) \end{array}$$

### Simplex method:

Step 0 [Initialization] Present a given LP problem in standard form and set up initial tableau.

Step 1 [Optimality test] If all entries in the objective row are nonnegative — stop: the tableau represents an optimal solution.

Step 2 [Find entering variable] Select (the most) negative entry in the objective row. Mark its column to indicate the entering variable and the pivot column.

Step 3 [Find departing variable] For each positive entry in the pivot column, calculate the  $\theta$ -ratio by dividing that row's entry in the rightmost column by its entry in the pivot column. (If there are no positive entries in the pivot column — stop: the problem is unbounded.) Find the row with the smallest  $\theta$ -ratio, mark this row to indicate the departing variable and the pivot row.

Step 4 [Form the next tableau] Divide all the entries in the pivot row by its entry in the pivot column. Subtract from each of the other rows, including the objective row, the new pivot row multiplied by the entry in the pivot column of the row in question. Replace the label of the pivot row by the variable's name of the pivot column and go back to Step 1.

$$\begin{array}{ll} \text{maximize } z = 3x + 5y + 0u + 0v \\ \text{subject to } x + y + u = 4 \\ x + 3y + v = 6 \\ x \geq 0, y \geq 0, u \geq 0, v \geq 0 \end{array}$$

Refer, Anany Levitin's "Introduction to Design and Analysis of Algorithms" for problem solution.

## 2. Explain in detail about Maximum Flow Problem [Apr/May 2019]

Problem of maximizing the flow of a material through a transportation network (e.g., pipeline system, communications or transportation networks)

Formally represented by a connected weighted digraph with  $n$  vertices numbered from 1 to  $n$  with the following properties:

- ✓ contains exactly one vertex with no entering edges, called the *source* (numbered 1)
- ✓ contains exactly one vertex with no leaving edges, called the *sink* (numbered  $n$ )
- ✓ has positive integer weight  $u_{ij}$  on each directed edge  $(i,j)$ , called the *edge capacity*, indicating the upper bound on the amount of the material that can be sent from  $i$  to  $j$  through this edge

### Definition of flow:

A *flow* is an assignment of real numbers  $x_{ij}$  to edges  $(i,j)$  of a given network that satisfy the following:

- **flow-conservation requirements:** The total amount of material entering an intermediate vertex must be equal to the total amount of the material leaving the vertex
- **capacity constraints**

$$0 \leq x_{ij} \leq u_{ij} \text{ for every edge } (i,j) \in E$$

### Flow value and Maximum Flow Problem

Since no material can be lost or added to by going through intermediate vertices of the network, the total amount of the material leaving the source must end up at the sink:

$$\sum_{j: (1,j) \in E} x_{1j} = \sum_{j: (j,n) \in E} x_{jn}$$

The *value* of the flow is defined as the total outflow from the source (= the total inflow into the sink).

### Maximum-Flow Problem as LP problem

$$\text{Maximize } v = \sum_{j: (1,j) \in E} x_{1j}$$

$$j: (1,j) \in E$$

### Augmenting Path (Ford-Fulkerson) Method

- Start with the zero flow ( $x_{ij} = 0$  for every edge)
- On each iteration, try to find a *flow-augmenting path* from source to sink, which a path along which some additional flow can be sent
- If a flow-augmenting path is found, adjust the flow along the edges of this path to get a flow of increased value and try again
- If no flow-augmenting path is found, the current flow is maximum

### Finding a Flow augmenting Path

To find a flow-augmenting path for a flow  $x$ , consider paths from source to sink in the underlying undirected graph in which any two consecutive vertices  $i,j$  are either:

- connected by a directed edge ( $i$  to  $j$ ) with some positive unused capacity  $r_{ij} = u_{ij} - x_{ij}$  known as *forward edge* ( $\rightarrow$ )
- connected by a directed edge ( $j$  to  $i$ ) with positive flow  $x_{ji}$  known as *backward edge* ( $\leftarrow$ )

If a flow-augmenting path is found, the current flow can be increased by  $r$  units by increasing  $x_{ij}$  by  $r$  on each forward edge and decreasing  $x_{ji}$  by  $r$  on each backward edge where  $r = \min \{r_{ij} \text{ on all forward edges, } x_{ji} \text{ on all backward edges}\}$

- Assuming the edge capacities are integers,  $r$  is a positive integer
- On each iteration, the flow value increases by at least 1
- Maximum value is bounded by the sum of the capacities of the edges leaving the source; hence the augmenting-path method has to stop after a finite number of iterations

- The final flow is always maximum, its value doesn't depend on a sequence of augmenting paths used
- The augmenting-path method doesn't prescribe a specific way for generating flow-augmenting paths
- Selecting a bad sequence of augmenting paths could impact the method's efficiency.

### Definition of a Cut:

Let  $X$  be a set of vertices in a network that includes its source but does not include its sink, and let  $X^c$ , the complement of  $X$ , be the rest of the vertices including the sink. The *cut* induced by this partition of the vertices is the set of all the edges with a tail in  $X$  and a head in  $X^c$ .

*Capacity of a cut* is defined as the sum of capacities of the edges that compose the cut.

- We'll denote a cut and its capacity by  $C(X, X^c)$  and  $c(X, X^c)$
- Note that if all the edges of a cut were deleted from the network, there would be no directed path from source to sink
- *Minimum cut* is a cut of the smallest capacity in a given network

### Max-Flow Min-Cut Theorem

- The value of maximum flow in a network is equal to the capacity of its minimum cut
- The shortest augmenting path algorithm yields both a maximum flow and a minimum cut:
  - maximum flow is the final flow produced by the algorithm
  - minimum cut is formed by all the edges from the labeled vertices to unlabeled vertices on the last iteration of the algorithm
  - all the edges from the labeled to unlabeled vertices are full, i.e., their flow amounts are equal to the edge capacities, while all the edges from the unlabeled to labeled vertices, if any, have zero flow amounts on them.

Refer, Anany Levitin's "Introduction to Design and Analysis of Algorithms" for problem solution.

### 3. Outline the stable marriage problem with example. [Nov/Dec 2019]

The Stable Marriage Problem states that given  $N$  men and  $N$  women, where each person has ranked all members of the opposite sex in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. If there are no such people, all the marriages are "stable".

Consider the following example.

Let there be two men  $m_1$  and  $m_2$  and two women  $w_1$  and  $w_2$ .

Let  $m_1$ 's list of preferences be  $\{w_1, w_2\}$

Let  $m_2$ 's list of preferences be  $\{w_1, w_2\}$

Let  $w_1$ 's list of preferences be  $\{m_1, m_2\}$

Let  $w_2$ 's list of preferences be  $\{m_1, m_2\}$

The matching  $\{ \{m_1, w_2\}, \{w_1, m_2\} \}$  is not stable because  $m_1$  and  $w_1$  would prefer each other over their assigned partners. The matching  $\{m_1, w_1\}$  and  $\{m_2, w_2\}$  is stable because there are no two people of opposite sex that would prefer each other over their assigned partners.

It is always possible to form stable marriages from lists of preferences (See references for proof). Following is Gale–Shapley algorithm to find a stable matching:

The idea is to iterate through all free men while there is any free man available. Every free man goes to all women in his preference list according to the order. For every woman he goes to, he checks if the woman is free, if yes, they both become engaged. If the woman is not free, then the woman chooses either says no to him or dumps her current engagement according to her preference list. So an engagement done once can be broken if a woman gets better option. Time Complexity of Gale-Shapley Algorithm is  $O(n^2)$ .

```

Initialize all men and women to free
while there exist a free man m who still has a woman w to propose to
{
    w = m's highest ranked such woman to whom he has not yet proposed
    if w is free
        (m, w) become engaged
    else some pair (m', w) already exists
        if w prefers m to m'
            (m, w) become engaged
            m' becomes free
        else
            (m', w) remain engaged
}

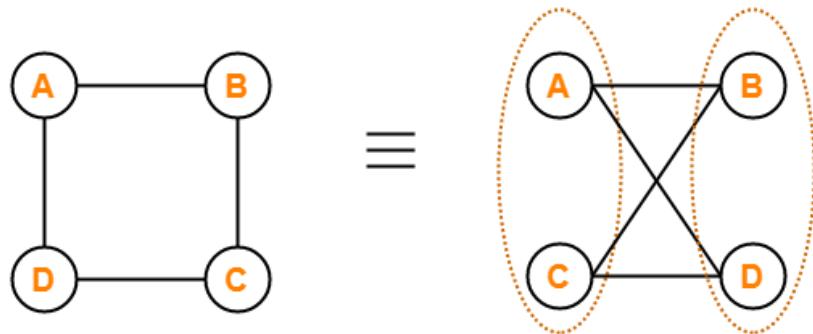
```

Input is a 2D matrix of size  $(2*N) * N$  where  $N$  is number of women or men. Rows from 0 to  $N-1$  represent preference lists of men and rows from  $N$  to  $2*N - 1$  represent preference lists of women. So men are numbered from 0 to  $N-1$  and women are numbered from  $N$  to  $2*N - 1$ . The output is list of married pairs.

#### **4. What is bipartite graph? Is the subset of bipartite graph is bipartite? Outline with example. [Nov/Dec 2019]**

A bipartite graph is a special kind of graph with the following properties-

- It consists of two sets of vertices X and Y.
- The vertices of set X join only with the vertices of set Y.
- The vertices within the same set do not join.



### Example of Bipartite Graph

Here,

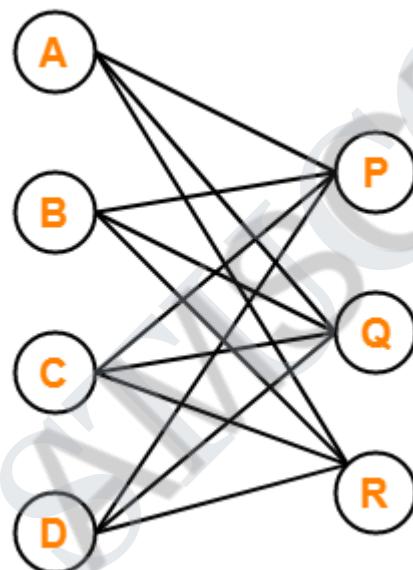
The vertices of the graph can be decomposed into two sets.

The two sets are  $X = \{A, C\}$  and  $Y = \{B, D\}$ .

The vertices of set X join only with the vertices of set Y and vice-versa.

The vertices within the same set do not join.

Therefore, it is a bipartite graph.



Complete Bi-partite graph.

Here,

This graph is a bipartite graph as well as a complete graph.

Therefore, it is a complete bipartite graph.

This graph is called as  $K_{4,3}$ .

Refer Anany Levitin's "Introduction to Design and Analysis of Algorithms" for example.

**7. Solve the following equation using Simplex Method. [Apr/May 2019]**

Maximize:  $18x_1 + 12.5x_2$

Subject to  $x_1 + x_2 \leq 20$

$x_1 \leq 12$

$x_2 \leq 16$

$x_1, x_2 \geq 0$

Refer Anany Levitin's "Introduction to Design and Analysis of Algorithms" for example and notes for solution.

## UNIT V COPING WITH LIMITATIONS OF ALGORITHMIC POWER

### PART A

#### **1. Define NP Completeness and NP Hard. [Apr/May 2019]**

The problems whose solutions have computing times are bounded by polynomials of small degree.

#### **2. State Hamiltonian Circuit Problem [Nov/Dec 2019, Apr/May 2019]**

Hamiltonian circuit problem is a problem of finding a Hamiltonian circuit. Hamiltonian circuit is a circuit that visits every vertex exactly once and return to the starting vertex.

#### **3. Define P and NP Problems. [Nov/Dec 2018]**

In computational complexity theory, P, also known as PTIME or DTIME(n), is a fundamental complexity class. It contains all decision problems that can be solved by a deterministic Turing machine using a polynomial amount of computation time, or polynomial time.

**NP:** the class of decision problems that are solvable in polynomial time on a nondeterministic machine (or with a nondeterministic algorithm). (A deterministic computer is what we know). A nondeterministic computer is one that can “guess” the right answer or solution think of a nondeterministic computer as a parallel machine that can freely spawn an infinite number of processes

#### **4. Define sum of subset Problem. [Nov/Dec 2018]**

Subset sum problem is a problem, which is used to find a subset of a given set  $S=\{S_1, S_2, S_3, \dots, S_n\}$  of  $n$  positive integers whose sum is equal to given positive integer  $d$ .

#### **5. Differentiate Feasible and Optimal Solution. [Nov/Dec 2017]**

Feasible solution means set which contains all the possible solution which follow all the constraints.

An **optimal solution** is a feasible **solution** where the objective function reaches its maximum (or minimum) value – for example, the most profit or the least cost. A globally **optimal solution** is one where there are no other feasible **solutions** with better objective function values

#### **6. Explain Promising and Non-Promising Node [Nov/Dec 2017]**

A node in a state-space tree is said to be promising if it corresponds to a partially constructed solution that may still lead to a complete solution.

A node in a state-space tree is said to be promising if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise it is called non-promising.

**7. State the reason for terminating search path at the current node in branch and bound algorithm.[Nov/Dec 2016]**

In general, we terminate a search path at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:

- The value of the node's bound is not better than the value of the best solution seen so far.
- The node represents no feasible solutions because the constraints of the problem are already violated.
- The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made)—in this case we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

**8. How is lower bound found by problem reduction? [Apr/May 2018]**

If problem P is at least as hard as problem Q ,then a lower bound for Q is also a lower bound for P. Hence ,find problem Q with a known lower bound that can be reduced to problem P in question. then any algorithm that solves P will also solve Q.

**9. What are tractable and non-tractable problems? [Apr/May 2018]**

**Problems** that are solvable by polynomial time algorithms as being **tractable**, and **problems** that require super polynomial time as being **intractable**.

- Sometimes the line between what is an 'easy' **problem** and what is a 'hard' **problem** is a fine one

**10. Compare backtracking and branch bound techniques.**

Backtracking is applicable only to non optimization problems.

Backtracking generates state space tree in depth first manner.

Branch and bound is applicable only to optimization problem.

Branch and bound generated a node of state space tree using best first rule.

**11.What are the searching techniques that are commonly used in Branch-and-Bound method.**

The searching techniques that are commonly used in Branch-and-Bound method are:

- i. FIFO
- ii. LIFO
- iii. LC
- iv. Heuristic search

**12.Illustrate 8 – Queens problem.**

The problem is to place eight queens on a 8 x 8 chessboard so that no two queen “attack” that is, so that no two of them are on the same row, column or on the diagonal.

**13. Show the purpose of lower bound. [MAY/JUNE 2016]**

Lower bound of a problem is an estimate on a minimum amount of work needed to solve a given problem

**14.Define chromatic number of the graph.**

The m – color ability optimization problem asks for the smallest integer m for which the graph G can be colored. This integer is referred to as the chromatic number of the graph.

**15.What is Absolute approximation?**

A is an absolute approximation algorithm if there exists a constant k such that, for every instance I of P,  $|A^*(I) - A(I)| \leq k$ . For example, Planar graph coloring.

**16.What is Relative approximation?**

A is an relative approximation algorithm if there exists a constant k such that, for every instance I of P,  $\max\{A^*(I), A(I)\} / A^*(I) \leq k$ . For example, Vertex cover.

**17.Show the application for Knapsack problem**

The Knapsack problem is problem in combinatorial optimization. It derives its name from the maximum problem of choosing possible essential that can fit into one bag to be carried on a trip. A similar problem very often appears in business, combinatory, complexitytheory, cryptography and applied mathematics.

**18. Define Branch-and-Bound method.**

The term Branch-and-Bound refers to all the state space methods in which all children of the E-node are generated before any other live node can become the E-node.

**19. Define backtracking.**

Depth first node generation with bounding function is called backtracking. The backtracking algorithm has its virtue the ability to yield the answer with far fewer than m trials.

**20.List the factors that influence the efficiency of the backtracking algorithm?**

The efficiency of the backtracking algorithm depends on the following four factors. They are:

- The time needed to generate the next  $x_{BTL}$
- The number of  $x_k$  satisfying the explicit constraints.
- The time for the bounding functions  $B_k$
- The number of  $x_k$  satisfying the  $B_k$

**21. When is a problem said to be NP Hard?**

A problem is said to be NP-hard if everything in NP can be transformed in polynomial time into it, and a problem is NP-complete if it is both in NP and NP-hard.

**PART-B & C****1. Elaborate how backtracking technique can be used to solve the n-queens problem. Explain with an example. [Nov/Dec 2019]**

The problem is to place eight queens on a  $8 \times 8$  chessboard so that no two queen -attack|| that is, so that no two of them are on the same row, column or on the diagonal.

|   |  |   |   |  |   |   |   |
|---|--|---|---|--|---|---|---|
|   |  |   | Q |  |   |   |   |
|   |  |   |   |  | Q |   |   |
|   |  |   |   |  |   |   | Q |
|   |  | Q |   |  |   |   |   |
|   |  |   |   |  |   |   | Q |
|   |  |   |   |  |   | Q |   |
| Q |  |   |   |  |   |   |   |
|   |  |   | Q |  |   |   |   |

**Algorithm**

**isValid(board, row, col)**

**Input:** The chess board, row and the column of the board.

**Output:** True when placing a queen in row and place position is a valid or not.

Begin

```
if there is a queen at the left of current col, then
    return false
if there is a queen at the left upper diagonal, then
    return false
if there is a queen at the left lower diagonal, then
    return false;
return true //otherwise it is valid place
```

End

**solveNQueen(board, col)**

**Input:** The chess board, the col where the queen is trying to be placed.

**Output:** The position matrix where queens are placed.

Begin

```
if all columns are filled, then
    return true
for each row of the board, do
    if isValid(board, i, col), then
        set queen at place (i, col) in the board
        if solveNQueen(board, col+1) = true, then
            return true
        otherwise remove queen from place (i, col) from board.
```

done

return false

End

## 2. Explain Backtracking technique.

Backtracking technique is a refinement of this approach. Backtracking is a surprisingly simple approach and can be used even for solving the hardest Sudoku puzzle.

Problems that need to find an element in a domain that grows exponentially with the size of the input, like the Hamiltonian circuit and the Knapsack problem, are not solvable in polynomial time. Such problems can be solved by the exhaustive search technique, which requires identifying the correct solution from many candidate solutions.

### Steps to achieve Goal:

- Backtracking possible by constructing the state-space tree, this is a tree of choices.
- The root of the state-space tree indicates the initial state, before the search for the solution begins.
- The nodes of each level of this tree signify the candidate solutions for the corresponding component.
- A node of this tree is considered to be promising if it represents a partially constructed solution that can lead to a complete solution, else they are considered to be non-promising.
- The leaves of the tree signify either the non-promising dead-ends or the complete solutions.
- Depth-First-search method usually for constructing these state-space-trees.

- If a node is promising, then a child-node is generated by adding the first legitimate choice of the next component and the processing continues for the child node.

### **Example**

- This technique is illustrated by the following figure.
- Here the algorithm goes from the start node to node 1 and then to node 2.
- When no solution is found it backtracks to node1 and goes to the next possible solution node
  - i. But node 3 is also a dead-end.

**3.**

### **4. Give solution to Hamiltonian circuit using Backtracking technique**

- The graph of the Hamiltonian circuit is shown below.
- In the below Hamiltonian circuit, circuit starts at vertex i, which is the root of the state-space tree.
- From i, we can move to any of its adjoining vertices which are j, k, and l. We first select j, and then move to k, then l, then to m and thereon to n. But this proves to be a dead-end.
- So, we backtrack from n to m, then to l, then to k which is the next alternative solution. But moving from k to m also leads to a dead-end.
- So, we backtrack from m to k, then to j. From there we move to the vertices n, m, k, l and correctly return to i. Thus the circuit traversed is i->j->n->m->k->l->i.

### **5. Give solution to Subset sum problem using Backtracking technique [Apr/May 2019]**

- In the Subset-Sum problem, we have to find a subset of a given set  $S = \{s_1, s_2, \dots, s_n\}$  of  $n$  positive integers whose sum is equal to a positive integer  $t$ .
- Let us assume that the set  $S$  is arranged in ascending order. For example, if  $S = \{2, 3, 5, 8\}$  and if  $t = 10$ , then the possible solutions are  $\{2, 3, 5\}$  and  $\{2, 8\}$ .
- The root of the tree is the starting point and its left and right children represent the inclusion and exclusion of 2.
- Similarly, the left node of the first level represents the inclusion of 3 and the right node the exclusion of 3.
- Thus the path from the root to the node at the  $i$ th level shows the first  $i$  numbers that have been included in the subsets that the node represents.
- Thus, each node from level 1 records the sum of the numbers  $Ssum$  along the path upto that particular node.
- If  $Ssum$  equals  $t$ , then that node is the solution. If more solutions have to be found, then we can backtrack to that node's parent and repeat the process. The process is terminated for any non-promising node that meets any of the following two conditions

### **6. Explain P, NP and NP complete problems.**

#### **Definition:**

An algorithm solves a problem in polynomial time if its worst-case time efficiency belongs to  $O(P(n))$ .

$P(n)$  is a polynomial of the problem 's input size  $n$ .

Tractable:

Problems that can be solved in polynomial time are called tractable.

Intractable:

Problems that cannot be solved in polynomial time are called intractable. We cannot solve arbitrary instances in reasonable amount of time.

Huge difference between running time. Sum and composition of 2 polynomial results in polynomial .Development of extensive theory called computational complexity.

**Definition: II**

P and NP Problems:

Class P:

It is a class of decision problems that can be solved in polynomial time by deterministic algorithm, where as deterministic algorithm is every operation uniquely defined.

Decision problems:

A problem with yes/no answers called decision problems.

Restriction of P to decision problems. Sensible to execute problems not solvable in polynomial time because of their large output.

Eg: Subset of the given set

Many important problems that are decision problem can be reduced to a series of decision problems that are easier to study.

Undecidable problem:

Some decision problems cannot be solved at all by any algorithm.

Halting Problem:

Given a computer program and an input to it, determine whether the program halt at input or continue work indefinitely on it.

**Definition: III (A non-deterministic algorithm)**

Here two stage procedure:

Non-deterministic (Guessing stage)

Deterministic (Verification Stage)

A Non-deterministic polynomial means, time efficiency of its verification stage is polynomial.

**Definition: IV**

Class NP is the class of decision problem that can be solved by non-deterministic polynomial algorithm. This class of problems is called non-deterministic polynomial.

$$P \subseteq NP$$

$P=NP$  imply many hundreds of difficult combinational decision problem can be solved by polynomial time

**NP-Complete:**

Many well-known decision problems known to be NP-Complete where  $P=NP$  is more doubt. Any problem can be reduced to polynomial time.

**Definition: V**

A decision problem D1 is polynomially reducible to a decision problem D2. If there exists a function t, t transforms instances D1 to D2

**Definition: VI**

A decision problem D is said to be NP Complete if it belongs to class NP. Every problem in NP is polynomially reducible to p.

**7. Explain the approximation algorithm for the travelling salesman problem (TSP) [Apr/May 2019]**

- a. Nearest neighbor algorithm
- b. Twice –around the tree algorithm

**Nearest Neighbor algorithm:**

- i. Simply greedy method
- ii. Based on the nearest neighbor heuristic
- iii. Always go for nearest unvisited city

**Algorithm:**

Step1: Choose the arbitrary city as the start

Step2: Repeat all cities (unvisited)

Step3: Return to the starting city

It is difficult to solve the travelling salesman problem approximately.

However, there is a very important subset of instances called Euclidean, for which we can make a non-trivial assertion about the accuracy of the algorithm.

➤ Triangle inequality:

$$d[i,j] \leq d[i,k] + d[k,j] \text{ for any triple of cities}$$

➤ Symmetry:

$$d[i,j] = d[j,i] \text{ for any pair of}$$

cities I and j The accuracy ratio for any such instance with  $n \geq 2$  cities.

$$f(S_a)/f(S^*) \leq 1/2 \lceil \log_2 n \rceil + 1$$

where  $f(S_a)$  and  $f(S^*)$  are the length of the nearest neighbour and shortest tour respectively.

Twice –around the tree algorithm:

Twice around the tree algorithm is an approximation algorithm for the TSP with Euclidean distances. Hamiltonian circuit & spanning tree.

✓ Polynomial time:

Within polynomial time twice around the tree can be solved by prim's or kruskal's algorithm.

- length of the tour  $S_a$
- Optimal tour  $S^*$ , i.e.  $S_a$  twice of  $S^*$
- $f(S_a) \leq 2 f(S^*)$  (Hamiltonian circuit)

- Removing a edge from  $S^*$  yields spanning tree  $T$  of weight  $w(T)$   $w(T) > w(S^*)$
- $f(S^*) > w(T)/Tree \geq w(T^*)/Tree \geq f(S^*)/2 > w(T^*)$
- The Length of the walk  $\geq$  Length of the tour  $(Sa)$   $2f(S^*) > f(Sa)$

## 7. Outline the steps to find approximate solution to NP-Hard optimization problems using approximation algorithms with an example. [Nov/Dec 2019]

Given an optimization problem  $P$ , an algorithm  $A$  is said to be an approximation algorithm for  $P$ , if for any given instance  $I$ , it returns an approximate solution, that is a feasible solution.

$P$  An optimization problem

$A$  An approximation algorithm

$I$  An instance of  $P$

$A^*$

$(I)$  Optimal value for the instance  $I$

$A(I)$  Value for the instance  $I$  generated by  $A$

### 1. Absolute approximation

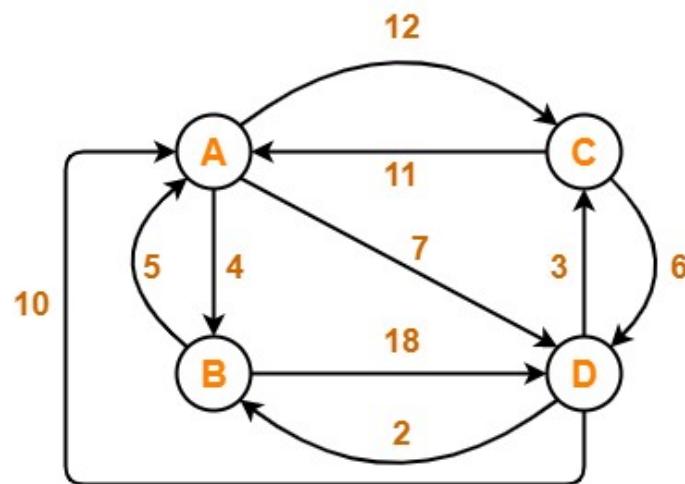
- $A$  is an absolute approximation algorithm if there exists a constant  $k$  such that, for every instance  $I$  of  $P$ ,  $|A^*(I) - A(I)| \leq k$ .
- For example, Planar graph coloring.

### 2. Relative approximation

- $A$  is an relative approximation algorithm if there exists a constant  $k$  such that, for every instance  $I$  of  $P$ ,  $\max \{A^*(I)/A(I), A(I) / A^*(I)\} \leq k$ .
- Vertex cover.

## ASSIGNMENT QUESTIONS

- 1.** Solve Travelling Salesman Problem using Branch and Bound Algorithm in the following graph-



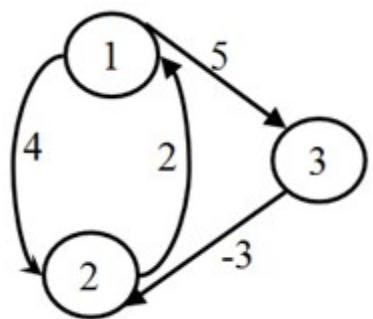
- 2.** Solve the following 0/1 Knapsack Problem using Branch & Bound Algorithm (both LC BB & FIFO BB)

[Hint: LC BB Method is explained in the Slide Uploaded. For FIFO Method the child nodes are explored in First in First out manner.]    **Capacity =10**

| ITEMS | Weights | Profits |
|-------|---------|---------|
| A     | 2       | \$40    |
| B     | 3.14    | \$50    |
| C     | 1.98    | \$100   |
| D     | 5       | \$95    |
| E     | 3       | \$30    |

- 3.** Solve the following sum of subset problem using backtracking:  $w= \{1, 3, 4, 5\}$ ,  $m=8$ . Find the possible subsets of 'w' that sum to 'm'. [Draw state space tree to find the solution]

4. Find the **shortest paths** between all pairs of vertices in a graph given below



## UNIT I

### INTRODUCTION TO DATA STRUCTURES

Introduction – Basic terminology – Data structures – Data structure operations - ADT – Algorithms: Complexity, Time – Space trade off - Mathematical notations and functions - Asymptotic notations – Linear and Binary search - Bubble sort - Insertion sort

---

#### INTRODUCTION

**Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way.**

Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have data player's name "Virat" and age 26. Here "Virat" is of String data type and 26 is of integer data type.

We can organize this data as a record like Player record. Now we can collect and store player's records in a file or database as a data structure. For example: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33.

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily.

#### Basic Terminology of Data Organization:

**Data :** The term 'DATA' simply refers to a value or a set of values. These values may present anything about something, like it may be roll no of a student, marks, name of an employee, address of person etc.

**Data item :** A data item refers to a single unit of value. For eg. roll no of a student, marks, name of an employee, address of person etc. are data items. Data items that can be divided into sub items are called **group items** (Eg. Address, date, name), where as those who can not be divided in to sub items are called **elementary items** (Eg. Roll no, marks, city, pin code etc.).

**Entity** - with similar attributes ( e.g all employees of an organization) form an entity set

**Information:** processed data, Data with given attribute

**Field** is a single elementary unit of information representing an attribute of an entity

**Record** is the collection of field values of a given entity

**File** is the collection of records of the entities in a given entity set

| Name | Age | Sex | Roll Number | Branch |
|------|-----|-----|-------------|--------|
| A    | 17  | M   | 109cs0132   | CSE    |
| B    | 18  | M   | 109ee1234   | EE     |

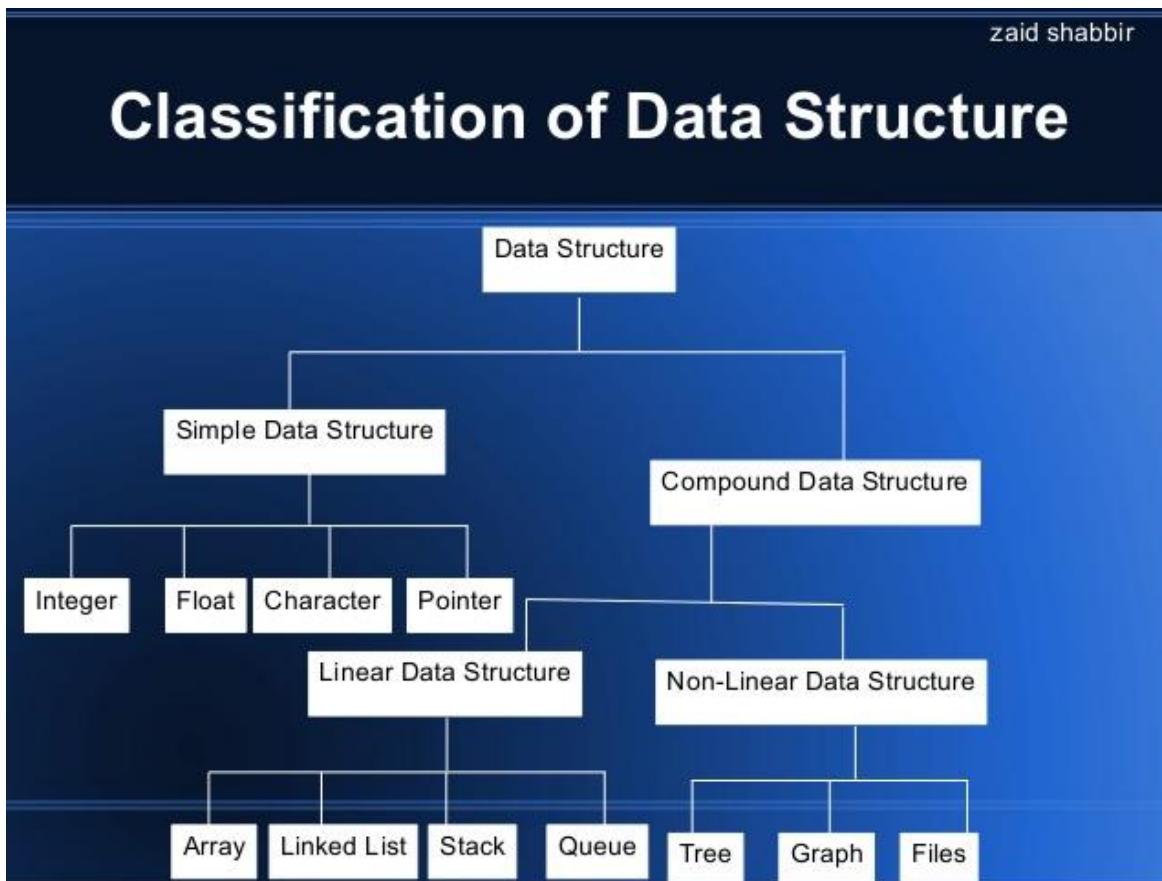
### **Basic types of Data Structures**

Anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as Primitive Data Structures.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of Abstract Data Structure are :

- Array
- Linked List
- Stack
- Queue
- Tree
- Graph

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required.



## Basic Operations

The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.

- Traversing
- Searching
- Insertion
- Deletion
- Sorting
- Merging

- (1) **Traversing:** Accessing each record exactly once so that certain items in the record may be processed.
- (2) **Searching:** Finding the location of a particular record with a given key value, or finding the location of all records which satisfy one or more conditions.
- (3) **Inserting:** Adding a new record to the structure.

- (4) **Deleting:** Removing the record from the structure.
- (5) **Sorting:** Managing the data or record in some logical order (Ascending or descending order).
- (6) **Merging:** Combining the record in two different sorted files into a single sorted file.

## Abstract Data Types (ADT)

An abstract data type (ADT) refers to a set of data values and associated operations that are specified accurately, independent of any particular implementation. With an ADT, we know what a specific data type can do, but how it actually does it is hidden. Simply hiding the implementation

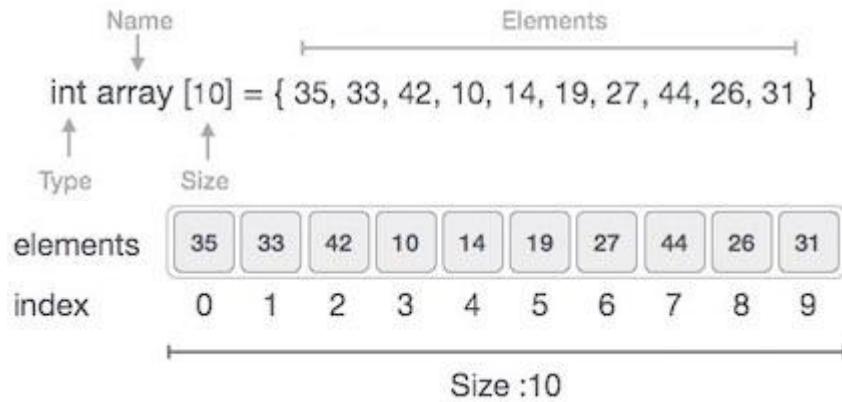
## Data Structure - Arrays

Array is a container which can hold fix number of items and these items should be of same type. Most of the data structures make use of array to implement their algorithms. Following are important terms to understand the concepts of Array.

- **Element** – each item stored in an array is called an element.
- **Index** – each location of an element in an array has a numerical index which is used to identify the element.

## Array Representation

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



As per above shown illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch element at index 6 as 27.

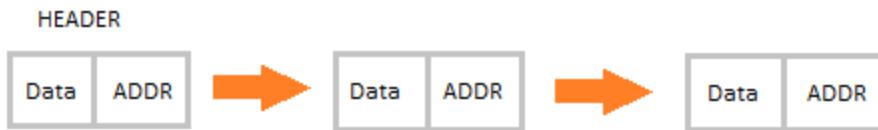
## Basic Operations

Following are the basic operations supported by an array.

- **Traverse** – print all the array elements one by one.
- **Insertion** – add an element at given index.
- **Deletion** – delete an element at given index.
- **Search** – search an element using given index or by value.
- **Update** – update an element at given index.

## Data Structure – Linked Lists

Linked List is a linear data structure and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain. Linked Lists are used to create trees and graphs.



## Advantages of Linked Lists

- They are dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.

## Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

## Applications of Linked Lists

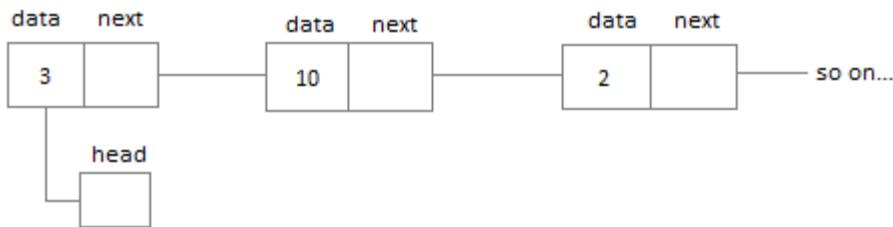
Linked lists are used to implement stacks, queues, graphs, etc.

Linked lists let you insert elements at the beginning and end of the list.

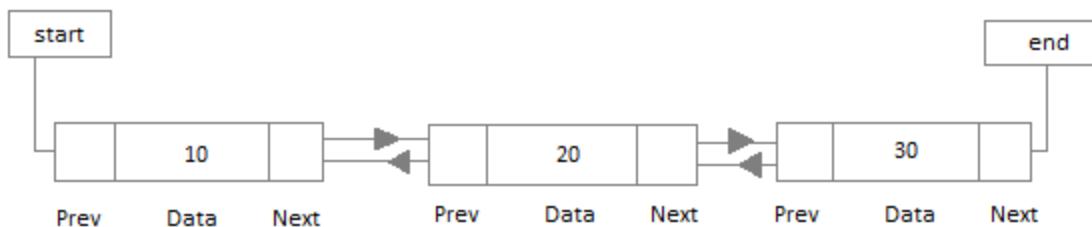
In Linked Lists we don't need to know the size in advance.

## Types of Linked Lists

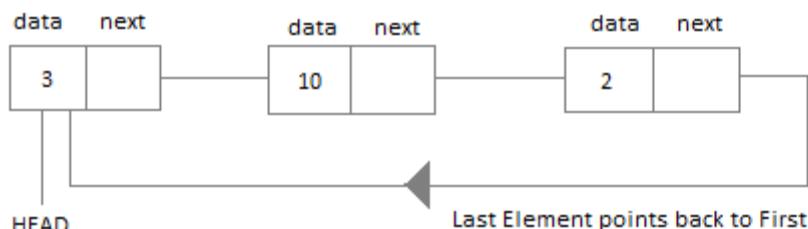
**Singly Linked List :** Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in sequence of nodes. The operations we can perform on singly linked lists are insertion, deletion and traversal.



**Doubly Linked List :** In a doubly linked list, each node contains two links the first link points to the previous node and the next link points to the next node in the sequence.



**Circular Linked List :** In the circular linked list the last node of the list contains the address of the first node and forms a circular chain.



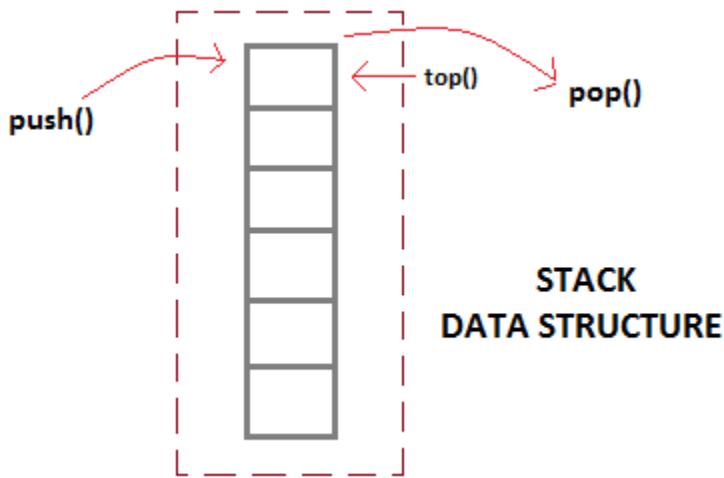
## Data Structure - Stack

### Stacks

Stack is an abstract data type with a bounded (predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an

element is added, it goes on the top of the stack, the only element that can be removed is the element that was at the top of the stack, just like a pile of objects.

## Stack Data Structure



## Basic features of Stack

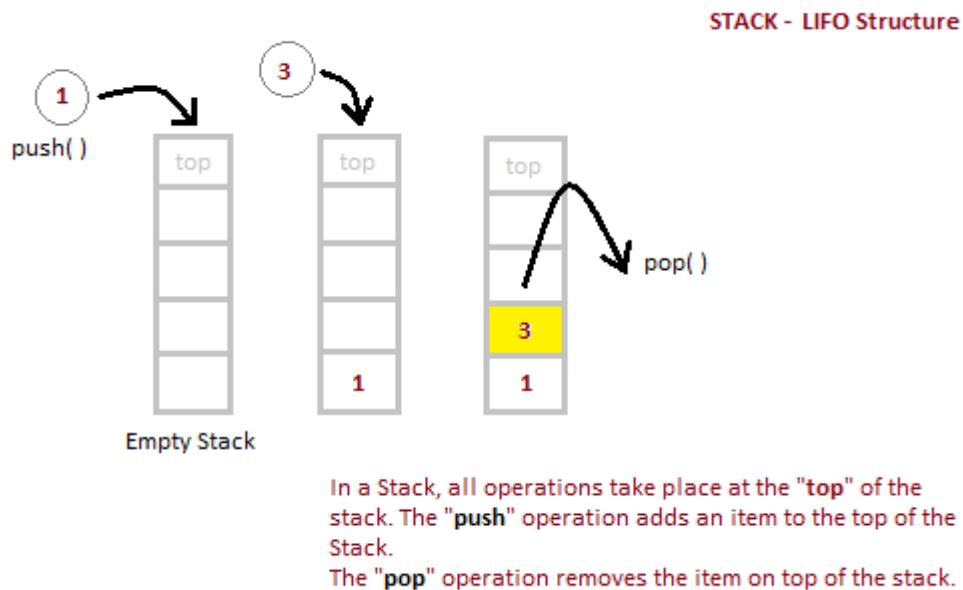
1. Stack is an ordered list of similar data type.
2. Stack is a LIFO structure. (Last in First out).
3. push() function is used to insert new elements into the Stack and pop() is used to delete an element from the stack. Both insertion and deletion are allowed at only one end of Stack called Top.
4. Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.

## Applications of Stack

- The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
- There are other uses also like : Parsing, Expression Conversion(Infix to Postfix, Postfix to Prefix etc) and many more.

## Implementation of Stack

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.



| Position of Top | Status of Stack           |
|-----------------|---------------------------|
| -1              | Stack is Empty            |
| 0               | Only one element in Stack |
| N-1             | Stack is Full             |
| N               | Overflow state of Stack   |

## Analysis of Stacks

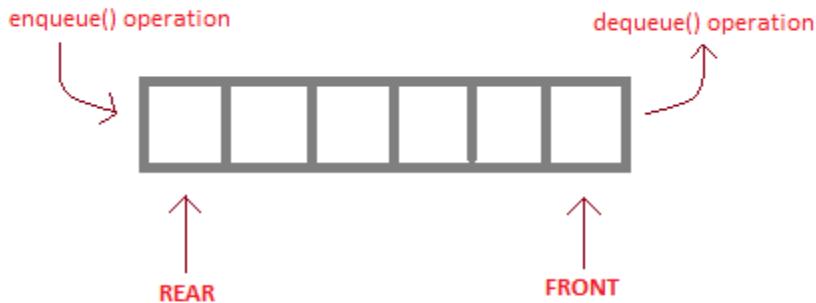
Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

- Push Operation : O(1)
- Pop Operation : O(1)
- Top Operation : O(1)
- Search Operation : O(n)

## Queue Data Structures

Queue is also an abstract data type or a linear data structure, in which the first element is inserted from one end called REAR(also called tail), and the deletion of existing element takes place from the other end called as FRONT(also called head). This makes queue as FIFO data structure, which means that element inserted first will also be removed first.

The process to add an element into queue is called Enqueue and the process of removal of an element from queue is called Dequeue.



**enqueue( )** is the operation for adding an element into Queue.

**dequeue( )** is the operation for removing an element from Queue .

### QUEUE DATA STRUCTURE

## Basic features of Queue

1. Like Stack, Queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO( First in First Out ) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. peek( ) function is oftenly used to return the value of first element without dequeuing it.

## Applications of Queue

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios :

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

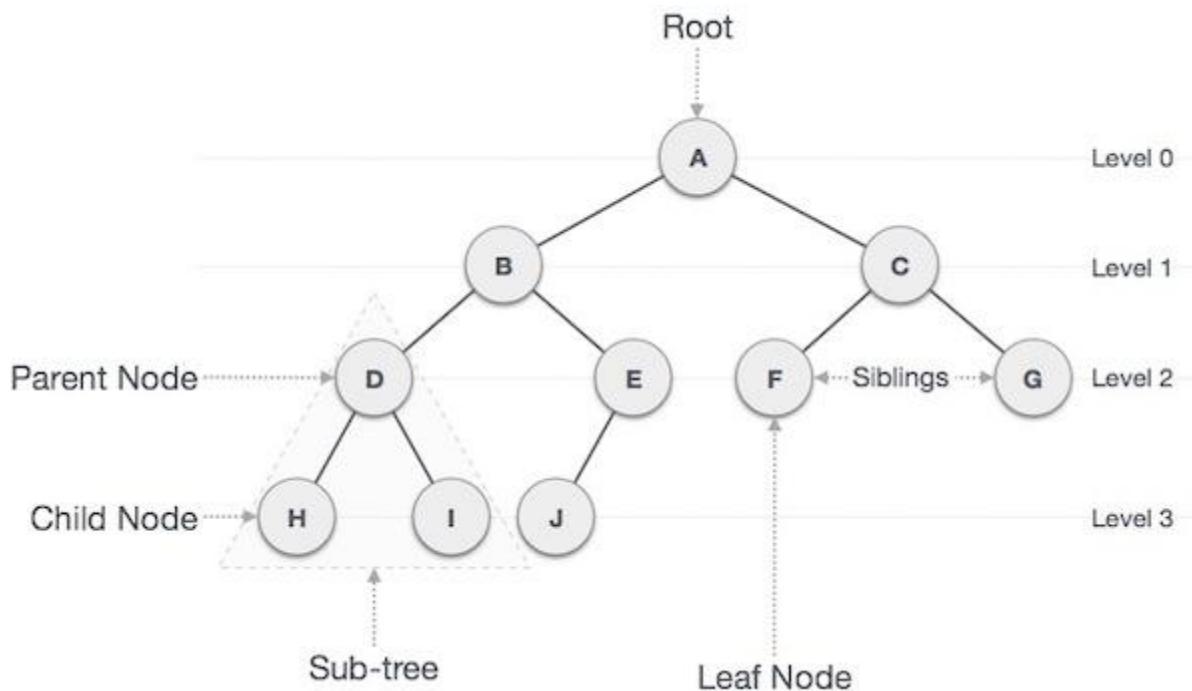
### **Analysis of Queue**

- Enqueue : O(1)
- Dequeue : O(1)
- Size : O(1)

### **Data Structure - Tree**

Tree represents nodes connected by edges. We'll going to discuss binary tree or binary search tree specifically.

Binary Tree is a special datastructure used for data storage purposes. A binary tree has a special condition that each node can have two children at maximum. A binary tree have benefits of both an ordered array and a linked list as search is as quick as in sorted array and insertion or deletion operation are as fast as in linked list.



## Terms

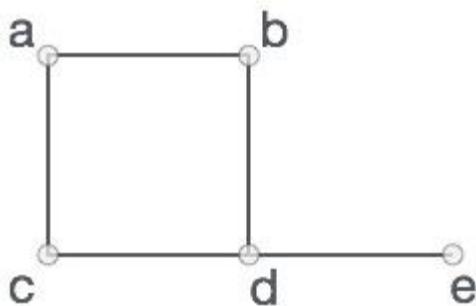
Following are important terms with respect to tree.

- **Path** – Path refers to sequence of nodes along the edges of a tree.
- **Root** – Node at the top of the tree is called root. There is only one root per tree and one path from root node to any node.
- **Parent** – Any node except root node has one edge upward to a node called parent.
- **Child** – Node below a given node connected by its edge downward is called its child node.
- **Leaf** – Node which does not have any child node is called leaf node.
- **Subtree** – Subtree represents descendants of a node.
- **Visiting** – Visiting refers to checking value of a node when control is on the node.
- **Traversing** – Traversing means passing through nodes in a specific order.
- **Levels** – Level of a node represents the generation of a node. If root node is at level 0, then its next child node is at level 1, its grandchild is at level 2 and so on.
- **keys** – Key represents a value of a node based on which a search operation is to be carried out for a node.

## Data Structure - Graph

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



In the above graph,

$$V = \{a, b, c, d, e\}$$

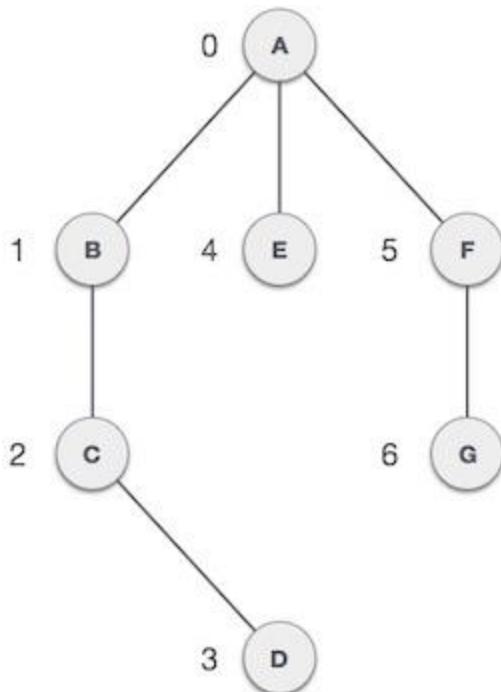
$$E = \{ab, ac, bd, cd, de\}$$

### Graph Data Structure

Mathematical graphs can be represented in data-structure. We can represent a graph using an array of vertices and a two dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms –

- **Vertex** – Each node of the graph is represented as a vertex. In example given below, labeled circle represents vertices. So A to G are vertices. We can represent them using an array as shown in image below. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** – Edge represents a path between two vertices or a line between two vertices. In example given below, lines from A to B, B to C and so on represents edges. We can use a two dimensional array to represent array as shown in image below. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

- **Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge. In example given below, B is adjacent to A, C is adjacent to B and so on.
- **Path** – Path represents a sequence of edges between two vertices. In example given below, ABCD represents a path from A to D.



## Basic Operations

Following are basic primary operations of a Graph which are following.

- **Add Vertex** – add a vertex to a graph.
- **Add Edge** – add an edge between two vertices of a graph.
- **Display Vertex** – display a vertex of a graph.

## Algorithms Basics

Algorithm is a step by step procedure, which defines a set of instructions to be executed in certain order to get the desired output. Algorithms are generally created independent

of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

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

From data structure point of view, following are some important categories of algorithms

- Search – Algorithm to search an item in a data structure.
- Sort – Algorithm to sort items in certain order
- Insert – Algorithm to insert item in a data structure
- Update – Algorithm to update an existing item in a data structure
- Delete – Algorithm to delete an existing item from a data structure

### **Characteristics of an Algorithm**

Not all procedures can be called an algorithm. An algorithm should have the below mentioned characteristics –

- Unambiguous – Algorithm should be clear and unambiguous. Each of its steps (or phases), and their input/outputs should be clear and must lead to only one meaning.
- Input – An algorithm should have 0 or more well defined inputs.
- Output – An algorithm should have 1 or more well defined outputs, and should match the desired output.
- Finiteness – Algorithms must terminate after a finite number of steps.
- Feasibility – Should be feasible with the available resources.
- Independent – An algorithm should have step-by-step directions which should be independent of any programming code.

## Algorithm Analysis

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

1. **Time Complexity**
2. **Space Complexity**

Suppose X is an algorithm and n is the size of input data, the time and space used by the Algorithm X are the two main factors which decide the efficiency of X.

- **Time Factor** – The time is measured by counting the number of key operations such as comparisons in sorting algorithm
- **Space Factor** – The space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm  $f(n)$  gives the running time and / or storage space required by the algorithm in terms of n as the size of input data.

## Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. Its the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

Space required by an algorithm is equal to the sum of the following two components –

- **A fixed part** that is a space required to store certain data and variables that are independent of the size of the problem. For example simple variables & constant used and program size etc.
- **A variable part** is a space required by variables, whose size depends on the size of the problem. For example dynamic memory allocation, recursion stacks space etc.

## An algorithm generally requires space for following components:

- **Instruction Space:** It is the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
- **Data Space:** It is the space required to store all the constants and variables value.
- **Environment Space:** It is the space required to store the environment information needed to resume the suspended function.

Space complexity  $S(P)$  of any algorithm  $P$  is  $S(P) = C + SP(I)$  Where  $C$  is the fixed part and  $S(I)$  is the variable part of the algorithm which depends on instance characteristic  $I$ . Following is a simple example that tries to explain the concept –

## Asymptotic Notations

The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

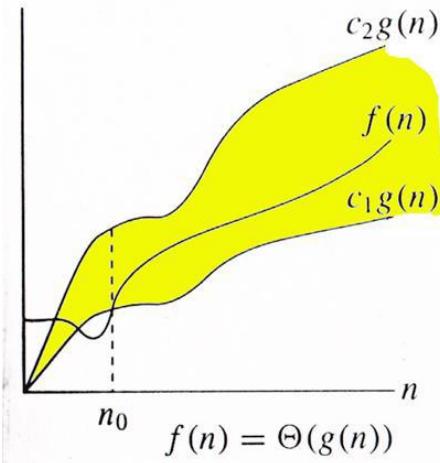
### 1) $\Theta$ Notation:

The theta notation bounds a function from above and below, so it defines exact asymptotic behavior. A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.  $3n^3 + 6n^2 + 6000 = \Theta(n^3)$

Dropping lower order terms is always fine because there will always be a  $n^0$  after which  $\Theta(n^3)$  beats  $\Theta(n^2)$  irrespective of the constants involved. 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  $n_0$ .

## 2. Big O Notation:

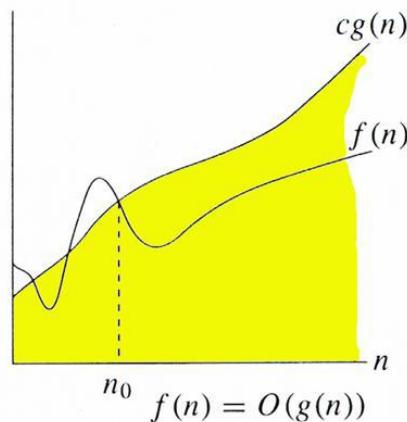
The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is  $O(n^2)$ . Note that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

1. The worst case time complexity of Insertion Sort is  $\Theta(n^2)$ .
2. The best case time complexity of Insertion Sort is  $\Theta(n)$ .

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.

$O(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$

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

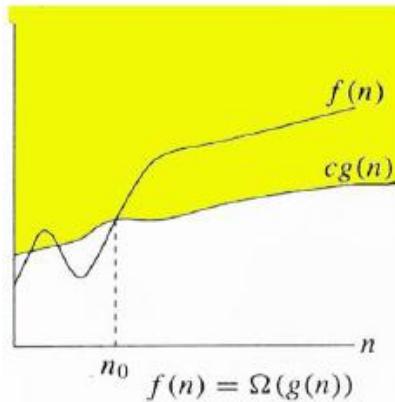


### 3) $\Omega$ Notation:

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. As discussed in the previous post, the best case performance of an algorithm is generally not useful; the Omega notation is the least used notation among all three.

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

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



## Mathematical Analysis of Recursive Algorithms

Many algorithms are recursive in nature. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size  $n$  as a function of  $n$  and the running time on inputs of smaller sizes. For example in [Merge Sort](#), to sort a given array, we divide it in two halves and recursively repeat the process for the two halves. Finally we merge the results. Time complexity of Merge Sort can be written as  $T(n) = 2T(n/2) + cn$ . There are many other algorithms like Binary Search, Tower of Hanoi, etc. There are mainly three ways for solving recurrences.

**1) Substitution Method:** We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

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

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

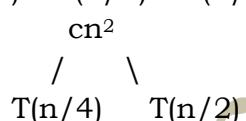
We need to prove that  $T(n) \leq cn\log n$ . We can assume that it is true for values smaller than  $n$ .

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\leq cn/2\log(n/2) + n \\ &\leq cn\log n - cn\log 2 + n \\ &\leq cn\log n - cn + n \\ &\leq cn\log n \end{aligned}$$

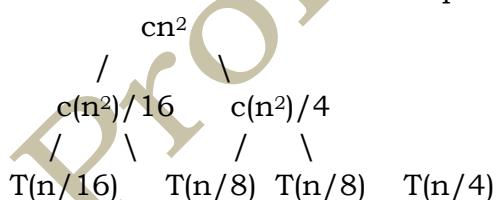
**2) Recurrence Tree Method:**

In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically an arithmetic or geometric series. For example consider the recurrence relation

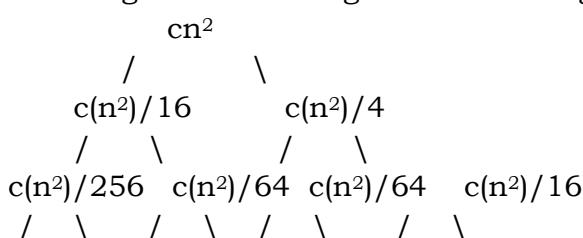
$$T(n) = T(n/4) + T(n/2) + cn^2$$



If we further break down the expression  $T(n/4)$  and  $T(n/2)$ , we get following recursion tree.



Breaking down further gives us following



To know the value of  $T(n)$ , we need to calculate sum of tree nodes level by level. If we sum the above tree level by level, we get the following series

$$T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + \dots$$

The above series is geometrical progression with ratio  $5/16$ . To get an upper bound, we can sum the infinite series. We get the sum as  $(n^2)/(1 - 5/16)$  which is  $O(n^2)$

### 3) Master Method:

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

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

There are following three cases:

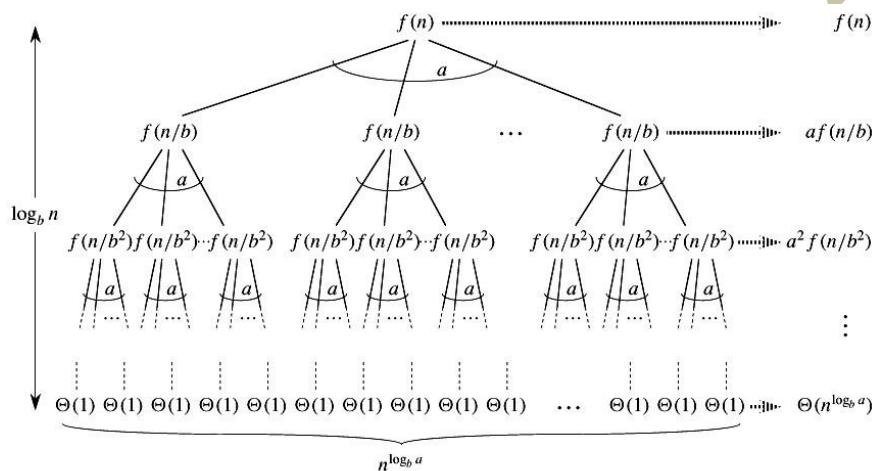
1. If  $f(n) = \Theta(n^c)$  where  $c < \log_b a$  then  $T(n) = \Theta(n^{\log_b a})$

2. If  $f(n) = \Theta(n^c)$  where  $c = \log_b a$  then  $T(n) = \Theta(n^c \log n)$

3. If  $f(n) = \Theta(n^c)$  where  $c > \log_b a$  then  $T(n) = \Theta(f(n))$

#### How does this work?

Master method is mainly derived from recurrence tree method. If we draw recurrence tree of  $T(n) = aT(n/b) + f(n)$ , we can see that the work done at root is  $f(n)$  and work done at all leaves is  $\Theta(n^c)$  where  $c$  is  $\log_b a$ . And the height of recurrence tree is  $\log_b n$ .



In recurrence tree method, we calculate total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2). If work done at root is asymptotically more, then our result becomes work done at root (Case3).

#### Examples of some standard algorithms whose time complexity can be evaluated using Master Method

Merge Sort:  $T(n) = 2T(n/2) + \Theta(n)$ . It falls in case 2 as  $c$  is 1 and  $\log_b a$  is also 1. So the solution is  $\Theta(n \log n)$

Binary Search:  $T(n) = T(n/2) + \Theta(1)$ . It also falls in case 2 as  $c$  is 0 and  $\log_b a$  is also 0. So the solution is  $\Theta(\log n)$

**Notes:**

**1)** It is not necessary that a recurrence of the form  $T(n) = aT(n/b) + f(n)$  can be solved using Master Theorem. The given three cases have some gaps between them. For example, the recurrence  $T(n) = 2T(n/2) + n/\text{Log}n$  cannot be solved using master method.

**2)** Case 2 can be extended for  $f(n) = \Theta(n^c \text{Log}^{kn})$

If  $f(n) = \Theta(n^c \text{Log}^{kn})$  for some constant  $k \geq 0$  and  $c = \log_b a$ , then  $T(n) = \Theta(n^c \text{Log}^{k+1}n)$

Practice Problems and Solutions on Master Theorem. (Refer Notes)

**References:**

[http://en.wikipedia.org/wiki/Master\\_theorem](http://en.wikipedia.org/wiki/Master_theorem)

MIT Video Lecture on Asymptotic Notation | Recurrences | Substitution, Master Method  
Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Prof. Selva Mary G

## Mathematical Analysis of Nonrecursive Algorithms

The core of the algorithm analysis: to find out how the number of the basic operations depends on the size of the input.

**There are four rules to count the operations:**

### **Rule 1: for loops - the size of the loop times the running time of the body**

The running time of a for loop is at most the running time of the statements inside the loop times the number of iterations.

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

#### **a. Find the running time of statements when executed only once:**

The statements in the loop heading have fixed number of operations, hence they have constant running time  $O(1)$  when executed only once. The statement in the loop body has fixed number of operations, hence it has a constant running time when executed only once.

#### **b. Find how many times each statement is executed.**

```
for( i = 0; i < n; i++)      // i = 0; executed only once: O(1)
                            // i < n; n + 1 times O(n)
                            // i++ n times O(n)
                            // total time of the loop heading:
                            // O(1) + O(n) + O(n) = O(n)
    sum = sum + i;          // executed n times, O(n)
```

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

Loop running time is:  $O(n)$

Mathematical analysis of how many times the statements in the body are executed

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

If

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

then the time is  $O(n)$

### **Rule 2: Nested loops – the product of the size of the loops times the running time of the body**

The total running time is the running time of the inside statements times the product of the sizes of all the loops

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

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

Mathematical analysis:

Inner loop:

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

Outer loop:

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

Inner loop:

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



Outer loop:

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

What happens if the inner loop does not start from 0?

sum = 0;

for( i = 0; i < n; i++)

    for( j = i; j < n; j++)

        sum++;

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

i = 0, inner loop runs n times

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

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

...

i = n - 2, inner loop runs 2 times

i = n - 1, inner loop runs once.

Thus we get: (1 + 2 + ... + n) = n\*(n+1)/2 = O(n<sup>2</sup>)

### **General rule for nested loops:**

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

Example:

sum = 0;

for( i = 0; i < n; i++)

    for( j = 0; j < 2n; j++)

        sum++;

We have one operation inside the loops, and the product of the sizes is 2n<sup>2</sup>

Hence the running time is O(2n<sup>2</sup>) = O(n<sup>2</sup>)

Note: if the body contains a function call, its running time has to be taken into consideration.

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

Assume that the running time of `function(sum)` is known to be  $\log(n)$ .

Then the total running time will be  $O(n^2 \log(n))$

### Rule 3: Consecutive program fragments

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

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

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

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

### Rule 4: If statement

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

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

Summary

Steps in analysis of nonrecursive algorithms:

- Decide on parameter  $n$  indicating input size
- Identify algorithm's basic operation
- Check whether the number of time the basic operation is executed depends on some additional property of the input. If so, determine worst, average, and best case for input of size  $n$
- Count the number of operations using the rules above.

### Exercise

a.

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

b.

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

c.

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

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

d.

```
sum = 0;
for( i = 0; i < n; i++)
sum++;
val = 1;
for( j = 0; j < n*n; j++)
val = val * j;
```

e.

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

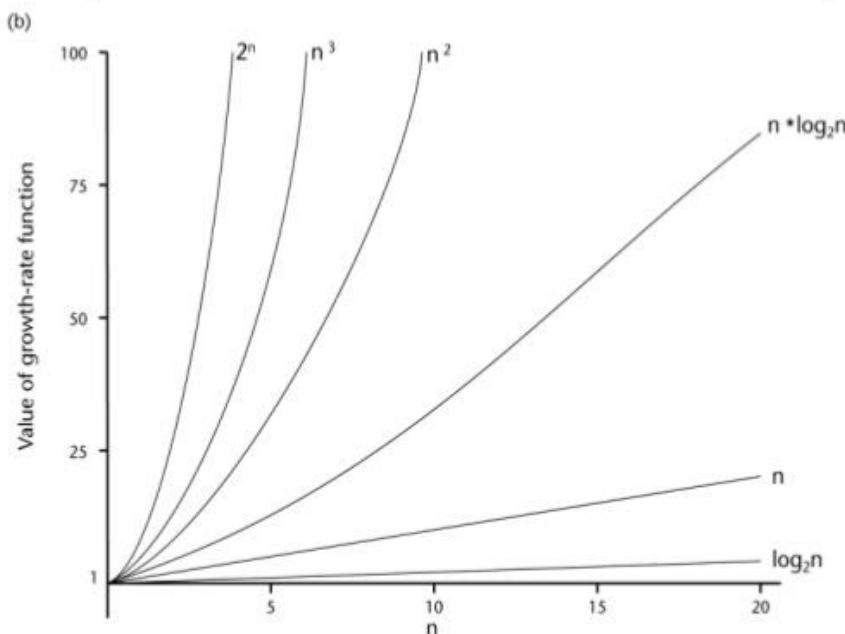
The complexity of the function `compute_val(x,y)` is given to be  $O(n \log n)$

**Solutions:**

- (a)  $O(n^3)$
- (b)  $O(n^2)$
- (c)  $O(n^5)$
- (d)  $O(n^2)$
- (e)  $O(n^3 \log(n))$

**Order of Growth Functions**

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


**A Comparison of Growth-Rate Functions (cont.)**


## Linear and Binary search

An algorithm is a step-by-step procedure or method for solving a problem by a computer in a given number of steps. The steps of an algorithm may include repetition depending upon the problem for which the algorithm is being developed. The algorithm is written in human readable and understandable form. To search an element in a given array, it can be done in two ways linear search and Binary search.

### Linear Search

A linear search is the basic and simple search algorithm. A linear search searches an element or value from an array till the desired element or value is not found and it searches in a sequence order. It compares the element with all the other elements given in the list and if the element is matched it returns the value index else it return -1. Linear Search is applied on the unsorted or unordered list when there are fewer elements in a list.

#### Pseudocode:-

```
# Input: Array D, integer key  
# Output: first index of key in D, or -1 if not found  
For i = 0 to last index of D:  
    if D[i] == key:  
        return i  
    return -1
```

#### Example with Implementation

To search the element 5 it will go step by step in a sequence order.

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

## Asymptotic Analysis

### Worst Case Analysis (Usually Done)

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. For Linear Search, the worst case happens when the element to be searched (target in the above code) is not present in the array. When target is not present, the search() functions compares it with all the elements of array one by one. Therefore, the worst case time complexity of linear search would be  $\Theta(n)$ .

### Average Case Analysis (Sometimes done)

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. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of target not being present in array).

The key is equally likely to be in any position in the array

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

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

...

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

...

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

### Best Case Analysis (Bogus)

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. In the linear search problem, the best case occurs when Target is present at the first location. The number of operations in the best case is constant (not dependent on  $n$ ). So time complexity in the best case would be  $\Theta(1)$

## Binary Search

Binary Search is applied on the sorted array or list. In binary search, we first compare the value with the elements in the middle position of the array. If the value is matched, then we return the value. If the value is less than the middle element, then it must lie in the lower half of the array and if it's greater than the element then it must lie in the upper half of the array. We repeat this procedure on the lower (or upper) half of the array. Binary Search is useful when there are large numbers of elements in an array.

To search an element 13 from the sorted array or list.

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

In the above program logic, we are first comparing the middle number of the list, with the target, if it matches we return. If it doesn't, we see whether the middle number is greater than or smaller than the target.

If the Middle number is greater than the Target, we start the binary search again, but this time on the left half of the list, that is from the start of the list to the middle, not beyond that.

If the Middle number is smaller than the Target, we start the binary search again, but on the right half of the list, that is from the middle of the list to the end of the list.

## Complexity Analysis

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

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

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

$$\begin{aligned} &= 2 + 2 + T\left(\frac{n}{2^2}\right) \quad // 2^{\text{nd}} \text{ iteration} \\ &= 2 + 2 + 2 + T(n/2^3) \quad // 3^{\text{rd}} \text{ iteration} \end{aligned}$$

...

$$\begin{aligned} &= i * 2 + T(n/2^i) \quad // i^{\text{th}} \text{ iteration} \\ ... &= k * 2 + T(1) \end{aligned}$$

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

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

So we expect binary search to be significantly more efficient than linear search for large values of  $n$ .

## Bubble Sort

Bubble Sort is an algorithm which is used to sort  $N$  elements that are given in a memory for eg: an Array with  $N$  number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.

## Bubble Sort for Data Structures

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 5 | 1 | 6 | 2 | 4 | 3 |
|---|---|---|---|---|---|

Lets take this Array.

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 5 | 1 | 6 | 2 | 4 | 3 |
| 1 | 5 | 6 | 2 | 4 | 3 |
| 1 | 5 | 2 | 6 | 4 | 3 |
| 1 | 5 | 2 | 4 | 6 | 3 |
| 1 | 5 | 2 | 4 | 3 | 6 |

Here we can see the Array after the first iteration.

Similarly, after other consecutive iterations, this array will get sorted.

## Sorting using Bubble Sort Algorithm

Let's consider an array with values {5, 1, 6, 2, 4, 3}

```
int a[6] = {5, 1, 6, 2, 4, 3};
```

```
int i, j, temp;
```

```
for(i=0; i<6; i++)
```

```
{
```

```
    for(j=0; j<6-i-1; j++)
```

```
{
```

```
        if( a[j] > a[j+1])
```

```
{
```

```
            temp = a[j];
```

```
a[j] = a[j+1];
```

```
a[j+1] = temp;
```

```
}
```

```
}
```

```
}
```

```
//now you can print the sorted array after this
```

Above is the algorithm, to sort an array using Bubble Sort. Although the above logic will sort an unsorted array, still the above algorithm isn't efficient and can be enhanced

further. Because as per the above logic, the for loop will keep going for six iterations even if the array gets sorted after the second iteration.

Hence we can insert a flag and can keep checking whether swapping of elements is taking place or not. If no swapping is taking place that means the array is sorted and we can jump out of the for loop.

```
int a[6] = {5, 1, 6, 2, 4, 3};  
int i, j, temp;  
for(i=0; i<6; i++)  
{  
    int flag = 0;      //taking a flag variable  
    for(j=0; j<6-i-1; j++)  
    {  
        if( a[j] > a[j+1])  
        {  
            temp = a[j];  
            a[j] = a[j+1];  
            a[j+1] = temp;  
            flag = 1;      //setting flag as 1, if swapping occurs  
        }  
    }  
    if(!flag)          //breaking out of for loop if no swapping takes place  
    {  
        break;  
    }  
}
```

In the above code, if in a complete single cycle of j iteration(inner for loop), no swapping takes place, and flag remains 0, then we will break out of the for loops, because the array has already been sorted.

## Complexity Analysis of Bubble Sorting

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

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

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

$$\text{i.e } O(n^2)$$

Hence the complexity of Bubble Sort is  **$O(n^2)$** .

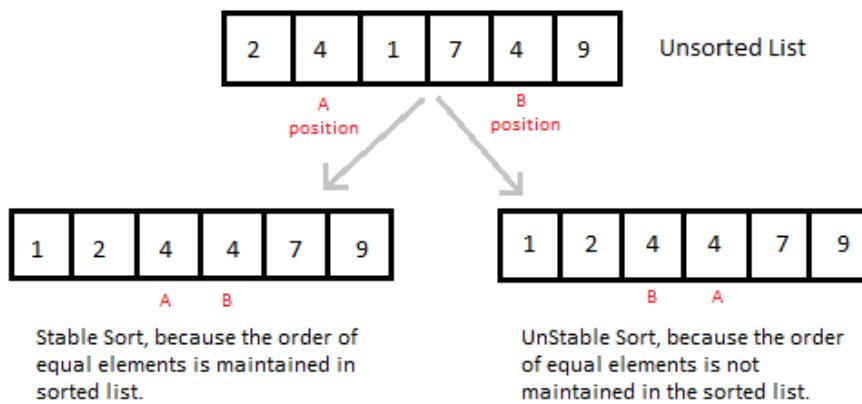
The main advantage of Bubble Sort is the simplicity of the algorithm. Space complexity for Bubble Sort is  **$O(1)$** , because only single additional memory space is required for **temp** variable

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

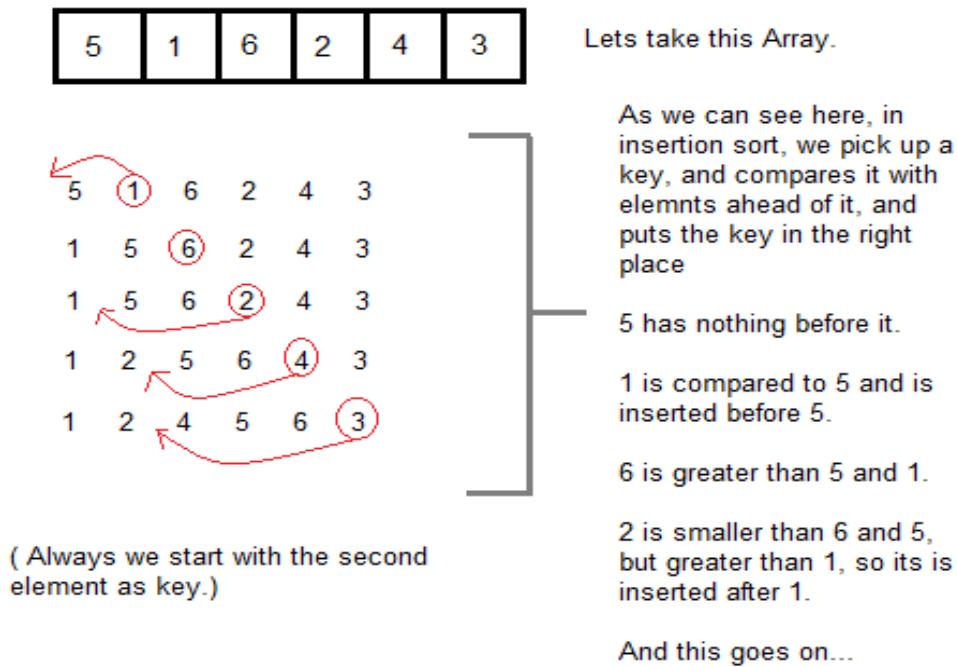
## Insertion Sorting

It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.

1. It has one of the simplest implementation
2. It is efficient for smaller data sets, but very inefficient for larger lists.
3. Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
4. It is better than Selection Sort and Bubble Sort algorithms.
5. Its space complexity is less, like Bubble Sorting, insertion sort also requires a single additional memory space.
6. It is Stable, as it does not change the relative order of elements with equal keys



## How Insertion Sorting Works



## Sorting using Insertion Sort Algorithm

```

int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, key;
for(i=1; i<6; i++)
{
    key = a[i];
    j = i-1;
    while(j>=0 && key < a[j])
    {
        a[j+1] = a[j];
        j--;
    }
    a[j+1] = key;
}

```

}

Now lets, understand the above simple insertion sort algorithm. We took an array with 6 integers. We took a variable key, in which we put each element of the array, in each pass, starting from the second element, that is  $a[1]$ .

Then using the while loop, we iterate, until  $j$  becomes equal to zero or we find an element which is greater than key, and then we insert the key at that position.

In the above array, first we pick 1 as key, we compare it with 5(element before 1), 1 is smaller than 5, we shift 1 before 5. Then we pick 6, and compare it with 5 and 1, no shifting this time. Then 2 becomes the key and is compared with, 6 and 5, and then 2 is placed after 1. And this goes on, until complete array gets sorted.

### **Complexity Analysis of Insertion Sorting**

- Worst Case Time Complexity :  $O(n^2)$
- Best Case Time Complexity :  $O(n)$
- Average Time Complexity :  $O(n^2)$
- Space Complexity :  $O(1)$

## UNIT II ARRAYS & LINKED LIST

Arrays - Operations on Arrays – Insertion and Deletion- Applications on Arrays- Multidimensional Arrays- Sparse Matrix- Linked List Implementation – Insertion-Linked List- Deletion and Search- Applications of Linked List-Polynomial Arithmetic-Cursor Based Implementation – Methodology-Cursor Based Implementation-Circular Linked List-Circular Linked List – Implementation-Applications of Circular List -Joseph Problem-Doubly Linked List-Doubly Linked List Insertion-Doubly Linked List Insertion variations-Doubly Linked List Deletion -Doubly Linked List Deletion

### **LINEAR (OR ORDERED) LISTS ADT**

List ADT instances are of the form  $[l_0, l_1, l_2, \dots, l_{n-1}]$ , where  $l_i$  denotes a list of elements with  $n \geq 0$  is finite list where size is  $n$ .

#### **LINEAR LIST OPERATIONS**

Some of the linear list operations are as follows:

1. **printList**

This function is used to print the element in a list.

2. **makeEmpty**

This is used to create an empty list.

3. **Find**

This function is used to find an element in a given list.

Example:

list: [34, 12, 52, 16, 12]

find(52)  $\rightarrow$  3

4. **Insert**

This is used to insert a new element into the list.

For example, insert(x,3) will insert an element 'x' at the 3<sup>rd</sup> index position as follows,

[34, 12, 52, x, 16, 12]

5. **Remove**

This function is used to remove an element from the given list.

For example, remove(52) will remove the element 52 from the list as follows, [34, 12, x, 16, 12].

6. **findKth**

This is used to find/search an element at the kth index in a given list.

► remove(52)  $\rightarrow$  34, 12, x, 16, 12

► : retrieve the element at a certain position

### **IMPLEMENTATION OF LINEAR LISTS**

A linear list ADT is implemented using two standard techniques namely array-based techniques and linked list-based techniques.

#### **ARRAY IMPLEMENTATION OF LIST**

Array is a collection of specific number of data stored in a consecutive memory location.

- Insertion and Deletion operation are expensive as it requires more data movement
- Find and Printlist operations takes constant time.
- Even if the array is dynamically allocated, an estimate of the maximum size of the list is required which considerably wastes the memory space.

## **LINKED LIST IMPLEMENTATION**

Linked list consists of series of nodes. Each node contains the element and a pointer to its successor node. The pointer of the last node points to NULL.

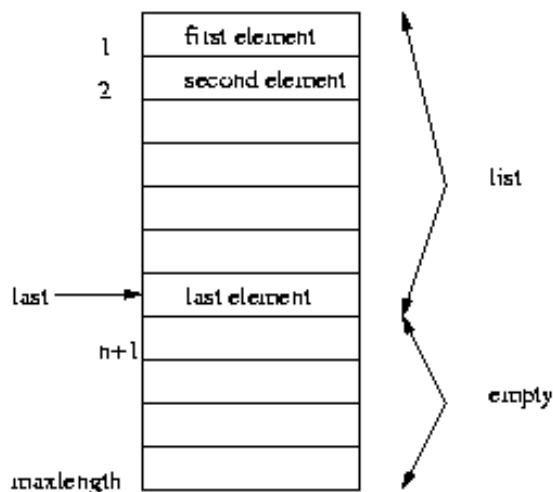
Insertion and deletion operations are easily performed using linked list.

## **TYPES OF LINKED LIST**

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List.

## **ARRAY IMPLEMENTATION**

In an array implementation, elements are stored in contiguous array positions as shown in the below figure.

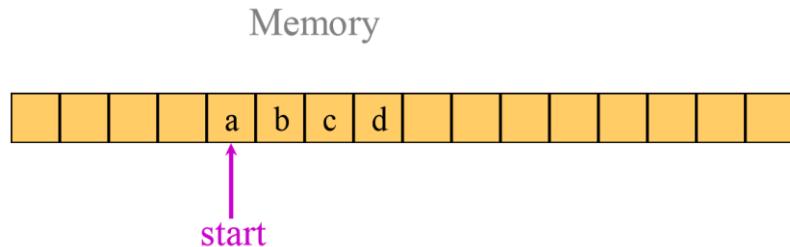


Arrays could be represented in the following forms:

1. 1-D representation
2. 2-D representation
3. Multi-dimensional representation

## **1D ARRAY REPRESENTATION**

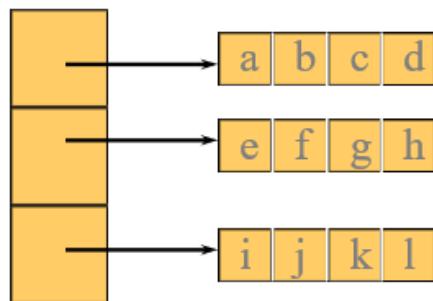
In 1D array representation, the elements will be mapped into a single dimensional contiguous memory location. Let  $x$  be an array with elements  $[a, b, c, d]$ . The below figure shows how the array elements will be stored in memory.



A 2D array ‘ $a$ ’ of size  $3 \times 4$  will be declared as follows: `int a[3][4]`. The tabular representation of the 2D array ‘ $a$ ’ is shown as follows:

|           |           |           |           |
|-----------|-----------|-----------|-----------|
| $a[0][0]$ | $a[0][1]$ | $a[0][2]$ | $a[0][3]$ |
| $a[1][0]$ | $a[1][1]$ | $a[1][2]$ | $a[1][3]$ |
| $a[2][0]$ | $a[2][1]$ | $a[2][2]$ | $a[2][3]$ |

The method of storing elements in a 2D array is called as array-of-arrays representation. This will require 4 1D arrays of size 3,4,4,4 to store the array ‘ $a$ ’. The below figure shows how elements of ‘ $a$ ’ of size  $3 \times 4$  will be stored in a memory.



## APPLICATIONS ON ARRAYS

An Arrays in data Structure is a container which can hold a fixed number of items and these items should be of the same type.

- Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables. Many databases, small and large, consist of one-dimensional arrays whose elements are records.
- Arrays are used to implement other data structures, such as lists, heaps, hash tables, deques, queues and stacks.
- One or more large arrays are sometimes used to emulate in-program dynamic memory allocation, particularly memory pool allocation. Historically, this has sometimes been the only way to allocate "dynamic memory" portably.
- Arrays can be used to determine partial or complete control flow in programs, as a compact alternative to (otherwise repetitive) multiple “if” statements. They are known

in this context as control tables and are used in conjunction with a purpose-built interpreter whose control flow is altered according to values contained in the array. The array may contain subroutine pointers (or relative subroutine numbers that can be acted upon by SWITCH statements) that direct the path of the execution.

## IMPLEMENTATION OF ARRAYS – INSERTION AND DELETION

### //To Insert An Element In An Array

```
#include<stdio.h>

int insertElement(int arr[], int elements, int keyToBeInserted, int size)

{
    if (elements >= size)
        return elements;
    arr[elements] = keyToBeInserted;
    return (elements + 1);
}

int main()

{
    int array[20] = { 23,35,45,76,34,92 };
    int size = sizeof(array) / sizeof(array[0]);
    int elements = 6;
    int i, keyToBeInserted = 65;
    printf("\n Before Insertion: ");
    for (i = 0; i < elements; i++)
    {
        printf("%d ", array[i]);
    }
    elements = insertElement(array, elements, keyToBeInserted, size);
    printf("\n After Insertion: ");
    for (i = 0; i < elements; i++)
    {
        printf("%d ",array[i]);
    }
}
```

```
    return 0;  
}
```

### **Output:**

Before Insertion: 23 35 45 76 34 92

After Insertion: 23 35 45 76 34 92 65

### **Explanation:**

Main function call the insert function to insert an element in an array by passing the parameters arr[] = array, elements = number of elements present in the array and keyToBeInserted = element to be inserted in the array with the size of the array. It checks if the maximum space of the array is already full and return the elements. If not then the element is inserted at the last index and the new array size is returned.

### **//To Delete An Element In An Array**

```
#include<stdio.h>  
  
int deleteElement(int array[], int size, int keyToDelete)  
{  
    int pos = findElement(array, size, keyToDelete);  
    int i;  
    if (pos == - 1)  
    {  
        printf("Element not found");  
        return size;  
    }  
    for (i = pos; i < size - 1; i++)  
    {  
        array[i] = array[i + 1];  
    }  
    return size - 1;  
}  
  
int findElement(int array[], int size, int keyToDelete)  
{  
    int i;  
    for (i = 0; i < size; i++)
```

```

if (array[i] == keyToDelete)
    return i;
else
    return -1;
}

int main()
{
    int array[] = { 31, 27, 3, 54, 67, 32 };
    int size = sizeof(array) / sizeof(array[0]);
    int i, keyToDelete = 67;
    printf("n Before Deletion: ");
    for (i = 0; i < size; i++)
        printf("%d ", array[i]);
    deleteElement(array, size, keyToDelete);
    printf("n After Deletion: ");
    for (i = 0; i < size; i++)
        printf("%d ", array[i]);
    return 0;
}

```

### **Output:**

Before Deletion: 31 27 3 54 67 32

After Insertion: 31 27 3 54 32

### **Explanation:**

Main function call the delete function to delete an element by passing the parameters array[] is the array from which element needs to be deleted ,size of the array and keyToDelete is the element to be deleted from the array. If element is not found then it prints Element not found. Else it deletes the element & moves rest of the element by one position.

### **//To Search an Element in an Array**

```

#include<stdio.h>

int findElement(int array[], int size, int keyToBeSearched)
{

```

```

int i;
for (i = 0; i < size; i++)
    if (array[i] == keyToBeSearched)
        return i;
    else
        return - 1;
}
int main()
{
    int array[] = { 41, 29, 32, 54, 19, 3};
    int size = sizeof(array) / sizeof(array[0]);
    int keyToBeSearched = 67;
    int pos = findElement(array, size, keyToBeSearched);
    if(pos== -1)
    {
        printf("n Element %d not found", keyToBeSearched);
    }
    else
    {
        printf("n Position of %d: %d", keyToBeSearched ,pos+1);
    }
    return 0;
}

```

### **Output:**

Position of 19:5

### **EXPLANATION**

Main function call sub function to search the element in the given array by passing the parameters the array from which element needs to be deleted, size of the array and keyToFind is the element to be search from the array. Running a for loop for Finding & returning the position of the element.

## MULTI-DIMENSIONAL SPARSE MATRIX

A matrix is a two-dimensional data object made of m rows and n columns. Generally, a matrix is represented to have m X n values. When the value of m is equal to n, then it is called as a **square matrix**.

There may be a situation in which a matrix may contain a greater number of ‘0’ values (elements) than NON-ZERO values. Such matrix is known as **sparse matrix**. But when a sparse matrix is represented with 2-dimensional array with all ZERO and non-ZERO values, lot of space is wasted to represent that matrix.

For example, consider a matrix of size 200 X 200 containing only 10 non-zero elements. In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of the matrix are filled with zero values. That means, totally we allocate  $200 \times 200 \times 2 = 80000$  bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 80000 times. This results in an increased time and space complexity. An example sparse matrix is given below:

$$\begin{vmatrix} & 1 & 0 & 0 \\ & 0 & 0 & 0 \\ & 0 & 4 & 0 \end{vmatrix}$$

## SPARSE MATRIX REPRESENTATIONS

In order to make it simple, a sparse matrix can be represented using two representations.

1. Triplet Representation (Array Representation)
2. Linked Representation

### TRIPLET REPRESENTATION (ARRAY REPRESENTATION)

In this representation, only non-zero values are considered along with their row and column index values. The array index [0,0] stores the total number of rows, [0,1] index stores the total number of columns and [0,1] index has the total number of non-zero values in the sparse matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image.

| Rows | Columns | Values |
|------|---------|--------|
| 5    | 6       | 6      |
| 0    | 4       | 9      |
| 1    | 1       | 8      |
| 2    | 0       | 4      |
| 2    | 3       | 2      |
| 3    | 5       | 5      |
| 4    | 2       | 2      |

In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image. Here the first row in the right-side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. The second row is filled with 0, 4, & 9 which

indicates the non-zero value 9 is at the 0th-row 4th column in the Sparse matrix. In the same way, the remaining non-zero values also follow a similar pattern.

## LINKED LIST REPRESENTATION

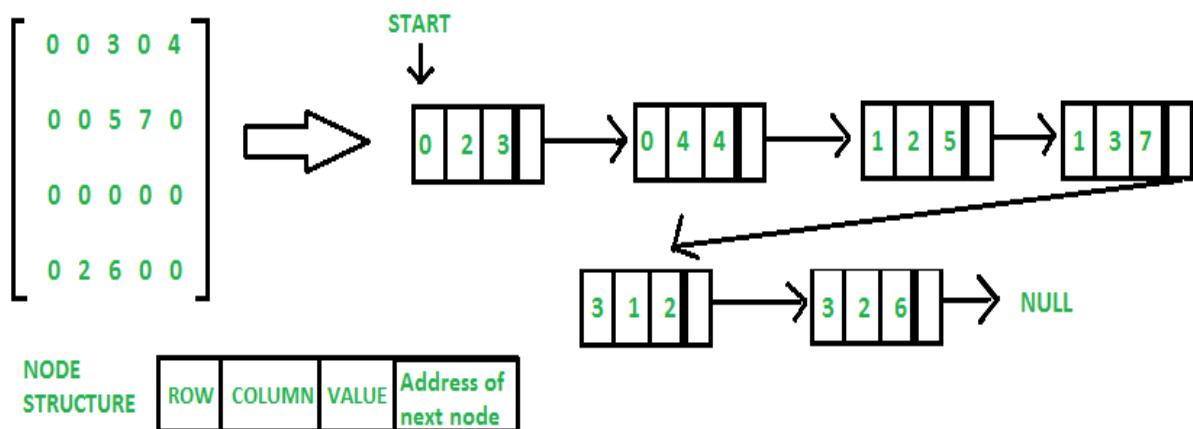
A sparse matrix could be represented using a linked list data structure also. In linked list, each node has four fields. These four fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non-zero element located at index – (row, column)
- **Next node:** Address of the next node

For example, consider a 4 X 5 matrix,

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$

The above sparse matrix could be represented as follows:



The first field of the node holds the row index, second field will have the column index, third field will have the value of the element and the address of the next node will be stored in the next field.

A node in a sparse matrix could be declared as follows:

```

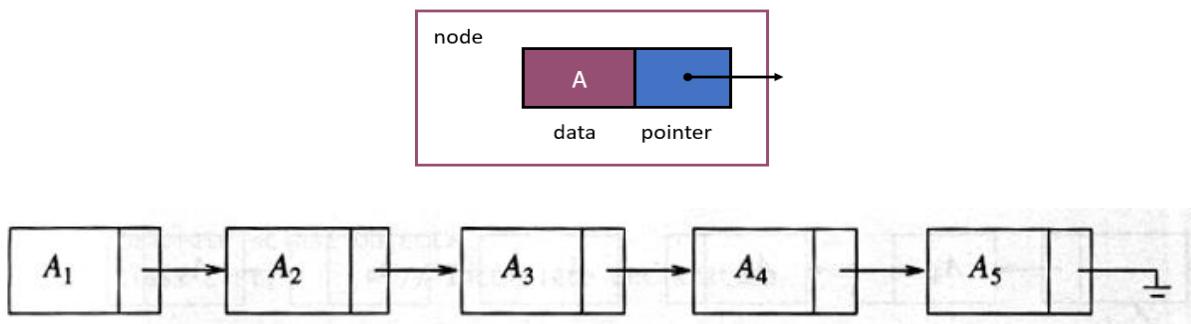
struct Node
{
    int value;
    int row_position;
    int column_postion;
    struct Node *next;
};

```

## LINKED LIST

Linked list consists of series of connected nodes. Each node contains the element (a piece of data) and a pointer to its successor node. The pointer of the last node points to NULL. A linked list can grow or shrink in size as the program runs. A linked list representation requires no estimate of the maximum size of the list. Space wastage is reduced in linked list representation.

The below figure shows a node structure with data and a pointer to the next node and a list of elements ( $A_1, A_2, A_3, A_4, A_5$ ) stored using linked list representation. Insertion and deletion operations are easily performed using linked list, when compared to using arrays.

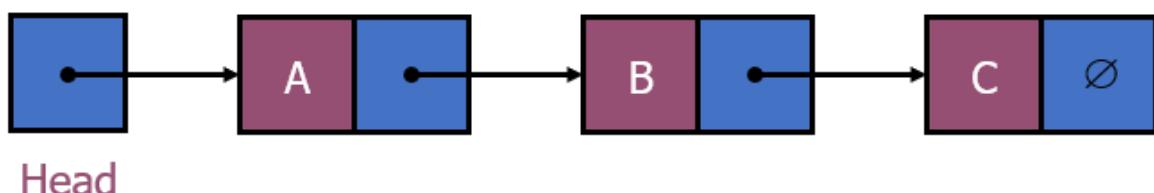


There are different types of linked list available.

## TYPES OF LINKED LIST

### 1. Singly Linked List

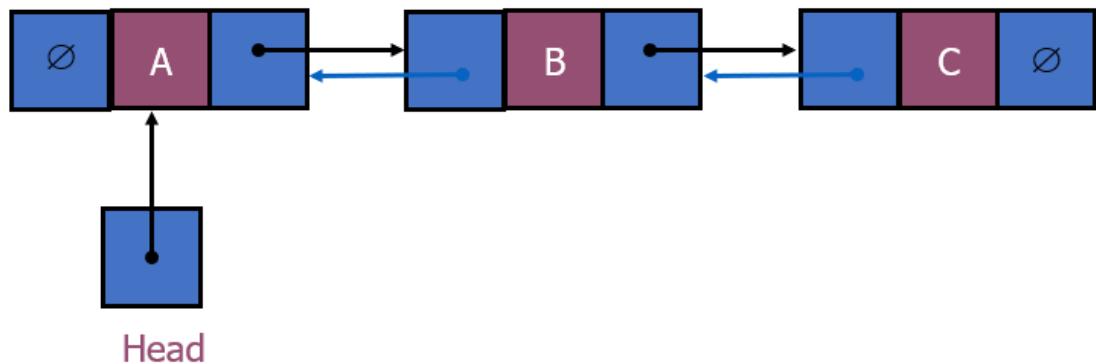
Each node points in a singly linked list has a successor. There will be NULL value in the next pointer field of the last node in the list



### 2. Doubly Linked List

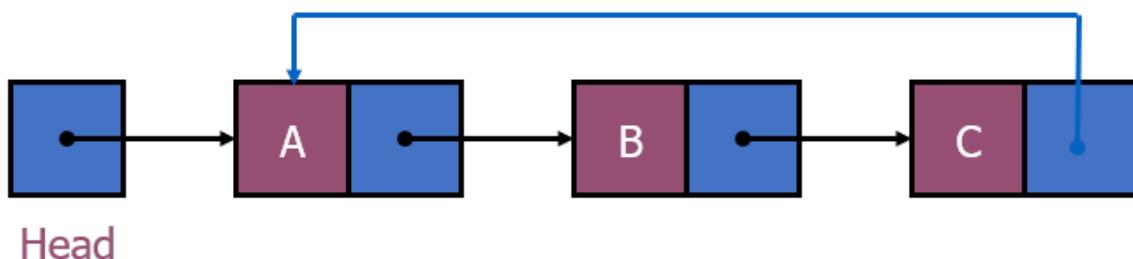
In a doubly linked list, each node points to not only successor but the predecessor also. There are two NULL: at the first and last nodes in the list. Advantage of having a doubly linked

list is that in given a list, it is easy to visit its predecessor. It will be convenient to traverse lists backwards



### 3. Circular Linked List.

In a circular linked list, The last node points to the first node of the list.



## SINGLY LINKED LIST IMPLEMENTATION

A node in a linked list is usually of structure data type. The declaration of the node structure using pointers is as follows:

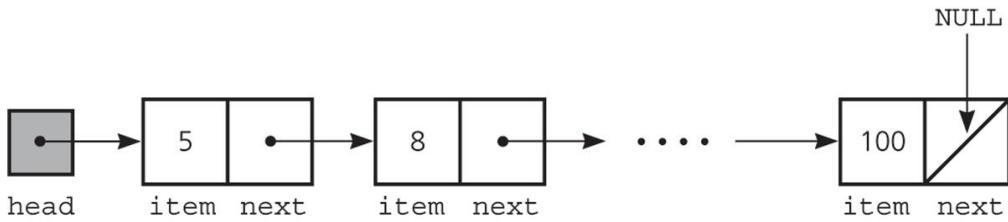
```
struct Node
{
    int data;
    struct Node *next;
}*head = NULL;
```

A node is dynamically allocated as follows:

```
node *p;
p = malloc(sizeof(Node));
```

By using malloc function, memory will be allocated for the node with size of type struct. The head pointer points to the first node in a linked list. If head is NULL, the linked list is empty and generally represented as **head=NULL**.

A sample linked list is shown below:



Linked lists provide flexibility in allowing the items to be rearranged efficiently.

– Insert an element.

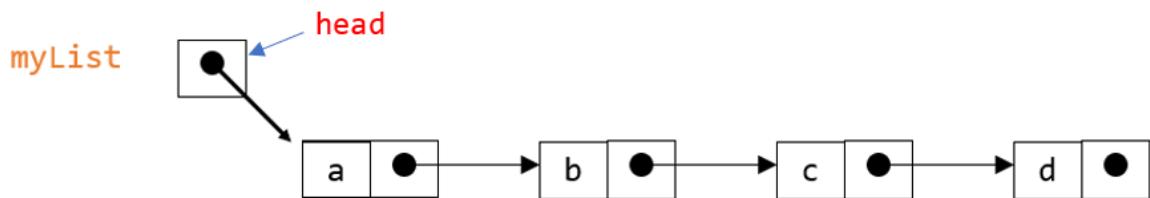
– Delete an element.

## INSERTING A NEW NODE

In a linked list, inserting a new node has the following possible cases.

1. Insert into an empty list
2. Insert in front
3. Insert at back
4. Insert in middle

A sample linked list is shown below:

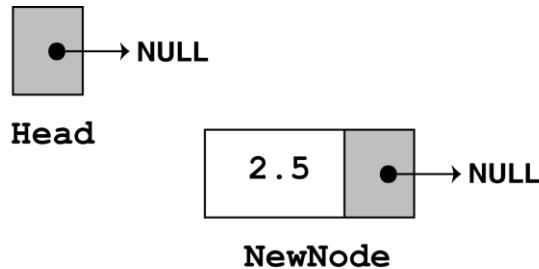


- Each node contains a value and a link to its successor (the last node has no successor)
- The header points to the first node in the list (or contains the null link if the list is empty)
- The entry point into a linked list is called the **head** of the list.
- It should be noted that head is not a separate node, but the reference to the first node.
- If the list is empty then the head is a null reference.

## INSERTING AT BEGINNING OF THE LIST

// Creating a new node

```
newNode = new ListNode;  
newNode->value = num;
```

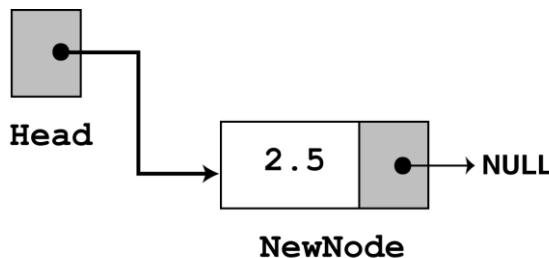


// If it's the first node set it to point header

```
head=newNode;
head->next = NULL;
```

//If already elements are there in the list

```
newNode->next = head;
head=newNode;
```

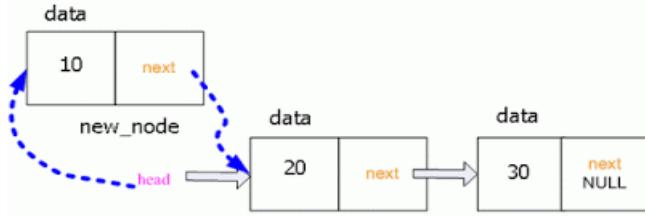


```
void insertAtBeginning(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    if(head == NULL)
    {
        newNode->next = NULL;
        head = newNode;
    }
    else
    {
        newNode->next = head;
        head = newNode;
    }
    printf("\nOne node inserted!!!\n");
}
```

### Routine to Insert a new node at the beginning

Initially, allocate memory for node using malloc function and insert the value in the data field. Now if the list is empty, make the head point to the new node created. Else, copy the address of the existing linked list from the head and store it in the new node's next pointer. Then store the address of the new node in the head. An example figure is shown below.

### Example:



Initially, the linked list has elements 20 and 30. Node with data 20 is the first node, whose address is stored in the head. Now to insert a new node with data 10, copy the address of node with data 20 from head and store in new nodes next pointer. Then make the head point to the new node.

### INSERTING AT END OF THE LIST

Steps involved in inserting a new node at the end in the list are as follows:

**Step 1:** Create a **newNode** with given value and **newNode → next** as **NULL**.

**Step 2:** Check whether list is **Empty** (**head == NULL**).

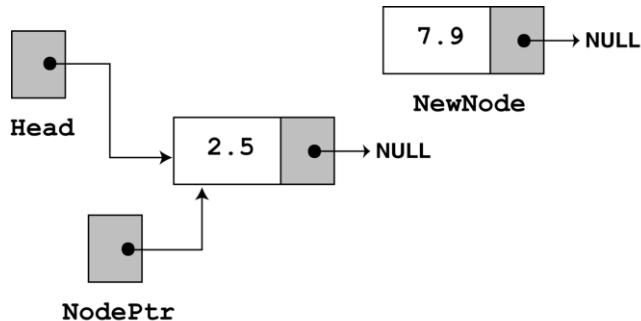
**Step 3:** If it is **Empty** then, set **head = newNode**.

**Step 4:** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

**Step 5:** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).

**Step 6:** Set **temp → next = newNode**.

1. Use a Node Pointer to trace to the current position



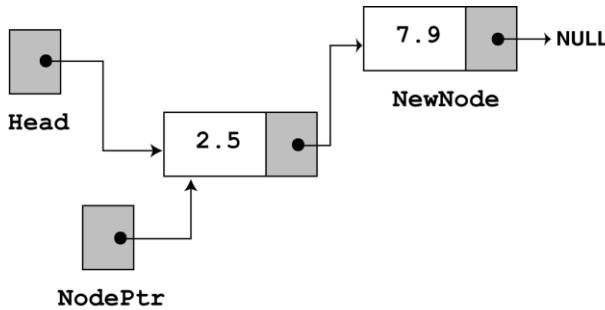
2. If the Node pointer does not points to NULL then move the Node pointer to the next node until it points to NULL

**nodePtr=nodePtr->next;**

3. When it reaches NULL append the newNode at the last

**nodePtr ->next=newNode;**

**newNode->next=NULL;**



```

void insertAtEnd(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if(head == NULL)
        head = newNode;
    else
    {
        struct Node *temp = head;
        while(temp->next != NULL)
            temp = temp->next;
        temp->next = newNode;
    }
    printf("\nOne node inserted!!!\n");
}

```

### Routine To Insert An Element At The End

#### INSERTING A NEW NODE IN THE MIDDLE

Steps involved in inserting a new node in the middle of the list are as follows:

**Step 1:** Create a **newNode** with given value.

**Step 2:** Check whether list is **Empty (head == NULL)**

**Step 3:** If it is Empty then, set **newNode → next = NULL** and **head = newNode**.

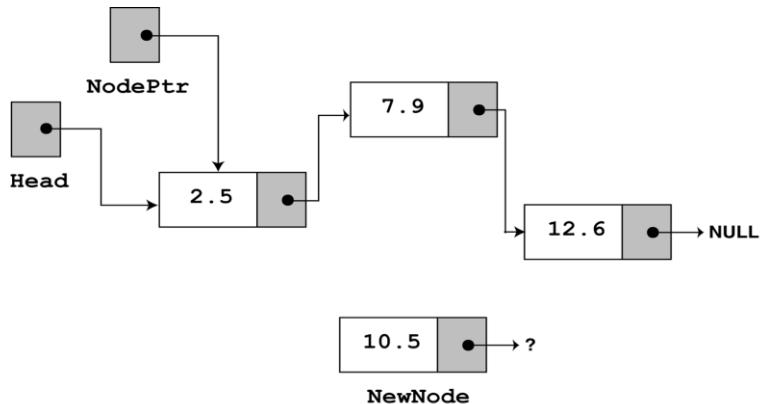
**Step 4:** If it is **Not Empty** then, **define a node pointer temp** and initialize with **head**.

**Step 5:** Keep **moving the temp to its next node** until it reaches to the node after which we want to insert the **newNode** (until **temp → data is equal to location**, here location is the node value after which we want to insert the **newNode**).

**Step 6:** Every time check whether **temp is reached to last node or not**. If it is reached to last node then display '**Given node is not found in the list!!! Insertion not possible!!!**' and terminate the function. Otherwise move the **temp to next node**.

**Step 7:** Finally, Set '**newNode → next = temp → next**' and '**temp → next = newNode**'

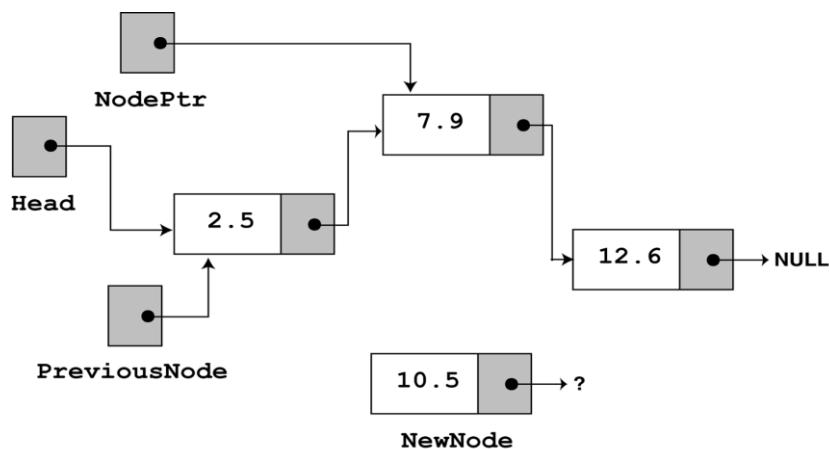
1. Use a Node Pointer to trace to the current position and to store the value of the next address.



2. Use a Previous Node Pointer to trace to the current position

```
previousNode=nodePtr;
```

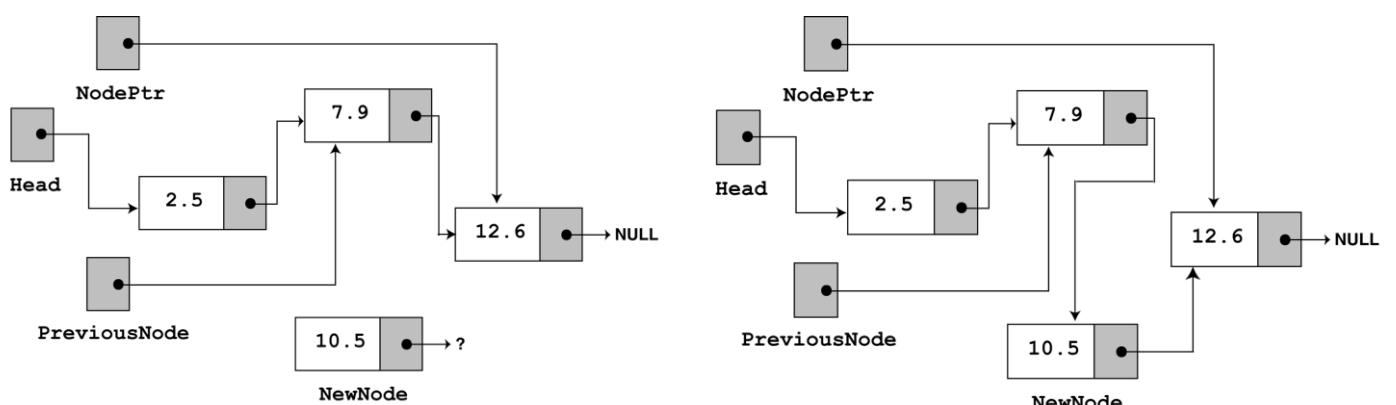
```
nodePtr=nodePtr->next;
```



3. The New Node Pointer find the next node to insert the node

```
previousNode->next=newNode;
```

```
newnode->next=nodePtr;
```



```

void insertBetween(int value, int loc)
{
    struct Node *newNode, *prev_ptr, *cur_ptr;
    newNode = (struct Node*)malloc(sizeof(struct
Node));
    newNode->data = value;
    cur_ptr = head;
    if(head == NULL)
    {
        newNode->next = NULL;
        head = newNode;
    }
    else
    {
        for(i=1;i<loc;i++)
        {
            prev_ptr = cur_ptr;
            cur_ptr = cur_ptr->next;
        }
        prev_ptr->next = newNode;
        newNode->next = cur_ptr;
    }
    printf("\nOne node inserted!!!\n");
}

```

---

### Routine to Insert a New Node in Between

## DELETING A NODE

In a linked list, deleting a node has the following possible cases.

1. Delete at the front
2. Delete at the back
3. Delete in the middle

### DELETING A NODE AT THE FRONT (Deleting from Beginning of the list)

Steps involved in deleting a node from the beginning of the list are as follows:

**Step 1:** Check whether list is **Empty** (**head == NULL**)

**Step 2:** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.

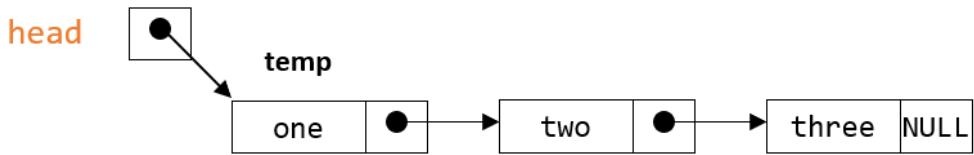
**Step 3:** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with **head**.

**Step 4:** Check whether list is having only one node (**temp → next == NULL**)

**Step 5:** If it is **TRUE** then set **head = NULL** and delete **temp** (Setting **Empty** list conditions)

**Step 6:** If it is **FALSE** then set **head = temp → next**, and delete **temp**.

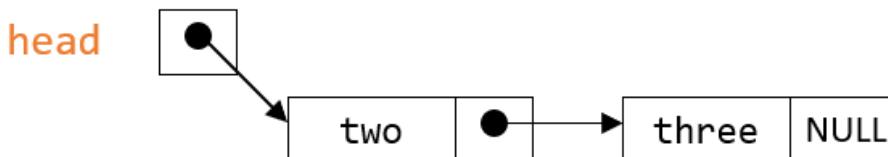
- To delete the first element, change the link in the header



- Assign head to a temp. **temp=head;**
- If the temp is not NULL, check whether the data to be deleted is head.
- If true then delete the head node and make the next node as head.

**head=temp->next;**

**free(temp);**



```

void removeBeginning()
{
    if(head == NULL)
        printf("\n\nList is Empty!!!!");
    else
    {
        struct Node *temp = head;
        if(head->next == NULL)
        {
            head = NULL;
            free(temp);
        }
        else
        {
            head = temp->next;
            free(temp);
            printf("\nOne node deleted!!!\n\n");
        }
    }
}

```

Routine to Delete a Node From the Beginning

## DELETING FROM END OF THE LIST

Steps involved in deleting a node from the end of the list are as follows:

**Step 1:** Check whether list is Empty (**head == NULL**)

**Step 2:** If it is Empty then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.

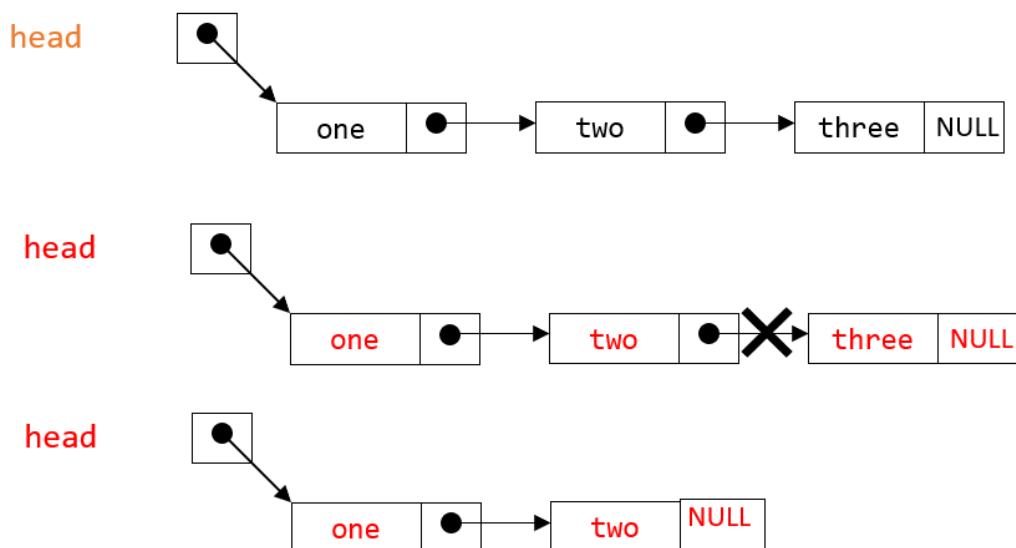
**Step 3:** If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize '**temp1** → **head**.

**Step 4:** Check whether list has only one Node (**temp1 → next == NULL**)

**Step 5:** If it is **TRUE**. Then, set **head = NULL** and **delete temp1**. And terminate the function. (Setting Empty list condition)

**Step 6:** If it is **FALSE**. Then, set '**temp2 = temp1**' and **move temp1 to its next node**. Repeat the same until it reaches to the last node in the list. (**until temp1 → next == NULL**)

**Step 7:** Finally, Set **temp2 → next = NULL** and **delete temp1**.



```
void removeEnd()  
{  
    if(head == NULL)  
    {  
        printf("\nList is Empty!!!\n");  
    }  
    else  
    {  
        struct Node *temp1 = head, *temp2;  
        if(head->next == NULL)  
            head = NULL;  
        else  
        {  
            while(temp1->next != NULL)  
            {  
                temp2 = temp1;  
                temp1 = temp1->next;  
            }  
            temp2->next = NULL;  
        }  
        free(temp1);  
        printf("\nOne node deleted!!!\n\n");  
    }  
}
```

**Routine to delete a node from the end**

## **DELETING A SPECIFIC NODE FROM THE LIST**

Steps involved in deleting a specific node from the list are as follows:

**Step 1:** Check whether list is **Empty (head == NULL)**

**Step 2:** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.

**Step 3:** If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize '**temp1**' with **head**.

**Step 4:** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.

**Step 5:** If it is reached to the last node then display '**Given node not found in the list! Deletion not possible!!!**'. And terminate the function.

**Step 6:** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

**Step 7:** If list has only one node and that is the node to be deleted, then set **head = NULL** and delete **temp1 (free(temp1))**.

**Step 8:** If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).

**Step 9:** If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.

**Step 10:** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == NULL**).

**Step 11:** If **temp1** is last node then set **temp2 → next = NULL** and delete **temp1 (free(temp1))**.

**Step 12:** If **temp1** is not first node and not last node then set **temp2 → next = temp1 → next** and delete **temp1 (free(temp1))**.

1. Use a Node Pointer to trace to the current position and also to store the value of the next address
2. Use a Previous Node Pointer to trace to the current position

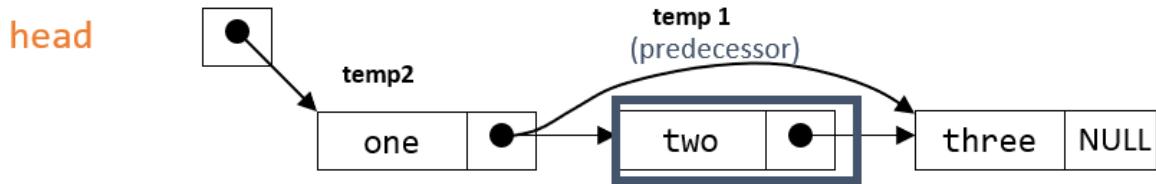
**temp2=temp1;**

**temp1=temp1->next;**

3. If the element to be deleted is found then

**temp2->next=temp1->next;**

**free( temp1);**



```

void removeSpecific(int delValue)
{
    struct Node *temp1 = head, *temp2;
    while(temp1->data != delValue)
    {
        if(temp1 -> next == NULL){
            printf("\nGiven node not found in the list!!!!");
        }
        temp2 = temp1;
        temp1 = temp1 -> next;
    }
    temp2 -> next = temp1 -> next;
    free(temp1);
    printf("\nOne node deleted!!!\n\n");
}

```

### Routine to delete a specific node

## APPLICATIONS OF LINKED LIST

The applications of a linked list are as follows. They are

- Used to implement **stacks and queues**.
- Used to implement the **Adjacency list representation** of graphs.
- Used to perform **dynamic memory allocation**.
- Used to **Maintain directory of names**.
- Used to **perform arithmetic operations** on long integers.
- Used to **manipulate polynomials** to store constants.
- Used to **represent sparse matrices**.

## POLYNOMIAL ARITHMETIC

The manipulation of symbolic polynomials, has a classic example of list processing. In general, we want to represent the polynomial:

$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0x^{e_0}$$

Where the  $a_i$  are nonzero coefficients and the  $e_i$  are nonnegative integer exponents such that  $e_{m-1} > e_{m-2} > \dots > e_1 > e_0 \geq 0$ .

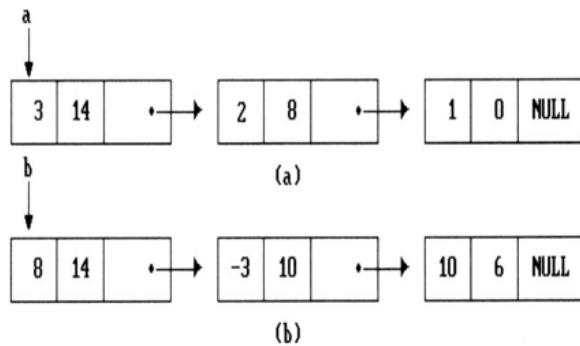
We will represent each term as a node containing **coefficient** and **exponent** fields, as well as a pointer to the next term. A node structure for a polynomial is shown below.



Assuming that the coefficients are integers, the node structure will be declared as follows:

```
struct link
{
    int coeff;
    int pow;
    struct link *next;
};
```

Consider the polynomials  $a = 3x^{14} + 2x^8 + 1$  and  $b = 8x^{14} - 3x^{10} + 10x^6$ . The linked list representation of the polynomials a and b are shown below.



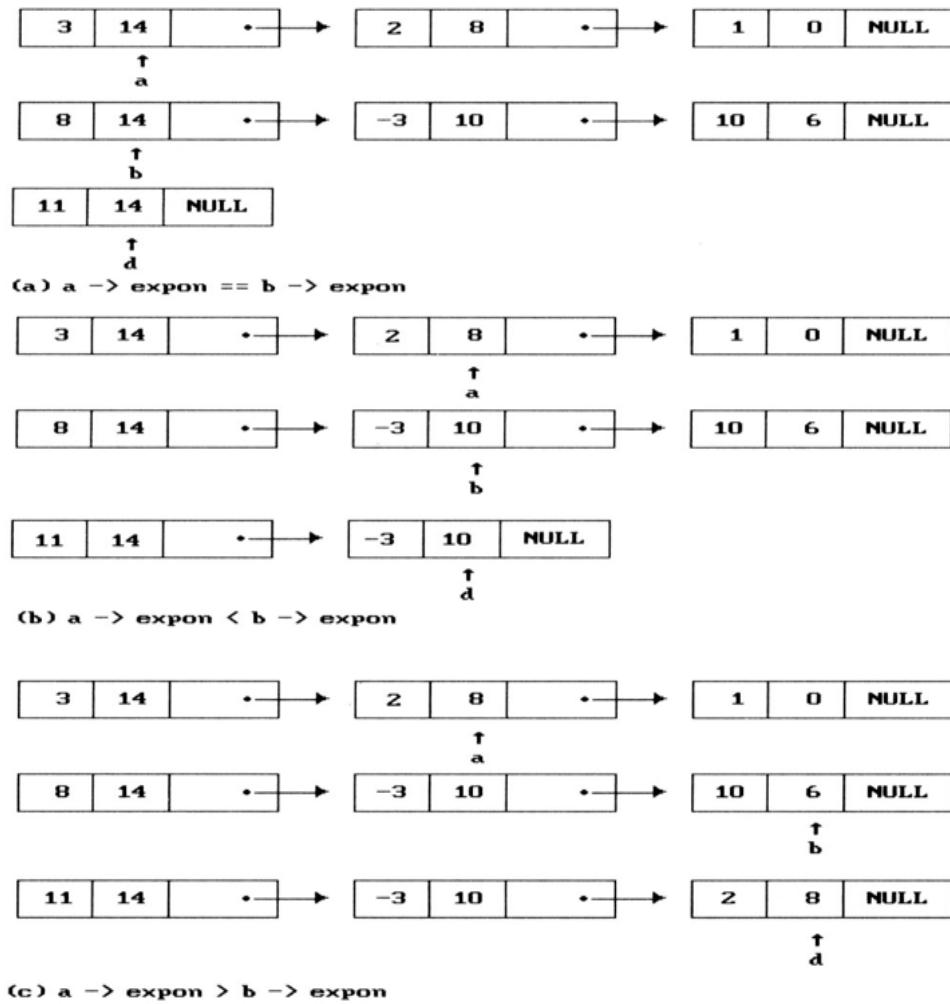
```
void create(struct link *node)
{
    char ch;
    do
    {
        printf("\n enter coeff:");
        scanf("%d",&node->coeff);
        printf("\n enter power:");
        scanf("%d",&node->pow);
        node=(struct link*)malloc(sizeof(struct link));
        node->next=NULL;
        printf("\n continue(y/n):");
        ch=getch();
    }
    while(ch=='y' || ch=='Y');
}
```

### Routine to Create a Polynomial Linked List

## POLYNOMIAL ADDITION

- To add two polynomials, we examine their terms starting at the nodes pointed to by  $a$  and  $b$ .
  - If the exponents of the two terms are equal
    - add the two coefficients and create a new term for the result.
  - If the exponent of the current term in  $a$  is less than  $b$ 
    - create a duplicate term of  $b$  and attach this term to the result, called  $d$
    - advance the pointer to the next term in  $b$ .
  - We take a similar action on  $a$  if  $a->expon > b->expon$ .

The figure shows generating the first three term of  $d = a+b$  below.



From the above figure it is observed that when the exponents of  $a$  and  $b$  are equal, coefficients of  $a$  and  $b$  are added in the new polynomial  $d$ . Else if the exponent of  $a$  is bigger than exponent of  $b$ , link the corresponding node of  $a$  to  $d$  and vice versa. Routine to create a linked list for a polynomial is give below.

```

void polyadd(struct link *poly1,struct link *poly2,struct link *poly)
{
    while(poly1->next && poly2->next)
    {
        if(poly1->pow>poly2->pow)
        {
            poly->pow=poly1->pow;
            poly->coeff=poly1->coeff;
            poly1=poly1->next;
        }
        else if(poly1->pow<poly2->pow)
        {
            poly->pow=poly2->pow;
            poly->coeff=poly2->coeff;
            poly2=poly2->next;
        }
        else
        {
            poly->pow=poly1->pow;
            poly->coeff=poly1->coeff+poly2->coeff;
            poly1=poly1->next;
            poly2=poly2->next;
        }
        poly->next=(struct link *)malloc(sizeof(struct link));
        poly=poly->next;
        poly->next=NULL;
    }

    while(poly1->next || poly2->next)
    {
        if(poly1->next)
        {
            poly->pow=poly1->pow;
            poly->coeff=poly1->coeff;
            poly1=poly1->next;
        }
        if(poly2->next)
        {
            poly->pow=poly2->pow;
            poly->coeff=poly2->coeff;
            poly2=poly2->next;
        }
        poly->next=(struct link *)malloc(sizeof(struct link));
        poly=poly->next;
        poly->next=NULL;
    }
}

```

### Routine to Add Two Polynomials

## CURSOR BASED - IMPLEMENTATION METHODOLOGY

If linked lists are required and pointers are not available, then an alternate implementation must be used. The alternate method we will describe is known as a cursor implementation.

The two important items present in a pointer implementation of linked lists are

1. The data is stored in a collection of structures. Each structure contains the data and a pointer to the next structure.
2. A new structure can be obtained from the system's global memory by a call to malloc and released by a call to free.

Our cursor implementation must be able to simulate this. The logical way to satisfy condition 1 is to have a global array of structures. For any cell in the array, its array index can be used in place of an address. The type declarations for a cursor implementation of linked lists is given below.

```
typedef unsigned int node_ptr;

struct node
{
    element_type element;
    node_ptr next;
};

typedef node_ptr LIST;
typedef node_ptr position;
struct node CURSOR_SPACE[ SPACE_SIZE ];
```

### Declarations For Cursor Implementation Of Linked Lists

We must now simulate condition 2 by allowing the equivalent of malloc and free for cells in the CURSOR\_SPACE array. To do this, we will keep a list (the freelist) of cells that are not in any list. The list will use cell 0 as a header. The initial configuration is shown in the figure below.

| Slot | Element | Next |
|------|---------|------|
| 0    | ?       | 1    |
| 1    | ?       | 2    |
| 2    | ?       | 3    |
| 3    | ?       | 4    |
| 4    | ?       | 5    |
| 5    | ?       | 6    |
| 6    | ?       | 7    |
| 7    | ?       | 8    |
| 8    | ?       | 9    |
| 9    | ?       | 10   |
| 10   | ?       | 0    |

An Initialized Cursor Space

A value of 0 for next is the equivalent of a pointer. The initialization of CURSOR\_SPACE is a straightforward loop, which we leave as an exercise. To perform an malloc, the first element (after the header) is removed from the freelist.

To perform a free, we place the cell at the front of the freelist. The below routine shows the cursor implementation of malloc and free.

```

position cursor_alloc( void )
{
    position p;
    p = CURSOR_SPACE[0].next;
    CURSOR_SPACE[0].next = CURSOR_SPACE[p].next;
    return p;
}

void cursor_free( position p)
{
    CURSOR_SPACE[p].next = CURSOR_SPACE[0].next;
    CURSOR_SPACE[0].next = p;
}

```

### Routines: cursor-alloc and cursor-free

Notice that if there is no space available, our routine does the correct thing by setting  $p = 0$ . This indicates that there are no more cells left, and also makes the second line of cursor\_new a nonoperation (no-op). Given this, the cursor implementation of linked lists is straightforward. For consistency, we will implement our lists with a header node.

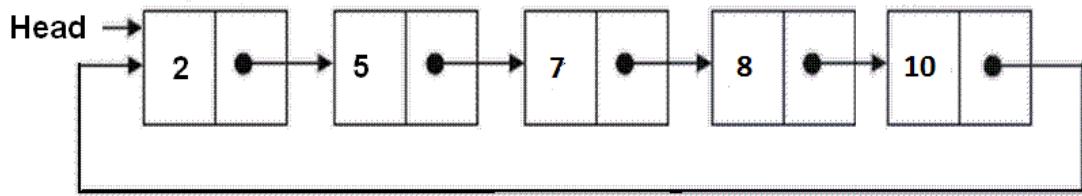
An example of cursor implementation of linked list is shown below. In the figure, if the value of L is 5 and the value of M is 3, then L represents the list a, b, e, and M represents the list c, d, f.

| Slot  | Element | Next |
|-------|---------|------|
| <hr/> |         |      |
| 0     | -       | 6    |
| 1     | b       | 9    |
| 2     | f       | 0    |
| 3     | header  | 7    |
| 4     | -       | 0    |
| 5     | header  | 10   |
| 6     | -       | 4    |
| 7     | c       | 8    |
| 8     | d       | 2    |
| 9     | e       | 0    |
| 10    | a       | 1    |

### Example of a cursor implementation of linked lists

## CIRCULAR LINKED LIST

Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.



### ADVANTAGES OF CIRCULAR LINKED LISTS

- 1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
- 2) Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
- 3) Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.
- 4) Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

### OPERATIONS ON CIRCULARLY LINKED LIST

In a circular linked list, following operations are performed,

1. Insertion
2. Deletion
3. Display

First perform the following steps before implementing actual operations.

**Step 1** - Include all the header files which are used in the program.

**Step 2** - Declare all the user defined functions.

**Step 3** - Define a Node structure with two members data and next

**Step 4** - Define a Node pointer 'head' and set it to NULL.

**Step 5** - Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

## Circular linked list – Creation

```
head = new;
new->next=head;
```

### Insertion in a Circular Linked List

In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting at Beginning of the list
2. Inserting at End of the list
3. Inserting at Specific location in the list

### INSERTING AT BEGINNING OF THE LIST

Steps to **insert a new node at beginning** of the circular linked list...

**Step 1** - Create a **newNode** with given value.

**Step 2** - Check whether list is **Empty (head == NULL)**

**Step 3** - If it is **Empty** then, set **head = newNode** and **newNode→next = head** .

**Step 4** - If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.

**Step 5** - Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp → next == head**').

**Step 6** - Set '**newNode → next =head**', '**head = newNode**' and '**temp → next = head**'.

### INSERTING AT END OF THE LIST

Steps to insert a new node at end of the circular linked list...

**Step 1** - Create a **newNode** with given value.

**Step 2** - Check whether list is **Empty (head == NULL)**.

**Step 3** - If it is **Empty** then, set **head = newNode** and **newNode → next = head**.

**Step 4** - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

**Step 5** - Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next == head**).

**Step 6** - Set **temp → next = newNode** and **newNode → next = head**.

## **INSERTING AT SPECIFIC LOCATION IN THE LIST (AFTER A NODE)**

Steps to insert a new node at a specific location of the circular linked list...

**Step 1** - Create a **newNode** with given value.

**Step 2** - Check whether list is **Empty (head == NULL)**

**Step 3** - If it is **Empty** then, set **head = newNode** and **newNode → next = head**.

**Step 4** - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

**Step 5** - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the **newNode** (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the **newNode**).

**Step 6** - Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display '**Given node is not found in the list!!! Insertion not possible!!!**' and terminate the function. Otherwise move the **temp** to next node.

**Step 7** - If **temp** is reached to the exact node after which we want to insert the **newNode** then check whether it is last node (**temp → next == head**).

**Step 8** - If **temp** is last node then set **temp → next = newNode** and **newNode → next = head**.

**Step 8** - If **temp** is not last node then set **newNode → next = temp → next** and **temp → next = newNode**.

## **CLL – INSERTION (ROUTINE)**

```
Insertion at first  
new->next = head  
head =new;  
temp->next= head;  
Insertion at middle  
n1->next = new;  
new->next = n2;  
n2->next = head;  
Insertion at last  
n2->next=new;  
new->next = head;
```

## **DELETION**

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

### **DELETING FROM BEGINNING OF THE LIST**

The following steps to delete a node from beginning of the circular linked list...

- Step 1** - Check whether list is **Empty** (**head == NULL**)
- Step 2** - If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- Step 3** - If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.
- Step 4** - Check whether list is having only one node (**temp1 → next == head**)
- Step 5** - If it is **TRUE** then set **head = NULL** and delete **temp1** (Setting **Empty** list conditions)
- Step 6** - If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next == head** )
- Step 7** - Then set **head = temp2 → next**, **temp1 → next = head** and delete **temp2**.

### **DELETING FROM END OF THE LIST**

The following steps to delete a node from end of the circular linked list...

- **Step 1** - Check whether list is **Empty** (**head == NULL**)
- **Step 2** - If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- **Step 3** - If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4** - Check whether list has only one Node (**temp1 → next == head**)
- **Step 5** - If it is **TRUE**. Then, set **head = NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6** - If it is **FALSE**. Then, set '**temp2 = temp1**' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1 → next == head**)
- **Step 7** - Set **temp2 → next = head** and delete **temp1**.

### **DELETING A SPECIFIC NODE FROM THE LIST**

The following steps to delete a specific node from the circular linked list...

- Step 1** - Check whether list is **Empty** (**head == NULL**)
- Step 2** - If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- Step 3** - If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize '**temp1**' with **head**.
- Step 4** - Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.

**Step 5** - If it is reached to the last node then display '**Given node not found in the list! Deletion not possible!!!**'. And terminate the function.

**Step 6** - If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next == head**)

**Step 7** - If list has only one node and that is the node to be deleted then set **head = NULL** and delete **temp1 (free(temp1))**.

**Step 8** - If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).

**Step 9** - If **temp1** is the first node then set **temp2 = head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next, temp2 → next = head** and delete **temp1**.

**Step 10** - If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).

**Step 11** - If **temp1** is last node then set **temp2 → next = head** and delete **temp1 (free(temp1))**.

**Step 12** - If **temp1** is not first node and not last node then set **temp2 → next = temp1 → next** and delete **temp1 (free(temp1))**.

## APPLICATIONS OF CIRCULAR LIST -JOSEPH PROBLEM

There are n people standing in a circle waiting to be executed. The counting out begins at some point in the circle and proceeds around the circle in a fixed direction. In each step, a certain number of people are skipped and the next person is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last person remains, who is given freedom. Given the total number of persons n and a number m which indicates that m-1 persons are skipped and m-th person is killed in circle. The task is to choose the place in the initial circle so that you are the last one remaining and so survive.

Examples:

```
Input : Length of circle : n = 4
```

```
Count to choose next : m = 2
```

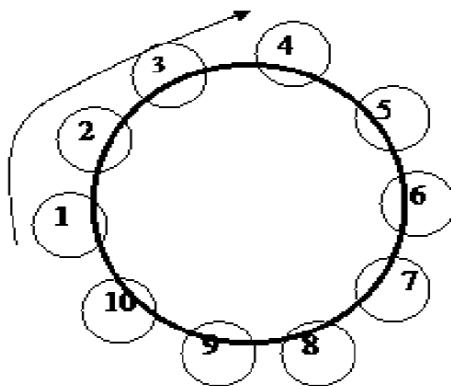
```
Output : 1
```

```
Input : n = 5
```

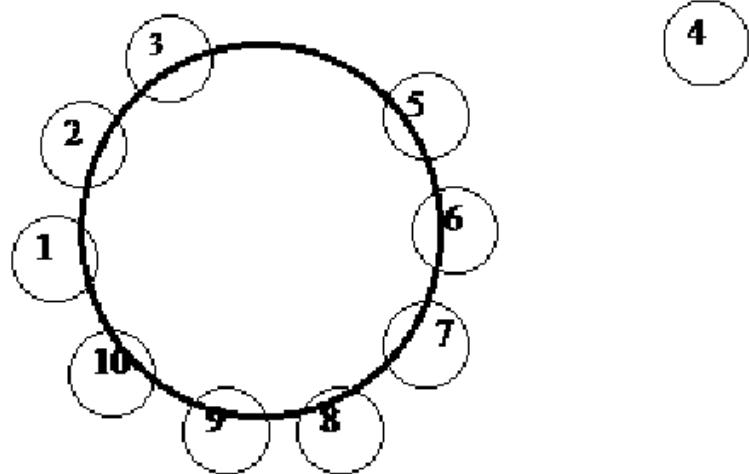
```
m = 3
```

```
Output : 4
```

**N=10, M=3**

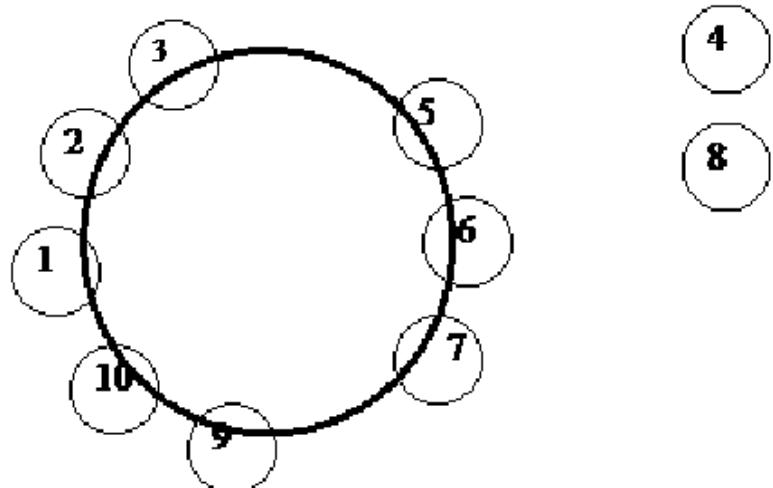


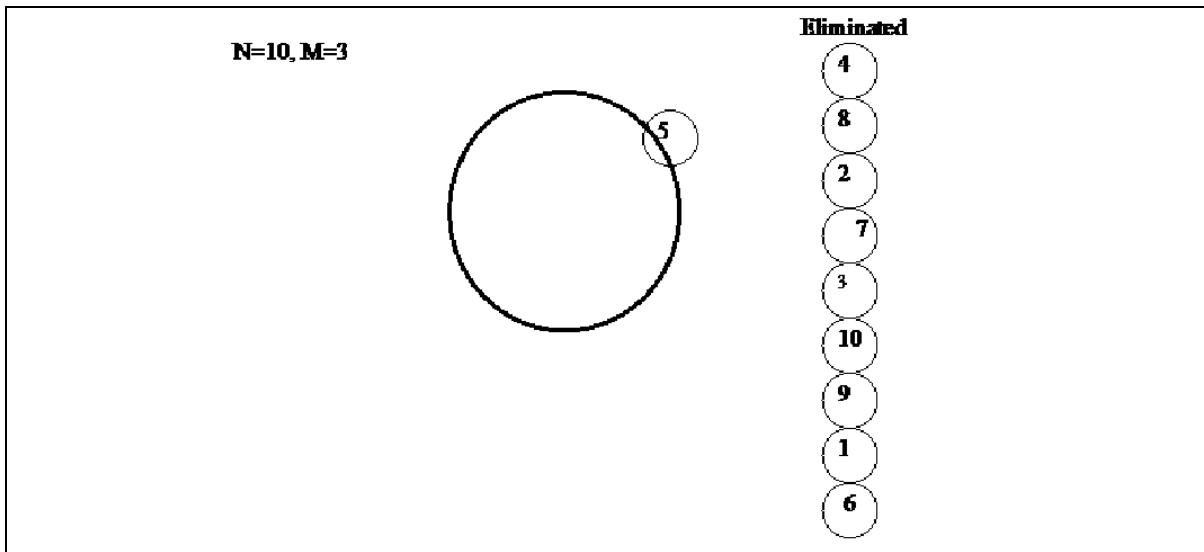
**N=10, M=3**



**Eliminated**

**N=10, M=3**





```

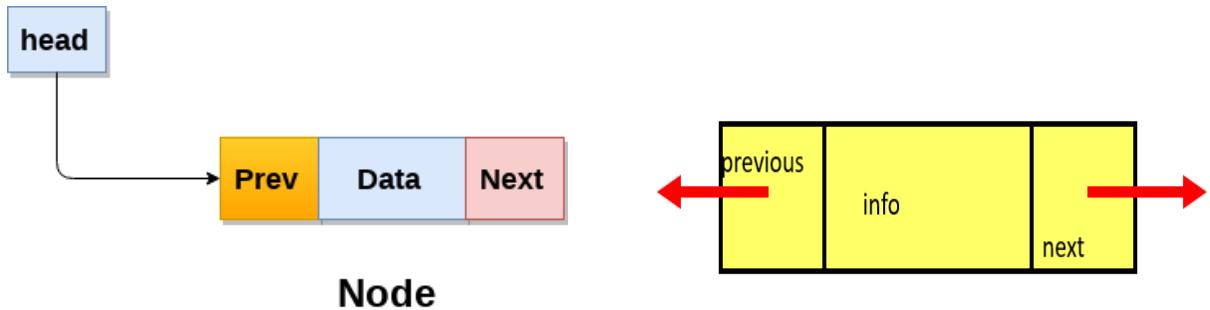
int josephus(int m, node *head)
{
    node *f;
    int c=1;
    while(head->next!=head)
    {
        c=1;
        while(c!=m)
        {
            f=head;
            head=head->next;
            c++;
        }
        f->next=head->next;
        printf("%d->",head->data); //sequence in which nodes getting deleted
        head=f->next;
    }
    printf("\n");
    printf("Winner is:%d\n",head->data);
    return;
}

```

### ROUTINE FOR JOSEPHUS PROBLEM

#### DOUBLY LINKED LIST

- Each node contains a value, a link to its successor (if any), and a link to its predecessor (if any)
- The header points to the first node in the list and to the last node in the list (or contains null links if the list is empty)
- A simple node structure of a doubly linked list is shown below:



A node structure in doubly linked list is declared as follows:

```
struct node
{
    struct node *left;
    int data;
    struct node *right;
};
```

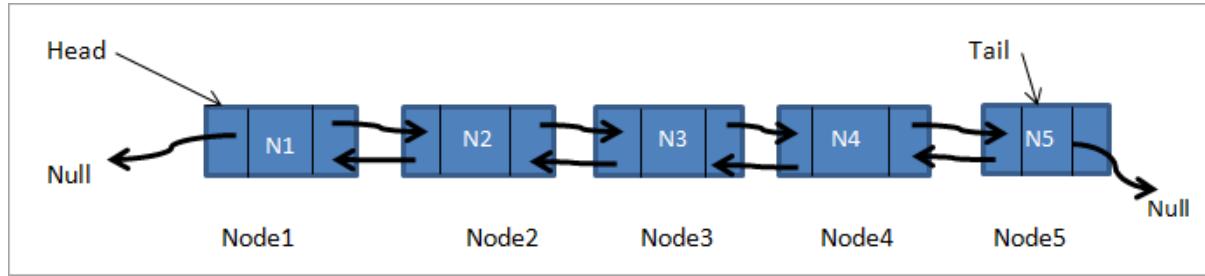
### **OPERATIONS IN A DOUBLY LINKED LIST ARE AS FOLLOWS**

1. Node Creation
2. Insertion at beginning
3. Insertion at end
4. Insertion after specified node
5. Deletion at beginning
6. Deletion at end
7. Deletion of the node having given data
8. Searching
9. Traversing

### **NODE CREATION**

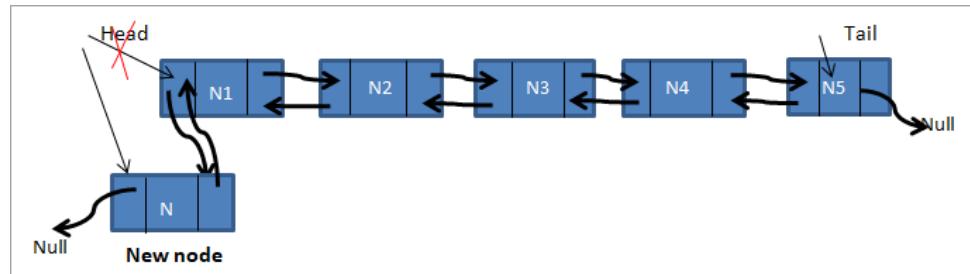
```
struct node
{
    struct node *left;
    int data;
    struct node *right;
};
struct node *head;
```

## INSERTION AT BEGINNING



```
void insert_at_begin()
{
    struct node *temp;
    int item;
    temp = (struct node *)malloc(sizeof(struct node));
    printf("\nEnter Item value");
    scanf("%d",&item);

    if(head==NULL)
    {
        temp->right = NULL;
        temp->left=NULL;
        temp->data=item;
        head=temp;
    }
    else
    {
        temp->data=item;
        temp->left=NULL;
        temp->right = head;
        head->left=temp;
        head=temp;
    }
    printf("\nInsertion completed\n");
}
```



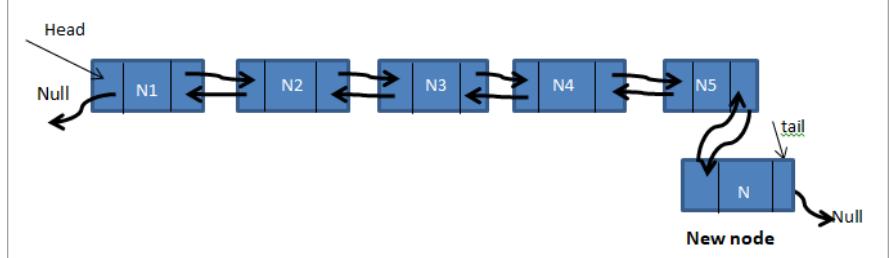
## INSERTION AT END

```
void insert_at_last()
{
    struct node *temp,*temp;
    int item;
    temp = (struct node *) malloc(sizeof(struct node));
    printf("\nEnter value");
    scanf("%d",&item);
```

```

tempptr->data=item;
if(head == NULL)
{
    tempptr->right = NULL;
    tempptr->left = NULL;
    head = tempptr;
}
else
{
    temp = head;
    while(temp->right!=NULL)
    {
        temp = temp->right;
    }
    temp->right = tempptr;
    tempptr ->left=temp;
    tempptr->right = NULL;
}
printf("\n insertion completed \n");
}

```



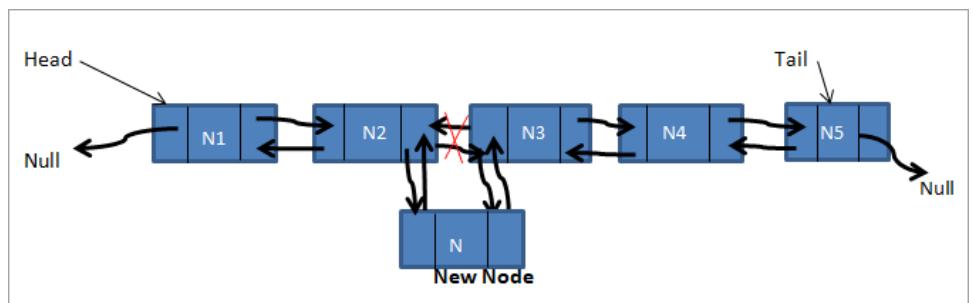
## INSERTION AFTER SPECIFIED NODE

```

void insert_after_spcific_node()
{
    struct node *tempPTR,*temp;
    int item,loc,i;
    tempPTR = (struct node *)malloc(sizeof(struct node));

    temp=head;
    printf("Enter the location");
    scanf("%d",&loc);
    for(i=0;i<loc;i++)
    {
        temp = temp->right;
        if(temp == NULL)
        {
            printf("\n There are less than %d elements", loc);
            return;
        }
    }
    printf("Enter data");
    scanf("%d",&item);
    tempPTR->data = item;
}

```



```

temp->right = temp->right;
temp->left = temp;
temp->right = temp;
temp->right->left=temp;
printf("\n insertion completed \n");

}

```

### **DELETION AT BEGINNING**

```

void delete_at_begin()
{
    struct node *temp;
    if(head == NULL)
    {
        printf("\n UNDERFLOW");
    }
    else if(head->right == NULL)
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    }
    else
    {
        temp = head;
        head = head -> right;
        head -> left = NULL;
        free(temp);
        printf("\nnode deleted\n");
    }
}

```

}

### **DELETION AT END**

```

void delete_at_last()
{
    struct node *temp;
    if(head == NULL)
    {
        printf("\n UNDERFLOW");
    }
    else if(head->right == NULL)
    {
        head = NULL;
    }
}

```

```

        free(head);
        printf("\nnode deleted\n");
    }
else
{
    tempptr = head;
    if(tempptr->right != NULL)
    {
        tempptr = tempptr -> right;
    }
    tempptr -> left -> right = NULL;
    free(tempptr);
    printf("\nnode deleted\n");
}
}

```

**DISPLAY\_FULL\_LIST LIST**

```

void display_full_list()
{
    struct node *tempptr;
    printf("\n printing values...\n");
    tempptr = head;
    while(tempptr != NULL)
    {
        printf("%d\n",tempptr->data);
        tempptr=tempptr->right;
    }
}

```

#### **MAIN FUNCTION:**

```

#include<stdio.h>
#include<stdlib.h>
struct node
{
    struct node *prev;
    struct node *next;
    int data;
};
struct node *head;
void insert_at_begin();
void insert_at_last();
void insert_after_specific_node();
void delete_at_begin();
void delete_at_last();

```

```
void deletion_specified();
void display_full_list();
void search();
void main ()
{
int option =0;
while(option != 9)
{
    printf("\nChoose option from below menu\n");
    printf("\n1.Insert begining\n 2.Insert last\n 3.Insert any random location \n 4.Delete
the first node \n 5.Delete the last node\n6.Delete the node after the given
data\n7.Search\n8.Show\n9.Exit\n");
    printf("\nEnter your option?\n");
    scanf("\n%d",&option);
    switch(option)
    {
        case 1:
            insert_at_begin();
            break;
        case 2:
            insert_at_last();
            break;
        case 3:
            insert_after_specific_node();
            break;
        case 4:
            delete_at_begin();
            break;
        case 5:
            delete_at_last();
            break;
        case 6:
            deletion_specified();
            break;
        case 7:
            search();
            break;
        case 8:
            display_full_list();
            break;
        case 9:
            exit(0);
            break;
        default:
```

```
    printf("Please enter valid option..");
}
}
}
```

**OUTPUT:**

Choose option from below menu

- 1.Insert begining
- 2.Insert last
- 3.Insert any random location
- 4.Delete the first node
- 5. Delete the last node
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your option?

1

Enter Item value 10

Insertion completed.

Choose option from below menu

- 1.Insert begining
- 2.Insert last
- 3.Insert any random location
- 4.Delete the first node
- 5. Delete the last node
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your option?

1

Enter Item value 20

Insertion completed.

Choose option from below menu

- 1.Insert begining
- 2.Insert last
- 3.Insert any random location
- 4.Delete the first node
- 5. Delete the last node
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your option?

8

Printing values...

20

10

Choose option from below menu

1.Insert begining

2.Insert last

3.Insert any random location

4.Delete the first node

5. Delete the last node

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your option?

4

Node deleted

Choose option from below menu

1.Insert begining

2.Insert last

3.Insert any random location

4.Delete the first node

5. Delete the last node

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your option?

8

Printing values...

10

Choose option from below menu

1.Insert begining

2.Insert last

3.Insert any random location

4.Delete the first node

5. Delete the last node

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your option?

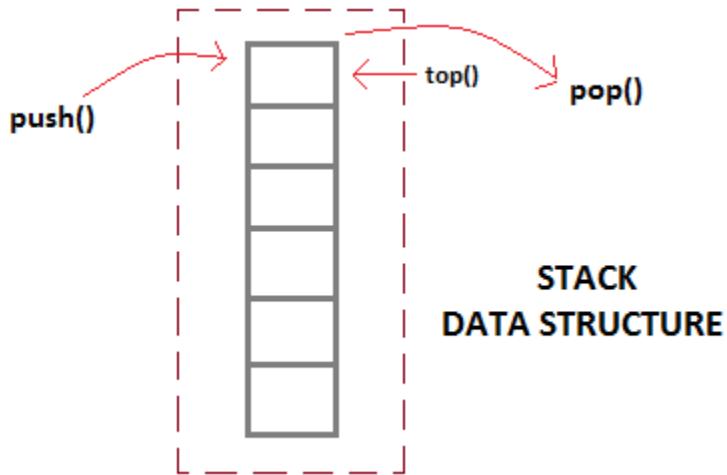
**Explanation:**

This program having many functions each performing one operation in doubly linked list. When the program is executed, list of options or operation that we can perform in the linked list displayed. Each option is associated with one function which is called by the switch case based on the user input. When the input 1 is given after executing program, the function `insert_at_begin();` is called. It creating the new node with the user entered data 10. Again new node is inserted at the beginning using the same function with value 20. Right now the doubly list having the two node. The first node data is 20 and second is 10. We can display the list by calling display full list function through option 8. Then the first node is deleted from the list by giving option 4 that calling `deleting_at_begin` function. After this operation, the list would have only one value that is 10. Likewise, all the operation in the doubly linked can be performed using above writing function.

## UNIT III- STACK & QUEUE

### WHAT IS STACK DATA STRUCTURE?

Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the **top** of the stack and the only element that can be removed is the element that is at the top of the stack, just like a pile of objects.



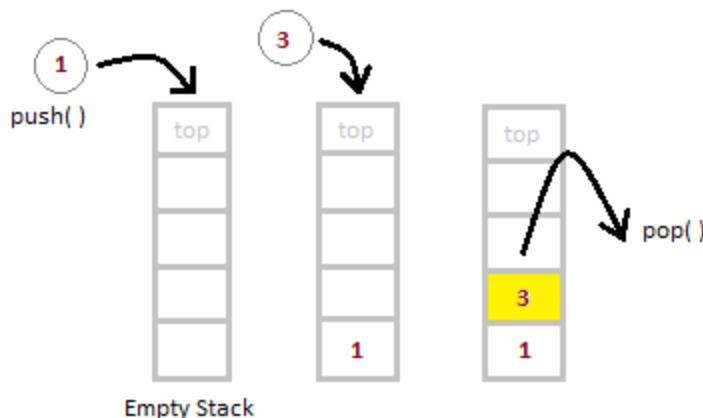
### Basic features of Stack

1. Stack is an **ordered list of similar data type**.
2. Stack is a **LIFO**(Last in First out) structure or we can say **FILO**(First in Last out).
3. **push()** function is used to insert new elements into the Stack and **pop()** function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called **Top**.
4. Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

### IMPLEMENTATION OF STACK DATA STRUCTURE

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.

### STACK - LIFO Structure



In a Stack, all operations take place at the "top" of the stack. The "push" operation adds an item to the top of the Stack.  
The "pop" operation removes the item on top of the stack.

## Implementation of Stack using Array

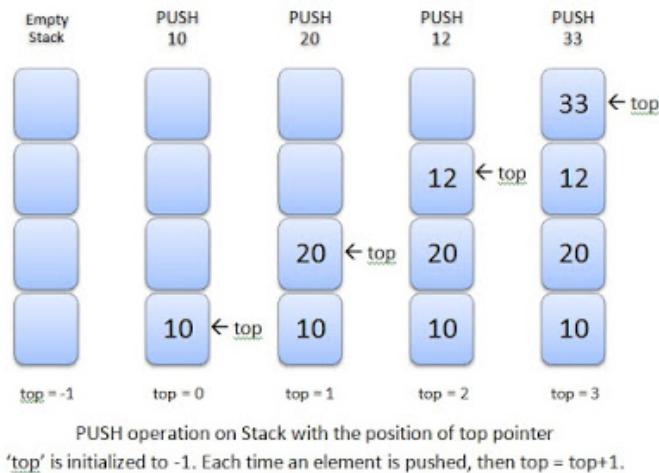
- A stack data structure can be implemented using a one-dimensional array
- Array stores only a fixed number of data values
- Implementation is very simple
- Define a one dimensional array of specific size
  - insert or delete the values into the array by using **LIFO principle** with the help of a variable called '**top**'
  - Initially, the top is set to -1
  - To insert a value into the stack, increment the top value by one and then insert
  - To delete a value from the stack, delete the top value and decrement the top value by one

### Stack Array Creation

- Include all the **header files** which are used in the program and define a constant '**SIZE**' with specific value.
- Declare all the **functions** used in stack implementation.
- Create a one dimensional array with fixed size (**int stack[SIZE]**)
- Define a integer variable '**top**' and initialize with '**-1**'. (**int top = -1**)
- In main method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

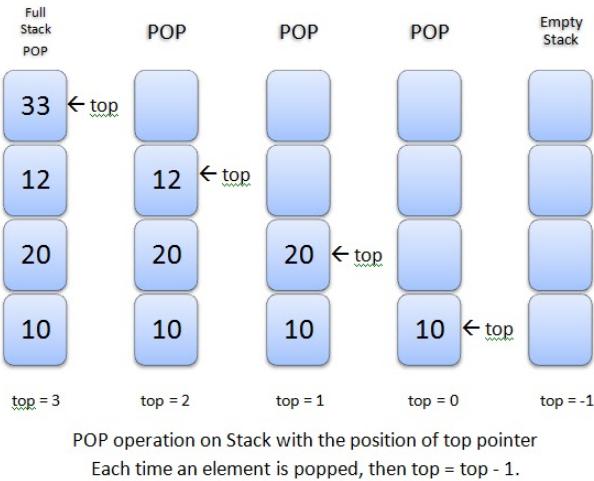
## Push Algorithm

- `push(val)` is a function used to insert a new element into stack at **top** position.
- Push function takes one integer value as parameter
  1. Check whether **stack** is **FULL**. (**top == SIZE-1**)
  2. If it is **FULL**, then display "**Stack is FULL!!! Insertion is not possible!!!**" and terminate the function.
  3. If it is **NOT FULL**, then increment **top** value by one (**top++**) and set **stack[top]** to value (**stack[top] = value**)



## Pop Algorithm

- `pop()` is a function used to delete an element from the stack from **top** position
- Pop function does not take any value as parameter
  1. Check whether **stack** is **EMPTY**. (**top == -1**)
  2. If it is **EMPTY**, then display "**Stack is EMPTY!!! Deletion is not possible!!!**" and terminate the function
  3. If it is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one (**top--**)



## Display Algorithm

- **display()** - Displays the elements of a Stack
  1. Check whether stack is **EMPTY**. (**top == -1**)
  2. If it is **EMPTY**, then display "**Stack is EMPTY!!!**" and terminate the function
  3. If it is **NOT EMPTY**, then define a variable '**i**' and initialize with **top**.  
Display **stack[i]** value and decrement **i** value by one (**i--**)
  4. Repeat above step until **i** value becomes '**0**'.

## Disadvantages of Array

- The amount of data must be specified at the beginning of the implementation
- Stack implemented using an array is not suitable, when the size of data is unknown

## C Program CODE

```
#include<stdio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{
    top=-1;
    printf("\n Enter the size of STACK[MAX=100]:");
    scanf("%d",&n);
    printf("\n\t STACK OPERATIONS USING ARRAY");
    printf("\n\t-----");
    printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
    do
```

```

{
    printf("\n Enter the Choice:");
    scanf("%d",&choice);
    switch(choice)
    {
        case 1:
        {
            push();
            break;
        }
        case 2:
        {
            pop();
            break;
        }
        case 3:
        {
            display();
            break;
        }
        case 4:
        {
            printf("\n\t EXIT POINT ");
            break;
        }
        default:
        {
            printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
        }
    }
}
while(choice!=4);
return 0;
}

void push()
{
    if(top>=n-1)
    {
        printf("\n\tSTACK is over flow");
    }
    else
    {
        printf(" Enter a value to be pushed:");
        scanf("%d",&x);
    }
}

```

```

        top++;
        stack[top]=x;
    }
}
void pop()
{
    if(top<=-1)
    {
        printf("\n\t Stack is under flow");
    }
    else
    {
        printf("\n\t The popped elements is %d",stack[top]);
        top--;
    }
}
void display()
{
    if(top>=0)
    {
        printf("\n The elements in STACK \n");
        for(i=top; i>=0; i--)
            printf("\n%d",stack[i]);
        printf("\n Press Next Choice");
    }
    else
    {
        printf("\n The STACK is empty");
    }
}

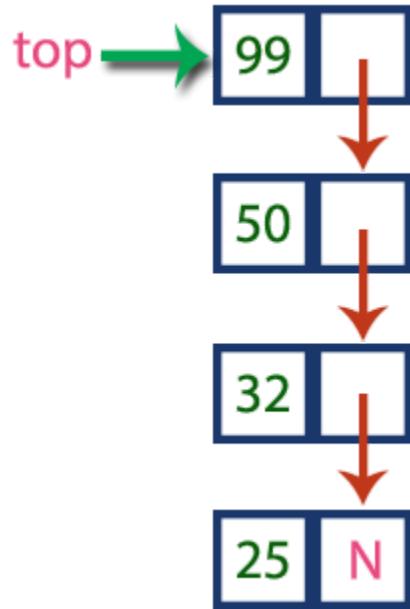
```

## Implementation of Stack using Linked List

A stack data structure can be implemented by using a linked list data structure. The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first element must be always **NULL**.

## Example



In the above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32, 50 and 99.

## Stack Operations using Linked List

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

- **Step 1** - Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2** - Define a '**Node**' structure with two members **data** and **next**.
- **Step 3** - Define a **Node** pointer '**top**' and set it to **NULL**.
- **Step 4** - Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

### **push(value)** - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

- **Step 1** - Create a **newNode** with given value.
- **Step 2** - Check whether stack is **Empty** (**top == NULL**)
- **Step 3** - If it is **Empty**, then set **newNode → next = NULL**.
- **Step 4** - If it is **Not Empty**, then set **newNode → next = top**.
- **Step 5** - Finally, set **top = newNode**.

### **pop()** - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

- **Step 1** - Check whether stack is **Empty** (**top == NULL**).
- **Step 2** - If it is **Empty**, then display "Stack is Empty!!! Deletion is not possible!!!" and terminate the function
- **Step 3** - If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
- **Step 4** - Then set '**top = top → next**'.
- **Step 5** - Finally, delete '**temp**'. (**free(temp)**).

### **display() - Displaying stack of elements**

We can use the following steps to display the elements (nodes) of a stack...

- **Step 1** - Check whether stack is **Empty** (**top == NULL**).
- **Step 2** - If it is **Empty**, then display '**Stack is Empty!!!**' and terminate the function.
- **Step 3** - If it is **Not Empty**, then define a Node pointer '**temp**' and initialize with **top**.
- **Step 4** - Display '**temp → data --->**' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp → next != NULL**).
- **Step 5** - Finally! Display '**temp → data ---> NULL**'.

### **C Program Code**

```
#include<stdio.h>
```

```
#include<conio.h>
```

```
struct Node
```

```
{
```

```
    int data;
```

```
    struct Node *next;
```

```
}*top = NULL;
```

```
void push(int);
```

```
void pop();
```

```
void display();
```

```
void main()
```

```
{
```

```
    int choice, value;
```

```
    clrscr();
```

```

printf("\n:: Stack using Linked List ::\n");
while(1){
    printf("\n***** MENU *****\n");
    printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice){
        case 1: printf("Enter the value to be insert: ");
            scanf("%d", &value);
            push(value);
            break;
        case 2: pop(); break;
        case 3: display(); break;
        case 4: exit(0);
        default: printf("\nWrong selection!!! Please try again!!!\n");
    }
}
}

void push(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    if(top == NULL)
        newNode->next = NULL;
    else
        newNode->next = top;
    top = newNode;
    printf("\nInsertion is Success!!!\n");
}

```

```

void pop()
{
    if(top == NULL)
        printf("\nStack is Empty!!!\n");
    else{
        struct Node *temp = top;
        printf("\nDeleted element: %d", temp->data);
        top = temp->next;
        free(temp);
    }
}

void display()
{
    if(top == NULL)
        printf("\nStack is Empty!!!\n");
    else{
        struct Node *temp = top;
        while(temp->next != NULL){
            printf("%d--->",temp->data);
            temp = temp -> next;
        }
        printf("%d--->NULL",temp->data);
    }
}

```

## Array vs Linked List

| <b>ARRAY</b>                                            | <b>LINKED LIST</b>                                                                                     |
|---------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| Array is a collection of elements of similar data type. | Linked List is an ordered collection of elements of same type, which are connected to each other using |

|                                                                                                                                                                                                                                                                                                 |                                                                                                                                                                                                                                                                                                  |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                                                                                                                                                                                                                                                                 | pointers.                                                                                                                                                                                                                                                                                        |
| <p>Array supports <b>Random Access</b>, which means elements can be accessed directly using their index, like <b>arr[0]</b> for 1st element, <b>arr[6]</b> for 7th element etc.</p> <p>Hence, accessing elements in an array is <b>fast</b> with a constant time complexity of <b>O(1)</b>.</p> | <p>Linked List supports <b>Sequential Access</b>, which means to access any element/node in a linked list, we have to sequentially traverse the complete linked list, upto that element.</p> <p>To access <b>nth</b> element of a linked list, time complexity is <b>O(n)</b>.</p>               |
| In an array, elements are stored in <b>contiguous memory location</b> or consecutive manner in the memory.                                                                                                                                                                                      | <p>In a linked list, new elements can be stored anywhere in the memory.</p> <p>Address of the memory location allocated to the new element is stored in the previous node of linked list, hence forming a link between the two nodes/elements.</p>                                               |
| <p>In array, <b>Insertion and Deletion</b> operation takes more time, as the memory locations are consecutive and fixed.</p>                                                                                                                                                                    | <p>In case of linked list, a new element is stored at the first free and available memory location, with only a single overhead step of storing the address of memory location in the previous node of linked list.</p> <p>Insertion and Deletion operations are <b>fast</b> in linked list.</p> |
| <p>Memory is allocated as soon as the array is declared, at <b>compile time</b>. It's also known as <b>Static Memory Allocation</b>.</p>                                                                                                                                                        | <p>Memory is allocated at <b>runtime</b>, as and when a new node is added. It's also known as <b>Dynamic Memory Allocation</b>.</p>                                                                                                                                                              |
| <p>In array, each element is independent and can be accessed using it's index value.</p>                                                                                                                                                                                                        | <p>In case of a linked list, each node/element points to the next, previous, or maybe both nodes.</p>                                                                                                                                                                                            |
| <p>Array can <b>single dimensional, two dimensional or multidimensional</b></p>                                                                                                                                                                                                                 | <p>Linked list can be <b>Linear(Singly), Doubly</b> or <b>Circular</b> linked list.</p>                                                                                                                                                                                                          |
| <p>Size of the array must be specified at time of array declaration.</p>                                                                                                                                                                                                                        | <p>Size of a Linked list is variable. It grows at runtime, as more nodes are added to it.</p>                                                                                                                                                                                                    |
| <p>Array gets memory allocated in the <b>Stack</b> section.</p>                                                                                                                                                                                                                                 | <p>Whereas, linked list gets memory allocated in <b>Heap</b> section.</p>                                                                                                                                                                                                                        |

## APPLICATIONS OF STACK

1. Infix to Postfix Conversion
2. Postfix Evaluation
3. Balancing Symbols
4. Nested Functions
5. Tower of Hanoi

## INFIX TO POSTFIX CONVERSION

- Expression conversion is the most important application of stacks
- Given an infix expression can be converted to both prefix and postfix notations
- Based on the Computer Architecture either Infix to Postfix or Infix to Prefix conversion is followed

### What is Infix, Postfix & Prefix?

- **Infix Expression :** The operator appears in-between every pair of operands. **operand1 operator operand2 (a+b)**
- **Postfix expression:** The operator appears in the expression after the operands. **operand1 operand2 operator (ab+)**
- **Prefix expression:** The operator appears in the expression before the operands. **operator operand1 operand2 (+ab)**

| Infix                   | Prefix                | Postfix               |
|-------------------------|-----------------------|-----------------------|
| (A + B) / D             | / + A B D             | A B + D /             |
| (A + B) / (D + E)       | / + A B + D E         | A B + D E + /         |
| (A - B / C + E)/(A + B) | / + - A / B C E + A B | A B C / - E + A B + / |
| B ^ 2 - 4 * A * C       | - ^ B 2 * * 4 A C     | B 2 ^ 4 A * C * -     |

### Why postfix representation of the expression?

- Infix expressions are readable and solvable by humans because of easily distinguishable order of operators, but compiler doesn't have integrated order of operators.
- The compiler scans the expression either from left to right or from right to left.
- Consider the below expression: a op1 b op2 c op3 d  
If op1 = +, op2 = \*, op3 = +

$$a + b * c + d$$

- The compiler first scans the expression to evaluate the expression  $b * c$ , then again scan the expression to add  $a$  to it. The result is then added to  $d$  after another scan.
- The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.
- The corresponding expression in postfix form is:  $abc^*+d+$ .
- The postfix expressions can be evaluated easily in a single scan using a stack.

## **ALGORITHM**

Step 1 : Scan the Infix Expression from left to right.

Step 2 : If the scanned character is an operand, append it with final Infix to Postfix string.

Step 3 : Else,

Step 3.1 : If the precedence order of the scanned(incoming) operator is greater than the precedence order of the operator in the stack (or the stack is empty or the stack contains a '(' ), push it on stack.

Step 3.2 : Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

Step 4 : If the scanned character is an '(', push it to the stack.

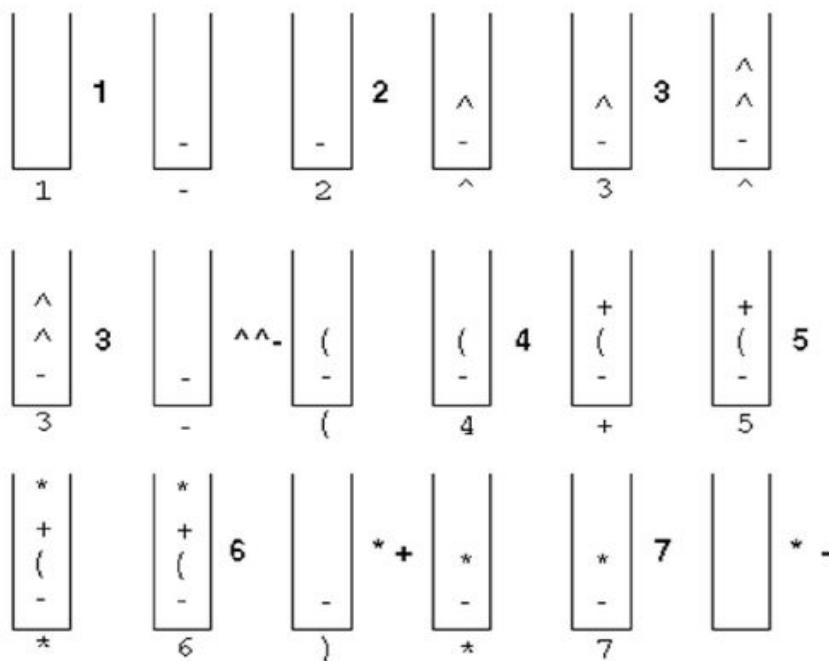
Step 5 : If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.

Step 6 : Repeat steps 2-6 until infix expression is scanned.

Step 7 : Print the output

Step 8 : Pop and output from the stack until it is not empty.

Infix: 1 - 2 ^ 3 ^ 3 - ( 4 + 5 \* 6 ) \* 7



Postfix: 1 2 3 3 ^ ^ - 4 5 6 \* + 7 \* -

## Postfix Expression Evaluation

### Algorithm

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator ( $+$ ,  $-$ ,  $*$ ,  $/$  etc.,), then perform TWO pop operations and store the two popped operands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! perform a pop operation and display the popped value as final result.

Infix Expression  $(5 + 3) * (8 - 2)$   
 Postfix Expression 5 3 + 8 2 - \*

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

| Reading Symbol          | Stack Operations                                                                         | Evaluated Part of Expression                                                                                               |
|-------------------------|------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| Initially               | Stack is Empty                                                                           | Nothing                                                                                                                    |
| 5                       | push(5)                                                                                  | Nothing                                                                                                                    |
| 3                       | push(3)                                                                                  | Nothing                                                                                                                    |
| +                       | value1 = pop(); // 3<br>value2 = pop(); // 5<br>result = value2 + value1<br>push(result) | value1 = pop(); // 3<br>value2 = pop(); // 5<br>result = 5 + 3; // 8<br>Push( 8 )<br><b>(5 + 3)</b>                        |
| 8                       | push(8)                                                                                  | (5 + 3)                                                                                                                    |
| 2                       | push(2)                                                                                  | (5 + 3)                                                                                                                    |
| -                       | value1 = pop(); // 2<br>value2 = pop(); // 8<br>result = 8 - 2; // 6<br>push( 6 )        | value1 = pop(); // 2<br>value2 = pop(); // 8<br>result = 8 - 2; // 6<br>Push( 6 )<br><b>(8 - 2)</b><br>(5 + 3) , (8 - 2)   |
| *                       | value1 = pop();<br>value2 = pop();<br>result = value2 * value1<br>push(result)           | value1 = pop(); // 6<br>value2 = pop(); // 8<br>result = 8 * 6; // 48<br>Push( 48 )<br><b>(6 * 8)</b><br>(5 + 3) * (8 - 2) |
| \$<br>End of Expression | result = pop()                                                                           | Display (result)<br><b>48</b><br>As final result                                                                           |

Infix Expression  $(5 + 3) * (8 - 2) = 48$   
 Postfix Expression 5 3 + 8 2 - \* value is **48**

## BALANCING SYMBOLS

- Stacks can be used to check if the given expression has balanced symbols or not.
- The algorithm is very much useful in compilers.

1. Create a stack
2. while ( end of input is not reached ) {
3. If the character read is not a symbol to be balanced, ignore it.
4. If the character is an opening delimiter like ( , { or [ , PUSH it into the stack.
5. If it is a closing symbol like ) , } , ] , then if the stack is empty report an error, otherwise POP the stack.
6. If the symbol POP-ed is not the corresponding delimiter, report an error.
7. At the end of the input, if the stack is not empty report an error.

Eg: [a+(b\*c)+{(d-e)}]

|       |                        |
|-------|------------------------|
| [     | Push [                 |
| [ (   | Push (                 |
| [     | ) and ( matches, Pop ( |
| [ {   | Push {                 |
| [ { ( | Push (                 |
| [ {   | matches, pop (         |
| [     | Matches, pop {         |
|       | Matches, pop [         |

Thus, parenthesis match here

## Applications of Stack: Function Call and Return

### 2 types of Memory

#### Stack Memory

Stack Memory is a special region of the computer's memory that stores temporary variables created by each function (including the `main()` function). The stack is a "LIFO" (last in, first out) data structure, that is managed and optimized by the CPU quite closely. Every time a function declares a new variable, it is "pushed" onto the stack. Then every time a function exits, **all** of the variables pushed onto the stack by that function, are popped (that is to say, they are deleted). Once a stack variable is freed, that region of memory becomes available for other stack variables.

A key to understanding the stack is the notion that **when a function exits**, all of its variables are popped off of the stack (and hence lost forever). Thus stack variables are **local** in nature. This is related to a concept we saw earlier known as **variable scope**, or local vs global variables.

## Heap Memory

The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU. It is a more free-floating region of memory (and is larger). To allocate memory on the heap, you must use `malloc()` or `calloc()`, which are built-in C functions. Once you have allocated memory on the heap, you are responsible for using `free()` to deallocate that memory once you don't need it any more. If you fail to do this, your program will have what is known as a **memory leak**. That is, memory on the heap will still be set aside (and won't be available to other processes). As we will see in the debugging section, there is a tool called `valgrind` that can help you detect memory leaks.

Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.

## Nested Function Calls

Consider the code snippet below:

```
main()
{
    ...
    foo();
    bar();
}

foo()
{
    ...
    bar();
    ...
}

bar()
{
    ...
}
```

The growth and shrinking of stack as the program executes is given below:

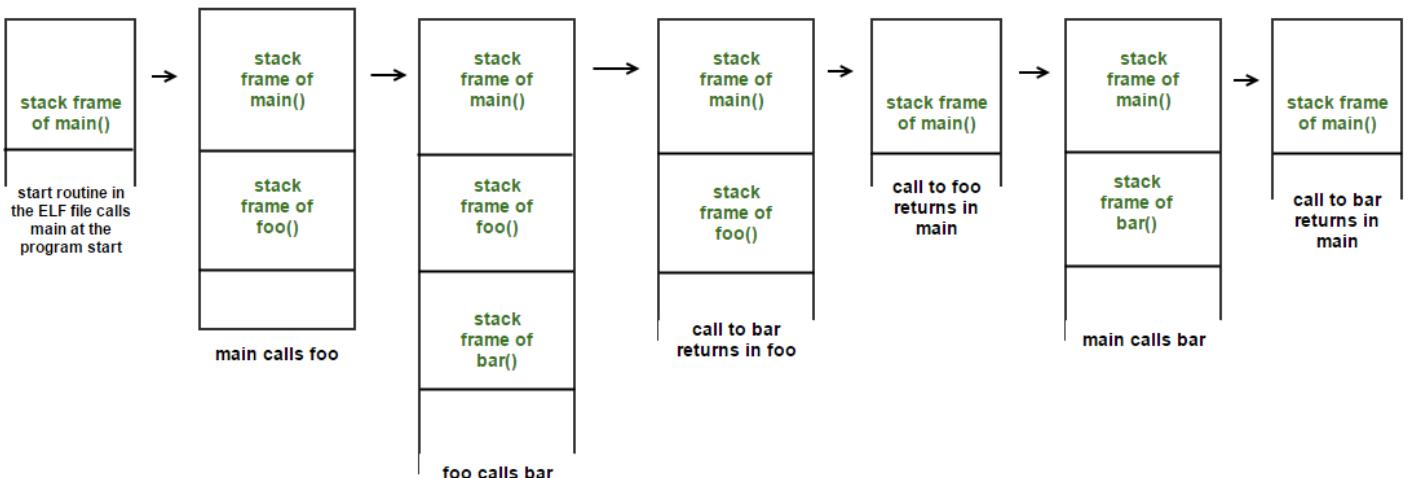


Image Source: <https://loonytek.com/2015/04/28/call-stack-internals-part-1/>

- Each call to a function pushes the function's activation record (or stack frame) into the stack memory
- Activation record mainly consists of: local variables of the function and function parameters
- When the function finishes execution and returns, its activation record is popped

## Recursion

- A recursive function is a function that calls itself during its execution.
- Each call to a recursive function pushes a new activation record (or stack frame) into the stack memory.
- Each new activation record will consist of freshly created local variables and parameters for that specific function call.
- So, even though the same function is called multiple times, a new memory space will be created for each call in the stack memory.

## Handling of Recursion by Stack Memory

When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to calling function and different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

Example:

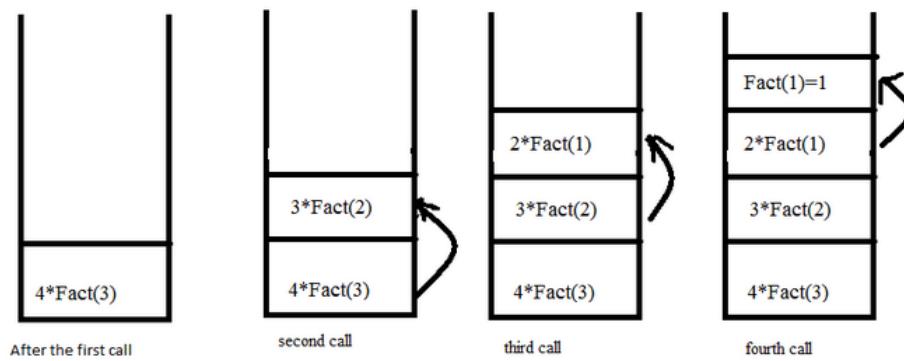
```

int fact(int n)
{
    if (n==1)
        return 1;
    else
        return (n*fact(n-1));
}

fact(4);

```

When function call happens previous variables gets stored in stack



Returning values from base case to caller function

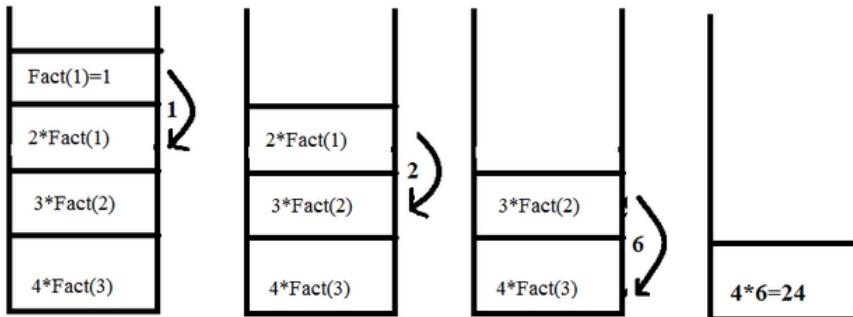


Image Source: <https://stackoverflow.com/questions/19865503/can-recursion-be-named-as-a-simple-function-call>

## Applications of Recursion: Towers of Hanoi

### Towers of Hanoi Problem:

There are 3 pegs and  $n$  disks. All the  $n$  disks are stacked in 1 peg in the order of their sizes with the largest disk at the bottom and smallest on top. All the disks have to be transferred to another peg.

### Rules for transfer:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- No disk may be placed on top of a smaller disk.

### Basic Solution to the problem:

**Step 1** – Move n-1 disks from **source** to **aux**

**Step 2** – Move n<sup>th</sup> disk from **source** to **dest**

**Step 3** – Move n-1 disks from **aux** to **dest**

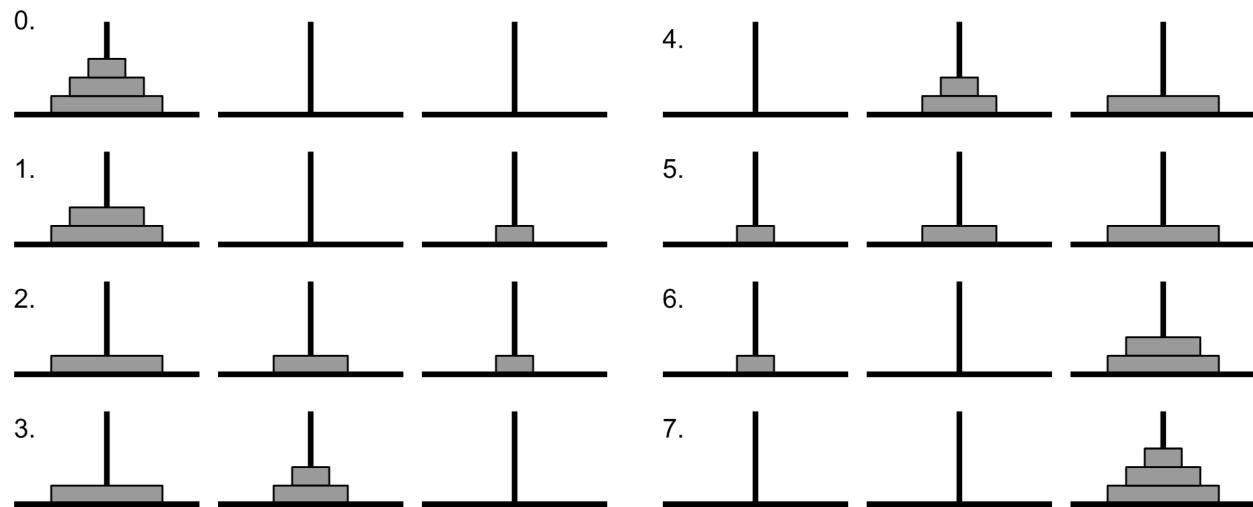


Image Source: <https://www.includehelp.com/data-structure-tutorial/tower-of-hanoi-using-recursion.aspx>

### Towers of Hanoi – Recursive Solution

```
/* N = Number of disks
   Beg, Aux, End are the pegs */
Tower(N, Beg, Aux, End)
Begin
  if N = 1 then
    Print: Beg --> End;
  else
    Call Tower(N-1, Beg, End, Aux);
    Print: Beg --> End;
    Call Tower(N-1, Aux, Beg, End);
  endif
End
```

## The Queue ADT

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing elements are performed at two different positions. The insertion is performed at one end and deletion is performed at another end. In a queue data structure, the insertion operation is performed at a position which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front**'. In queue data structure, the insertion and deletion operations are performed based on **FIFO** (**First In First Out**) principle.

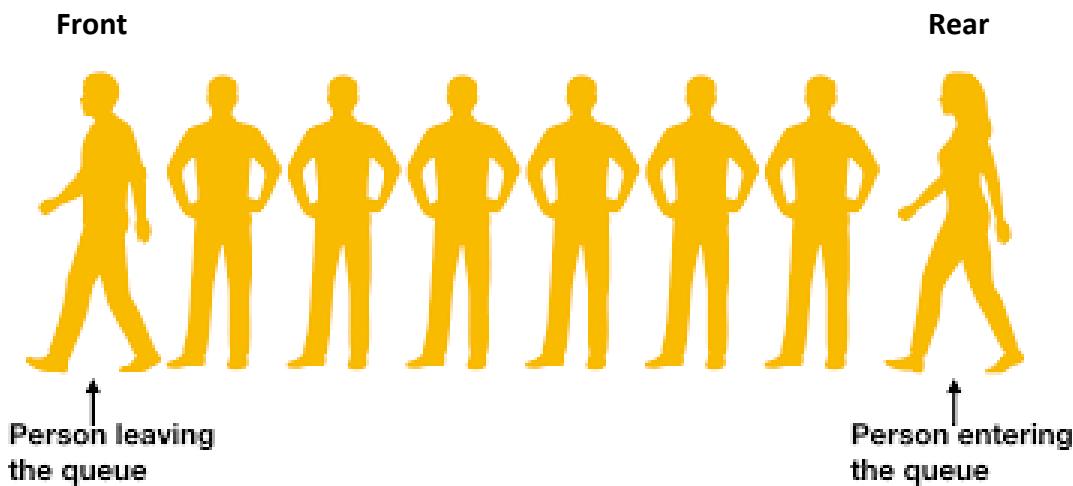


Image Source: <https://www.codesdope.com/course/data-structures-queue/>

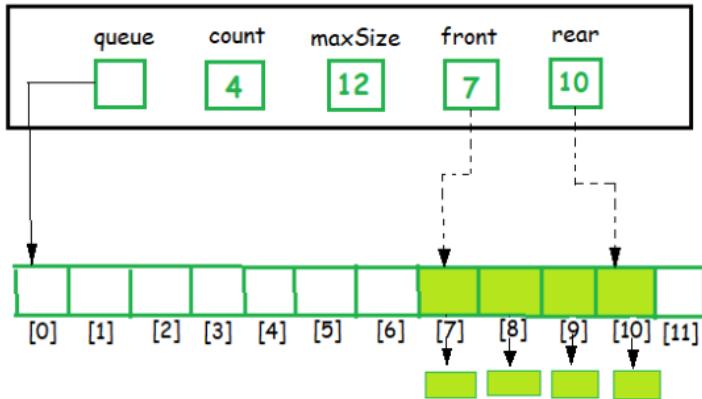
In a queue data structure, the insertion operation is performed using a function called "**enQueue()**" and deletion operation is performed using a function called "**deQueue()**".

### Common Queue Operations

- **enqueue()** – Insert an element at the end of the queue.
- **dequeue()** – Remove and return the first element of the queue, if the queue is not empty.
- **peek()** – Return the element of the queue without removing it, if the queue is not empty.
- **size()** – Return the number of elements in the queue.
- **isEmpty()** – Return true if the queue is empty, otherwise return false.
- **isFull()** – Return true if the queue is full, otherwise return false.

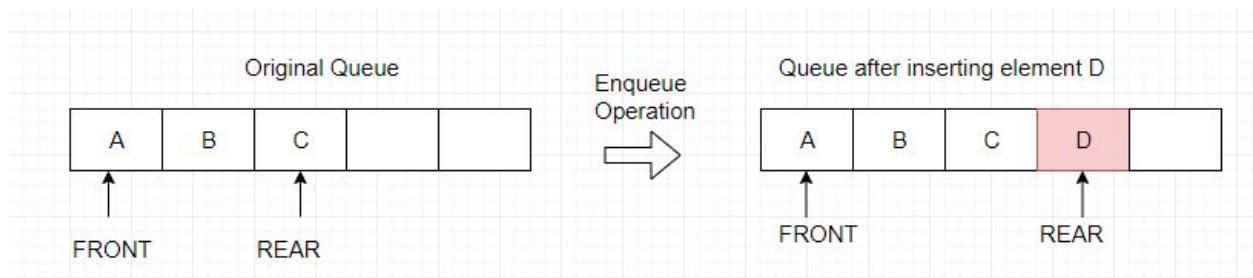
## Array Implementation of Queue

In queue, insertion and deletion happen at the opposite ends.

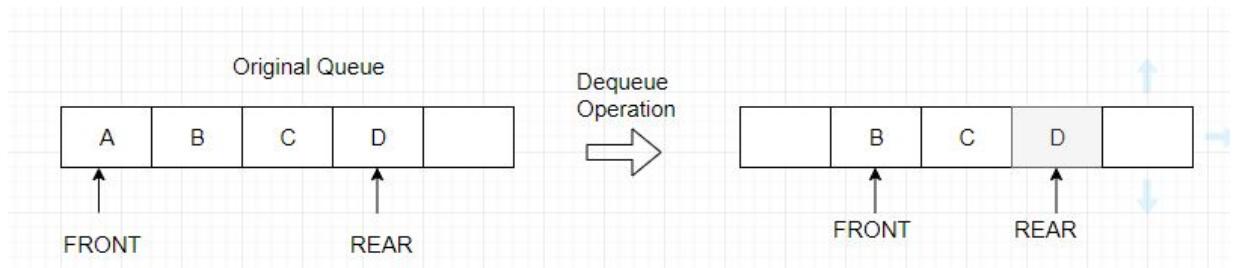


To implement a queue using array, create an array *arr* of size *n* and take two variables *front* and *rear* both of which will be initialized to 0 which means the queue is currently empty. Element *rear* is the index upto which the elements are stored in the array and *front* is the index of the first element of the array. Now, some of the implementation of queue operations are as follows:

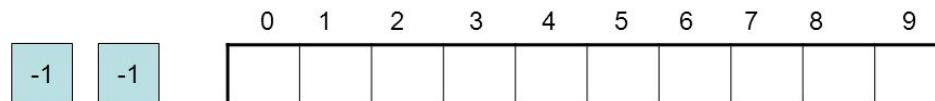
1. **Enqueue:** Addition of an element to the queue. Adding an element will be performed after checking whether the queue is full or not. If *rear* < *n* which indicates that the array is not full then store the element at *arr[rear]* and increment *rear* by 1 but if *rear* == *n* then it is said to be an Overflow condition as the array is full.



2. **Dequeue:** Removal of an element from the queue. An element can only be deleted when there is at least an element to delete i.e. *rear* > 0. Now, element at *arr[front]* can be deleted but all the remaining elements have to shifted to the left by one position in order for the dequeue operation to delete the second element from the left on another dequeue operation.

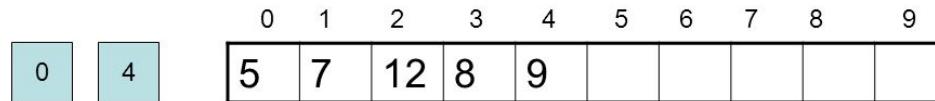


3. **Front:** Get the front element from the queue i.e. *arr[front]* if queue is not empty.
4. **Display:** Print all element of the queue. If the queue is non-empty, traverse and print all the elements from index *front* to *rear*.



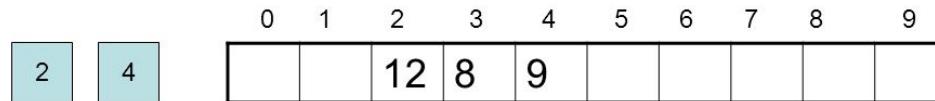
front rear

Representation of queue in memory



front rear

Representation of queue in memory



front rear

Representation of queue in memory

Image Source: <https://slideplayer.com/slide/4151401/>

## Enqueue Operation:

```
void Enqueue(int data)
{
    // check queue is full or not
    if (capacity == rear) {
        printf("\nQueue is full\n");
        return;
    }

    // insert element at the rear
    else {
        queue[rear] = data;
        rear++;
    }
    return;
}
```

## Dequeue Operation:

```
void Dequeue()
```

```

{
    // if queue is empty
    if (front == rear) {
        printf("\nQueue is empty\n");
        return;
    }

    // shift all the elements from index 2 till rear
    // to the left by one
    else {
        for (int i = 0; i < rear - 1; i++) {
            queue[i] = queue[i + 1];
        }

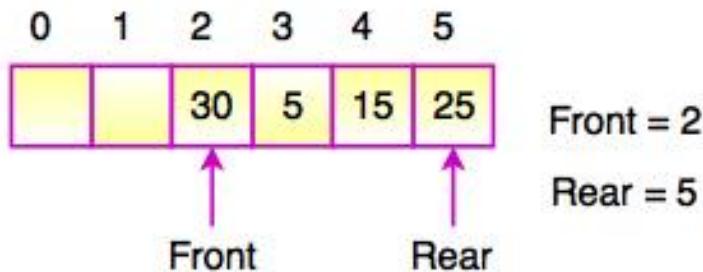
        // decrement rear
        rear--;
    }
    return;
}

```

## Disadvantage of Array Implementation of Queue

Although, the technique of creating a queue is easy, but there are some drawbacks of using this technique to implement a queue.

- **Memory wastage :** The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.



<https://www.tutorialride.com/data-structures/queue-in-data-structure.htm>

- **Deciding the array size**

One of the most common problem with array implementation is the size of the array which requires to be declared in advance. Due to the fact that, the queue can be extended at runtime depending upon the problem, the extension in the array size is a time taking process and almost impossible to be performed at runtime since a lot of reallocations take place. Due to this reason, we can declare the array large enough so that we can store queue elements as enough as possible

but the main problem with this declaration is that, most of the array slots (nearly half) can never be reused. It will again lead to memory wastage.

## Linked List Implementation of Queue

Due to the drawbacks discussed in the previous section, the array implementation cannot be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

The storage requirement of linked representation of a queue with n elements is  $O(n)$  while the time requirement for operations is  $O(1)$ .

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

### Node Creation:

```
struct Node
{
    int data;
    struct Node *next;
};
```

### Enqueue Operation:

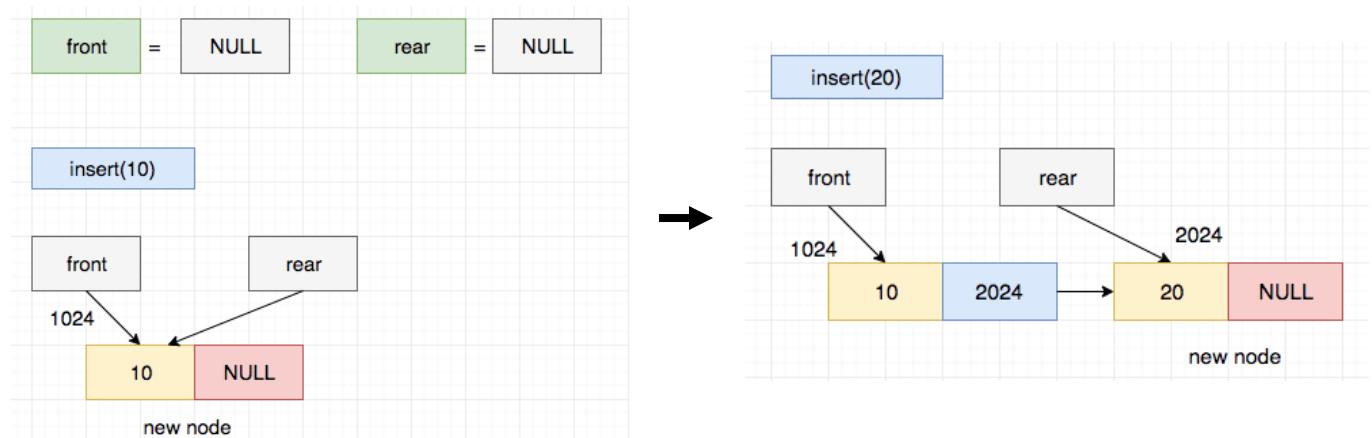


Image Source: <https://www.log2base2.com/data-structures/queue/queue-using-linked-list.html>

```
void enqueue(int value){
    struct Node *newNode;
```

```

newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = value;
newNode -> next = NULL;
if(front == NULL)
    front = rear = newNode;
else{
    rear -> next = newNode;
    rear = newNode;
}
}

```

Dequeue Operation:

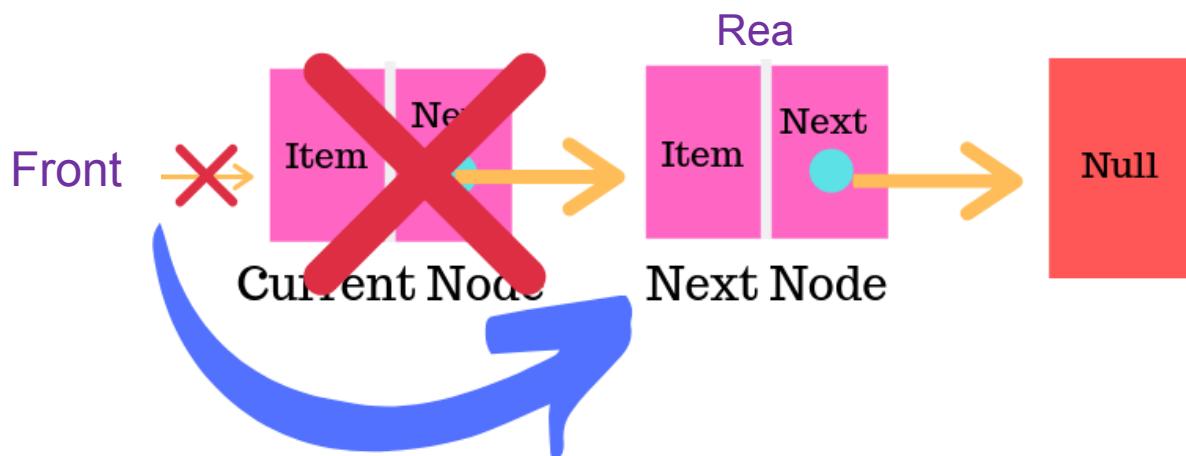


Image Source: <https://learnersbucket.com/tutorials/data-structures/implement-queue-using-linked-list/>

```

void dequeue ()
{
    if(front == NULL)
        printf("\nQueue is Empty!!!\n");
    else{
        struct Node *temp = front;
        front = front -> next;
        free(temp);
    }
}

```

## CIRCULAR QUEUE

Circular Queue is a linear data structure in which first position of the queue is connected with last position of the queue.

In other words, The queue is considered as a circular queue when the positions 0 and MAX-1 are adjacent. It is also referred as RING BUFFER.

## Principle Used

FIFO (First In First Out) (First entered element is processed first) is used for performing the operation.

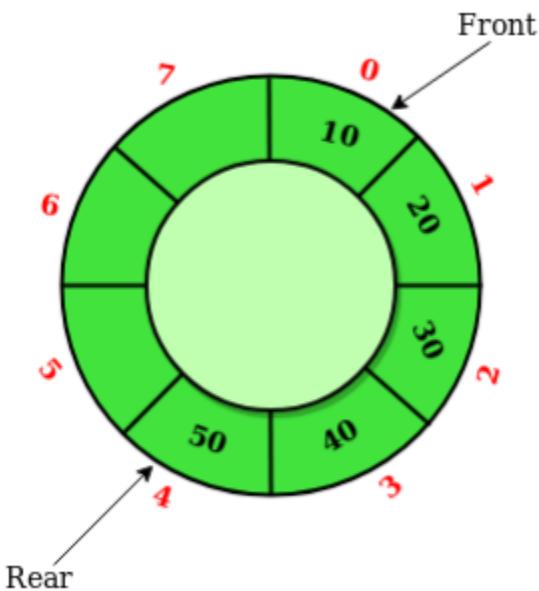
## Operations Involved

- Enqueue- is the process of inserting an element in REAR end
- Dequeue – is the process of removing element from FRONT end

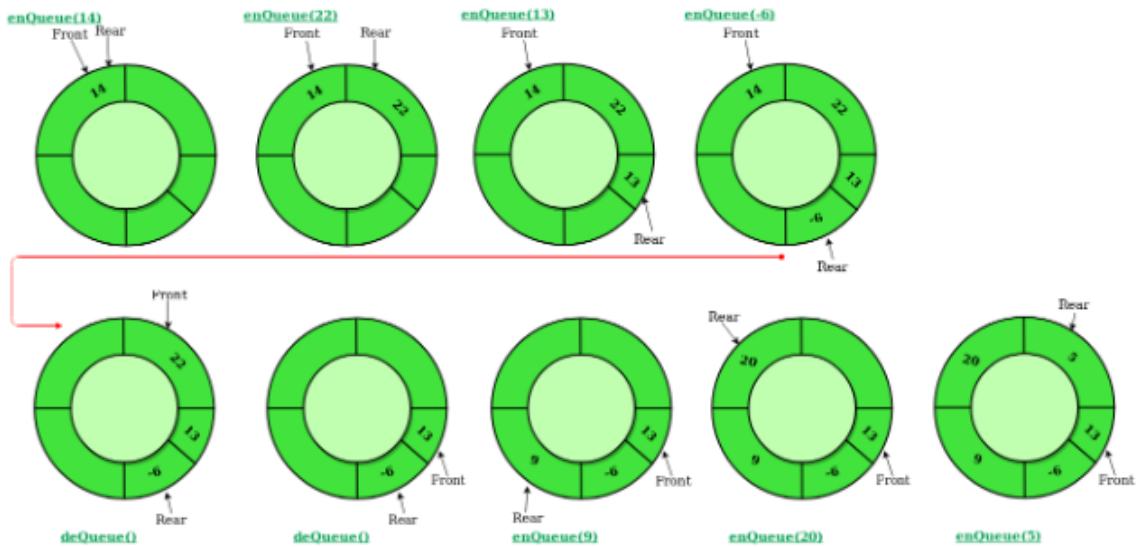
## Variables Used

- MAX- Number of entries in the array
- Front – is the index of front queue entry in an array (Get the front item from queue)
- Rear – is the index of rear queue entry in an array.(Get the last item from queue)

## Concepts of Circular Queue



## Illustration of Enqueue and Dequeue



## Conditions used in Circular Queue

- Front must point to the first element.
- The queue will be empty if Front = Rear.
- When a new element is added the queue is incremented by value one(Rear = Rear + 1).
- When an element is deleted from the queue the front is incremented by one (Front = Front + 1).

## Steps Involved in Enqueue

Check whether queue is Full or not by using the following condition

```
((rear == SIZE-1 && front == 0) || (rear == front-1))
```

If queue is full, display the queue is full else we can insert an element by incrementing rear pointer.

## Steps Involved in Dequeue

1. Check whether queue is Empty or not by using the following condition

```
if(front == -1).
```

2. If it is empty then display Queue is empty, else we can delete the element.

## **Difference between Queue and Circular Queue**

- In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.

The unused memory locations in the case of ordinary queues can be utilized in circular queues.

## **Circular Queue Example-ATM**

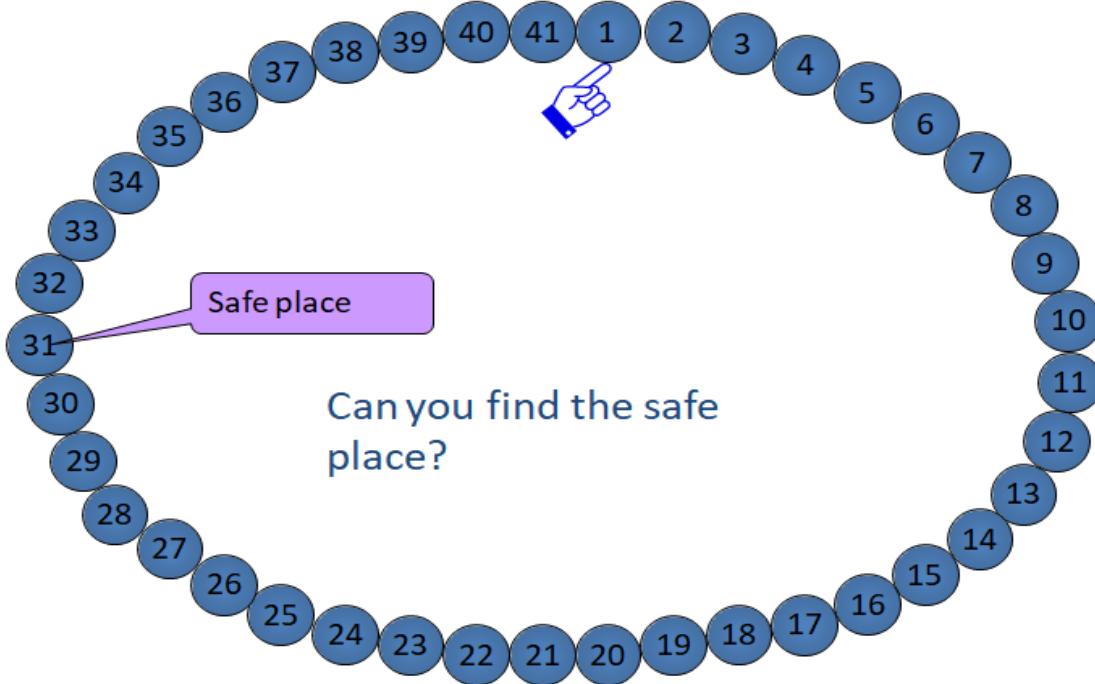
ATM is the best example for the circular queue. It does the following using circular queue

1. ATM sends request over private network to central server
2. Each request takes some amount of time to process.
3. More request may arrive while one is being processed.
4. Server stores requests in queue if they arrive while it is busy.
5. Queue processing time is negligible compared to request processing time.

## **Josephus problem**

Flavius Josephus is a Jewish historian living in the 1st century. According to his account, he and his 40 comrade soldiers were trapped in a cave, surrounded by Romans. They chose suicide over capture and decided that they would form a circle and start killing themselves using a step of three. As Josephus did not want to die, he was able to find the safe place, and stayed alive with his comrade, later joining the Romans who captured them.

## **Can you find the safe place?**



### The first algorithm: Simulation

- We can find  $f(n)$  using simulation.
  - Simulation is a process to imitate the real objects, states of affairs, or process.
  - We do not need to “kill” anyone to find  $f(n)$ .
- The simulation needs

- (1) a **model** to represents “n people in a circle”
- (2) a way to **simulate** “kill every 2<sup>nd</sup> person”
- (3) knowing when to stop

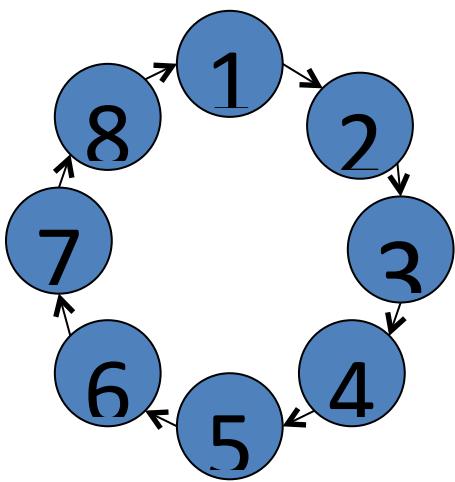
### Model n people in a circle

- We can use “data structure” to model it.
- This is called a “circular linked list”.
  - Each node is of some “**struct**” data type
  - Each link is a “**pointer**”

```
struct node {
```

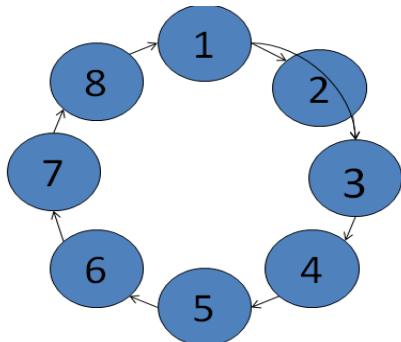
```
    int ID;
```

```
    struct node *next;  
}
```



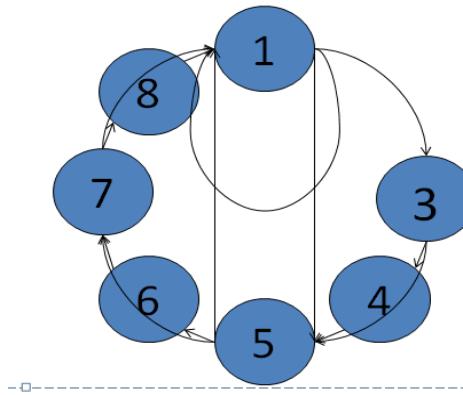
#### Kill every 2<sup>nd</sup> person

- Remove every 2<sup>nd</sup> node in the circular linked list.
  - You need to maintain the circular linked structure after removing node 2
  - The process can continue until ...



#### Knowing when to stop

- Stop when there is only one node left
  - How to know that?
  - When the \*next is pointing to itself
  - It's ID is  $f(n)$   $f(8) = 1$



## Priority Queue

- Priority Queue is an extension of a queue with following properties.
- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

## Priority Queue Example

**Example:**

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| X | Y | A | M | B | C | N | O | P | Z |
| 1 | 1 | 2 | 3 | 2 | 2 | 3 | 3 | 3 | 1 |

## Operations Involved

- **insert(item, priority):** Inserts an item with given priority.
- **getHighestPriority():** Returns the highest priority item.
- **pull highest priority element:** remove the element from the queue that has the *highest priority*, and return it.

## Implementation of priority queue

- Circular array

- Multi-queue implementation
- Double Link List
- Heap Tree

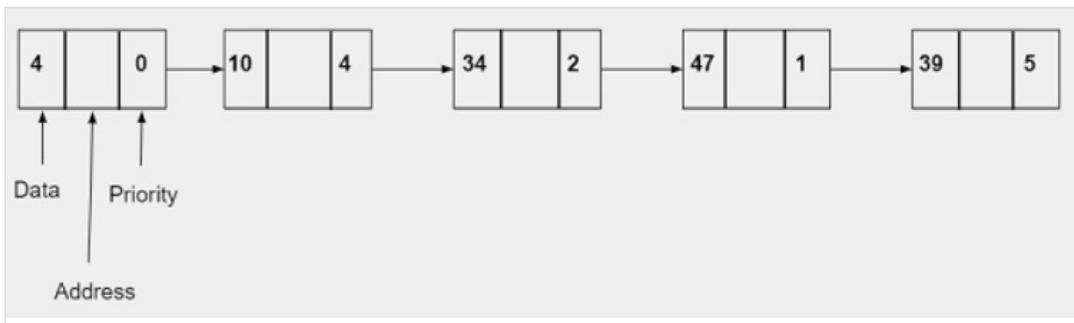
### **Node of linked list in priority queue**

It comprises of three parts

- **Data** – It will store the integer value.
- **Address** – It will store the address of a next node
- **Priority** –It will store the priority which is an integer value. It can range from 0-10 where 0 represents the highest priority and 10 represents the lowest priority.

Example

### **Input**



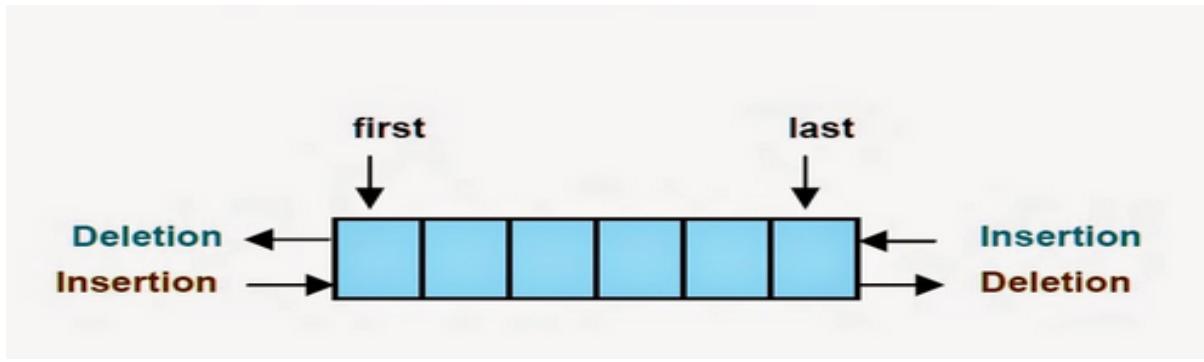
### **Output**



### **Double Ended Queue**

- Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**).

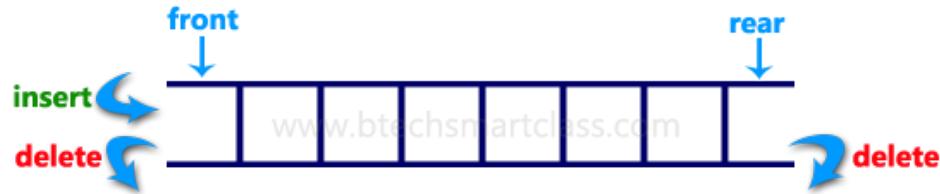
### **Double Ended Queue**



Double Ended Queue can be represented in TWO ways

### **Input Restricted Double Ended Queue**

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



### **Output Restricted Double Ended Queue**

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends



## **References**

1. [https://www.tutorialspoint.com/data\\_structures\\_algorithms/stack\\_algorithm.htm](https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm)
2. <https://www.codesdope.com/course/data-structures-stacks/>
3. <http://www.firmcodes.com/write-a-c-program-to-implement-a-stack-using-an-array-and-linked-list/>
4. [http://www.btechsmartclass.com/data\\_structures/stack-using-linked-list.html](http://www.btechsmartclass.com/data_structures/stack-using-linked-list.html)
5. <http://www.exploredatabase.com/2018/01/stack-abstract-data-type-data-structure.html>
6. <https://www.geeksforgeeks.org/stack-set-2-infix-to-postfix/>
7. <https://www.hackerearth.com/practice/notes/stacks-and-queues/>
8. <https://github.com/niinpatel/Parenthesis-Matching-with-Reduce-Method>
9. <https://www.studytonight.com/data-structures/stack-data-structure>
10. <https://www.programming9.com/programs/c-programs/230-c-program-to-convert-infix-to-postfix-expression-using-stack>
11. Data Structures, Seymour Lipschutz, Schaum's Outline, 2014.
12. [https://gribblelab.org/CBootCamp/7\\_Memory\\_Stack\\_vs\\_Heap.html](https://gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html)
13. <https://www.geeksforgeeks.org/recursion/>
14. <https://www.javatpoint.com/linked-list-implementation-of-queue>
15. Book Reference for Circular queue: Referred from Seymour Lipschutz, Data Structures with C

## UNIT IV

### TREES

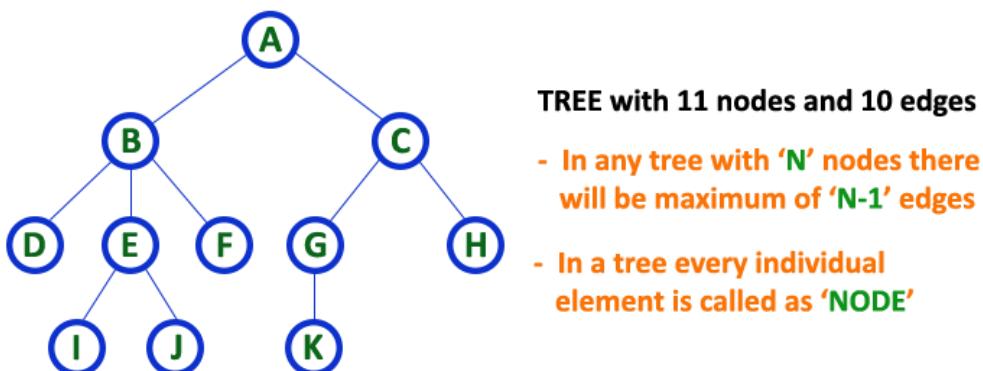
General trees – Terminology – Representation of trees – Tree traversal- Binary tree – Representation – Expression tree – Binary tree traversal - Threaded Binary Tree - Binary Search Tree – Construction - Searching - Binary Search Tree- Insertion – Deletion - AVL trees – Rotation AVL trees – Insertion – Deletion - B-Trees - Splay trees - Red-Black Trees

---

## Trees

- A In linear data structure, data is organized in sequential order and in non-linear data structure, data is organized in random order. Tree is a very popular data structure used in wide range of applications. A tree data structure can be defined as follows...
- Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.
- A tree data structure can also be defined as follows...
- Tree data structure is a collection of data (Node) which is organized in hierarchical structure and this is a recursive definition
- In tree data structure, every individual element is called as Node. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure.
- In a tree data structure, if we have N number of nodes then we can have a maximum of  $N-1$  number of links.

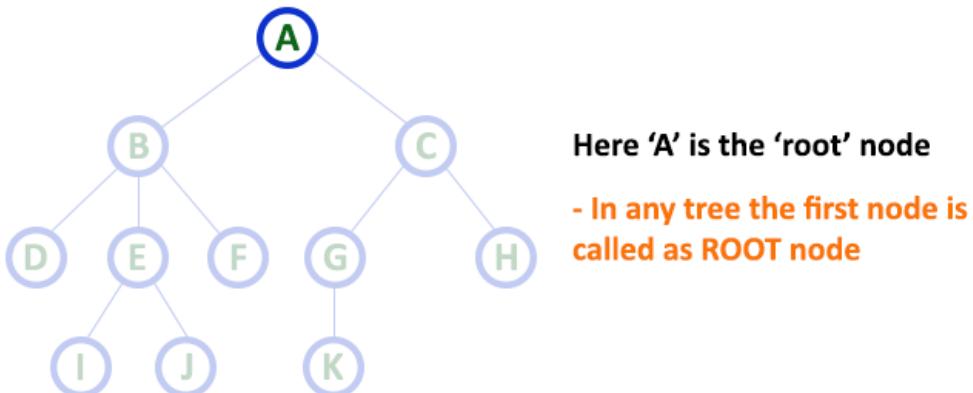
### Example



## Terminology

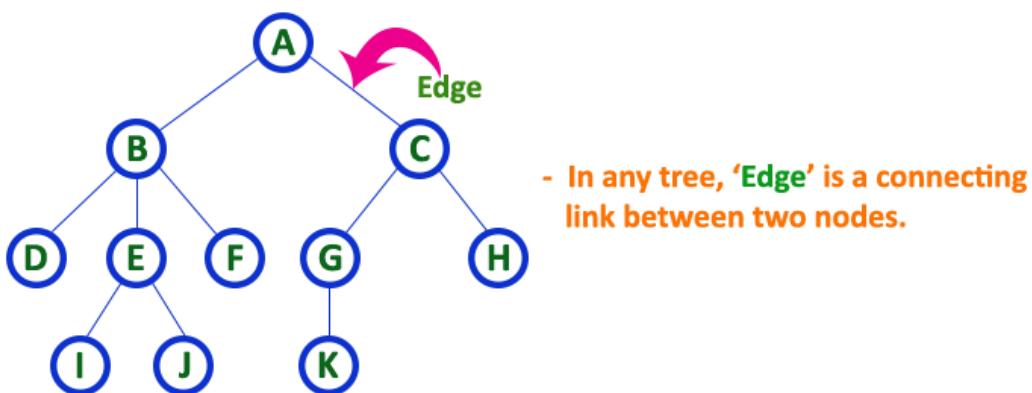
In a tree data structure, we use the following terminology...

### 1. Root



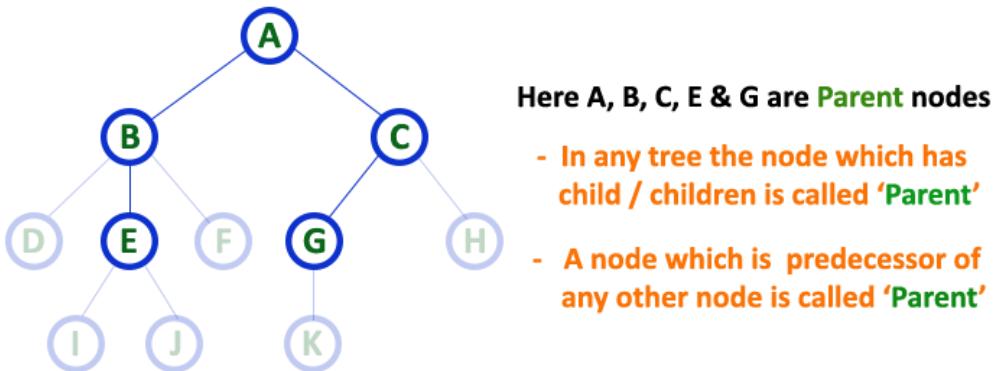
In a tree data structure, the first node is called as Root Node. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

### 2. Edge



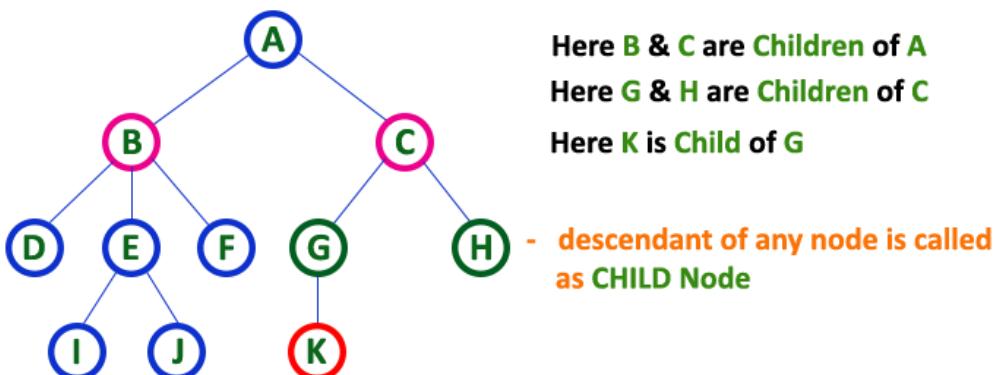
In a tree data structure, the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.

### 3. Parent



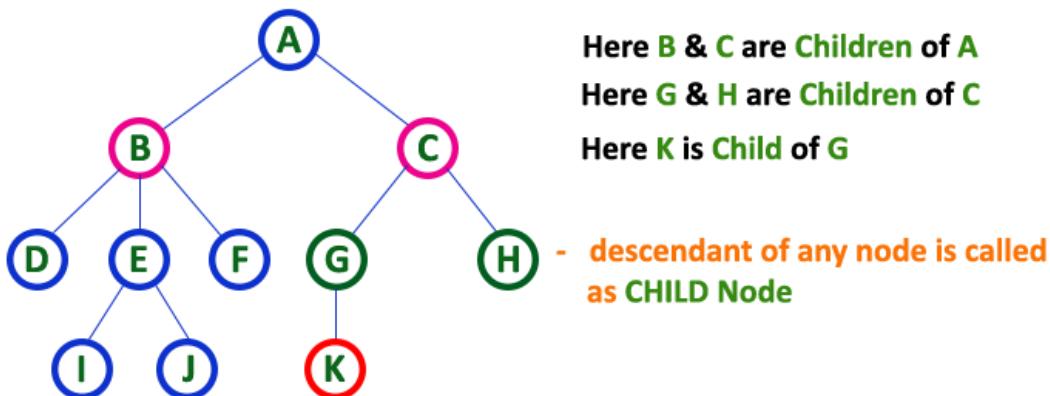
In a tree data structure, the node which is predecessor of any node is called as PARENT NODE. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "The node which has child / children".

### 4. Child



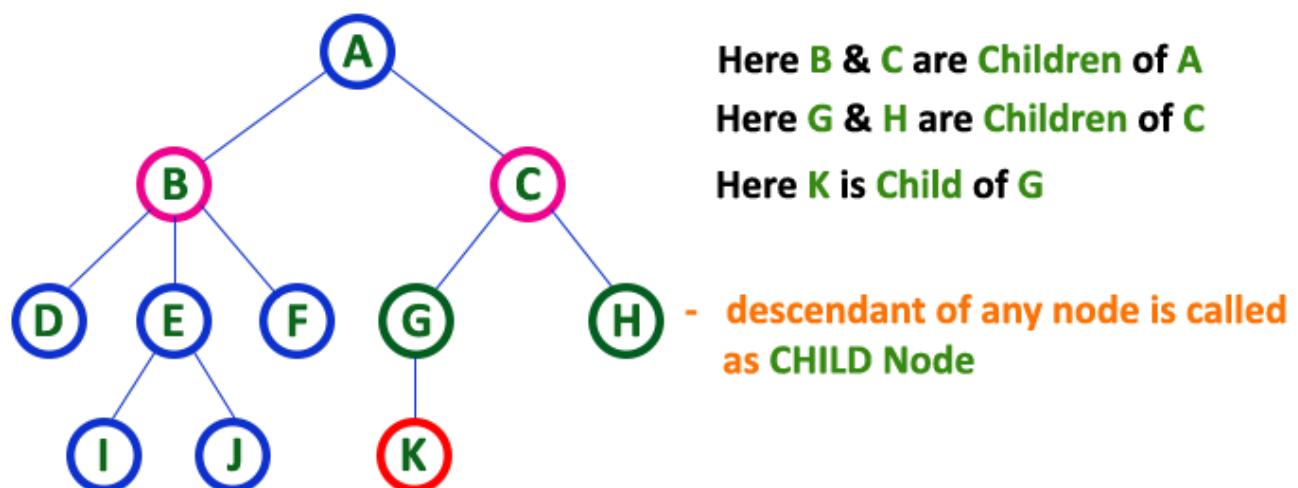
In a tree data structure, the node which is descendant of any node is called as CHILD Node. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.

## 5. Siblings



In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with same parent are called as **Sibling nodes**.

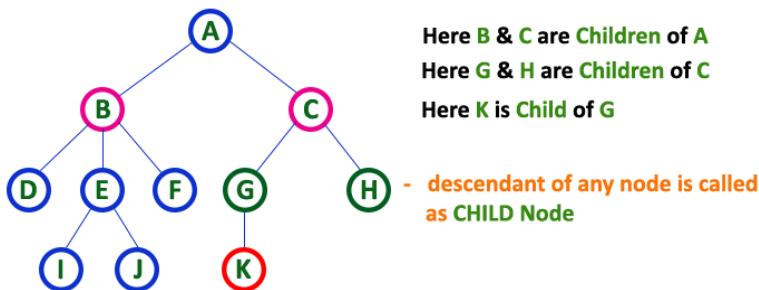
## 6. Leaf



In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

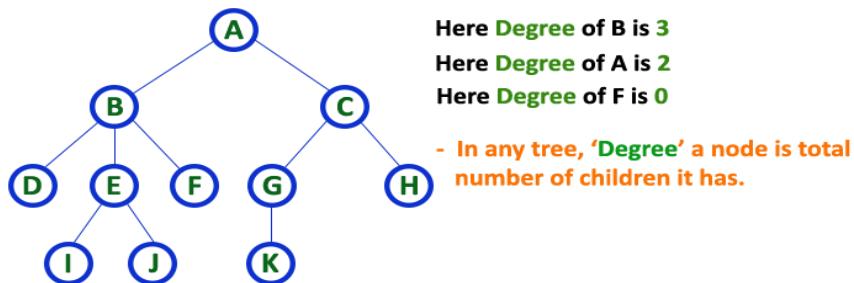
In a tree data structure, the leaf nodes are also called as External Nodes. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node.

## 7. Internal Nodes



In a tree data structure, the node which has atleast one child is called as INTERNAL Node. In simple words, an internal node is a node with atleast one child. In a tree data structure, nodes other than leaf nodes are called as Internal Nodes. The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes.

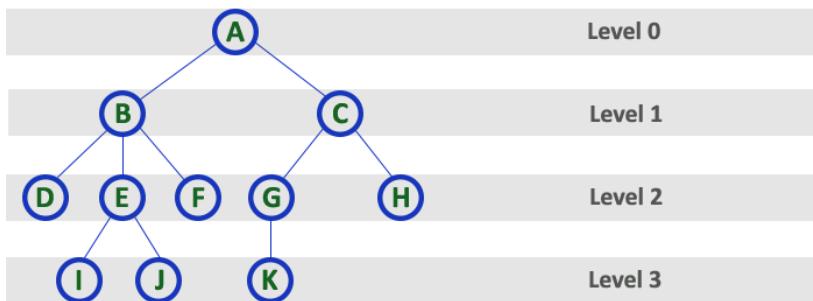
## 8. Degree



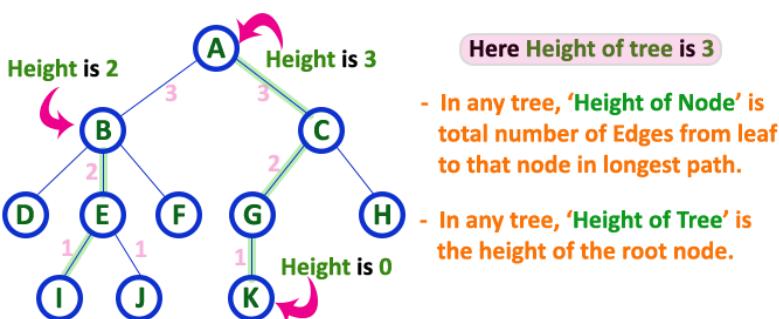
In a tree data structure, the total number of children of a node is called as DEGREE of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'

## 9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).

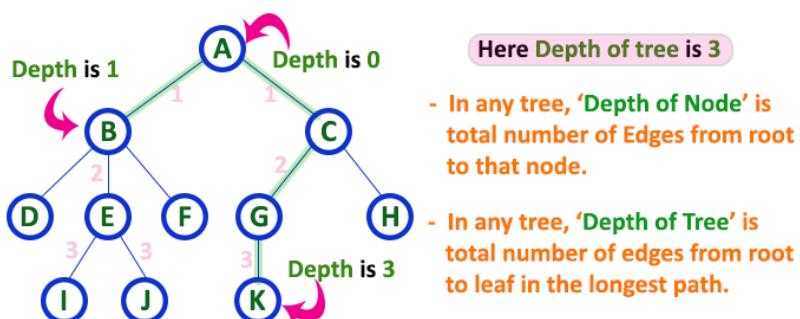


## 10. Height



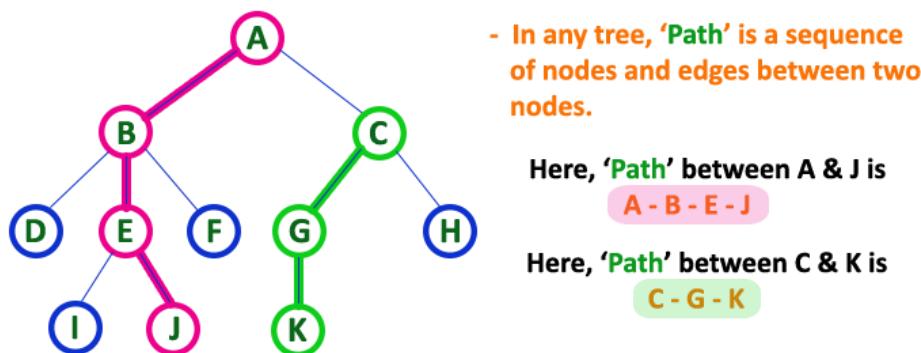
In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'!

## 11. Depth



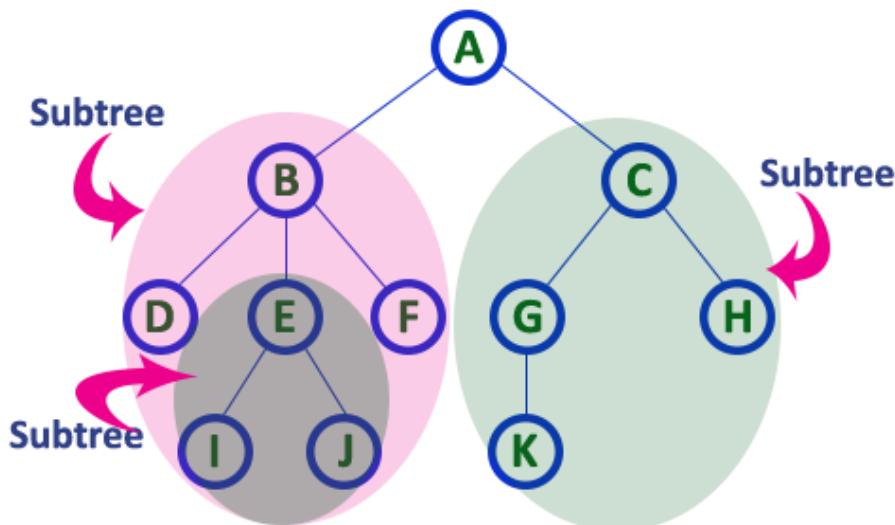
In a tree data structure, the total number of edges from root node to a particular node is called as DEPTH of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be Depth of the tree. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, depth of the root node is '0'!

## 12. Path



In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes. Length of a Path is total number of nodes in that path. In below example the path A - B - E - J has length 4.

## 13. Sub Tree



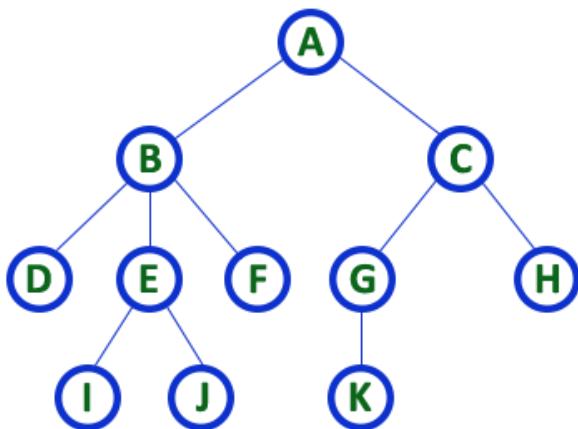
In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.

## Tree Representations

A tree data structure can be represented in two methods. Those methods are as follows...

- List Representation
- Left Child - Right Sibling Representation

Consider the following tree...

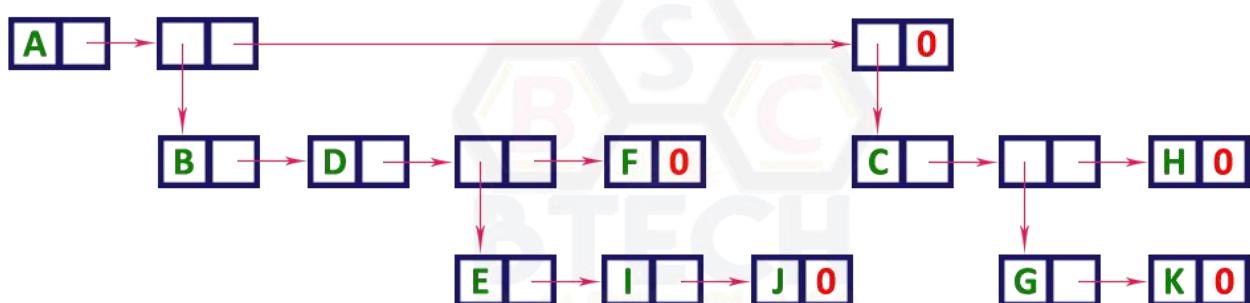


**TREE with 11 nodes and 10 edges**

- In any tree with ' $N$ ' nodes there will be maximum of ' $N-1$ ' edges
- In a tree every individual element is called as 'NODE'

### 1. List Representation

In this representation, we use two types of nodes one for representing the node with data and another for representing only references. We start with a node with data from root node in the tree. Then it is linked to an internal node through a reference node and is linked to any other node directly. This process repeats for all the nodes in the tree. The above tree example can be represented using List representation as follows...



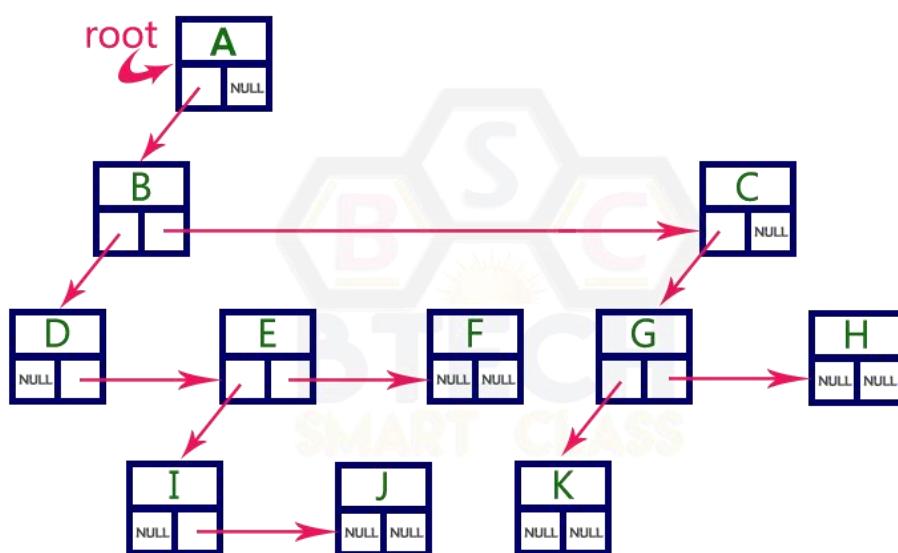
## 2. Left Child - Right Sibling Representation

In this representation, we use list with one type of node which consists of three fields namely Data field, Left child reference field and Right sibling reference field. Data field stores the actual value of a node, left reference field stores the address of the left child and right reference field stores the address of the right sibling node. Graphical representation of that node is as follows...



In this representation, every node's data field stores the actual value of that node. If that node has left child, then left reference field stores the address of that left child node otherwise that field stores NULL. If that node has right sibling then right reference field stores the address of right sibling node otherwise that field stores NULL.

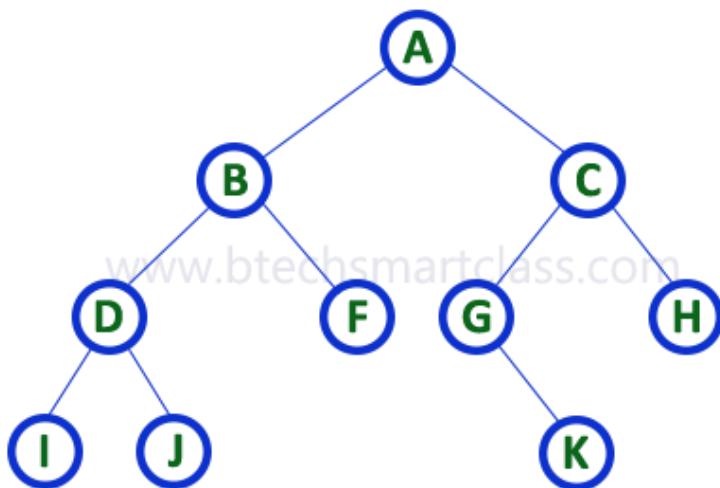
The above tree example can be represented using Left Child - Right Sibling representation as follows...



## Binary Tree

- In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as left child and the other is known as right child.
- **A tree in which every node can have a maximum of two children is called as Binary Tree.**
- In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

Example



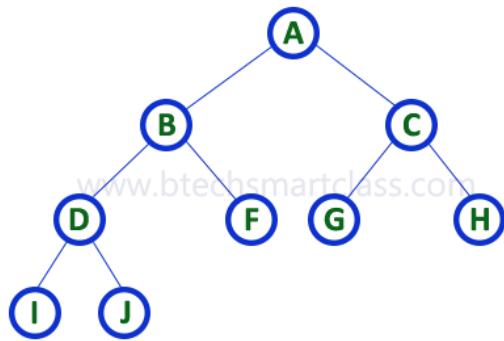
There are different types of binary trees and they are...

### 1. Strictly Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows...

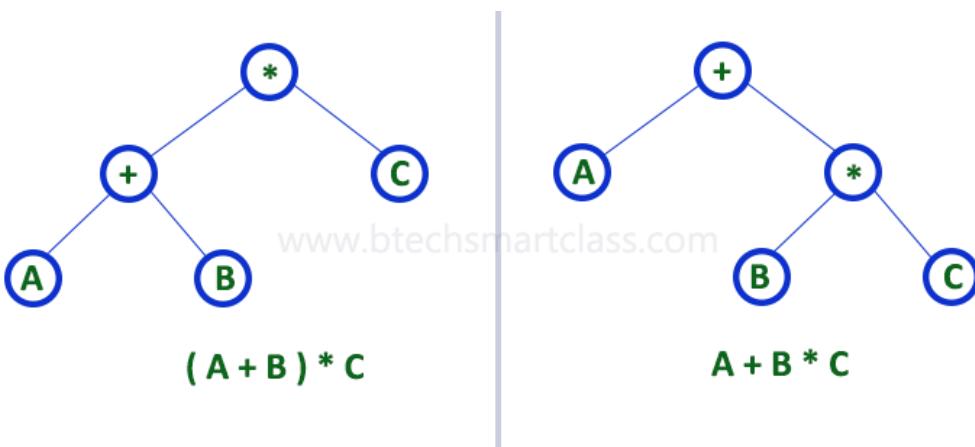
**A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree**

Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree



Strictly binary tree data structure is used to represent mathematical expressions.

Example

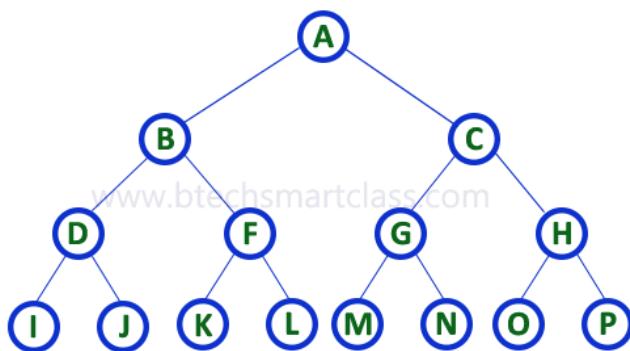


## 2. Complete Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2<sup>level</sup> number of nodes. For example at level 2 there must be  $2^2 = 4$  nodes and at level 3 there must be  $2^3 = 8$  nodes.

**A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.**

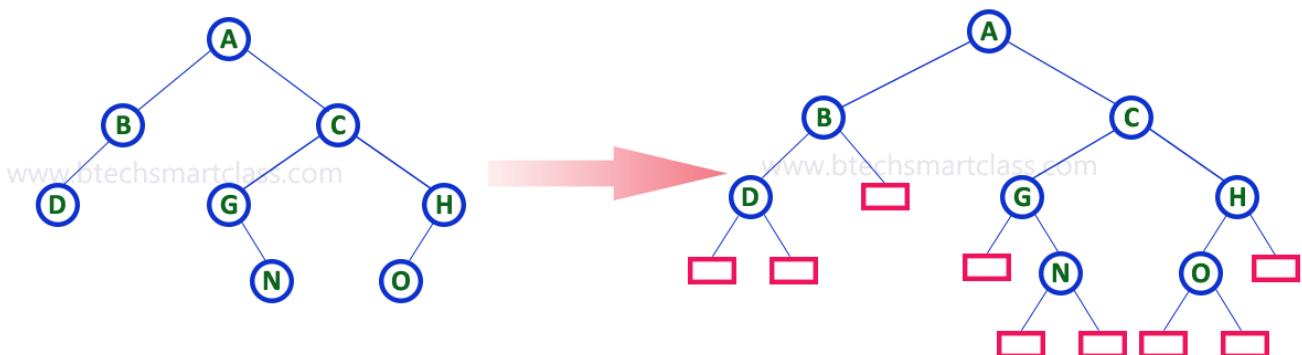
**Complete binary tree is also called as Perfect Binary Tree**



### 3. Extended Binary Tree

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.

The full binary tree obtained by adding dummy nodes to a binary tree is called as **Extended Binary Tree**.



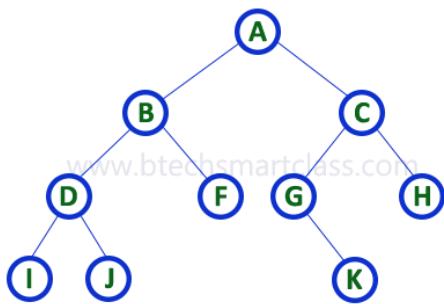
In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes (In pink colour).

## Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

- Array Representation
- Linked List Representation

Consider the following binary tree...



### 1. Array Representation

In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree.

Consider the above example of binary tree and it is represented as follows...

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | B | C | D | F | G | H | I | J | - | - | - | K | - | - | - | - | - | - | - |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of  $2n+1 - 1$ .

It is found that if n is the number or index of a node, then its left child occurs at  $(2n + 1)$ th position & right child at  $(2n + 2)$  th position of the array. If any node does not have any of its child, then null value is stored at the corresponding index of the array.

## 2. Linked List Representation

We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

In this linked list representation, a node has the following structure...

|                           |             |                            |
|---------------------------|-------------|----------------------------|
| <b>Left Child Address</b> | <b>Data</b> | <b>Right Child Address</b> |
|---------------------------|-------------|----------------------------|

**Struct node**

```
{
struct node * lc;
int data;
struct node * rc;
};
```

**Creating TREE**

```
struct node * buildtree(int n);

{
    struct node * temp=NULL;

    if( a[n] != NULL)

    { temp = (struct node *) malloc(sizeof(struct node));

        temp-> lc=buildtree(2n + 1);

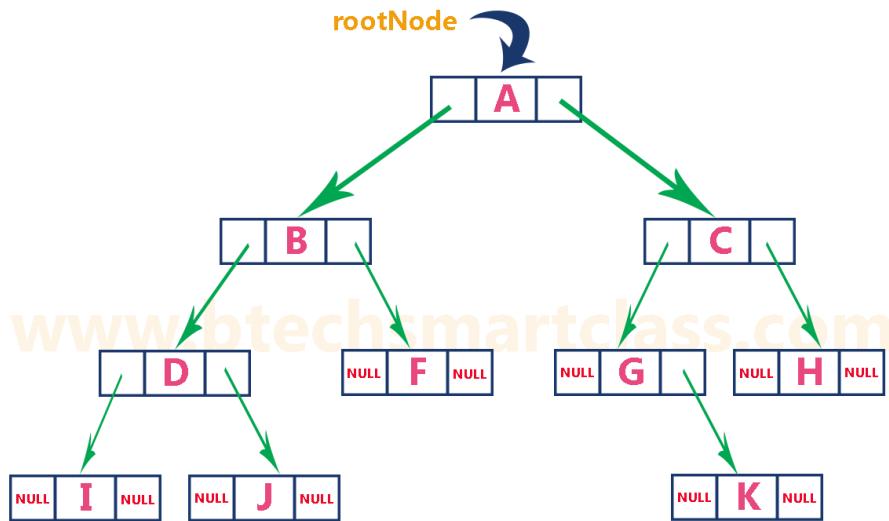
        temp-> data= a[n];

        temp-> rc=buildtree(2n + 2);

    } return temp;

}
```

The above example of binary tree represented using Linked list representation is shown as follows...



## Binary Tree Traversals

In any binary tree displaying order of nodes depends on the traversal method.

**Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.**

There are three types of binary tree traversals.

1. In - Order Traversal
2. Pre - Order Traversal
3. Post - Order Traversal

### In-order Traversal

Until all nodes are traversed –

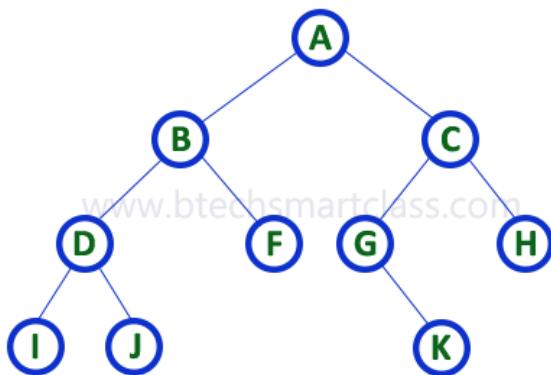
- **Step 1** – Recursively traverse left subtree.
- **Step 2** – Visit root node.
- **Step 3** – Recursively traverse right subtree.

## Algorithm

The algorithm for inorder traversal is as follows.

```
void inorder(struct node * root);
{
if(root != NULL)
{
inorder(roo-> lc);
printf("%d\t",root->data);
inorder(root->rc);
}
}
```

Consider the following binary tree...



In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the left most child. So first we visit 'I' then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is

visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit 'G' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node 'C' and next visit C's right child 'H' which is the right most child in the tree so we stop the process.

That means here we have visited in the order of **I - D - J - B - F - A - G - K - C - H** using In-Order Traversal.

**In-Order Traversal for above example of binary tree is**  
**I - D - J - B - F - A - G - K - C - H**

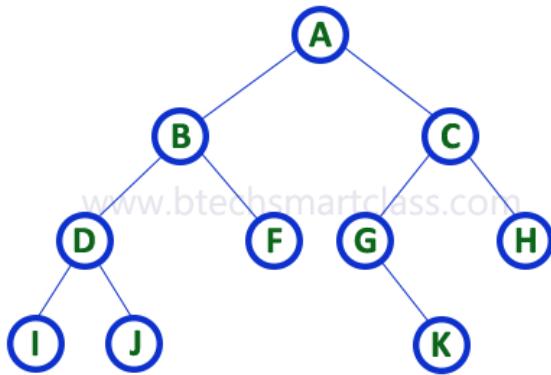
## 2. Pre-Order Traversal

Until all nodes are traversed –

- Step 1 – Visit root node.
- Step 2 – Recursively traverse left subtree.
- Step 3 – Recursively traverse right subtree.

```
void preorder(struct node * root);
{
    if(root != NULL)
    {
        printf("%d\t",root->data);
        preorder(root->lc);
        preorder(root->rc);
    }
}
```

Consider the following binary tree...



In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.

In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the left most child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'H'. With this we have completed node C's root and left parts. Next visit C's right child 'H' which is the right most child in the tree. So we stop the process.

That means here we have visited in the order of **A-B-D-I-J-F-C-G-K-H** using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is  
**A - B - D - I - J - F - C - G - K - H**

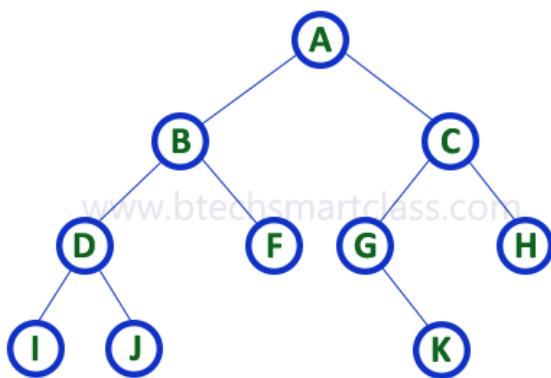
## Post-order Traversal

Until all nodes are traversed –

- Step 1 – Recursively traverse left subtree.
- Step 2 – Recursively traverse right subtree.
- Step 3 – Visit root node.

```
void postorder(struct node * root);
{
    if(root != NULL)
    {
        postorder(root->lc);
        postorder(root->rc);
        printf("%d\t",root->data);
    }
}
```

Consider the following binary tree...



In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using Post-Order Traversal.

**Post-Order Traversal for above example binary tree is**

**I - J - D - F - B - K - G - H - C - A**

## **Threaded Binary Tree**

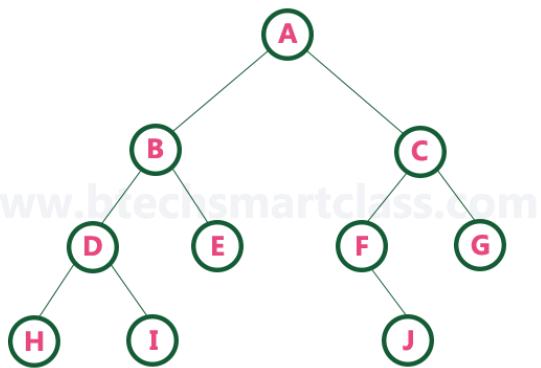
A binary tree is represented using array representation or linked list representation. When a binary tree is represented using linked list representation, if any node is not having a child we use NULL pointer in that position. In any binary tree linked list representation, there are more number of NULL pointer than actual pointers. Generally, in any binary tree linked list representation, if there are  $2N$  number of reference fields, then  $N+1$  number of reference fields are filled with NULL ( $N+1$  are NULL out of  $2N$ ). This NULL pointer does not play any role except indicating there is no link (no child).

A. J. Perlis and C. Thornton have proposed new binary tree called "*Threaded Binary Tree*", which make use of NULL pointer to improve its traversal processes. In threaded binary tree, NULL pointers are replaced by references to other nodes in the tree, called *threads*.

**Threaded Binary Tree is also a binary tree in which all left child pointers that are NULL (in Linked list representation) points to its in-order predecessor, and all right child pointers that are NULL (in Linked list representation) points to its in-order successor.**

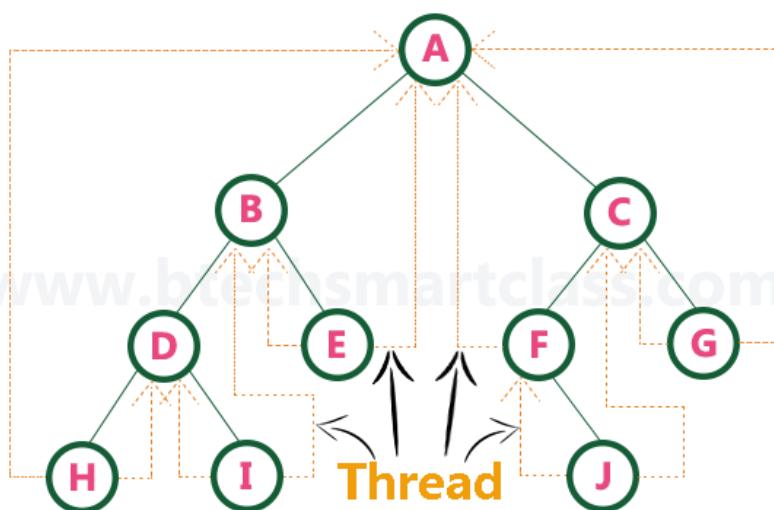
**If there is no in-order predecessor or in-order successor, then it point to root node.**

Consider the following binary tree...



To convert above binary tree into threaded binary tree, first find the in-order traversal of that tree... In-order traversal of above binary tree... H - D - I - B - E - A - F - J - C - G

When we represent above binary tree using linked list representation, nodes H, I, E, F, J and G left child pointers are NULL. This NULL is replaced by address of its in-order predecessor, respectively (I to D, E to B, F to A, J to F and G to C), but here the node H does not have its in-order predecessor, so it points to the root node A. And nodes H, I, E, J and G right child pointers are NULL. This NULL pointers are replaced by address of its in-order successor, respectively (H to D, I to B, E to A, and J to C), but here the node G does not have its in-order successor, so it points to the root node A. Above example binary tree become as follows after converting into threaded binary tree.



In above figure threads are indicated with dotted links.

## Binary Search Tree

In a binary tree, every node can have maximum of two children but there is no order of nodes based on their values. In binary tree, the elements are arranged as they arrive to the tree, from top to bottom and left to right.

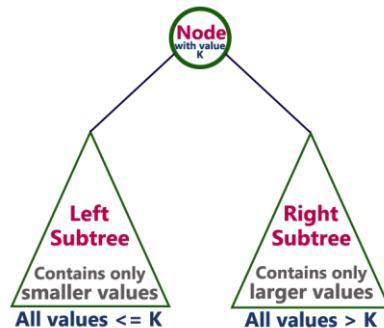
A binary tree has the following time complexities...

1. **Search Operation - O(n)**
2. **Insertion Operation - O(1)**
3. **Deletion Operation - O(n)**

To enhance the performance of binary tree, we use special type of binary tree known as **Binary Search Tree**. Binary search tree mainly focus on the search operation in binary tree. Binary search tree can be defined as follows...

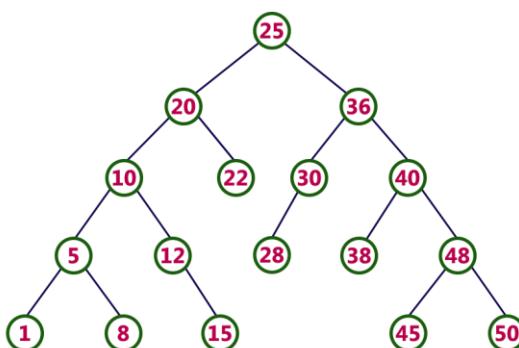
**Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.**

In a binary search tree, all the nodes in left subtree of any node contains smaller values and all the nodes in right subtree of that contains larger values as shown in following figure...



### Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.



**Every Binary Search Tree is a binary tree but all the Binary Trees need not to be binary search trees.**

---

## Operations on a Binary Search Tree

The following operations are performed on a binary search tree...

1. **Search**
2. **Insertion**
3. **Deletion**

### Search Operation in BST

In a binary search tree, the search operation is performed with  $O(\log n)$  time complexity. The search operation is performed as follows...

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the value of root node in the tree.
- **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function
- **Step 4:** If both are not matching, then check whether search element is smaller or larger than that node value.
- **Step 5:** If search element is smaller, then continue the search process in left subtree.
- **Step 6:** If search element is larger, then continue the search process in right subtree.
- **Step 7:** Repeat the same until we found exact element or we completed with a leaf node
- **Step 8:** If we reach to the node with search value, then display "Element is found" and terminate the function.
- **Step 9:** If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

## Insertion Operation in BST

In a binary search tree, the insertion operation is performed with **O(log n)** time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

- **Step 1:** Create a newNode with given value and set its **left** and **right** to **NULL**.
- **Step 2:** Check whether tree is Empty.
- **Step 3:** If the tree is **Empty**, then set **root** to **newNode**.
- **Step 4:** If the tree is **Not Empty**, then check whether value of **newNode** is **smaller** or **larger** than the node (here it is root node).
- **Step 5:** If **newNode** is **smaller** than **or equal** to the node, then move to its **left child**. If **newNode** is **larger** than the node, then move to its **right child**.
- **Step 6:** Repeat the above step until we reach to a **leaf** node (e.i., reach to **NULL**).
- **Step 7:** After reaching a leaf node, then insert the **newNode** as **left child** if **newNode** is **smaller or equal** to that leaf else insert it as**right child**.

## Deletion Operation in BST

In a binary search tree, the deletion operation is performed with **O(log n)** time complexity. Deleting a node from Binary search tree has following three cases...

- **Case 1: Deleting a Leaf node (A node with no children)**
- **Case 2: Deleting a node with one child**
- **Case 3: Deleting a node with two children**

### Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

- **Step 1:** Find the node to be deleted using **search operation**
- **Step 2:** Delete the node using **free** function (If it is a leaf) and terminate the function.

### Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

- **Step 1:** Find the node to be deleted using **search operation**
- **Step 2:** If it has only one child, then create a link between its parent and child nodes.
- **Step 3:** Delete the node using **free** function and terminate the function.

### **Case 3: Deleting a node with two children**

We use the following steps to delete a node with two children from BST...

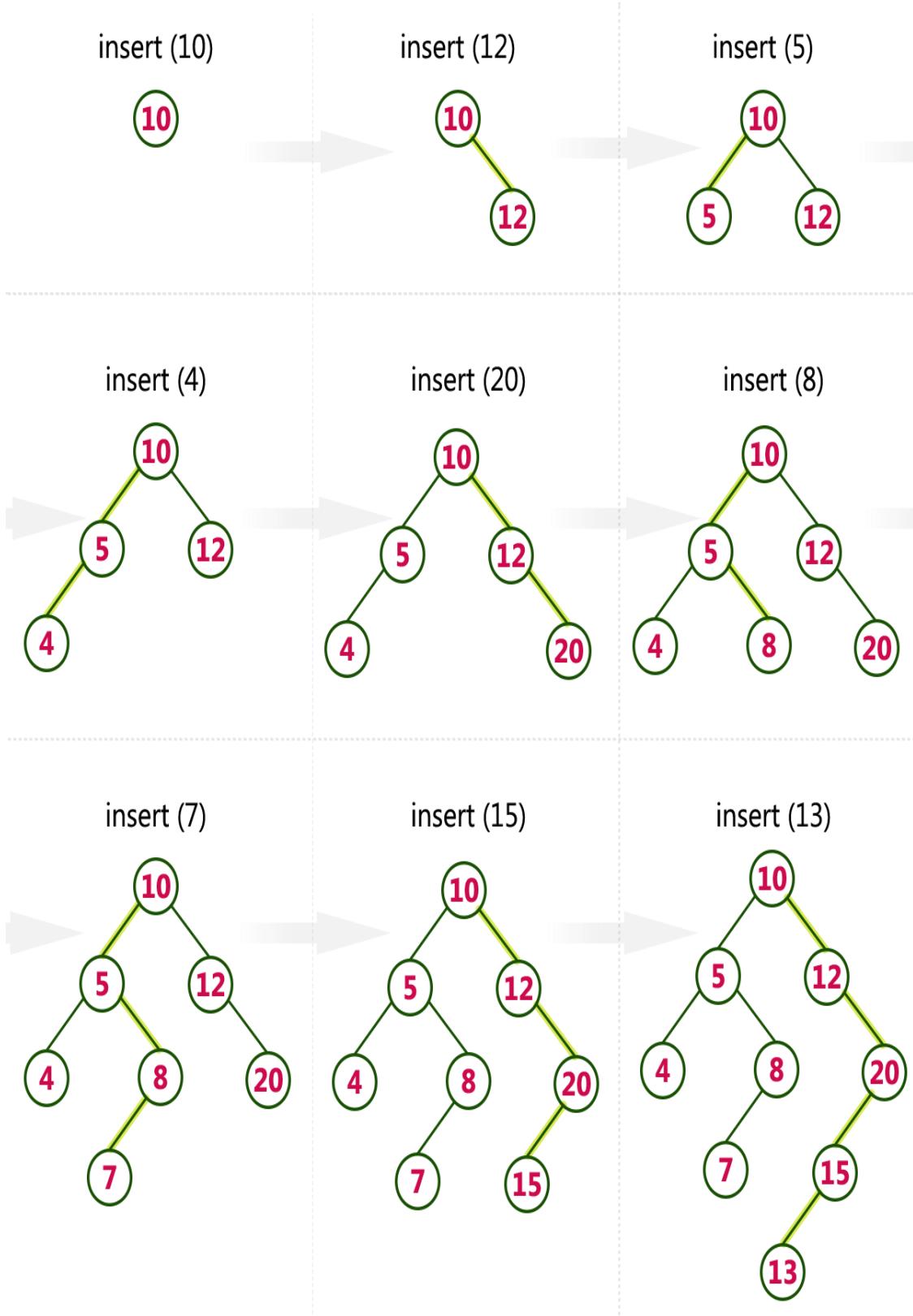
- **Step 1:** Find the node to be deleted using **search operation**
- **Step 2:** If it has two children, then find the **largest** node in its **left subtree** (OR) the **smallest** node in its **right subtree**.
- **Step 3:** Swap both **deleting node** and node which found in above step.
- **Step 4:** Then, check whether deleting node came to **case 1** or **case 2** else goto steps 2
- **Step 5:** If it comes to **case 1**, then delete using case 1 logic.
- **Step 6:** If it comes to **case 2**, then delete using case 2 logic.
- **Step 7:** Repeat the same process until node is deleted from the tree.

### **Example**

Construct a Binary Search Tree by inserting the following sequence of numbers...

**10,12,5,4,20,8,7,15 and 13**

Above elements are inserted into a Binary Search Tree as follows..



## AVL Tree

AVL tree is a self balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced, if the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if for every node, height of its children differ by at most one. In an AVL tree, every node maintains an extra information known as balance factor. The AVL tree was introduced in the year of 1962 by **Adelson-Velsky and Landis**.

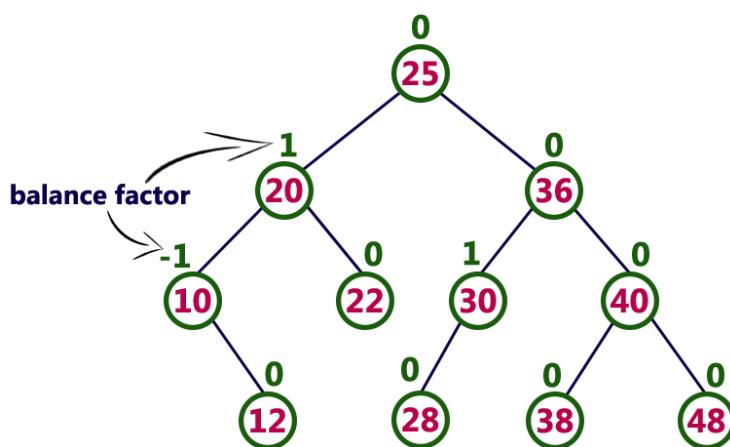
An AVL tree is defined as follows...

**An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.**

**Balance factor of a node is the difference between the heights of left and right subtrees of that node.** The balance factor of a node is calculated either height of left subtree - height of right subtree (OR) height of right subtree - height of left subtree. In the following explanation, we are calculating as follows...

**Balance factor = heightOfLeftSubtree - heightOfRightSubtree**

Example



The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

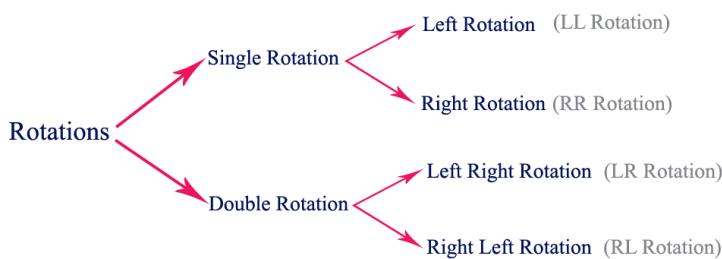
Every AVL Tree is a binary search tree but all the Binary Search Trees need not to be AVL trees.

### AVL Tree Rotations

In AVL tree, after performing every operation like insertion and deletion we need to check the **balance factor** of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. We use **rotation** operations to make the tree balanced whenever the tree is becoming imbalanced due to any operation. Rotation operations are used to make a tree balanced.

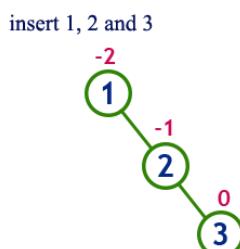
**Rotation is the process of moving the nodes to either left or right to make tree balanced.**

There are **four** rotations and they are classified into **two** types.

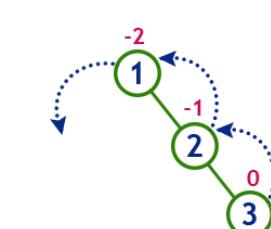


### Single Left Rotation (LL Rotation)

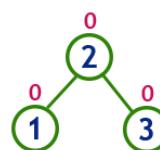
In LL Rotation every node moves one position to left from the current position. To understand LL Rotation, let us consider following insertion operations into an AVL Tree...



Tree is imbalanced



To make balanced we use  
LL Rotation which moves  
nodes one position to left

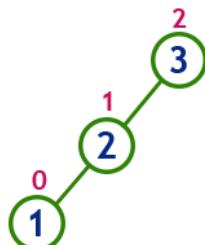


After LL Rotation  
Tree is Balanced

## Single Right Rotation (RR Rotation)

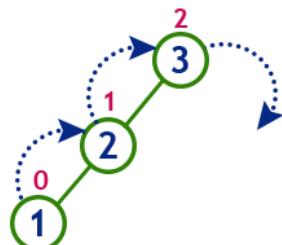
In RR Rotation every node moves one position to right from the current position. To understand RR Rotation, let us consider following insertion operations into an AVL Tree...

insert 3, 2 and 1

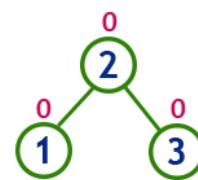


Tree is imbalanced

because node 3 has balance factor 2



To make balanced we use  
RR Rotation which moves  
nodes one position to right

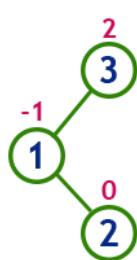


After RR Rotation  
Tree is Balanced

## Left Right Rotation (LR Rotation)

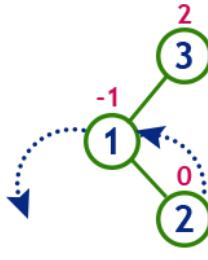
The LR Rotation is combination of single left rotation followed by single right rotation. In LR Rotation, first every node moves one position to left then one position to right from the current position. To understand LR Rotation, let us consider following insertion operations into an AVL Tree...

insert 3, 1 and 2



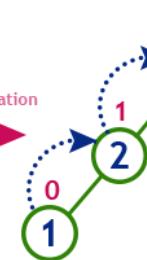
Tree is imbalanced

because node 3 has balance factor 2



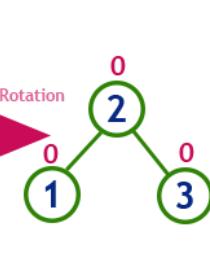
LL Rotation

After LL Rotation



RR Rotation

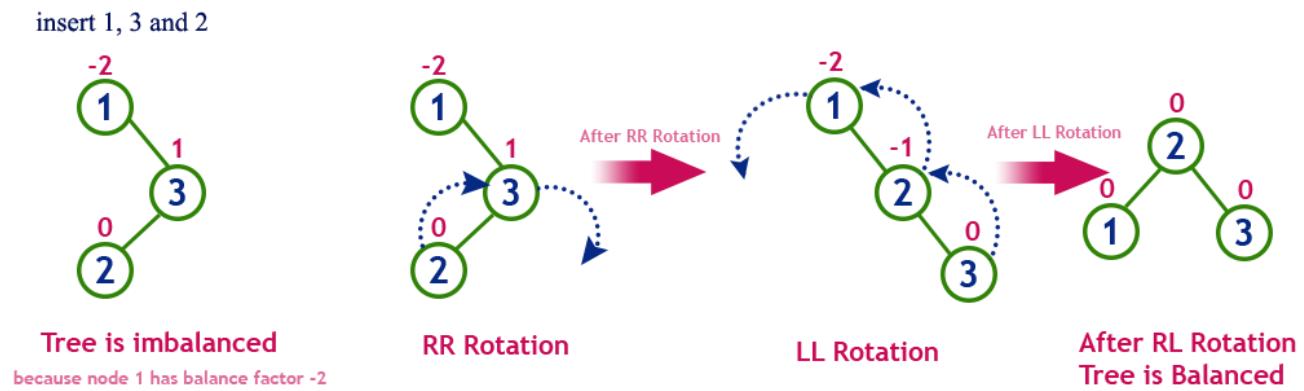
After RR Rotation



After LR Rotation  
Tree is Balanced

## Right Left Rotation (RL Rotation)

The RL Rotation is combination of single right rotation followed by single left rotation. In RL Rotation, first every node moves one position to right then one position to left from the current position. To understand RL Rotation, let us consider following insertion operations into an AVL Tree...



## Operations on an AVL Tree

The following operations are performed on an AVL tree...

1. Search
2. Insertion
3. Deletion

### Search Operation in AVL Tree

In an AVL tree, the search operation is performed with  $O(\log n)$  time complexity. The search operation is performed similar to Binary search tree search operation. We use the following steps to search an element in AVL tree...

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the value of root node in the tree.
- **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function

- **Step 4:** If both are not matching, then check whether search element is smaller or larger than that node value.
- **Step 5:** If search element is smaller, then continue the search process in left subtree.
- **Step 6:** If search element is larger, then continue the search process in right subtree.
- **Step 7:** Repeat the same until we found exact element or we completed with a leaf node
- **Step 8:** If we reach to the node with search value, then display "Element is found" and terminate the function.
- **Step 9:** If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

### Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with  $O(\log n)$  time complexity. In AVL Tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

- **Step 1:** Insert the new element into the tree using Binary Search Tree insertion logic.
- **Step 2:** After insertion, check the **Balance Factor** of every node.
- **Step 3:** If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
- **Step 4:** If the **Balance Factor** of any node is other than **0 or 1 or -1** then tree is said to be imbalanced. Then perform the suitable **Rotation** to make it balanced. And go for next operation.

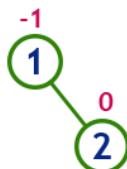
**Example: Construct an AVL Tree by inserting numbers from 1 to 8.**

insert 1



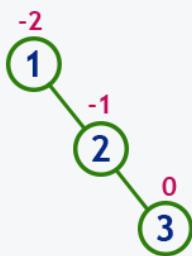
Tree is balanced

insert 2

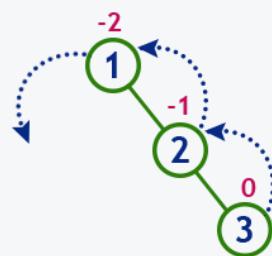


Tree is balanced

insert 3

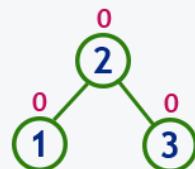


Tree is imbalanced



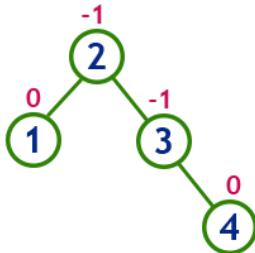
LL Rotation

After LL Rotation



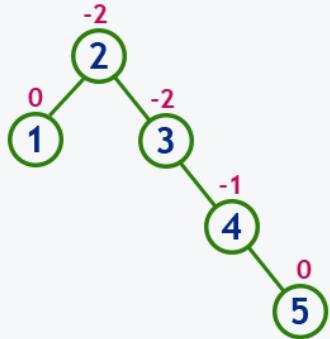
Tree is balanced

insert 4

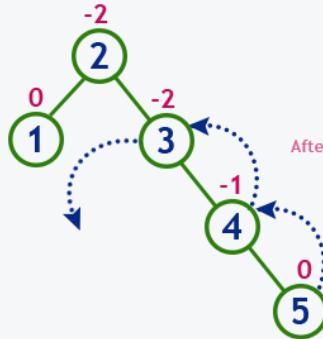


Tree is balanced

insert 5

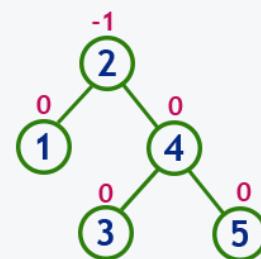


Tree is imbalanced

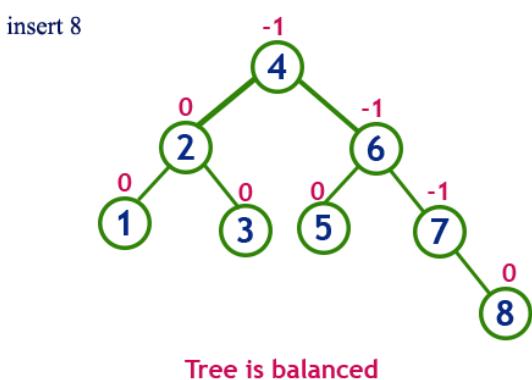
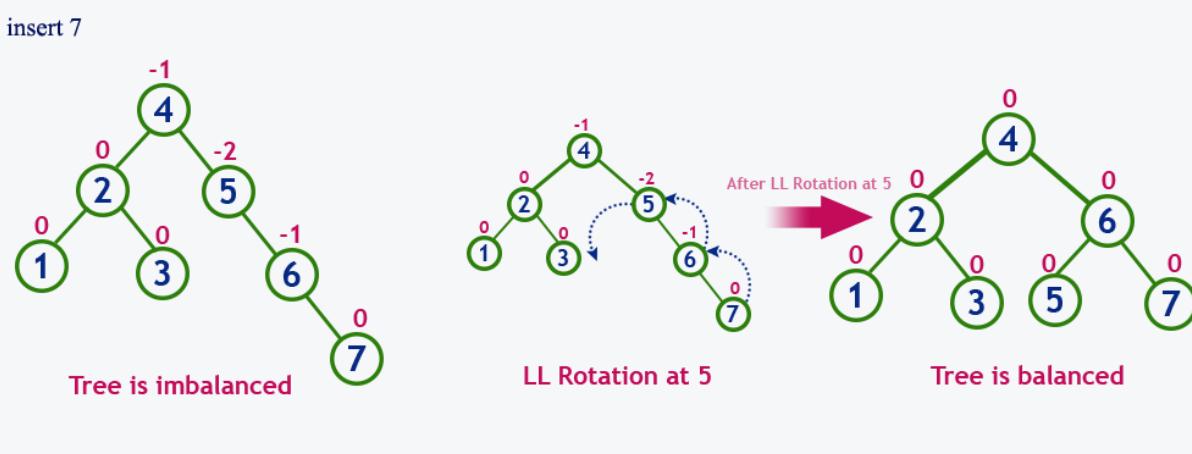
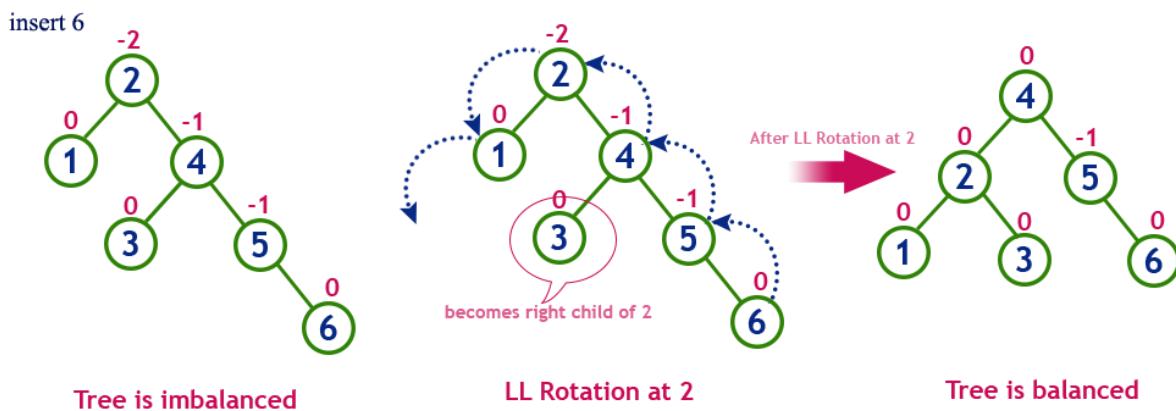


LL Rotation at 3

After LL Rotation at 3



Tree is balanced



### Deletion Operation in AVL Tree

In an AVL Tree, the deletion operation is similar to deletion operation in BST. But after every deletion operation we need to check with the Balance Factor condition. If the tree is balanced after deletion then go for next operation otherwise perform the suitable rotation to make the tree Balanced.

## B - Trees

In a binary search tree, AVL Tree, Red-Black tree etc., every node can have only one value (key) and maximum of two children but there is another type of search tree called B-Tree in which a node can store more than one value (key) and it can have more than two children. B-Tree was developed in the year of 1972 by **Bayer and McCreight** with the name *Height Balanced m-way Search Tree*. Later it was named as B-Tree.

**B-Tree can be defined as follows...**

**B-Tree is a self-balanced search tree with multiple keys in every node and more than two children for every node.**

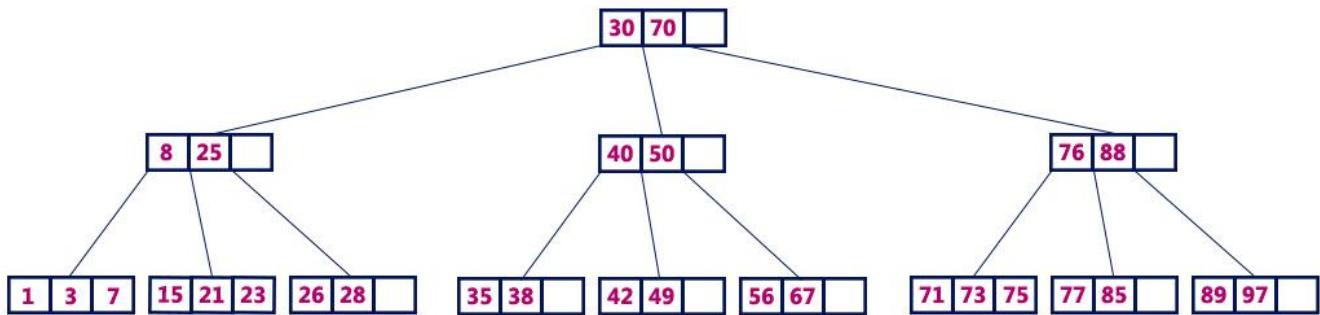
Here, number of keys in a node and number of children for a node is depend on the order of the B-Tree. Every B-Tree has order.

**B-Tree of Order  $m$**  has the following properties...

- **Property #1** - All the **leaf nodes** must be **at same level**.
- **Property #2** - All nodes except root must have at least  $[m/2]-1$  keys and maximum of  $m-1$  keys.
- **Property #3** - All non leaf nodes except root (i.e. all internal nodes) must have at least  $m/2$  children.
- **Property #4** - If the root node is a non leaf node, then it must have **at least 2** children.
- **Property #5** - A non leaf node with  $n-1$  keys must have  $n$  number of children.
- **Property #6** - All the **key values within a node** must be in **Ascending Order**.

For example, B-Tree of Order 4 contains maximum 3 key values in a node and maximum 4 children for a node.

B-Tree of Order 4



## Operations on a B-Tree

The following operations are performed on a B-Tree...

- Search
- Insertion
- Deletion

### Search Operation in B-Tree

- In a B-Ttree, the search operation is similar to that of Binary Search Tree. In a Binary search tree, the search process starts from the root node and every time we make a 2-way decision (we go to either left subtree or right subtree). In B-Tree also search process starts from the root node but every time we make n-way decision where n is the total number of children that node has. In a B-Ttree, the search operation is performed with  $O(\log n)$  time complexity. The search operation is performed as follows...

- Step 1: Read the search element from the user
- Step 2: Compare, the search element with first key value of root node in the tree.
- Step 3: If both are matching, then display "Given node found!!!" and terminate the function
- Step 4: If both are not matching, then check whether search element is smaller or larger than that key value.
- Step 5: If search element is smaller, then continue the search process in left subtree.

- Step 6: If search element is larger, then compare with next key value in the same node and repeate step 3, 4, 5 and 6 until we found exact match or comparision completed with last key value in a leaf node.
- Step 7: If we completed with last key value in a leaf node, then display "Element is not found" and terminate the function.

### Insertion Operation in B-Tree

In a B-Tree, the new element must be added only at leaf node. That means, always the new keyValue is attached to leaf node only. The insertion operation is performed as follows...

- Step 1: Check whether tree is Empty.
- Step 2: If tree is Empty, then create a new node with new key value and insert into the tree as a root node.
- Step 3: If tree is Not Empty, then find a leaf node to which the new key value cab be added using Binary Search Tree logic.
- Step 4: If that leaf node has an empty position, then add the new key value to that leaf node by maintaining ascending order of key value within the node.
- Step 5: If that leaf node is already full, then split that leaf node by sending middle value to its parent node. Repeat tha same until sending value is fixed into a node.
- Step 6: If the spilting is occurring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Construct a B-Tree of Order 3 by inserting numbers from 1 to 10.

Construct a B-Tree of order 3 by inserting numbers from 1 to 10.

#### insert(1)

Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.



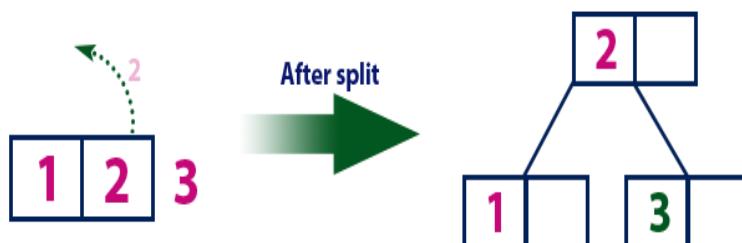
#### insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.



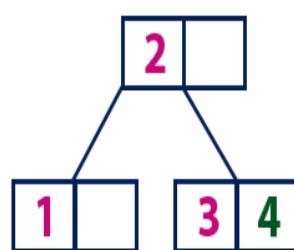
#### insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't have an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't have a parent. So, this middle value becomes a new root node for the tree.



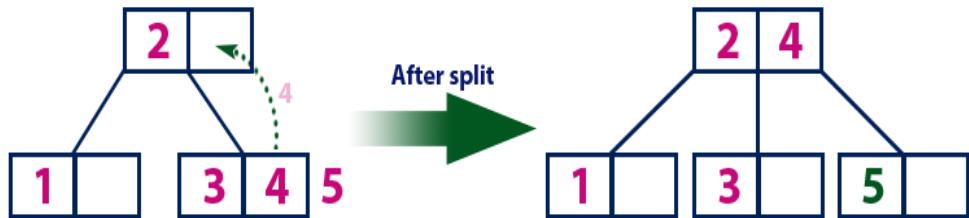
#### insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.

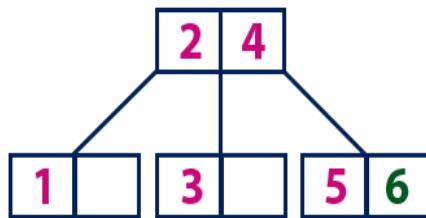


**insert(5)**

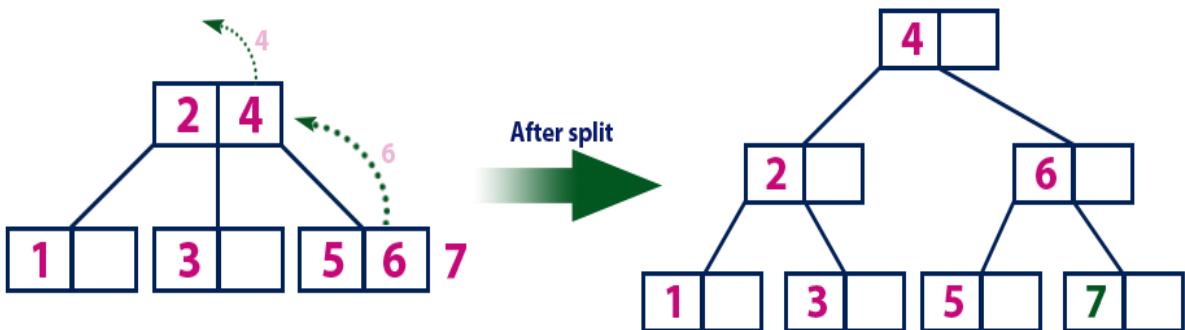
Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.

**insert(6)**

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.

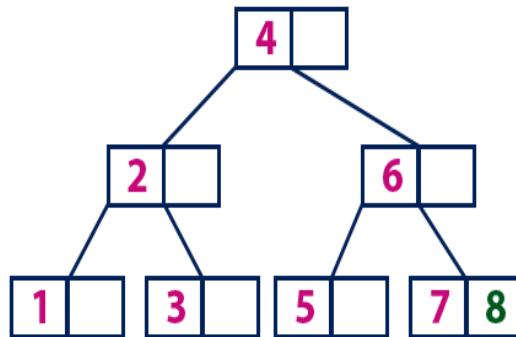
**insert(7)**

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.

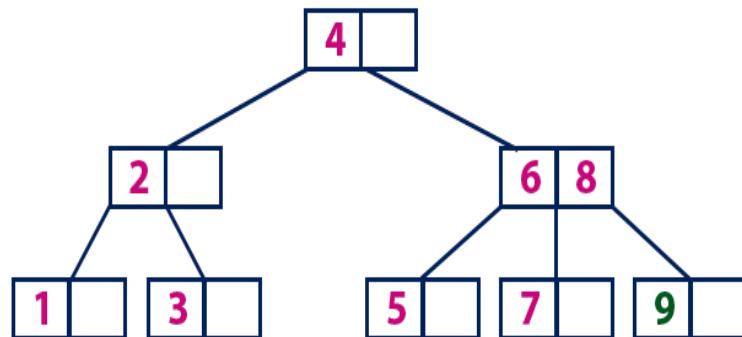


**insert(8)**

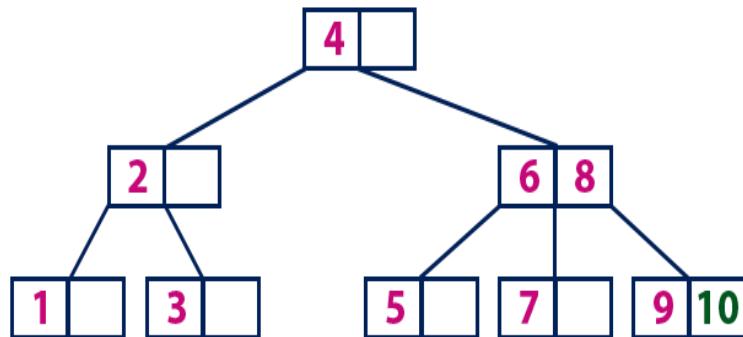
Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.

**insert(9)**

Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.

**insert(10)**

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8'. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.



## Splay Tree

- Splay tree is another variant of binary search tree. In a splay tree, the recently accessed element is placed at the root of the tree. A splay tree is defined as follows...
- Splay Tree is a self - adjusted Binary Search Tree in which every operation on an element rearrange the tree so that the element is placed at the root position of the tree.
- In a splay tree, every operation is performed at root of the tree. All the operations on a splay tree are involved with a common operation called "Splaying".
- **Splaying an element is the process of bringing it to the root position by performing suitable rotation operations.**
- In a splay tree, splaying an element rearrange all the elements in the tree so that splayed element is placed at root of the tree.

With the help of splaying an element we can bring most frequently used element closer to the root of the tree so that any operation on those element performed quickly. That means the splaying operation automatically brings more frequently used elements closer to the root of the tree.

Every operation on a splay tree performs the splaying operation. For example, the insertion operation first inserts the new element as it inserted into the binary search tree, after insertion the newly inserted element is splayed so that it is placed at root of the tree. The search operation in a splay tree is search the element using binary search process then splay the searched element so that it placed at the root of the tree.

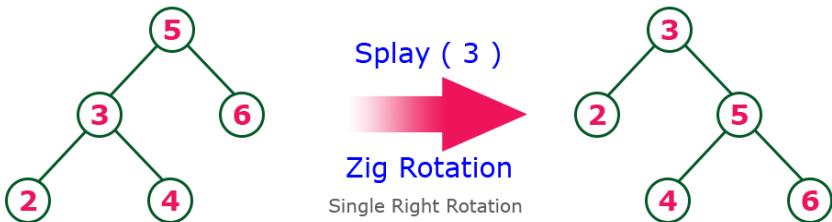
In a splay tree, to splay any element we use the following rotation operations...

### Rotations in Splay Tree

1. Zig Rotation
2. Zag Rotation
3. Zig - Zig Rotation
4. Zag - Zag Rotation
5. Zig - Zag Rotation
6. Zag - Zig Rotation

## Zig Rotation

The Zig Rotation in a splay tree is **similar to the single right rotation in AVL Tree rotations**. In zig rotation every node moves one position to the right from its current position. Consider the following example...



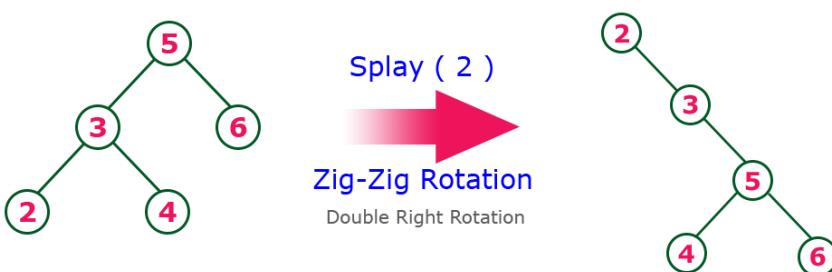
## Zag Rotation

The Zag Rotation in a splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation every node moves one position to the left from its current position. Consider the following example...



## Zig-Zig Rotation

The Zig-Zig Rotation in a splay tree is a double zig rotation. In zig-zig rotation every node moves two position to the right from its current position. Consider the following example...



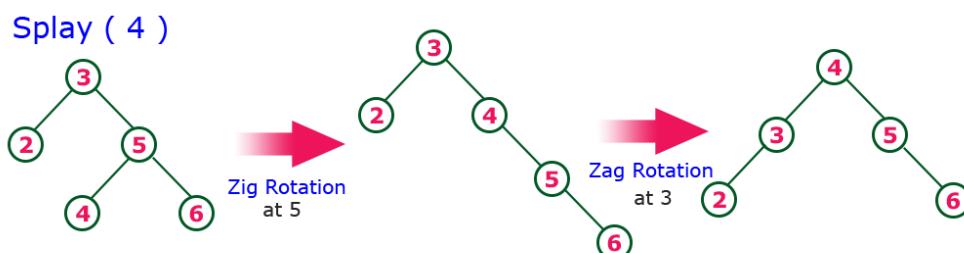
### Zag-Zag Rotation

The Zag-Zag Rotation in a splay tree is a double zig rotation. In zag-zag rotation every node moves two position to the left from its current position. Consider the following example...

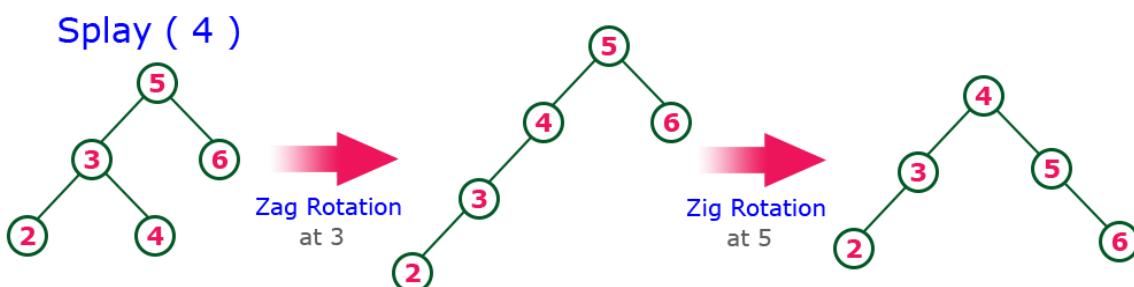


### Zig-Zag Rotation

The Zig-Zag Rotation in a splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation every node moves one position to the right followed by one position to the left from its current position. Consider the following example...



### Zag-Zig Rotation



The Zag-Zig Rotation in a splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation every node moves one position to the left followed by one position to the right from its current position. Consider the following example...

### Note

**Every Splay tree must be a binary search tree but it is need not to be balanced tree.**

### Insertion Operation in Splay Tree

The insertion operation in Splay tree is performed using following steps...

- Step 1: Check whether tree is Empty.
- Step 2: If tree is Empty then insert the newNode as Root node and exit from the operation.
- Step 3: If tree is not Empty then insert the newNode as a leaf node using Binary Search tree insertion logic.
- Step 4: After insertion, Splay the newNode

### Deletion Operation in Splay Tree

In a Splay Tree, the deletion operation is similar to deletion operation in Binary Search Tree. But before deleting the element first we need to splay that node then delete it from the root position then join the remaining tree.

## Red-Black Trees

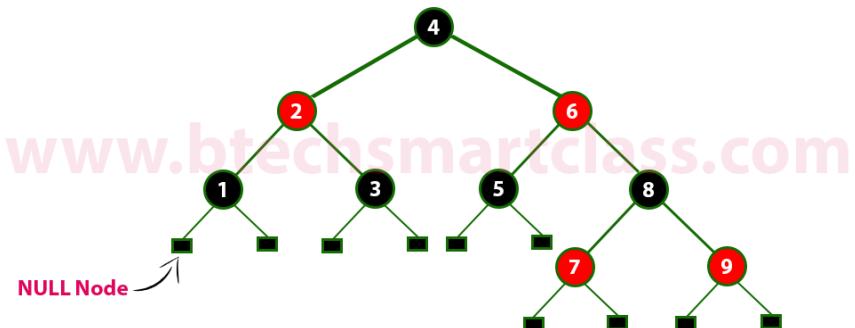
- Red - Black Tree is another variant of Binary Search Tree in which every node is colored either RED or BLACK. We can define a Red Black Tree as follows...
- **Red Black Tree is a Binary Search Tree in which every node is colored either RED or BLACK.**
- In a Red Black Tree the color of a node is decided based on the Red Black Tree properties. Every Red Black Tree has the following properties.

## Properties of Red Black Tree

- Property #1: Red - Black Tree must be a Binary Search Tree.
- Property #2: The ROOT node must colored BLACK.
- Property #3: The children of Red colored node must colored BLACK. (There should not be two consecutive RED nodes).
- Property #4: In all the paths of the tree there must be same number of BLACK colored nodes.
- Property #5: Every new node must inserted with RED color.
- Property #6: Every leaf (e.i. NULL node) must colored BLACK.

### Example

The following is a Red Black Tree which created by inserting numbers from 1 to 9.



The above tree is a Red Black tree and every node is satisfying all the properties of Red Black Tree.

**Every Red Black Tree is a binary search tree but all the Binary Search Trees need not to be Red Black trees.**

### Insertion into RED BLACK Tree:

- In a Red Black Tree, every new node must be inserted with color RED.
- The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property.

- After every insertion operation, we need to check all the properties of Red Black Tree. If all the properties are satisfied then we go to next operation otherwise we need to perform following operation to make it Red Black Tree.
  - Recolor
  - Rotation followed by Recolor

The insertion operation in Red Black tree is performed using following steps...

- Step 1: Check whether tree is Empty.
- Step 2: If tree is Empty then insert the newNode as Root node with color Black and exit from the operation.
- Step 3: If tree is not Empty then insert the newNode as a leaf node with Red color.
- Step 4: If the parent of newNode is Black then exit from the operation.
- Step 5: If the parent of newNode is Red then check the color of parent node's sibling of newNode.
- Step 6: If it is Black or NULL node then make a suitable Rotation and Recolor it.
- Step 7: If it is Red colored node then perform Recolor and Recheck it. Repeat the same until tree becomes Red Black Tree.

Example

Create a RED BLACK Tree by inserting following sequence of number  
8, 18, 5, 15, 17, 25, 40 & 80.

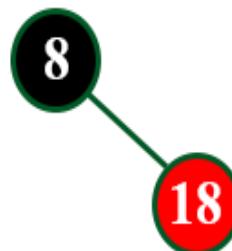
### insert ( 8 )

Tree is Empty. So insert newNode as Root node with black color.



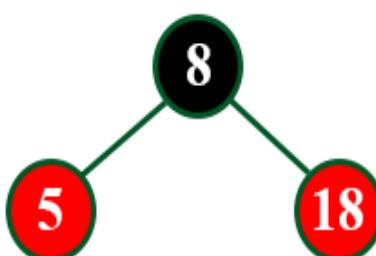
### insert ( 18 )

Tree is not Empty. So insert newNode with red color.



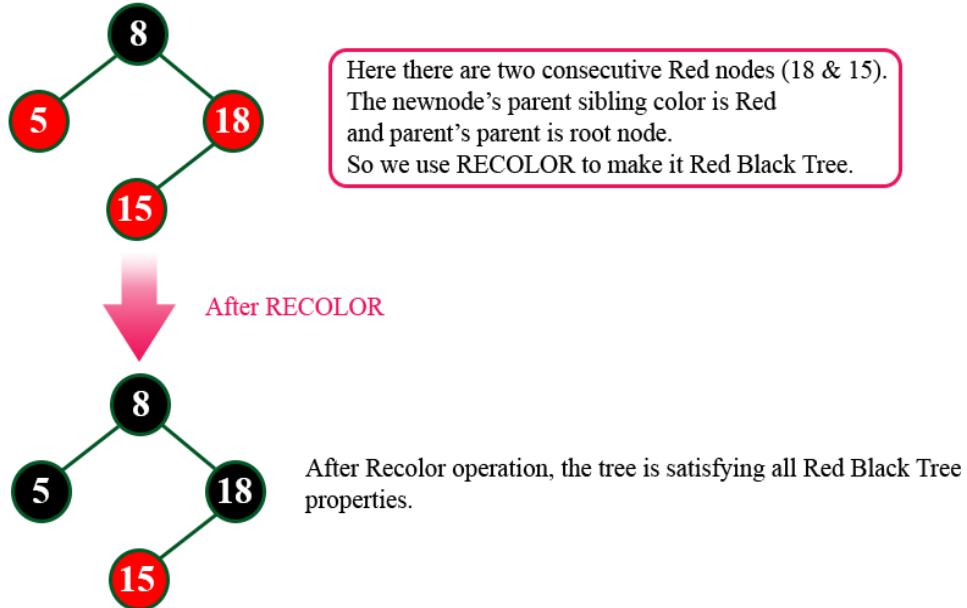
### insert ( 5 )

Tree is not Empty. So insert newNode with red color.

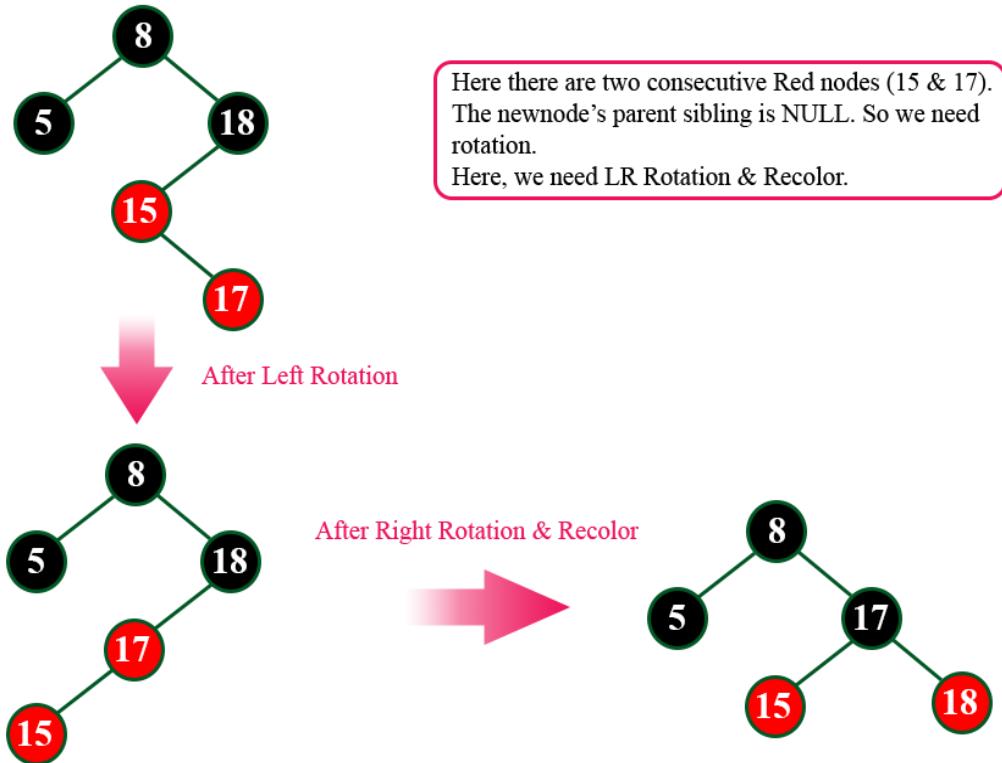


**insert ( 15 )**

Tree is not Empty. So insert newNode with red color.

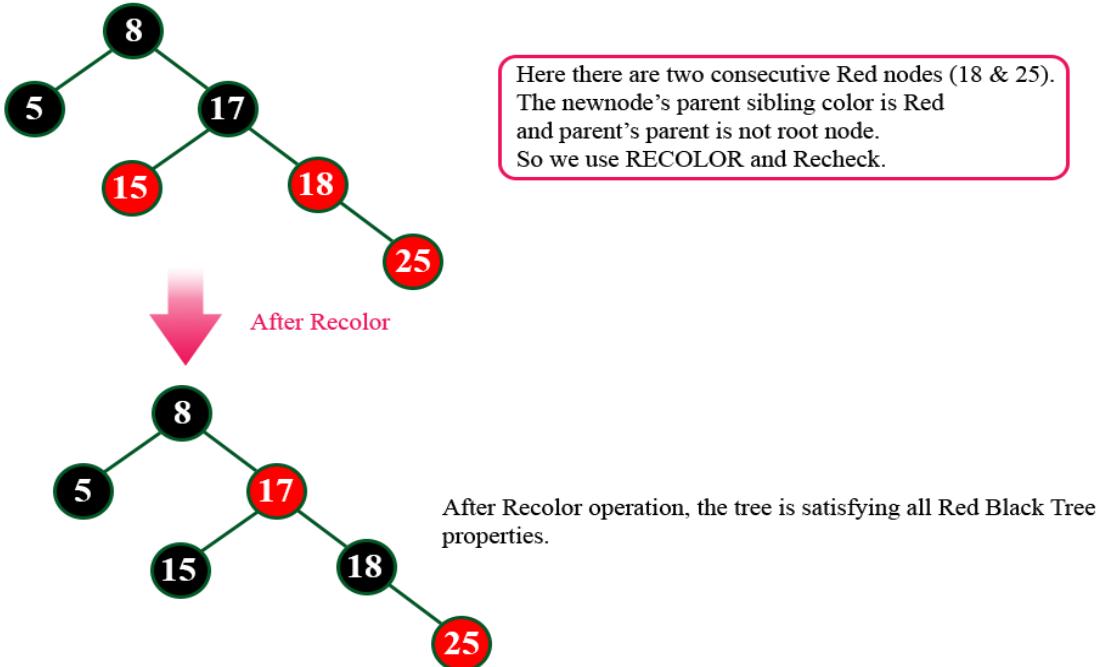
**insert ( 17 )**

Tree is not Empty. So insert newNode with red color.

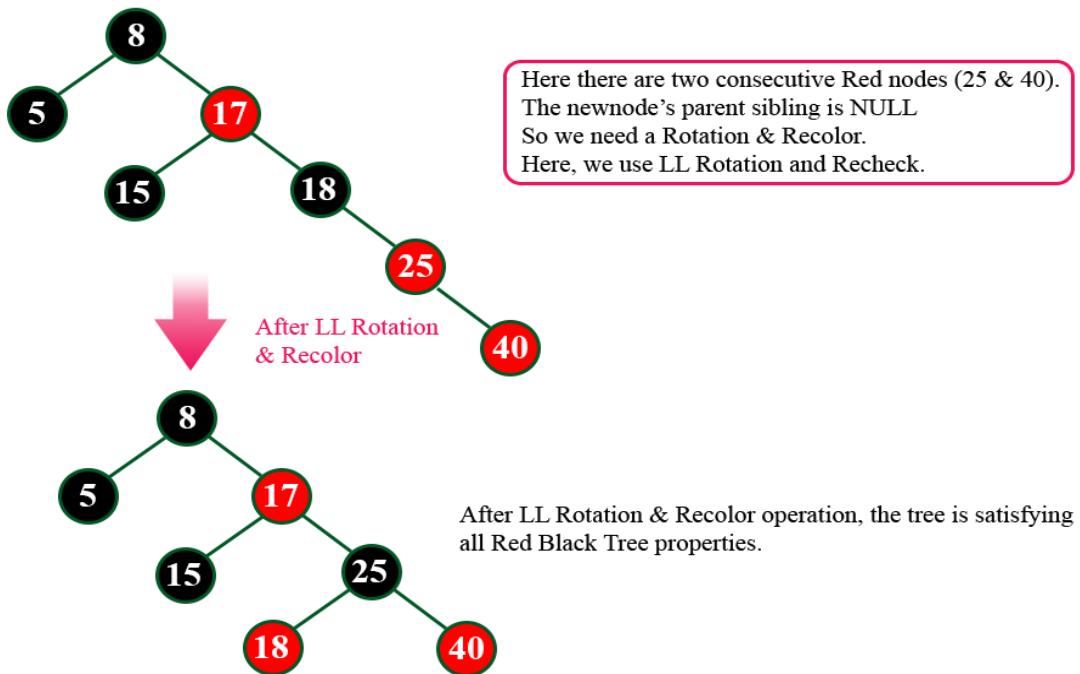


**insert ( 25 )**

Tree is not Empty. So insert newNode with red color.

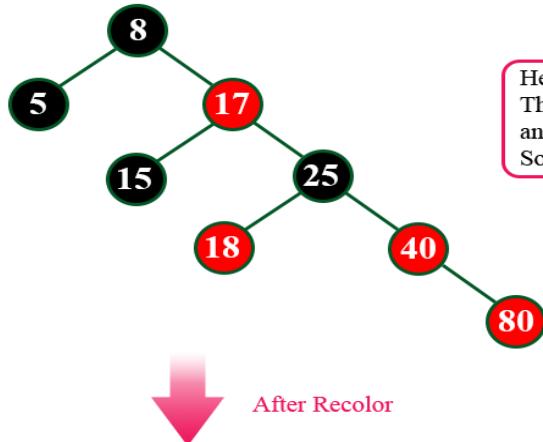
**insert ( 40 )**

Tree is not Empty. So insert newNode with red color.

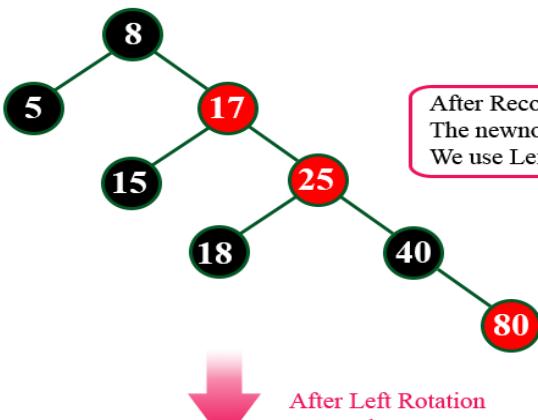


**insert ( 80 )**

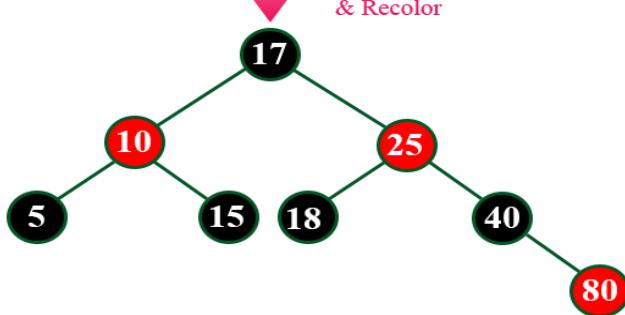
Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (40 & 80).  
The newnode's parent sibling color is Red  
and parent's parent is not root node.  
So we use RECOLOR and Recheck.



After Recolor again there are two consecutive Red nodes (17 & 25).  
The newnode's parent sibling color is Black. So we need Rotation.  
We use Left Rotation & Recolor.



Finally above tree is satisfying all the properties of Red Black Tree and it is a perfect Red Black tree.

### **Deletion Operation in Red Black Tree**

In a Red Black Tree, the deletion operation is similar to deletion operation in BST. But after every deletion operation we need to check with the Red Black Tree properties. If any of the property is violated then make suitable operation like Recolor or Rotation & Recolor.

## UNIT V

### GRAPHS

Graph Terminology, Graph Traversal, Topological sorting - Minimum spanning tree – Prims - Kruskals - Network flow problem - Shortest Path Algorithm: Dijkstra - Graph Search: Depth First Search, Breadth First Search - Hashing: Hash functions, Collision avoidance, Separate chaining - Open addressing: Linear probing, Quadratic Probing, Double hashing, Rehashing, Extensible Hashing

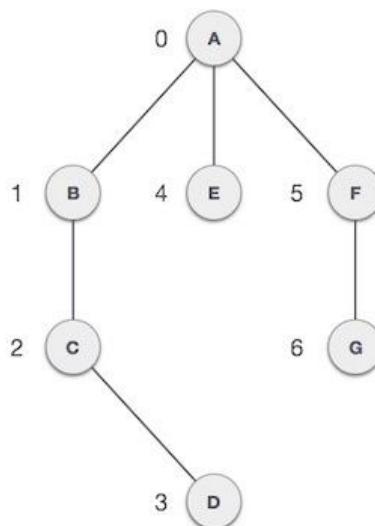
---

## GRAPHS

### Graph Terminology

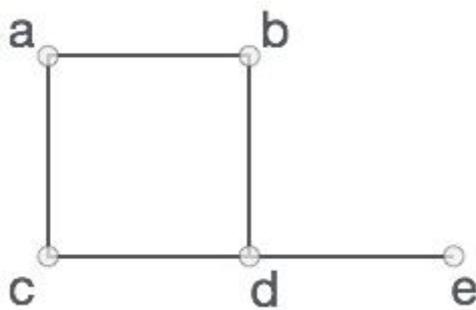
Mathematical graphs can be represented in data-structure. We can represent a graph using an array of vertices and a two dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms –

- **Vertex** – Each node of the graph is represented as a vertex. In example given below, labeled circle represents vertices. So A to G are vertices. We can represent them using an array as shown in image below. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** – Edge represents a path between two vertices or a line between two vertices. In example given below, lines from A to B, B to C and so on represents edges. We can use a two dimensional array to represent array as shown in image below. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge. In example given below, B is adjacent to A, C is adjacent to B and so on.
- **Path** – Path represents a sequence of edges between two vertices. In example given below, ABCD represents a path from A to D.



A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

Formally, a graph is a pair of sets  $(V, E)$ , where  $V$  is the set of vertices and  $E$  is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



In the above graph,

$$V = \{a, b, c, d, e\}$$

$$E = \{ab, ac, bd, cd, de\}$$

## Basic Operations

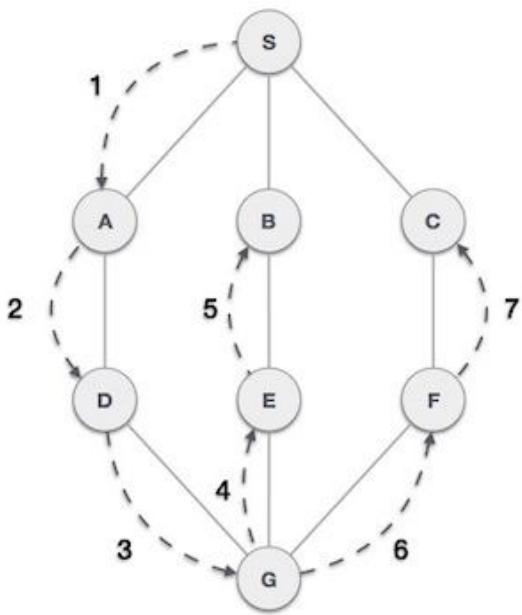
Following are basic primary operations of a Graph which are following.

- **Add Vertex** – add a vertex to a graph.
- **Add Edge** – add an edge between two vertices of a graph.
- **Display Vertex** – display a vertex of a graph.

## Graph Traversal

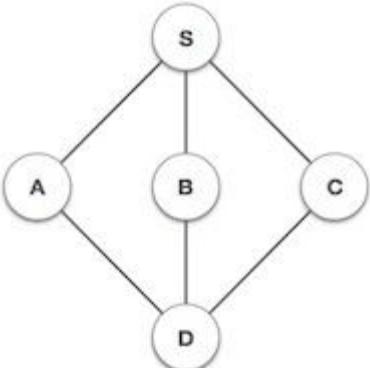
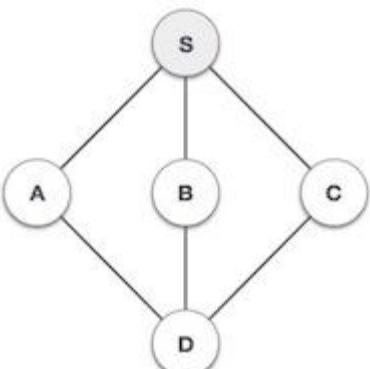
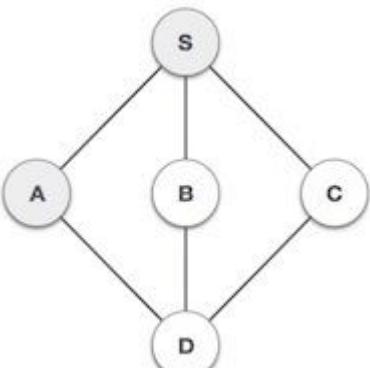
### 1. Depth First Traversal

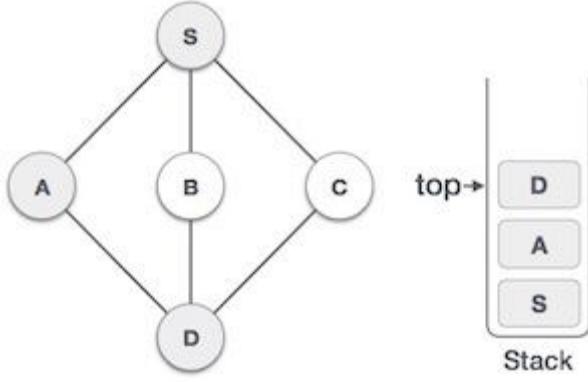
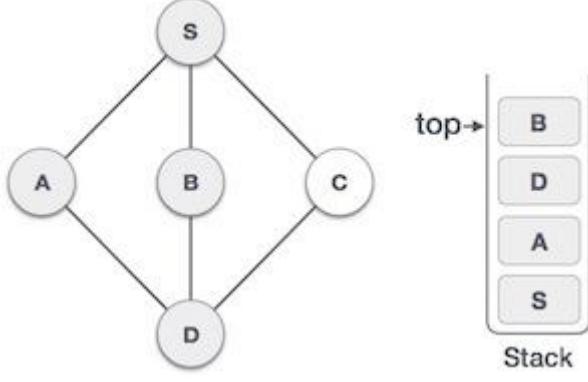
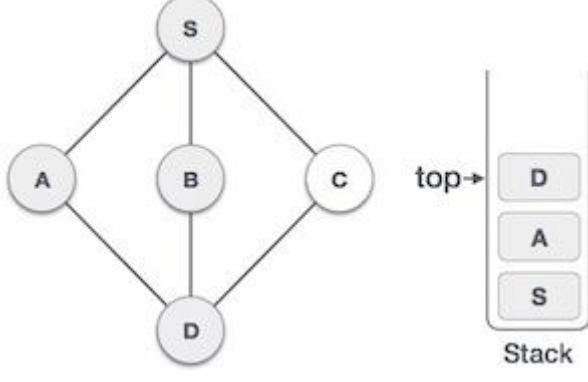
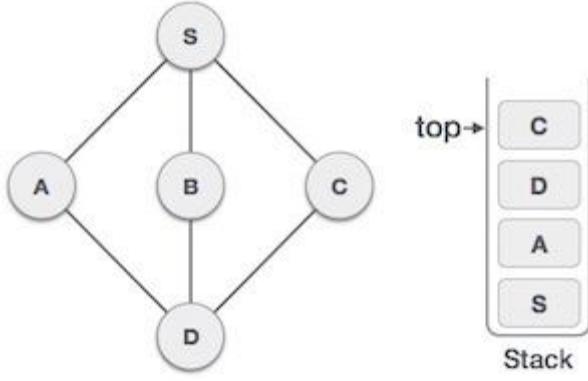
**Depth First Search algorithm(DFS)** traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.



As in example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs following rules.

- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex found, pop up a vertex from stack. (It will pop up all the vertices from the stack which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until stack is empty.

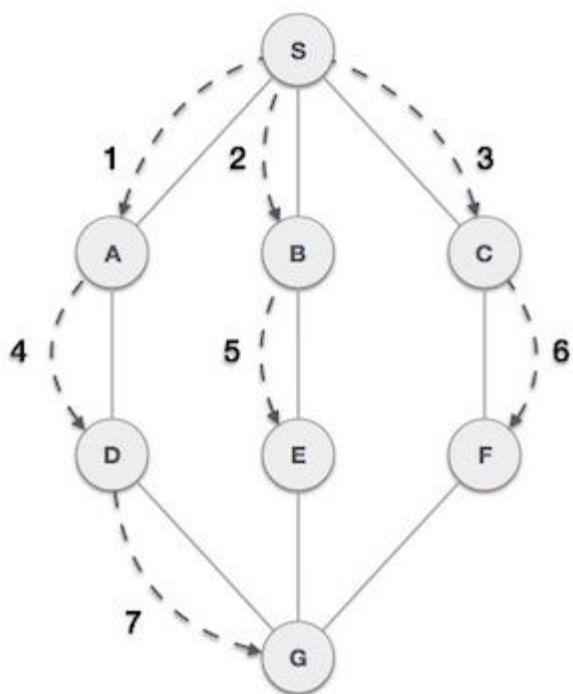
| Step | Traversal                                                                                                                            | Description                                                                                                                                                                                                              |
|------|--------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1.   | <br><span style="float: right;">Stack</span>        | Initialize the stack                                                                                                                                                                                                     |
| 2.   | <br><span style="float: right;">top→ Stack</span>  | Mark <b>S</b> as visited and put it onto the stack. Explore any unvisited adjacent node from <b>S</b> . We have three nodes and we can pick any of them. For this example, we shall take the node in alphabetical order. |
| 3.   | <br><span style="float: right;">top→ Stack</span> | Mark <b>A</b> as visited and put it onto the stack. Explore any unvisited adjacent node from <b>A</b> . Both <b>S</b> and <b>D</b> are adjacent to <b>A</b> but we are concerned for unvisited nodes only.               |

|    |                                                                                                               |                                                                                                                                                                                                                    |
|----|---------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 4. |  <p>top →</p> <p>Stack</p>   | <p>Visit <b>D</b> and mark it visited and put onto the stack. Here we have <b>B</b> and <b>C</b> nodes which are adjacent to <b>D</b> and both are unvisited. But we shall again choose in alphabetical order.</p> |
| 5. |  <p>top →</p> <p>Stack</p>  | <p>We choose <b>B</b>, mark it visited and put onto stack. Here <b>B</b> does not have any unvisited adjacent node. So we pop <b>B</b> from the stack.</p>                                                         |
| 6. |  <p>top →</p> <p>Stack</p> | <p>We check stack top for return to previous node and check if it has any unvisited nodes. Here, we find <b>D</b> to be on the top of stack.</p>                                                                   |
| 7. |  <p>top →</p> <p>Stack</p> | <p>Only unvisited adjacent node is from <b>D</b> is <b>C</b> now. So we visit <b>C</b>, mark it visited and put it onto the stack.</p>                                                                             |

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node which has unvisited adjacent node. In this case, there's none and we keep popping until stack is empty.

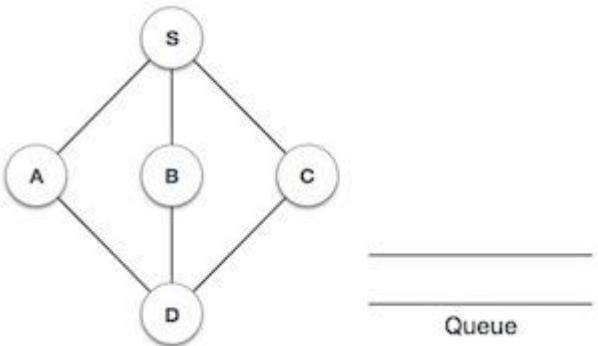
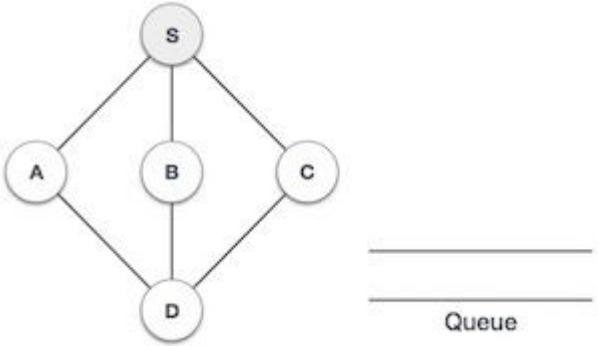
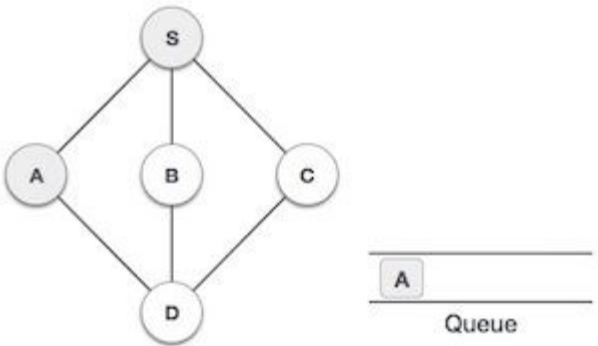
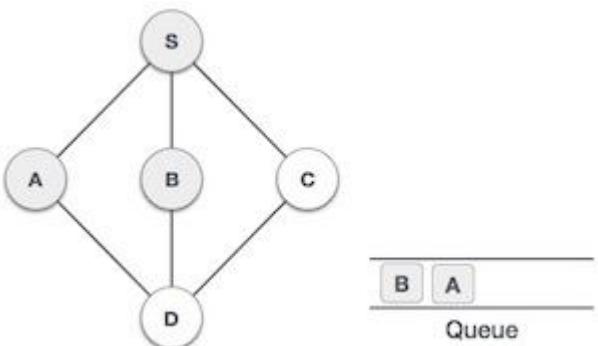
## 2. Breadth First Traversal

Breadth First Search algorithm(BFS) traverses a graph in a breadth wards motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.



As in example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs following rules.

- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex found, remove the first vertex from queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until queue is empty.

| Step | Traversal                                                                           | Description                                                                                                                                                    |
|------|-------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1.   |    | Initialize the queue.                                                                                                                                          |
| 2.   |   | We start from visiting <b>S</b> (starting node), and mark it visited.                                                                                          |
| 3.   |  | We then see unvisited adjacent node from <b>S</b> . In this example, we have three nodes but alphabetically we choose <b>A</b> mark it visited and enqueue it. |
| 4.   |  | Next unvisited adjacent node from <b>S</b> is <b>B</b> . We mark it visited and enqueue it.                                                                    |

|    |                                                                                                                                                                              |                                                                                               |
|----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|
| 5. | <p>Graph diagram showing nodes S, A, B, C, D. S is the root. A is left child of S. B is middle child of S. C is right child of S. D is bottom child of B. Queue: C, B, A</p> | Next unvisited adjacent node from <b>S</b> is <b>C</b> . We mark it visited and enqueue it.   |
| 6. | <p>Graph diagram showing nodes S, A, B, C, D. S is the root. A is left child of S. B is middle child of S. C is right child of S. D is bottom child of B. Queue: C, B</p>    | Now <b>S</b> is left with no unvisited adjacent nodes. So we dequeue and find <b>A</b> .      |
| 7. | <p>Graph diagram showing nodes S, A, B, C, D. S is the root. A is left child of S. B is middle child of S. C is right child of S. D is bottom child of B. Queue: D, C, B</p> | From <b>A</b> we have <b>D</b> as unvisited adjacent node. We mark it visited and enqueue it. |

At this stage we are left with no unmarked (unvisited) nodes. But as per algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied the program is over.

## Topological sort

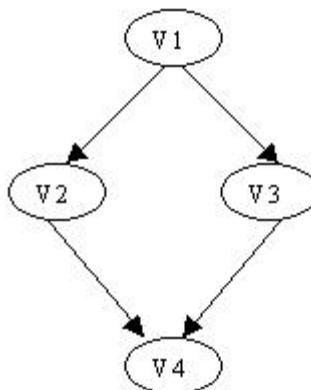
**Topological sort:** an ordering of the vertices in a directed acyclic graph, such that:

If there is a path from **u** to **v**, then **v** appears after **u** in the ordering.

### Types of graphs:

The graphs should be **directed**: otherwise for any edge  $(u,v)$  there would be a path from  $u$  to  $v$  and also from  $v$  to  $u$ , and hence they cannot be ordered.

The graphs should be **acyclic**: otherwise for any two vertices  $u$  and  $v$  on a cycle  $u$  would precede  $v$  and  $v$  would precede  $u$ . The ordering may not be unique:



V1, V2, V3, V4 and V1, V3, V2, V4 are legal orderings

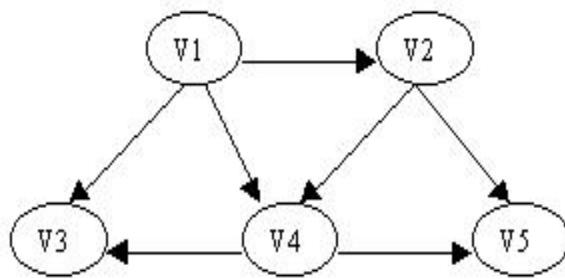
**Degree** of a vertex U: the number of edges  $(U,V)$  - outgoing edges

**Indegree** of a vertex U: the number of edges  $(V,U)$  - incoming edges

The algorithm for topological sort uses "indegrees" of vertices.

### Algorithm

1. Compute the indegrees of all vertices
2. Find a vertex **U** with indegree 0 and print it (store it in the ordering)  
If there is no such vertex then there is a cycle  
and the vertices cannot be ordered. Stop.
3. Remove **U** and all its edges **(U,V)** from the graph.
4. Update the indegrees of the remaining vertices.
5. Repeat steps 2 through 4 while there are vertices to be processed.

**Example**

1. Compute the indegrees:    V1: 0                  V2: 1                  V3: 2                  V4: 2                  V5: 2
2. Find a vertex with indegree 0: V1
3. Output V1 , remove V1 and update the indegrees:

Sorted: V1

Remove edges: (V1,V2) , (V1, V3) and (V1,V4)

Updated indegrees: V2: 0                  V3: 1                  V4: 1                  V5: 2

The process is depicted in the following table:

|        | Indegree |    |       |          |             |                |
|--------|----------|----|-------|----------|-------------|----------------|
| Sorted |          | V1 | V1,V2 | V1,V2,V4 | V1,V2,V4,V3 | V1,V2,V4,V3,V5 |
| V1     | 0        |    |       |          |             |                |
| V2     | 1        | 0  |       |          |             |                |
| V3     | 2        | 1  | 1     | 0        |             |                |
| V4     | 2        | 1  | 0     |          |             |                |
| V5     | 2        | 2  | 1     | 0        | 0           |                |

One possible sorting: V1, V2, V4, V3, V5

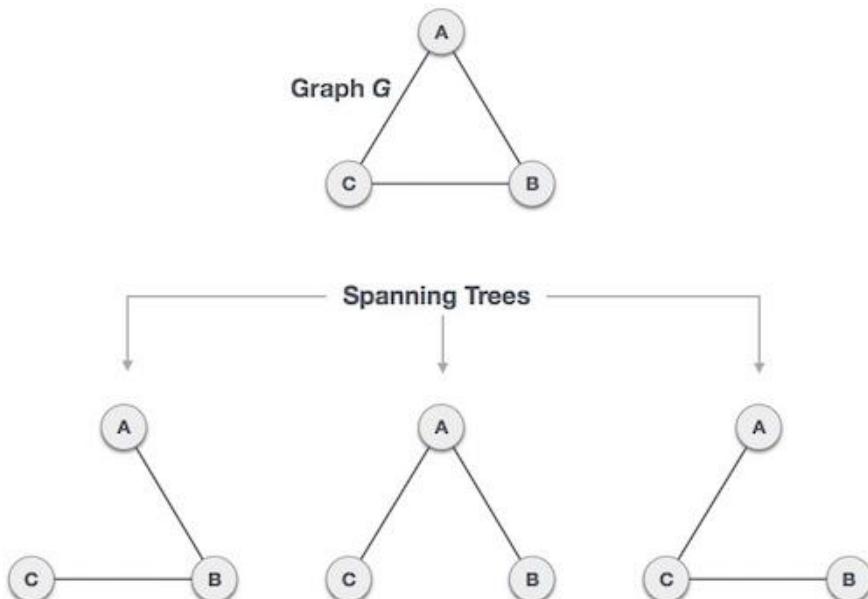
Another sorting: V1, V2, V4, V5, V3

**Complexity** of this algorithm:  $O(|V|^2)$ ,  $|V|$  - the number of vertices. To find a vertex of indegree 0 we scan all the vertices -  $|V|$  operations. We do this for all vertices:  $|V|^2$

## Minimum Spanning Tree

A **spanning tree** is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it can not be disconnected. By this definition we can draw a conclusion that every connected & undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it can not spanned to all its vertices.

We found three spanning trees off one complete graph. A complete undirected graph can have maximum  $n^{n-2}$  number of spanning trees, where n is number of nodes. In addressed example, n is 3, hence  $3^{3-2} = 3$  spanning trees are possible.



### General properties of spanning tree

We now understand that one graph can have more than one spanning trees. Below are few properties of spanning tree of given connected graph G –

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have same number of edges and vertices.
- Spanning tree does not have any cycle (loops)
- Removing one edge from spanning tree will make the graph disconnected i.e. spanning tree is **minimally connected**.
- Adding one edge to a spanning tree will create a circuit or loop i.e. spanning tree is **maximally acyclic**.

## Mathematical properties of spanning tree

- Spanning tree has  $n-1$  edges, where  $n$  is number of nodes (vertices)
- From a complete graph, by removing maximum  $e-n+1$  edges, we can construct a spanning tree.
- A complete graph can have maximum  $n^{n-2}$  number of spanning trees.

So we can conclude here that spanning trees are subset of a connected Graph G and disconnected Graphs do not have spanning tree.

## Application of Spanning Tree

Spanning tree is basically used to find minimum paths to connect all nodes in a graph.  
Common application of spanning trees are –

- **Civil Network Planning**
- **Computer Network Routing Protocol**
- **Cluster Analysis**

Lets understand this by a small example. Consider city network as a huge graph and now plan to deploy telephone lines such a way that in minimum lines we can connect to all city nodes. This is where spanning tree comes in the picture.

## Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

## Minimum Spanning-Tree Algorithm

We shall learn about two most important spanning tree algorithms here –

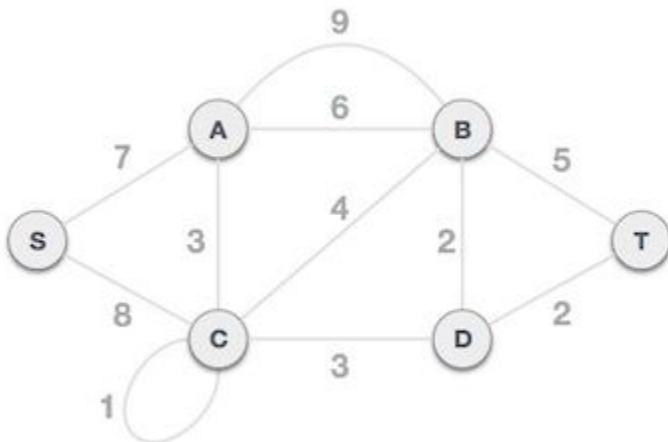
- Kruskal's Algorithm
- Prim's Algorithm

Both are greedy algorithms.

## Kruskal's Algorithm

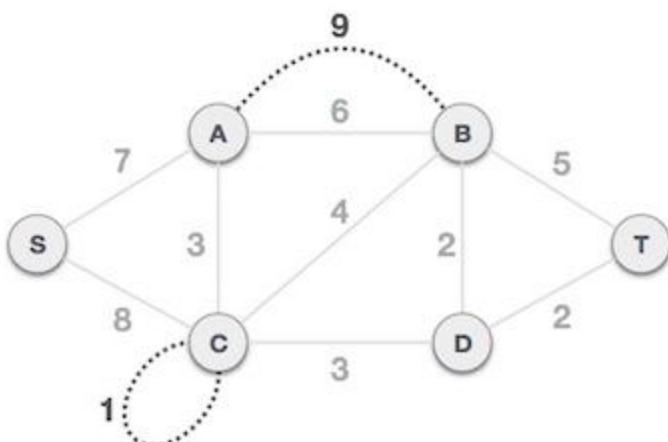
Kruskal's algorithm to find minimum cost spanning tree uses greedy approach. This algorithm treats the graph as a forest and every node it as an individual tree. A tree connects to another only and only if it has least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm we shall take the following example –

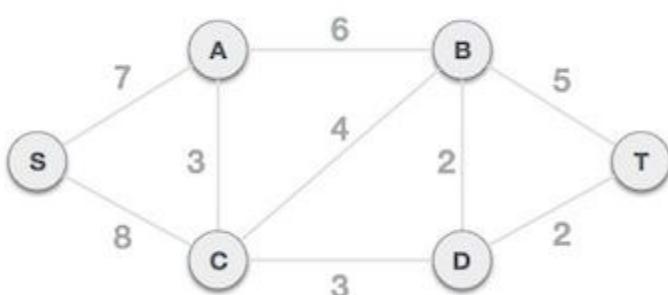


Step 1 - Remove all loops & Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has least cost associated and remove all others.



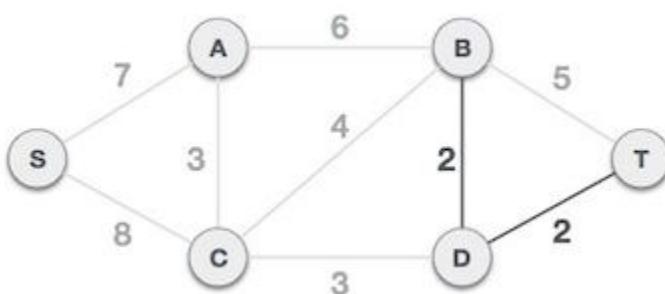
Step 2 - Arrange all edges in their increasing order of weight

Next step is to create a set of edges & weight and arrange them in ascending order of weightage (cost).

| B, D | D, T | A, C | C, D | C, B | B, T | A, B | S, A | S, C |
|------|------|------|------|------|------|------|------|------|
| 2    | 2    | 3    | 3    | 4    | 5    | 6    | 7    | 8    |

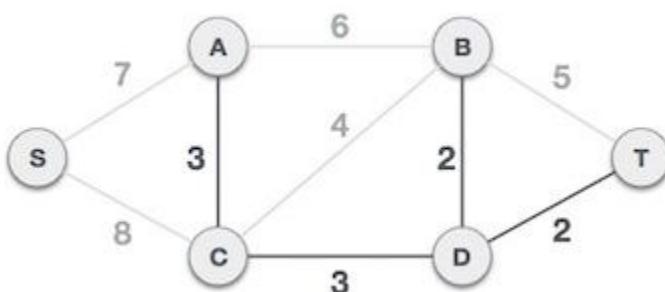
Step 3 - Add the edge which has least weightage

Now we start adding edges to graph beginning from the one which has least weight. At all time, we shall keep checking that the spanning properties are remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in graph.

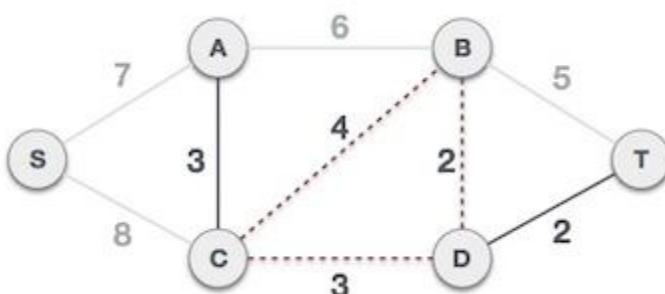


The least cost is 2 and edges involved are B,D and D,T so we add them. Adding them does not violate spanning tree properties so we continue to our next edge selection.

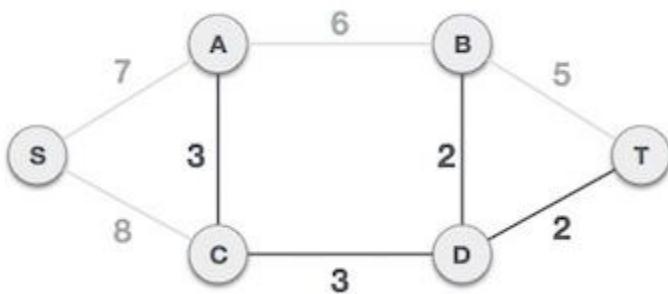
Next cost is 3, and associated edges are A,C and C,D. So we add them –



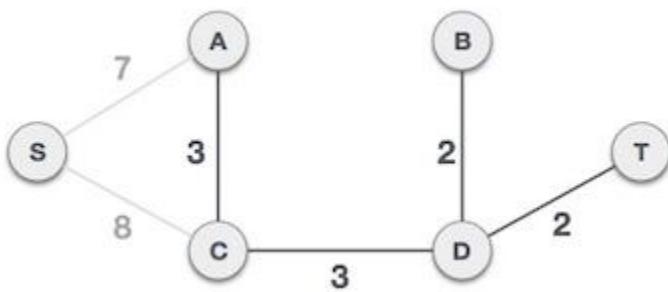
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph



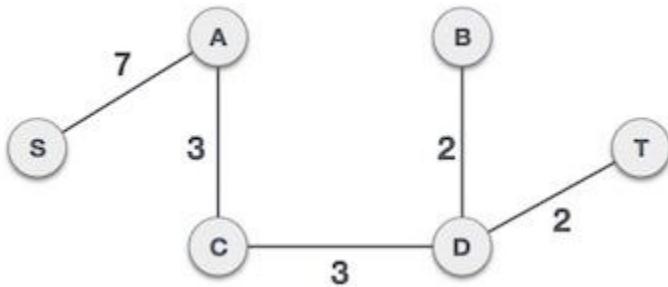
...and we ignore it. And in the process we shall ignore/avoid all edges which create circuit.



We observe that edges with cost 5 and 6 also create circuits and we ignore them and move on.



Now we are left with only one node to be added. Between two least cost edges available 7, 8 we shall add the edge with cost 7.



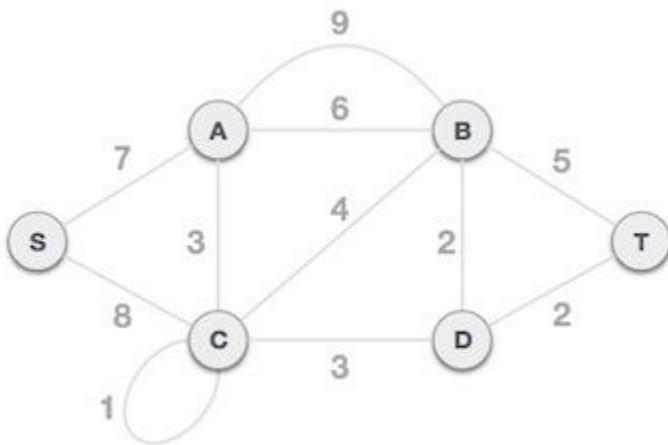
By adding edge S,A we have included all the nodes of the graph and we have minimum cost spanning tree.

### Algorithm

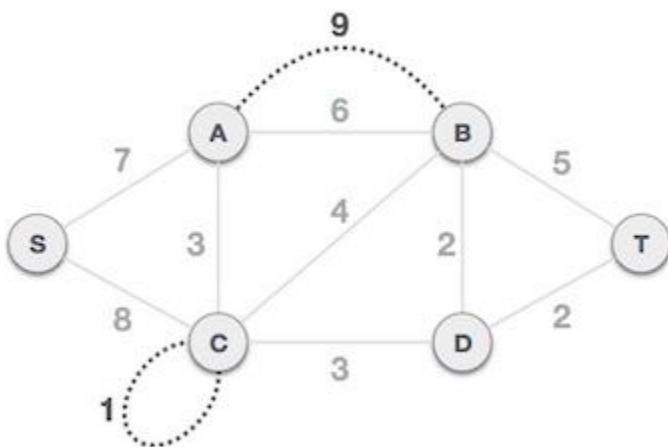
1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far.  
If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are ( $V-1$ ) edges in the spanning tree.

### Prims Algorithm

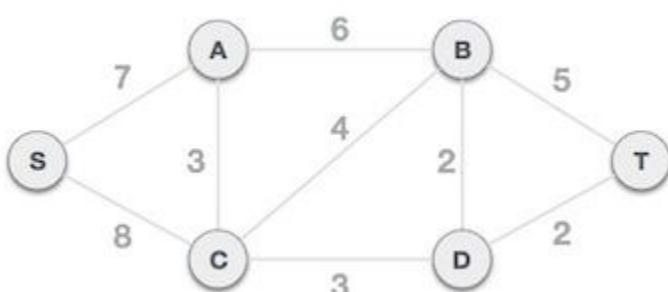
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses greedy approach. Prim's algorithm shares similarity with **shortest path first** algorithms. Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph. To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –



#### Step 1 - Remove all loops & Parallel Edges



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

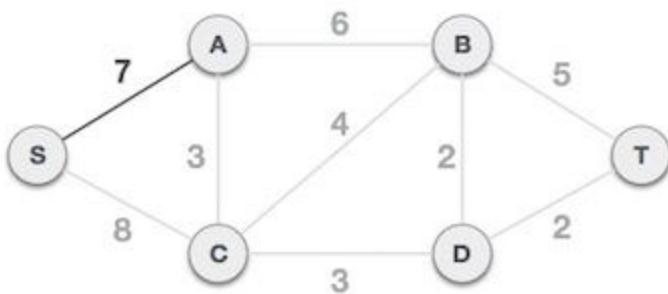


#### Step 2 - Choose any arbitrary node as root node

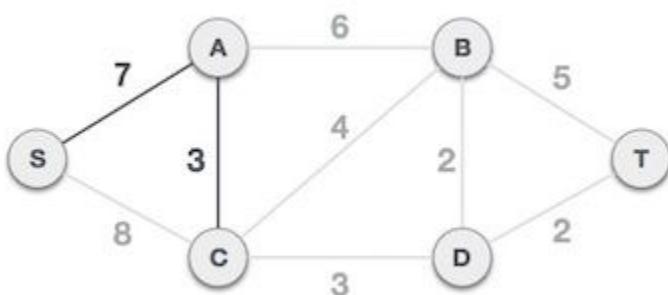
In this case, we choose **S** node as root node of Prim's spanning tree. This node is arbitrarily chosen so any node can be root node. One may wonder why can any video be a root node, so the answer is, in spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge which will join it to the rest of the tree.

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

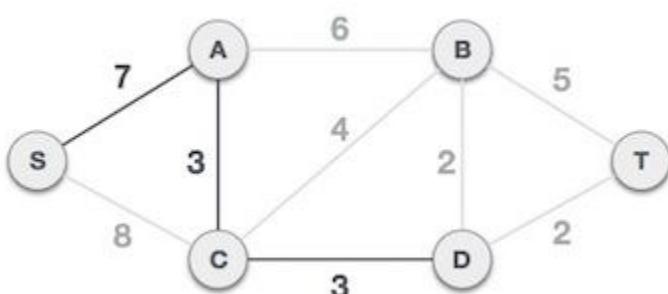
After choosing root node **S**, we see that **S,A** and **S,C** are two edges with weight 7 and 8 respectively. And we choose **S,A** edge as it is lesser than the other.



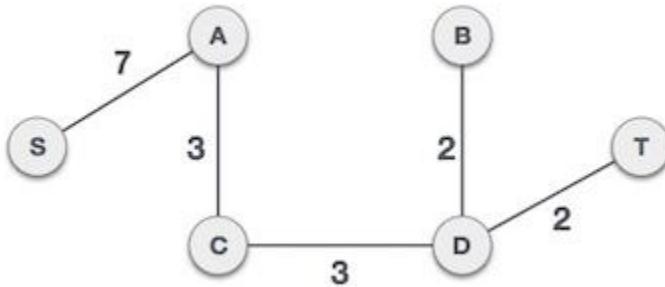
Now, the tree **S-7-A** is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



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



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



We may find that the output spanning tree of the same graph using two different algorithms is same.

### Algorithm

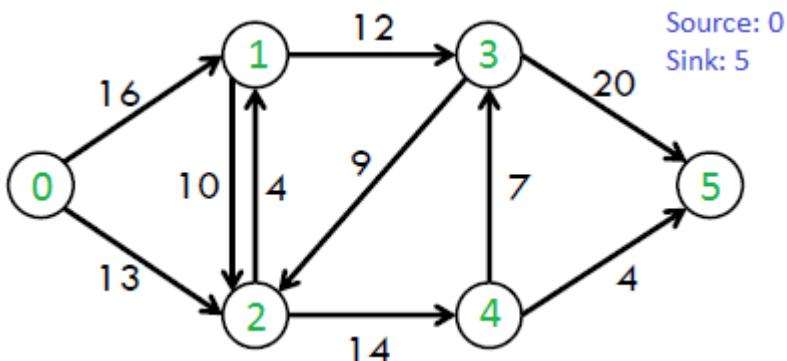
- 1) Create a set **mstSet** that keeps track of vertices already included in MST.
- 2) Assign a key value to all vertices in the input graph. Initialize all key values as **INFINITE**. Assign key value as 0 for the first vertex so that it is picked first.
- 3) While **mstSet** doesn't include all vertices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

### Network Flow Problem

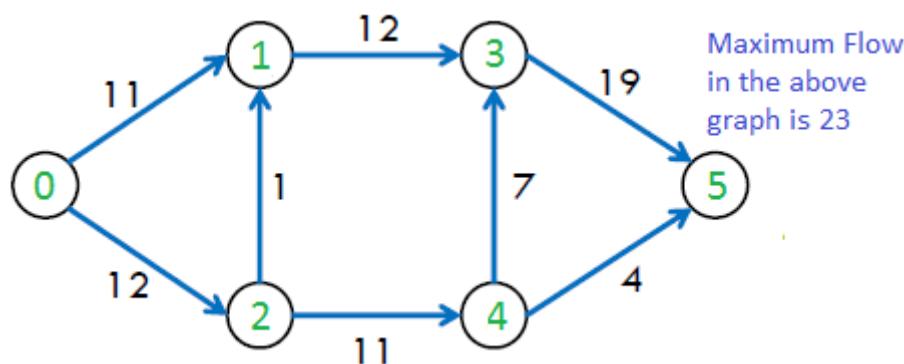
Given a graph which represents a flow network where every edge has a capacity. Also given two vertices *source's'* and *sink 't'* in the graph, find the maximum possible flow from **s** to **t** with following constraints:

- a) **Flow on an edge doesn't exceed the given capacity of the edge.**
- b) **Incoming flow is equal to outgoing flow for every vertex except s and t.**

For example, consider the following graph from CLRS book.



The maximum possible flow in the above graph is 23.



### Augmenting Path

The idea behind the algorithm is as follows: as long as there is a path from the source (start node) to the sink (end node), with available capacity on all edges in the path, we send flow along one of the paths. Then we find another path, and so on. A path with available capacity is called an augmenting path.

### Ford-Fulkerson Algorithm

The following is simple idea of Ford-Fulkerson algorithm:

- 1) Start with initial flow as 0.
- 2) While there is a augmenting path from source to sink.  
Add this path-flow to flow.
- 3) Return flow.

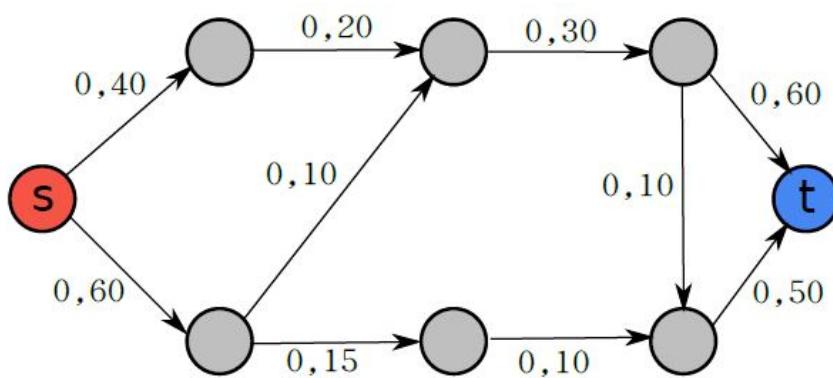
**Example**

Figure 3: A network with zero flow.

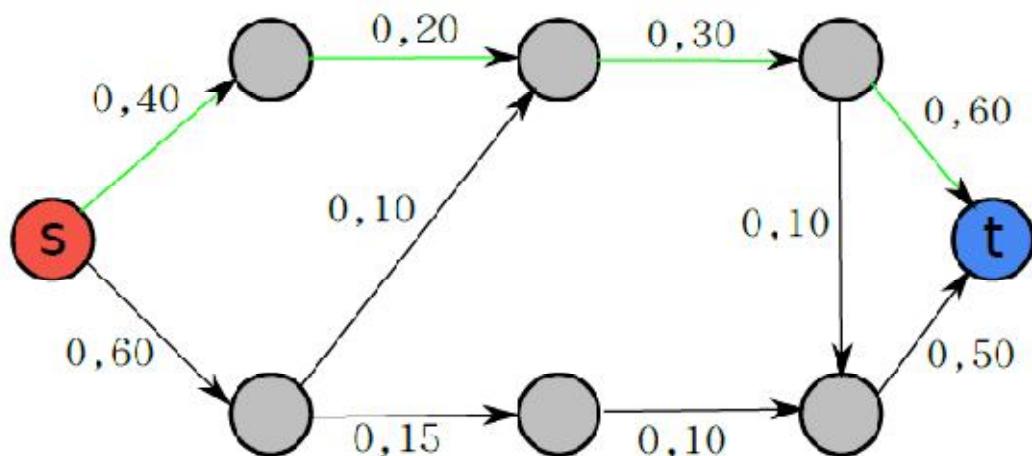


Figure 4: Find an augmenting path.

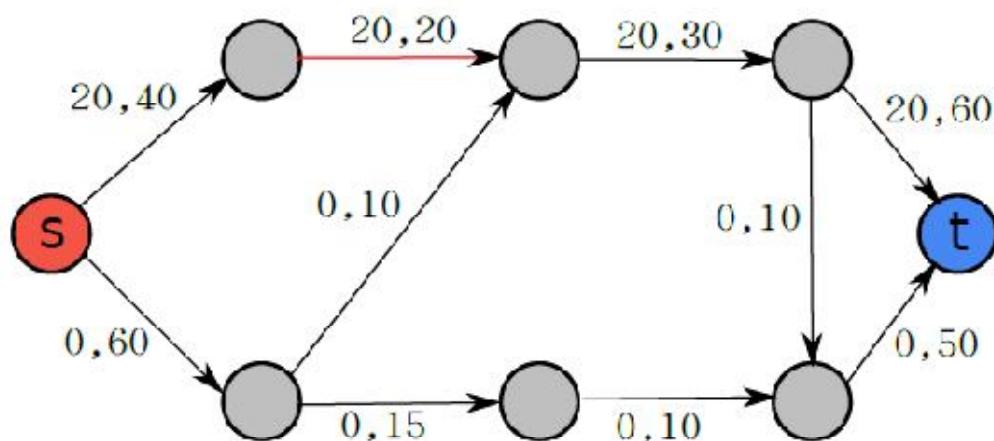


Figure 5: Send flow along the path.

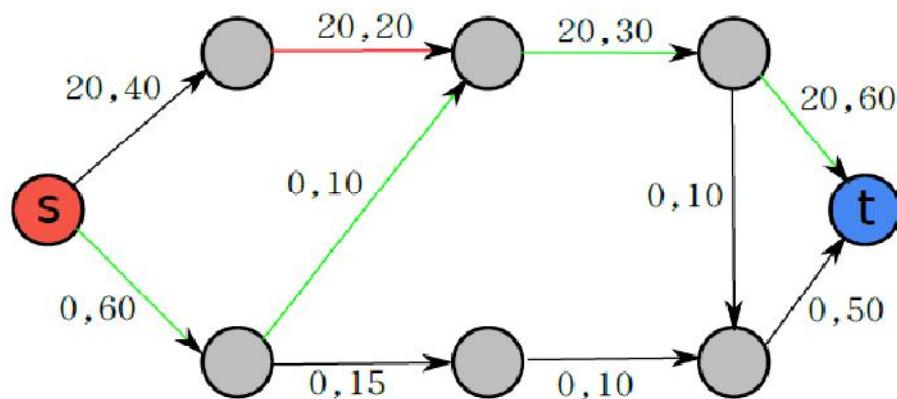


Figure 6: Find an augmenting path.

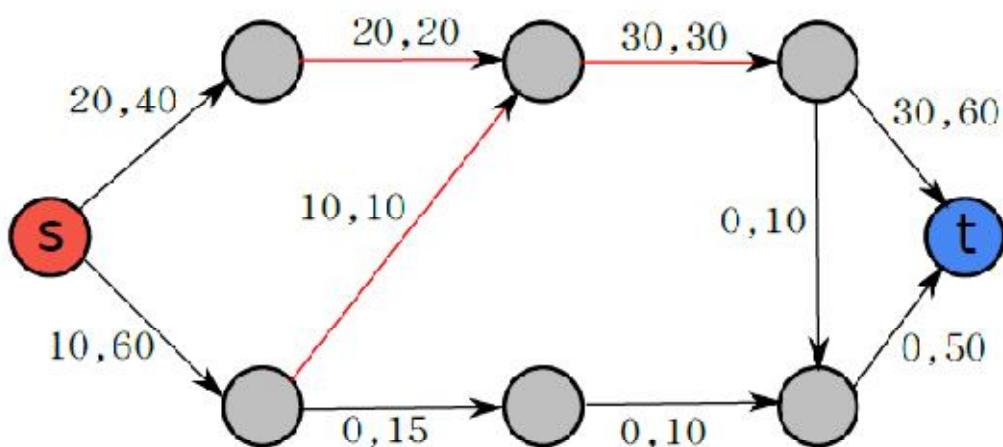


Figure 7: Send flow along the path.

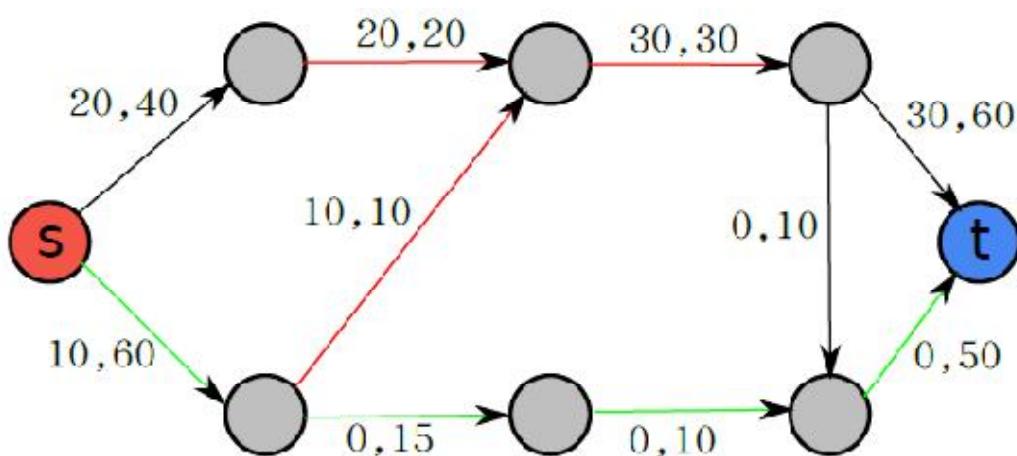


Figure 8: Find an augmenting path.

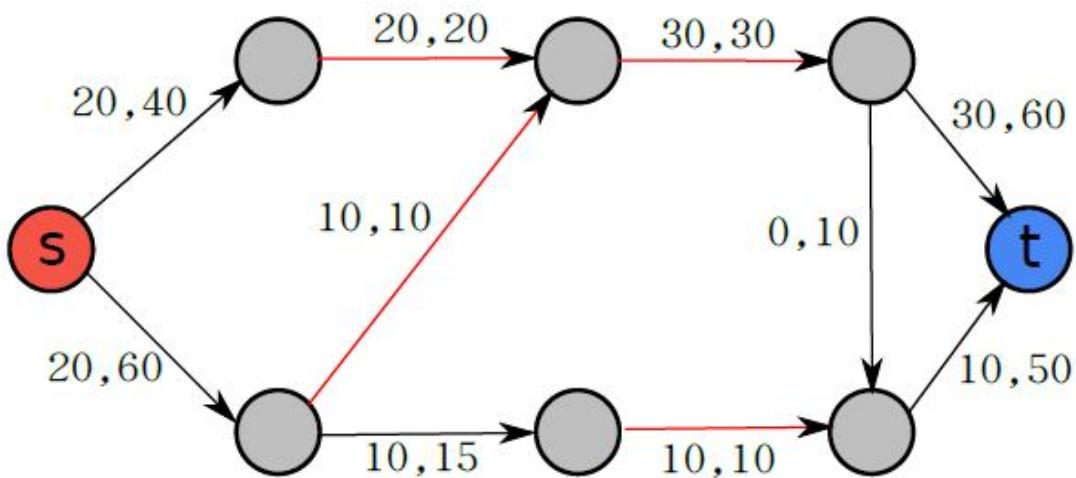


Figure 9: Send flow along the path.

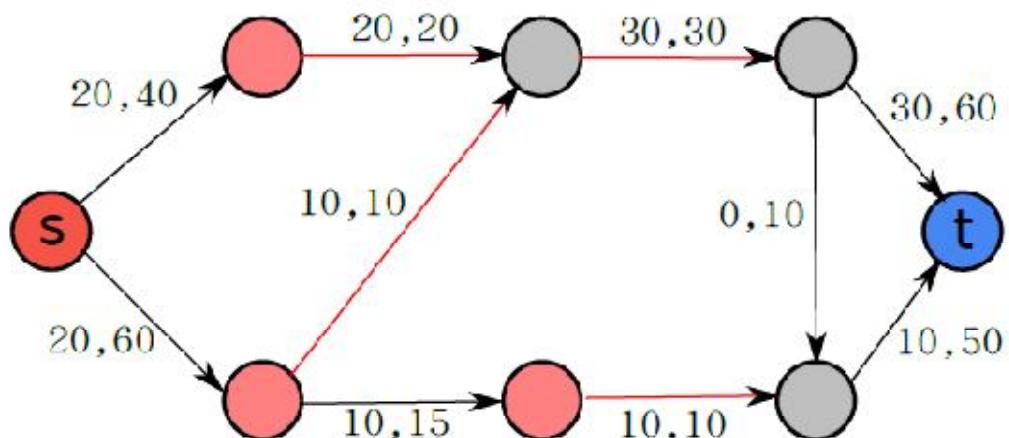


Figure 10: No more augmenting paths can be found. Label all vertices that can be reached via non-saturated edges as "belonging to the source".

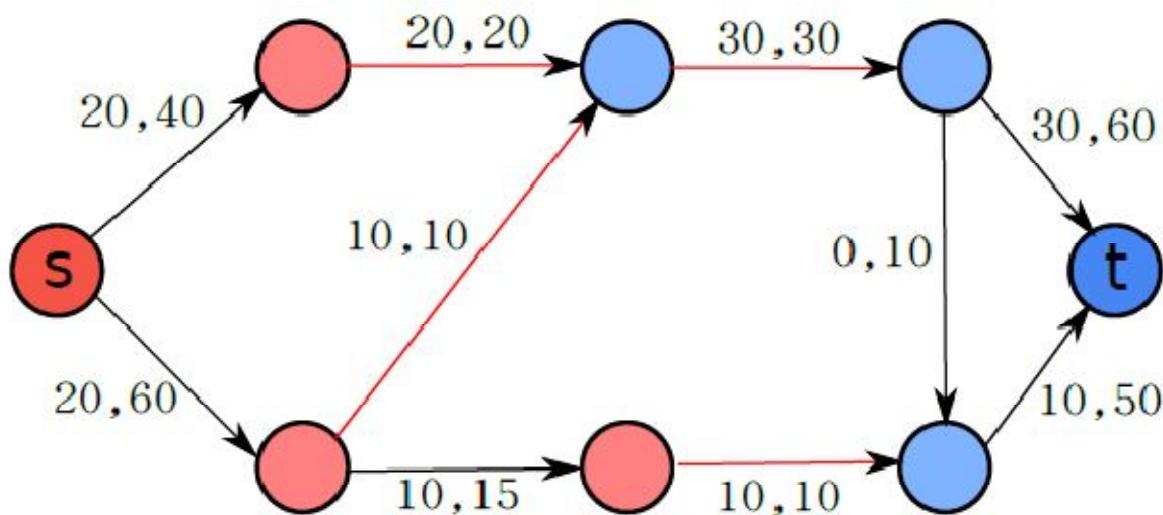


Figure 11: Label all remaining vertices as “belonging to the sink”.

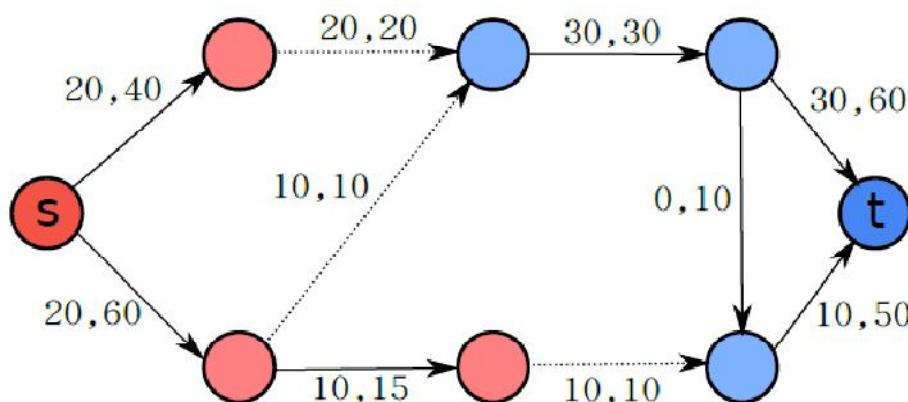


Figure 12: The edges on the boundary of this labeling form a minimum s-t cut.

## Shortest Path Algorithm: Dijkstra

### Objective:

We start with a weighted, directed graph with a weight function  $w$  defined on its edges. The objective is to find the shortest path to every vertices from the start vertex  $S$ .

### Graph Initialization

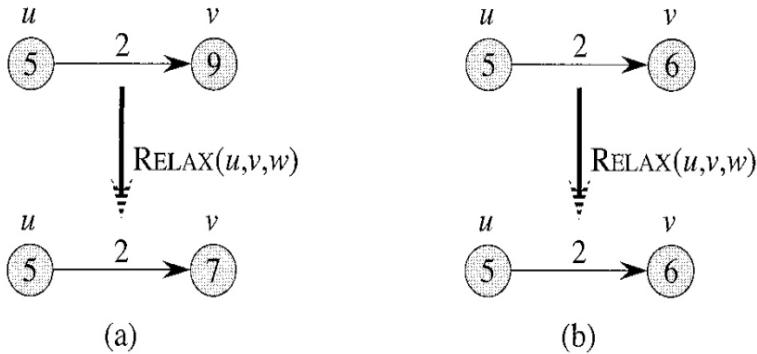
We will generally consider a distinguished vertex, called the **source**, and usually denoted by  $S$ .

```

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

```

## Relaxing an Edge



**Figure 24.3** Relaxing an edge  $(u, v)$  with weight  $w(u, v) = 2$ . The shortest-path estimate of each vertex appears within the vertex. **(a)** Because  $v.d > u.d + w(u, v)$  prior to relaxation, the value of  $v.d$  decreases. **(b)** Here,  $v.d \leq u.d + w(u, v)$  before relaxing the edge, and so the relaxation step leaves  $v.d$  unchanged.

$\text{RELAX}(u, v, w)$

```

1   if  $v.d > u.d + w(u, v)$ 
2        $v.d = u.d + w(u, v)$ 
3        $v.\pi = u$ 

```

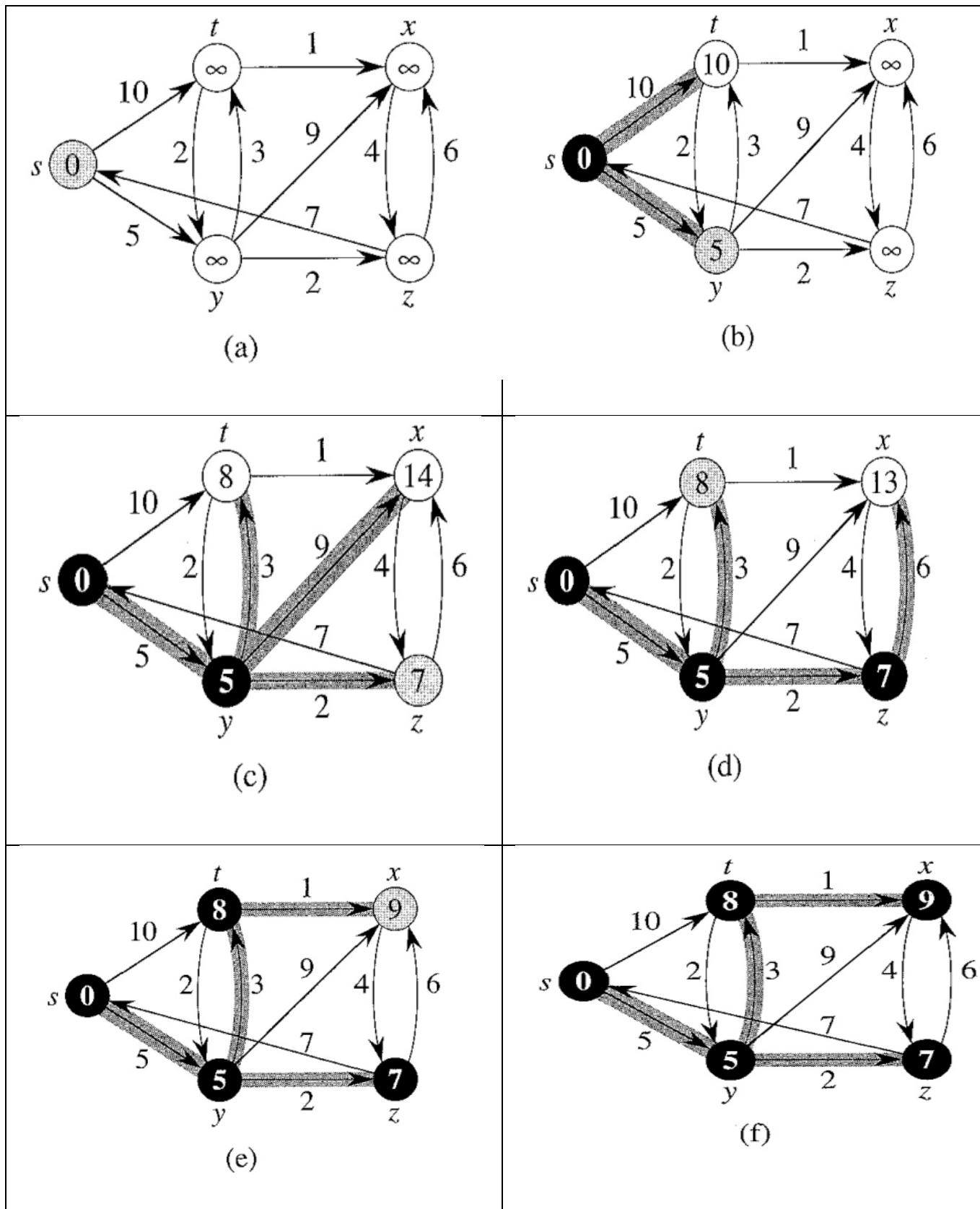
## Algorithm

$\text{DIJKSTRA}(G, w, s)$

```

1   INITIALIZE-SINGLE-SOURCE( $G, s$ )
2    $S = \emptyset$ 
3    $Q = G.V$ 
4   while  $Q \neq \emptyset$ 
5        $u = \text{EXTRACT-MIN}(Q)$ 
6        $S = S \cup \{u\}$ 
7       for each vertex  $v \in G.\text{Adj}[u]$ 
8            $\text{RELAX}(u, v, w)$ 

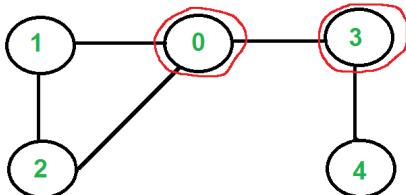
```

**Example**

## Bi-connected components

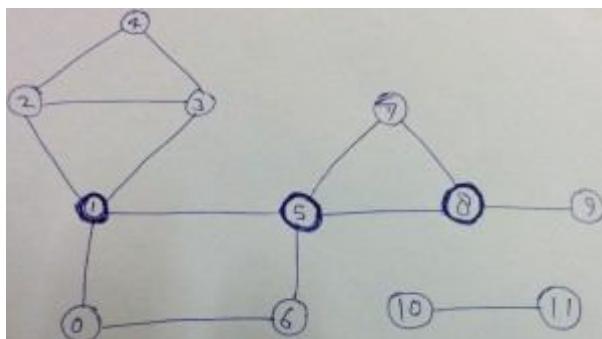
A biconnected component is a maximal biconnected subgraph.

**Biconnected Graph:** A biconnected graph is a connected graph on two or more vertices having no articulation vertices. A vertex in an undirected connected graph is an articulation point (or cut vertex) iff removing it (and edges through it) disconnects the graph. Articulation points represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more disconnected components.



Articulation points are 0 and 3

To find biconnected component in a graph using algorithm by John Hopcroft and Robert Tarjan.



In above graph, following are the biconnected components:

- 4–2 3–4 3–1 2–3 1–2
- 8–9
- 8–5 7–8 5–7
- 6–0 5–6 1–5 0–1
- 10–11

Idea is to store visited edges in a stack while DFS on a graph and keep looking for Articulation Points (highlighted in above figure). As soon as an Articulation Point u is

found, all edges visited while DFS from node u onwards will form one biconnected component. When DFS completes for one connected component, all edges present in stack will form a biconnected component. If there is no Articulation Point in graph, then graph is biconnected and so there will be one biconnected component which is the graph itself.

## Hashing

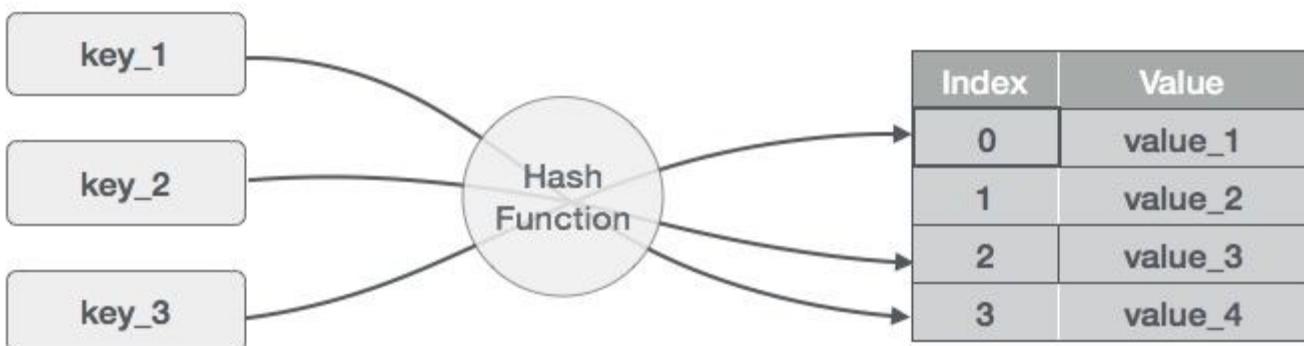
### Hash Table

Hash Table is a data structure which store data in associative manner. In hash table, data is stored in array format where each data value has its own unique index value. Access of data becomes very fast if we know the index of desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of size of data. Hash Table uses array as a storage medium and uses hash technique to generate index where an element is to be inserted or to be located from.

## Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and following items are to be stored. Item are in (key,value) format.



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

| S.n. | Key | Hash            | Array Index |
|------|-----|-----------------|-------------|
| 1    | 1   | $1 \% 20 = 1$   | 1           |
| 2    | 2   | $2 \% 20 = 2$   | 2           |
| 3    | 42  | $42 \% 20 = 2$  | 2           |
| 4    | 4   | $4 \% 20 = 4$   | 4           |
| 5    | 12  | $12 \% 20 = 12$ | 12          |
| 6    | 14  | $14 \% 20 = 14$ | 14          |
| 7    | 17  | $17 \% 20 = 17$ | 17          |
| 8    | 13  | $13 \% 20 = 13$ | 13          |
| 9    | 37  | $37 \% 20 = 17$ | 17          |

### Linear Probing

As we can see, it may happen that the hashing technique used here, creates already used index of the array. In such case, we can search the next empty location in the array by looking into the next cell until we found an empty cell. This technique is called linear probing.

| S.n. | Key | Hash           | Array Index | After Linear Probing, Array Index |
|------|-----|----------------|-------------|-----------------------------------|
| 1    | 1   | $1 \% 20 = 1$  | 1           | 1                                 |
| 2    | 2   | $2 \% 20 = 2$  | 2           | 2                                 |
| 3    | 42  | $42 \% 20 = 2$ | 2           | 3                                 |

|   |    |                 |    |    |
|---|----|-----------------|----|----|
| 4 | 4  | $4 \% 20 = 4$   | 4  | 4  |
| 5 | 12 | $12 \% 20 = 12$ | 12 | 12 |
| 6 | 14 | $14 \% 20 = 14$ | 14 | 14 |
| 7 | 17 | $17 \% 20 = 17$ | 17 | 17 |
| 8 | 13 | $13 \% 20 = 13$ | 13 | 13 |
| 9 | 37 | $37 \% 20 = 17$ | 17 | 18 |

### Basic Operations

Following are basic primary operations of a hashtable which are following.

- **Search** – search an element in a hashtable.
- **Insert** – insert an element in a hashtable.
- **delete** – delete an element from a hashtable.

### DataItem

Define a data item having some data, and key based on which search is to be conducted in hashtable.

```
struct DataItem {
    int data;
    int key;
};
```

### HashMethod

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){
    return key % SIZE;
}
```

### SearchOperation

Whenever an element is to be searched. Compute the hash code of the key passed and locate the element using that hashcode as index in the array. Use linear probing to get element ahead if element not found at computed hash code.

```

struct DataItem *search(int key){
    //get the hash
    int hashIndex = hashCode(key);
    //move in array until an empty
    while(hashArray[hashIndex] != NULL){
        if(hashArray[hashIndex]->key == key)
            return hashArray[hashIndex];
        //go to next cell
        ++hashIndex;
        //wrap around the table
        hashIndex %= SIZE;
    }
    return NULL;
}

```

### Insert Operation

Whenever an element is to be inserted. Compute the hash code of the key passed and locate the index using that hashcode as index in the array. Use linear probing for empty location if an element is found at computed hash code.

```

void insert(int key,int data){
    struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
    item->data = data;
    item->key = key;
    //get the hash
    int hashIndex = hashCode(key);
    //move in array until an empty or deleted cell
    while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1){
        //go to next cell
        ++hashIndex;
        //wrap around the table
        hashIndex %= SIZE;
    }
}

```

```
}

hashArray[hashIndex] = item;

}
```

### Delete Operation

Whenever an element is to be deleted. Compute the hash code of the key passed and locate the index using that hashcode as index in the array. Use linear probing to get element ahead if an element is not found at computed hash code. When found, store a dummy item there to keep performance of hashtable intact.

```
struct DataItem* delete(struct DataItem* item){

    int key = item->key;
    //get the hash
    int hashIndex = hashCode(key);
    //move in array until an empty
    while(hashArray[hashIndex] !=NULL){
        if(hashArray[hashIndex]->key == key){
            struct DataItem* temp = hashArray[hashIndex];
            //assign a dummy item at deleted position
            hashArray[hashIndex] = dummyItem;
            return temp;
        }
        //go to next cell
        ++hashIndex;
        //wrap around the table
        hashIndex %= SIZE;
    }
    return NULL;
}
```

## What is Collision?

Since a hash function gets us a small number for a key which is a big integer or string, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique.

## What are the chances of collisions with large table?

Collisions are very likely even if we have big table to store keys. An important observation is Birthday Paradox. With only 23 persons, the probability that two people have same birthday is 50%.

## How to handle Collisions?

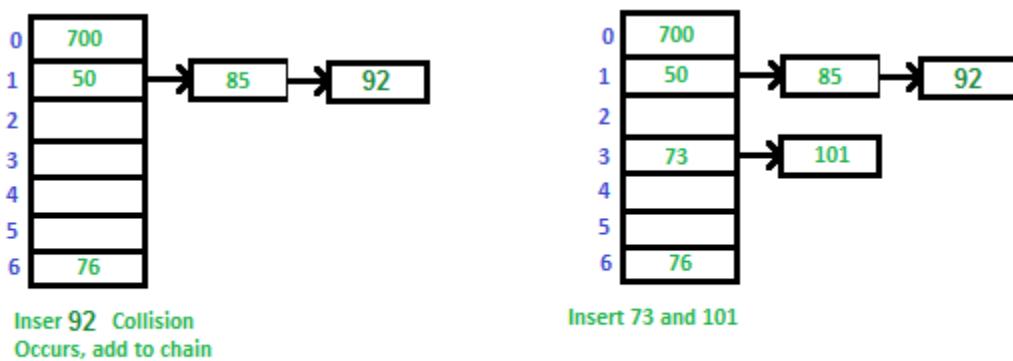
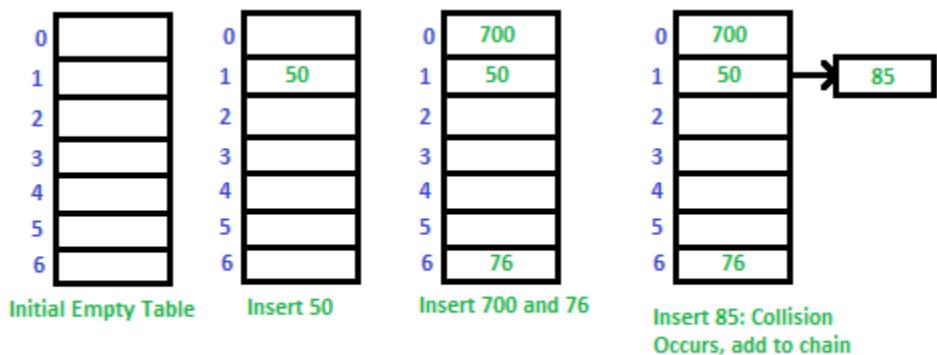
There are mainly two methods to handle collision:

- 1) Separate Chaining
- 2) Open Addressing

### Separate Chaining:

The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let us consider a simple hash function as "key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



### Advantages:

- 1) Simple to implement.
- 2) Hash table never fills up, we can always add more elements to chain.
- 3) Less sensitive to the hash function or load factors.
- 4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

### Disadvantages:

- 1) Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.
- 2) Wastage of Space (Some Parts of hash table are never used)
- 3) If the chain becomes long, then search time can become  $O(n)$  in worst case.
- 4) Uses extra space for links.

## Open Addressing

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of table must be greater than or equal to total number of keys (Note that we can increase table size by copying old data if needed).

**Insert(k):** Keep probing until an empty slot is found. Once an empty slot is found, insert k.

**Search(k):** Keep probing until slot's key doesn't become equal to k or an empty slot is reached.

**Delete(k):** Delete operation is interesting. If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as "deleted".

Insert can insert an item in a deleted slot, but search doesn't stop at a deleted slot.

### Open Addressing is done following ways:

#### a) Linear Probing:

In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also.

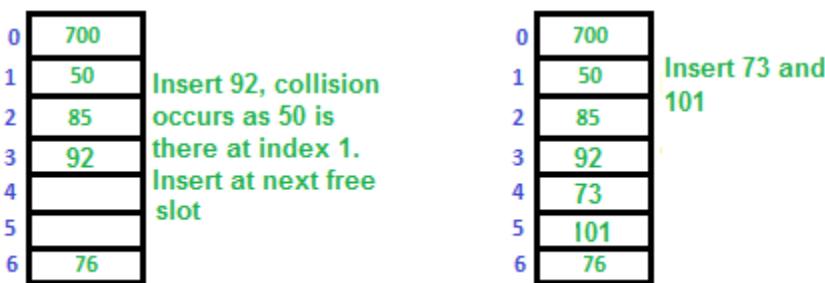
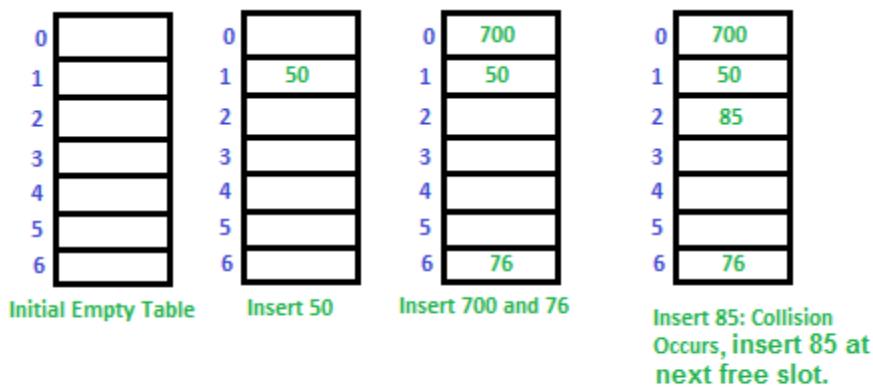
let  $\text{hash}(x)$  be the slot index computed using hash function and  $S$  be the table size

**If slot  $\text{hash}(x) \% S$  is full, then we try  $(\text{hash}(x) + 1) \% S$**

**If  $(\text{hash}(x) + 1) \% S$  is also full, then we try  $(\text{hash}(x) + 2) \% S$**

**If  $(\text{hash}(x) + 2) \% S$  is also full, then we try  $(\text{hash}(x) + 3) \% S$**

Let us consider a simple hash function as "key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



### Clustering:

The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

### b) Quadratic Probing

We look for  $i^2$ 'th slot in  $i$ 'th iteration.

let  $\text{hash}(x)$  be the slot index computed using hash function.

**If slot  $\text{hash}(x) \% S$  is full, then we try  $(\text{hash}(x) + 1*1) \% S$**

**If  $(\text{hash}(x) + 1*1) \% S$  is also full, then we try  $(\text{hash}(x) + 2*2) \% S$**

**If  $(\text{hash}(x) + 2*2) \% S$  is also full, then we try  $(\text{hash}(x) + 3*3) \% S$**

### c) Double Hashing

We use another hash function  $\text{hash2}(x)$  and look for  $i*\text{hash2}(x)$  slot in  $i$ 'th rotation.

let  $\text{hash}(x)$  be the slot index computed using hash function.

**If slot  $\text{hash}(x) \% S$  is full, then we try  $(\text{hash}(x) + 1*\text{hash2}(x)) \% S$**

**If  $(\text{hash}(x) + 1*\text{hash2}(x)) \% S$  is also full, then we try  $(\text{hash}(x) + 2*\text{hash2}(x)) \% S$**

**If  $(\text{hash}(x) + 2*\text{hash2}(x)) \% S$  is also full, then we try  $(\text{hash}(x) + 3*\text{hash2}(x)) \% S$**

**Comparison of above three:**

Linear probing has the best cache performance, but suffers from clustering. One more advantage of Linear probing is easy to compute.

Quadratic probing lies between the two in terms of cache performance and clustering.

Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

**Open Addressing vs. Separate Chaining****Advantages of Chaining:**

- 1) Chaining is Simpler to implement.
- 2) In chaining, Hash table never fills up, we can always add more elements to chain. In open addressing, table may become full.
- 3) Chaining is Less sensitive to the hash function or load factors.
- 4) Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.
- 5) Open addressing requires extra care for to avoid clustering and load factor.

**Advantages of Open Addressing**

- 1) Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.
- 2) Wastage of Space (Some Parts of hash table in chaining are never used). In Open addressing, a slot can be used even if an input doesn't map to it.
- 3) Chaining uses extra space for links.

**Rehashing:**

- It is a closed hashing technique.
- If the table gets too full, then the rehashing method builds new table that is about twice as big and scan down the entire original hash table, comparing the new hash value for each element and inserting it in the new table.
- Rehashing is very expensive since the running time is  $O(N)$ , since there are  $N$  elements to rehash and the table size is roughly  $2N$

Rehashing can be implemented in several ways like

1. Rehash , as soon as the table is half full
2. Rehash only when an insertion fails

Example : Suppose the elements 13, 15, 24, 6 are inserted into an open addressing hash table of size 7 and if linear probing is used when collision occurs.

|   |    |
|---|----|
| 0 | 6  |
| 1 | 15 |
| 2 |    |
| 3 | 24 |
| 4 |    |
| 5 |    |
| 6 | 13 |

If 23 is inserted, the resulting table will be over 70 percent full.

|   |    |
|---|----|
| 0 | 6  |
| 1 | 15 |
| 2 | 23 |
| 3 | 24 |
| 4 |    |
| 5 |    |
| 6 | 13 |

A new table is created. The size of the new table is 17, as this is the first prime number that is twice as large as the old table size.

|    |    |
|----|----|
| 0  |    |
| 1  |    |
| 2  |    |
| 3  |    |
| 4  |    |
| 5  |    |
| 6  | 6  |
| 7  | 23 |
| 8  | 24 |
| 9  |    |
| 10 |    |
| 11 |    |
| 12 |    |
| 13 | 13 |
| 14 |    |
| 15 | 1  |
| 16 |    |

### Advantages

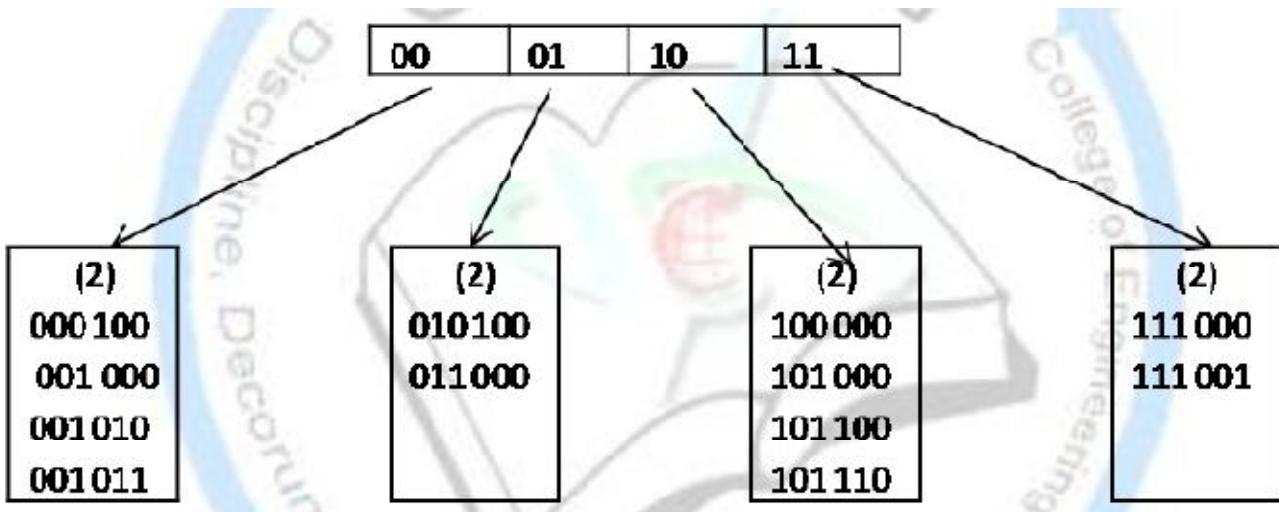
- Programmer doesn't worry about the table size
- Simple to implement

### Extendible Hashing:

- When open addressing or separate hashing is used, collisions could cause several blocks to be examined during a Find operation, even for a well distributed hash table.
- Furthermore , when the table gets too full, an extremely expensive rehashing step must be performed, which requires  $O(N)$  disk accesses.
- These problems can be avoided by using extendible hashing.
- Extendible hashing uses a tree to insert keys into the hash table.

### Example:

- Consider the key consists of several 6 bit integers.
- The root of the “tree” contains 4 pointers determined by the leading 2 bits.
- In each leaf the first 2 bits are identified and indicated in parenthesis.
- D represents the number of bits used by the root(directory)
- The number of entries in the directory is  $2^D$



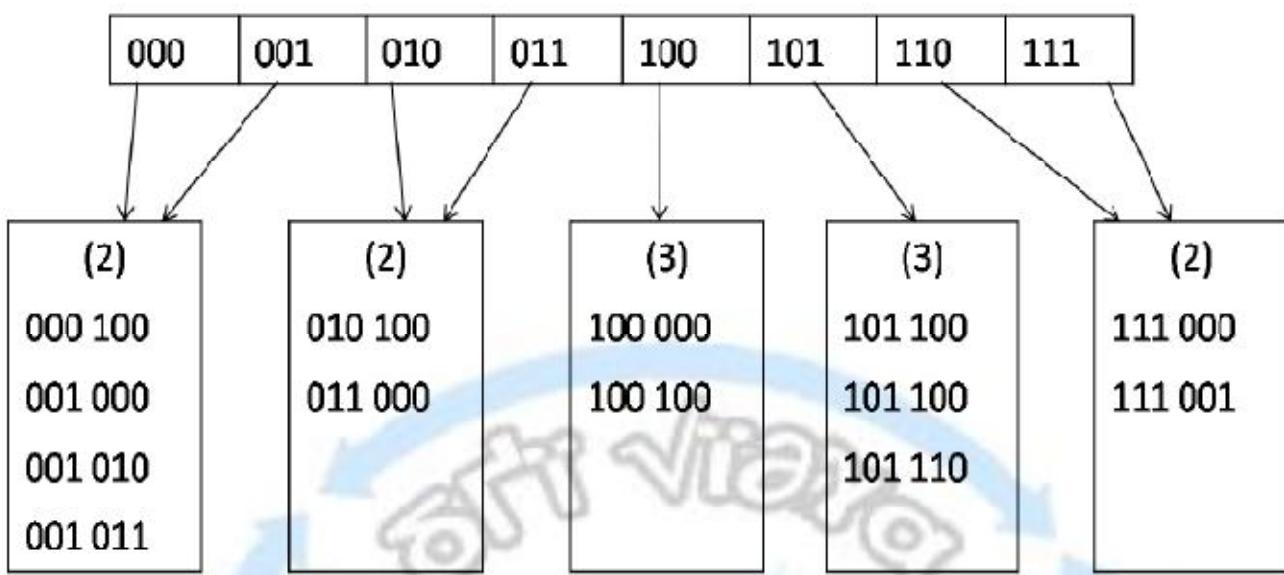
Suppose to insert the key 100100.

This would go to the third leaf but as the third leaf is already full.

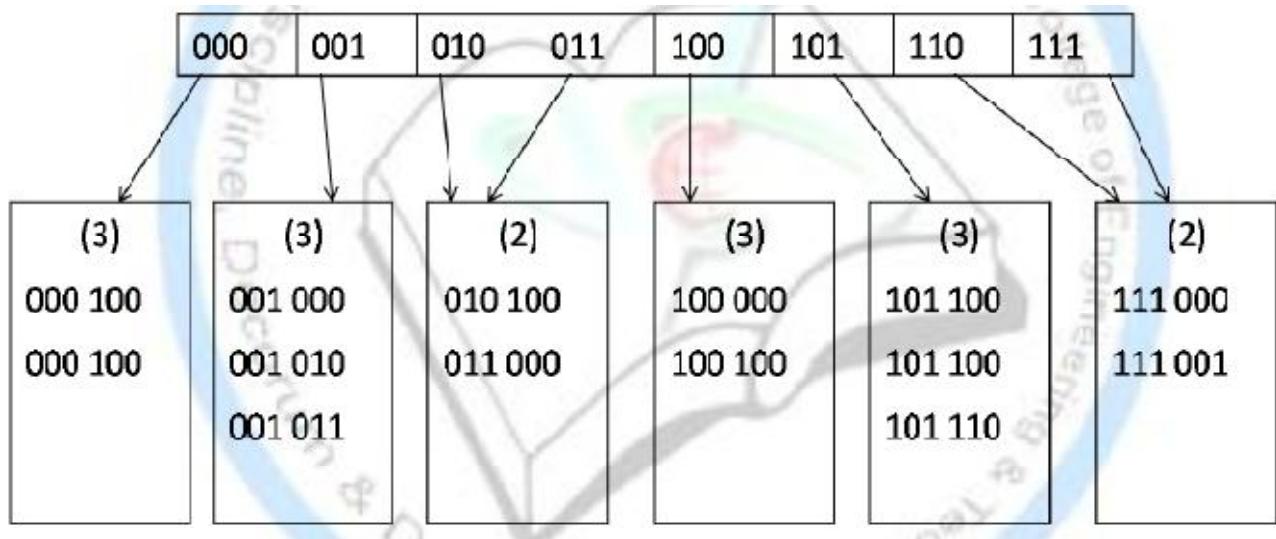
So split this

leaf into two leaves, which are now determined by the first three bits.

Now the directory size is increased to 3.



Similarly if the key 000100 is to be inserted, then the first leaf is split into 2 leaves.



Advantages & Disadvantages:

- Advantages
  - Provides quick access times for insert and find operations on large databases.
- Disadvantages
  - This algorithm does not work if there are more than M duplicates

## 2.5 Asymptotic tight bound

### Theta notation

The function  $f(n) = \Theta(g(n))$  if and only if there exist positive constants  $c_1, c_2, n_0$  such that  $c_1g(n) \leq f(n) \leq c_2g(n), \forall n \geq n_0$ . Theta can be used to denote tight bounds of an algorithm. i.e.,  $g(n)$  is a lower bound as well as an upper bound for  $f(n)$ . Note that  $f(n) = \Theta(g(n))$  if and only if  $f(n) = \Omega(g(n))$  and  $f(n) = O(g(n))$ .

### Examples

1.  $3n + 10^{10}$

$$3n \leq 3n + 10^{10} \leq 4n, \forall n \geq 10^{10}$$

$$\Rightarrow 3n + 10^{10} = \Theta(n)$$

Note that the first inequality captures  $3n + 10^{10} = \Omega(n)$  and the later one captures  $3n + 10^{10} = O(n)$ .

2.  $10n^2 + 4n + 2 = \Theta(n^2)$

$$10n^2 \leq 10n^2 + 4n + 2 \leq 20n^2, \forall n \geq 1, c_1 = 10, c_2 = 20$$

3.  $6(2^n) + n^2 = \Theta(2^n)$

$$6(2^n) \leq 6(2^n) + n^2 \leq 12(2^n), \forall n \geq 1, c_1 = 6, c_2 = 12$$

4.  $2n^2 + n \log n + 1 = \Theta(n^2)$

$$2n^2 \leq 2n^2 + n \log_2 n + 1 \leq 5.n^2, \forall n \geq 2, c_1 = 2, c_2 = 5$$

5.  $n\sqrt{n} + n \log_2(n) + 2 = \Theta(n\sqrt{n})$

$$n\sqrt{n} \leq n\sqrt{n} + n \log_2(n) + 2 \leq 5.n\sqrt{n}, \forall n \geq 2, c_1 = 1, c_2 = 5$$

### Remark:

- $3n + 2 \neq \Theta(1)$ . Reason:  $3n + 2 \neq O(1)$
- $3n + 3 \neq \Theta(n^2)$ . Reason:  $3n + 3 \neq \Omega(n^2)$
- $n^2 \neq \Theta(2^n)$ . Reason:  $n^2 \neq \Omega(2^n)$  Proof: Note that  $f(n) \leq g(n)$  if and only if  $\log(f(n)) \leq \log(g(n))$ . Suppose  $n^2 = \Omega(2^n)$ , then by definition  $n^2 \geq c.2^n$  where  $c$  is a positive constant. Then,  $\log(n^2) \geq \log(c.2^n)$ , and  $2\log(n) \geq n\log(2)$ , which is a contradiction.

## 3 Properties of Asymptotic notation

### 1. Reflexivity

$$f(n) = O(f(n))$$

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

$$f(n) = \theta(f(n))$$

### 2. Symmetry

$$f(n) = \theta(g(n)) \text{ if and only if } g(n) = \theta(f(n))$$

### Proof:

**Necessary part:**  $f(n) = \theta(g(n)) \Rightarrow g(n) = \theta(f(n))$

By the definition of  $\theta$ , there exists positive constants  $c_1, c_2, n_o$  such that  $c_1.g(n) \leq f(n) \leq c_2.g(n)$  for all  $n \geq n_o$

$$\Rightarrow g(n) \leq \frac{1}{c_1}.f(n) \text{ and } g(n) \geq \frac{1}{c_2}.f(n)$$

$$\Rightarrow \frac{1}{c_2}f(n) \leq g(n) \leq \frac{1}{c_1}f(n)$$

Since  $c_1$  and  $c_2$  are positive constants,  $\frac{1}{c_1}$  and  $\frac{1}{c_2}$  are well defined. Therefore, by the definition of  $\theta$ ,  $g(n) = \theta(f(n))$

**Sufficiency part:**  $g(n) = \theta(f(n)) \Rightarrow f(n) = \theta(g(n))$

By the definition of  $\theta$ , there exists positive constants  $c_1, c_2, n_o$  such that  $c_1.f(n) \leq g(n) \leq c_2.f(n)$  for all  $n \geq n_o$

$$\begin{aligned}\Rightarrow f(n) &\leq \frac{1}{c_1}.g(n) \text{ and } f(n) \geq \frac{1}{c_2}.g(n) \\ \Rightarrow \frac{1}{c_2}.g(n) &\leq f(n) \leq \frac{1}{c_1}.g(n)\end{aligned}$$

By the definition of  $\theta$ ,  $f(n) = \theta(g(n))$

This completes the proof of Symmetry property.

### 3. Transitivity

$f(n) = O(g(n))$  and  $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$

**Proof:**

$$f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$

By the definition of Big-Oh( $O$ ), there exists positive constants  $c, n_o$  such that  $f(n) \leq c.g(n)$  for all  $n \geq n_o$

$$\begin{aligned}\Rightarrow f(n) &\leq c_1.g(n) \\ \Rightarrow g(n) &\leq c_2.h(n) \\ \Rightarrow f(n) &\leq c_1.c_2.h(n) \\ \Rightarrow f(n) &\leq c.h(n), \text{ where, } c = c_1.c_2\end{aligned}$$

By the definition,  $f(n) = O(h(n))$  **Note :** Theta( $\Theta$ ) and Omega( $\Omega$ ) also satisfies Transitivity Property.

### 4. Transpose Symmetry

$f(n) = O(g(n))$  if and only if  $g(n) = \Omega(f(n))$

**Proof:**

**Necessity:**  $f(n) = O(g(n)) \Rightarrow g(n) = \Omega(f(n))$

By the definition of Big-Oh ( $O$ )

$$\Rightarrow f(n) \leq c.g(n) \quad \text{for some positive constant } c$$

$$\Rightarrow g(n) \geq \frac{1}{c}f(n)$$

By the definition of Omega ( $\Omega$ ) ,  $g(n) = \Omega(f(n))$

**Sufficiency:**  $g(n) = \Omega(f(n)) \Rightarrow f(n) = O(g(n))$

By the definition of Omega ( $\Omega$ ),  $\quad \text{for some positive constant } c$

$$\Rightarrow g(n) \geq c.f(n)$$

$$\Rightarrow f(n) \leq \frac{1}{c}g(n)$$

By the definition of Big-Oh( $O$ ) ,  $f(n) = O(g(n))$

Therefore, Transpose Symmetry is proved.

#### 4 Some more Observations on Asymptotic Notation

**Lemma 1.** Let  $f(n)$  and  $g(n)$  be two asymptotic non-negative functions.

Then,  $\max(f(n), g(n)) = \theta(f(n) + g(n))$

*Proof.* Without loss of generality, assume  $f(n) \leq g(n)$ ,  $\Rightarrow \max(f(n), g(n)) = g(n)$

Consider,  $g(n) \leq \max(f(n), g(n)) \leq g(n)$

$$\Rightarrow g(n) \leq \max(f(n), g(n)) \leq f(n) + g(n)$$

$$\Rightarrow \frac{1}{2}g(n) + \frac{1}{2}g(n) \leq \max(f(n), g(n)) \leq f(n) + g(n)$$

From what we assumed, we can write

$$\Rightarrow \frac{1}{2}f(n) + \frac{1}{2}g(n) \leq \max(f(n), g(n)) \leq f(n) + g(n)$$

$$\Rightarrow \frac{1}{2}(f(n) + g(n)) \leq \max(f(n), g(n)) \leq f(n) + g(n)$$

By the definition of  $\theta$  ,

$$\max(f(n), g(n)) = \theta(f(n) + g(n))$$

**Lemma 2.** For two asymptotic functions  $f(n)$  and  $g(n)$ ,  $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$

*Proof.* Without loss of generality, assume  $f(n) \leq g(n)$

$$\Rightarrow O(f(n)) + O(g(n)) = c_1 f(n) + c_2 g(n)$$

From what we assumed, we can write

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

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

$$\leq c g(n)$$

$$\leq c \max(f(n), g(n))$$

By the definition of Big-Oh(O),

$$O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$$

### Remarks :

1. If  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c, c \in \mathbb{R}^+$  then  $f(n) = \theta(g(n))$
2. If  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq c, c \in \mathbb{R}$  (c can be 0) then  $f(n) = O(g(n))$
3. If  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ , then  $f(n) = O(g(n))$  and  $g(n) \neq O(f(n))$
4. If  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \geq c, c \in \mathbb{R}$  (c can be  $\infty$ ) then  $f(n) = \Omega(g(n))$
5. If  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$ , then  $f(n) = \Omega(g(n))$  and  $g(n) \neq \Omega(f(n))$
6. **L'Hôpital Rule :** If  $f(n)$  and  $g(n)$  are both differentiable with derivatives  $f'(n)$  and  $g'(n)$ , respectively, and if  $\lim_{n \rightarrow \infty} f(n) = \lim_{n \rightarrow \infty} g(n) = \infty$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{f'(n)}{g'(n)}$$

whenever the limit on the right exists.

- Remark 1 characterizes tight bounds ( $\theta$ ).
- Remark 2 characterizes all upper bounds ( $O$ ).
- Remark 3 characterizes loose upper bounds (Little Oh).
- Remark 4 characterizes all lower bounds ( $\Omega$ ).
- Remark 5 characterizes loose lower bounds (Little omega).

**Lemma 3.** Show that  $\log n = O(\sqrt{n})$ , however,  $\sqrt{n} \neq O(\log n)$

*Proof.*

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{\log n}{\sqrt{n}}$$

Applying L'Hôpital Rule,

$$\begin{aligned}\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} &= \lim_{n \rightarrow \infty} \frac{\frac{1}{n}}{\frac{1}{2} \cdot n^{-\frac{1}{2}}} \\ &= \lim_{n \rightarrow \infty} \frac{2}{\sqrt{n}} = 0\end{aligned}$$

From Remark 3,  $f(n) = O(g(n)) \Rightarrow \log n = O(\sqrt{n})$ .

Proof for  $\sqrt{n} \neq O(\log n)$

$$\begin{aligned}\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} &= \lim_{n \rightarrow \infty} \frac{\sqrt{n}}{\log n} \\ \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} &= \lim_{n \rightarrow \infty} \frac{\frac{1}{2} \cdot n^{-\frac{1}{2}}}{\frac{1}{n}} \\ &= \lim_{n \rightarrow \infty} \frac{\sqrt{n}}{2} = \infty\end{aligned}$$

From Remark 3,  $f(n) = \Omega(g(n)) \Rightarrow \sqrt{n} = \Omega(\log n) \Rightarrow \sqrt{n} \neq O(\log n)$

**Note:** There are asymptotic functions which can not be compared using any of the above notation. For example, the following two functions  $f(n)$  and  $g(n)$  are such that  $f(n) \neq O(g(n))$  and  $g(n) \neq O(f(n))$

- $f(n) = n$  and  $g(n) = n^{1+\sin(n)}$
- $f(n) = n\cos^2(n)$  and  $g(n) = n\sin^2(n)$

| function      | $n = 0$ | $n = \frac{\pi}{2}$ | $n = \pi$ | $n = \frac{3\pi}{2}$ | $n = 2\pi$ | $n = \frac{5\pi}{2}$ |
|---------------|---------|---------------------|-----------|----------------------|------------|----------------------|
| $\sin(n)$     | 0       | 1                   | 0         | -1                   | 0          | 1                    |
| $1 + \sin(n)$ | 1       | 2                   | 1         | 0                    | 1          | 2                    |
| $\cos(n)$     | 1       | 0                   | -1        | 0                    | 1          | 0                    |
| $\cos^2(n)$   | 1       | 0                   | 1         | 0                    | 1          | 0                    |
| $\sin^2(n)$   | 0       | 1                   | 0         | 1                    | 0          | 1                    |

It can be observed that neither  $n = O(n^{1+\sin(n)})$  nor  $n = \Omega(n^{1+\sin(n)})$  as  $n^{1+\sin(n)}$  is a periodic function that oscillates between  $n^0$  and  $n^2$ . Similarly,  $n\cos^2(n) \neq O(n\sin^2(n))$ , and  $n\cos^2(n) \neq \Omega(n\sin^2(n))$  as both the functions are periodic that oscillates with a phase shift of  $\frac{\pi}{2}$ . Note that if  $g(n) = n^{2+\sin(n)}$ , then  $f(n)$  and  $g(n)$  are asymptotically comparable.

**Acknowledgements:** Lecture contents presented in this module and subsequent modules are based on the text books mentioned at the reference and most importantly, author has greatly learnt from lectures by algorithm exponents affiliated to IIT Madras/IMSc; Prof C. Pandu Rangan, Prof N.S.Narayanaswamy, Prof Venkatesh Raman, and Prof Anurag Mittal. Author sincerely acknowledges all of them. Special thanks to Teaching Assistants Mr.Renjith.P and Ms.Dhanalakshmi.S for their sincere and dedicated effort and making this scribe possible. Author has benefited a lot by teaching this course to senior undergraduate students and junior undergraduate students who have also contributed to this scribe in many ways. Author sincerely thank all of them.

**References:**

1. E.Horowitz, S.Sahni, S.Rajasekaran, Fundamentals of Computer Algorithms, Galgotia Publications.
2. T.H. Cormen, C.E. Leiserson, R.L.Rivest, C.Stein, Introduction to Algorithms, PHI.
3. Sara Baase, A.V.Gelder, Computer Algorithms, Pearson.

# Master Theorem - Examples

Master's Theorem is a popular method for solving the recurrence relations.

Master's theorem solves recurrence relations of the form-

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

## **Master's Theorem**

Here,  $a \geq 1$ ,  $b > 1$ ,  $k \geq 0$  and  $p$  is a real number.

## Master Theorem Cases-

To solve recurrence relations using Master's theorem, we compare  $a$  with  $b^k$ .

Then, we follow the following cases-

### Case-01:

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

### Case-02:

If  $a = b^k$  and

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

### Case-03:

If  $a < b^k$  and

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

1) Solve the following recurrence relation using Master's theorem-

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

## Solution

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

Then, we have-

$$a = 3$$

$$b = 2$$

$$k = 2$$

$$p = 0$$

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

Clearly,  $a < b^k$ .

So, we follow case-03.

Since  $p = 0$ , so we have-

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

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

Thus,

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

**2) Solve the following recurrence relation using Master's theorem-**

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

## **Solution-**

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

Then, we have-

$$a = 2$$

$$b = 2$$

$$k = 1$$

$$p = 1$$

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

Clearly,  $a = b^k$ .

So, we follow case-02.

Since  $p = 1$ , so we have-

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

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

Thus,

$$\boxed{T(n) = \theta(n \log^2 n)}$$

**3) Solve the following recurrence relation using Master's theorem**

$$T(n) = 2T(n/4) + n^{0.51}$$

## Solution-

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

Then, we have-

$$a = 2$$

$$b = 4$$

$$k = 0.51$$

$$p = 0$$

Now,  $a = 2$  and  $b^k = 4^{0.51} = 2.0279$ .

Clearly,  $a < b^k$ .

So, we follow case-03.

Since  $p = 0$ , so we have-

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

$$T(n) = \theta(n^{0.51\log^0 n})$$

Thus,

$$\boxed{T(n) = \theta(n^{0.51})}$$

4) Solve the following recurrence relation using Master's theorem-

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

## Solution-

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

Then, we have-

$$a = \sqrt{2}$$

$$b = 2$$

$$k = 0$$

$$p = 1$$

Now,  $a = \sqrt{2} = 1.414$  and  $b^k = 2^0 = 1$ .

Clearly,  $a > b^k$ .

So, we follow case-01.

So, we have-

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

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

$$T(n) = \theta(n^{1/2})$$

Thus,

$$\boxed{T(n) = \theta(\sqrt{n})}$$

**5) Solve the following recurrence relation using Master's theorem-**

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

## **Solution-**

The given recurrence relation does not correspond to the general form of Master's theorem.

- So, it can not be solved using Master's theorem.

### **6) Solve the following recurrence relation using Master's theorem-**

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

## **Solution**

We write the given recurrence relation as  $T(n) = 3T(n/3) + n$ .

- This is because in the general form, we have  $\theta$  for function  $f(n)$  which hides constants in it.
- Now, we can easily apply Master's theorem.

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

Then, we have-

$$a = 3$$

$$b = 3$$

$$k = 1$$

$$p = 0$$

Now,  $a = 3$  and  $b^k = 3^1 = 3$ .

Clearly,  $a = b^k$ .

So, we follow case-02.

Since  $p = 0$ , so we have-

$$T(n) = \theta(n^{\log_b a} \cdot n^0)$$

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

$$T(n) = \theta(n^1 \cdot n^0)$$

Thus,

$$\boxed{T(n) = \theta(n \log n)}$$

### **7) Form a recurrence relation for the following code and solve it using Master's theorem-**

```

A(n)
{
if(n<=1)
return 1;
else
return A(√n);
}

```

## Solution-

- We write a recurrence relation for the given code as  $T(n) = T(\sqrt{n}) + 1$ .
- Here 1 = Constant time taken for comparing and returning the value.
- We can not directly apply Master's Theorem on this recurrence relation.
- This is because it does not correspond to the general form of Master's theorem.
- However, we can modify and bring it in the general form to apply Master's theorem.

Let-

$$n = 2^m \dots\dots (1)$$

Then-

$$T(2^m) = T(2^{m/2}) + 1$$

Now, let  $T(2^m) = S(m)$ , then  $T(2^{m/2}) = S(m/2)$

So, we have-

$$S(m) = S(m/2) + 1$$

Now, we can easily apply Master's Theorem.

We compare the given recurrence relation with  $S(m) = aS(m/b) + \theta(m^k \log^p m)$ .

Then, we have-

$$a = 1$$

$$b = 2$$

$$k = 0$$

$$p = 0$$

Now,  $a = 1$  and  $b^k = 2^0 = 1$ .

Clearly,  $a = b^k$ .

So, we follow case-02.

Since  $p = 0$ , so we have-

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

$$S(m) = \Theta(m^{\log_2 1} \cdot \log^{0+1} m)$$

$$S(m) = \Theta(m^0 \cdot \log^1 m)$$

Thus,

$$S(m) = \Theta(\log m) \dots\dots(2)$$

Now,

- From (1), we have  $n = 2^m$ .
- So,  $\log n = m \log 2$  which implies  $m = \log_2 n$ .

Substituting in (2), we get-

$$S(m) = \Theta(\log \log_2 n)$$

# Design and Analysis of Algorithms (18CSC204J)

## Unit - 1

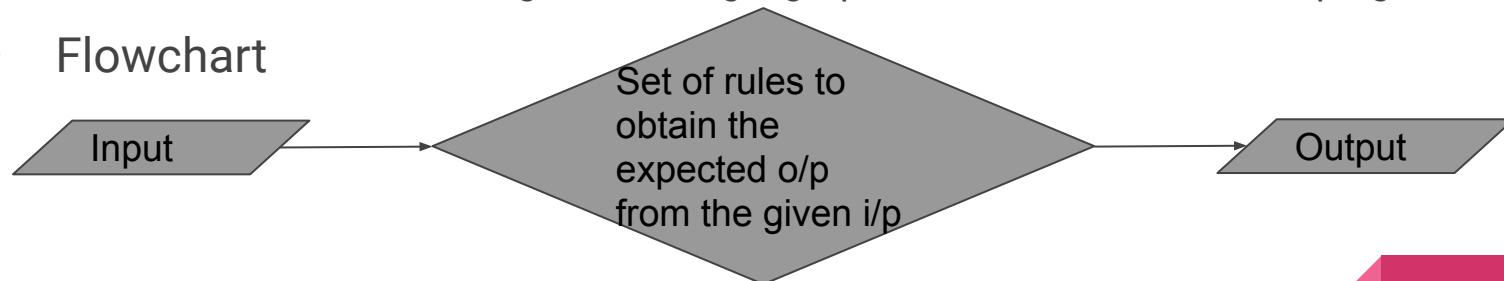
By:  
Aarti Sharma  
Asst Professor  
CSE Dept  
SRMIST Delhi-NCR Campus

# Introduction : Algorithm Design

## ❖ What is Algorithm?

- Set of rules or instructions that step by step
- Defines how a work is to be executed
- In order to get the expected results
- Can be described using natural language, pseudo code, flowchart, or a programming language

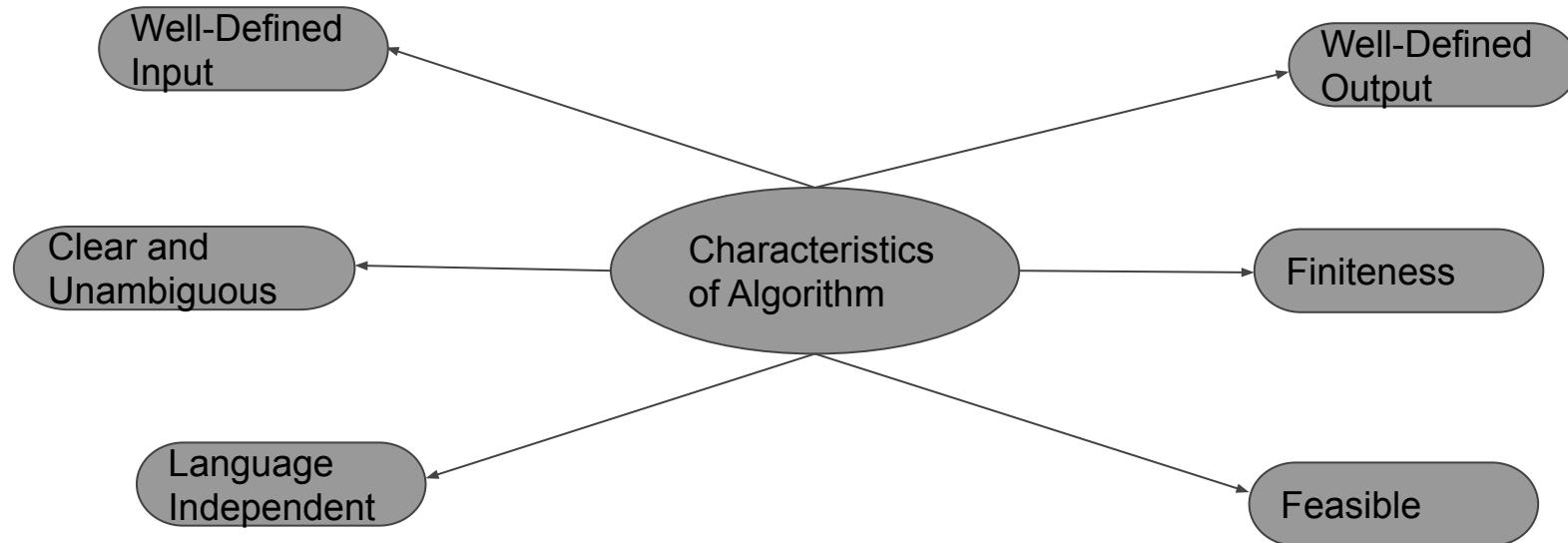
## ❖ Flowchart



# Example of Algorithm

- ❖ How to cook a new dish?
  - Read the instructions
  - Step by step execute the instructions sequentially
  - Result is obtained i.e. the new dish is cooked
- ❖ Write Algorithm for making Tea....
- ❖ Thus, Algorithms are written to help to do a task in programming to obtain the expected output

# Characteristics of Algorithms



# Advantages and Disadvantages of Algorithms

- ❖ **Advantages are:**

- It is easy to understand.
- Algorithm is a step-wise representation of a solution to a given problem.
- In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

- ❖ **Disadvantages are:**

- Writing an algorithm takes a long time so it is time-consuming.
- Branching and Looping statements are difficult to show in Algorithms.

# How to Design an Algorithm?

- ❖ Pre-requisite to write an Algorithm:
  - The **problem** that is to be solved by this algorithm
  - The **constraints** of the problem that must be considered while solving the problem
  - The **input** to be taken to solve the problem
  - The **output** to be expected when the problem is solved
  - The **solution** to this problem, in the given constraints
- ❖ Then the algorithm is written with the help of above parameters such that it solves the problem

# Example of an Algorithm

- ❖ Algorithm to add three numbers and print the sum
  - **Step 1: Fulfilling the pre-requisites**

As discussed above, in order to write an algorithm, its pre-requisites must be fulfilled

- The problem that is to be solved by this algorithm:** Add 3 numbers and print their sum
- The constraints of the problem that must be considered while solving the problem:** The numbers must contain only digits and no other characters
- The input to be taken to solve the problem:** The three numbers to be added
- The output to be expected when the problem the is solved:** The sum of the three numbers taken as the input
- The solution to this problem, in the given constraints:** The solution consists of adding the 3 numbers. It can be done with the help of ‘+’ operator, or bit-wise, or any other method

# Example of an Algorithm (contd...)

## ➤ Step 2: Designing the algorithm

Now let's design the algorithm with the help of above pre-requisites:

**Algorithm to add 3 numbers and print their sum:**

- ❑ START
- ❑ Declare 3 integer variables num1, num2 and num3.
- ❑ Take the three numbers, to be added, as inputs in variables num1, num2, and num3 respectively.
- ❑ Declare an integer variable sum to store the resultant sum of the 3 numbers.
- ❑ Add the 3 numbers and store the result in the variable sum.
- ❑ Print the value of variable sum
- ❑ END

# Example of an Algorithm (contd...)

- Step 3: Testing the algorithm by implementing it.

Inorder to test the algorithm, let's implement it in C language

- Program in C for above algorithm

```
#include <stdio.h>
int main()
{
    int num1, num2, num3;          // Variables to take the input of the 3 numbers
    int sum;                      // Variable to store the resultant sum
    printf("Enter 3 numbers: ");   // Take the 3 numbers as input
    scanf("%d%d%d", &num1,&num2,&num3);
    printf("%d\n%d\n%d\n", num1,num2,num3);
    sum = num1 + num2 + num3;     // Calculate the sum using + operator and store it in variable sum
    printf("\nSum of the 3 numbers is: %d", sum); // Print the sum
    return 0;
}
```

# Correctness of Algorithms

- ❖ An algorithm is said to be correct with respect to a specification, in theoretical Computer Science
- ❖ The functional correctness refers to the input output behavior of algorithm means if for each input it produces the expected output
- ❖ Correctness is of two types:
  - **Partial Correctness:**
    - It means if the algorithm returns an answer then it is said to be partially correct
  - **Total Correctness:**
    - It means along with the answer, the algorithm must terminates also

# Analysis of Algorithm

## ❖ Priori Analysis:

- “Priori” means “before”, hence Priori analysis means checking the algorithm before its implementation
- In this, the algorithm is checked when it is written in the form of theoretical steps
- The Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed are constant and have no effect on the implementation
- This is done usually by the algorithm designer
- It is in this method, that the Algorithm Complexity is determined

# Analysis of Algorithm (contd...)

- ❖ **Posterior Analysis:**
  - “Posterior” means “after”, hence Posterior analysis means checking the algorithm after its implementation
  - In this, the algorithm is checked by implementing it in any programming language and executing it
  - This analysis helps to get the actual and real analysis report about correctness, space required, time consumed etc.
- **Time Factor:** Time is measured by counting the number of key operations such as comparisons in the sorting algorithm
- **Space Factor:** Space is measured by counting the maximum memory space required by the algorithm

# Analysis of Algorithm (contd...)

- ❖ **Space Complexity:** Space complexity of an algorithm refers to the amount of memory that this algorithm requires to execute and get the result. This can be for inputs, temporary operations, or outputs.
  - **How to calculate Space Complexity?**

The space complexity of an algorithm is calculated by determining following 2 components:

- **Fixed Part:** This refers to the space that is definitely required by the algorithm. For example, input variables, output variables, program size, etc.
- **Variable Part:** This refers to the space that can be different based on the implementation of the algorithm. For example, temporary variables, dynamic memory allocation, recursion stack space, etc.

# Analysis of Algorithm (contd...)

- ❖ **Time Complexity:** Time complexity of an algorithm refers to the amount of time that this algorithm requires to execute and get the result. This can be for normal operations, conditional if-else statements, loop statements, etc.

## ➤ How to calculate Time Complexity?

The time complexity of an algorithm is also calculated by determining following 2 components:

- **Constant time part:** Any instruction that is executed just once comes in this part. For example, input, output, if-else, switch, etc.
- **Variable Time Part:** Any instruction that is executed more than once, say  $n$  times, comes in this part. For example, loops, recursion, etc.

# Algorithm Design Paradigms

- ❖ It is the approach for constructing efficient solutions to the problems
  - They provide templates suited to solving a broad range of diverse problems
  - They can be translated into common control and data structures provided by most high-level languages
  - The temporal and spatial requirements of the algorithms which result can be precisely analysed
- ❖ We will study the following paradigms for designing our algorithms:
  - Simple or Basic Algorithms (Insertion sort, Bubble sort, Linear Search)
  - Divide and Conquer (Binary Search, Merge Sort, Quick Sort, Maximum Subarray Problem, Matrix Multiplication, Largest Subarray Sum, Closest Pair, Convex Hull)

# Algorithm Design Paradigms(contd...)

- Dynamic Programming (0/1 Knapsack Problem, Matrix Chain Multiplication, LCS, Optimal BST)
- Greedy Method (Huffman Coding, Knapsack problem, Tree Traversals, MST- Prim's & Kruskal's Algo)
- Backtracking (N Queen's Problem, Sum of Subsets, Hamiltonian Circuit, TSP, Graph Algos - DFS, BFS, Shortest Path algo)
- Randomization and Approximation Algorithm (Randomized Quick Sort, String Matching algorithm, Rabin Karp Algorithm, Vertex Covering, P/NP problems)

# Insertion Sort

- ❖ **Concept:** Insertion sort is a Simple Sorting Algorithm,
  - It is similar to the way one sorts the playing cards in hand
  - Array is split virtually into sorted and unsorted part
  - The values from the unsorted portion is picked and placed at the correct position in the sorted part
- ❖ **Algorithm:** The algorithm is as follows:

To sort an array of size n in ascending order:

1. Iterate from arr[1] to arr[n] over the array
2. Compare the current element (key) to its predecessor
3. If the key element is smaller than its predecessor, compare it to the elements before
4. Move the greater elements one position up to make space for the swapped element

# Insertion Sort(contd...)

- ❖ Example: Sort the given array with following values 7,4,3,1,8,2,6,5

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 3 | 1 | 8 | 2 | 6 | 5 |
| 4 | 7 | 3 | 1 | 8 | 2 | 6 | 5 |
| 3 | 4 | 7 | 1 | 8 | 2 | 6 | 5 |
| 1 | 3 | 4 | 7 | 8 | 2 | 6 | 5 |
| 1 | 3 | 4 | 7 | 8 | 2 | 6 | 5 |
| 1 | 2 | 3 | 4 | 7 | 8 | 6 | 5 |
| 1 | 2 | 3 | 4 | 6 | 7 | 8 | 5 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

# Insertion Sort(contd...)

- ❖ Pseudocode: Dry Run the Algorithm with one example 5,2,4,6,1,3

**Input:** A sequence of  $n$  numbers  $\square a_1, a_2, \dots, a_n \square$ .

**Output:** A permutation (reordering)  $(a'_1, a'_2, \dots, a'_n)$  of the input sequence such that  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

```
INSERTION-SORT ( $A$ )
1  for  $j \leftarrow 2$  to  $\text{length}[A]$ 
2      do  $\text{key} \leftarrow A[j]$ 
3          ▷ Insert  $A[j]$  into the sorted sequence  $A[1 \square j - 1]$ .
4           $i \leftarrow j - 1$ 
5          while  $i > 0$  and  $A[i] > \text{key}$ 
6              do  $A[i + 1] \leftarrow A[i]$ 
7                   $i \leftarrow i - 1$ 
8           $A[i + 1] \leftarrow \text{key}$ 
```

# Analysis of Insertion Sort

We start by presenting the INSERTION-SORT procedure with the time "cost" of each statement and the number of times each statement is executed. For each  $j = 2, 3, \dots, n$ , where  $n = \text{length}[A]$ , we let  $t_j$  be the number of times the **while** loop test in line 5 is executed for that value of  $j$ . When a **for** or **while** loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body. We assume that comments are not executable statements, and so they take no time.

| INSERTION-SORT ( $A$ )                                                         | <i>cost</i> | <i>times</i>             |
|--------------------------------------------------------------------------------|-------------|--------------------------|
| 1 <b>for</b> $j \leftarrow 2$ <b>to</b> $\text{length}[A]$                     | $c_1$       | $n$                      |
| 2 <b>do</b> $\text{key} \leftarrow A[j]$                                       | $c_2$       | $n - 1$                  |
| 3           ▷ Insert $A[j]$ into the sorted<br>sequence $A[1 \square j - 1]$ . | 0           | $n - 1$                  |
| 4 $i \leftarrow j - 1$                                                         | $c_4$       | $n - 1$                  |
| 5 <b>while</b> $i > 0$ and $A[i] > \text{key}$                                 | $c_5$       | $\sum_{j=2}^n t_j$       |
| 6 <b>do</b> $A[i + 1] \leftarrow A[i]$                                         | $c_6$       | $\sum_{j=2}^n (t_j - 1)$ |
| 7 $i \leftarrow i - 1$                                                         | $c_7$       | $\sum_{j=2}^n (t_j - 1)$ |
| 8 $A[i + 1] \leftarrow \text{key}$                                             | $c_8$       | $n - 1$                  |

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes  $c_i$  steps to execute and is executed  $n$  times will contribute  $c_i n$  to the total running time.<sup>[5]</sup> To compute  $T(n)$ , the running time of INSERTION-SORT, we sum the products of the *cost* and *times* columns, obtaining

# Analysis of Insertion Sort(Best Case) - $\Omega(n)$

$$\begin{aligned}T(n) &= c_1n + c_2(n - 1) + c_4(n - 1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) \\&\quad + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n - 1).\end{aligned}$$

Even for inputs of a given size, an algorithm's running time may depend on *which* input of that size is given. For example, in INSERTION-SORT, the best case occurs if the array is already sorted. For each  $j = 2, 3, \dots, n$ , we then find that  $A[i] \leq \text{key}$  in line 5 when  $i$  has its initial value of  $j - 1$ . Thus  $t_j = 1$  for  $j = 2, 3, \dots, n$ , and the best-case running time is

$$\begin{aligned}T(n) &= c_1n + c_2(n - 1) + c_4(n - 1) + c_5(n - 1) + c_8(n - 1) \\&= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).\end{aligned}$$

This running time can be expressed as  $an + b$  for constants  $a$  and  $b$  that depend on the statement costs  $c_i$ ; it is thus a **linear function** of  $n$ .

# Analysis of Insertion Sort(Worst Case) - O(n^2)

If the array is in reverse sorted order—that is, in decreasing order—the worst case results. We must compare each element  $A[j]$  with each element in the entire sorted subarray  $A[1 \square j - 1]$ , and so  $t_j = j$  for  $j = 2, 3, \dots, n$ . Noting that

$$\sum_{j=2}^n j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^n (j - 1) = \frac{n(n-1)}{2}$$

(see [Appendix A](#) for a review of how to solve these summations), we find that in the worst case, the running time of INSERTION-SORT is

$$\begin{aligned} T(n) &= c_1 n + c_2(n - 1) + c_4(n - 1) + c_5 \left( \frac{n(n+1)}{2} - 1 \right) \\ &\quad + c_6 \left( \frac{n(n-1)}{2} \right) + c_7 \left( \frac{n(n-1)}{2} \right) + c_8(n - 1) \\ &= \left( \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left( c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

This worst-case running time can be expressed as  $an^2 + bn + c$  for constants  $a$ ,  $b$ , and  $c$  that again depend on the statement costs  $c_i$ ; it is thus a **quadratic function** of  $n$ .

# Analysis of Insertion Sort(Average Case) - $\Theta(n^2)$

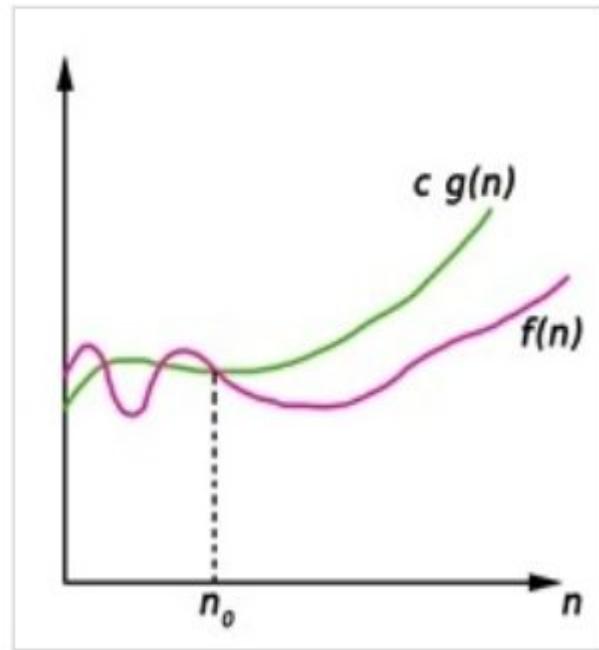
- ❖ The "average case" is often roughly as bad as the worst case
- ❖ Let us randomly choose  $n$  numbers and apply insertion sort
- ❖ How long does it take to determine where in subarray  $A[1:j-1]$  to insert element  $A[j]$ ?
- ❖ On average, half the elements in  $A[1:j-1]$  are less than  $A[j]$ , and half the elements are greater
- ❖ On average, therefore, we check half of the subarray  $A[1:j-1]$ , so  $t_j = j/2$
- ❖ If we work out the resulting average-case running time, it turns out to be a quadratic function of the input size, just like the worst-case running time
- ❖ **Do the analysis derive the equation and prove the average case analysis**
- ❖ **Further we will do only worst case analysis of the algorithms**

# Asymptotic Notations based on growth functions

- ❖ The word asymptotic here refers to a line that when gets closer to infinity will eventually get close to the curve
- ❖ Asymptotic Notations are used to represent the complexities of algorithms for asymptotic analysis
- ❖ These notations are mathematical tools to represent the complexities
- ❖ There are three notations that are commonly used:
  - Big Oh Notation  $O$
  - Big Omega Notation  $\Omega$
  - Big Theta Notation  $\Theta$

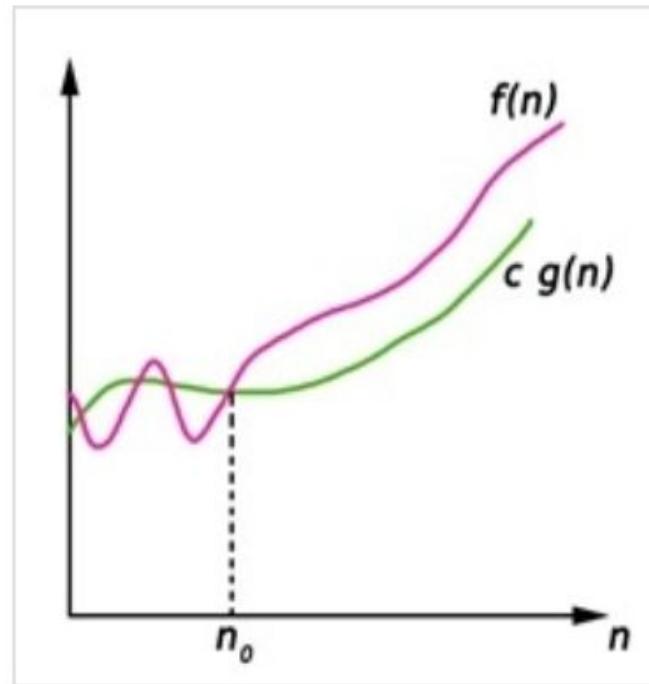
# Big Oh Notation (**O** - Notation)

- ❖ Big-Oh ( $O$ ) notation gives an upper bound for a function  $f(n)$  to within a constant factor
- ❖ Here  $f(n) = O(g(n))$ , If there are positive constants  $n_0$  and  $c$  such that, to the right of  $n_0$  the  $f(n)$  always lies on or below  $c \cdot g(n)$
- ❖  $O(g(n)) = \{ f(n) : \text{There exist positive constant } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c \cdot g(n), \text{ for all } n \geq n_0\}$



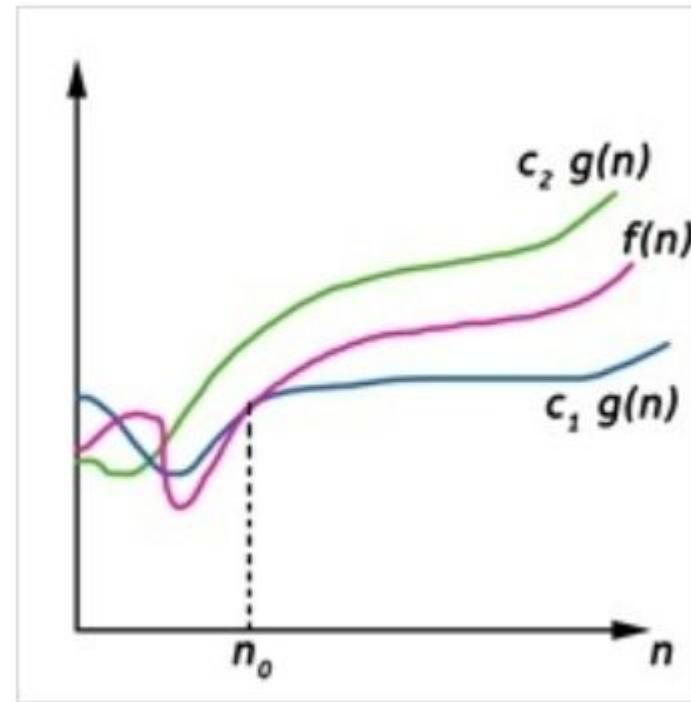
# Big Omega Notation ( $\Omega$ - Notation)

- ❖ Big-Omega ( $\Omega$ ) notation gives a lower bound for a function  $f(n)$  to within a constant factor
- ❖ Here  $f(n) = \Omega(g(n))$ , If there are positive constants  $n_0$  and  $c$  such that, to the right of  $n_0$  the  $f(n)$  always lies on or above  $c*g(n)$
- ❖  $\Omega(g(n)) = \{ f(n) : \text{There exist positive constant } c \text{ and } n_0 \text{ such that } 0 \leq c g(n) \leq f(n), \text{ for all } n \geq n_0\}$



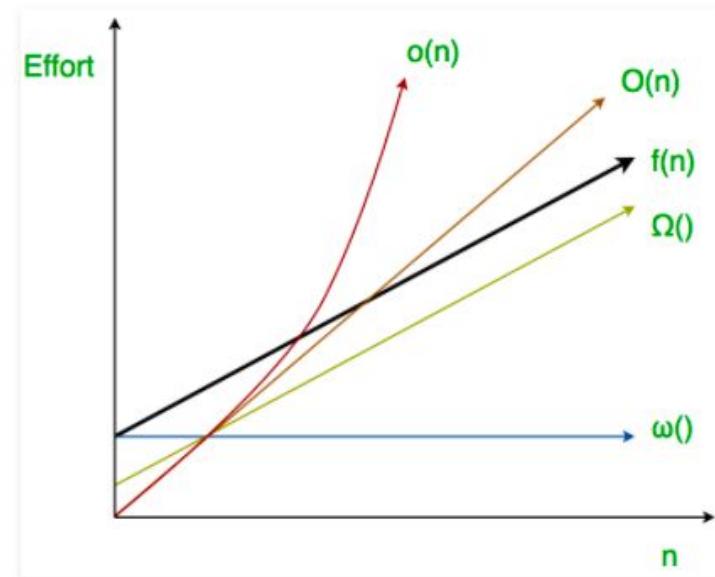
# Big Theta Notation ( $\Theta$ - Notation)

- ❖ Big-Theta( $\Theta$ ) notation gives bound for a function  $f(n)$  to within a constant factor
- ❖ Here  $f(n) = \Theta(g(n))$ , If there are positive constants  $n_0$  and  $c_1$  and  $c_2$  such that, to the right of  $n_0$  the  $f(n)$  always lies between  $c_1*g(n)$  and  $c_2*g(n)$  inclusive
- ❖  $\Theta(g(n)) = \{f(n) : \text{There exist positive constant } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \text{ for all } n \geq n_0\}$



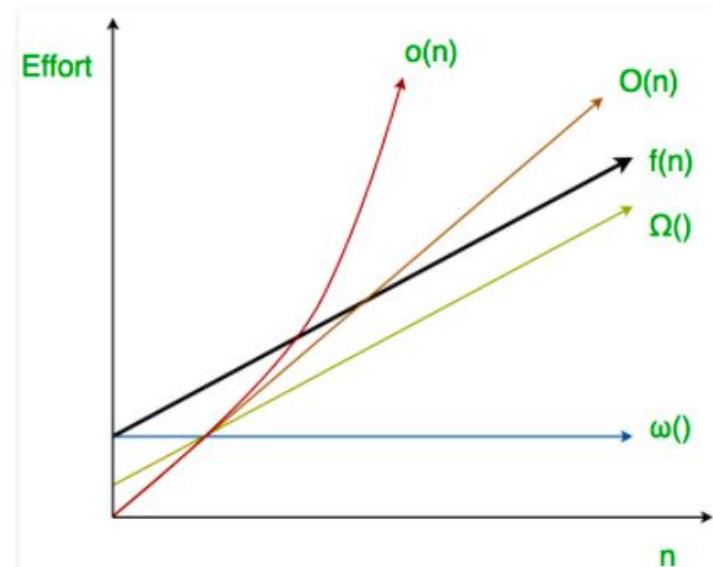
# Little oh Notation ( $o$ - Notation)

- ❖ Little-o notation is known as the loose upper bound, it can be used to describe the upper-bound that cannot be tight
- ❖ Let  $f(n)$  and  $g(n)$  be functions, we say that  $f(n)$  is  $o(g(n))$ , if for any real constant  $c > 0$ , there exists a constant  $n_0 \geq 1$ , such that  $0 \leq f(n) < c*g(n)$  for all  $n \geq n_0$
- ❖ Thus, little  $o()$  means **loose upper-bound** of  $f(n)$ . Little  $o$  is a rough estimate of the maximum order of growth whereas Big-O may be the actual order of growth



# Little Omega Notation ( $\omega$ - Notation)

- ❖ Little- $\omega$  notation is known as the loose upper bound, it can be used to describe the upper-bound that cannot be tight
- ❖ Let  $f(n)$  and  $g(n)$  be functions, we say that  $f(n)$  is  $\omega(g(n))$ , if for any real constant  $c > 0$ , there exists a constant  $n_0 \geq 1$ , such that  $f(n) > c * g(n) \geq 0$  for every integer  $n \geq n_0$
- ❖ Little Omega ( $\omega$ ) is a rough estimate of the order of the growth whereas Big Omega ( $\Omega$ ) may represent exact order of growth. We use  $\omega$  notation to denote a lower bound that is not asymptotically tight. And,  $f(n) \in \omega(g(n))$  if and only if  $g(n) \in o(f(n))$



# Properties of Asymptotic Notations

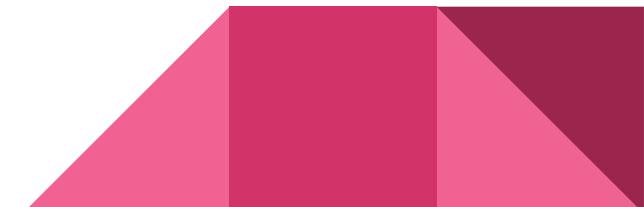
Discussed in class and refer to PDF notes

# Numericals on Asymptotic Notations

Discussed in class and refer to PDF notes

# Induction Method(Mathematical Induction)

- ❖ MI is used for proving the correctness of the algorithm.
- ❖ Generally, it is used for proving that the statement holds true for each natural number  $n$ .
- ❖ We go through 3 steps in it:
  - **Induction Hypothesis:** It defines the function to be proven for every  $n$ . Say  $F(n)$ .
  - **Induction Base:** The function is proven for an initial value, this is done by solving the induction hypothesis for  $n = 1$  or whichever initial value is appropriate.
  - **Induction Step:**  $F(n)$  once proved is correct, we go further one step, and assume  $F(n+1)$  is also correct.



# Example for Induction Method

Let us consider  $F(n)$  to be sum of first  $n$  natural numbers, Eg.  $F(3) = 3+2+1$ , prove that the following formula can be applied to any  $n$ ,

$$F(n) = (n+1) * n / 2$$

- ❖ **Induction Hypothesis:**  $F(n)$  defined as above formula
- ❖ **Induction Base:** In this step, we prove that  $F(1) = 1$ ,
  - $F(1) = ((1+1)*1)/2 = 1$
- ❖ **Induction Step:** In this step, we prove that if the formula applies to  $F(n)$ , it also applies to  $F(n+1)$  as follows:
  - $F(n+1) = (((n+1)+1)*(n+1))/2 = ((n+2)*(n+1))/2 \text{ ----- (I)}$
  - This is known as an implication ( $a \Rightarrow b$ ), we have to prove  $b$  is correct, proving that  $a$  is correct.
- ❖ 
$$\begin{aligned} F(n+1) &= F(n) + (n+1) = ((n+1)*n)/2 + (n+1) \\ &= (n^2 + n + 2n + 2)/2 = (n^2 + 3n + 2)/2 \\ &= ((n+2) * (n+1))/2 \text{ ----- (II)} \end{aligned}$$
 Thus, I = II (proved)

# Proof of Correctness

- ❖ The method above used to prove an algorithm's correctness is mathematical based, or rather function based,
- ❖ We can say the more the solution is similar to a real mathematical function, the easier is the proof.

## Exercises 2.3-3

Use mathematical induction to show that when  $n$  is an exact power of 2, the solution of the recurrence

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

is  $T(n) = n \lg n$ .

# Numericals on Mathematical Induction

Discussed in class and refer to PDF notes

# Recurrence Relations

- ❖ A recurrence is a relation or inequality that describes the function in terms of its value on smaller inputs
- ❖ Recurrence can be of the type:
  - $T(n) = \begin{cases} \Theta(1) & \text{----- if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{----- if } n > 1 \end{cases}$
  - Whose solution is claimed to be  $T(n) = \Theta(n \log n)$
- ❖ For solving recurrence, i.e. for obtaining the  $\Theta$  or  $\mathcal{O}$  bounds of the solution, we have the following 3 methods:
  - Substitution Method
  - Recursion Tree Method
  - Master Theorem Method

# Definition of a Recurrence Relation

- ❖ A recursive algorithm is one which makes a recursive call to itself with smaller inputs
- ❖ We often use a recurrence relation to describe the running time of a recursive algorithm
- ❖ A **recurrence relation** is an equation or inequality that describes a function in terms of its value on smaller inputs or as a function of preceding (or lower) terms
- ❖ Like all recursive functions, a recurrence also consists of two steps:
  - Basic step:
    - Here we have one or more constant values which are used to terminate recurrence
    - It is also known as initial conditions or base conditions.
  - Recursive steps:
    - This step is used to find new terms from the existing (preceding) terms
    - Thus in this step the recurrence compute next sequence from the k preceding values
    - This formula is called a recurrence relation (or recursive formula)
    - This formula refers to itself, and the argument of the formula must be on smaller values (close to the base value)
- ❖ Hence a recurrence has one or more initial conditions and a recursive formula, known as **recurrence relation**

# Examples of Recurrence Relation

Discussed in class and refer to PDF notes

# Substitution Method

- ❖ In this method, we find the solution by:
  - Guess the form of the solution
  - Then use Mathematical Induction to find the constants and show that the solution works
- ❖ Example of the recurrence using Substitution Method:

$$T(n) = 2T\lceil n/2 \rceil + n \quad \dots \quad (1)$$

**Solution:**

1. Guess the form of the solution

Let's assume that the solution of the above recurrence is  $O(n \log n)$  (worst case)

2. Now we will use Mathematical Induction to prove that the assumption in step 1 is correct

# Recurrence Solution using Substitution Method

$$\begin{aligned} T(n) &\leq cn \log n & \text{----- (2) for } c > 0 \\ &\leq 2c(n/2) \log(n/2) + n \end{aligned}$$

Putting (2) in (1)

$$\begin{aligned} &\leq cn \log(n/2) + n \\ &\leq cn \log n - cn \log 2 + n \\ &\leq cn \log n - cn + n \end{aligned}$$

Putting  $c = 1$  (apt value to diminish unnecessary terms)

$$T(n) \leq 1 * n \log n - n + n$$

$$T(n) \leq 1 * n \log n$$

1 here is constant value thus replacing 1 with  $c$

$$T(n) \leq cn \log n$$

**Base Case:**

$$\text{For } n = 1, T(1) = c * 1 \log 1 = 0$$

But  $T(1)$  should be 1, thus this equality doesn't hold for  $n = 1$

# Recurrence Solution using Substitution Method

For  $n = 2$ ,  $T(2) = c * 2 \log 2$

$$2 T(2/2) + 2 \leq c^2$$

$$2 T(1) + 2 \leq 2c$$

$$2 * 0 + 2 \leq 2c$$

$$2 \leq 2c$$

$c \geq 1$  Thus,  $T(n) \leq cn \log n$  for  $n = 2$

## Inductive step:

Let  $n = n/2$ ,  $T(n/2) \leq c.n/2 \log (n/2)$  is true

Now, we can say that it holds for  $n = n$ ,

$$T(n) \leq cn \log n$$

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

$$\leq 2 c(n/2) \log (n/2) + n$$

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

$$\leq cn \log n - cn \log 2 + n$$

$$\leq cn \log n - cn + n$$

$$\leq cn \log n \quad \forall c \geq 1$$

Thus,  $T(n) = O(n \log n)$

# Numericals for Substitution Method (Subtleties)

❖  $T(n) = T\lceil n/2 \rceil + T\lfloor n/2 \rfloor + 1$  ---- (1)

➤ **Solution:** Let's assume the solution to be  $O(n)$

$$T(n) \leq cn \text{ ---- (2)}$$

Substitute (2) in (1)

$$T(n) \leq c(n/2) + c(n/2) + 1$$

$T(n) \leq cn - d$  ---- (3) (As there is a +ve constant, then we need to subtract any constant 'd')

Now taking running time as  $O(n - d)$

Substitute (3) in (1)

$$T(n) \leq c(n/2) - d + c(n/2) - d + 1$$

$$T(n) \leq 2(c(n/2) - d) + 1$$

$$T(n) \leq cn - (2d - 1)$$

$T(n) \leq cn - d$  for  $d > 1$  Here, we are not able to get rid of the constant 'd'

Therefore,  $T(n) = O(n - d)$  is correct

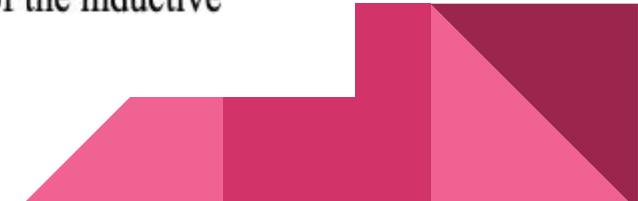
# Note while guessing the solution

## Avoiding pitfalls

It is easy to err in the use of asymptotic notation. For example, in the recurrence (4.4) we can falsely "prove"  $T(n) = O(n)$  by guessing  $T(n) \leq cn$  and then arguing

$$\begin{aligned} T(n) &\leq 2(c \lfloor n/2 \rfloor) + n \\ &\leq cn + n \\ &= O(n), \Leftarrow \text{wrong!!} \end{aligned}$$

since  $c$  is a constant. The error is that we haven't proved the *exact form* of the inductive hypothesis, that is, that  $T(n) \leq cn$ .



# Numericals for Substitution Method(Changing Variables)

❖  $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n$  ----- (1)

> Solution:

## 1. Guess the Solution:

Let's replace  $\log n$  by  $m$ , it means that  $n = 2^m$ , on replacing the values in (1) the modified recurrence becomes,

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

Now replace  $T(2^m)$  by  $S(m)$  in (2)

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

This is similar with recurrence for merge sort, thus we assume the solution to be  $O(m \log m)$

## 2. Using Mathematical Induction to find the constant and show that the solution works:

$$S(m) \leq cm \log m \text{ for } c > 0 \text{ ----- (4)}$$

Substituting (4) in (3),  $S(m) \leq 2c(m/2) \log(m/2) + m$

$$S(m) \leq cm (\log m - \log 2) + m$$

$$S(m) \leq cm \log m - cm + m$$

$$S(m) \leq cm \log m \text{ (for } c = 1\text{)}$$

Let's check for boundary condition,  $S(1) = 0$  for  $m = 1$ ,

$$m = 2, S(2) = 2S(1) + 2 = 2$$

$$m = 3, S(3) = 2S(1) + 3 = 3$$

$$m = 4, S(4) = 2S(2) + 4 \text{ thus it became free from } S(1).$$

Therefore,  $n_0 > 3$ ,

Replacing the value of  $m$  in  $T(n)$

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

# Iteration Method(Unrolling and Summing)

- ❖ The iteration method converts the recurrence into a summation and then relies on techniques for bounding summations to solve the recurrence
- ❖ We unroll (or substituting) the given recurrence back to itself until not getting a regular pattern (or series)
- ❖ We generally follow the following steps to solve any recurrence:
  - Expand the recurrence
  - Express the expansion as a summation by plugging the recurrence back into itself until you see a pattern
  - Evaluate the summation by using the arithmetic or geometric summation formulae

# Numericals of Iteration Method

Example1: Consider a recurrence relation of algorithm1 of section 1.8

$$M(n) = \begin{cases} b & \text{if } n = 1 \\ c + M(n - 1) & n \geq 2 \end{cases} \quad \begin{array}{l} (\text{base case}) \\ (\text{Recursive step}) \end{array}$$

Solution: Here

$$M(1) = b \quad \dots \dots \dots (1)$$

$$M(n) = c + M(n - 1) \quad \dots \dots \dots (2)$$

Now substitute the value of  $M(n-1)$  in equation(2),

$$\begin{aligned} M(n) &= c + M(n - 1) \\ &= c + \{c + M(n - 2)\} \\ &= c + c + M(n - 2) \\ &= c + c + c + M(n - 3) \quad [\text{By substituting } M(n-2) \text{ in equation (2)}] \end{aligned}$$

$$= \dots \dots \dots$$

$$= c + c + c + \dots \dots \dots (n - 1) \text{ times} + M(n - (n - 1))$$

$$= (n - 1)c + M(1) = nc - c + b = nc + (b - c) = O(n)$$

Example2: Consider a recurrence relation of algorithm2 of section 1.8

$$\begin{pmatrix} T(1) = a & (\text{base case}) \\ T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n & (\text{Recursive step}) \end{pmatrix}$$

Solution:

Solution: Here

$$T(1) = b \quad \dots \dots \dots (1)$$

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \quad \dots \dots \dots (2)$$

When we are solving recurrences we always omit the sealing or floor because it won't affect the result. Hence we can write the equation 2 as:

$$T(n) = T\left(\frac{n}{2}\right) + n \quad \dots \dots \dots (3)$$

Now substitute the value of  $T\left(\frac{n}{2}\right)$  in equation (3),

# Numericals of Iteration Method(contd...)

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

$$= n + \left\{ \frac{n}{2} + T\left(\frac{n}{4}\right) \right\} = n + \frac{n}{2} + T\left(\frac{n}{4}\right)$$

$$= n + \frac{n}{2} + \frac{n}{4} + T\left(\frac{n}{8}\right) \quad (\text{By substituting } T\left(\frac{n}{4}\right) \text{ in equation 3})$$

$$= \underbrace{n + \frac{n}{2} + \frac{n}{4} + \cdots \cdots \frac{n}{2^{k-1}}}_{k=\log_2 n \text{ terms (total)}} + T\left(\frac{n}{2^k}\right)$$

$$= \frac{n[1 - n^{\log_2 \frac{1}{2}}]}{(1 - \frac{1}{2})} \quad [\text{Using log property } a^{\log_b n} = n^{\log_b a}]$$

$$= 2n[1 - n^{\log_2 1 - \log_2 2}] = 2n[1 - n^{0-1}] = 2n\left[1 - \frac{1}{n}\right]$$

for getting boundary condition;

$$T\left(\frac{n}{2^k}\right) = T(1) \Rightarrow \frac{n}{2^k} = 1 \Rightarrow k = \log_2 n. \quad [\text{Taking log both side}]$$

$$= 2n - 2 = O(n)$$

Thus we have a G.P. series:

$$\underbrace{n + \frac{n}{2} + \frac{n}{4} + \cdots \cdots \frac{n}{2^{k-1}}}_{k=\log_2 n \text{ terms (total)}} + T(1) = \underbrace{n + \frac{n}{2} + \frac{n}{4} + \cdots \cdots \frac{n}{2^{k-1}}}_{k=\log_2 n \text{ terms}} + b$$

$$= \frac{n[1 - \left(\frac{1}{2}\right)^{\log_2 n}]}{(1 - \frac{1}{2})} \quad [\text{By using GP series sum formula } S_n = \frac{a(1-x^n)}{1-x}]$$

# Mathematical Formulae for Algorithms

## Sum formula for Arithmetic Series

Given a arithmetic Series:  $a, a + d, a + 2d, \dots, a + (n - 1).d$

$$\sum_{i=0}^{n-1} (a + id) = \frac{n}{2} [2a + (n - 1).d] = \frac{n}{2} [a + l]$$

where  $a$  is a first term,  $d$  is a common difference and

$l = a + (n - 1).d$  is a last or  $n^{\text{th}}$  term

## Sum formula for Geometric Series

For  $\text{real } r \neq 1$ , the summation

$$\sum_{i=0}^{n-1} ax^k = a + ax + ax^2 + ax^3 + \dots + ax^{n-1}$$

is a **geometric** or **exponential** series and has a value

I)  $\sum_{k=0}^{n-1} ax^k = \frac{a(x^n - 1)}{x - 1}$  (when  $x > 1$ )

II)  $\sum_{k=0}^{n-1} ax^k = \frac{a(1 - x^n)}{1 - x}$  (when  $x < 1$ )

When the summation is *infinite* and  $|x| < 1$ , we have

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

# Mathematical Formulae for Algorithms(contd...)

## Logarithmic Formulae

The following logarithmic formulae are quite useful for solving recurrence relation.

For all real  $a > 0, b > 0, c > 0$ , and  $n$ ;

$$1. \ b^{\log_b a} = a$$

$$2. \ \log_c(ab) = \log_c a + \log_c b$$

$$3. \ \log_b a^n = n \log_b a$$

$$4. \ \log_b \left(\frac{m}{n}\right) = \log_b m - \log_b n \text{ where } a > 0 \text{ and } a \neq 1$$

$$5. \ \log_b a = \frac{\log_c a}{\log_c b}$$

$$6. \ \log_b a = 1/\log_a b$$

$$7. \ a^{\log_b n} = n^{\log_b a}$$

(Note: Proof is left as an exercise)

**Remark:** Since changing the base of a logarithm from one constant to another only changes the value of the logarithm by a constant factor (see property 4), we don't care about the base of a "log" when we are writing a time complexity using O, Ω, or θ-notations. We always write 2 as the base of a log. **For Example:**  $\log_{3/2} n = \frac{\log_2 n}{\log_2 \frac{3}{2}} = O(\log_2 n)$

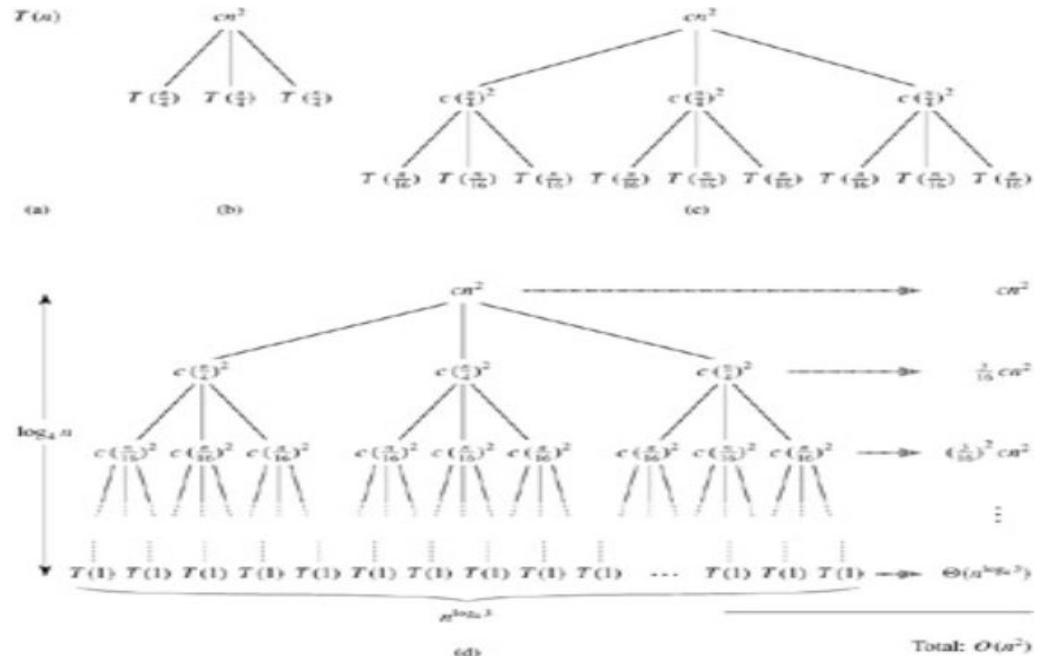
# Recursion Tree

- ❖ In a recursion tree, each node represents the cost of the single problem, in the set of recursive functions
- ❖ The set cost per level is computed by doing the summation of the cost within each level of the tree
- ❖ Further, we need to sum all the per level set of costs, to determine the total cost of all levels of the recursion tree
- ❖ A recursion tree is used to generate a good guess which we can then verify by the substitution method
- ❖ When using a recursion tree to generate a good guess, you can often tolerate a small amount of "sloppiness," since you will be verifying your guess later on
- ❖ If you are very careful when drawing out a recursion tree and summing the costs, however, you can use a recursion tree as a direct proof of a solution to a recurrence

# Numericals for Recursion Tree

- ❖ Consider the following recurrence  $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$
- ❖ Because we know that floors and ceilings are usually insubstantial in solving recurrences
- ❖ As we tend to find the upper bound for the solution, thereby replacing  $\Theta$  by big O and then by constant c
- ❖ We create a recursion tree for the recurrence  $T(n) = 3T(n/4) + cn^2$  for  $c > 0$
- ❖ The recursion tree is given in Figure 4.1 in the next slide
- ❖ For convenience, we assume that n is an exact power of 4

# Numericals for Recursion Tree(contd...)



**Figure 4.1:** The construction of a recursion tree for the recurrence  $T(n) = 3T(n/4) + cn^2$ . Part (a) shows  $T(n)$ , which is progressively expanded in (b)-(d) to form the recursion tree. The fully expanded tree in part (d) has height  $\log_4 n$  (it has  $\log_4 n + 1$  levels).

# Numericals for Recursion Tree(contd...)

Solve the following recurrences using Recursion Tree.

1.  $T(n) = 2 T(n/2) + n$
2.  $T(n) = T(\lfloor n/3 \rfloor) + T(\lceil 2n/3 \rceil) + n$
3.  $T(n) = 2T(n-1) + 1$
4.  $T(n) = 3T(n/4) + n^2$
5.  $T(n) = 2T(n/2) + \sqrt{n}$

Solution of few in the PDF notes of Unit - 1

# Analysis of codes

including sequential statements,  
loops,  
conditional statements

# Algorithm for Linear Search

Worst case analysis O(n)  
Best Case analysis O(1)  
Average Case analysis  
 $O((n+1)/2)$

Algorithm: (**Linear search**)

/\* **Input:** A **linear** list A with n elements and a searching element  $x$ .

**Output:** Finds the location LOC of  $x$  in the array A (by returning an index )  
or return LOC=0 to indicate  $x$  is not present in A. \*/

1. [Initialize]: Set K=1 and LOC=0.
2. Repeat step 3 and 4 **while**(LOC == 0 && K ≤ n)
3.     **if**(x == A[K])
4.         {
5.             LOC=K
6.             K = K + 1;
7.         }
8.     **if**(LOC == 0)
9.         printf(" x is not present in the given array A);
10.    **else**
11.         printf(" x is present in the given array A at location A[LOC]);
12. Exit [end of algorithm]

# Algorithm for Merge Sort

Worst Case Analysis  $O(n \log n)$

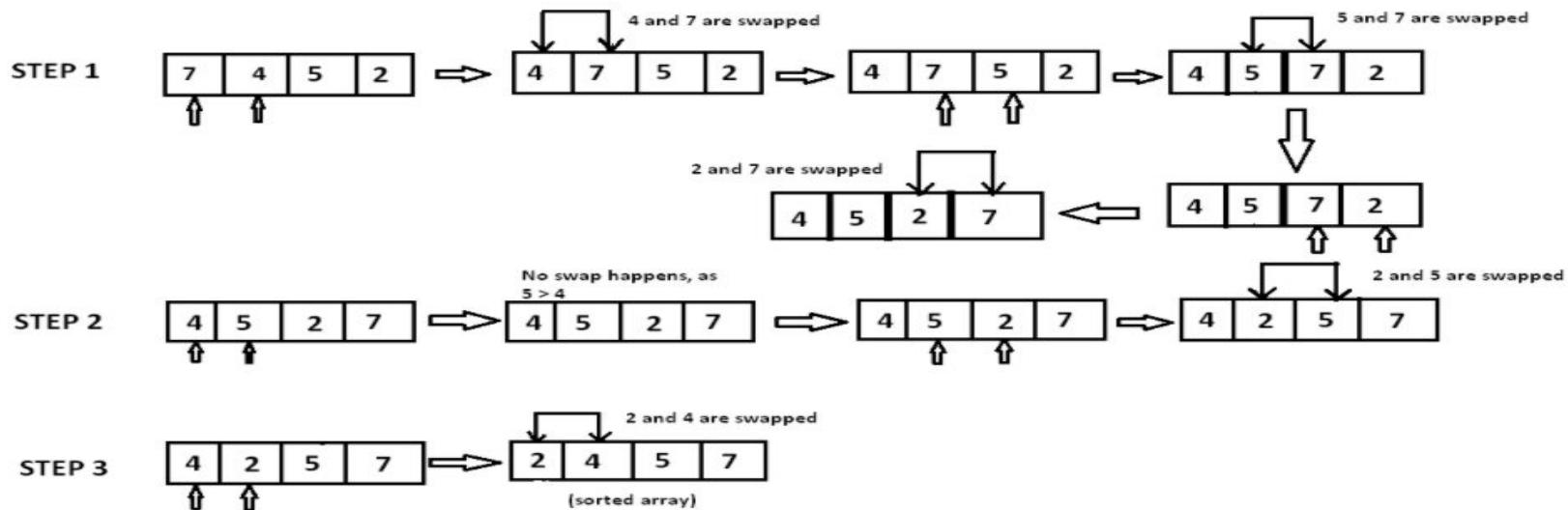
```
MERGE( $A, p, q, r$ )
1  $n_1 \leftarrow q - p + 1$ 
2  $n_2 \leftarrow r - q$ 
3 create arrays  $L[1 \square n_1 + 1]$  and  $R[1 \square n_2 + 1]$ 
4 for  $i \leftarrow 1$  to  $n_1$ 
5   do  $L[i] \leftarrow A[p + i - 1]$ 
6 for  $j \leftarrow 1$  to  $n_2$ 
7   do  $R[j] \leftarrow A[q + j]$ 
8  $L[n_1 + 1] \leftarrow \infty$ 
9  $R[n_2 + 1] \leftarrow \infty$ 
10  $i \leftarrow 1$ 
11  $j \leftarrow 1$ 
12 for  $k \leftarrow p$  to  $r$ 
13   do if  $L[i] \leq R[j]$ 
14     then  $A[k] \leftarrow L[i]$ 
15      $i \leftarrow i + 1$ 
16   else  $A[k] \leftarrow R[j]$ 
17      $j \leftarrow j + 1$ 
```

```
MERGE-SORT( $A, p, r$ )
1 if  $p < r$ 
2   then  $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
3     MERGE-SORT( $A, p, q$ )
4     MERGE-SORT( $A, q + 1, r$ )
5     MERGE( $A, p, q, r$ )
```

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

# Bubble Sort

- ❖ Bubble sort is based on the idea of repeatedly comparing pairs of adjacent elements and then swapping their positions if they exist in the wrong order
- ❖ Assume that  $A[]$  is an unsorted array of  $n$  elements. This array needs to be sorted in ascending order
- ❖ Example:



# Bubble Sort(contd...)

- ❖ Algorithm For Bubble Sort:

Bubble\_Sort(A)

1. for  $j \leftarrow 0$  to  $\text{length}[A] - 1$
2.     for  $i \leftarrow 0$  to  $\text{length}[A] - j - 1$
3.         if( $A[i] > A[i+1]$ )
4.             // Swap the position of elements
5.             Do  $\text{temp} \leftarrow A[i]$
6.              $A[i] \leftarrow A[i+1]$
7.              $A[i+1] \leftarrow \text{temp}$

The **worst and average case Complexity is  $O(n^2)$**  because the entire array needs to be iterated for every element.

# Quick Sort

Quicksort, like merge sort, is based on the divide-and-conquer paradigm introduced in [Section 2.3.1](#). Here is the three-step divide-and-conquer process for sorting a typical subarray  $A[p \square r]$ .

- **Divide:** Partition (rearrange) the array  $A[p \square r]$  into two (possibly empty) subarrays  $A[p \square q - 1]$  and  $A[q + 1 \square r]$  such that each element of  $A[p \square q - 1]$  is less than or equal to  $A[q]$ , which is, in turn, less than or equal to each element of  $A[q + 1 \square r]$ . Compute the index  $q$  as part of this partitioning procedure.
- **Conquer:** Sort the two subarrays  $A[p \square q - 1]$  and  $A[q + 1 \square r]$  by recursive calls to quicksort.
- **Combine:** Since the subarrays are sorted in place, no work is needed to combine them: the entire array  $A[p \square r]$  is now sorted.

The following procedure implements quicksort.

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

To sort an entire array  $A$ , the initial call is  $\text{QUICKSORT}(A, 1, \text{length}[A])$ .

## Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray  $A[p \square r]$  in place.

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

## Worst Case Analysis $O(n^2)$

# Binary Search

```
Procedure Binary_search(A,n,x)
    A ← sorted array
    n ← size of array
    x ← value to be searched
    Set lowerBound = 1
    Set upperBound = n
    while x not found
        if upperBound < lowerBound
            EXIT: x does not exists.
        set midPoint = lowerBound + ( upperBound - lowerBound ) / 2
        if A[midPoint] < x
            set lowerBound = midPoint + 1
        if A[midPoint] > x
            set upperBound = midPoint - 1
        if A[midPoint] = x
            EXIT: x found at location midPoint
    end while
end procedure
```

**Worst Case Analysis :  $O(\log_2 n)$**

# References

- ❖ <https://www.geeksforgeeks.org/>
- ❖ [https://cgi.csc.liv.ac.uk/~ped/teachadmin/algor/algor\\_complete.html](https://cgi.csc.liv.ac.uk/~ped/teachadmin/algor/algor_complete.html)
- ❖ [https://en.wikipedia.org/wiki/Correctness\\_\(computer\\_science\)#:~:text=In%20theoretical%20computer%20science%2C%20correctness.it%20produces%20the%20expected%20output\)](https://en.wikipedia.org/wiki/Correctness_(computer_science)#:~:text=In%20theoretical%20computer%20science%2C%20correctness.it%20produces%20the%20expected%20output))
- ❖ Introduction to Algorithms, Cormen
- ❖ [http://160592857366.free.fr/joe/ebooks/ShareData/EDA\\_Chapter4%20Fundamentals%20of%20Algorithms.pdf](http://160592857366.free.fr/joe/ebooks/ShareData/EDA_Chapter4%20Fundamentals%20of%20Algorithms.pdf)
- ❖ <https://stackabuse.com/mathematical-proof-of-algorithm-correctness-and-efficiency/#loopinvariants>
- ❖ <https://www.hackerearth.com/practice/algorithms/sorting/bubble-sort/tutorial/>
- ❖ [https://www.tutorialspoint.com/design\\_and\\_analysis\\_of\\_algorithms/index.htm](https://www.tutorialspoint.com/design_and_analysis_of_algorithms/index.htm)
- ❖ <http://ccf.ee.ntu.edu.tw/~yen/courses/ds17/chapter-2.pdf>
- ❖ <https://www.mathsisfun.com/algebra/mathematical-induction.html>
- ❖

# Design and Analysis of Algorithms

## (18CSC204J)

### Unit - 2

By:  
Aarti Sharma  
Asst Professor  
CSE Dept  
SRMIST Delhi-NCR  
Campus

# Introduction to Divide and Conquer

- ❖ This technique can be divided into the following three parts:
  - **Divide:** This involves dividing the problem into some sub problem
  - **Conquer:** Sub problem by calling recursively until sub problem solved
  - **Combine:** The Sub problem Solved so that we will get find problem solution
- ❖ The following are some standard algorithms that follows Divide and Conquer algorithm
  - Binary Search
  - Quick Sort
  - Merge Sort
  - Closest Pair of Points
  - Strassen's Algorithm
  - Largest Sub-array Sum
  - Finding maximum and minimum in an array

Generalised Recurrence of DAC technique:

$$T(n) = \begin{cases} O(1) & \text{if } n < c \\ D(n) + aT(n/b) + C(n) & \text{otherwise} \end{cases}$$

# DAC algorithm

```
DAC(a, i, j)
{
    if(small(a, i, j))
        return(Solution(a, i, j))

    else
        m = divide(a, i, j)          // f1(n)
        b = DAC(a, i, mid)          // T(n/2)
        c = DAC(a, mid+1, j)        // T(n/2)
        d = combine(b, c)           // f2(n)

    return(d)
}
```

## Recurrence Relation for DAC algorithm :

This is recurrence relation for above program

$$T(n) = \begin{cases} O(1) & \text{if } n \text{ is small} \\ f1(n) + 2T(n/2) + f2(n) & \end{cases}$$

## Example:

To find the maximum and minimum element in a given array.

**Input:** { 70, 250, 50, 80, 140, 12, 14 }

**Output:** The minimum number in a given array  
is : 12

The maximum number in a given array is : 250

# Approach to find Maximum and Minimum element of an Array

- ❖ To find the maximum and minimum element from a given array is an application for divide and conquer
- ❖ In this problem, we will find the maximum and minimum elements in a given array
- ❖ In this problem, we are using a divide and conquer approach(DAC) which has three steps divide, conquer and combine
- ❖ **For Maximum:**
  - we are using the recursive approach to find maximum where we will see that only two elements are left and then we can easily using condition i.e. if( $a[\text{index}] > a[\text{index}+1]$ .)
  - In a program line  $a[\text{index}]$  and  $a[\text{index}+1]$ )condition will ensure only two elements in left

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

# Finding Maximum and Minimum element of an Array(contd...)

- ❖ In the above condition, we have checked the left side condition to find out the maximum
- ❖ Now, we will see the right side condition to find the maximum
- ❖ Recursive function to check the right side at the current index of an array
  1. *max = DAC\_Max(a, index+1, l);*
  2. *// Recursive call*
- ❖ Now, we will compare the condition and check the right side at the current index of a given array
- ❖ In the given program, we are going to implement this logic to check the condition on the right side at the current index
  1. *// Right element will be maximum.*
  2. *if(a[index]>max)*
  3. *return a[index];*
  4. *// max will be maximum element in a given array.*
  5. *else*
  6. *return max;*
  7. *}*
- ❖ Similarly for finding minimum, we will compare with min instead of max, and instead of > values we will check < values in the array

# Divide and Conquer Approach for finding Max & Min

**Algorithm:**

MaxMin(i, j, max, min)

**input:** array of N elements, i lower bound, j upper bound

**output:** max: largest value, min: smallest value

if ( $i == j$ ) then max = min = a( $i$ )

else

if ( $i = j - 1$ ) then

if ( $a(i) < a(j)$ ) then

max = a( $j$ )

min = a( $i$ )

else

else

max = a( $i$ )

min = a( $j$ )

mid =  $(i + j) / 2$

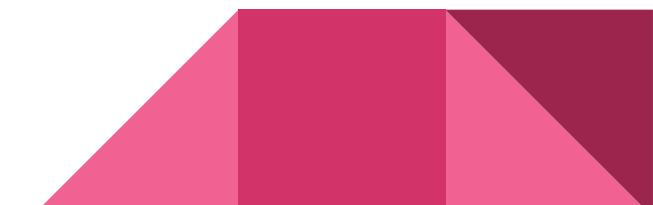
maxmin(i, mid, max, min)

maxmin(mid+1, j, max1, min1)

if ( $max < max1$ ) then max = max1

if ( $min > min1$ ) then min = min1

end



# Analysis of Maxmin algorithm

- ❖ **Complexity Analysis**
  - if size= 1 return current element as both max and min //base condition
  - else if size= 2 one comparison to determine max and min //base condition
  - else /\* if size > 2 find mid and call recursive function \*/
    - recur for max and min of left half recur for max and min of right half
    - one comparison determines to max of the two subarray, update max
    - one comparison determines min of the two subarray, update min
  - finally return or print the min/max in whole array
- ❖ Recurrence relation can be written as  $T(n) = 2(T / 2) + 2$  solving which give you  $T(n) = 3 / 2n - 2$  which is the exact no. of comparisons but still the **worst time complexity** will be  $T(n) = O(n)$  and **best case time** complexity will be  $O(1)$  when you have only one element in array, which will be candidate for both max and min
- ❖ **Note: No algorithm based on comparison can do less than this**

# Binary Search

- ❖ Binary Search is a searching algorithm
- ❖ In each step, the algorithm compares the input element  $x$  with the value of the middle element in array
- ❖ If the values match, return the index of the middle
- ❖ Otherwise, if  $x$  is less than the middle element, then the algorithm recurs for left side of the middle element, else recurs for the right side of the middle element
- ❖ **Procedure of Binary Search**
  - For a binary search to work, it is mandatory for the target array to be sorted
  - The following is our sorted array and let us assume that we need to search the location of value 31 using binary search

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 14 | 19 | 26 | 27 | 31 | 33 | 35 | 42 | 44 |
| 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  |

# Binary Search(contd...)

- ❖ First, we shall determine half of the array by using this formula :  $\text{mid} = \text{low} + (\text{high} - \text{low})/2$
- ❖ Here it is,  $0 + (9 - 0) / 2 = 4$  (integer value of 4.5). So, 4 is the mid of the array

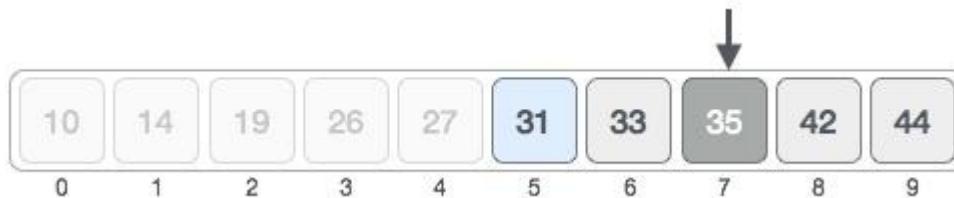


- ❖ Now we compare the value stored at location 4, with the value being searched, i.e. 31
- ❖ We find that the value at location 4 is 27, which is not a match
- ❖ As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array



# Binary Search (contd...)

- ❖ We change our low to mid + 1 and find the new mid value again
  - $\text{low} = \text{mid} + 1$
  - $\text{mid} = \text{low} + (\text{high} - \text{low}) / 2$
- ❖ Our new mid is 7 now
- ❖ We compare the value stored at location 7 with our target value 31

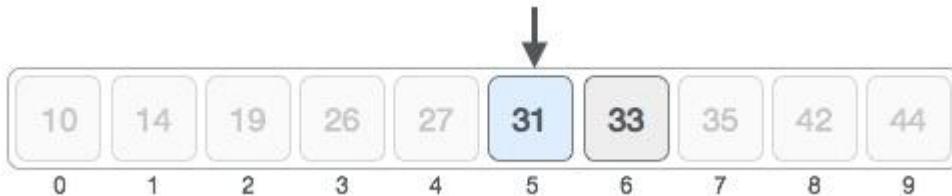


- ❖ The value stored at location 7 is not a match, rather it is more than what we are looking for
- ❖ So, the value must be in the lower part from this location



# Binary Search(contd...)

- ❖ Hence, we calculate the mid again, which is 5 this time



- ❖ On comparing the value stored at location 5 with our target value, we get a match



- ❖ We conclude that the target value 31 is stored at location 5
- ❖ Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers

# Algorithm for Binary Search

Worst Case Running Time :  
 $\log_2 n$

## Iterative Approach of Binary Search

```
binary_search_iterative(elements, target):
    start = 0
    end = len(elements) - 1
    while start <= end:
        # // indicates floor division
        midpoint = (start + end) // 2
        if elements[midpoint] == target:
            return midpoint
        else:
            if target < elements[midpoint]:
                end = midpoint - 1
            else:
                start = midpoint + 1
    return None
```

## Recursive Approach of Binary Search

```
binary_search_recursive(elements, target, start=0,
end=None):
    if end is None:
        end = len(elements) - 1
    if start > end:
        return 'Value not found in list'
    midpoint = (start + end) // 2
    if target == elements[midpoint]:
        return midpoint
    if target < elements[midpoint]:
        return binary_search_recursive(elements, target, start,
midpoint - 1)
    # target > elements[midpoint]
    return binary_search_recursive(elements, target, midpoint
+ 1, end)
```

# Analysis of Binary Search Algorithm

- ❖ **Worst case analysis:** The key is not in the array
- ❖ Let  $T(n)$  be the number of comparisons done in the worst case for an array of size  $n$
- ❖ For the purposes of analysis, assume  $n$  is a power of 2, i.e.  $n = 2^k$
- ❖ Then  $T(n) = 2 + T(n/2)$ 
  - $= 2 + 2 + T(n/2^2)$  // 2nd iteration
  - $= 2 + 2 + 2 + T(n/2^3)$  // 3rd iteration
  - ....
  - $= i * 2 + T(n/2^i)$  // ith iteration
  - ....  $= k * 2 + T(1)$
- ❖ Note that  $k = \log_2 n$ , and that  $T(1) = 2$
- ❖ So  $T(n) = 2 \log_2 n + 2 = O(\log_2 n)$
- ❖ So we expect binary search to be significantly more efficient than linear search for large values of  $n$

# Merge Sort

- ❖ Merge Sort is a sorting algorithm based on Divide and Conquer Technique
- ❖ The algorithm divides the array in two halves which are unsorted lists, recursively sorts them and finally merges the two sorted halves to get a sorted list
- ❖ The two unsorted lists are sorted by continually calling the merge-sort algorithm; we eventually get a list of size 1 which is already sorted
- ❖ The two lists of size 1 are then merged
- ❖ **Steps for applying Merge Sort :**
  - Input the total number of elements that are there in an array (number\_of\_elements)
  - Input the array (array[number\_of\_elements])
  - Then call the function MergeSort() to sort the input array. MergeSort() function sorts the array in the range [left,right] i.e. from index left to index right inclusive
  - Merge() function merges the two sorted parts
  - Sorted parts will be from [left, mid] and [mid+1, right]
  - After merging output the sorted array

# Algorithm for Merge Sort

Worst Case Analysis  $O(n \log_2 n)$

MERGE( $A, p, q, r$ )

```
1   $n_1 \leftarrow q - p + 1$ 
2   $n_2 \leftarrow r - q$ 
3  create arrays  $L[1 \square n_1 + 1]$  and  $R[1 \square n_2 + 1]$ 
4  for  $i \leftarrow 1$  to  $n_1$ 
5    do  $L[i] \leftarrow A[p + i - 1]$ 
6  for  $j \leftarrow 1$  to  $n_2$ 
7    do  $R[j] \leftarrow A[q + j]$ 
8   $L[n_1 + 1] \leftarrow \infty$ 
9   $R[n_2 + 1] \leftarrow \infty$ 
10  $i \leftarrow 1$ 
11  $j \leftarrow 1$ 
12 for  $k \leftarrow p$  to  $r$ 
13   do if  $L[i] \leq R[j]$ 
14     then  $A[k] \leftarrow L[i]$ 
15      $i \leftarrow i + 1$ 
16   else  $A[k] \leftarrow R[j]$ 
17      $j \leftarrow j + 1$ 
```

MERGE-SORT( $A, p, r$ )

```
1 if  $p < r$ 
2   then  $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
3     MERGE-SORT( $A, p, q$ )
4     MERGE-SORT( $A, q + 1, r$ )
5     MERGE( $A, p, q, r$ )
```

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

# Recurrence of Merge Sort

- ❖ The recurrence for  $T(n)$ , the worst-case running time of merge sort on  $n$  numbers. Merge sort on just one element takes constant time. When we have  $n > 1$  elements, we break down the running time as follows
  - Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus,  $D(n) = \Theta(1)$
  - Conquer: We recursively solve two subproblems, each of size  $n/2$ , which contributes  $2T(n/2)$  to the running time
  - Combine: We have already noted that the MERGE procedure on an  $n$ -element subarray takes time  $\Theta(n)$ , so  $C(n) = \Theta(n)$
- ❖ When we add the functions  $D(n)$  and  $C(n)$  for the merge sort analysis, we are adding a function that is  $\Theta(n)$  and a function that is  $\Theta(1)$
- ❖ This sum is a linear function of  $n$ , that is,  $\Theta(n)$
- ❖ Adding it to the  $2T(n/2)$  term from the "conquer" step gives the recurrence for the worst-case running time  $T(n)$  of merge sort
- ❖ Recurrence Merge Sort:

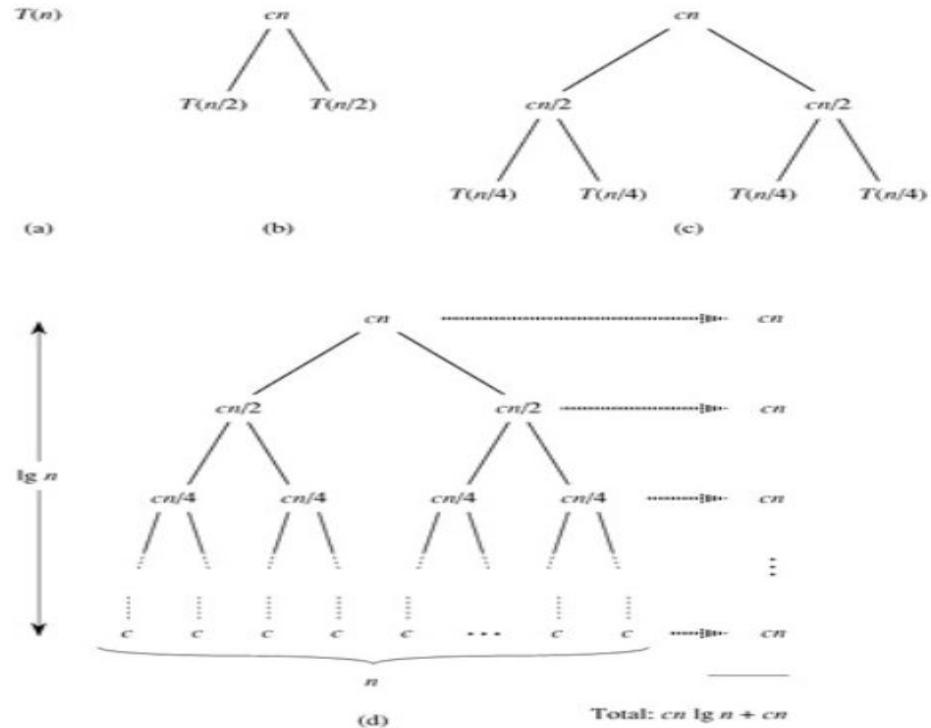
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

# Analysis of Merge Sort Using Recursion Tree

- ❖ Let us rewrite recurrence of merge sort,

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

- ❖ The fully expanded tree in part (d) has  $\lg n + 1$  levels (i.e., it has height  $\lg n$ , as indicated), and each level contributes a total cost of  $cn$
- ❖ The total cost, therefore, is  $cn \lg n + cn$ , which is  $\Theta(n \lg n)$
- ❖ Next, we add the costs across each level of the tree
- ❖ The top level has total cost  $cn$ , the next level down has total cost  $c(n/2) + c(n/2) = cn$ , the level after that has total cost  $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$ , and so on



# Analysis of Merge Sort (contd...)

- ❖ In general, the level  $i$  below the top has  $2^i$  nodes, each contributing a cost of  $c(n/2^i)$ , so that the  $i$ th level below the top has total cost  $2^i c(n/2^i) = cn$
- ❖ At the bottom level, there are  $n$  nodes, each contributing a cost of  $c$ , for a total cost of  $cn$
- ❖ The total number of levels of the "recursion tree" in Figure d is  $\lg n + 1$
- ❖ Now assume as an inductive hypothesis that the number of levels of a recursion tree for  $2^i$  nodes is  $\lg 2^i + 1 = i + 1$  (since for any value of  $i$ , we have that  $\lg 2^i = i$ )
- ❖ Because we are assuming that the original input size is a power of 2, the next input size to consider is  $2^{i+1}$
- ❖ A tree with  $2^{i+1}$  nodes has one more level than a tree of  $2^i$  nodes, and so the total number of levels is  $(i + 1) + 1 = \lg 2^{i+1} + 1$
- ❖ To compute the total cost represented by the recurrence, we simply add up the costs of all the levels
- ❖ There are  $\lg n + 1$  levels, each costing  $cn$ , for a total cost of  $cn(\lg n + 1) = cn \lg n + cn$ . Ignoring the low-order term and the constant  $c$  gives the desired result of  $\Theta(n \lg n)$

# Quick Sort

- ❖ Quick Sort is a sorting algorithm based on Divide and Conquer Method
- ❖ Quick sort works by picking a pivot element, and partitioning a given array  $A[p \dots r]$  into two non-empty sub array,  $A[p \dots q]$  and  $A[q+1 \dots r]$  such that every key in  $A[p \dots q]$  is less than or equal to every key in  $A[q+1 \dots r]$
- ❖ Then the algorithm, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side
- ❖ Finally, the algorithm recursively sorts the subarrays on left and right of pivot element
- ❖ Then the two sub arrays are sorted by recursive calls to Quick sort
- ❖ The exact position of the partition depends on the given array and index  $q$  is computed as a part of the partitioning procedure

# Algorithm for Quick Sort

Quicksort, like merge sort, is based on the divide-and-conquer paradigm introduced in [Section 2.3.1](#). Here is the three-step divide-and-conquer process for sorting a typical subarray  $A[p \square r]$ .

- **Divide:** Partition (rearrange) the array  $A[p \square r]$  into two (possibly empty) subarrays  $A[p \square q - 1]$  and  $A[q + 1 \square r]$  such that each element of  $A[p \square q - 1]$  is less than or equal to  $A[q]$ , which is, in turn, less than or equal to each element of  $A[q + 1 \square r]$ . Compute the index  $q$  as part of this partitioning procedure.
- **Conquer:** Sort the two subarrays  $A[p \square q - 1]$  and  $A[q + 1 \square r]$  by recursive calls to quicksort.
- **Combine:** Since the subarrays are sorted in place, no work is needed to combine them: the entire array  $A[p \square r]$  is now sorted.

The following procedure implements quicksort.

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

To sort an entire array  $A$ , the initial call is  $\text{QUICKSORT}(A, 1, \text{length}[A])$ .

## Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray  $A[p \square r]$  in place.

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

## Worst Case Analysis $O(n^2)$



# Analysis of Quick Sort

## ❖ Best Case Analysis

- The best thing that could happen in Quick sort would be that each partitioning stage divides the array exactly in half
- In other words, the best to be a median of the keys in  $A[p \dots r]$  every time procedure 'Partition' is called
- The procedure 'Partition' always split the array to be sorted into two equal sized arrays
- If the procedure 'Partition' produces two regions of size  $n/2$
- The recurrence relation is then
  - $T(n) = T(n/2) + T(n/2) + \Theta(n)$
  - $= 2T(n/2) + \Theta(n)$
- And from case 2 of Master theorem,
- $T(n) = (n \lg n)$

## ❖ Worst Case Analysis

- Let  $T(n)$  be the worst-case time for QUICK SORT on input size  $n$
- We have a recurrence,
- $T(n) = \max_{1 \leq q \leq n-1} (T(q) + T(n-q)) + \dots \quad (n) \text{ ----- 1}$
- where  $q$  runs from 1 to  $n-1$ , since the partition produces two regions, each having size at least 1
- Now we guess that  $T(n) \leq cn^2$  for some constant  $c$
- Substituting our guess in equation 1, we get
  - $T(n) = \max_{1 \leq q \leq n-1} (cq^2 + c(n - q)^2) + \Theta(n)$
  - $= c\max(q^2 + (n-q)^2) + \Theta(n)$
- Since the second derivative of expression  $q^2 + (n-q)^2$  with respect to  $q$  is positive
- Therefore, expression achieves a maximum over the range  $1 \leq q \leq n-1$  at one of the endpoints
- This gives the bound  $\max(q^2 + (n - q)^2) \leq n^2 + 2(n - 1)$
- Continuing with our bounding of  $T(n)$  we get
- $T(n) \leq c [n^2 - 2(n-1)] + (n) = cn^2 - 2c(n-1) + (n)$
- Since we can pick the constant so that the  $2c(n-1)$  term dominates the  $(n)$  term we have  $T(n) \leq cn^2$
- Thus the **worst-case running time of quick sort is  $(n^2)$**

# Strassen's Matrix Multiplication

Refer to PDF notes

# Maximum Sub-array Problem

- ❖ You are given a one dimensional array that may contain both positive and negative integers, find the sum of contiguous subarray of numbers which has the largest sum
- ❖ For example, if the given array is {-2, -5, **6**, -2, -3, 1, 5, -6}, then the maximum subarray sum is 7
- ❖ Using **Divide and Conquer** approach, we can find the maximum subarray sum in  $O(n \log n)$  time
- ❖ Following is the Divide and Conquer algorithm:
  1. Divide the given array in two halves
  2. Return the maximum of following three
    - a. Maximum subarray sum in left half (Make a recursive call)
    - b. Maximum subarray sum in right half (Make a recursive call)
    - c. Maximum subarray sum such that the subarray crosses the midpoint
- ❖ The lines 2.a and 2.b are simple recursive calls
- ❖ How to find maximum subarray sum such that the subarray crosses the midpoint? We can easily find the crossing sum in linear time
- ❖ The idea is simple, find the maximum sum starting from mid point and ending at some point on left of mid, then find the maximum sum starting from mid + 1 and ending with sum point on right of mid + 1
- ❖ Finally, combine the two and return

# Maximum Sub-Array Sum

```
#include <limits.h>
#include <stdio.h>
// A utility function to find maximum of two integers
int max(int a, int b) { return (a > b) ? a : b; }
// A utility function to find maximum of three integers
int max(int a, int b, int c) { return max(max(a, b), c); }
// Find the maximum possible sum in arr[] such that arr[m] is part of it
int maxCrossingSum(int arr[], int l, int m, int h)
{
    // Include elements on left of mid
    int sum = 0;
    int left_sum = INT_MIN;
    for (int i = m; i >= l; i--) {
        sum = sum + arr[i];
        if (sum > left_sum)
            left_sum = sum;
    }
    // Include elements on right of mid
    sum = 0;
    int right_sum = INT_MIN;
    for (int i = m + 1; i <= h; i++) {
        sum = sum + arr[i];
        if (sum > right_sum)
            right_sum = sum;
    }
    return max(left_sum + right_sum, left_sum, right_sum);
}
// Return sum of elements on left and right of mid returning only left_sum+right_sum
// will fail for [-2, 1]
}
```

```
// Returns sum of maximum sum subarray in aa[l..h]
int maxSubArraySum(int arr[], int l, int h)
{
    // Base Case: Only one element
    if (l == h)
        return arr[l];

    // Find middle point
    int m = (l + h) / 2;

    /* Return maximum of following three possible cases
     * a) Maximum subarray sum in left half
     * b) Maximum subarray sum in right half
     * c) Maximum subarray sum such that the subarray
     * crosses the midpoint */
    return max(maxSubArraySum(arr, l, m),
              maxSubArraySum(arr, m + 1, h),
              maxCrossingSum(arr, l, m, h));
}

/*Driver program to test maxSubArraySum*/
int main()
{
    int arr[] = { 2, 3, 4, 5, 7 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int max_sum = maxSubArraySum(arr, 0, n - 1);
    printf("Maximum contiguous sum is %dn", max_sum);
    getchar();
    return 0;
}
```

# Time Complexity Analysis of Maximum Subarray Problem

- ❖ The maxSubArraySum() is a recursive method and time complexity can be expressed as following recurrence relation
- ❖  $T(n) = 2T(n/2) + \Theta(n)$
- ❖ This recurrence is similar to Merge Sort Algorithm
- ❖ It falls in case II of Master Method
- ❖ Hence, the and solution of the above recurrence is  $O(n \log n)$

# Master Theorem

Refer to PDF notes

# Master Theorem Examples

Refer to PDF notes

# Finding Closest Pair Problem

Refer to PDF Notes

# Convex Hull Problem

Refer to PDF Notes

# References

- ❖ <https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/>
- ❖ [https://www.tutorialspoint.com/data\\_structures\\_algorithms/binary\\_search\\_algorithm.htm](https://www.tutorialspoint.com/data_structures_algorithms/binary_search_algorithm.htm)
- ❖ <https://levelup.gitconnected.com/divide-conquer-and-binary-search-f3645e43f05d>
- ❖ <https://www.geeksforgeeks.org/maximum-subarray-sum-using-divide-and-conquer-algorithm/>
- ❖

# Design and Analysis of Algorithms

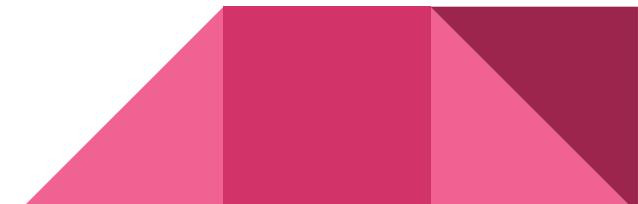
## (18CSC204J)

### Unit - 3

By:  
Aarti Sharma  
Asst Professor  
CSE Dept  
SRMIST Delhi-NCR  
Campus

# The Greedy Algorithm Introduction

- ❖ Greedy algorithms are simple and straightforward
- ❖ They are shortsighted in their approach
- ❖ A greedy algorithm is similar to a dynamic programming algorithm, but the difference is that solutions to the sub problems do not have to be known at each stage
- ❖ It is used to solve the optimization problems



# Optimization Problems

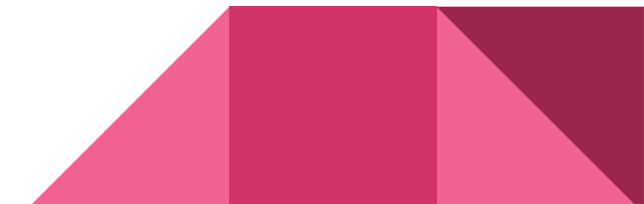
- ❖ An optimization problem is one in which you want to find, not just a solution, but the best solution
- ❖ A “greedy algorithm” sometimes works well for optimization problems
- ❖ A greedy algorithm works in phases, At each phase:
  - You take the best you can get right now, without regard for future consequences
  - You hope that by choosing a local optimum at each step, you will end up at a global optimum

# Characteristics and Features

- ❖ To construct the solution in an optimal way, Algorithm Maintains two sets
  - One contains chosen items and
  - The other contains rejected items
- ❖ Greedy algorithms make good local choices in the hope that, They result in
  - An optimal solution
  - Feasible solutions

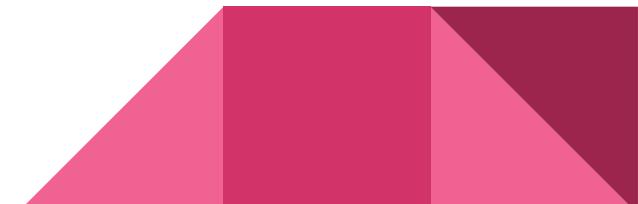
# Greedy Property

- ❖ It consists of two property:
  - **Greedy-Choice Property:**
    - It says that a globally optimal solution can be arrived at by making a locally optimal choice
  - **Optimal Substructure:**
    - An optimal global solution contains the optimal solutions of all its sub problems



# Greedy Approach

- ❖ Greedy Algorithm works by making the decision that seems most promising at any moment; it never reconsiders this decision, whatever situation may arise later
- ❖ As an example consider the problem of "Making Change"
- ❖ Coins available are:
  - 100 cents
  - 25 cents
  - 10 cents
  - 15 cents
  - 1 cent



# Examples to be solved using Greedy Algorithm

- ❖ There are various problems solved using Greedy Approach
- ❖ To name the few like:
  - Job Sequencing Problem
  - Activity Selection Problem
  - Huffman Encoding
  - Fractional Knapsack Problem
  - Tree Traversals
    - Inorder
    - Pre Order
    - Post Order
  - Minimum Spanning Tree
    - Kruskals
    - Prims

# Huffman Coding using Greedy Algorithm

- ❖ Huffman coding is a lossless data compression algorithm
- ❖ The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters
- ❖ The most frequent character gets the smallest code and the least frequent character gets the largest code.
- ❖ The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character
- ❖ This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream
- ❖ Let us understand prefix codes with a counter example
  - Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1
  - This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b
  - If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”

# Huffman Algorithm

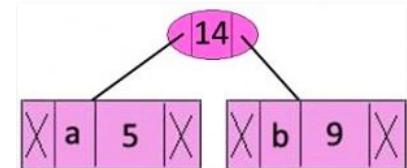
- ❖ There are mainly two major parts in Huffman Coding
  - Build a Huffman Tree from input characters
  - Traverse the Huffman Tree and assign codes to characters
- ❖ **Steps to build Huffman Tree**
- ❖ Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree
  - Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
  - Extract two nodes with the minimum frequency from the min heap
  - Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap
  - Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete

# Example for Huffman Algorithm

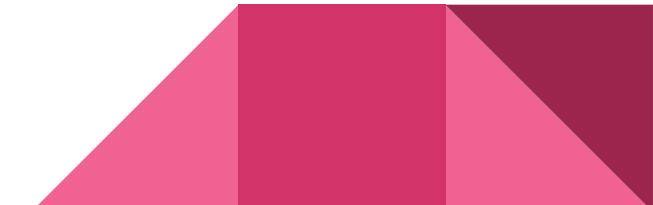
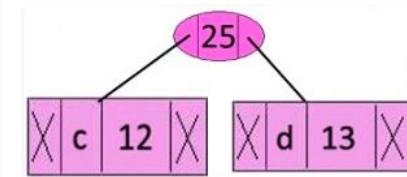
- ❖ Let's consider an example:

- **Step 1:** Build a min heap that contains 6 nodes where each node represents root of a tree with single node
- **Step 2:** Extract two minimum frequency nodes from min heap. Add a new internal node with frequency  $5 + 9 = 14$
- Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements
- **Step 3:** Extract two minimum frequency nodes from heap. Add a new internal node with frequency  $12 + 13 = 25$

| character | Frequency |
|-----------|-----------|
| a         | 5         |
| b         | 9         |
| c         | 12        |
| d         | 13        |
| e         | 16        |
| f         | 45        |

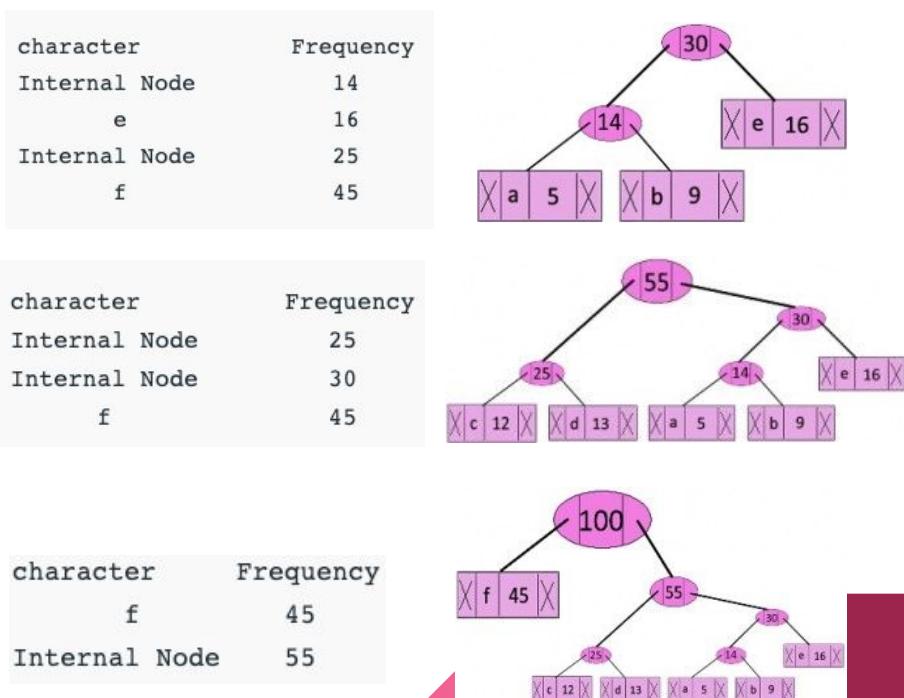


| character     | Frequency |
|---------------|-----------|
| c             | 12        |
| d             | 13        |
| Internal Node | 14        |
| e             | 16        |
| f             | 45        |



# Example (contd...)

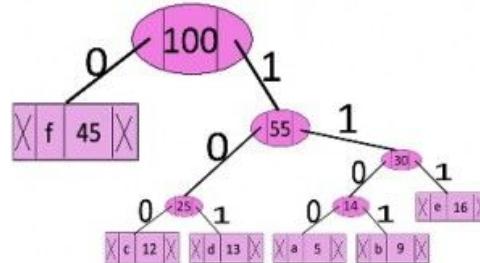
- ❖ Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes
  - **Step 4:** Extract two minimum frequency nodes. Add a new internal node with frequency  $14 + 16 = 30$
- ❖ Now min heap contains 3 nodes
  - **Step 5:** Extract two minimum frequency nodes. Add a new internal node with frequency  $25 + 30 = 55$
- ❖ Now min heap contains 2 nodes
  - **Step 6:** Extract two minimum frequency nodes. Add a new internal node with frequency  $45 + 55 = 100$



# Example (contd...)

- ❖ Now min heap contains only one node
- ❖ Since the heap contains only one node, the algorithm stops here
- ❖ **Steps to print codes from Huffman Tree:**
  - Traverse the tree formed starting from the root
  - Maintain an auxiliary array
  - While moving to the left child, write 0 to the array
  - While moving to the right child, write 1 to the array
  - Print the array when a leaf node is encountered

| character     | Frequency |
|---------------|-----------|
| Internal Node | 100       |



The codes are as follows:

| character | code-word |
|-----------|-----------|
| f         | 0         |
| c         | 100       |
| d         | 101       |
| a         | 1100      |
| b         | 1101      |
| e         | 111       |

# Time Complexity

- ❖ ***Time complexity:***  $O(n \log n)$  where  $n$  is the number of unique characters
- ❖ If there are  $n$  nodes, `extractMin()` is called  $2*(n - 1)$  times
- ❖ `extractMin()` takes  $O(\log n)$  time as it calls `minHeapify()`
- ❖ So, overall complexity is  $O(n \log n)$
- ❖ If the input array is sorted, there exists a linear time algorithm

## Applications of Huffman Coding:

1. They are used for transmitting fax and text.
2. They are used by conventional compression formats like PKZIP, GZIP, etc.

It is useful in cases where there is a series of frequently occurring characters

# Knapsack Problem

- ❖ Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible
- ❖ The knapsack problem is in combinatorial optimization problem
- ❖ It appears as a subproblem in many, more complex mathematical models of real-world problems
- ❖ One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints
- ❖ Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack
- ❖ **Applications**
  - In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example
    - Finding the least wasteful way to cut raw materials
    - Portfolio optimization
    - Cutting stock problems

# Fractional Knapsack Problem

- ❖ A thief is robbing a store and can carry a maximal weight of  $W$  into his knapsack
- ❖ There are  $n$  items available in the store and weight of  $i$ th item is  $w_i$  and its profit is  $p_i$
- ❖ What items should the thief take?
- ❖ In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit
- ❖ Hence, the objective of the thief is to maximize the profit
- ❖ Based on the nature of the items, Knapsack problems are categorized as
  - Fractional Knapsack
  - 0-1 Knapsack

# Fractional Knapsack

- ❖ In this case, items can be broken into smaller pieces, hence the thief can select fractions of items
- ❖ According to the problem statement,
  - There are  $n$  items in the store
  - Weight of  $i^{\text{th}}$  item  $w_i > 0$
  - Profit for  $i^{\text{th}}$  item  $p_i > 0$  and
  - Capacity of the Knapsack is  $W$
- ❖ In this version of Knapsack problem, items can be broken into smaller pieces
- ❖ So, the thief may take only a fraction  $x_i$  of  $i^{\text{th}}$  item
  - $0 \leq x_i \leq 1$
- ❖ The  $i^{\text{th}}$  item contributes the weight  $x_i \cdot w_i$  to the total weight in the knapsack and profit  $x_i \cdot p_i$  to the total profit
- ❖ Hence, the objective of this algorithm is to
  - Maximize  $\sum_{i=1}^n (x_i \cdot p_i)$
  - subject to constraint,  $\sum_{i=1}^n (x_i \cdot w_i) \leq W$
- ❖ It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit
- ❖ Thus, an optimal solution can be obtained by  $\sum_{i=1}^n (x_i \cdot w_i) = W$
- ❖ In this context, first we need to sort those items according to the value of  $p_i/w_i$ , so that  $p_i + 1/w_i + 1 \leq p_i/w_i$
- ❖ Here,  $x$  is an array to store the fraction of items

# Algorithm

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

for  $i = 1$  to  $n$

    do  $x[i] = 0$

    weight = 0

    for  $i = 1$  to  $n$

        if weight +  $w[i] \leq W$  then

$x[i] = 1$

            weight = weight +  $w[i]$

        else

$x[i] = (W - \text{weight}) / w[i]$

            weight =  $W$

        break

    return  $x$

## Analysis

If the provided items are already sorted into a decreasing order of  $p_i/w_i$ , then the while loop takes a time in  $O(n)$ ; Therefore, the total time including the sort is in  $O(n \log n)$

# Example

Let us consider that the capacity of the knapsack  $W = 60$  and the list of provided items are shown in the following table –

| Item                      | A   | B   | C   | D   |
|---------------------------|-----|-----|-----|-----|
| Profit                    | 280 | 100 | 120 | 120 |
| Weight                    | 40  | 10  | 20  | 24  |
| Ratio $(\frac{p_i}{w_i})$ | 7   | 10  | 6   | 5   |

As the provided items are not sorted based on  $\frac{p_i}{w_i}$ . After sorting, the items are as shown in the following table.

| Item                      | B   | A   | C   | D   |
|---------------------------|-----|-----|-----|-----|
| Profit                    | 100 | 280 | 120 | 120 |
| Weight                    | 10  | 40  | 20  | 24  |
| Ratio $(\frac{p_i}{w_i})$ | 10  | 7   | 6   | 5   |

- ❖ After sorting all the items according to  $p_i/w_i$ ,
- ❖ First all of B is chosen as weight of B is less than the capacity of the knapsack
- ❖ Next, item A is chosen, as the available capacity of the knapsack is greater than the weight of A
- ❖ Now, C is chosen as the next item
- ❖ However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of C
- ❖ Hence, fraction of C (i.e.  $(60 - 50)/20$ ) is chosen
- ❖ Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected
- ❖ The total weight of the selected items is  $10 + 40 + 20 * (10/20) = 60$
- ❖ And the total profit is  $100 + 280 + 120 * (10/20) = 380 + 60 = 440$
- ❖ This is the optimal solution. We cannot gain more profit selecting any different combination of items

# Tree Traversals

- ❖ Given a Binary tree, Traverse it using DFS using recursion
- ❖ Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways
- ❖ Generally there are 2 widely used ways for traversing trees:
  - DFS or Depth First Search
  - BFS or Breadth First Search
- ❖ DFS (Depth-first search) is technique used for traversing tree or graph
- ❖ Here backtracking is used for traversal
- ❖ In this traversal first the deepest node is visited and then backtracks to its parent node if no sibling of that node exist
- ❖ Below are the Tree traversals through DFS using recursion:

- **Inorder Traversal**

- Algorithm Inorder(tree)
  - Traverse the left subtree, i.e., call Inorder(left-subtree)
  - Visit the root
  - Traverse the right subtree, i.e., call Inorder(right-subtree)

# Tree Traversals (contd...)

- ❖ **Uses of Inorder:** In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed can be used
- ❖ **Preorder Traversal**

Algorithm Preorder(tree)

1. Visit the root
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

- ❖ **Uses of Preorder:** Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree
- ❖ **Postorder Traversal**

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root

## Uses of Postorder:

Postorder traversal is used to delete the tree. Postorder traversal is also useful to get the postfix expression of an expression tree.

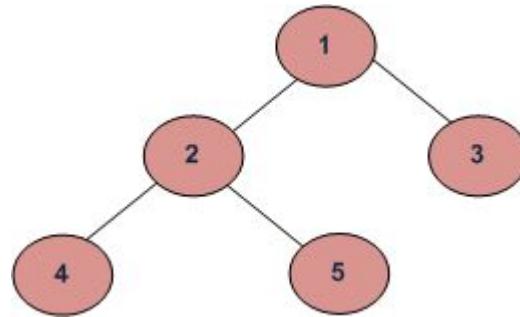
# Example of Tree Traversal

Consider the tree on the right and write its traversals.

Solution:

The Depth First Traversals of this Tree will be:

- (a) Inorder (Left, Root, Right) : 4 2 5 1 3
- (b) Preorder (Root, Left, Right) : 1 2 4 5 3
- (c) Postorder (Left, Right, Root) : 4 5 2 3 1



# Minimum Spanning Tree

- ❖ Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together
- ❖ A single graph can have many different spanning trees
- ❖ A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree
- ❖ The weight of a spanning tree is the sum of weights given to each edge of the spanning tree
- ❖ *How many edges does a minimum spanning tree has?*
  - A minimum spanning tree has  $(V - 1)$  edges where V is the number of vertices in the given graph

# Kruskals Algorithm steps

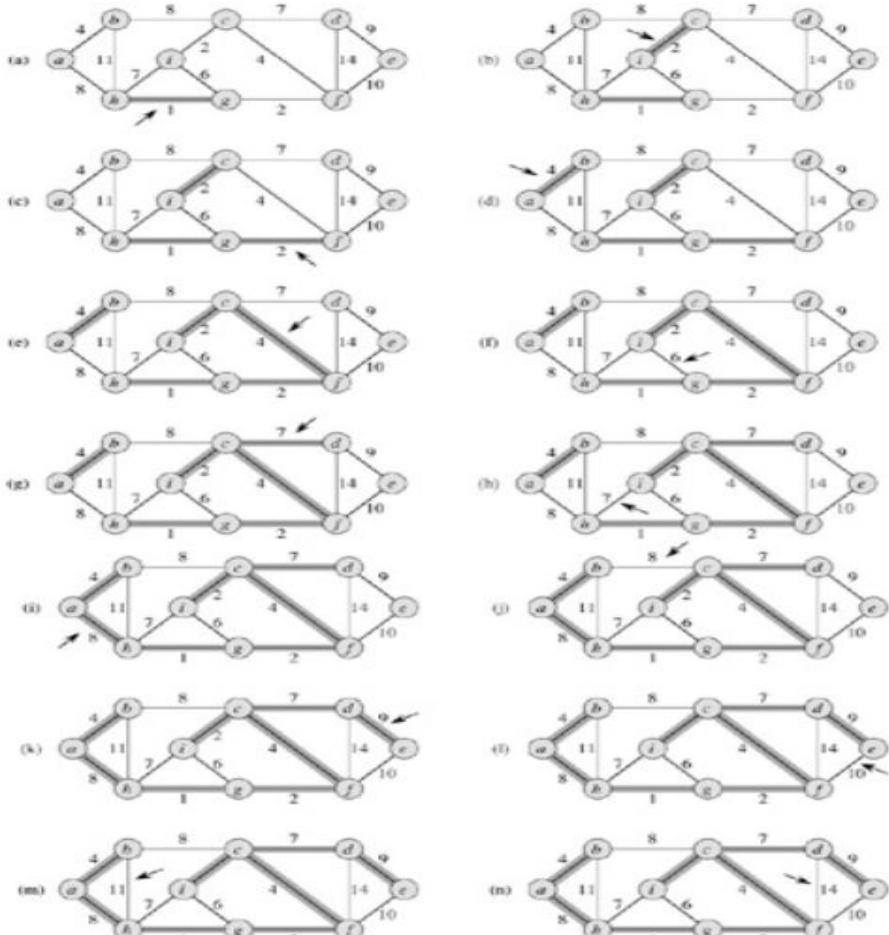
Below are the steps for finding MST using Kruskal's algorithm

- 1. Sort all the edges in non-decreasing order of their weight.*
- 2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.*
- 3. Repeat step#2 until there are  $(V-1)$  edges in the spanning tree.*

# Kruskals Algorithm

MST-KRUSKAL( $G, w$ )

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



# Time Complexity of Kruskals Algorithm

- ❖ Sorting of edges takes  $O(E \log E)$  time
- ❖ After sorting, we iterate through all edges and apply find-union algorithm
- ❖ The find and union operations can take atmost  $O(\log V)$  time
- ❖ So overall complexity is  $O(E \log E + E \log V)$  time
- ❖ The value of  $E$  can be atmost  $O(V^2)$ , so  $O(\log V)$  are  $O(\log E)$  same
- ❖ Therefore, overall time complexity is  $O(E \log E)$  or  $O(E \log V)$

# Prims Algorithm

- ❖ Prim's algorithm is also a [Greedy algorithm](#)
- ❖ It starts with an empty spanning tree
- ❖ The idea is to maintain two sets of vertices
- ❖ The first set contains the vertices already included in the MST, the other set contains the vertices not yet included
- ❖ At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges
- ❖ After picking the edge, it moves the other endpoint of the edge to the set containing MST
- ❖ A group of edges that connects two set of vertices in a graph is called [cut in graph theory](#)
- ❖ *So, at every step of Prim's algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices)*
- ❖ **How does Prim's Algorithm Work?** The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning Tree*. And they must be connected with the minimum weight edge to make it a *Minimum Spanning Tree*.

# Prims Algorithm (contd...)

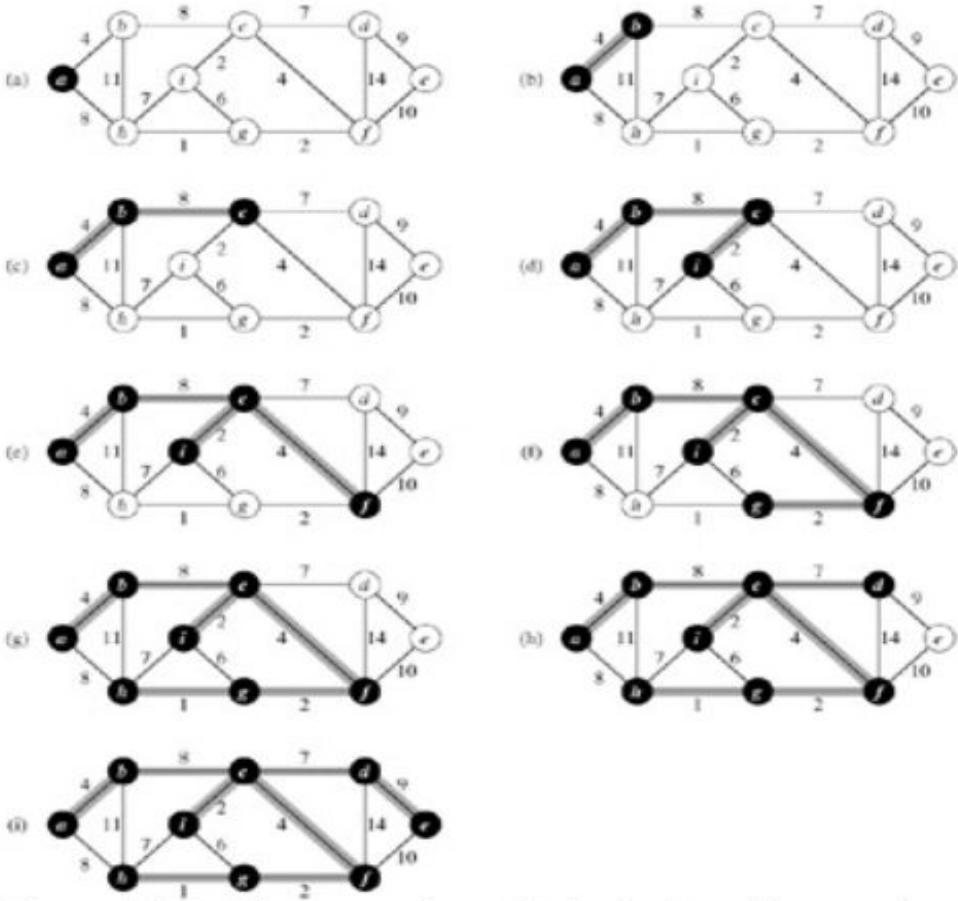
## *Algorithm*

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

# Algorithm

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

1. **For** each  $u \in V[G]$   
    **do**  $\text{key}[u] \leftarrow \infty$   
    **π**[ $u$ ]  $\leftarrow \text{NIL}$
2.  $\text{key}[r] \leftarrow 0$
3.  $Q \leftarrow V[G]$
4. **while**  $Q \neq \emptyset$   
    **do**  $u \leftarrow \text{EXTRACT-MIN}(Q)$   
        **for** each  $v \in \text{Adj}[u]$   
            **do if**  $v \in Q$  and  $w(u, v) < \text{key}[v]$   
                **then**  $\pi[v] \leftarrow u$   
                 $\text{key}[v] \leftarrow w(u, v)$



# Introduction to Dynamic Programming

- ❖ Dynamic Programming is mainly an optimization over plain **recursion**
- ❖ Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming
- ❖ The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later
- ❖ This simple optimization reduces time complexities from exponential to polynomial
- ❖ The intuition behind dynamic programming is that we trade space for time, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later
- ❖ For example, if we write simple recursive solution for **Fibonacci Numbers**, we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear
- ❖ Let's try to understand this by taking an example of Fibonacci numbers.
  - $\text{Fibonacci}(n) = 1$ ; if  $n = 0$
  - $\text{Fibonacci}(n) = 1$ ; if  $n = 1$
  - $\text{Fibonacci}(n) = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2)$
  - So, the first few numbers in this series will be: 1, 1, 2, 3, 5, 8, 13, 21... and so on!

# Comparison of Fibonacci Series Algorithm

- ❖ A code for it using pure recursion:

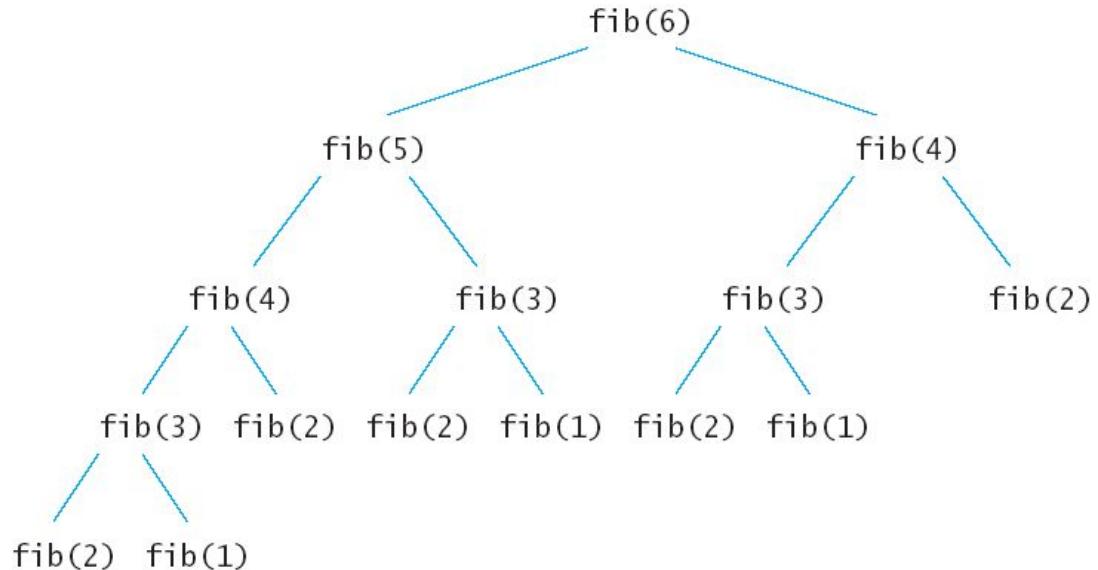
```
int fib (int n) {  
    if (n < 2)  
        return 1;  
    return fib(n-1) + fib(n-2);  
}
```

- ❖ Using Dynamic Programming approach with memoization:

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

Are we using a different recurrence relation in the two codes? No. Are we doing anything different in the two codes? Yes.

In the recursive code, a lot of values are being recalculated multiple times. We could do good with calculating each unique quantity only once. Take a look at the image to understand that how certain values were being recalculated in the recursive way:



# How to Solve a problem using DP?

- ❖ Dynamic Programming (DP) is a technique that solves some particular type of problems in Polynomial Time. Dynamic Programming solutions are faster than exponential brute method and can be easily proved for their correctness. Before we study how to think Dynamically for a problem, we need to learn:

- **Overlapping Subproblems:**

- Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems
    - Dynamic Programming is mainly used when solutions of same subproblems are needed again and again
    - In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to be recomputed
    - So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again
    - For example, [Binary Search](#) doesn't have common subproblems
    - If we take an example of following recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again

- **Optimal Substructure:**

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

# How to Solve a problem using DP?(contd...)

- ❖ There are following two different ways to store the values so that these values can be reused:
  - **Memoization (Top Down):**
    - The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions
    - We initialize a lookup array with all initial values as NIL
    - Whenever we need the solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise, we calculate the value and put the result in the lookup table so that it can be reused later
  - **Tabulation (Bottom Up):**
    - The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from table
    - For example, for the same Fibonacci number, we first calculate fib(0) then fib(1) then fib(2) then fib(3) and so on
    - So literally, we are building the solutions of subproblems bottom-up

# How to Solve a problem using DP? (contd...)

- ❖ **Steps to solve a DP:**
  - Show that the problem can be broken down into optimal sub-problems (Identify if it is a DP problem)
  - Recursively define the value of the solution by expressing it in terms of optimal solutions for smaller sub-problems (Decide a state expression with least parameters)
  - Compute the value of the optimal solution in bottom-up fashion (Formulate state relationship)
  - Construct an optimal solution from the computed information (Do tabulation or add memoization)
- ❖ **Step 1 : How to classify a problem as a Dynamic Programming Problem?**
  - Typically, all the problems that require to maximize or minimize certain quantity or counting problems that say to count the arrangements under certain condition or certain probability problems can be solved by using Dynamic Programming
  - All dynamic programming problems satisfy the overlapping subproblems property and most of the classic dynamic problems also satisfy the optimal substructure property. Once, we observe these properties in a given problem, be sure that it can be solved using DP

# How to Solve a problem using DP? (contd...)

- ❖ **Step 2 : Deciding the state**
  - DP problems are all about state and their transition, this is the most basic step which must be done very carefully because the state transition depends on the choice of state definition you make. Hence, let's see what do we mean by the term "state"
  - **State:** A state can be defined as the set of parameters that can uniquely identify a certain position or standing in the given problem. This set of parameters should be as small as possible to reduce state space
- ❖ So, our first step will be deciding a state for the problem after identifying that the problem is a DP problem
- ❖ As we know DP is all about using calculated results to formulate the final result
- ❖ So, our next step will be to find a relation between previous states to reach the current state
- ❖ **Step 3 : Formulating a relation among the states**
  - This part is the hardest part of for solving a DP problem and requires a lot of intuition, observation, and practice
- ❖ **Step 4 : Adding memoization or tabulation for the state**
  - This is the easiest part of a dynamic programming solution
  - We just need to store the state answer so that next time that state is required, we can directly use it from our memory

# Explanation of how to form a relation among states

- ❖ Let's consider this example:
- ❖ Let's think dynamically about this problem
- ❖ So, first of all, we decide a state for the given problem
- ❖ We will take a parameter n to decide state as it can uniquely identify any subproblem
- ❖ So, our state dp will look like state(n)
- ❖ Here, state(n) means the total number of arrangements to form n by using {1, 3, 5} as elements
- ❖ Now, we need to compute state(n)
- ❖ **How to do it?**
  - So here the intuition comes into action. As we can only use 1, 3 or 5 to form a given number
  - Let us assume that we know the result for  $n = 1, 2, 3, 4, 5, 6$  ; being termilogistic let us say we know the result for the
  - state ( $n = 1$ ), state ( $n = 2$ ), state ( $n = 3$ ) ..... state ( $n = 6$ )
  - Now, we wish to know the result of the state ( $n = 7$ ). See, we can only add 1, 3 and 5. Now we can get a sum total of 7 by the following 3 ways:

Given 3 numbers {1, 3, 5}, we need to tell the total number of ways we can form a number 'N' using the sum of the given three numbers.  
(allowing repetitions and different arrangements)

Total number of ways to form 6 is: 8

1+1+1+1+1+1

1+1+1+3

1+1+3+1

1+3+1+1

3+1+1+1

3+3

1+5

5+1

# Explanation (contd...)

## 1) Adding 1 to all possible combinations of state (n = 6)

Eg : [ (1+1+1+1+1+1) + 1]

[ (1+1+1+3) + 1]

[ (1+1+3+1) + 1]

[ (1+3+1+1) + 1]

[ (3+1+1+1) + 1]

[ (3+3) + 1]

[ (1+5) + 1]

[ (5+1) + 1]

## 2) Adding 3 to all possible combinations of state (n = 4);

Eg : [(1+1+1+1) + 3]

[(1+3) + 3]

[(3+1) + 3]

## 3) Adding 5 to all possible combinations of state(n = 2)

Eg : [ (1+1) + 5]

Now, think carefully and satisfy yourself that the above three cases are covering all possible ways to form a sum total of 7;

Therefore, we can say that result for

$$\text{state}(7) = \text{state}(6) + \text{state}(4) + \text{state}(2)$$

or

$$\text{state}(7) = \text{state}(7-1) + \text{state}(7-3) + \text{state}(7-5)$$

In general,

$$\text{state}(n) = \text{state}(n-1) + \text{state}(n-3) + \text{state}(n-5)$$

# Coding for above problem using DP

- ❖ Code using above strategy:

```
int solve(int n)
{
    // base case
    if (n < 0)
        return 0;
    if (n == 0)
        return 1;

    return solve(n-1) + solve(n-3) + solve(n-5);
}
```

- ❖ The above code seems exponential as it is calculating the same state again and again
- ❖ So, we just need to add a memoization

- ❖ Adding memoization to the above code

```
// initialize to -1
int dp[MAXN];
// this function returns the number of
// arrangements to form 'n'
int solve(int n)
{
    // base case
    if (n < 0)
        return 0;
    if (n == 0)
        return 1;
    // checking if already calculated
    if (dp[n]!=-1)
        return dp[n];
    // storing the result and returning
    return dp[n] = solve(n-1) + solve(n-3) +
    solve(n-5);
}
```

# Examples that can be solved using Dynamic Programming

- ❖ Few problems that can be solved using DP:
  - 0/1 Knapsack Problem
  - Matrix Chain Multiplication (MCM)
  - Longest Common Subsequence (LCS)
  - Optimal Binary Search Tree (OBST)
  - All Pair Shortest Path Algorithm:
    - Floyd-Warshall Algorithm
    - Bellman-Ford Algorithm
  - Subset Sum Problem
  - Scheduling Jobs to maximize CPU utilization

# 0-1 Knapsack Problem

- ❖ Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack
- ❖ In other words, given two integer arrays  $\text{val}[0..n-1]$  and  $\text{wt}[0..n-1]$  which represent values and weights associated with n items respectively
- ❖ Also given an integer W which represents knapsack capacity, find out the maximum value subset of  $\text{val}[]$  such that sum of the weights of this subset is smaller than or equal to W
- ❖ You cannot break an item, either pick the complete item or don't pick it (0-1 property)

## 0-1 Knapsack Problem

$\text{value}[] = \{60, 100, 120\};$

$\text{weight}[] = \{10, 20, 30\};$

$W = 50;$

Solution: 220

Weight = 10; Value = 60;

Weight = 20; Value = 100;

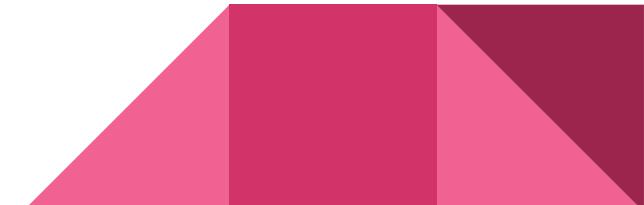
Weight = 30; Value = 120;

Weight = (20+10); Value = (100+60);

Weight = (30+10); Value = (120+60);

Weight = (30+20); Value = (120+100);

Weight = (30+20+10) > 50



# 0-1 Knapsack Problem DP solution

- ❖ Re-computation of same subproblems can be avoided by constructing a temporary array  $K[][]$  in bottom-up manner
- ❖ Following is Dynamic Programming based implementation
- ❖ In the Dynamic programming we will work considering the same cases as mentioned in the recursive approach
- ❖ In a  $DP[][]$  table let's consider all the possible weights from '1' to 'W' as the columns and weights that can be kept as the rows
- ❖ The state  $DP[i][j]$  will denote maximum value of 'j-weight' considering all values from '1' to  $i^{\text{th}}$
- ❖ So if we consider ' $w_i$ ' (weight in ' $i^{\text{th}}$ ' row) we can fill it in all columns which have 'weight values  $> w_i$ '
- ❖ Now two possibilities can take place:
  - Fill ' $w_i$ ' in the given column
  - Do not fill ' $w_i$ ' in the given column
- ❖ Now we have to take a maximum of these two possibilities, formally if we do not fill ' $i^{\text{th}}$ ' weight in ' $j^{\text{th}}$ ' column then  $DP[i][j]$  state will be same as  $DP[i-1][j]$  but if we fill the weight,  $DP[i][j]$  will be equal to the value of ' $w_i$ ' + value of the column weighing ' $j-w_i$ ' in the previous row
- ❖ So we take the maximum of these two possibilities to fill the current state

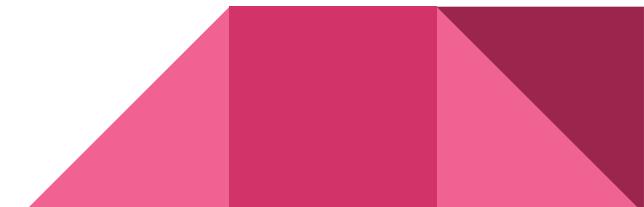
# Algorithm for 0-1 Knapsack Problem

**KNAPSACK (n, W)**

1. for  $w = 0, W$
2. do  $T [0,w] \leftarrow 0$
3. for  $i=0, n$
4.     do  $T [i, 0] \leftarrow 0$
5. for  $i = 0, n$
6.     for  $j = 0, W$
7.         do if  $(w_i \leq w \text{ & } v_i + T [i-1, j - w_i] > T [i - 1, j])$
8.             then  $T [i, W] \leftarrow v_i + T [i - 1, j - w_i]$
9.         else  $T [i, W] \leftarrow T [i - 1, j]$

## Time Complexity-

- ❖ Each entry of the table requires constant time  $\theta(1)$  for its computation
- ❖ It takes  $\theta(nw)$  time to fill  $(n+1)(w+1)$  table entries
- ❖ It takes  $\theta(n)$  time for tracing the solution since tracing process traces the  $n$  rows
- ❖ Thus, overall  $\theta(nw)$  time is taken to solve 0/1 knapsack problem using dynamic programming



# 0-1 Knapsack Example

- ❖ A thief enters a house for robbing it. He can carry a maximal weight of 5 kg into his bag. There are 4 items in the house with the following weights and values. What items should thief take if he either takes the item completely or leaves it completely?

- ❖ **Given-**

- Knapsack capacity ( $w$ ) = 5 kg
- Number of items ( $n$ ) = 4

- ❖ **Step-01:**

- Draw a table say 'T' with  $(n+1) = 4 + 1 = 5$  number of rows and  $(w+1) = 5 + 1 = 6$  number of columns
- Fill all the boxes of 0<sup>th</sup> row and 0<sup>th</sup> column with 0

- ❖ **Step-02:**

- Start filling the table row wise top to bottom from left to right using the formula-
  - $T(i, j) = \max \{ T(i-1, j), \text{value}_i + T(i-1, j - \text{weight}_i) \}$

- ❖ **Step-03:**

- The last entry represents the maximum possible value that can be put into the knapsack
- So, maximum possible value that can be put into the knapsack = 7

- ❖ **Step-04:**

- We mark the rows labelled "1" and "2".
- Thus, items that must be put into the knapsack to obtain the maximum value 7 are- **Item-1 and Item-2**

| Item          | Weight (kg) | Value (\$) |
|---------------|-------------|------------|
| Mirror        | 2           | 3          |
| Silver nugget | 3           | 4          |
| Painting      | 4           | 5          |
| Vase          | 5           | 6          |

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
|   | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 |   |   |   |   |   |
| 2 | 0 |   |   |   |   |   |
| 3 | 0 |   |   |   |   |   |
| 4 | 0 |   |   |   |   |   |

T-Table

# 0-1 Knapsack Example (contd...)

## Finding T(1,1)-

We have,

- $i = 1, j = 1, (\text{value})_i = (\text{value})_1 = 3, (\text{weight})_i = (\text{weight})_1 = 2$

Substituting the values, we get-

$$T(1,1) = \max \{ T(1-1, 1), 3 + T(1-1, 1-2) \}$$

$$T(1,1) = \max \{ T(0,1), 3 + T(0,-1) \}$$

$$T(1,1) = T(0,1) \{ \text{Ignore } T(0,-1) \}$$

$$T(1,1) = 0$$

## Finding T(1,2)-

We have,

- $i = 1, j = 2, (\text{value})_i = (\text{value})_1 = 3, (\text{weight})_i = (\text{weight})_1 = 2$

Substituting the values, we get-

$$T(1,2) = \max \{ T(1-1, 2), 3 + T(1-1, 2-2) \}$$

$$T(1,2) = \max \{ T(0,2), 3 + T(0,0) \}$$

$$T(1,2) = \max \{ 0, 3+0 \}$$

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

## Finding T(1,3)-

We have,

- $i = 1, j = 3, (\text{value})_i = (\text{value})_1 = 3, (\text{weight})_i = (\text{weight})_1 = 2$

Substituting the values, we get-

$$T(1,3) = \max \{ T(1-1, 3), 3 + T(1-1, 3-2) \}$$

$$T(1,3) = \max \{ T(0,3), 3 + T(0,1) \}$$

$$T(1,3) = \max \{ 0, 3+0 \}$$

$$T(1,3) = 3$$

## Finding T(1,4)-

We have,

- $i = 1, j = 4, (\text{value})_i = (\text{value})_1 = 3, (\text{weight})_i = (\text{weight})_1 = 2$

Substituting the values, we get-

$$T(1,4) = \max \{ T(1-1, 4), 3 + T(1-1, 4-2) \}$$

$$T(1,4) = \max \{ T(0,4), 3 + T(0,2) \}$$

$$T(1,4) = \max \{ 0, 3+0 \}$$

$$T(1,4) = 3$$

## Finding T(1,5)-

We have,

- $i = 1, j = 5, (\text{value})_i = (\text{value})_1 = 3, (\text{weight})_i = (\text{weight})_1 = 2$

Substituting the values, we get-

$$T(1,5) = \max \{ T(1-1, 5), 3 + T(1-1, 5-2) \}$$

$$T(1,5) = \max \{ T(0,5), 3 + T(0,3) \}$$

$$T(1,5) = \max \{ 0, 3+0 \}$$

$$T(1,5) = 3$$

Similarly, compute all the entries

|   | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 3 | 3 | 3 | 3 |
| 2 | 0 | 0 | 3 | 4 | 4 | 7 |
| 3 | 0 | 0 | 3 | 4 | 5 | 7 |
| 4 | 0 | 0 | 3 | 4 | 5 | 7 |

T-Table

# Matrix Chain Multiplication (MCM)

- ❖ Matrix Chain Multiplication Problem can be stated as "find the optimal parenthesization of a chain of matrices to be multiplied such that the number of scalar multiplication is minimized"
- ❖ It is a Method under Dynamic Programming in which previous output is taken as input for next
- ❖ Here, Chain means one matrix's column is equal to the second matrix's row [always]
- ❖ In general,
  - If  $A = [a_{ij}]$  is a  $p \times q$  matrix
  - $B = [b_{ij}]$  is a  $q \times r$  matrix
  - $C = [c_{ij}]$  is a  $p \times r$  matrix
- ❖ Then,
  - $AB = C$  if  $c_{ij} = \sum_{k=1}^q a_{ik} b_{kj}$
- ❖ Given following matrices  $\{A_1, A_2, A_3, \dots, A_n\}$  and we have to perform the matrix multiplication, which can be accomplished by a series of matrix multiplications
  - $A_1 \times A_2 \times A_3 \times \dots \times A_n$
- ❖ Matrix Multiplication operation is **associative** in nature rather commutative. By this, we mean that we have to follow the above matrix order for multiplication but we are free to **parenthesize** the above multiplication depending upon our need
- ❖ In general, for  $1 \leq i \leq p$  and  $1 \leq j \leq r$
- ❖ It can be observed that the total entries  $C[i, j] = \sum_{k=1}^q A[i, k]B[k, j]$  matrix is of dimension  $p \times r$ . Also each entry takes  $O(q)$  times to compute, thus the total time to compute all possible entries for the matrix 'C' which is a multiplication of 'A' and 'B' is proportional to the product of the dimension  $p \cdot q \cdot r$
- ❖ It is also noticed that we can save the number of operations by reordering the parenthesis

# Matrix Chain Example

- ❖ Let us have 3 matrices,  $A_1, A_2, A_3$  of order  $(10 \times 100)$ ,  $(100 \times 5)$  and  $(5 \times 50)$  respectively
- ❖ Three Matrices can be multiplied in two ways:
  - $\mathbf{A_1, (A_2, A_3)}$ : First multiplying( $A_2$  and  $A_3$ ) then multiplying and resultant with  $A_1$
  - $\mathbf{(A_1, A_2), A_3}$ : First multiplying( $A_1$  and  $A_2$ ) then multiplying and resultant with  $A_3$
- ❖ No of Scalar multiplication in Case 1 will be:
  - $(100 \times 5 \times 50) + (10 \times 100 \times 50) = 25000 + 50000 = 75000$
- ❖ No of Scalar multiplication in Case 2 will be:
  - $(100 \times 10 \times 5) + (10 \times 5 \times 50) = 5000 + 2500 = 7500$
- ❖ To find the best possible way to calculate the product, we could simply parenthesize the expression in every possible fashion and count each time how many scalar multiplication are required
- ❖ **Number of ways for parenthesizing the matrices:**
  - There are very large numbers of ways of parenthesizing these matrices. If there are  $n$  items, there are  $(n-1)$  ways in which the outermost pair of parenthesis can place
  - $(A_1) (A_2, A_3, A_4, \dots, A_n)$
  - Or  $(A_1, A_2) (A_3, A_4, \dots, A_n)$
  - Or  $(A_1, A_2, A_3) (A_4, \dots, A_n)$
  - .....
  - Or  $(A_1, A_2, A_3, \dots, A_{n-1}) (A_n)$

# Matrix Chain Example (contd...)

- ❖ It can be observed that after splitting the kth matrices, we are left with two parenthesized sequence of matrices: one consist 'k' matrices and another consist 'n-k' matrices
- ❖ Now there are 'L' ways of parenthesizing the left sublist and 'R' ways of parenthesizing the right sublist then the Total will be L.R:

$$\triangleright p(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} p(k)p(n-k) & \text{if } n \geq 2 \end{cases}$$

- ❖ Also  $p(n) = c(n-1)$  where  $c(n)$  is the nth **Catalan number**

$$\triangleright c(n) = \frac{1}{n+1} \binom{2n}{n}$$

- ❖ On applying Stirling's formula we have

$$\triangleright c(n) = \left( \frac{4^n}{n^{1.5}} \right)$$

# MCM Algorithm

## MATRIX-CHAIN-ORDER (p)

```
1. n = length[p]-1
2. for i ← 1 to n
3.   do m [i, i] ← 0
4. for l ← 2 to n // l is the chain length
5.   do for i ← 1 to n-l + 1
6.     do j ← i+l -1
7.     m[i,j] ← ∞
8.     for k ← i to j-1
9.       do q ← m [i, k] + m [k + 1, j] + pi-1 pk pj
10.    If q < m [i,j]
11.      then m [i,j] ← q
12.      s [i,j] ← k
13. return m and s
```

## PRINT-OPTIMAL-PARENS (s, i, j)

1. if i=j  
2. then print "A"  
3. else print "("  
4. PRINT-OPTIMAL-PARENS (s, i, s [i, j])  
5. PRINT-OPTIMAL-PARENS (s, s [i, j] + 1, j)  
6. print ")"

**Analysis:** There are three nested loops. Each loop executes a maximum n times.

1. l, length, O (n) iterations.
2. i, start, O (n) iterations.
3. k, split point, O (n) iterations

Body of loop constant complexity

**Total Complexity is: O (n<sup>3</sup>)**

# Example of MCM

**Question:** P {7, 1, 5, 4, 2}

**Solution:** Here, P is the array of a dimension of matrices.  
So here we will have 4 matrices:

A<sub>17x1</sub> A<sub>21x5</sub> A<sub>35x4</sub> A<sub>44x2</sub>

i.e.

First Matrix A<sub>1</sub> have dimension 7 x 1

Second Matrix A<sub>2</sub> have dimension 1 x 5

Third Matrix A<sub>3</sub> have dimension 5 x 4

Fourth Matrix A<sub>4</sub> have dimension 4 x 2

Let say, From P = {7, 1, 5, 4, 2} - (Given)

And P is the Position

p<sub>0</sub> = 7, p<sub>1</sub> = 1, p<sub>2</sub> = 5, p<sub>3</sub> = 4, p<sub>4</sub> = 2.

Length of array P = number of elements in P

∴ length (p) = 5

From step 3

Follow the steps in Algorithm in Sequence

According to Step 1 of Algorithm Matrix-Chain-Order

**Step - 1**

n ← length [p]-1

Where n is the total number of elements

And length [p] = 5 ∴ n = 5 - 1 = 4

**n = 4**

Now we construct two tables m and s.

Table m has dimension [1.....n, 1.....n]

Table s has dimension [1.....n-1, 2.....n]

|   | 1 | 2  | 3  | 4  |
|---|---|----|----|----|
| 1 | 0 | 35 | 48 | 42 |
| 2 |   | 0  | 20 | 28 |
| 3 |   |    | 0  |    |
| 4 |   |    |    | 0  |

m-Table  
[1....n, 1.....n]

|   | 2 | 3 | 4 |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 2 |   | 2 | 3 |
| 3 |   |   | 3 |

n-Table  
[1....n-1, 2.....n]

# Example of MCM (contd...)

Now we will make use of only s table to get an optimal solution.

Use of step 4 Algorithm

Initial call of step 4 is (s, 1, n)

Where i=1

j = n and n= 4

i ≠ j

else part

print "("

Print-optimal-Parens (s, i, s [i, j])

Print-optimal-Parens (s, 1, 4)

here i==j yes

print "A<sub>1</sub>"

Print-optimal-Parens (s, s [i,j] + 1,j)

Print-optimal-Parens (s, 2, 4)

print "("

Print-optimal (s, 2, 3)

Print ")"

Print-optimal (s, 4, 4)

Print ")"

Starting from the beginning we are parenthesizing Matrix Chain Multiplication:

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

# Longest Common Subsequence (LCS)

- ❖ A subsequence of a given sequence is just the given sequence with some elements left out
- ❖ Given two sequences X and Y, we say that the sequence Z is a common sequence of X and Y if Z is a subsequence of both X and Y.
- ❖ In the longest common subsequence problem, we are given two sequences  $X = (x_1 x_2 \dots x_m)$  and  $Y = (y_1 y_2 \dots y_n)$  and wish to find a maximum length common subsequence of X and Y
- ❖ LCS Problem can be solved using dynamic programming
- ❖ **Characteristics of LCS:**
  - A brute-force approach we find all the subsequences of X and check each subsequence to see if it is also a subsequence of Y, this approach requires exponential time making it impractical for the long sequence
  - Given a sequence  $X = (x_1 x_2 \dots x_m)$  we define the ith prefix of X for  $i=0, 1, 2 \dots m$  as  $X_i = (x_1 x_2 \dots x_i)$ . For example: if  $X = (A, B, C, B, C, A, B, C)$  then  $X_4 = (A, B, C, B)$
- ❖ **Step 1: Optimal Substructure of an LCS:** Let  $X = (x_1 x_2 \dots x_m)$  and  $Y = (y_1 y_2 \dots y_n)$  be the sequences and let  $Z = (z_1 z_2 \dots z_k)$  be any LCS of X and Y
  - 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 is an LCS of  $X_{m-1}$  and Y
  - If  $x_m \neq y_n$ , then  $z_k \neq y_n$  implies that Z is an LCS of X and  $Y_{n-1}$

# LCS (contd...)

- ❖ **Step 2: Recursive Solution:** LCS has overlapping subproblems property because to find LCS of X and Y, we may need to find the LCS of  $X_{m-1}$  and  $Y_{n-1}$ . If  $x_m \neq y_n$ , then we must solve two subproblems finding an LCS of X and  $Y_{n-1}$ . Whenever one of these LCS's longer is an LCS of x and y. But each of these subproblems has the subproblems of finding the LCS of  $X_{m-1}$  and  $Y_{n-1}$ 
  - Let  $c[i,j]$  be the length of LCS of the sequence  $X_i$  and  $Y_j$ . If either  $i=0$  and  $j=0$ , one of the sequences has length 0, so the LCS has length 0
  - The optimal substructure of the LCS problem given the recurrence formula
- ❖  
❖  
❖  
❖  
$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j - 1], c[i - 1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$
- ❖ **Step 3: Computing the length of an LCS:** let two sequences  $X = (x_1, x_2, \dots, x_m)$  and  $Y = (y_1, y_2, \dots, y_n)$  as inputs. It stores the  $c[i,j]$  values in the table  $c[0, \dots, m, 0, \dots, n]$ . Table  $b[1, \dots, m, 1, \dots, n]$  is maintained which help us to construct an optimal solution.  $c[m, n]$  contains the length of an LCS of X, Y
- ❖ **Step 4: Constructing an LCS:** The initial call is PRINT-LCS ( $b$ ,  $X$ ,  $X.length$ ,  $Y.length$ )

# LCS Algorithm

## LCS-LENGTH (X, Y)

```
1. m ← length [X]
2. n ← length [Y]
3. for i ← 0 to m
4.     do c [i,0] ← 0
5. for j ← 0 to m
6.     do c [0,j] ← 0
7. for i ← 1 to m
8.     do for j ← 1 to n
9.         do if  $x_i = y_j$ 
10.            then c [i,j] ← c [i-1,j-1] + 1
11.            b [i,j] ← " $\nwarrow$ "
12.        else if c[i-1,j] ≥ c[i,j-1]
13.            then c [i,j] ← c [i-1,j]
14.            b [i,j] ← " $\uparrow$ "
15.        else c [i,j] ← c [i,j-1]
16.            b [i,j] ← " $\leftarrow$ "
17. return c and b.
```

## PRINT-LCS (b, x, i, j)

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

# Example of LCS

**Example:** Determine the LCS of  $(1,0,0,1,0,1,0,1)$  and  $(0,1,0,1,1,0,1,1,0)$ .

**Solution:** let  $X = (1,0,0,1,0,1,0,1)$  and  $Y = (0,1,0,1,1,0,1,1,0)$ .

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

We are looking for  $c[8, 9]$ . The following table is built.

$$x = (1,0,0,1,0,1,0,1)$$

$$y = (0,1,0,1,1,0,1,1,0)$$

| i | j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| y | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |   |
| 0 | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 2 | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 3 | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 |
| 4 | 1 | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 4 | 4 | 4 |
| 5 | 0 | 0 | 1 | 2 | 3 | 3 | 3 | 4 | 4 | 4 | 5 |
| 6 | 1 | 0 | 1 | 2 | 3 | 4 | 4 | 4 | 5 | 5 | 5 |
| 7 | 0 | 0 | 1 | 2 | 3 | 4 | 4 | 5 | 5 | 5 | 6 |
| 8 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 6 | 6 | 6 |

From the table we can deduct that  $\text{LCS} = 6$ . There are several such sequences, for instance  $(1,0,0,1,1,0)$   $(0,1,0,1,0,1)$  and  $(0,0,1,1,0,1)$

# Optimal Binary Search Tree (OBST)

- ❖ A Binary Search Tree (BST) is a tree where the key values are stored in the internal nodes
- ❖ The external nodes are null nodes
- ❖ The keys are ordered lexicographically, i.e. for each internal node all the keys in the left sub-tree are less than the keys in the node, and all the keys in the right sub-tree are greater
- ❖ When we know the frequency of searching each one of the keys, it is quite easy to compute the expected cost of accessing each node in the tree
- ❖ An optimal binary search tree is a BST, which has minimal expected cost of locating each node
- ❖ Dynamic Approach Formula

$$c[i, j] = \begin{cases} \min_{i \leq k \leq j} \{ c[i, k - 1] + c[k + 1, j] \} + \sum_{l=i}^j p_l & \text{if } i < j \\ p_i & \text{if } i = j \\ 0 & \text{if } i > j \end{cases}$$

where  $c[i, j]$  in which  $c$  stands for cost, “  $p_i$  ” stands for frequency of node  $i$

# Optimal BST DP Approach

- Optimal BST  $T$  must have subtree  $T'$  for keys  $k_i \dots k_j$  which is optimal for those keys
  - Cut and paste proof: if  $T'$  not optimal, improving it will improve  $T$ , a contradiction
- Algorithm for finding optimal tree for sorted, distinct keys  $k_i \dots k_j$ :
  - For each possible root  $k_r$  for  $i \leq r \leq j$
  - Make optimal subtree for  $k_i, \dots, k_{r-1}$
  - Make optimal subtree for  $k_{r+1}, \dots, k_j$
  - Select root that gives best total tree
- Formula:  $e(i, j) = \text{expected number of comparisons for optimal tree for keys } k_i \dots k_j$

$$e(i, j) = \begin{cases} 0, & \text{if } i = j + 1 \\ \min_{i \leq r \leq j} \{ e(i, r - 1) + e(r + 1, j) + w(i, j) \}, & \text{if } i \leq j \end{cases}$$

- where  $w(i, j) = \sum_{k=i}^j p_k$  is the increase in cost if  $k_i \dots k_j$  is a subtree of a node
- Work bottom up and remember solution

# Optimal BST DP Algorithm

- Brute Force: try all tree configurations
  - $\Omega(4^n / n^{3/2})$  different BSTs with  $n$  nodes
- DP: bottom up with table: for all possible contiguous sequences of keys and all possible roots, compute optimal subtrees

```
for size in 1 .. n loop          -- All sizes of sequences
    for i in 1 .. n-size+1 loop   -- All starting points of sequences
        j := i + size - 1
        e(i, j) := float'max;
        for r in i .. j loop       -- All roots of sequence ki .. kj
            t := e(i, r-1) + e(r+1, j) + w(i, j)
            if t < e(i, j)  then
                e(i, j) := t
                root(i, j) := r
            end if
        end loop
    end loop
end loop
```

- $\Theta(n^3)$

# Algorithm OBST

Optimal-Binary-Search-Tree( $p, q, n$ )

$e[1 \dots n + 1, 0 \dots n], w[1 \dots n + 1, 0 \dots n], root[1 \dots n + 1, 0 \dots n]$

for  $i = 1$  to  $n + 1$  do

$e[i, i - 1] := q_i - 1$

$w[i, i - 1] := q_i - 1$

for  $l = 1$  to  $n$  do

    for  $i = 1$  to  $n - l + 1$  do

$j = i + l - 1$   $e[i, j] := \infty$

$w[i, j] := w[i, i - 1] + p_j + q_j$

        for  $r = i$  to  $j$  do

$t := e[i, r - 1] + e[r + 1, j] + w[i, j]$

            if  $t < e[i, j]$

$e[i, j] := t$

$root[i, j] := r$

return  $e$  and  $root$

- ❖  $n$  number of distinct keys  $< k_1, k_2, k_3, \dots k_n >$
- ❖ The probability of accessing a key  $K_i$  is  $p_i$
- ❖ Some dummy keys ( $d_0, d_1, d_2, \dots d_n$ ) are added as some searches may be performed for the values which are not present in the Key set K. We assume, for each dummy key  $d_i$  probability of access is  $q_i$
- ❖ The algorithm requires  $O(n^3)$  time, since three nested for loops are used. Each of these loops takes on at most  $n$  values

# References

- ❖ [https://www.tutorialspoint.com/design\\_and\\_analysis\\_of\\_algorithms/design\\_and\\_analysis\\_of\\_algorithms\\_fractional\\_knapsack.htm#:~:text=The%20Greedy%20algorithm%20could%20be,reasonably%20in%20a%20good%20time](https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_fractional_knapsack.htm#:~:text=The%20Greedy%20algorithm%20could%20be,reasonably%20in%20a%20good%20time)
- ❖ <https://www.geeksforgeeks.org/fractional-knapsack-problem/>
- ❖ <https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/>
- ❖ <https://www.slideshare.net/Amjad061/greedy-algorihm>
- ❖ <https://www.geeksforgeeks.org/dfs-traversal-of-a-tree-using-recursion/>
- ❖ <https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/>
- ❖ <https://www.hackerearth.com/practice/algorithms/dynamic-programming/introduction-to-dynamic-programming-1/tutorial/>
- ❖ <https://www.geeksforgeeks.org/tabulation-vs-memoization/>
- ❖ <https://www.geeksforgeeks.org/dynamic-programming/#concepts>
- ❖ <https://www.gatevidyalay.com/tag/0-1-knapsack-problem-ppt/>
- ❖ <https://www.javatpoint.com/matrix-chain-multiplication-algorithm>
- ❖ [https://www.tutorialspoint.com/design\\_and\\_analysis\\_of\\_algorithms/design\\_and\\_analysis\\_of\\_algorithms\\_optimal\\_cost\\_binary\\_search\\_trees.htm](https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_optimal_cost_binary_search_trees.htm)
- ❖ <https://www.youtube.com/watch?v=vLS-zRCHo-Y>
- ❖ <https://medium.com/@dhruvilkotecha/optimal-binary-search-tree-b91665db450c>
- ❖ <https://www.radford.edu/~nokie/classes/360/dp-opt-bst.html>

# Design and Analysis of Algorithms

## (18CSC204J)

### Unit - 4

By:  
Aarti Sharma  
Asst Professor  
CSE Dept  
SRMIST Delhi-NCR  
Campus

# Introduction to Backtracking

- ❖ Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree)
- ❖ According to the wiki definition,
  - Backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem
- ❖ Backtracking is a systematic way of trying out different sequences of decisions until we find one that "works"
- ❖ There are three types of problems in backtracking –
  1. Decision Problem – In this, we search for a feasible solution
  2. Optimization Problem – In this, we search for the best solution
  3. Enumeration Problem – In this, we find all feasible solutions
- ❖ **In recursion, the function calls itself until it reaches a base case whereas In backtracking, we use recursion to explore all the possibilities until we get the best result for the problem**

# Backtracking (contd...)

- ❖ Backtracking is undoubtedly quite simple - we "explore" each node, as follows:
- ❖ **To "explore" node N:**
  - If N is a goal node, return "success"
  - If N is a leaf node, return "failure"
  - For each child C of N,
    - Explore C
    - If C was successful, return "success"
  - Return "failure"
- ❖ Backtracking algorithm determines the solution by systematically searching the solution space for the given problem
- ❖ Backtracking is a **depth-first search** with any bounding function
- ❖ All solution using backtracking is needed to satisfy a complex set of constraints which may be explicit or implicit
- ❖ **Explicit Constraint** is ruled, which restrict each vector element to be chosen from the given set
- ❖ **Implicit Constraint** is ruled, which determine which each of the tuples in the solution space, actually satisfy the criterion function

# Advantages

- ❖ Comparison with the Dynamic Programming, Backtracking Approach is more effective in some cases
- ❖ Backtracking Algorithm is the best option for solving tactical problem
- ❖ Also Backtracking is effective for constraint satisfaction problem
- ❖ In greedy Algorithm, getting the Global Optimal Solution is a long procedure and depends on user statements but in Backtracking It Can Easily getable
- ❖ Backtracking technique is simple to implement and easy to code
- ❖ Different states are stored into stack so that the data or Info can be usable anytime
- ❖ The accuracy is granted

# Disadvantages

- ❖ Backtracking Approach is not efficient for solving strategic Problem
- ❖ The overall runtime of Backtracking Algorithm is normally slow
- ❖ To solve Large Problem Sometime it needs to take the help of other techniques like Branch and bound
- ❖ Need Large amount of memory space for storing different state function in the stack for big problem
- ❖ Thrashing is one of the main problem of Backtracking
- ❖ The Basic Approach Detects the conflicts too late

# Application of Backtracking

- ❖ Optimization and tactical problems
- ❖ Constraints Satisfaction Problem
- ❖ Electrical Engineering
- ❖ Robotics
- ❖ Artificial Intelligence
- ❖ Genetic and bioinformatics Algorithm
- ❖ Materials Engineering
- ❖ Network Communication
- ❖ Solving puzzles and path

# Problems solved using Backtracking

- ❖ Sum of Subset
- ❖ N- Queens Problem
- ❖ Hamiltonian Cycle
- ❖ Sudoku Puzzle
- ❖ Maze Generation

# Sum Of Subsets Problem

- ❖ The Subset-Sum Problem is to find a subset  $s'$  of the given set  $S = (S_1, S_2, S_3, \dots, S_n)$  where the elements of the set  $S$  are  $n$  positive integers in such a manner that  $s' \subseteq S$  and sum of the elements of subset  $s'$  is equal to some positive integer ' $X$ '
- ❖ The Subset-Sum Problem can be solved by using the backtracking approach
- ❖ In this implicit tree is a binary tree, The root of the tree is selected in such a way that represents that no decision is yet taken on any input
- ❖ We assume that the elements of the given set are arranged in increasing order:  $S_1 \leq S_2 \leq S_3 \dots \leq S_n$
- ❖ The left child of the root node indicated that we have to include ' $S_1$ ' from the set ' $S$ ' and the right child of the root indicates that we have to exclude ' $S_1$ '
- ❖ Each node stores the total of the partial solution elements
- ❖ If at any stage the sum equals to ' $X$ ' then the search is successful and terminates
- ❖ The dead end in the tree appears only when either of the two inequalities exists:
  - The sum of  $s'$  is too large i.e.  $s' + S_i + 1 > X$
  - The sum of  $s'$  is too small i.e.  $s' + \sum_{j=i+1}^n S_j < X$

# Algorithm

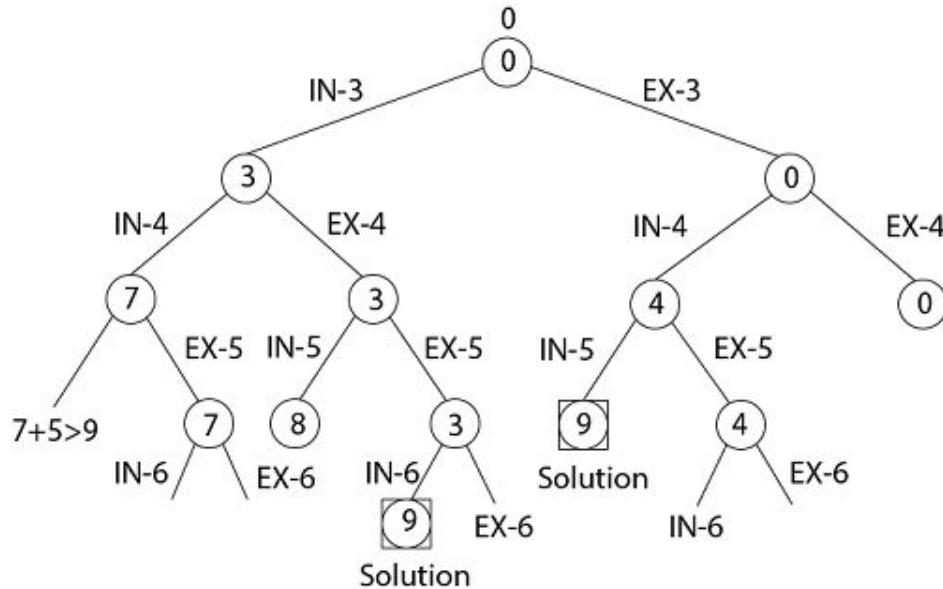
- ❖ First, organize the numbers in non decreasing order, Then generating a tree, we can find the solution by adding the weight of the nodes
- ❖ Note that, here more than necessary nodes can be generated
- ❖ The algorithm will search for the weighted amount of the nodes, If it goes beyond given Number N, then it stops generating nodes and move to other parent nodes for finding solution.

```
SumOfSubset(s,k,y){  
    X[k] = 1;  
    If(s+w[k] = m)  
        Write (x[1:n]);  
    Else if((s+w[k] + w[k+1]) <= m)  
        SumOfSubset(s+w[k], k+1, y-w[k]);  
    If ((s+ y-w[k]>=m) &&(s + w[k+1] <=m)) {  
        X[k] =0;  
        SumOfSubset(s,k+1,y-w[k]);  
    }  
}
```

Complexity :  $O(2^n)$

# Example

- ❖ Given a set  $S = (3, 4, 5, 6)$  and  $X = 9$ . Obtain the subset sum using Backtracking approach
- ❖ Solution:
- ❖ Initially  $S = (3, 4, 5, 6)$  and  $X = 9$ 
  - $S' = (\emptyset)$
- ❖ The implicit binary tree for the subset sum problem is shown as fig:
- ❖ The number inside a node is the sum of the partial solution elements at a particular level
- ❖ Thus, if our partial solution elements sum is equal to the positive integer 'X' then at that time search will terminate, or it continues if all the possible solution needs to be obtained



# N-Queens Problem

- ❖ N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal
- ❖ It can be seen that for n =1, the problem has a trivial solution, and no solution exists for n =2 and n =3. So first we will consider the 4 queens problem and then generate it to n - queens problem
- ❖ Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4
- ❖ Since, we have to place 4 queens such as  $q_1, q_2, q_3$  and  $q_4$  on the chessboard, such that no two queens attack each other
- ❖ In such a conditional each queen must be placed on a different row, i.e., we put queen "i" on row "i"

|   |   |   |   |   |
|---|---|---|---|---|
|   | 1 | 2 | 3 | 4 |
| 1 |   |   |   |   |
| 2 |   |   |   |   |
| 3 |   |   |   |   |
| 4 |   |   |   |   |

4x4 chessboard

# Procedure of n-Queens Problem

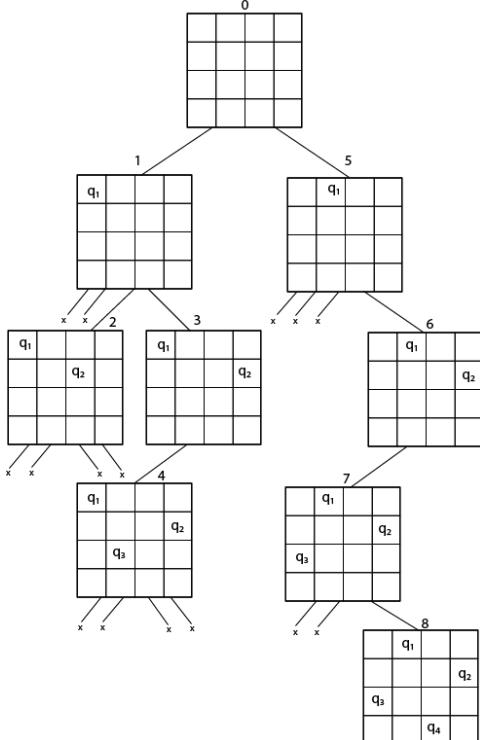
- ❖ Now, we place queen  $q_1$  in the very first acceptable position (1, 1)
- ❖ Next, we put queen  $q_2$  so that both these queens do not attack each other
- ❖ We find that if we place  $q_2$  in column 1 and 2, then the dead end is encountered
- ❖ Thus the first acceptable position for  $q_2$  in column 3, i.e. (2, 3) but then no position is left for placing queen ' $q_3$ ' safely
- ❖ So we backtrack one step and place the queen ' $q_2$ ' in (2, 4), the next best possible solution
- ❖ Then we obtain the position for placing ' $q_3$ ' which is (3, 2), But later this position also leads to a dead end, and no place is found where ' $q_4$ ' can be placed safely
- ❖ Then we have to backtrack till ' $q_1$ ' and place it to (1, 2) and then all other queens are placed safely by moving  $q_2$  to (2, 4),  $q_3$  to (3, 1) and  $q_4$  to (4, 3)
- ❖ That is, we get the solution (2, 4, 1, 3)
- ❖ This is one possible solution for the 4-queens problem
- ❖ For another possible solution, the whole method is repeated for all partial solutions
- ❖ The other solutions for 4 - queens problems is (3, 1, 4, 2) i.e.

|       |  |       |       |
|-------|--|-------|-------|
|       |  | $q_1$ |       |
|       |  |       | $q_2$ |
| $q_3$ |  |       |       |
|       |  |       | $q_4$ |

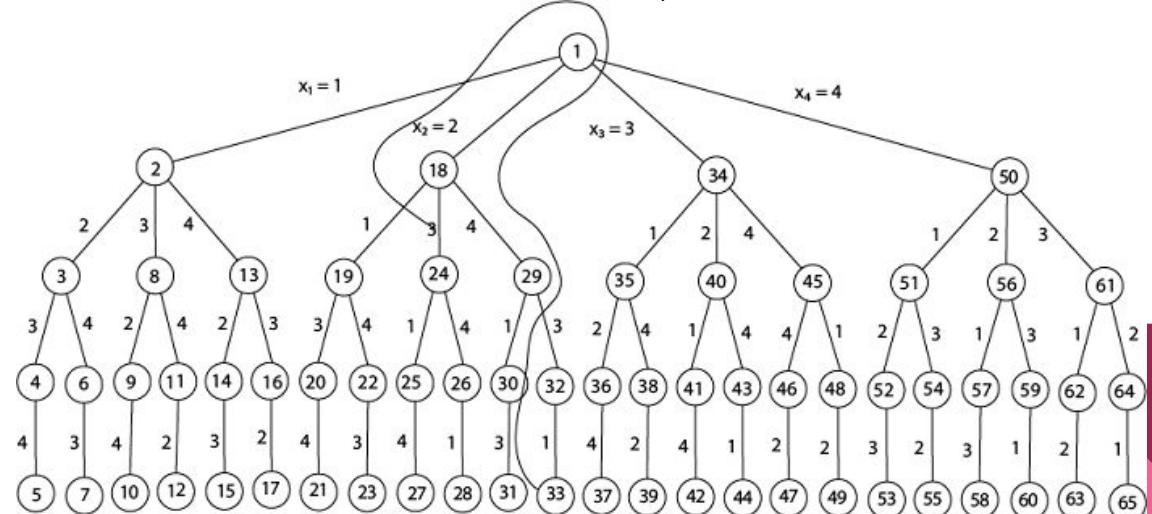
|   |   |       |       |       |
|---|---|-------|-------|-------|
|   | 1 | 2     | 3     | 4     |
| 1 |   |       | $q_1$ |       |
| 2 |   | $q_2$ |       |       |
| 3 |   |       |       | $q_3$ |
| 4 |   |       | $q_4$ |       |

# n-Queens Problem (contd...)

- The implicit tree for 4 - queen problem for a solution (2, 4, 1, 3) is as follows:



- Fig shows the complete state space for 4 - queens problem
- But we can use backtracking method to generate the necessary node and stop if the next node violates the rule, i.e., if two queens are attacking
- It can be seen that all the solutions to the 4 queens problem can be represented as 4 - tuples ( $x_1, x_2, x_3, x_4$ ) where  $x_i$  represents the column on which queen "q<sub>i</sub>" is placed.



# Algorithm

Complexity:  $2^n$

- ❖ Using place, we give a precise solution to then n- queens problem

Place (k, i)

{

For j  $\leftarrow$  1 to k - 1

do if ( $x[j] = i$ )

or ( $Abs(x[j]) - i$ ) = ( $Abs(j - k)$ )

then return false;

return true;

}

- ❖ Place (k, i) return true if a queen can be placed in the kth row and ith column otherwise return is false
- ❖  $x[]$  is a global array whose final  $k - 1$  values have been set.  $Abs(r)$  returns the absolute value of r

```
N - Queens (k, n)
{
    For i  $\leftarrow$  1 to n
        do if Place (k, i)
    then
    {
        x [k]  $\leftarrow$  i;
        if (k ==n) then
            write (x
[1.....n));
        else
            N - Queens (k +
1, n);
    }
}
```

# Hamiltonian Circuits

Given a graph  $G = (V, E)$  we have to find the Hamiltonian Circuit using Backtracking approach. We start our search from any arbitrary vertex say 'a.' This vertex 'a' becomes the root of our implicit tree. The first element of our partial solution is the first intermediate vertex of the Hamiltonian Cycle that is to be constructed. The next adjacent vertex is selected by alphabetical order. If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex 'a' then we say that **dead end** is reached. In this case, we backtrack one step, and again the search begins by selecting another vertex and backtrack the element from the partial; solution must be removed. The search using backtracking is successful if a Hamiltonian Cycle is obtained.

Complexity:  $O(2^n n^2)$

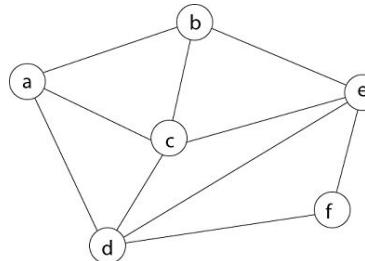
# Algorithm

```
Hamilton_Cycle(k){  
    Repeat{  
        NextVal(k);  
        If(x[k]==0)  
            then Return;  
        If (k ==n)  
            then Write (x[1:n]);  
            else Hamilton_Cycle(k+1);  
    } Until (false);  
}
```

```
NextVal(k){  
    Repeat{  
        X[k] = (x[k]+1) mod (n+1);  
        If(x[k]=0) then return;  
        If (G[x[k-1],x[k]] != 0 ) then{  
            For j = 1 to k-1 do  
                If(x[j]=x[k]) then  
                    Break;  
                If(j = k) then  
                    If ((k<n or k=n) && G[x[n],x[1]] != 0)  
                        then return;  
        }  
    } Until (false);  
}
```

# Example Hamiltonian Cycle

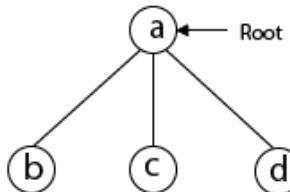
- ❖ **Example:** Consider a graph  $G = (V, E)$  shown in fig. we have to find a Hamiltonian circuit using Backtracking method



- ❖ **Solution:** Firstly, we start our search with vertex 'a' this vertex 'a' becomes the root of our implicit tree

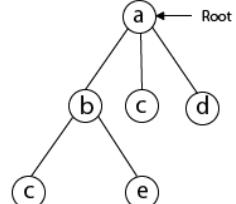


- ❖ Next, we choose vertex 'b' adjacent to 'a' as it comes first in lexicographical order (b, c, d)

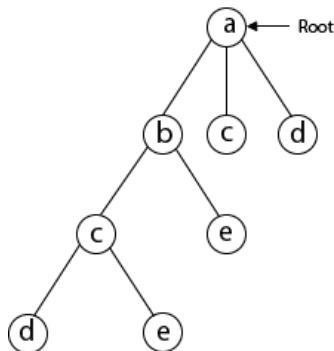


# Example Hamiltonian Cycle(contd...)

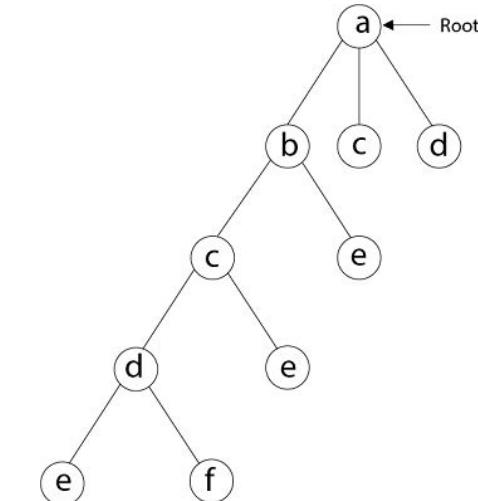
- ❖ Next, we select 'c' adjacent to 'b'



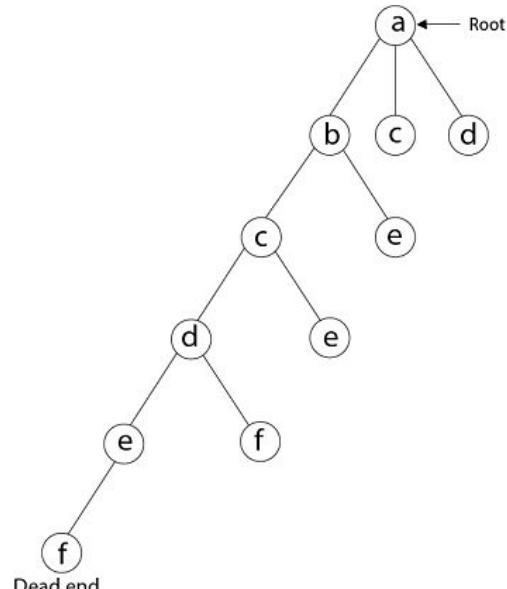
- ❖ Next, we select 'd' adjacent to 'c'



- ❖ Next, we select 'e' adjacent to 'd'

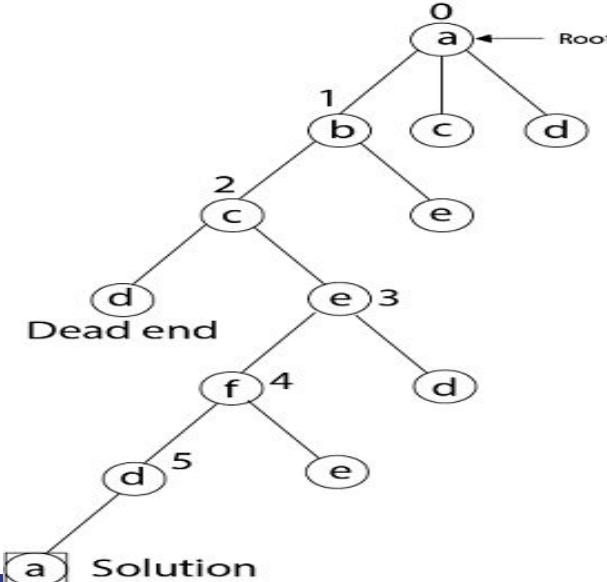
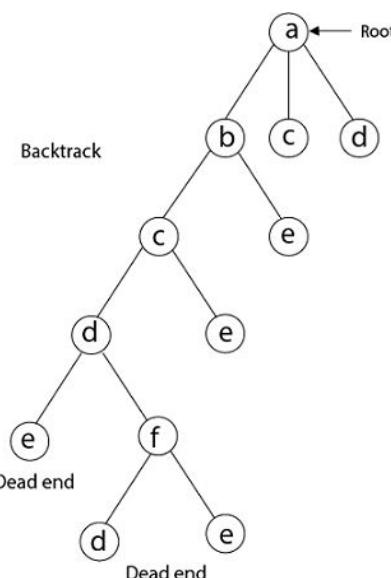
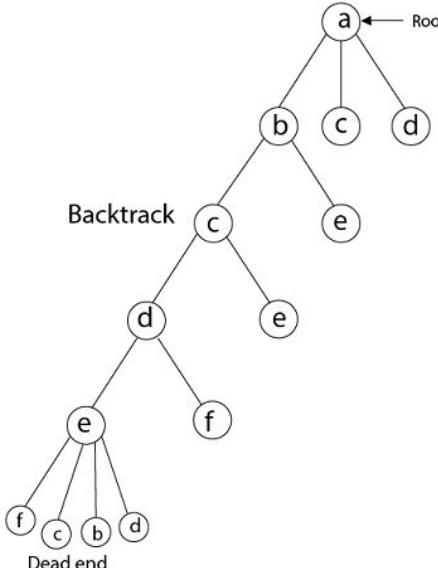


- ❖ Next, we select vertex 'f' adjacent to 'e.' The vertex adjacent to 'f' is d and e, but they have already visited. Thus, we get the dead end, and we backtrack one step and remove the vertex 'f' from partial solution



# Example Hamiltonian Cycle(contd...)

- ❖ From backtracking, the vertex adjacent to 'e' is b, c, d, and f from which vertex 'f' has already been checked, and b, c, d have already visited. So, again we backtrack one step. Now, the vertex adjacent to d are e, f from which e has already been checked, and adjacent of 'f' are d and e. If 'e' vertex, revisited them we get a dead state. So again we backtrack one step.
- ❖ Now, adjacent to c is 'e' and adjacent to 'e' is 'f' and adjacent to 'f' is 'd' and adjacent to 'd' is 'a.' Here, we get the Hamiltonian Cycle as all the vertex other than the start vertex 'a' is visited only once. (a - b - c - e - f - d - a)



# Graph Algorithms

- ❖ There are different types of algorithms based on graph:
  - Traversal Algorithms
    - Breadth First Search
    - Depth First Search
  - Path Finding Algorithms
    - Single Source Shortest Path
      - Dijkstra's Algorithm
    - All Pair Shortest Path
      - Warshall's Algorithm

# Traversing a Graph

\* Traversing a Graph: Visiting all the nodes in a graph systematically is called "Traversal". Graph is represented by its nodes & edges, so traversal of each node in the traversing in graph.

There are 2 graph traversal methods:

1. Breadth First Search (BFS)
2. Depth First Search (DFS)

In BFS, we use Queue for keeping nodes, which will be used for next processing &

In DFS, we use Stack for keeping nodes for next processing.

# Depth First Search(DFS)

- ❖ Traverses a tree data structure by exploring as far as possible down each branch before backtracking
- ❖ It's used on deeply hierarchical data and is a precursor to many other graph algorithms
- ❖ Depth-First Search is preferred when the tree is more balanced or the target is closer to an endpoint
- ❖ Depth-First Search is often used in gaming simulations where each choice or action leads to another, expanding into a tree-shaped graph of possibilities
- ❖ It will traverse the choice tree until it discovers an optimal solution path (e.g., win)

1.) Depth - First - Search (DFS): DFS is the traversal technique, which is used whenever it is possible to search the Graph deeper.

Given as input graph  $G = (V, E)$  & a source vertex  $S$ , from where the searching starts.

First, we travel or visit the starting node, then we travel through each node along a path, which begins at  $S$ . That is, we visit a neighbour vertex of  $S$  & again a neighbour of  $s$  & so on.

DFS works on both directed & undirected Graphs.

Algo:      DFS (vertex  $i$ ).

1. Initialize a stack  $S$  & vertex  $w$ .
2. Push  $i$  in the stack  $S$ .
3. Repeat step 4 to 5 while (stack  $S$  is not empty).
4.       $i = \text{pop}(S)$ .
5.      if ( $\text{! visited}[i]$ ) then
  - a.)       $\text{visited}[i] = 1$ .
  - b.)      for each  $w$  adjacent to  $i$ 
    - c.)      if ( $\text{! visited}[w]$ ) then  
push  $w$  in the stack  $S$ .
6. Return

If  $\text{visited}[w] = 0$ , then vertex is not visited.  
if  $= 1$ , then vertex is visited.

# Breadth First Search(BFS)

- ❖ Traverses a tree data structure by fanning out to explore the nearest neighbors and then their sub-level neighbors
- ❖ It's used to locate connections and is a precursor to [many other graph algorithms](#)
- ❖ BFS is preferred when the tree is less balanced or the target is closer to the starting point
- ❖ It can also be used to find the shortest path between nodes or avoid the recursive processes of depth-first search
- ❖ Breadth-First Search can be used to locate neighbor nodes in peer-to-peer networks like BitTorrent, GPS systems to pinpoint nearby locations and social network services to find people within a specific distance

2.) Breadth - First Search (BFS): BFS is the technique which uses Queue for traversing all the nodes of the graph.

In this first we take any node as the starting node then we take all the adjacent nodes to that starting node. Similarly, we take for all other adjacent nodes & so on

We maintain the status of visited node in one array so that no node can be traversed again.

BFS also works on both Directed & Undirected Graph

Algo:

BFS(vertex  $v$ )

1. Initialize a Queue  $\emptyset$ .
2. Set visited [ $v$ ] = 1.
3. Add vertex  $v$  to Queue  $\emptyset$ .
4. Repeat step 5 to 7 while ( $\emptyset$  is not empty).
5. Delete the element  $v$  from the Queue  $\emptyset$ .
6. for add vertices adjacent from  $v$ .
  7. if (!visited [ $w$ ]) then  
set visited [ $w$ ] = 1. & add vertex  $w$  to the Queue  $\emptyset$ .
8. Return

# Shortest Path Algorithms

- ❖ Calculates a shortest path forest (group) containing all shortest paths between the nodes in the graph
- ❖ Commonly used for understanding alternate routing when the shortest route is blocked or becomes sub-optimal
- ❖ All-Pairs Shortest Path is used to evaluate alternate routes for situations such as a freeway backup or network capacity
- ❖ It's also key in logical routing to offer multiple paths; for example, call routing alternatives

\* Shortest Path: A path from source vertex  $s$  to  $t$  is the shortest path from  $s$  to  $t$  if there is no path from  $s$  to  $t$  with lower weight.

In a weighted graph, or network, in order to find a shortest path between 2 nodes  $s$  &  $t$  is such that the sum of the weights of the arcs on the path is minimized.

To represent the network, we assume a weight function, such that  $\text{weight}(i, j)$  is the weight of the arc from  $i$  to  $j$ .

If there is no edge (or arc) from  $i$  to  $j$ ,  $\text{weight}(i, j)$  is set to an arbitrarily large value to indicate the infinite cost of going directly from  $i$  to  $j$ .

If there is a path from  $u$  to  $v$  vertex then the shortest paths are not necessarily unique.

There are 2 methods to find the shortest paths:

1. Dijkstra's Algorithm // Single-Source Shortest Paths
2. Warshall's Algorithm // All-Pair Shortest Paths

2) Warshall's Algorithm: The algorithm considers the "intermediate" vertices of a shortest path.

Let  $G$  be a directed graph with  $m$  nodes,  $v_1, v_2, \dots, v_m$ .

Suppose we want to find the path matrix  $D^{(k)}$  of the graph  $G$ .

Warshall gave an algorithm for this purpose i.e. much more efficient than calculating the powers of the adjacency matrix  $A$ .

Let  $d_{ij}^{(k)}$  be the weight of a shortest path from vertex  $i$  to vertex  $j$ , with all intermediate vertices in the set  $\{1, 2, \dots, k\}$ .

A recursive def " is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min(d_{ij}^{(k)}, d_{ik}^{(k)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1 \end{cases}$$

$w_{ij} = \begin{cases} 0 & \text{if } i=j \\ w(i,j) & \text{if } i \neq j \text{ & } (i,j) \in E \\ \infty & \text{if } i+j \notin E \end{cases}$

thus,  $d_{ij} = \begin{cases} 0 & \text{if } i=j \\ w(i,j) & \text{if } i \neq j \text{ & } (i,j) \in E \\ \infty & \text{otherwise} \end{cases}$

Algo: FLOYD-WARSHAL( $w$ )

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

# Branch and Bound

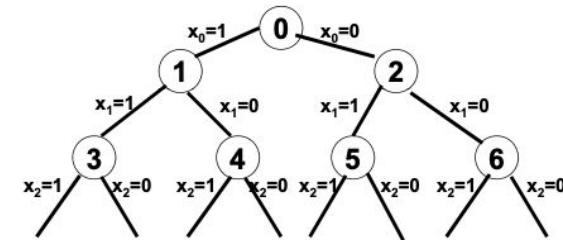
- ❖ Solves Optimisation Problem
- ❖ The problems should be of minimization type
  - If not then convert the maximization problem to minimization to solve the problem
- ❖ B & B is similar to Backtracking
- ❖ All the possible solutions at each level are considered at the starting itself
- ❖ BFS is followed
- ❖ Level by level completion
- ❖ State Space Tree is constructed
  - (using Queue) FIFO B & B
  - Or (using Stack) LIFO B & B
  - Or Least Cost B & B

# Branch & Bound (contd...)

- ❖ It refers to All State Space Search method in which all children of E-node are generated before any other live node can become E-node
- ❖ Technique for solving mixed (or pure) integer programming problems, based on tree search
  - Yes/no or 0/1 decision variables, designated  $x_i$
  - Problem may have continuous, usually linear variables
  - $O(2^n)$  complexity
    - Relies on upper and lower bounds to limit the number of combinations examined while looking for a solution
    - Dominance at a distance
      - Solutions in one part of tree can dominate other parts of tree
      - DP only has local dominance: dominance: states in same stage dominate dominate
    - Handles master/subproblem framework better than DP
    - Same problem size as dynamic programming, perhaps a little larger: data specific, a few hundred 0/1 variables
  - Branch-and-cut is a more sophisticated, related method
    - May solve problems with a few thousand 0/1 variables
    - Its code and math are complex
    - If you need branch-and-cut, use a commercial solver

# Generating Tree Nodes

- ❖ Every tree node is a problem state
  - It is generally associated with one 0-1 variable, sometimes a group
  - Other 0-1 variables are implicitly defined by the path from the root to this node
    - We sometimes store all  $\{x\}$  at each node rather than tracing back
  - Still other 0-1 variables associated with nodes below the current node in the tree have unknown values, since the path to those nodes has not been built yet
- ❖ Tree nodes are generated dynamically as the program progresses
  - Live node is node that has been generated but not all of its children have been generated yet
  - E-node is a live node currently being explored. Its children are being generated
  - Dead node is a node either:
    - Not to be explored further or
    - All of whose children have already been explored



# Managing Live Tree Nodes

- ❖ Branch and bound keeps a list of live nodes
- ❖ Four strategies are used to manage the list:
  - Depth first search: As soon as child of current E-node is generated, the child becomes the new E-node
    - Parent becomes E-node only after child's subtree is explored
    - Horowitz and Sahni call this 'backtracking'
  - In the other 3 strategies, the E-node remains the E-node until it is dead. Its children are managed by:
    - Breadth first search: Children are put in queue
    - D-search Child t t k : Children are put on stack
    - Least cost search: Children are put on heap
  - We use bounding functions (upper and lower bounds) to kill live nodes without generating all their children
    - Somewhat analogous to pruning in dynamic programming

# Knapsack Problem using B & B

- ❖ The  $x_i$  are 0-1 variables, like the DP and unlike the greedy version
- ❖ To solve it using B & B, we need it to be treated as minimisation problem instead of maximisation problem
- ❖ That can be done by changing the formula to -  $\sum p_i x_i$
- ❖ So now instead of **maximising**  $\sum p_i x_i$ , we try to **minimise** -  $\sum p_i x_i$

$$\max \sum_{0 \leq i < n} p_i x_i$$

s.t.

$$\sum_{0 \leq i < n} w_i x_i \leq M$$

$$x_i = 0, 1$$

$$p_i \geq 0, w_i \geq 0, 0 \leq i < n$$

# 0/1 Knapsack Example

Q: Consider the knapsack instance  
 $n=4$ ,  $(p_1, p_2, p_3, p_4) = (10, 10, 12, 18)$ ,  $(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$  and capacity of the knapsack  $W=15$ . Apply branch & bound technique to find out the solution of the problem.

$\rightarrow M$

| Item | v  | w | v/w |
|------|----|---|-----|
| 1    | 10 | 2 | 5   |
| 2    | 10 | 4 | 2.5 |
| 3    | 12 | 6 | 2   |
| 4    | 18 | 9 | 2   |

my companion

upper bound equals  
 $v + (W-w)(v_i+1/w_{i+1})$

E-node

$$\begin{cases} i=0 \\ v=0, w=0 \\ UB = 75 \end{cases}$$

$$up.b = v + (15-0)\left(\frac{10}{12}\right)$$

$$\begin{array}{l} \text{with } 1 \\ \text{without } 1 \end{array} = 75$$

live

$$\begin{cases} v=10, w=2 \\ UB=42.5 \end{cases}$$

$$\begin{cases} v=0, w=0 \\ UB=37.5 \end{cases}$$

with no 2 without 2

$$\begin{array}{l} \text{with } 2 \\ \text{without } 2 \end{array} = \begin{cases} v=20, w=6 \\ UB=38 \end{cases}$$

$$\begin{cases} v=10, w=2 \\ UB=36 \end{cases}$$

$$\begin{array}{l} \text{with } 3 \\ \text{without } 3 \end{array} = \begin{array}{l} \begin{array}{l} v=20+(15-6)\left(\frac{12}{12}\right) \\ = 20+9\times 2 \\ = 20+18 \\ = 38 \end{array} \\ \begin{array}{l} v=20, w=6 \\ UB=38 \end{array} \end{array}$$

$$\begin{array}{l} \text{with } 4 \\ \text{without } 4 \end{array} = \begin{cases} v=32, w=12 \\ UB=39 \end{cases}$$

$$\begin{cases} v=20, w=6 \\ UB=38 \end{cases}$$

with no 4

without 4

$$\begin{cases} v=59, w=21 \\ UB=39 \end{cases}$$

$$\begin{cases} v=32, w=12 \\ UB=32 \end{cases}$$

$$\begin{cases} v=20, w=6 \\ UB=38 \end{cases}$$

$$\begin{cases} v=20, w=6 \\ UB=20 \end{cases}$$

$v < 59$   
 $w > 21$   
 (Not feasible)

$$\begin{array}{l} \text{with } 4 \\ \text{without } 4 \end{array} = \begin{cases} v=20, w=6 \\ UB=38 \end{cases}$$

$$\begin{cases} v=20, w=6 \\ UB=20 \end{cases}$$

Consider this

[ 1 1 0 1 ] ✓  
 (1110) ✓

# Algorithm

```
procedure UBOUND (  $p, w, k, M$  )
    // $p, w, k$  and  $M$  have the same meaning as in Algorithm 7.11//
    // $W(i)$  and  $P(i)$  are respectively the weight and profit of the  $i$ th object//
    global  $W(1:n), P(1:n); \text{integer } i, k, n$ 
     $b \leftarrow p; c \leftarrow w$ 
    for  $i \leftarrow k + 1$  to  $n$  do
        if  $c + W(i) \leq M$  then  $c \leftarrow c + W(i); b \leftarrow b - P(i)$  endif
    repeat
    return ( $b$ )
end UBOUND
```

Algorithm 8.5 Function  $u(\cdot)$  for knapsack problem

# Example of 0/1 Knapsack using LCB&B

\* 0/1 Knapsack Problem using B&B.

|        | 1  | 2  | 3  | 4  |
|--------|----|----|----|----|
| Profit | 10 | 10 | 12 | 18 |
| Weight | 2  | 4  | 6  | 9  |

$$m = 15$$

$$n = 4$$

This is the maximization problem

But B&B solves minimization prob.

We will solve by LCB&B.

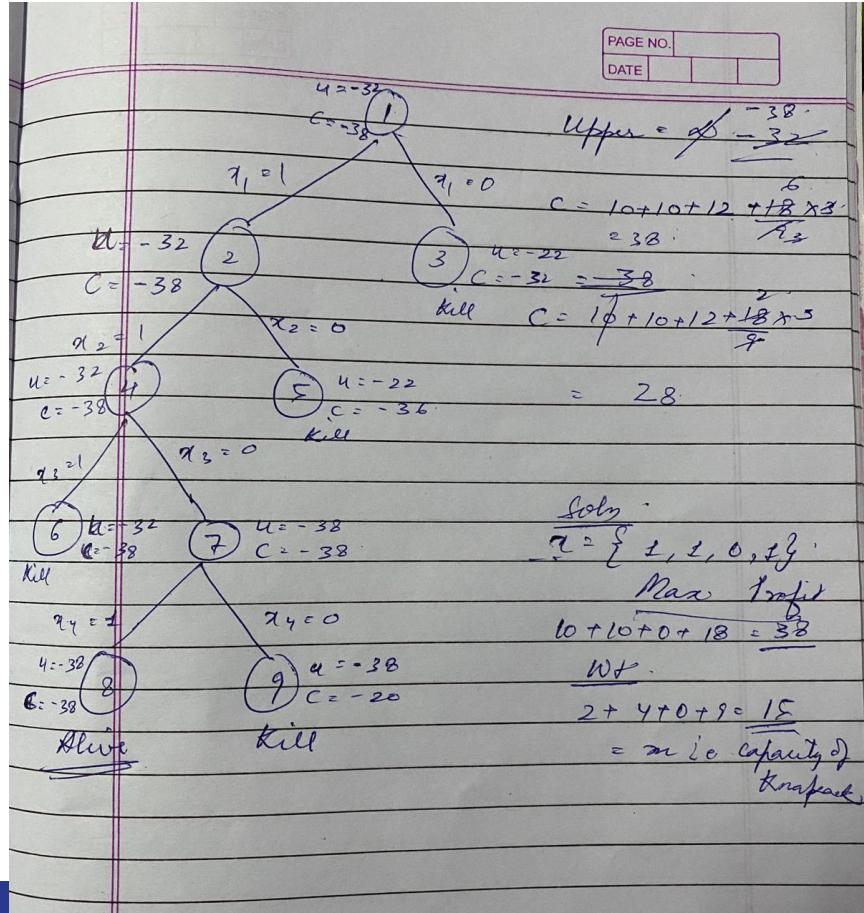
$$(\text{Upper bound}) \quad U = - \sum_{i=1}^n p_i x_i \leq m.$$

$$C = - \sum_{i=1}^n p_i x_i \quad (\text{with fraction}).$$

$$S = \{x_1, x_3\}$$

$$S = \{1, 0, 1, 0\}$$

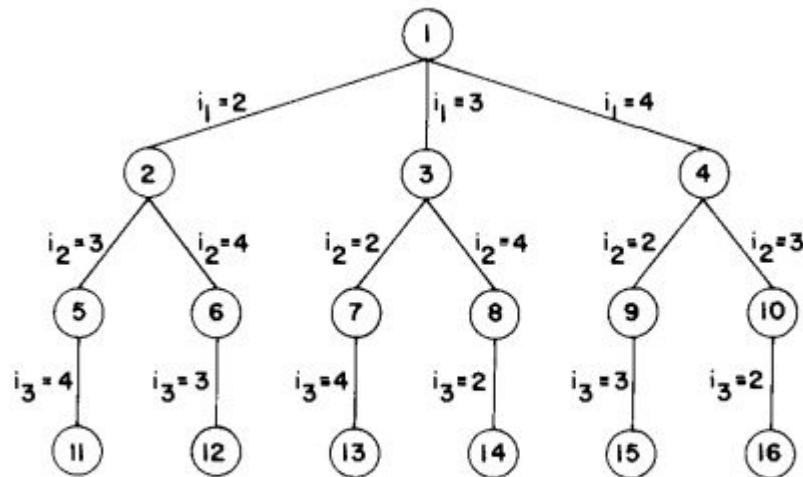
Converting to minimization by -ve sign



# Travelling Salesman Problem(TSP) using B&B

Let  $G = (V, E)$  be a directed graph defining an instance of the traveling salesperson problem. Let  $C_{ij}$  be the cost of edge  $(i, j)$ ,  $C_{ij} = \infty$  if  $(i, j) \sim E$  and let  $|V| = n$ . Without loss of generality, we may assume that every tour starts and ends at vertex 1. So, the solution space  $S$  is given by  $S = \{l, 7r \mid 7r \text{ is a permutation of } (2, 3, \dots, n)\}$ .  $|S| = (n - 1)!$ . The size of  $S$  may be reduced by restricting  $S$  so that  $(1, i_1, i_2, \dots, i_{-1}, 1) \in S$  iff  $(i_h, i_{j+1}) \in E, 0 \leq j \leq n - 1, i_0 = i_{-1} = 1$ .  $S$  may be organized into a state space tree similar to that for the n-queens problem. The worst case complexity of this algorithm is  $O(n^2 \cdot 2^n)$

# TSP using B&B(contd...)



State space tree for the traveling salesperson problem with  $n = 4$   
and  $i_0 = i_4 = 1$

# TSP using B&B(contd...)

In order to use LC-branch-and-bound to search the traveling salesperson state space tree, we need to define a cost function  $c(\cdot)$  and two other functions  $c(\cdot)$  and  $u(\cdot)$  such that  $c(R) \sim c(R) \sim u(R)$  for all nodes  $R$ .  $c(\cdot)$  is such that the solution node with least  $c(\cdot)$  corresponds to a shortest tour in  $G$ .

One choice for  $c(\cdot)$  is:

$$c(A) = \begin{cases} \text{length of tour defined by the path from the root to } A \text{ if } A \text{ is a leaf} \\ \text{cost of a minimum cost leaf in the subtree } A \text{ if } A \text{ is not a leaf} \end{cases}$$

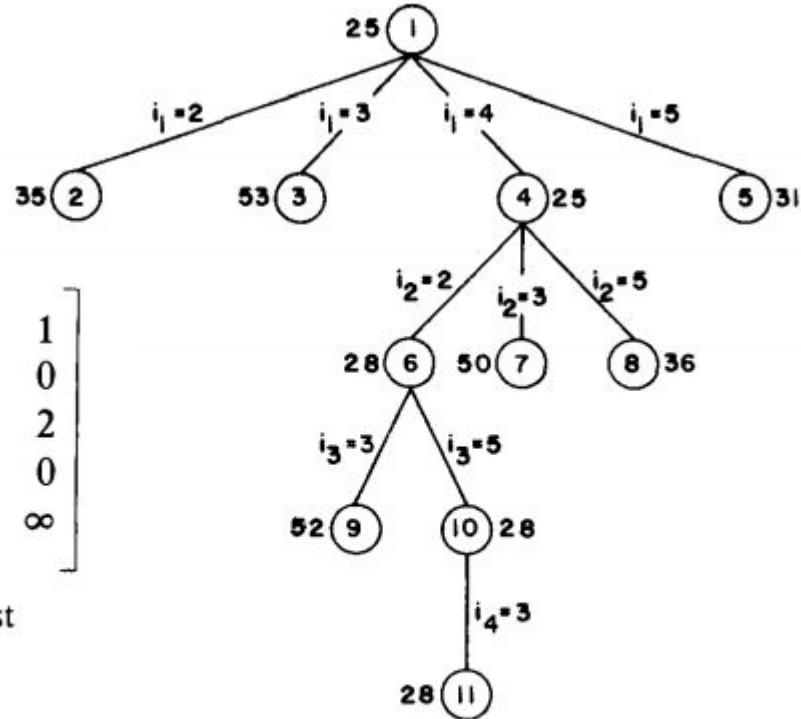
# TSP using B&B(contd...)

|          |          |          |          |          |
|----------|----------|----------|----------|----------|
| $\infty$ | 20       | 30       | 10       | 11       |
| 15       | $\infty$ | 16       | 4        | 2        |
| 3        | 5        | $\infty$ | 2        | 4        |
| 19       | 6        | 18       | $\infty$ | 3        |
| 16       | 4        | 7        | 16       | $\infty$ |

(a) Cost Matrix

|          |          |          |          |          |
|----------|----------|----------|----------|----------|
| $\infty$ | 10       | 17       | 0        | 1        |
| 12       | $\infty$ | 11       | 2        | 0        |
| 0        | 3        | $\infty$ | 0        | 2        |
| 15       | 3        | 12       | $\infty$ | 0        |
| 11       | 0        | 0        | 12       | $\infty$ |

(b) Reduced Cost  
Matrix  
 $L = 25$



Numbers outside the nodes are  $c$  values

Figure 8.13 State space tree generated by procedure LCBB.

# Steps for finding solution of TSP using B&B

1. Obtain the reduced matrix by subtracting the least no. of each row from every element of that row
2. Further reduce the matrix by subtracting the least no. of each column from every element of that column (**Compute the minimum cost of the reduced matrix = sum of all the least values obtained for reducing the matrix rowwise and columnwise and say it as L**)
3. Now traverse each node from 1st node and find the possible way i.e 1-->2, 1-->3, 1-->4, 1-->5 .
  - a. For 1-->2, make each value in row 1 and column 2 as infinity.
  - b. And further change value (2,1) as infinity.
  - c. Then compute the reduced matrix (rowwise and columnwise reduction)
  - d. **Compute the minimum cost = L + sum of least values rowwise and columnwise + cost(1,2)**
4. Repeat for each node 3,4,5 the steps a-d of step 3.
5. Now compute the values from node with minimum cost node(x) to every other node except for node 1 i.e x-->, x-->, x--> (by repeating steps 3 & 4)
  - a. Choose the base matrix at each level from the previous level matrix.
6. Draw the state space tree with the Minimum cost of each node labelled and further move forward by choosing that node whose cost is minimum.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix} \quad \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix} \quad \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

a) path 1,2; node 2

b) path 1,3; node 3

c) path 1,4;node 4

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix} \quad \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & 0 & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix} \quad \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

d) path 1,5; node 5

e) path 1,4,2; node 6

f) path 1,4,3; node 7

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix} \quad \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix} \quad \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

g) path 1,4,5; node 8

h) path 1,4,2,3; node 9

i) path 1,4,2,5; node 10

Figure 8.14 Reduced cost matrices corresponding to nodes in Figure 8.13

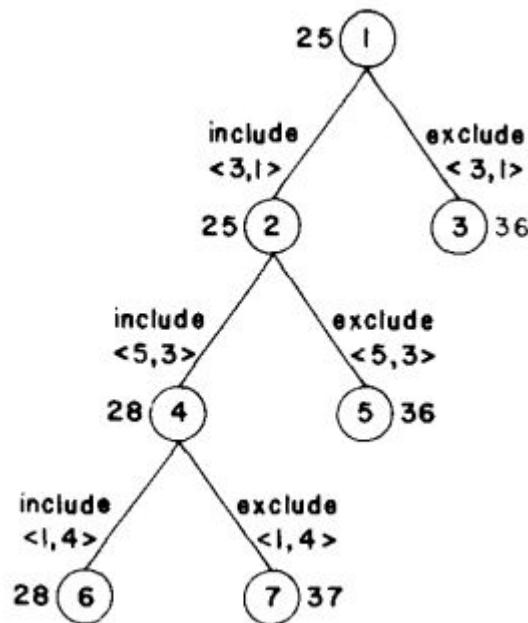
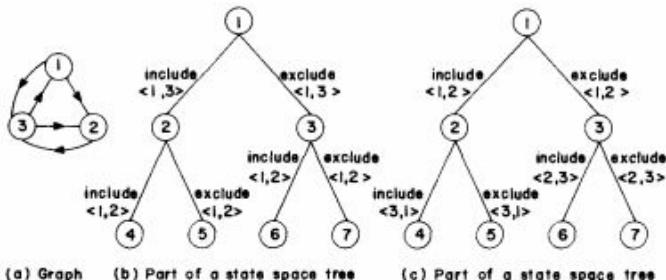


Figure 8.15 An example

# References

- ❖ <https://www.geeksforgeeks.org/backtracking-introduction/>
- ❖ <https://www.javatpoint.com/hamiltonian-circuit-problems>
- ❖ [https://www.youtube.com/watch?v=xFv\\_HI4B83A](https://www.youtube.com/watch?v=xFv_HI4B83A)
- ❖ <https://www.tutorialspoint.com/Hamiltonian-Cycle>
- ❖ <https://www.javatpoint.com/n-queens-problems>
- ❖ <https://neo4j.com/blog/graph-algorithms-neo4j-15-different-graph-algorithms-and-what-they-do/>
- ❖ [https://ocw.mit.edu/courses/civil-and-environmental-engineering/1-204-computer-algorithms-in-systems-engineering-spring-2010/lecture-notes/MIT1\\_204S10\\_lec16.pdf](https://ocw.mit.edu/courses/civil-and-environmental-engineering/1-204-computer-algorithms-in-systems-engineering-spring-2010/lecture-notes/MIT1_204S10_lec16.pdf)
- ❖

# Introduction to Branch and Bound

- ❖ The branch-and-bound design strategy is very similar to backtracking in that a state space tree is used to solve a problem
- ❖ The differences are that the branch-and-bound method
  - does not limit us to any particular way of traversing the tree, and
  - is used only for optimization problems
- ❖ A branch-and-bound algorithm computes a number (bound) at a node to determine whether the node is promising
- ❖ The number is a bound on the value of the solution that could be obtained by expanding beyond the node
- ❖ If that bound is no better than the value of the best solution found so far, the node is non-promising  
Otherwise, it is promising
- ❖ The backtracking algorithm for the 0-1 Knapsack problem is actually a branch-and-bound algorithm
- ❖ A backtracking algorithm, however, does not exploit the real advantage of using branch-and-bound
- ❖ Besides using the bound to determine whether a node is promising, we can compare the bounds of promising nodes and visit the children of the one with the best bound
- ❖ This approach is called best-first search with branch-and-bound pruning
- ❖ The implementation of this approach is a modification of the breadth-first search with branch-and-bound pruning

# B & B: An enhancement of backtracking

- ❖ Applicable to optimization problems
- ❖ Uses a lower bound for the value of the objective function for each node (partial solution) so as to:
  - guide the search through state-space
  - rule out certain branches as “unpromising”

# Design and Analysis of Algorithms

## (18CSC204J)

### Unit - 5

By:  
Aarti Sharma  
Asst Professor  
CSE Dept  
SRMIST Delhi-NCR  
Campus

# String Matching Algorithms

- ❖ String matching algorithms have greatly influenced computer science and play an essential role in various real-world problems
- ❖ It helps in performing time-efficient tasks in multiple domains
- ❖ These algorithms are useful in the case of searching a string within another string
- ❖ String matching is also used in the Database schema, Network systems
- ❖ The Pattern Searching algorithms are sometimes also referred to as String Searching Algorithms and are considered as a part of the String algorithms
- ❖ These algorithms are useful in the case of searching a string within another string

Text : A A B A A C A A D A A B A A B A

Pattern : A A B A



Pattern Found at 0, 9 and 12

# String Matching Algorithms(contd...)

- ❖ Given a text array,  $T [1.....n]$ , of  $n$  character and a pattern array,  $P [1.....m]$ , of  $m$  characters
- ❖ The problems are to find an integer  $s$ , called **valid shift** where  $0 \leq s < n-m$  and  $T [s+1.....s+m] = P [1.....m]$
- ❖ In other words, to find even if  $P$  in  $T$ , i.e., where  $P$  is a substring of  $T$
- ❖ The item of  $P$  and  $T$  are character drawn from some finite alphabet such as {0, 1} or {A, B .....Z, a, b..... Z}
- ❖ Given a string  $T [1.....n]$ , the **substrings** are represented as  $T [i.....j]$  for some  $0 \leq i \leq j \leq n-1$ , the string formed by the characters in  $T$  from index  $i$  to index  $j$ , inclusive
- ❖ This process that a string is a substring of itself (take  $i = 0$  and  $j = m$ )
- ❖ The **proper substring** of string  $T [1.....n]$  is  $T [1.....j]$  for some  $0 < i \leq j \leq n-1$ . That is, we must have either  $i > 0$  or  $j < m-1$
- ❖ Using these descriptions, we can say given any string  $T [1.....n]$ , the substrings are
  - $T [i.....j] = T [i] T [i + 1] T [i+2].....T [j]$  **for** some  $0 \leq i \leq j \leq n-1$
  - **Note:** If  $i > j$ , then  $T [i.....j]$  is equal to the empty string or null, which has length zero

# Types of String Matching Algorithms

- ❖ String Matching Algorithms can broadly be classified into two types of algorithms –
  - Exact String Matching Algorithms
  - Approximate String Matching Algorithms
- ❖ **Exact string matching algorithms** is to find one, several, or all occurrences of a defined string (pattern) in a large string (text or sequences) such that each matching is perfect
- ❖ All alphabets of patterns must be matched to corresponding matched subsequence
- ❖ These are further classified into four categories:
  - **Algorithms based on character comparison**
    - **Naive Algorithm:**
      - It slides the pattern over text one by one and check for a match.
      - If a match is found, then slides by 1 again to check for subsequent matches
    - **KMP (Knuth Morris Pratt) Algorithm:**
      - The idea is whenever a mismatch is detected, we already know some of the characters in the text of the next window
      - So, we take advantage of this information to avoid matching the characters that we know will anyway match.

# Types of String Matching Algorithms(contd...)

- Deterministic Finite Automaton (DFA) method:
  - Automaton Matcher Algorithm:
    - It starts from the first state of the automata and the first character of the text
    - At every step, it considers next character of text, and look for the next state in the built finite automata and move to a new state
- Algorithms based on Bit (parallelism method):
  - Aho-Corasick Algorithm:
    - It finds all words in  $O(n + m + z)$  time where n is the length of text and m be the total number characters in all words and z is total number of occurrences of words in text
    - This algorithm forms the basis of the original Unix command fgrep
- Hashing-string matching algorithms:
  - Rabin Karp Algorithm:
    - It matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters

# Types of String Matching Algorithms(contd...)

- ❖ **Approximate String Matching Algorithms** (also known as **Fuzzy String Searching**) searches for substrings of the input string
- ❖ More specifically, the approximate string matching approach is stated as follows:
  - Suppose that we are given two strings, text  $T[1\dots n]$  and pattern  $P[1\dots m]$
  - The task is to find all the occurrences of patterns in the text whose edit distance to the pattern is at most  $k$
  - Some well known edit distances are – Levenshtein edit distance and Hamming edit distance
  - These techniques are used when the quality of the text is low, there are spelling errors in the pattern or text, finding DNA subsequences after mutation, heterogeneous databases
  - Some approximate string matching algorithms are:
    - **Naive Approach:**
      - It slides the pattern over text one by one and check for approximate matches
      - If they are found, then slides by 1 again to check for subsequent approximate matches
    - Sellers Algorithm (Dynamic Programming)
    - Shift or Algorithm (Bitmap Algorithm)

# Applications of String Matching Algorithms

- ❖ Applications of String Matching Algorithms:
  - Plagiarism Detection
  - Bioinformatics and DNA sequencing
  - Digital Forensics
  - Spelling Checker
  - Spam Filters
  - Search engines or content search in large databases
  - Intrusion detection system

# Naive String Matching Algorithm

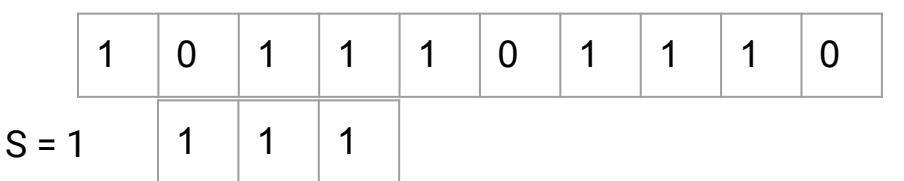
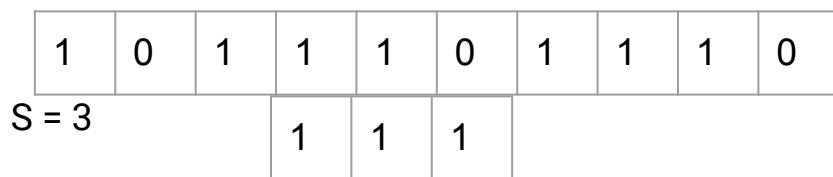
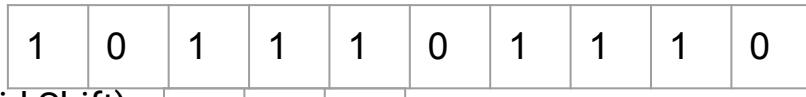
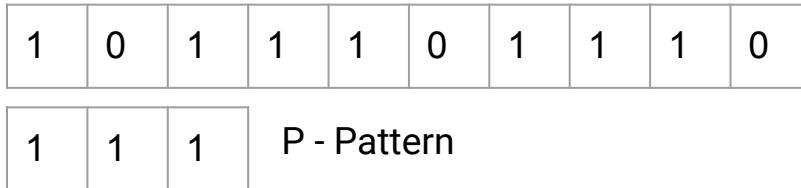
- ❖ The naïve approach tests all the possible placement of Pattern P [1.....m] relative to text T [1.....n]
- ❖ We try shift  $s = 0, 1.....n-m$ , successively and for each shift  $s$ . Compare  $T [s+1.....s+m]$  to  $P [1.....m]$
- ❖ The naïve algorithm finds all valid shifts using a loop that checks the condition  $P [1.....m] = T [s+1.....s+m]$  for each of the  $n - m + 1$  possible value of  $s$
- ❖ **NAIVE-STRING-MATCHER (T, P)**
  - $n \leftarrow \text{length } [T]$
  - $m \leftarrow \text{length } [P]$
  - for  $s \leftarrow 0$  to  $n - m$ 
    - do if  $P [1.....m] = T [s + 1....s + m]$ 
      - then print "Pattern occurs with shift"  $s$
- ❖ **Analysis:** This for loop from 3 to 5 executes for  $n-m + 1$ (we need at least  $m$  characters at the end) times and in iteration we are doing  $m$  comparisons, So the total complexity is  $O (n-m+1).m$

# Example of Naive String Matching

- Suppose  $T = 1011101110$ ,  $P = 111$  Find all the Valid Shift

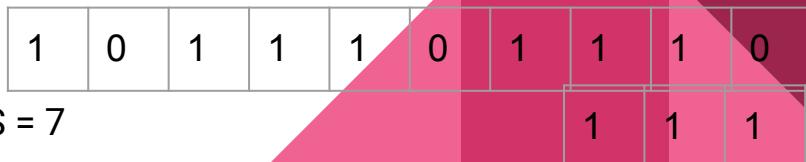
- Solution:  $T$  - Text

$S = 2$  (Valid Shift)



$S = 4$

$S = 5$  not valid.  $S = 6$  (Valid Shift)



$S = 7$

# Rabin-Karp Algorithm

- ❖ The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared
- ❖ If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence
- ❖ If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence
- ❖ In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match
- ❖ Complexity:
- ❖ The running time of **RABIN-KARP-MATCHER** in the worst case scenario  $\Theta((n-m+1)m)$  but it has a good average case running time
- ❖ If the expected number of strong shifts is small  **$O(1)$**  and prime  $q$  is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time  **$O(n+m)$**  plus the time to require to process spurious hits

## RABIN-KARP-MATCHER ( $T, P, d, q$ )

1.  $n \leftarrow \text{length}[T]$
2.  $m \leftarrow \text{length}[P]$
3.  $h \leftarrow d^{m-1} \bmod q$
4.  $p \leftarrow 0$
5.  $t_0 \leftarrow 0$
6. for  $i \leftarrow 1$  to  $m$  // Pre-Processing
7. do  $p \leftarrow (dp + P[i]) \bmod q$
8.  $t_0 \leftarrow (dt_0 + T[i]) \bmod q$
9. for  $s \leftarrow 0$  to  $n-m$  // Matching
10. do if  $p = t_s$
11. then if  $P[1.....m] = T[s+1.....s+m]$
12. then "Pattern occurs with shift"  $s$
13. If  $s < n-m$
14. then  $t_{s+1} \leftarrow (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$

# Example of Rabin-Karp Algorithm

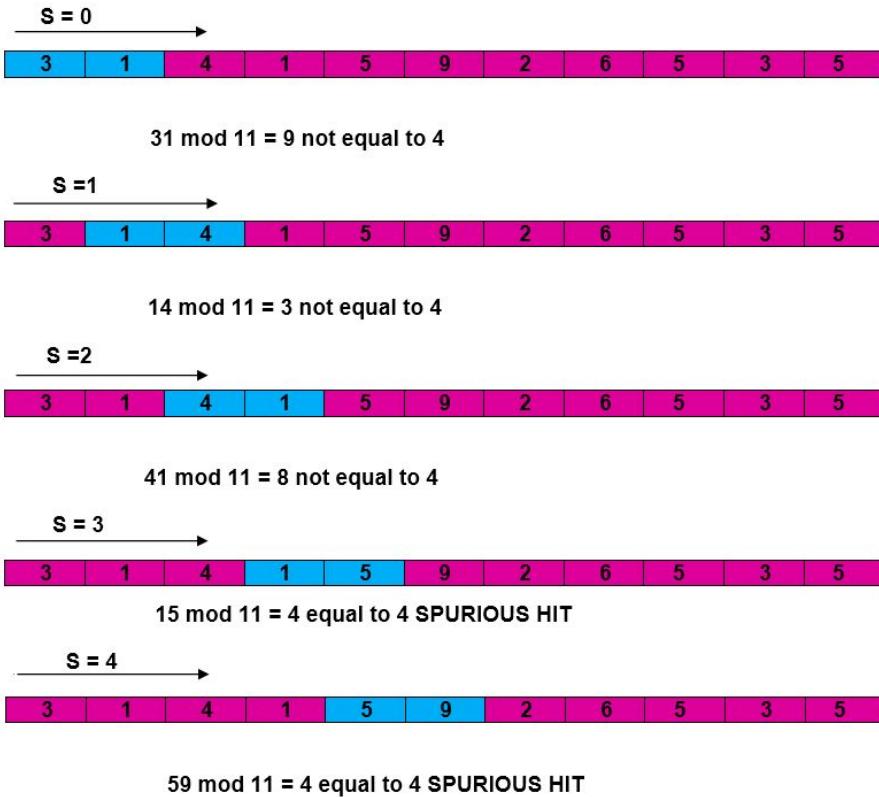
- ❖ For string matching, working module q = 11, how many spurious hits does the Rabin-Karp matcher encounters in Text T = 31415926535.....
  - T = **31415926535**, P = **26**
  - Here T.Length = **11** so Q = **11** And P mod Q = **26 mod 11 = 4**
  - Now find the exact match of P mod Q
- ❖ Solution:

T =

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|

P =

|   |   |
|---|---|
| 2 | 6 |
|---|---|



**S = 5**



$92 \bmod 11 = 4$  equal to 4 SPURIOUS HIT

**S = 6**



$26 \bmod 11 = 4$  EXACT MATCH

**S = 7**



**S = 9**



$35 \bmod 11 = 2$  not equal to 4

The Pattern occurs with shift 6.

**S = 7**



$65 \bmod 11 = 10$  not equal to 4

**S = 8**



$53 \bmod 11 = 9$  not equal to 4

# Introduction to Randomized Algorithms

- ❖ We call an algorithm randomized if its behavior is determined not only by its input but also by values produced by a random-number generator
- ❖ We shall assume that we have at our disposal a random-number generator **RANDOM**
- ❖ A call to **RANDOM(a, b)** returns an integer between a and b, inclusive, with each such integer being equally likely
  - **RANDOM(0, 1)** produces 0 with probability  $\frac{1}{2}$ , and it produces 1 with probability  $\frac{1}{2}$
  - A call to **RANDOM(3, 7)** returns either 3, 4, 5, 6 or 7, each with probability  $\frac{1}{5}$
- ❖ Each integer returned by **RANDOM** is independent of the integers returned on previous calls
- ❖ You may imagine **RANDOM** as rolling a  $(b - a + 1)$ -sided die to obtain its output
- ❖ Thus we can say, An algorithm that uses random numbers to decide what to do next anywhere in its logic is called a Randomized Algorithm

# The Hiring Problem

- ❖ Suppose you are using an employment agency to hire an office assistant
    - The agency sends you one candidate per day: interview and decide
    - Cost to interview is  $c_i$  per candidate (fee to agency)
    - Cost to hire is  $c_h$  per candidate (includes firing prior assistant and fee to agency)
    - $c_h > c_i$
    - You always hire the best candidate seen so far
  - ❖ **Algorithm:**

```
Hire-Assistant(n)
1 best = 0          // fictional least qualified candidate
2 for i = 1 to n
3   interview candidate i // paying cost  $c_i$ 
4   if candidate i is better than candidate best
5     best = i
6   hire candidate i // paying cost  $c_h$ 
```
  - ❖ What is the cost of this strategy?
    - If we interview  $n$  candidates and hire  $m$  of them, cost is  $\Theta(c_i n + c_h m)$
    - We interview all  $n$  and  $c_i$  is small, so we focus on  $c_h m$
    - $c_h m$  varies with each run and depends on interview order
    - This is a common paradigm: finding the maximum or minimum in a sequence by examining each element, and changing the winner  $m$  times
- ## ❖ Best Case
- If each candidate is worse than all who came before, we hire one candidate:
    - $\Theta(c_i n + c_h) = \Theta(c_i n)$
- ## ❖ Worst Case
- If each candidate is better than all who came before, we hire all  $n$  ( $m = n$ ):
    - $\Theta(c_i n + c_h n) = \Theta(c_h n)$  since  $c_h > c_i$

# Probabilistic Analysis with Indicator Random Variables

- ❖ Here we introduce a technique for computing the expected value of a random variable, even when there is dependence between variables
- ❖ Two informal definitions will get us started:
  - A **random variable** (e.g.,  $X$ ) is a variable that takes on any of a range of values according to a probability distribution
  - The **expected value** of a random variable (e.g.,  $E[X]$ ) is the average value we would observe if we sampled the random variable repeatedly
- ❖ **Indicator Random Variables**
  - Given sample space  $S$  and event  $A$  in  $S$ , define the **indicator random variable**
- ❖ We will see that indicator random variables simplify analysis by letting us work with the probability of the values of a random variable separately
- ❖ **Lemma 1**
  - For an event  $A$ , let  $X_A = I\{A\}$ . Then the expected value  $E[X_A] = \Pr\{A\}$  (the probability of event  $A$ )
  - *Proof:* Let  $\neg A$  be the complement of  $A$ 
    - Then  $E[X_A] = E[I\{A\}]$  (by definition)
    - $= 1 * \Pr\{A\} + 0 * \Pr\{\neg A\}$  (definition of expected value)
    - $= \Pr\{A\}$

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs ,} \\ 0 & \text{if } A \text{ does not occur} \end{cases}$$

# Randomised Version of Hiring Problem

- ❖ Assume that the candidates arrive in random order (we can enforce that by randomization if needed)
- ❖ Let  $X$  be the random variable for the number of times we hire a new office assistant
- ❖ Define indicator random variables  $X_1, X_2, \dots, X_n$  where  $X_i = I\{\text{candidate } i \text{ is hired}\}$
- ❖ We will rely on these properties:
  - $X = X_1 + X_2 + \dots + X_n$  (*The total number of hires is the sum of whether we did each individual hire (1) or not (0)*)
  - Lemma 1 implies that  $E[X_i] = \Pr\{\text{candidate } i \text{ is hired}\}$
- ❖ We need to compute  $\Pr\{\text{candidate } i \text{ is hired}\}$ :
  - Candidate  $i$  is hired iff candidate  $i$  is better than candidates 1, 2, ...,  $i-1$
  - Assumption of random order of arrival means any of the first  $i$  candidates are equally likely to be the best one so far
  - Thus,  $\Pr\{\text{candidate } i \text{ is the best so far}\} = 1/i$  (*Intuitively, as you add more candidates each candidate is less and less likely to be better than all the ones prior. More precisely, candidate 2 has probability 1/2 of being the best of the first 2 candidates; candidate 3 has probability 1/3 of being the best of the first 3 candidates; etc.*)

# Randomised Version of Hiring Problem(contd...)

- ❖ By Lemma 1,  $E[X_i] = 1/i$ , a fact that lets us compute  $E[X]$  (*notice that this follows the proof pattern outlined above!*):

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^n X_i\right] \\ &= \sum_{i=1}^n E[X_i] \\ &= \sum_{i=1}^n 1/i \end{aligned}$$

$$\begin{aligned} H_n &= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} \\ &= \sum_{k=1}^n \frac{1}{k} \\ &= \ln n + O(1). \end{aligned}$$

- ❖ The sum is a harmonic series, From formula A7 in appendix A, the  $n^{\text{th}}$  **harmonic number** is given above as  $H_n$
- ❖ Thus, the expected hiring cost is  $\Theta(c_h \ln n)$ , much better than worst case  $\Theta(c_h n)!$  ( $\ln$  is the natural log, where  $\ln n = \Theta(\lg n)$ )
- ❖ The modification to HIRE-ASSISTANT is trivial: add a line at the beginning that randomizes the list of candidates 1. *randomly permute the list of candidates*

# Randomized Quick Sort

- ❖ We randomized our algorithm by explicitly permuting the input in previous example
- ❖ We could do so for quicksort also, but a different randomization technique, called random sampling, yields a simpler analysis
- ❖ Instead of always using  $A[r]$  as the pivot, we will use a randomly chosen element from the subarray  $A[p \dots r]$
- ❖ We do so by exchanging element  $A[r]$  with an element chosen at random from  $A[p \dots r]$
- ❖ This modification, in which we randomly sample the range  $p, \dots, r$ , ensures that the pivot element  $x = A[r]$  is equally likely to be any of the  $r - p + 1$  elements in the subarray
- ❖ Because the pivot element is randomly chosen, we expect the split of the input array to be reasonably well balanced on average
- ❖ The changes to PARTITION and QUICKSORT are small. In the new partition procedure, we simply implement the swap before actually partitioning:

```
RANDOMIZED-PARTITION(A, p, r)
1  i ← RANDOM(p, r)
2  exchange A[r] ↔ A[i]
3  return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2    then q ← RANDOMIZED-PARTITION(A, p, r)
3    RANDOMIZED-QUICKSORT(A, p, q - 1)
4    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

- ❖ The new quicksort calls RANDOMIZED-PARTITION in place of PARTITION:

# Partition Algo of Quick Sort

```
PARTITION( $A, p, r$ )
```

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

$$\begin{aligned} T(n) &= T(n - 1) + T(0) + \Theta(n) \\ &= T(n - 1) + \Theta(n). \end{aligned}$$

$$\begin{aligned} T(n) &\leq \max_{0 \leq q \leq n-1} (cq^2 + c(n - q - 1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \leq q \leq n-1} (q^2 + (n - q - 1)^2) + \Theta(n). \end{aligned}$$

Worst Case Complexity  
:  $O(n^2)$

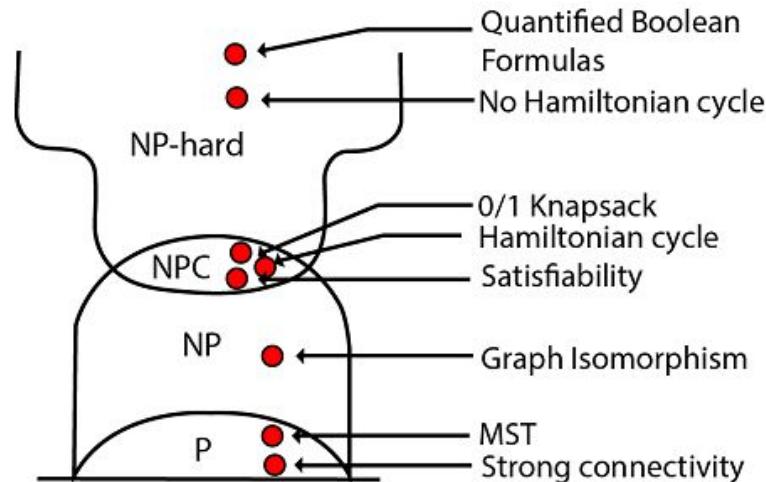


# Introduction to Complexity Classes

- ❖ Almost all the algorithms we have studied thus far have been polynomial-time algorithms: on inputs of size  $n$ , their worst-case running time is  $O(n^k)$  for some constant  $k$
- ❖ It is natural to wonder whether all problems can be solved in polynomial time, the answer is no
- ❖ For example, there are problems, such as Turing's famous "Halting Problem," that cannot be solved by any computer, no matter how much time is provided
- ❖ An interesting class of problems, called the "NP- complete" problems, whose status is unknown
- ❖ No polynomial-time algorithm has yet been discovered for an NP-complete problem, nor has anyone yet been able to prove that no polynomial-time algorithm can exist for any one of them
- ❖ Problems of P vs NP class
  - Shortest vs. longest simple paths
  - Euler tour vs. hamiltonian cycle

# Complexity Classes

Pictorial representation of all NP classes which includes NP, NP-hard, and NP-complete



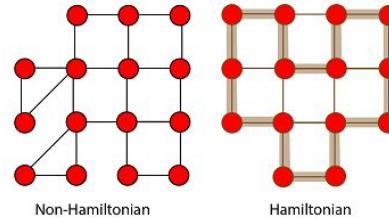
# P-type Problems

- ❖ The class P consists of those problems that are solvable in polynomial time
- ❖ More specifically, they are problems that can be solved in time  $O(n^k)$  for some constant k, where n is the size of the input to the problem
- ❖ The set of decision-based problems come into the division of P Problems who can be solved or produced an output within polynomial time
- ❖ P problems being easy to solve
- ❖ If we produce an output according to the given input within a specific amount of time such as within a minute, hours, this is known as Polynomial time
- ❖ Before talking about the class of NP-complete problems, it is essential to introduce the notion of a verification algorithm
- ❖ Many problems are hard to solve, but they have the property that it easy to authenticate the solution if one is provided
- ❖ We say that a function  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  is polynomial-time computable if there exists a polynomial-time algorithm A that, given any input  $x \in \{0, 1\}^*$ , produces as output  $f(x)$

# Introduction to NP type problems

- ❖ The class NP consists of those problems that are "verifiable" in polynomial time
- ❖ What we mean here is that if we were somehow given a "certificate" of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem
- ❖ For example, in the hamiltonian-cycle problem, given a directed graph  $G = (V, E)$ , a certificate would be a sequence  $v_1, v_2, v_3, \dots, v_{|V|}$  of  $|V|$  vertices
  - It is easy to check in polynomial time that  $(v_i, v_{i+1}) \in E$  for  $i = 1, 2, 3, \dots, |V| - 1$  and that  $(v_{|V|}, v_1) \in E$  as well
- ❖ Any problem in P is also in NP, since if a problem is in P then we can solve it in polynomial time without even being given a certificate
- ❖ The term "NP" does not mean "not polynomial" Originally, the term meant "non-deterministic polynomial", It means according to the one input number of output will be produced
- ❖ The set of all decision-based problems came into the division of NP Problems who can't be solved or produced an output within polynomial time but verified in the **polynomial time**
- ❖ NP class contains P class as a subset, NP problems being hard to solve
- ❖ If we produce an output according to the given input but there are no time constraints is known as Non-Polynomial time
- ❖ But yes output will produce but time is not fixed yet

# Hamiltonian Cycle Problem



- ❖ Consider the Hamiltonian cycle problem:
  - Given an undirected graph G, does G have a cycle that visits each vertex exactly once? There is no known polynomial time algorithm for this dispute
- ❖ It means you can't build a Hamiltonian cycle in a graph with a polynomial time even if there is no specific path is given for the Hamiltonian cycle with the particular vertex, yet you can't verify the Hamiltonian cycle within the polynomial time
- ❖ Let us understand that a graph above did have a Hamiltonian cycle
- ❖ It would be easy for someone to convince of this, They would similarly say: "the period is hv3, v7, v1....v13i.
- ❖ We could then inspect the graph and check that this is indeed a legal cycle and that it visits all of the vertices of the graph exactly once
- ❖ Thus, even though we know of no efficient way to solve the Hamiltonian cycle problem, there is a beneficial way to verify that a given cycle is indeed a Hamiltonian cycle
- ❖ For the verification in the Polynomial-time of an undirected Hamiltonian cycle graph G
- ❖ There must be exact/specific/definite path that must be given of Hamiltonian cycle then you can verify in the polynomial time
- ❖ A piece of information which contains in the given path of a vertex is known as certificate

# Relation of P and NP classes

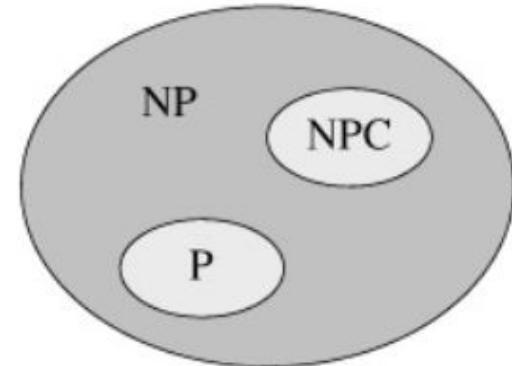
- ❖ The relationship between P and NP class is as follows:

- **P contains in NP:**

- Observe that P contains in NP
    - In other words, if we can solve a problem in polynomial time, we can indeed verify the solution in polynomial time
    - More formally, we do not need to see a certificate (there is no need to specify the vertex/intermediate of the specific path) to solve the problem; we can explain it in polynomial time anyway

- **P = NP:**

- However, it is not known whether  $P = NP$
    - It seems you can verify and produce an output of the set of decision-based problems in NP classes in a polynomial time which is impossible because according to the definition of NP classes you can verify the solution within the polynomial time
    - So this relation can never be held

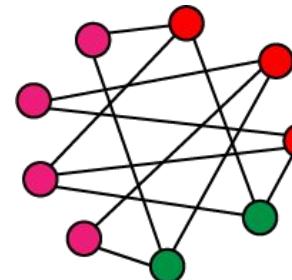


# Reductions

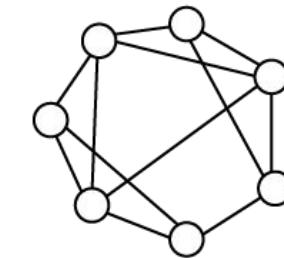
- ❖ The class NP-complete (NPC) problems consist of a set of decision problems (a subset of class NP) that no one knows how to solve efficiently
- ❖ But if there were a polynomial solution for even a single NP-complete problem, then every problem in NPC will be solvable in polynomial time
- ❖ For this, we need the concept of reductions
- ❖ Suppose there are two problems, A and B
  - You know that it is impossible to solve problem A in polynomial time
  - You want to prove that B cannot be solved in polynomial time
  - We want to show that  $(A \notin P) \Rightarrow (B \notin P)$
- ❖ Polynomial Time Reduction
  - We say that Decision Problem  $L_1$  is Polynomial time Reducible to decision Problem  $L_2$  ( $L_1 \leq_p L_2$ ) if there is a polynomial time computation function f such that for all  $x$ ,  $x \in L_1$  if and only if  $x \in L_2$

# Reduction Example

- ❖ Consider an example to illustrate reduction:
- ❖ The following problem is well-known to be NPC:
- ❖ **3-color:** Given a graph  $G$ , can each of its vertices be labeled with one of 3 different colors such that two adjacent vertices do not have the same label (color)
- ❖ Coloring arises in various partitioning issues where there is a constraint that two objects cannot be assigned to the same set of partitions
- ❖ The phrase "coloring" comes from the original application which was in map drawing
- ❖ Two countries that contribute a common border should be colored with different colors
- ❖ It is well known that planar graphs can be colored (maps) with four colors
- ❖ There exists a polynomial time algorithm for this
- ❖ But deciding whether this can be done with 3 colors is hard, and there is no polynomial time algorithm for it



3-Colorable



Not 3-Colorable

# NP Hard Problems

- ❖ Here you have to satisfy the following points to come into the division of NP-hard
  - If we can solve this problem in polynomial time, then we can solve all NP problems in polynomial time
  - If you convert the issue into one form to another form within the polynomial time

# Introduction to NP Complete Problem

- ❖ A problem is in the class NPC-and we refer to it as being NP-complete-if it is in NP and is as "hard" as any problem in NP
- ❖ A problem is in NP-complete, if
  - It is in NP
  - It is NP-hard
- ❖ L is NP-complete if
  - $L \in NP$  and
  - $L' \leq p L$  for some known NP-complete problem  $L'$  Given this formal definition, the complexity classes are:
    - **P:** is the set of decision problems that are solvable in polynomial time
    - **NP:** is the set of decision problems that can be verified in polynomial time
    - **NP-Hard:** L is NP-hard if for all  $L' \in NP$ ,  $L' \leq p L$ , Thus if we can solve L in polynomial time, we can solve all NP problems in polynomial time
    - **NP-Complete** L is NP-complete if
      - $L \in NP$  and
      - L is NP-hard
  - If any NP-complete problem is solvable in polynomial time, then every NP-Complete problem is also solvable in polynomial time
  - Conversely, if we can prove that any NP-Complete problem cannot be solved in polynomial time, every NP-Complete problem cannot be solvable in polynomial time

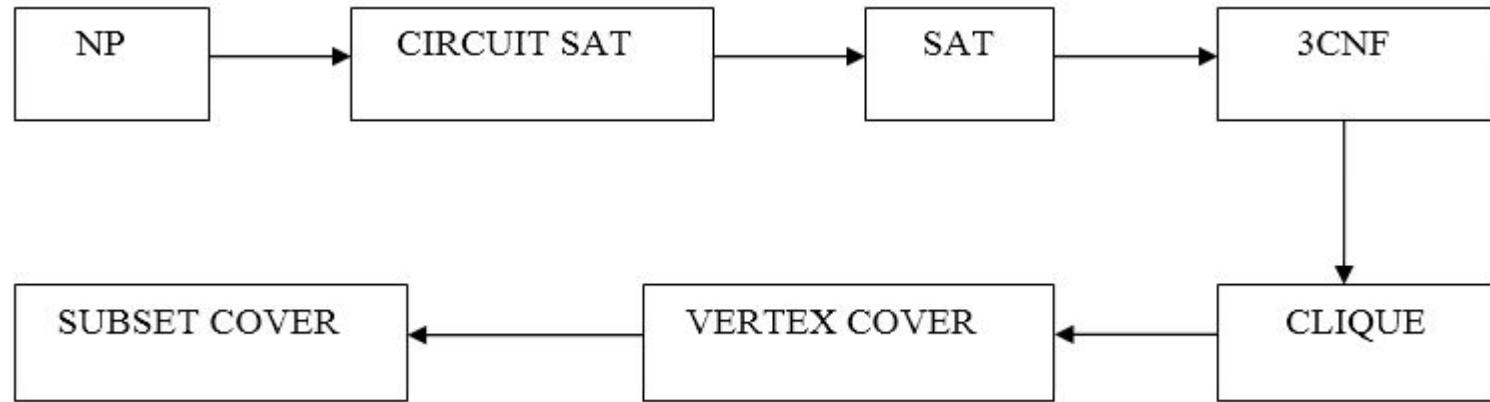
# NPC Problems Reduction

- ❖ If the solution of NPC problem does not exist then the conversion from one NPC problem to another NPC problem within the polynomial time
- ❖ For this, you need the concept of reduction, If a solution of the one NPC problem exists within the polynomial time, then the rest of the problem can also give the solution in polynomial time (but it's hard to believe)
- ❖ For this, you need the concept of reduction
- ❖ Suppose there are two problems, **A** and **B**
  - You know that it is impossible to solve problem **A** in polynomial time
  - You want to prove that **B** cannot be solved in polynomial time
  - So you can convert the problem **A** into problem **B** in polynomial time
- ❖ **NP problem:** Suppose a DECISION-BASED problem is provided in which a set of inputs/high inputs you can get high output

# NPC Reductions (contd...)

- ❖ **Criteria to come either in NP-hard or NP-complete**
  - The point to be noted here, the output is already given, and you can verify the output/solution within the polynomial time but can't produce an output/solution in polynomial time
  - Here we need the concept of reduction because when you can't produce an output of the problem according to the given input then in case you have to use an emphasis on the concept of reduction in which you can convert one problem into another problem
- ❖ If you satisfy both points then your problem comes into the category of NP-complete class
- ❖ If you satisfy the only 2nd points then your problem comes into the category of NP-hard class
- ❖ So according to the given decision-based NP problem, you can decide in the form of yes or no
- ❖ If, yes then you have to do verify and convert into another problem via reduction concept
- ❖ If you are performing both, then decision-based NP problems are in NP compete

# NPC Problems



# Satisfiability Problem

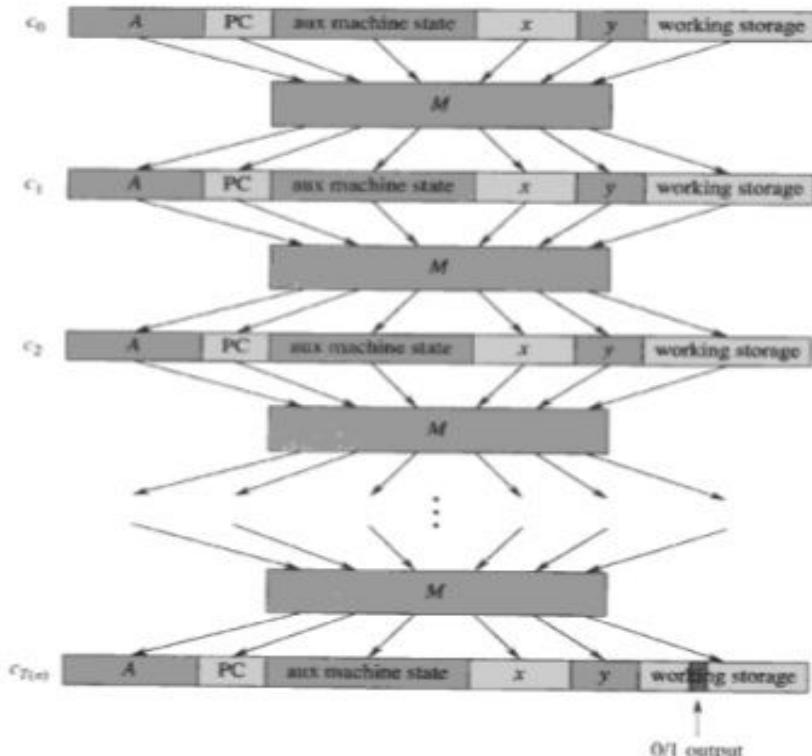
- ❖ Given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?
- ❖ The size of a boolean combinational circuit is the number of boolean combinational elements plus the number of wires in the circuit
- ❖ One can devise a graph like encoding that maps any given circuit C into a binary string C whose length is polynomial in the size of the circuit itself
- ❖ As a formal language, we can therefore define
  - $\text{CIRCUIT-SAT} = \{C : C \text{ is a satisfiable boolean combinational circuit}\}$
- ❖ The circuit-satisfiability problem arises in the area of computer-aided hardware optimization
- ❖ Given a circuit C, we might attempt to determine whether it is satisfiable by simply checking all possible assignments to the inputs
- ❖ Unfortunately, if there are k inputs, there are  $2^k$  possible assignments. When the size of C is polynomial in k, checking each one takes  $\Omega(2^k)$  time
- ❖ There is strong evidence that no polynomial-time algorithm exists that solves the circuit-satisfiability problem because circuit satisfiability is NP-complete

# Circuit Satisfiability belongs to class NP

- ❖ CIRCUIT-SAT is poly-time verifiable
  - Given (an encoding of) a CIRCUIT-SAT problem  $C$  and a certificate, which is an assignment of boolean values to (all) wires in  $C$
  - The algorithm is constructed as follows: just checks each gates and then the output wire of  $C$ :
    - If for every gate, the computed output value matches the value of the output wire given in the certificate and the output of the whole circuit is 1, then the algorithm outputs 1, otherwise 0
  - The algorithm is executed in poly time (even linear time)
  - An alternative certificate: a true assignment to the inputs

# Circuit SAT is NP Hard

- ❖ Suppose  $X$  is any problem in NP
  - Construct a poly-time algorithm  $F$  maps every problem instance  $x$  in  $X$  to a circuit  $C=f(x)$  such that the answer to  $x$  is YES if and only if  $C \in \text{CIRCUIT-SAT}$  (is satisfiable)
- ❖ Since  $X \in \text{NP}$ , there is a poly-time algorithm  $A$  which verifies  $X$
- ❖ Suppose the input length is  $n$  and Let  $T(n)$  denote the worst-case running time. Let  $k$  be the constant such that  $T(n)=O(n^k)$  and the length of the certificate is  $O(n^k)$
- ❖ Idea is to represent the computation of  $A$  as a sequence of configurations,  $c_0, c_1, \dots, c_i, c_{i+1}, \dots, c_{T(n)}$ , each  $c_i$  can be broken into
  - (program for  $A$ , program counter PC, auxiliary machine state, input  $x$ , certificate  $y$ , working storage) and
  - $c_i$  is mapped to  $c_{i+1}$  by the combinational circuit  $M$  implementing the computer hardware
  - The output of  $A$ : 0 or 1, is written to some designated location in working storage
  - If the algorithm runs for at most  $T(n)$  steps, the output appears as one bit in  $c_{T(n)}$
  - Note:  $A(x,y)=1$  or 0



**Figure 34.9** The sequence of configurations produced by an algorithm  $A$  running on an input  $x$  and certificate  $y$ . Each configuration represents the state of the computer for one step of the computation and, besides  $A$ ,  $x$ , and  $y$ , includes the program counter (PC), auxiliary machine state, and working storage. Except for the certificate  $y$ , the initial configuration  $c_0$  is constant. Each configuration is mapped to the next configuration by a boolean combinational circuit  $M$ . The output is a distinguished bit in the working storage.

# Circuit SAT is NP Hard (contd...)

- ❖ The reduction algorithm F constructs a single combinational circuit C as follows:
  - Paste together all  $T(n)$  copies of the circuit M
  - The output of the  $i$ th circuit, which produces  $c_i$ , is directly fed into the input of the  $(i+1)$ st circuit
  - All items in the initial configuration, except the bits corresponding to certificate  $y$ , are wired directly to their known values
  - The bits corresponding to  $y$  are the inputs to C
  - All the outputs to the circuit are ignored, except the one bit of  $c_{T(n)}$  corresponding to the output of A
- ❖ Two properties remain to be proven:
  - F correctly constructs the reduction, i.e., C is satisfiable if and only if there exists a certificate  $y$ , such that  $A(x,y) = 1$ 
    - Suppose there is a certificate  $y$ , such that  $A(x,y) = 1$ , Then if we apply the bits of  $y$  to the inputs of C, the output of C is the bit of  $A(x,y)$ , that is  $C(y) = A(x,y) = 1$ , so C is satisfiable
    - Suppose C is satisfiable, then there is a  $y$  such that  $C(y)=1$ . So,  $A(x,y) = 1$
  - F runs in poly time

# Circuit- SAT is NP Complete

- ❖ F runs in poly time
  - Poly space:
    - Size of x is n
    - Size of A is constant, independent of x
    - Size of y is  $O(n^k)$
    - Amount of working storage is poly in n since A runs at most  $O(n^k)$
    - M has size poly in length of configuration, which is poly in  $O(n^k)$ , and hence is poly in n
    - C consists of at most  $O(n^k)$  copies of M, and hence is poly in n
    - Thus, the C has poly space
  - The construction of C takes at most  $O(n^k)$  steps and each step takes poly time, so F takes poly time to construct C from x
- ❖ CIRCUIT-SAT is NP-complete
  - CIRCUIT-SAT belongs to NP, verifiable in poly time
  - CIRCUIT-SAT is NP-hard, every NP problem can be reduced to CIRCUIT-SAT in poly time
  - Thus CIRCUIT-SAT is NP-complete

# Formula Satisfiability(SAT)

- ❖ SAT definition
  - n boolean variables:  $x_1, x_2, \dots, x_n$
  - M boolean connectives: any boolean function with one or two inputs and one output, such as  $\wedge, \vee, \neg, \rightarrow, \leftrightarrow, \dots$  and
  - Parentheses
- ❖ A SAT  $\varphi$  is satisfiable if there exists an true assignment which causes  $\varphi$  to evaluate to 1. • SAT={< $\varphi$ >:  $\varphi$  is a satisfiable boolean formula}
- ❖ The historical honor of the first NP-complete problem ever shown
- ❖ SAT is NP-complete
  - SAT belongs to NP
    - Given a satisfying assignment, the verifying algorithm replaces each variable with its value and evaluates the formula, in poly time
  - SAT is NP-hard (show CIRCUIT-SAT $\leq_p$  SAT)

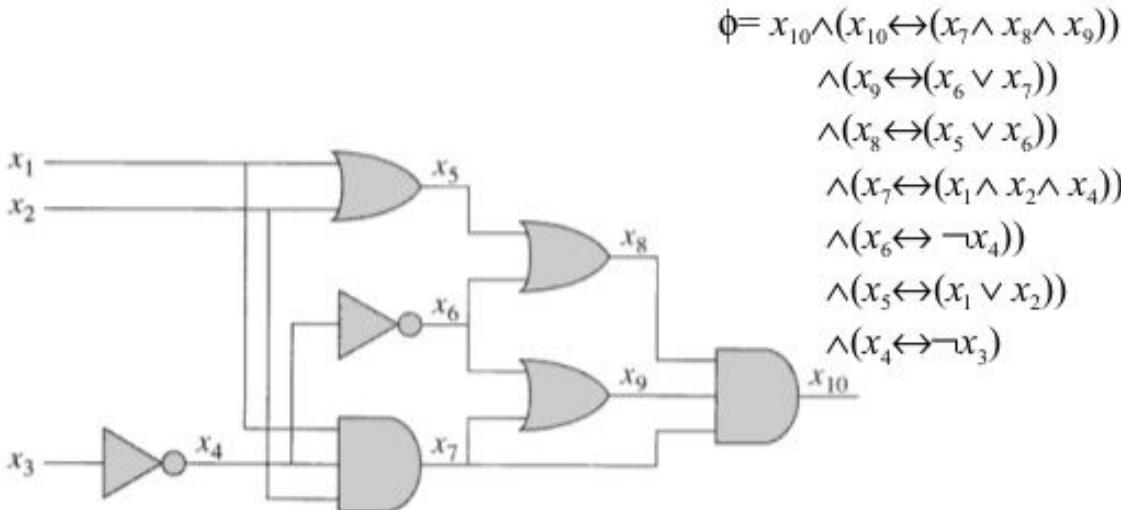
# SAT is NP Hard

- ❖ CIRCUIT-SAT $\leq p$  SAT, i.e., any instance of circuit satisfiability can be reduced in poly time to an instance of formula satisfiability
- ❖ Intuitive induction:
  - Look at the gate that produces the circuit output
  - Inductively express each of gate's inputs as formulas
  - Formula for the circuit is then obtained by writing an expression that applies the gate's function to its input formulas
- ❖ Unfortunately, this is not a poly reduction
  - Shared formula (the gate whose output is fed to 2 or more inputs of other gates) cause the size of generated formula to grow exponentially
- ❖ Correct reduction:
  - For every wire  $x_i$  of  $C$ , give a variable  $x_i$  in the formula
  - Every gate can be expressed as  $x_0 \leftrightarrow (x_{i1} \wedge x_{i2} \wedge \dots \wedge x_{il})$
  - The final formula  $\varphi$  is the AND of the circuit output variable and conjunction of all clauses describing the operation of each gate

# SAT is NP Complete(contd...)

- ❖ Correctness of the reduction
  - Clearly the reduction can be done in poly time
  - $C$  is satisfiable if and only if  $\varphi$  is satisfiable
    - If  $C$  is satisfiable, then there is a satisfying assignment this means that each wire of  $C$  has a well-defined value and the output of  $C$  is 1. Thus, the assignment of wire values to variables in  $\varphi$  makes each clause in  $\varphi$  evaluate to 1. So  $\varphi$  is 1
    - The reverse proof can be done in the same way

## Example of reduction of CIRCUIT-SAT to SAT



**Figure 34.10** Reducing circuit satisfiability to formula satisfiability. The formula produced by the reduction algorithm has a variable for each wire in the circuit.

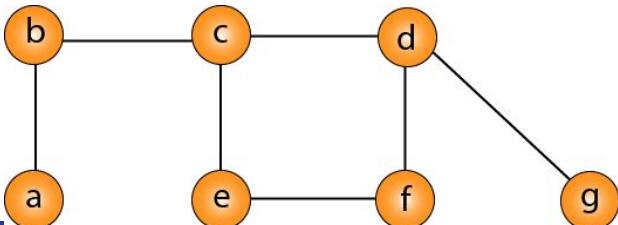
INCORRECT REDUCTION:  $\phi = x_{10} = x_7 \wedge x_8 \wedge x_9 = (x_1 \wedge x_2 \wedge x_4) \wedge (x_5 \vee x_6) \wedge (x_6 \vee x_7)$   
 $= (x_1 \wedge x_2 \wedge x_4) \wedge ((x_1 \vee x_2) \vee \neg x_4) \wedge (\neg x_4 \vee (x_1 \wedge x_2 \wedge x_4)) = \dots$

# Introduction to Approximation Algorithms

- ❖ An Approximate Algorithm is a way of approach **NP-COMPLETENESS** for the optimization problem
  - This technique does not guarantee the best solution
  - The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at the most polynomial time
  - Such algorithms are called approximation algorithm or heuristic algorithm
- ❖ For the traveling salesperson problem, the optimization problem is to find the shortest cycle, and the approximation problem is to find a short cycle
- ❖ For the vertex cover problem, the optimization problem is to find the vertex cover with fewest vertices, and the approximation problem is to find the vertex cover with few vertices
- ❖ An Approximate Algorithm returns a legal solution, but the cost of that legal solution may not be optimal
- ❖ Suppose we are considering for a **minimum size vertex-cover (VC)**, An approximate algorithm returns a VC for us, but the size (cost) may not be minimized

# Vertex Covering Problem

- ❖ A Vertex Cover of a graph  $G$  is a set of vertices such that each edge in  $G$  is incident to at least one of these vertices
- ❖ The decision vertex-cover problem was proven NPC
- ❖ Now, we want to solve the optimal version of the vertex cover problem, i.e., we want to find a minimum size vertex cover of a given graph
- ❖ We call such vertex cover an optimal vertex cover  $C^*$

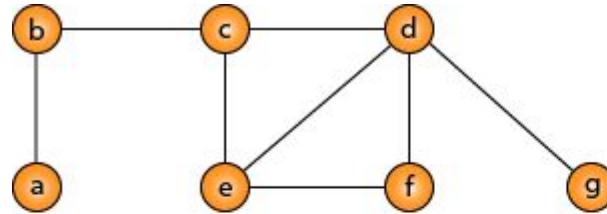


- ❖ An approximate algorithm for vertex cover:

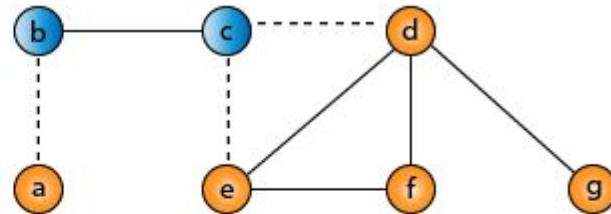
```
Approx-Vertex-Cover (G = (V, E))
{
    C = empty-set;
    E' = E;
    While E' is not empty do
    {
        Let (u, v) be any edge in E': (*)
        Add u and v to C;
        Remove from E' all edges incident to
        u or v;
    }
    Return C;
}
```

# Vertex Cover Problem (contd...)

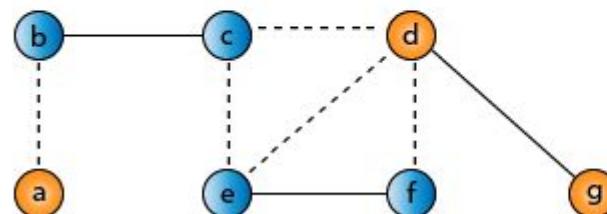
- The idea is to take an edge  $(u, v)$  one by one, put both vertices to  $C$ , and remove all the edges incident to  $u$  or  $v$ . We carry on until all edges have been removed.  $C$  is a VC. But how good is  $C$ ?



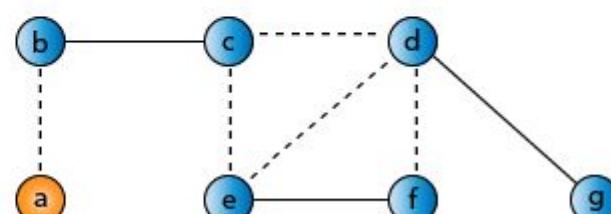
(1)



(2)



(3)



(4)

- $VC = \{b, c, d, e, f, g\}$

# References

- ❖ <https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/>
- ❖ <https://www.geeksforgeeks.org/algorithms-qq/pattern-searching/#algo>
- ❖ <https://www.javatpoint.com/daa-string-matching-introduction>
- ❖ <https://www.javatpoint.com/daa-rabin-karp-algorithm>
- ❖ Introduction To Algorithms, 2nd Edition by Cormen
- ❖ <http://www2.hawaii.edu/~suthers/courses/ics311f20/Notes/Topic-05.html#:~:text=An%20algorithm%20is%20randomized%20if,us%20choose%20the%20interview%20order.>
- ❖ <https://www.javatpoint.com/daa-approximation-algorithm-vertex-cover>
- ❖ <https://www.slideshare.net/rajendranjrf/np-completeness-71853663>
- ❖ <https://www.javatpoint.com/daa-approximate-algorithms>
- ❖