www.ignousite.com
Course Code : MCS-211
Course Title : Design and Analysis of Algorithm
Assignment Number : MCA(1)/211/Assign/2023
Maximum Marks : 100
Weightage : 30%
Last date of Submission : 30th April, 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. One efficient algorithm to find all prime numbers up to a given number n is the Sieve of Eratosthenes algorithm. This algorithm works by iteratively marking the multiples of each prime number as composite, and then moving on to the next unmarked number until all numbers up to n have been processed.
Here's how you can implement the Sieve of Eratosthenes algorithm to find all prime numbers up to 100:

1. Create a list of all numbers from 2 to n.
2. Start with the first prime number, 2, and mark all of its multiples as composite by setting their corresponding values in the list to 0.
3. Move on to the next unmarked number (in this case, 3) and repeat the process, marking all of its multiples as composite.
4. Continue in this manner, moving on to the next unmarked number and marking its multiples as composite until you reach the square root of n (rounded up to the nearest integer).
5. All unmarked numbers in the list up to n are prime numbers.

Here's the Python code to implement this algorithm:

```
def find_primes(n):
  # create a list of all numbers from 2 to n
  numbers = list(range(2, n+1))
  primes = []

  # iterate over all numbers up to the square root of n
  for i in range(2, int(n**0.5)+1):
    if numbers[i-2] != 0:
      # mark all multiples of i as composite
      for j in range(i**2, n+1, i):
        numbers[j-2] = 0
  # all unmarked numbers are prime
  for num in numbers:
    if num != 0:
      primes.append(num)
  return primes
```

You can test this function by calling find_primes(100), which will return the list [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97].

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

Ans. Searching problems in computer science involve finding a specific item or value in a data structure. This can include searching for an element in an array, a key in a dictionary, or a node in a graph. The goal is to efficiently locate the item you're looking for and return it, if it exists.

Example: Let's consider an example problem of searching for a specific element in an array of integers. Suppose we have an array of n integers and we want to check if a given element x is present in the array. One approach to solving this problem is the linear search algorithm, which involves iterating over the array one element at a time until the element is found or the end of the array is reached.

ii) String Processing

Ans. String processing is an important area of computer science that deals with the manipulation of strings, which are sequences of characters. There are various string processing problems in computer science, and they can be categorized based on the type of operation or task that needs to be performed on the strings.

Example: One example problem in string processing is the "Longest Common Subsequence" problem. Given two strings s1 and s2, the problem is to find the longest subsequence that is common to both s1 and s2. A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

iii) Geometric Problems

Ans. Geometric problems are problems in computer science that involve geometry, such as finding the intersection of two lines or the closest pair of points in a set. These problems can be solved using various algorithms and data structures, and they have many practical applications in fields such as computer graphics, robotics, and geographic information systems.

Example: One example of a geometric problem in computer science is the problem of finding the convex hull of a set of points in the plane. The convex hull is the smallest convex polygon that contains all of the points in the set. This problem has many practical applications, such as in computer graphics, where it is used to draw the outline of a shape, and in robotics, where it is used to plan the movement of a robot in a 2D environment.

iv) Numerical Problems

Ans. Numerical problems in computer science refer to computational problems that involve numerical calculations or operations. These problems can arise in various domains such as scientific simulations, data analysis, cryptography, and financial modeling.
Example: One example of a numerical problem in computer science is the calculation of the value of Pi using the Monte Carlo method. Pi is a mathematical constant that represents the ratio of the circumference of a circle to its diameter, and its value is approximately 3.14159. The Monte Carlo method is a statistical technique that uses random sampling to estimate the value of a parameter or solve a problem.

Q2. a) Using induction prove that, for all positive integers n,

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

Ans. To prove this statement using induction, we will need to show that it holds true for the base case of n = 1, and then show that if it holds true for n = k, it also holds true for n = k + 1.

Base case (n = 1):

If we substitute n = 1 into the formula on the right-hand side of the equation, we get:

1(1+1)(2(1)+1)/6 = 1(2)(3)/6 = 1

If we substitute n = 1 into the left-hand side of the equation, we get:

12 = 1

Therefore, both sides of the equation evaluate to 1 when n = 1, and the base case is true.

Inductive step (n = k + 1):

Assuming the formula holds true for n = k, we will now show that it also holds true for n = k + 1.

First, we can rewrite the formula for n = k as:

$1^2 + 2^2 + ... + k^2 = k(k+1)(2k+1)/6$

Next, we can substitute k+1 for n in the original formula:

$1^2 + 2^2 + ... + k^2 + (k+1)^2 = (k+1)((k+1)+1)(2(k+1)+1)/6$

Simplifying the right-hand side:

$(k+1)(k+2)(2k+3)/6$

Expanding the left-hand side:

$k(k+1)(2k+1)/6 + (k+1)^2$

Substituting the formula for n=k in the first term:

$(k+1)(2k+1)(k/6 + 1/2) + (k+1)^2$

Simplifying:

$(k+1)[(2k+1)(k/6 + 1/2) + (k+1)]$

$(k+1)[(2k^2 + 7k + 6)/6]$

$(k+1)(k+2)(2k+3)/6$

Therefore, we have shown that if the formula holds true for n = k, then it also holds true for n = k + 1. By the principle of mathematical induction, the formula is true for all positive integers 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. Purpose of asymptotic analysis: The purpose of asymptotic analysis in computer science is to analyze the performance of algorithms and data structures as the input size grows towards infinity. It helps us to understand how an algorithm or a data structure will perform as the problem size increases, without needing to examine specific input sizes.

Asymptotic analysis provides a high-level view of the efficiency of an algorithm or data structure by examining how the runtime or memory usage scales with respect to the input size. This analysis can be used to compare different algorithms or data structures and to select the one that is the most efficient for a given problem.

The most common measures used in asymptotic analysis are time complexity and space complexity. Time complexity measures how the runtime of an algorithm or operation scales with respect to the input size, while space complexity measures how the memory usage scales with respect to the input size.

Drawbacks of asymptotic analysis: Asymptotic analysis is a powerful tool for analyzing the performance of algorithms and data structures, but there are some drawbacks to this technique:

1. It does not consider the constant factors: Asymptotic analysis only considers the growth rate of the algorithm or data structure and does not take into account the constant factors. In some cases, the constant factors can have a significant impact on the actual running time of the algorithm, which may not be captured by the asymptotic analysis.
2. It may not reflect the practical performance: Asymptotic analysis is based on the assumption that the input size is large enough to make the leading term of the complexity dominate the other terms. However, in practice, the input size may not be large enough to achieve this condition, which means that the asymptotic analysis may not accurately reflect the actual performance of the algorithm or data structure.
3. It assumes worst-case scenario: Asymptotic analysis usually assumes the worst-case scenario when analyzing the performance of an algorithm or data structure. However, in practice, the input may not always be in the worst-case scenario, which means that the actual performance may be better than what is predicted by the asymptotic analysis.
4. It may not account for special cases: Asymptotic analysis assumes that the input is uniformly distributed, and it does not account for special cases where the input is not uniformly distributed. For example, if an algorithm performs well on most inputs but has poor performance on a few special cases, this may not be captured by the asymptotic analysis.

Big-O notation: Big-O notation is a mathematical notation used to describe the upper bound of the growth rate of a function or algorithm. It is commonly used in computer science to describe the time or space complexity of algorithms.

The notation is written as O(g(n)), where g(n) is a function that describes the upper bound of the growth rate of another function f(n) as n approaches infinity. In other words, O(g(n)) represents the set of functions that grow no faster than g(n) asymptotically.

Example: For example, let's say we have an algorithm that sorts an array of n elements using bubble sort. The worst-case time complexity of bubble sort is $O(n^2)$, which means that the running time of the algorithm will not exceed $c * n^2$ for some constant c and for large enough n.

To understand this, we can plot the growth rate of the bubble sort algorithm against the growth rate of the function $f(n) = n^2$. As n gets larger and larger, the growth rate of bubble sort approaches that of $f(n) = n^2$, but it never exceeds it. This is why we say that the time complexity of bubble sort is $O(n^2)$.

In practical terms, this means that as the size of the array to be sorted grows, the running time of the bubble sort algorithm will increase at most quadratically. It also means that if we compare bubble sort to other sorting algorithms with different time complexities, we can use Big-O notation to determine which algorithm is most efficient for a given problem.

Q3. a) Evaluate the polynomial $p(x)= 5x^5+4x^4-3x^3-2x^2+9x+11$ at x=3 using Horner's rule. Analyze 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 will follow these steps:

Step 1: Write the polynomial in Horner's form, which is:

$p(x) = ((5x + 4)x^3 - 3x^2 - 2)x + 9 + 11/1$

Step 2: Starting from the rightmost coefficient, 11, multiply it by the value of x, which is 3. Add the next coefficient, 9, to the result. This gives:

p(3) = 11 p(3) = 11/1 + 9 p(3) = 20

Step 3: Multiply the result from step 2 by the value of x, which is 3. Add the next coefficient, -2, to the result. This gives:

p(3) = 20 p(3) = -2 + (20 x 3) p(3) = 58

Step 4: Multiply the result from step 3 by the value of x, which is 3. Add the next coefficient, -3, to the result. This gives:

p(3) = 58 p(3) = -3 + (58 x 3) p(3) = 169

Step 5: Multiply the result from step 4 by the value of x, which is 3. Add the next coefficient, 4, to the result. This gives:

p(3) = 169 p(3) = 4 + (169 x 3) p(3) = 511

Step 6: Multiply the result from step 5 by the value of x, which is 3. Add the next coefficient, 5, to the result. This gives:

p(3) = 511 p(3) = 5 + (511 x 3) p(3) = 1548

Therefore, p(3) = 1548 when evaluated using Horner's rule.

Now let's analyze the computation using Horner's rule against the brute force method of polynomial evaluation.

The brute force method of polynomial evaluation involves directly computing each term of the polynomial and summing them together. For example, to evaluate p(3) using brute force, we would compute:

$p(3) = 5(3^5) + 4(3^4) - 3(3^3) - 2(3^2) + 9(3) + 11$

p(3) = 1215 + 324 - 81 - 18 + 27 + 11

p(3) = 1458

Comparing the two methods, we can see that Horner's rule requires fewer multiplications and additions than the brute force method. Specifically, Horner's rule requires 5 multiplications and 5 additions, while the brute force method requires 6 multiplications and 5 additions. Therefore, Horner's rule is generally more efficient for polynomial evaluation, especially for polynomials with high degree.

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 algorithm: Linear search is a simple algorithm for finding a particular value in an array or list. The algorithm works by sequentially checking each element of the array until a match is found or the end of the array is reached.

Here is the linear search algorithm:

```
linear_search(A, x):
  for i in range(len(A)):
    if A[i] == x:
      return i
  return -1
```

The linear_search function takes an array A and a value x as inputs. It then iterates through the elements of the array A one by one and checks if each element is equal to the value x. If a match is found, the index of the matching element is returned. If no match is found, the function returns -1.

The time complexity of linear search depends on the length of the array and the position of the target value. Here are the best case, worst case, and average case time complexities of linear search:

- **Best case**: The best case occurs when the target value is found at the first position of the array. In this case, the algorithm only needs to check one element, and the time complexity is O(1).
- **Worst case**: The worst case occurs when the target value is not in the array or is at the last position of the array. In this case, the algorithm needs to check every element of the array, and the time complexity is O(n), where n is the length of the array.
- **Average case**: The average case occurs when the target value is randomly distributed in the array. In this case, the algorithm needs to check, on average, half of the elements of the array. Therefore, the average case time complexity is O(n/2), which simplifies to O(n).

Let's now apply linear search to the following data:

13, 15, 2, 6, 14, 10, 8, 7, 3, 5, 19, 4, 17

- **Best case**: If we search for the value 13, which is the first element of the array, the algorithm will only need to check one element, and the time complexity will be O(1).
- **Worst case:** If we search for the value 20, which is not in the array, the algorithm will need to check every element of the array, and the time complexity will be O(13), where 13 is the length of the array.
- **Average case**: If we search for a randomly distributed value, such as 7, the algorithm will need to check, on average, half of the elements of the array, which is 6.5. Since the array has an odd number of elements, we can round up to 7. Therefore, the average case time complexity will be O(7).

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 this knapsack instance, we can use dynamic programming with a table of size (n+1) x (W+1). The entry in cell (i,j) of the table will contain the maximum value that can be obtained using items 1 to i, with a maximum capacity of j.

The recurrence relation for filling in the table is:

- if j < w[i], then cell (i,j) = cell (i-1,j) (we cannot add item i since it exceeds the capacity j)
- otherwise, cell (i,j) = max(cell(i-1,j), cell(i-1,j-w[i]) + p[i]) (we consider both the case where we do not take item i, and the case where we take it and add its value p[i] to the maximum value obtained using the remaining capacity j-w[i])

Using this recurrence relation, we can fill in the table starting from the top-left corner and moving towards the bottom-right corner. The final result will be in cell (n,W).

Here is the filled-in table for this instance:

	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

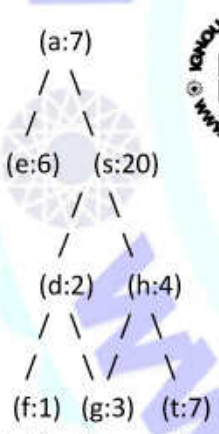
2 | 0	4	5	9	9	9	9	9	9	9	9	9	9	9	9	9
3 | 0	4	5	10	14	14	14	14	14	14	14	14	14	14	14	14
4 | 0	4	5	10	14	14	14	14	14	14	14	14	14	14	14	14
5 | 0	4	6	10	14	15	19	19	19	19	19	19	19	19	19	19
6 | 0	4	6	10	14	15	19	19	23	23	23	23	23	23	23	23
7 | 0	4	6	10	14	15	19	19	23	23	23	23	23	27	27	27

The optimal value is in cell (7,15), and it is 27. To determine which items were used to achieve this value, we can backtrack through the table from cell (7,15) towards cell (1,1), and whenever we encounter a cell where the value changes from the previous cell in the same row (i.e., we took item i), we add i to the list of chosen items. Using this method, we find that the optimal set of items consists of items 2, 4, 5, and 7, with a total weight of 15 and a total value of 27.

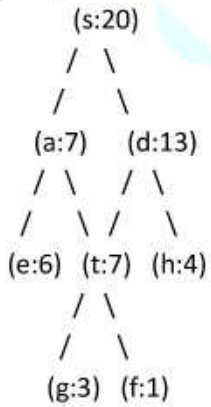
b) Make an optimal Huffman tree and design the Huffman code for the following set of frequencies: a:7, e:6, s:20, d:2, f:1, g:3, h:4, t:7.

Ans. To construct an optimal Huffman tree, we start by creating a leaf node for each symbol with its corresponding frequency. We then repeatedly merge the two nodes with the smallest frequencies until we are left with a single tree.
Here is the step-by-step process:

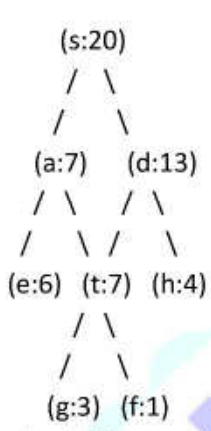
Step 1: Create leaf nodes for each symbol with its corresponding frequency.



Step 2: Find the two nodes with the smallest frequencies and merge them to create a new internal node with their sum as the frequency. Repeat until there is only one node left.



Step 3: Assign 0 to the left branch and 1 to the right branch of each internal node. The Huffman code for each symbol is obtained by traversing the tree from the root to the corresponding leaf node and concatenating the bits encountered along the way.



Therefore, the Huffman code for the given set of frequencies is:

a: 0
e: 100
s: 1
d: 1011
f: 10101
g: 10100
h: 10111
t: 1010

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: The recursive binary search algorithm is a way of searching for a specific element in a sorted array by dividing the array into halves and checking the middle element of each sub-array until the desired element is found.

Here is the recursive implementation of the binary search algorithm:

```
binarySearch(arr, left, right, x):
  if right >= left:
    mid = left + (right - left) // 2

    # If the element is present at the middle
    if arr[mid] == x:
      return mid

    # If the element is smaller than mid, then it can only be present in the left subarray
    elif arr[mid] > x:
      return binarySearch(arr, left, mid - 1, x)

    # Else the element can only be present in the right subarray
    else:
      return binarySearch(arr, mid + 1, right, x)
```

```
    else:
        # Element is not present in the array
        return -1
```

Program: Recursive binary search to find an element in a sorted array of 7 elements:

```
#include <stdio.h>

int binarySearch(int arr[], int left, int right, int target) {
    if (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid;
        }
        else if (arr[mid] > target) {
            return binarySearch(arr, left, mid - 1, target);
        }
        else {
            return binarySearch(arr, mid + 1, right, target);
        }
    }

    return -1; // target not found
}

int main() {
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    int target = 5;
    int n = sizeof(arr) / sizeof(arr[0]);

    int result = binarySearch(arr, 0, n - 1, target);

    if (result == -1) {
        printf("Element not found in array");
    }
    else {
        printf("Element found at index %d", result);
    }

    return 0;
}
```

Output:

Element found at index 4

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

Ans. Quick sort algorithm using Master Method: Quick Sort is a widely used sorting algorithm that has an average case time complexity of O(n log n), making it very efficient for large datasets. The Master Method can be used to analyze the time complexity of Quick Sort.

The Master Method states that if a recurrence relation of the form T(n) = aT(n/b) + f(n), where a >= 1 and b > 1 are constants and f(n) is a polynomial function, then:

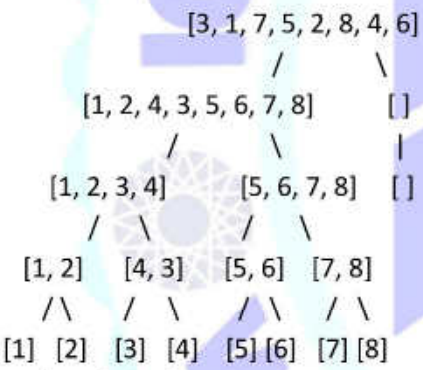
- If f(n) = O(n^(log_b a - ε)), where ε > 0 is a constant, then T(n) = Θ(n^(log_b a)).
- If f(n) = Θ(n^(log_b a)), then T(n) = Θ(n^(log_b a) log n).
- If f(n) = Ω(n^(log_b a + ε)), where ε > 0 is a constant, and if a * f(n/b) <= c * f(n) for some constant c < 1 and all sufficiently large n, then T(n) = Θ(f(n)).

For Quick Sort, we have:

- a = 2, since we are dividing the input into two parts
- b = 2, since we are dividing the input in half
- f(n) = O(n), since the partition step takes linear time

Using the Master Method, we can see that QuickSort has a time complexity of O(n log n). Specifically, we can see that log_b a = log_2 2 = 1, and f(n) = $O(n^1)$, so T(n) = Θ(n log n).

Recursion tree: Here is a visualization of the recursion tree for Quick Sort when sorting an array of size 8:



Each node in the tree represents a recursive call to Quick Sort, and the numbers in brackets represent the subarray being sorted. The tree has a depth of log_2 8 = 3, and at each level i, the size of the subarrays being sorted is n/2^i. The total number of nodes in the tree is 2^0 + 2^1 + 2^2 + ... + 2^d = 2^(d+1) - 1, where d is the depth of the tree. Therefore, the time complexity of Quick Sort can be expressed as O(n log n), where n is the size of the input array.

c) Write the algorithm for the divide and conquer strategy for Matrix multiplication. Also, analyze this algorithm.

Ans. Divide and conquer strategy for Matrix multiplication: The divide and conquer strategy for Matrix multiplication is an efficient algorithm that breaks down the matrix multiplication operation into smaller subproblems, solves them recursively, and combines the results to get the final product. The algorithm can be described as follows:

1. Divide each matrix A and B into four equal-sized submatrices of size n/2:
   A = [[A11, A12], [A21, A22]]
   B = [[B11, B12], [B21, B22]]
2. Compute the following 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)

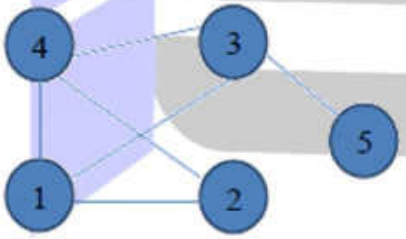
3. Compute the resulting submatrices C11, C12, C21, and C22:
   C11 = P5 + P4 - P2 + P6
   C12 = P1 + P2
   C21 = P3 + P4
   C22 = P5 + P1 - P3 - P7
4. Combine the resulting submatrices to get the final product:
   C = [[C11, C12], [C21, C22]]

Analyze this algorithm: The time complexity of this algorithm can be analyzed using the Master Method. In step 2, we are recursively computing 7 matrix products of size n/2, and in step 3, we are computing 4 submatrices of size n/2. Therefore, we can express the time complexity of this algorithm as follows:

T(n) = 7T(n/2) + O(n^2)

Using the Master Method, we can see that the time complexity of this algorithm is O(n^log_2 7), which is approximately O(n^2.81). This is an improvement over the naive algorithm for matrix multiplication, which has a time complexity of O(n^3). The divide and conquer algorithm is particularly useful for large matrices, where the savings in time become more significant.

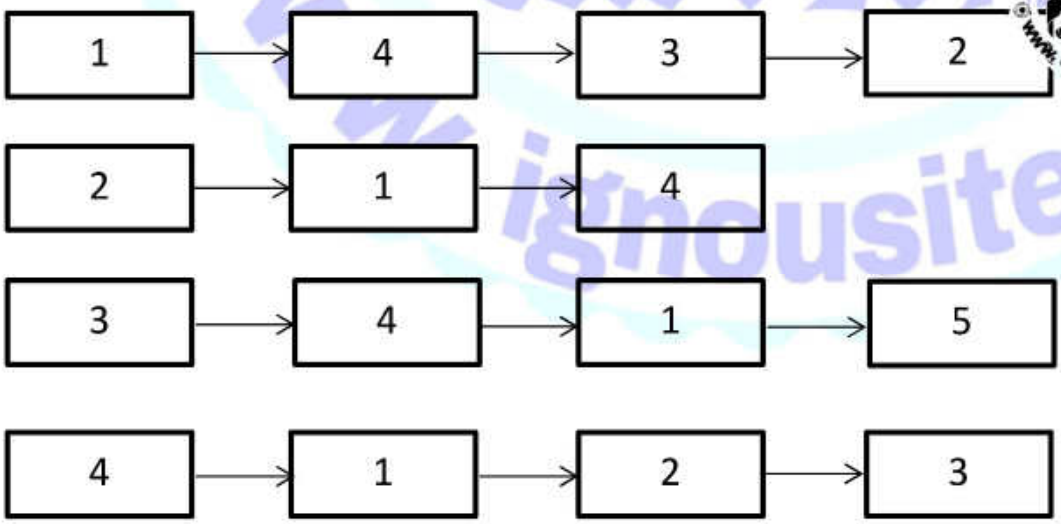
Q6. a) Write the adjacency list and draw adjacency graph for the graph given below.

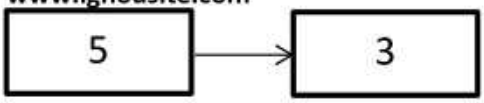


Ans.

The adjacency list:

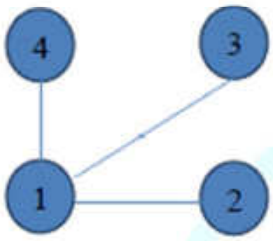
Vertex



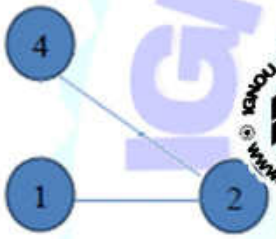


Adjacency graph:

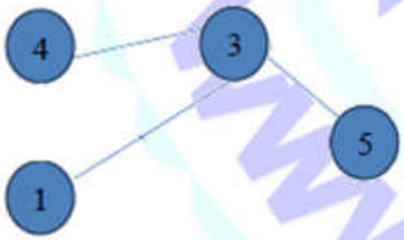
Vertex: 1



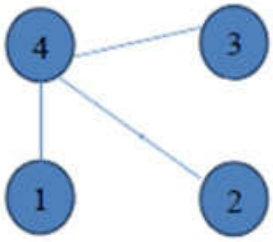
Vertex: 2



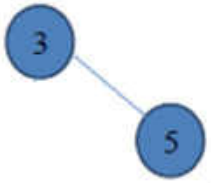
Vertex: 3



Vertex: 4



Vertex: 5



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

Ans: Algorithm of Topological sorting: Topological sorting is an algorithm used to sort a directed acyclic graph (DAG) in a linear ordering, such that for every directed edge (u, v), vertex u comes before vertex v in the ordering. In other words, it finds an order in which all nodes can be processed, without violating any dependencies.

The algorithm works as follows:

1. Find a vertex with no incoming edges, i.e., a vertex with in-degree 0. If there are multiple such vertices, choose any one.
2. Add the vertex to the sorted list and remove it from the graph.
3. Decrease the in-degree of all vertices adjacent to the vertex that was just removed, since their dependency has been resolved.
4. Repeat steps 1-3 until no vertices remain in the graph.

The algorithm can be implemented using a queue to keep track of the vertices with in-degree 0. Initially, all vertices with in-degree 0 are added to the queue. Then, the algorithm repeatedly dequeues a vertex, adds it to the sorted list, and updates the in-degree of its neighbors. If any of the neighbors now have in-degree 0, they are added to the queue.

Time complexity for topological sorting: The time complexity of Topological sorting is O(V + E), where V is the number of vertices and E is the number of edges in the graph. This is because we visit each vertex and edge exactly once. However, the algorithm can only be applied to DAGs, since it relies on the fact that there is at least one vertex with in-degree 0. If the graph has cycles, there will be no such vertex, and the algorithm will not work.

Overall, Topological sorting is a useful algorithm for dependency resolution in tasks such as scheduling, project management, and software engineering, where tasks need to be completed in a specific order without violating dependencies.

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. Working of Prim's algorithm for finding the minimum cost spanning tree: 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

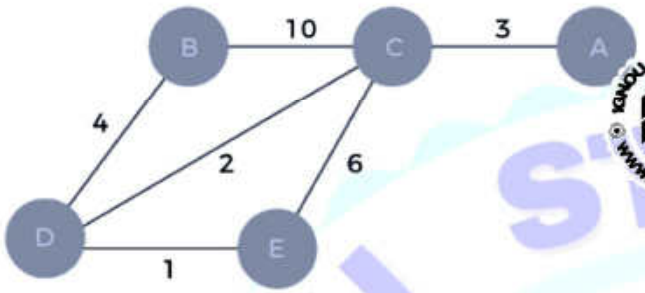
Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -

- 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.

Example:

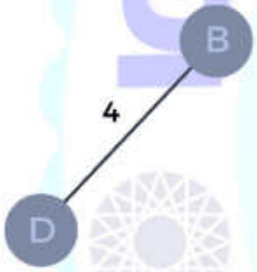
The working of prim's algorithm using an example. It will be easier to understand the prim's algorithm using an example.

Suppose, a weighted graph is -

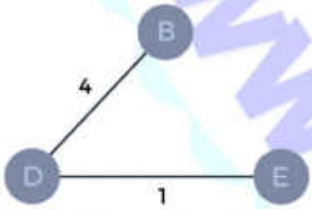


Step 1 - First, we have to choose a vertex from the above graph. Let's choose B.

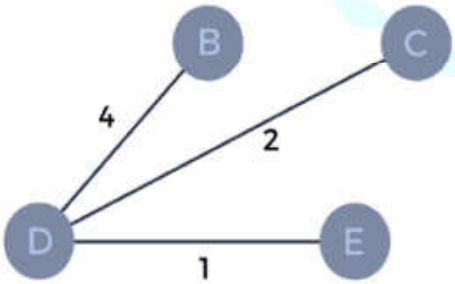
Step 2 - Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.



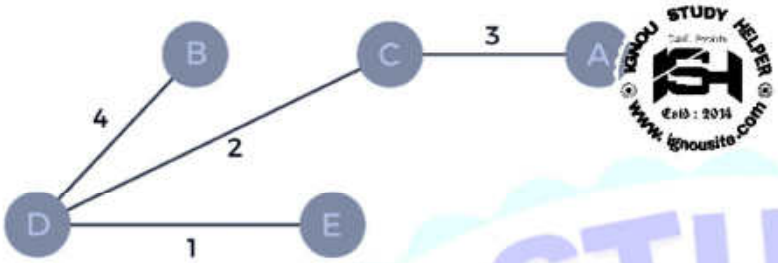
Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



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.

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 -

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. Working of Bellman-Ford algorithm for finding the shortest path: Dynamic Programming is used in the Bellman-Ford algorithm. It begins with a starting vertex and calculates the distances between other vertices that a single edge can reach. It then searches for a path with two edges, and so on. The Bellman-Ford algorithm uses the bottom-up approach.

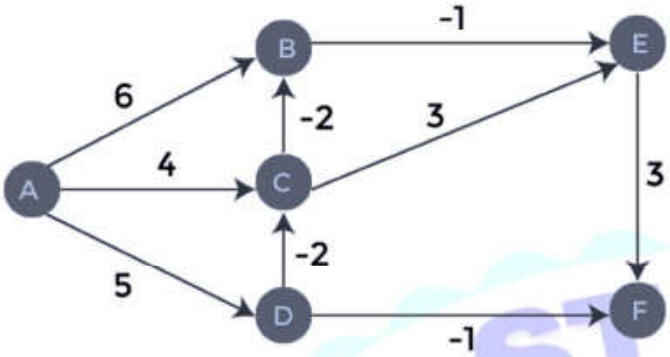
The Bellman-Ford algorithm works by grossly underestimating the length of the path from the starting vertex to all other vertices. The algorithm then iteratively relaxes those estimates by discovering new ways that are shorter than the previously overestimated paths.

Algorithm:

- The outer loop traverses from 0 : n-1.
- Loop over all edges, check if the next node distance > current node distance + edge weight, in this case update the next node distance to "current node distance + edge weight".

Example:

Consider the below graph:

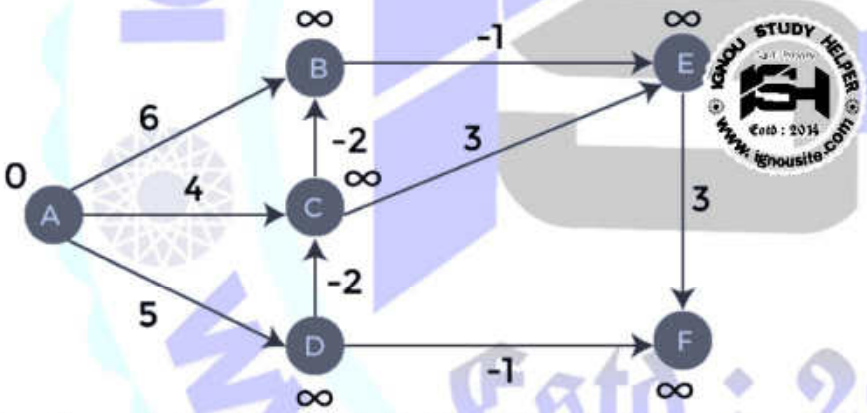


As we can observe in the above graph that some of the weights are negative. The above graph contains 6 vertices so we will go on relaxing till the 5 vertices. Here, we will relax all the edges 5 times. The loop will iterate 5 times to get the correct answer. If the loop is iterated more than 5 times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

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

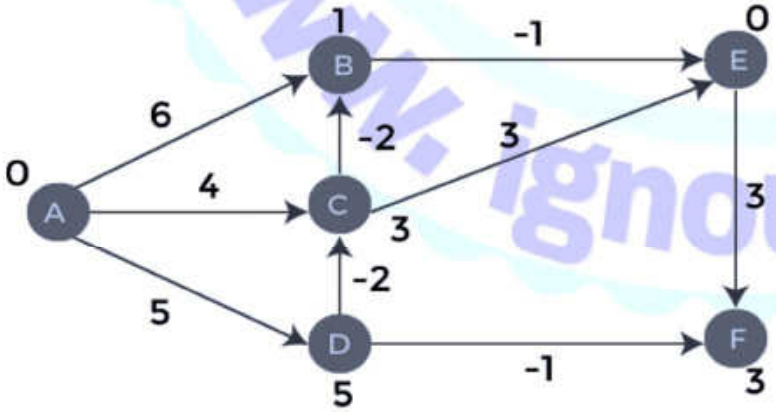
The next edge is (C, E). Since (3 + 3) equals to 6 which is greater than 5 so there would be no updation in the vertex E.

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

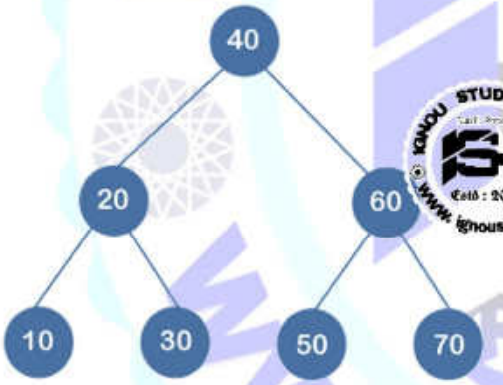
Q8. a) Explain the process of creating a optimal binary search with the help of an example.

Ans. Optional Binary search tree: Optional binary search tree, the nodes in the left subtree have lesser value than the root node and the nodes in the right subtree have greater value than the root node.

We know the key values of each node in the tree, and we also know the frequencies of each node in terms of searching means how much time is required to search a node. The frequency and key-value determine the overall cost of searching a node. The cost of searching is a very important factor in various applications. The overall cost of searching a node should be less. The time required to search a node in BST is more than the balanced binary search tree as a balanced binary search tree contains a lesser number of levels than the BST. There is one way that can reduce the cost of a binary search tree is known as an optimal binary search tree.

Example:

If the keys are 10, 20, 30, 40, 50, 60, 70



In the above tree, all the nodes on the left subtree are smaller than the value of the root node, and all the nodes on the right subtree are larger than the value of the root node. The maximum time required to search a node is equal to the minimum height of the tree, equal to logn.

Now we will see how many binary search trees can be made from the given number of keys.

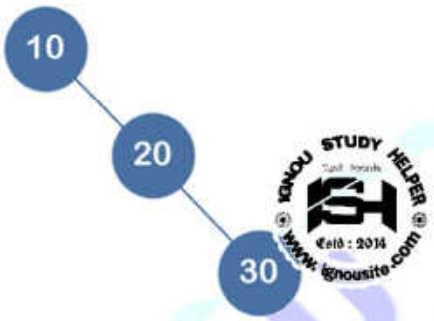
For example: 10, 20, 30 are the keys, and the following are the binary search trees that can be made out from these keys.

The Formula for calculating the number of trees:

$$\frac{{}^{2n}C_n}{n+1}$$

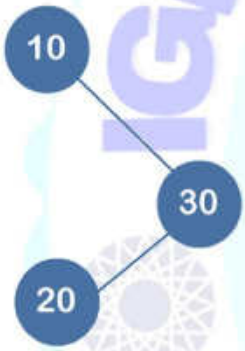
When we use the above formula, then it is found that total 5 number of trees can be created.

The cost required for searching an element depends on the comparisons to be made to search an element. Now, we will calculate the average cost of time of the above binary search trees.



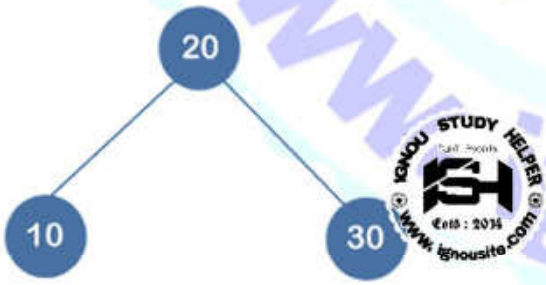
In the above tree, total number of 3 comparisons can be made. The average number of comparisons can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



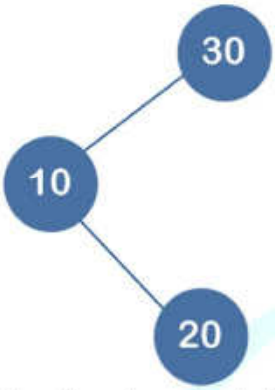
In the above tree, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



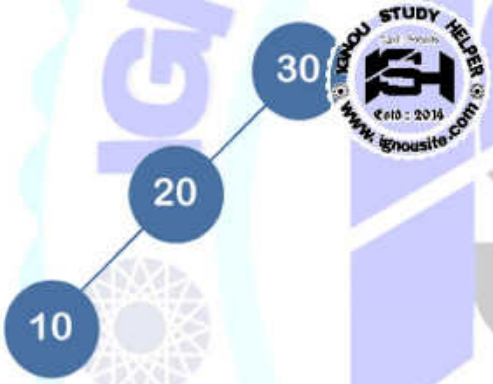
In the above tree, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+2}{3} = 5/3$$



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$

In the third case, the number of comparisons is less because the height of the tree is less, so it's a balanced binary search tree.

Till now, we read about the height-balanced binary search tree. To find the optimal binary search tree, we will determine the frequency of searching a key.

Let's assume that frequencies associated with the keys 10, 20, 30 are 3, 2, 5.

The above trees have different frequencies. The tree with the lowest frequency would be considered the optimal binary search tree. The tree with the frequency 17 is the lowest, so it would be considered as the optimal binary search tree.

b) Find an optimal parenthesizing of a matrix-chain product whose sequence 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 a matrix chain product, we can use dynamic programming to minimize the number of scalar multiplications required to compute the product. Let's first calculate the number of scalar multiplications required for each possible parenthesization of the chain. We'll use the following formula to calculate the number of scalar multiplications for a given parenthesization:

m[i,j] = min(m[i,k] + m[k+1,j] + p[i-1]*p[k]*p[j])

where m[i,j] is the minimum number of scalar multiplications required to compute the matrix product A[i]*A[i+1]*...*A[j], and p is the array of dimensions of the matrices in the chain.

Here's the table of values for the given matrix dimensions:

	A1	A2	A3	A4	A5
A1	0	735	1575	1312	3300
A2		0	1050	315	735	2520
A3			0	525	2625	2250
A4				0	1125	1260
A5					0	2700

From the above table, we can see that the minimum number of scalar multiplications required to compute the product is 3300. We can find the optimal parenthesization by tracing back the path that leads to this minimum value. Here's the optimal parenthesization:

((A1(A2A3))(A4A5))

This parenthesization will result in the minimum number of scalar multiplications required to compute the matrix chain product.

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. 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.

Programe:

```
#include<stdio.h>
#include<string.h>
int main (){
  char txt[80], pat[80];
```

```
  int q;
  printf ("Enter the container string \n");
  scanf ("%s", &txt);
  printf ("Enter the pattern to be searched \n");
  scanf ("%s", &pat);
  int d = 256;
  printf ("Enter a prime number \n");
  scanf ("%d", &q);
  int M = strlen (pat);
  int N = strlen (txt);
  int i, j;
  int p = 0;
  int t = 0;
  int h = 1;
  for (i = 0; i < M - 1; i++)
    h = (h * d) % q;
  for (i = 0; i < M; i++){
    p = (d * p + pat[i]) % q;
    t = (d * t + txt[i]) % q;
  }
  for (i = 0; i <= N - M; i++){
    if (p == t){
      for (j = 0; j < M; j++){
        if (txt[i + j] != pat[j])
        break;
      }
      if (j == M)
        printf ("Pattern found at index %d \n", i);
    }
    if (i < N - M){
      t = (d * (t - txt[i] * h) + txt[i + M]) % q;
      if (t < 0)
        t = (t + q);
    }
  }
  return 0;
}
```

Output:

```
Enter the container string
attainthtenbetan
Enter the pattern to be searched
ten
Enter a prime number
3
Pattern found at index 8
```

Steps:

Steps involved in using the Rabin-Karp algorithm to find the pattern "ten" in the text "attainthtenbetan":

Step 1: Preprocessing In this step, we calculate the hash value of the pattern string "ten" and store it in a variable. We will also calculate the hash values of all substrings of length 3 in the text and store them in a list. We will use a simple hash function that calculates the hash value of a string as the sum of the ASCII codes of its characters.

The hash value of "ten" is calculated as follows:

t = 116 (ASCII code for 't')

e = 101 (ASCII code for 'e')

n = 110 (ASCII code for 'n')

Hash value = t + e + n = 327

The hash values of all substrings of length 3 in the text "attainthtenbetan" are:

att = 291
tta = 315
tai = 307
ain = 296
int = 321
nth = 330
tht = 327
hte = 315
ten = 327
enb = 311
nbe = 310
bet = 316
eta = 309
tan = 324

Step 2: Matching In this step, we compare the hash value of the pattern string "ten" with the hash values of all substrings of length 3 in the text. If a hash value matches, we compare the corresponding substrings character by character to confirm the match.

In our case, the hash value of "ten" (327) matches with the hash value of the substring "tht" (327). We confirm the match by comparing the characters of the two strings:

ten = 116 101 110 tht = 116 104 116

The substrings do not match, so we continue the matching process. The next substring with a matching hash value is "ten" (327) again, this time at position 8 in the text. We confirm the match:

ten = 116 101 110 ten = 116 101 110

The substrings match, so we have found the pattern "ten" in the text.

Step 3: Output We output the position of the first character of the pattern in the text. In our case, the first character of the pattern "ten" occurs at position 8 in the text "attainthtenbetan", so we output 8 as the result.

Therefore, the Rabin Karp algorithm found the pattern "ten" in the text "attainthtenbetan" at position 8.

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

Ans. Knuth Morris Pratt (KMP) algorithm and Naïve string matching algorithm are two different algorithms used for pattern matching in strings. Here are the main differences between the two algorithms:

1. **Time complexity**: Naïve string matching algorithm has a time complexity of O(mn) where m is the length of the pattern and n is the length of the text. In the worst-case scenario, it may need to compare all characters in the text for every position of the pattern. On the other hand, the KMP algorithm has a time complexity of O(n+m) in the worst-case scenario, where n is the length of the text and m is the length of the pattern. The KMP algorithm achieves this better time complexity by skipping some comparisons in the text, which have already been made before.
2. **Preprocessing**: The naïve string matching algorithm does not require any preprocessing of the pattern or the text before matching. It starts matching the pattern from the first character of the text and slides it by one character for each unsuccessful match. The KMP algorithm requires preprocessing of the pattern to calculate the longest proper prefix that matches a proper suffix of the pattern. This information is then used to avoid unnecessary comparisons in the text.
3. **Comparison**: The naïve string matching algorithm compares every character of the pattern to the corresponding character in the text at every position of the pattern. The KMP algorithm compares only those characters in the text that have not been matched previously.
4. **Space complexity**: Naïve string matching algorithm has a space complexity of O(1) since it does not require any additional memory. The KMP algorithm requires O(m) additional memory to store the longest proper prefix that matches a proper suffix of the pattern.
5. **Worst-case scenario**: In the worst-case scenario, the naïve string matching algorithm takes O(mn) time complexity, which happens when the pattern has a lot of repeated characters, and the text has similar characters. The KMP algorithm performs well in this case since it takes O(n+m) time complexity.

Overall, the KMP algorithm is more efficient than the naïve string matching algorithm for most cases, especially when the pattern has a lot of repeated characters. However, the naïve algorithm is easier to understand and implement.

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

(i) Optimization and Decision Problems

Ans. Optimization problems and decision problems are two different types of computational problems in computer science and mathematics. Here are the main differences between the two:

1. Goal: In optimization problems, the goal is to find the best solution among many possible solutions. The solution could be the maximum or minimum value of an objective function or the best configuration of a system. In contrast, decision problems are concerned with finding a solution that satisfies a certain condition or criterion. The solution could be a yes or no answer, or it could be a specific value.
2. Input: Optimization problems usually have many possible input values, and the goal is to find the best output value for each input. In contrast, decision problems have a fixed input and require the determination of a certain property or feature of that input.
3. Solution: In optimization problems, there may be multiple optimal solutions, and the goal is to find the best one. In contrast, decision problems have a unique solution that can be either true or false.
4. Complexity: Optimization problems are usually harder to solve than decision problems, as they require finding the best solution among a large set of possible solutions. Decision problems are usually easier to solve as they require only determining the existence of a solution that satisfies a certain criterion.

Example:

An example of an optimization problem is the traveling salesman problem, which involves finding the shortest possible route that visits a set of cities and returns to the starting city. The goal is to minimize the total distance traveled while visiting all cities. There are many possible routes, and the optimization problem is to find the best one.

An example of a decision problem is the graph coloring problem, which involves determining if a given graph can be colored using a certain number of colors. The goal is to determine if there exists a valid coloring for the graph. The solution is either true (a valid coloring exists) or false (no valid coloring exists).

(ii) P and NP problems

Ans. P and NP are two classes of computational problems in computer science. P problems are those that can be solved in polynomial time, while NP problems are those for which a solution can be verified in polynomial time. Here's an example that illustrates the difference between the two:

Consider the problem of finding the shortest path between two cities in a graph. This problem is in the class P because it can be solved using algorithms such as Dijkstra's algorithm or the A* algorithm in polynomial time, i.e., the time it takes to solve the problem increases at most polynomially as the size of the problem increases.

Now consider the problem of finding the Hamiltonian cycle in a graph, which is an NP-complete problem. A Hamiltonian cycle is a cycle in a graph that visits each vertex exactly once. The problem is to determine whether a Hamiltonian cycle exists in a given graph. The brute-force approach to solving this problem involves checking all possible cycles in the graph, which can take exponential time. However, if someone claims to have found a Hamiltonian cycle in the graph, we can easily verify it in polynomial time by checking that it visits each vertex exactly once. Therefore, this problem is in the class NP.

The main difference between P and NP problems is that P problems can be solved efficiently, whereas NP problems are believed to be difficult to solve in polynomial time. However, it's important to note that just because a problem is in NP doesn't mean it's impossible to solve in polynomial time. Some NP problems may have polynomial time algorithms that have not been discovered yet, or they may have efficient approximate solutions.

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

Ans. NP Hard and NP complete problems: NP-hard and NP-complete are two classes of problems in computer science, particularly in the field of computational complexity theory.

A problem is said to be NP-hard (Non-deterministic Polynomial-time hard) if any problem in NP (Non-deterministic Polynomial-time) can be reduced to it in polynomial time. This means that an NP-hard problem is at least as hard as any problem in NP. NP-hard problems are generally considered difficult to solve, and there is no known polynomial-time algorithm that can solve all NP-hard problems.

A problem is said to be NP-complete if it is both NP-hard and in NP. This means that an NP-complete problem is one for which a solution can be verified in polynomial time, but finding a solution itself may require exponential time. NP-complete problems are considered among the most difficult problems in computer science and are of significant interest in theoretical computer science.

Any one problem of each:

One example of an NP-hard problem is the Traveling Salesman Problem (TSP), which asks to find the shortest possible route that visits a set of given cities and returns to the starting city. The problem is NP-hard because any problem in NP can be reduced to it in polynomial time.

One example of an NP-complete problem is the Boolean Satisfiability Problem (SAT), which asks whether a given Boolean formula can be satisfied by assigning true or false values to its variables. SAT is NP-complete because it is in NP and any problem in NP can be reduced to it in polynomial time.

Both TSP and SAT have important real-world applications and have been extensively studied in computer science and related fields.

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

Ans. Backtracking and Branch and Bound are two important techniques used in solving optimization problems. While both techniques involve a systematic search for the optimal solution, they differ in how they explore the search space.

Backtracking: Backtracking is a brute-force search technique that is used to find all the solutions to a problem by exploring all possible paths, but prunes the search tree when it determines that a partial solution cannot be completed to a valid solution. It involves building a tree structure of all possible solutions, exploring each path until a solution is found or a dead end is reached. When a dead end is reached, the algorithm backtracks to the last decision point and tries another path. Backtracking is often used when the problem size is small or the number of solutions is limited.

Example: For example, consider the problem of finding all possible permutations of a set of numbers {1, 2, 3}. The algorithm starts by choosing the first element (1) and then recursively generating all possible permutations of the remaining elements ({2, 3}). Once all the permutations of {2, 3} have been generated, the algorithm backtracks to the previous level and tries another permutation. This process continues until all permutations have been generated.

Branch and Bound: Branch and Bound is a more sophisticated optimization technique that is used to solve problems where the search space is too large to explore all possible solutions. It works by dividing the search space into smaller subspaces and solving each subspace separately, using an upper bound to prune subspaces that cannot contain a better solution than the current best solution. Branch and Bound is particularly effective when the solution space has a natural hierarchy, as in the case of an optimization problem with constraints.

Example: For example, consider the knapsack problem, where we have a set of items with weights and values, and we want to pack the knapsack with a maximum value while not exceeding a certain weight limit. The Branch and Bound algorithm starts by dividing the solution space into two subspaces, one where we include the first item and one where we exclude it. It then computes an upper bound for each subspace, which is the maximum possible value we can obtain if we choose the remaining items optimally. The algorithm then recursively divides each subspace into two more subspaces and computes upper bounds for them. The process continues until all subspaces have been explored or pruned based on their upper bounds. At the end, the best solution is the one with the highest value that meets the weight limit.