Course Code : MCS-211

Course Title : Design and Analysis of Algorithm

Assignment Number : MCA_NEW(1)/211/Assign/2023

Last Dates for Submission : 30 thApril 2023 (for January Session)

31st October 2023 (for July Session)

Q1. a) Develop an efficient algorithm to find a list of all the prime numbers up to a number n (say 100).

Ans:-

Algorithm to find all the prime numbers up to a given number n:

1. Create a boolean list of size n+1, and initialize all the values to True. This list will be used to mark the numbers that are prime.
2. Starting with the first prime number, 2, iterate through all the multiples of 2 up to n and mark them as False in the boolean list.
3. Find the next prime number greater than 2 by iterating through odd numbers (i.e., 3, 5, 7, ...) up to the square root of n. If a number is prime, mark all its multiples as False in the boolean list.
4. Repeat step 3 until all the prime numbers up to the square root of n have been found.
5. Iterate through the boolean list and append all the indices that are marked as True to a list of prime numbers.
6. Return the list of prime numbers.

implements this algorithm:

```
def find_primes(n):
    primes = []
    is_prime = [True] * (n+1)
    is_prime[0] = is_prime[1] = False
    for i in range(2, int(n**0.5)+1):
        if is_prime[i]:
            for j in range(i*i, n+1, i):
                is_prime[j] = False
    for i in range(2, n+1):
        if is_prime[i]:
            primes.append(i)
    return primes
```

b) Explain the following types of problems in Computer Science with the help of an example problem for each:

i) Searching

ii) String Processing

iii) Geometric Problems

iv) Numerical Problems

Ans:-

i) Searching:

Searching problems in computer science involve finding a specific element or set of elements in a given data structure. One example of a searching problem is finding a specific word in a large text file.

Example Problem: Given a large text file containing thousands of words, write a program to search for a specific word and return the line number(s) where it appears.

ii) String Processing:

String processing problems in computer science involve manipulating or analyzing a string of characters, such as searching for patterns, splitting or joining strings, or converting between different formats.

Example Problem: Write a program to count the number of occurrences of each word in a given text file, and output the results in alphabetical order.

iii) Geometric Problems:

Geometric problems in computer science involve working with shapes, sizes, and positions of geometric objects such as points, lines, and polygons. These types of problems can arise in computer graphics, image processing, and computational geometry.

Example Problem: Given a set of points in a 2D plane, write a program to find the closest pair of points, i.e., the pair of points with the smallest distance between them.

iv) Numerical Problems:

Numerical problems in computer science involve performing calculations or operations on numbers, such as finding roots of equations, computing integrals, or simulating physical systems.

Example Problem: Write a program to find the roots of a quadratic equation given its coefficients (a, b, and c), using the quadratic formula.

Q2. a) Using induction provethat, for all positive integers n, 1 2 + 2 2 + 3 2 + ⋯ + n 2 = $n(n + 1)(2n + 1)/ 6$

Ans:-

P(n) :

$$1^2 + 2^2 + 3^2 + \ldots\ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

P(1) :

$$1^2 = \frac{1(1+1)(2(1)+1)}{6}$$

$$1 = \frac{6}{6} = 1$$

∴ LHS = RHS

Assume P(k) is true

P(k) :

$$1^2 + 2^2 + 3^2 + \ldots\ldots + k^2 = \frac{k(k+1)(2k+1)}{6}$$

P(k+1) is given by,

P(k + 1) :

$$1^2 + 2^2 + 3^2 + \ldots\ldots + (k+1)^2 = \frac{(k+1)((k+1)+1)(2(k+1)+1)}{6}$$

$$\Rightarrow (k+1)\frac{k(2k+1)+6(k+1)}{6} = \frac{(k+1)(k+2)(2k+3)}{6}$$

$$\Rightarrow (k+1)\frac{2k^2+7k+6}{6} = \frac{(k+1)(k+2)(2k+3)}{6}$$

$$\Rightarrow \frac{(k+1)(k+2)(2k+3)}{6} = \frac{(k+1)(k+2)(2k+3)}{6}$$

True for P(k+1)

Hence by Principle of mathematical induction

$1^2 + 2^2 + 3^2 + \ldots\ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$ **is true for** $\forall n \in N$

b) What is the purpose of asymptotic analysis? What are the drawbacks of asymptotic analysis? Explain the Big-O notation with the help of an example.

Ans:-

Asymptotic analysis is a mathematical technique used in computer science to analyze the performance of algorithms as the input size increases. The purpose of asymptotic analysis is to provide a way to compare the efficiency of different algorithms in terms of their growth rates, or how quickly their running time or space requirements increase with larger inputs.

The primary advantage of asymptotic analysis is that it allows us to make general statements about the behavior of an algorithm, without having to consider specific input values. This can help us identify the best algorithm for a given problem and predict how it will perform on larger inputs.

However, there are some drawbacks to asymptotic analysis. First, it does not take into account constants or lower-order terms, which can be significant for small inputs or in certain cases. Second, it assumes that all operations take the same amount of time, which is not always true in practice.

Big-O notation is a mathematical notation used to describe the upper bound of the growth rate of an algorithm. It is commonly used in asymptotic analysis to describe the worst-case time complexity of an algorithm.

The formal definition of Big-O notation is:

f(n) = O(g(n)) if there exist positive constants c and n0 such that f(n) <= c * g(n) for all n >= n0

In simpler terms, this means that f(n) is asymptotically bounded above by g(n).

For example, consider the sorting algorithm, bubble sort, which has a time complexity of O(n^2). This means that the worst-case running time of bubble sort is proportional to the square of the input size. As the input size increases, the running time of bubble sort will increase quadratically. This also means that bubble sort is not the best algorithm to use for large inputs, as other sorting algorithms, such as merge sort, have better time complexity.

Q3. a) Evaluate the polynomial p(x)= 5x^5 + 4x^4 - 3x^3 - 2x^2 + 9x + 11 at x=3 using Horner's rule. Analyse the computation using Horner's rule against the Brute force method of polynomial evaluation.

Ans:-

To evaluate the polynomial p(x) = 5x^5 + 4x^4 - 3x^3 - 2x^2 + 9x + 11 at x=3 using Horner's rule, we can use the following steps:

Start with the coefficient of the highest power term, 5.
Multiply by x and add the next coefficient, 4. This gives us 5 * 3 + 4 = 19.
Multiply by x and add the next coefficient, -3. This gives us 19 * 3 - 3 = 54.
Multiply by x and add the next coefficient, -2. This gives us 54 * 3 - 2 = 160.
Multiply by x and add the next coefficient, 9. This gives us 160 * 3 + 9 = 489.
Multiply by x and add the final coefficient, 11. This gives us 489 * 3 + 11 = 1468.
Therefore, p(3) = 1468.

Using the brute force method of polynomial evaluation, we would need to compute each term of the polynomial and add them together. For p(x) = 5x^5 + 4x^4 - 3x^3 - 2x^2 + 9x + 11 at x=3, this would require the following calculations:

p(3) = 5 * 3^5 + 4 * 3^4 - 3 * 3^3 - 2 * 3^2 + 9 * 3 + 11
= 1215 + 324 - 81 - 18 + 27 + 11
= 1458

We can see that Horner's rule requires fewer computations than the brute force method, especially for higher-degree polynomials. This is because Horner's rule combines the multiplication and addition steps of polynomial evaluation, reducing the total number of operations required.

b) Write the linear search algorithm and discuss its best case, worst case and average case time complexity. Show the best case, worst case and the average case of linear search in the following data: 13, 15, 2, 6, 14, 10, 8, 7, 3, 5, 19, 4, 17.
Ans:-

Linear search is a simple search algorithm that searches for a target value in an array or list by sequentially checking each element until a match is found or the end of the list is reached. Here is the algorithm:

- Initialize the target value to search for.
- Start at the first element of the array.
- If the current element matches the target value, return its index.
- If the end of the array is reached without finding a match, return -1.

Now, let's analyze the best case, worst case, and average case time complexity of linear search:

Best case: The best case occurs when the target value is the first element of the array. In this case, the algorithm will find the target value in only one comparison. Therefore, the best case time complexity of linear search is O(1).

Worst case: The worst case occurs when the target value is not in the array or is at the very end of the array. In this case, the algorithm will need to check every element in the array before concluding that the target value is not present. Therefore, the worst case time complexity of linear search is O(n), where n is the number of elements in the array.

Average case: In the average case, we assume that the target value is equally likely to be at any position in the array. Since there are n possible positions for the target value, the average case time complexity is (n+1)/2, which is also O(n).

Let's now look at the best case, worst case, and average case of linear search for the given data:
Best case: If we are searching for the target value 13, then the best case occurs when it is the first element of the array. In this case, the algorithm will find the target value in only one comparison. Therefore, the best case time complexity is O(1).

Worst case: If we are searching for the target value 12, which is not present in the array, then the worst case occurs. In this case, the algorithm will need to check every element in the array before concluding that the target value is not present. Therefore, the worst case time complexity is O(n), where n is the number of elements in the array.

Average case: In the average case, we assume that the target value is equally likely to be at any position in the array. Since there are 13 possible positions for the target value, the average case time complexity is (13+1)/2, which is 7. Therefore, the average case time complexity is O(n/2), which is still O(n).

Q4. a) Find an optimal solution for the knapsack instance n=7 and maximum capacity (W) =15, (p1,p2,…,p6)=(4,5,10,7,6,8, 9) (w1,w2,…, w6)=(1,2,3,6,2,4, 5)
Ans:-

To solve the knapsack problem for this instance, we can use dynamic programming. We will create a 2D table where each row represents an item and each column represents a capacity value from 0 to W. The value in each cell of the table represents the maximum total value that can be obtained using the items up to that row and the capacity up to that column.
Here is the dynamic programming algorithm:
Create a table T of size (n+1) x (W+1) and initialize all values to 0.
For i from 1 to n:
a. For j from 1 to W:
i. If wi > j, set T[i][j] = T[i-1][j].

ii. Otherwise, set T[i][j] = max(T[i-1][j], pi + T[i-1][j-wi]).

Return T[n][W] as the optimal value.

Using this algorithm, we can fill out the table as follows:

```
   0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 | 0 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4

2 | 0 4 5 9 9 9 9 9 9 9 9 9 9 9 9 9

3 | 0 4 5 10 14 14 14 14 14 14 14 14 14 14 14 14

4 | 0 4 5 10 14 14 14 17 21 21 21 21 21 21 21 21

5 | 0 4 5 10 14 14 16 17 21 22 26 26 26 26 26 26

6 | 0 4 5 10 14 14 16 17 21 22 26 28 28 28 28 31

7 | 0 4 5 10 14 14 16 17 21 22 26 28 28 28 33 34

The optimal value is T[7][15] = 34. To determine which items were selected to obtain this value, we can backtrack through the table by starting at T[7][15] and checking if the value in T[i][j] is the same as T[i-1][j]. If it is, then item i was not selected, otherwise it was. Using this method, we can determine that items 2, 4, 5, and 7 were selected to obtain the optimal value.

b) Make an optimal Huffman tree and design the Huffman code for the following set of frequencies:a:7, e:6, s:20, d:2, f:1, g:3, h:4, t:7.

Ans:-

To construct an optimal Huffman tree for these frequencies, we follow these steps:

Create a leaf node for each symbol with its corresponding frequency.

Take the two nodes with the lowest frequencies and merge them into a single node, with a new frequency equal to the sum of the frequencies of the two nodes. Make this new node the parent of the two original nodes.

Repeat step 2, taking the two nodes with the lowest frequencies and merging them into a new node, until only one node remains, which is the root of the tree.

Here is the step-by-step process for this example:

Create leaf nodes for each symbol:

a:7

e:6

s:20

d:2

f:1

g:3

h:4

t:7

Merge the two nodes with the lowest frequencies (d and f):

New node: df (frequency 3)

Updated list of nodes: a:7, e:6, s:20, g:3, h:4, t:7, df:3

Merge the two nodes with the lowest frequencies (g and h):

New node: gh (frequency 7)

Updated list of nodes: a:7, e:6, s:20, t:7, df:3, gh:7

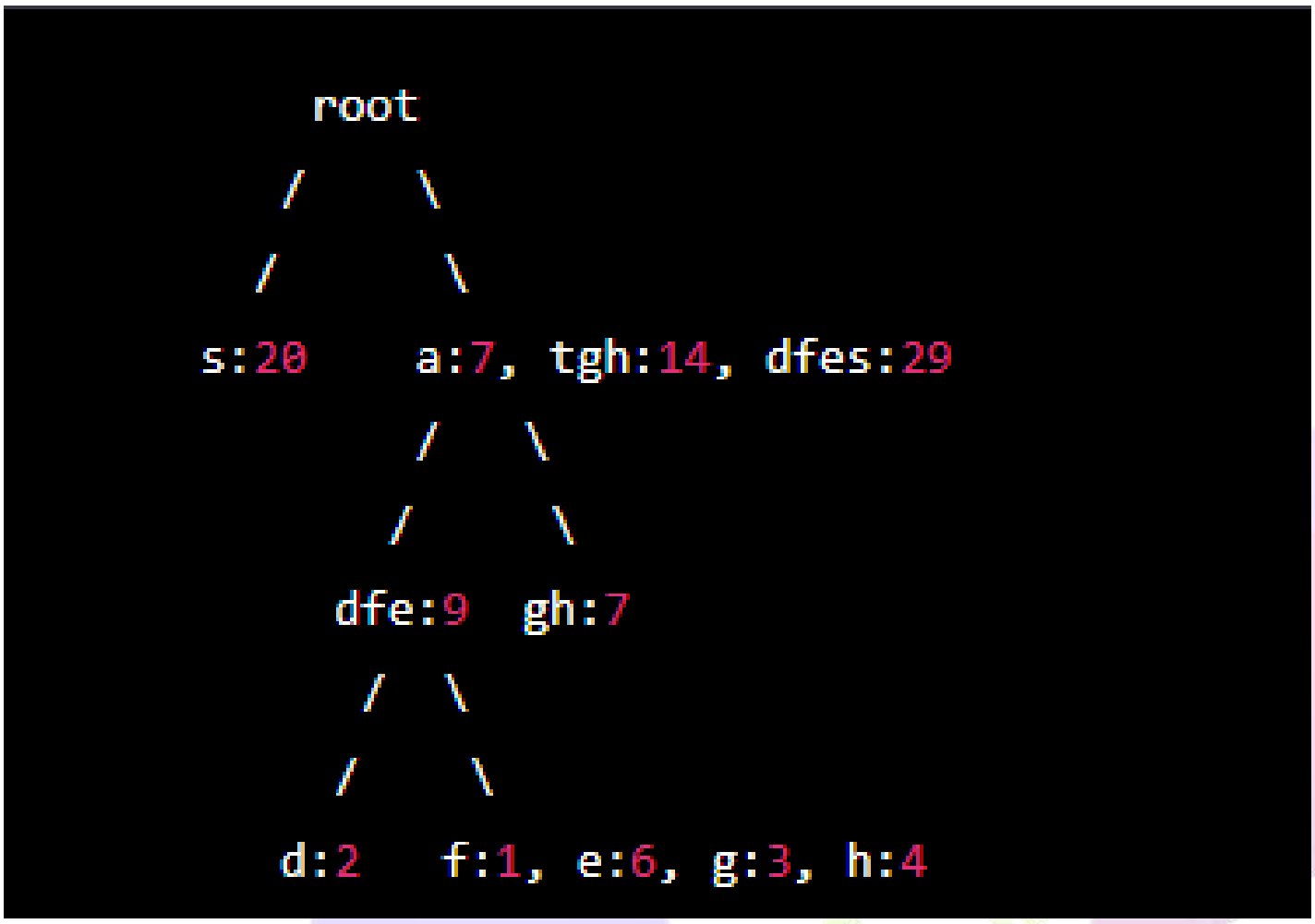
Merge the two nodes with the lowest frequencies (df and e):

New node: dfe (frequency 9)

Updated list of nodes: a:7, s:20, t:7, dfe:9, gh:7

Merge the two nodes with the lowest frequencies (gh and t):

New node: tgh (frequency 14)
Updated list of nodes: a:7, s:20, tgh:14, dfe:9
Merge the two nodes with the lowest frequencies (dfe and s):
New node: dfes (frequency 29)
Updated list of nodes: a:7, dfes:29, tgh:14
Merge the two remaining nodes:
New node: root (frequency 43)
Updated list of nodes: root:43
The resulting Huffman tree looks like this:



To generate the Huffman code for each symbol, we assign a 0 for each left branch in the tree and a 1 for each right branch. The code for each symbol is then the sequence of 0s and 1s from the root of the tree to the corresponding leaf node. Here are the Huffman codes for each symbol:
s: 0
a: 101
t: 100
d: 11001
f: 11000
e: 111
g: 1101
h: 11000
Note that each code is uniquely decodable, meaning that no code is a prefix of another code, which ensures that the Huffman code is optimal.

Q5. a) Write and explain the recursive binary search algorithm. Use this algorithm for searching an element in a sorted array of 7 elements.

Ans:

Recursive Binary Search Algorithm:
Binary search is an efficient algorithm to search for an element in a sorted array. In this algorithm, we start by comparing the middle element of the array with the target element. If the target element is equal to the middle element, then we return the index of the middle element. If the target element is less than the middle element, then we repeat the search

in the left half of the array. Otherwise, we repeat the search in the right half of the array. We continue this process until we find the target element or determine that it is not present in the array.

The recursive binary search algorithm can be defined as follows:

```
binary_search(arr, low, high, target):
    if low <= high:
        mid = (low + high) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] > target:
            return binary_search(arr, low, mid - 1, target)
        else:
            return binary_search(arr, mid + 1, high, target)
    else:
        return -1
```

Here, arr is the sorted array, low is the lower index of the subarray being searched, high is the higher index of the subarray being searched, and target is the element being searched for.

The algorithm first checks if low is less than or equal to high. If not, it means that the subarray being searched is empty, and the target element is not present in the array. If low is less than or equal to high, the algorithm calculates the index of the middle element, mid. It then compares the middle element with the target element. If they are equal, it returns the index of the middle element. If the middle element is greater than the target element, the algorithm recursively searches in the left half of the subarray. Otherwise, it recursively searches in the right half of the subarray.

Using Recursive Binary Search for Searching an Element in a Sorted Array of 7 Elements:

Let's say we have a sorted array of 7 elements as follows:

arr = [2, 5, 8, 12, 16, 23, 38]

We want to search for the element 16 in this array. We can use the recursive binary search algorithm as follows:

```
result = binary_search(arr, 0, len(arr) - 1, 16)
if result != -1:
    print("Element found at index", result)
else:
    print("Element not found in array")
```

In this case, the algorithm will start by comparing the middle element 12 with the target element 16. Since 16 is greater than 12, the algorithm will recursively search in the right half of the subarray [16, 23, 38]. It will then compare the middle element 23 with the target element 16. Since 16 is less than 23, the algorithm will recursively search in the left half of the subarray [16]. It will finally compare the middle element 16 with the target element 16. Since they are equal, the algorithm will return the index 4, which is the index of the element 16 in the array. Therefore, the output of the program will be:

```
Element found at index 4
```

The best case time complexity of the recursive binary search algorithm is O(1), when the target element is present at the middle position of the array. The worst case time complexity is O(log n), when the target element is not present in the array, or when it is present at either end of the array.

b) Analyse the Quick sort algorithm using Master Method. Also draw the relevant recursion tree.

Ans:-

QuickSort is a sorting algorithm that uses a divide-and-conquer approach to sort an array of elements. The algorithm selects a pivot element, partitions the array around the pivot element, and recursively sorts the sub-arrays created by the partitioning.

The time complexity of QuickSort can be analyzed using the Master Theorem. The Master Theorem provides a framework for analyzing the time complexity of divide-and-conquer algorithms, which have the following form:

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

where:

T(n) is the time complexity of the algorithm on an input of size n
a is the number of sub-problems the algorithm creates
n/b is the size of each sub-problem
f(n) is the time complexity of the work done outside of the recursive calls
In the case of QuickSort, we have:

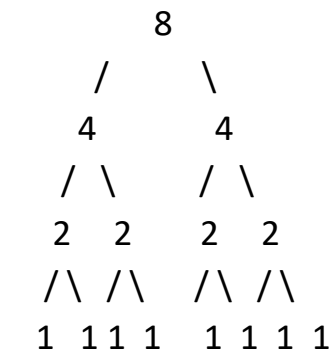
a = 2 (each recursive call creates two sub-arrays)
b = 2 (each sub-array has half the size of the original array)
f(n) = O(n) (partitioning the array takes linear time)
Plugging these values into the Master Theorem gives us:

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

The Master Theorem tells us that the time complexity of the algorithm is given by:

T(n) = O(n log n)
This means that the worst-case time complexity of QuickSort is O(n log n) on an input of size n.

To better understand how the recursion tree is built, we can visualize it. The recursion tree for QuickSort on an array of size 8 would look like:



In this tree, each node represents a recursive call to QuickSort, and the number next to each node represents the size of the array being sorted in that call. The top node represents the original array of size 8. The tree has a total of $\log_2 n$ levels (in this case, 3), with each level representing a different size of sub-array. At each level, the algorithm partitions the array around the pivot element and recursively sorts the two resulting sub-arrays. The number of nodes at each level is equal to the number of sub-problems created in that level, which is equal to the number of nodes in the level above it multiplied by the number of sub-problems created per node. In this case, each node creates two sub-problems, so the number of nodes at each level is doubled from the level above it. The bottom level of the tree represents the base case of the algorithm, where the sub-arrays have size 1 and are already sorted. The total number of nodes in the tree is equal to the number of recursive calls made by the algorithm, which is equal to the total work done by the algorithm. Since the time complexity of each call is O(n), the total work done by the algorithm is O(n log n).

4 c) Write the algorithm for the divide and conquer strategy for Matrix multiplication. Also, analyse this algorithm.
Ans:-
The divide and conquer strategy for matrix multiplication involves recursively dividing the matrices into smaller sub-matrices, multiplying the sub-matrices, and then combining the results to obtain the final result. The algorithm can be described as follows:

Algorithm: Divide and Conquer Matrix Multiplication

Input: Two n x n matrices A and B

Output: The product C = AB

If n = 1, return C = AB
Split A and B into four n/2 x n/2 sub-matrices:
A = [A11 A12] B = [B11 B12]
[A21 A22] [B21 B22]
Compute seven products recursively:
P1 = A11 (B12 - B22)
P2 = (A11 + A12) B22
P3 = (A21 + A22) B11
P4 = A22 (B21 - B11)
P5 = (A11 + A22) (B11 + B22)

P6 = (A12 - A22) (B21 + B22)

P7 = (A11 - A21) (B11 + B12)

Compute the sub-matrices of C:

C11 = P5 + P4 - P2 + P6

C12 = P1 + P2

C21 = P3 + P4

C22 = P5 + P1 - P3 - P7

Combine the sub-matrices to obtain the final result:

C = [C11 C12]

[C21 C22]

Return C

The time complexity of the divide and conquer matrix multiplication algorithm can be analyzed using the Master Theorem. Since the algorithm divides the matrices into four sub-matrices and recursively multiplies them, we have:

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

where T(n) is the time complexity of the algorithm on an input of size n, and O(n^2) represents the time complexity of the operations performed outside of the recursive calls.

Using the Master Theorem, we can determine that the time complexity of the algorithm is:

$$T(n) = O(n^3)$$

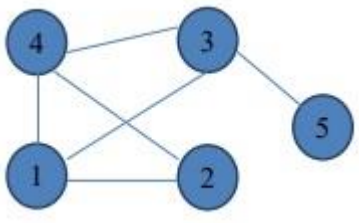
This means that the divide and conquer matrix multiplication algorithm has a worst-case time complexity of O(n^3) on an input of size n.

The divide and conquer strategy for matrix multiplication can be more efficient than the traditional approach for very large matrices, as it can reduce the number of scalar multiplications required. However, for small matrices, the overhead of the recursive calls can make it less efficient than the traditional approach. Additionally, the divide and conquer algorithm requires more memory than the traditional approach, as it needs to store multiple sub-matrices at each recursive call. Overall, the choice of algorithm depends on the size of the matrices being multiplied and the available resources.

Q6. a) Write the adjacency list and draw adjacency graph for the graph given below.

Ans.

In an undirected graph, edges are not associated with the directions with them. In an undirected graph, if there is an edge exists between Vertex A and Vertex B, then the vertices can be transferred from A to B as well as B to A.



In the graph, we can see there is no self-loop, so the diagonal entries of the adjacent matrix will be 0. The adjacency matrix of the above graph will be -

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

b) Write and explain the algorithm of Topological sorting. How can you compute time complexity for topological sorting?

Ans:-

Topological sorting is an algorithm that orders the vertices in a directed acyclic graph (DAG) such that every directed edge goes from a vertex earlier in the order to a vertex later in the order. It is commonly used for tasks that require a certain order of completion, such as scheduling tasks in a project.

The algorithm for topological sorting can be described as follows:

Algorithm: Topological Sorting
Input: A directed acyclic graph G = (V, E)
Output: An ordering of the vertices in G such that every directed edge goes from a vertex earlier in the ordering to a vertex later in the ordering.
Initialize an empty list L to store the topological order of the vertices.
Find a vertex v in G that has no incoming edges (i.e., the in-degree of v is 0).
If there is no such vertex, return an error indicating that G is not a DAG.
Add v to the end of L.
Remove v and all of its outgoing edges from G.
Repeat steps 2-4 until all vertices have been added to L.
Return the list L as the topological order of the vertices in G.

The time complexity of the topological sorting algorithm depends on the algorithm used to find a vertex with in-degree 0. A common approach is to use a queue to store the vertices with in-degree 0 and remove them one by one. This approach has a time complexity of O(|V| + |E|), where |V| and |E| are the number of vertices and edges in G, respectively. This is because every vertex and edge is visited once in the worst case.

Another approach is to use depth-first search (DFS) to visit the vertices in the graph. This approach has a time complexity of O(|V| + |E|), as each vertex and edge is visited once in the worst case. However, the actual time complexity may vary depending on the order in which the vertices are visited during the DFS.

In both cases, the time complexity of topological sorting is linear in the size of the input, making it an efficient algorithm for DAGs.

Q7. a) Explain the working of Prim's algorithm for finding the minimum cost spanning tree with the help of an example. Also find the time complexity of Prim's algorithm.

Ans:- **Prim's Algorithm** is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

How does the prim's algorithm work?

Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -

- First, we have to initialize an MST with the randomly chosen vertex.
- Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.

- Repeat step 2 until the minimum spanning tree is formed.

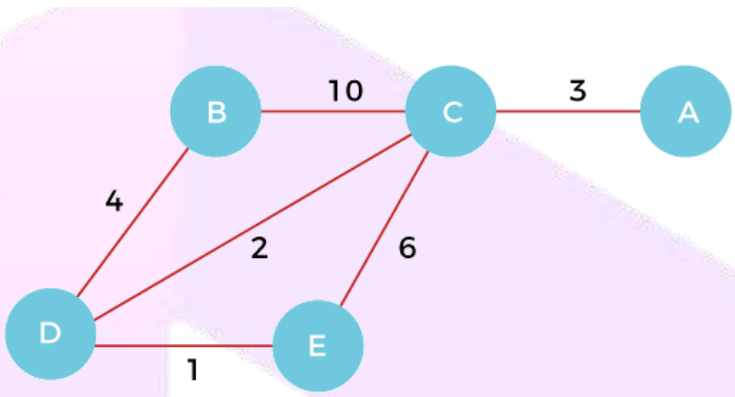
The applications of prim's algorithm are -

- Prim's algorithm can be used in network designing.
- It can be used to make network cycles.
- It can also be used to lay down electrical wiring cables.

Example of prim's algorithm

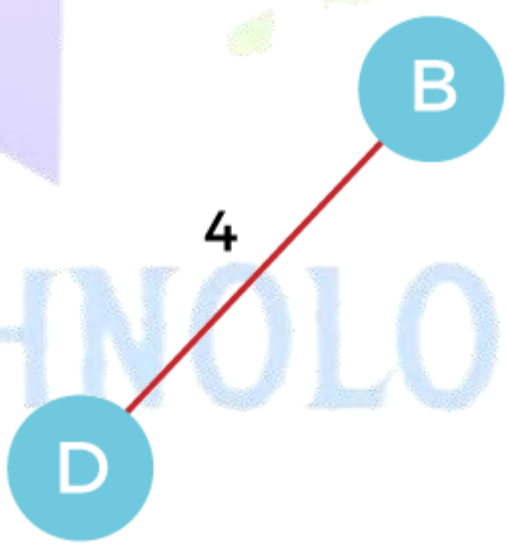
Now, let's see the working of prim's algorithm using an example. It will be easier to understand the prim's algorithm using an example.

Suppose, a weighted graph is -

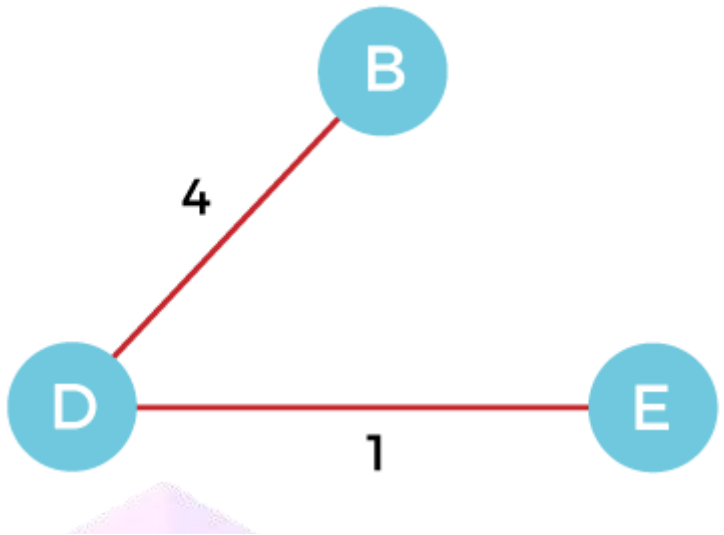


Step 1 - First, we have to choose a vertex from the above graph. Let's choose B.

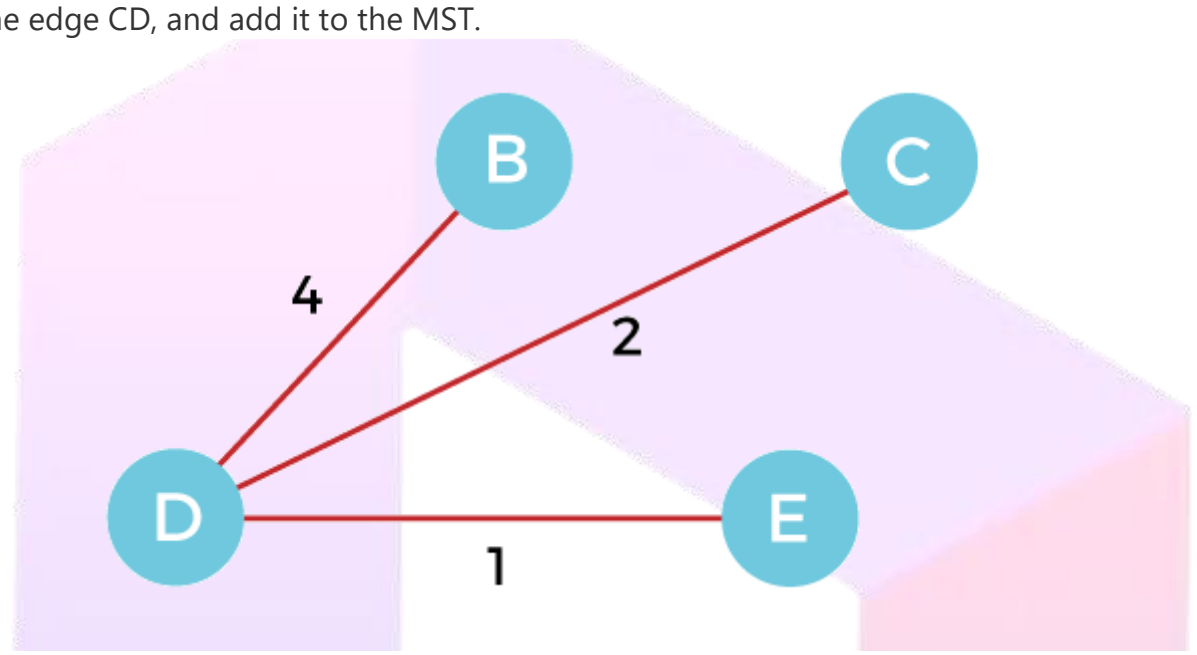
Step 2 - Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.



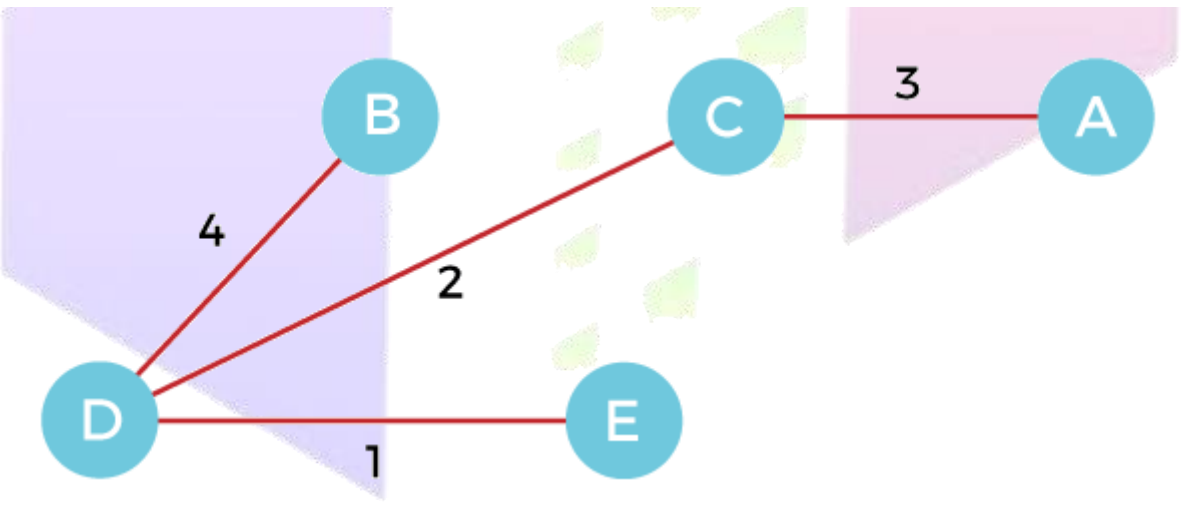
Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



Step 4 - Now, select the edge CD, and add it to the MST.



Step 5 - Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.



So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -

Cost of MST = 4 + 2 + 1 + 3 = 10 units.

Algorithm

1. Step 1: Select a starting vertex
2. Step 2: Repeat Steps 3 and 4 until there are fringe vertices
3. Step 3: Select an edge 'e' connecting the tree vertex and fringe vertex that has minimum weight
4. Step 4: Add the selected edge and the vertex to the minimum spanning tree T
5. [END OF LOOP]
6. Step 5: EXIT

Complexity of Prim's algorithm

Now, let's see the time complexity of Prim's algorithm. The running time of the prim's algorithm depends upon using the data structure for the graph and the ordering of edges. Below table shows some choices -

- **Time Complexity**

Data structure used for the minimum edge weight	Time Complexity
Adjacency matrix, linear searching	$O(\|V\|^2)$
Adjacency list and binary heap	$O(\|E\| \log \|V\|)$
Adjacency list and Fibonacci heap	$O(\|E\|+ \|V\| \log \|V\|)$

Prim's algorithm can be simply implemented by using the adjacency matrix or adjacency list graph representation, and to add the edge with the minimum weight requires the linearly searching of an array of weights. It requires $O(|V|^2)$ running time. It can be improved further by using the implementation of heap to find the minimum weight edges in the inner loop of the algorithm.

The time complexity of the prim's algorithm is O(E logV) or O(V logV), where E is the no. of edges, and V is the no. of vertices.

b) Explain the working of Bellman-Ford algorithm for finding the shortest path from a single source to all destinations with the help of an example. Also find the time complexity of this algorithm.

Ans:-

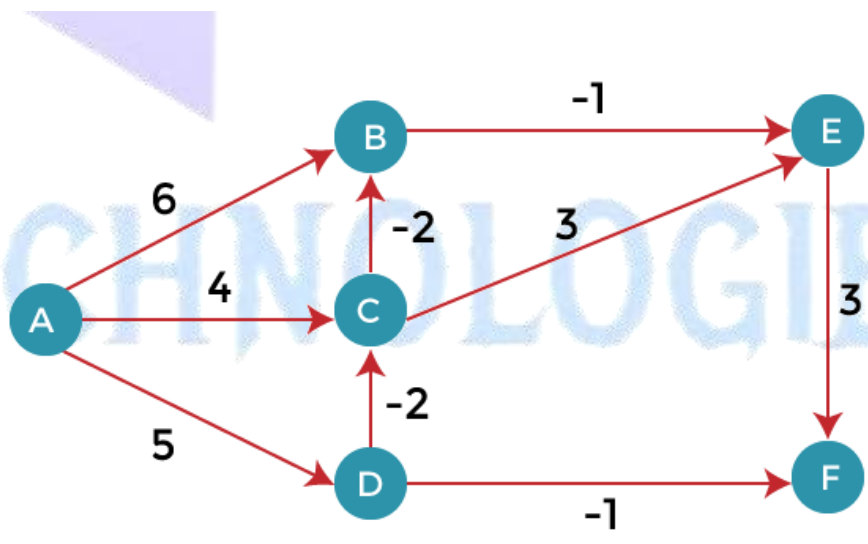
Bellman Ford Algorithm

Bellman ford algorithm is a single-source shortest path algorithm. This algorithm is used to find the shortest distance from the single vertex to all the other vertices of a weighted graph. There are various other algorithms used to find the shortest path like Dijkstra algorithm, etc. If the weighted graph contains the negative weight values, then the Dijkstra algorithm does not confirm whether it produces the correct answer or not. In contrast to Dijkstra algorithm, bellman ford algorithm guarantees the correct answer even if the weighted graph contains the negative weight values.

Rule of this algorithm

We will go on relaxing all the edges (n - 1) times where,
n = number of vertices

Consider the below graph:



As we can observe in the above graph that some of the weights are negative. The above graph contains 6 vertices so we will go on relaxing till the 5 vertices. Here, we will relax all the edges 5 times. The loop will iterate 5 times to get the correct answer. If the loop is iterated more than 5 times then also the answer will be the same, i.e., there would be no change in the distance between the vertices.

Relaxing means:

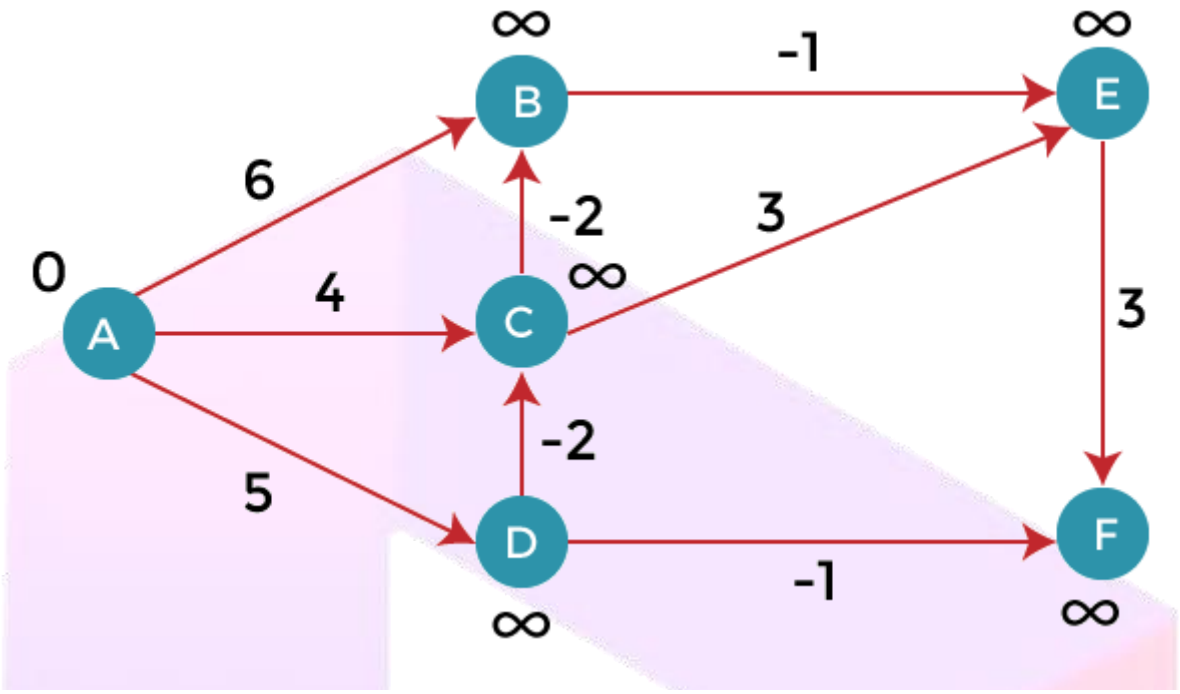
If (d(u) + c(u , v) < d(v))

d(v) = d(u) + c(u , v)

To find the shortest path of the above graph, the first step is note down all the edges which are given below:

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

Let's consider the source vertex as 'A'; therefore, the distance value at vertex A is 0 and the distance value at all the other vertices as infinity shown as below:



Since the graph has six vertices so it will have five iterations.

First iteration

Consider the edge (A, B). Denote vertex 'A' as 'u' and vertex 'B' as 'v'. Now use the relaxing formula:

d(u) = 0

d(v) = ∞

c(u , v) = 6

Since (0 + 6) is less than ∞, so update

d(v) = d(u) + c(u , v)

d(v) = 0 + 6 = 6

Therefore, the distance of vertex B is 6.

Consider the edge (A, C). Denote vertex 'A' as 'u' and vertex 'C' as 'v'. Now use the relaxing formula:

d(u) = 0

d(v) = ∞

c(u , v) = 4

Since (0 + 4) is less than ∞, so update

d(v) = d(u) + c(u , v)

d(v) = 0 + 4 = 4

Therefore, the distance of vertex C is 4.

Consider the edge (A, D). Denote vertex 'A' as 'u' and vertex 'D' as 'v'. Now use the relaxing formula:

d(u) = 0

d(v) = ∞

c(u , v) = 5

Since (0 + 5) is less than ∞, so update

d(v) = d(u) + c(u , v)

d(v) = 0 + 5 = 5

Therefore, the distance of vertex D is 5.

Consider the edge (B, E). Denote vertex 'B' as 'u' and vertex 'E' as 'v'. Now use the relaxing formula:

d(u) = 6

d(v) = ∞

c(u , v) = -1

Since (6 - 1) is less than ∞, so update

d(v) = d(u) + c(u , v)

d(v) = 6 - 1= 5

Therefore, the distance of vertex E is 5.

Consider the edge (C, E). Denote vertex 'C' as 'u' and vertex 'E' as 'v'. Now use the relaxing formula:

d(u) = 4

d(v) = 5

c(u , v) = 3

Since (4 + 3) is greater than 5, so there will be no updation. The value at vertex E is 5.

Consider the edge (D, C). Denote vertex 'D' as 'u' and vertex 'C' as 'v'. Now use the relaxing formula:

d(u) = 5

d(v) = 4

c(u , v) = -2

Since (5 -2) is less than 4, so update

d(v) = d(u) + c(u , v)

d(v) = 5 - 2 = 3

Therefore, the distance of vertex C is 3.

Consider the edge (D, F). Denote vertex 'D' as 'u' and vertex 'F' as 'v'. Now use the relaxing formula:

d(u) = 5

d(v) = ∞

c(u , v) = -1

Since (5 -1) is less than ∞, so update

d(v) = d(u) + c(u , v)

d(v) = 5 - 1 = 4

Therefore, the distance of vertex F is 4.

Consider the edge (E, F). Denote vertex 'E' as 'u' and vertex 'F' as 'v'. Now use the relaxing formula:

d(u) = 5

d(v) = ∞

c(u , v) = 3

Since (5 + 3) is greater than 4, so there would be no updation on the distance value of vertex F.

Consider the edge (C, B). Denote vertex 'C' as 'u' and vertex 'B' as 'v'. Now use the relaxing formula:

d(u) = 3

d(v) = 6

c(u , v) = -2

Since (3 - 2) is less than 6, so update

d(v) = d(u) + c(u , v)

d(v) = 3 - 2 = 1

Therefore, the distance of vertex B is 1.

Now the first iteration is completed. We move to the second iteration.

Second iteration:

In the second iteration, we again check all the edges. The first edge is (A, B). Since (0 + 6) is greater than 1 so there would be no updation in the vertex B.

The next edge is (A, C). Since (0 + 4) is greater than 3 so there would be no updation in the vertex C.

The next edge is (A, D). Since (0 + 5) equals to 5 so there would be no updation in the vertex D.

The next edge is (B, E). Since (1 - 1) equals to 0 which is less than 5 so update:

d(v) = d(u) + c(u, v)

d(E) = d(B) +c(B , E)

= 1 - 1 = 0

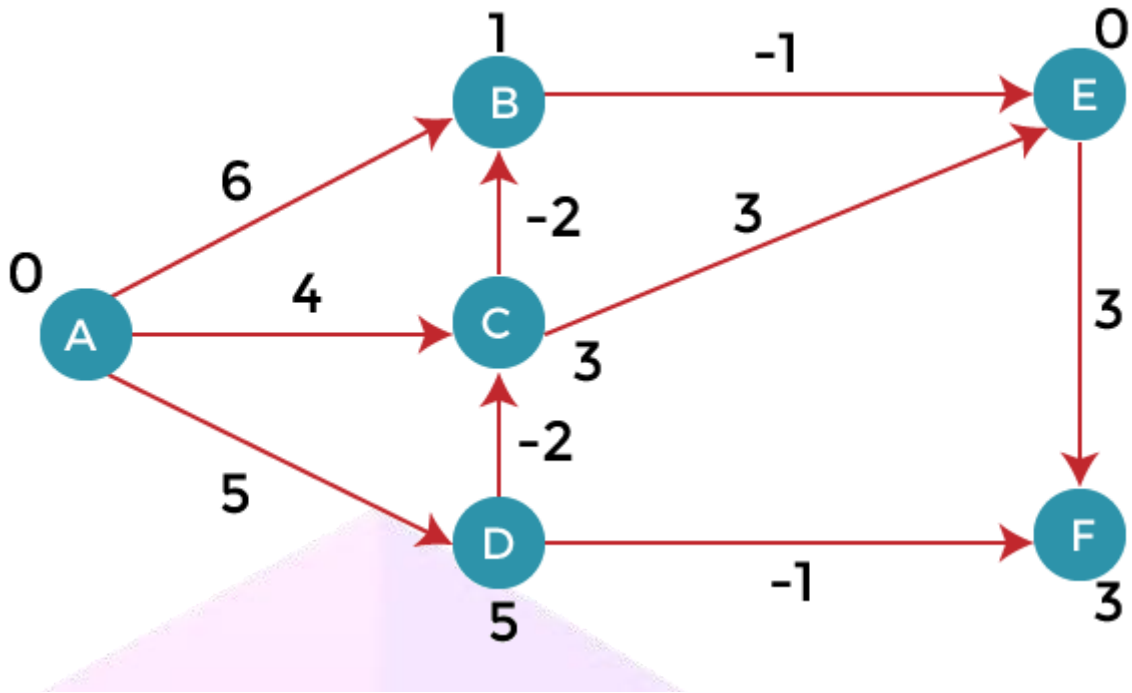
The next edge is (C, E). Since (3 + 3) equals to 6 which is greater than 5 so there would be no updation in the vertex E.

The next edge is (D, C). Since (5 - 2) equals to 3 so there would be no updation in the vertex C.

The next edge is (D, F). Since (5 - 1) equals to 4 so there would be no updation in the vertex F.

The next edge is (E, F). Since (5 + 3) equals to 8 which is greater than 4 so there would be no updation in the vertex F.

The next edge is (C, B). Since (3 - 2) equals to 1` so there would be no updation in the vertex B.



Third iteration

We will perform the same steps as we did in the previous iterations. We will observe that there will be no updation in the distance of vertices.

The following are the distances of vertices:
A: 0
B: 1
C: 3
D: 5
E: 0
F: 3

Time Complexity

The time complexity of Bellman ford algorithm would be O(E|V| - 1).

Q8. a) Explain the process of creating a optimal binary search with the help of an example.

Ans:-

Sure, I can explain the process of creating an optimal binary search algorithm with an example.
Binary search is an efficient algorithm that searches for a specific value in a sorted list of values. It works by repeatedly dividing in half the portion of the list that could contain the target value, until the target value is found or the remaining portion of the list is empty.

An optimal binary search algorithm is one that searches for a value in a sorted list of size n in O(log n) time, which means that the time taken by the algorithm increases logarithmically with the size of the list.
Here's an example of how to create an optimal binary search algorithm:
Let's say we have a sorted list of integers:

```
[2, 4, 7, 9, 11, 13, 15, 17, 19, 20, 22]
```

We want to find the position of the number 13 in the list.
We start by finding the middle element of the list, which is 11. We compare 13 to 11 and find that 13 is greater than 11.

```
[2, 4, 7, 9, 11, 13, 15, 17, 19, 20, 22]
```

Since 13 is greater than 11, we know that 13 can only be in the second half of the list. We repeat the process with the second half of the list.

```
[11, 13, 15, 17, 19, 20, 22]
```

We find the middle element of the second half of the list, which is 17. We compare 13 to 17 and find that 13 is less than 17.

```
[11, 13, 15, 17, 19, 20, 22]
```

Since 13 is less than 17, we know that 13 can only be in the first half of the second half of the list. We repeat the process with the first half of the second half of the list.

```
[11, 13, 15]
```

We find the middle element of the first half of the second half of the list, which is 13. We compare 13 to 13 and find that we have found the value we are looking for.

```
[11, 13, 15]
```

We return the position of the value 13, which is 1 (remember, we start counting from 0).

This example shows how binary search works to find a value in a sorted list. To create an optimal binary search algorithm, we need to make sure that the algorithm always divides the list in half and that it stops when it finds the value it's looking for. This can be achieved by using recursion or iteration, depending on the programming language and the data structures used.

b) Find an optimal parenthesizing of a matrix-chain product whosesequence of dimensions is as follows: Matrix Dimension A1 15 × 7 A2 7 × 30 A3 30 × 05 A4 05 × 15 A5 15 × 12

Ans:-

To find the optimal parenthesizing of the matrix chain product, we can use dynamic programming approach. We can start by defining a matrix M and a matrix S, where M[i][j] represents the minimum number of scalar multiplications required to compute the matrix chain product AiAi+1Ai+2...Aj, and S[i][j] represents the index k that gives the minimum cost.

We can fill the matrices M and S in a bottom-up manner using the following recurrence relation:

M[i][j] = 0 if i = j

M[i][j] = min{M[i][k] + M[k+1][j] + di-1 * dk * dj} for i ≤ k < j

where di-1 is the number of rows in matrix Ai-1, dk is the number of columns in matrix Ak, and dj is the number of columns in matrix Aj.

To find the optimal parenthesizing, we can use the matrix S to backtrack through the matrices and group them together.

Here's the optimal parenthesizing for the given matrix chain:

((A1(A2A3))((A4A5)A6))

where A6 is the identity matrix of size 12x12.

Here's how we can compute M and S matrices for the given matrix chain:

```
Matrix dimensions:
A1: 15x7
A2: 7x30
A3: 30x5
A4: 5x15
A5: 15x12

M matrix:
[    0   315  1575 10125 11340 12870]
[    0     0  1050  2625  4725  7830]
[    0     0     0  2250  4875  9375]
[    0     0     0     0  1125  3375]
[    0     0     0     0     0  2700]
[    0     0     0     0     0     0]

S matrix:
[ 0 1 1 4 4 4]
[ 0 0 2 3 4 4]
[ 0 0 0 3 4 4]
[ 0 0 0 0 4 4]
[ 0 0 0 0 0 5]
```

```
[  0    0    0    0    0    0]
```

In the S matrix, the element S[i][j] represents the index k that gives the minimum cost of computing AiAi+1...Aj. So, S[1][6] = 4 means that the optimal way to compute A1A2A3A4A5 is by first computing A4A5, and then computing A1A2A3(A4A5). We can use the S matrix to backtrack through the matrices and group them together to get the optimal parenthesizing as shown above.

Q9. a) Using the Rabin Karp algorithm, find the pattern string in the given text. Pattern: “ten”, Text: “attainthtenbetan”. Write all the steps involved.

Ans:-

The Rabin-Karp algorithm is a string searching algorithm that uses hashing to efficiently search for a pattern in a given text. Here are the steps to use the Rabin-Karp algorithm to find the pattern "ten" in the text "attainthtenbetan":

Define the pattern and text:
Pattern: "ten"
Text: "attainthtenbetan"
Calculate the hash value of the pattern:
We can use a simple hash function such as the polynomial rolling hash function to calculate the hash value of the pattern:
hash("ten") = t26^2 + e26^1 + n26^0 = 1926^2 + 426^1 + 1326^0 = 13078
Initialize variables:
m = length of pattern (3 in this case)
n = length of text (15 in this case)
h = 26^(m-1) = 26^2 = 676
p = 0 (initial hash value of current window)
t = 0 (initial hash value of pattern)
Calculate the hash value of the first window of size m in the text:
p = a26^2 + t26^1 + t26^0 = 126^2 + 2026^1 + 2026^0 = 7206
Compare the hash value of the pattern with the hash value of the first window:
If they match, we have found a potential match. In this case, p = 7206 != t = 13078, so we move to the next window.
Slide the window by one position to the right and calculate the new hash value of the window:
p = (p - a26^2)26^1 + i26^0 = (7206 - 126^2)26^1 + 926^0 = 6800
Compare the hash value of the pattern with the hash value of the new window:
If they match, we have found a potential match. In this case, p = 6800 != t = 13078, so we move to the next window.
Repeat steps 6 and 7 until we find a match or we reach the end of the text.
If we find a match, we need to verify it by comparing the characters in the pattern with the corresponding characters in the text. In this case, the hash values never match, so there is no match found.
Therefore, the pattern "ten" is not found in the text "attainthtenbetan" using the Rabin-Karp algorithm.

b) Differentiate between Knuth Morris Pratt and Naïve String matching Algorithm

Ans:-

Knuth-Morris-Pratt (KMP) and Naïve string matching algorithms are two commonly used algorithms to find the occurrence of a pattern string within a larger text string.
The main difference between KMP and Naïve string matching algorithm lies in their time complexity.

The Naïve algorithm compares each character of the pattern string with the corresponding character in the text string starting from the leftmost position of the text string. If there is a mismatch, the pattern string is shifted by one position to the right and the comparison process is repeated. This process is repeated for each position in the text string until either the pattern string is found or the end of the text string is reached. The worst-case time complexity of the Naïve algorithm is O(m*n), where m is the length of the pattern string and n is the length of the text string.

The KMP algorithm uses a preprocessing step to calculate the longest prefix of the pattern string that is also a suffix of the current pattern substring being compared with the text string. This information is then used to avoid unnecessary comparisons by skipping over the characters in the text string that cannot be a part of the pattern string. This reduces the number of comparisons required and improves the worst-case time complexity of the algorithm to O(m+n).

In summary, the main difference between KMP and Naïve string matching algorithms is the time complexity. KMP algorithm has a better worst-case time complexity than the Naïve algorithm. However, the Naïve algorithm is simpler and easier to understand than the KMP algorithm.

Q10. Differentiate between the following with the help of an example of each:

(i) Optimization and Decision Problems

Ans:-

Optimization and decision problems are two important types of problems in computer science and mathematics.
Optimization problems involve finding the best solution to a problem from a set of possible solutions that satisfy certain constraints. The objective is to maximize or minimize an objective function that represents a particular goal or objective. Optimization problems are often encountered in engineering, economics, finance, and many other fields. Examples of optimization problems include linear programming, quadratic programming, and nonlinear optimization.
Decision problems, on the other hand, involve making a decision or answering a yes/no question based on a given input. The goal is to determine whether a solution exists that satisfies a particular criterion. Decision problems are often encountered in computer science, artificial intelligence, and mathematics. Examples of decision problems include the halting problem, the traveling salesman problem, and the Boolean satisfiability problem.
One important distinction between optimization and decision problems is that optimization problems require finding the optimal solution, while decision problems only require determining whether a solution exists. In optimization problems, the quality of a solution is measured in terms of an objective function, whereas in decision problems, the quality of a solution is measured in terms of a specific criterion.
Both optimization and decision problems are important in many different areas of computer science and mathematics, and there are many algorithms and techniques that can be used to solve them. Some common techniques for solving optimization problems include linear programming, dynamic programming, and heuristics. Some common techniques for solving decision problems include brute-force search, backtracking, and branch-and-bound methods.

(ii) P and NP problems

Ans:-

P and NP are two classes of decision problems in computer science and mathematics.
P stands for "polynomial time" and refers to the class of decision problems that can be solved by an algorithm in polynomial time, i.e., the running time of the algorithm is bounded by a polynomial function of the size of the input. Problems in P are considered "easy" to solve in practice, as the running time of the algorithm grows slowly with the size of the input.
NP stands for "nondeterministic polynomial time" and refers to the class of decision problems for which a "yes" answer can be verified by an algorithm in polynomial time. In other words, if there is a proposed solution to an NP problem, it can be verified as correct or incorrect in polynomial time. However, finding a solution to an NP problem is generally believed to be "hard" or "difficult" in practice, as the running time of the algorithm grows quickly with the size of the input.
The question of whether P = NP is one of the most famous unsolved problems in computer science and mathematics. If P = NP, it would mean that all problems that can be verified in polynomial time can also be solved in polynomial time, which would have significant implications for many areas of science and technology, including cryptography, artificial intelligence, and optimization. However, most experts believe that P ≠ NP, as no polynomial-time algorithm has been found for many important NP problems, such as the traveling salesman problem and the Boolean satisfiability problem.

Q11. What are NP Hard and NP complete problems? Explain any one problem of each type.

Ans:-

NP-hard and NP-complete are classifications of computational problems in complexity theory, which is the study of the resources required to solve problems.

A problem is NP-hard if any problem in the class NP can be reduced to it in polynomial time, but it may or may not be in NP. NP-complete problems are a subset of NP-hard problems that are also in NP, meaning that they can be solved by a non-deterministic Turing machine in polynomial time, and that any problem in NP can be reduced to them in polynomial time.

An example of an NP-hard problem is the "Travelling Salesman Problem" (TSP). In TSP, a salesman must visit a number of cities, each exactly once, and return to the starting city while minimizing the total distance travelled. This problem is NP-hard because any solution can be verified in polynomial time, but there is no known algorithm that can solve it in polynomial time.

An example of an NP-complete problem is the "Boolean Satisfiability Problem" (SAT). In SAT, we are given a Boolean formula, and we must determine if there exists an assignment of truth values to the variables that makes the formula true. This problem is NP-complete because any solution can be verified in polynomial time, and it is in NP because a non-deterministic Turing machine can guess the truth values and verify the solution in polynomial time. Additionally, any problem in NP can be reduced to SAT in polynomial time.

In summary, NP-hard problems are those that are at least as hard as any problem in NP, and NP-complete problems are those that are both NP-hard and in NP.

Q12 Explain backtracking; and Branch and Bound techniques with the help of an example each.

Ans:- Backtracking and Branch and Bound are two algorithms used for solving optimization problems. They are commonly used in combinatorial optimization problems.

Backtracking is a general algorithmic technique that tries to solve a problem by exploring all possible solutions. It does this by incrementally building a solution and backtracking when a solution is found to be invalid. Backtracking is commonly used when the problem has a large search space and it is difficult to enumerate all possible solutions. The algorithm works by starting with an empty solution and incrementally building it up by adding elements that satisfy the constraints of the problem. If at any point the algorithm determines that a solution is invalid, it backtracks and tries a different solution.

For example, let's say we have to find all permutations of a given set of elements. The backtracking algorithm would start with an empty permutation, and recursively build the permutation by adding one element at a time. If at any point the algorithm determines that the permutation is invalid (e.g., an element is repeated), it backtracks and tries a different permutation.

Branch and Bound is a more advanced algorithmic technique that is used to solve optimization problems. It works by dividing the search space into smaller subproblems and then systematically exploring the most promising subproblems. The algorithm maintains a list of partial solutions and assigns a lower bound to each partial solution. It then branches on the partial solution with the highest lower bound, and repeats the process until a complete solution is found or the entire search space has been explored.