18CSC204J – DESIGN AND ANALYSIS OF ALGORITHMS LABORATORY

The objective of this lab is to teach students various algorithm design techniques and to explain them the importance of design techniques in context of applying those techniques in similar problems. Students will gain practical knowledge by writing and executing programs in C and C++ using various design techniques like Divide and Conquer, Greedy, Dynamic Programming, Backtracking, Branch and Bound and Randomization.

OUTCOMES:

Upon the completion of Design and Analysis of Algorithms practical course, the student will be able to:

1. **Design** and analyze the time and space efficiency of the various algorithm design techniques.
2. **Identity** the appropriate algorithm design technique for given problem.
3. **Understand** the applications of algorithm design techniques.
4. **Choose** the appropriate algorithm design method for a specified application.
5. **Understand** which algorithm design technique to use in different scenarios.
6. **Understand** and apply fundamental algorithmic problems including Tree traversals, Graph traversals.
7. **Compare** different implementations of algorithm design technique and to recognize the advantages and disadvantages of them.
8. **Compare** between different algorithm design techniques. Pick an appropriate one for a design situation.

COURSE LEARNING RATIONALE

The purpose of learning this course is to:

CLR-1: Design efficient algorithms in solving complex real time problems

CLR-2: Analyze various algorithm design techniques to solve real time problems in polynomial time

CLR-3: Utilize various approaches to solve greedy and dynamic algorithms

CLR-4: Utilize back tracking and branch and bound paradigms to solve exponential time problems

CLR-5: Analyze the need of approximation and randomization algorithms, utilize the importance Non polynomial algorithms

CLR-6: Construct algorithms that are efficient in space and time complexities

COURSE LEARNING OUTCOMES

At the end of this course, learners will be able to:

CLO-1: Apply efficient algorithms to reduce space and time complexity of both recurrent and non-recurrent relations

CLO-2 : Solve problems using divide and conquer approaches

CLO-3 : Apply greedy and dynamic programming types techniques to solve polynomial time problems

CLO-4 : Create exponential problems using backtracking and branch and bound approaches

CLO-5 : Interpret various approximation algorithms and interpret solutions to evaluate P type, NP Type, NPC, NP Hard problems

CLO-6 : Create algorithms that are efficient in space and time complexities by using divide conquer, greedy, backtracking technique

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

SCHOOL OF COMPUTING

18CSC204J – DESIGN AND ANALYSIS OF ALGORITHMS

Week Wise Lab Schedule

2022-2023 Even Semester

S.No	**Week**	**Exercise to be Completed**
1	Week 1	Linear search
2	Week 2	Bubble sort, Insertion sort
3	Week 3	Merge sort
4	Week 4	Quick sort, binary Search
5	Week 5	Strassen Matrix multiplication, Maximum Subarray Sum
6	Week 6	Finding maximum and minimum in an array, Convex hull problem
7	Week 7	Huffman coding, Knapsack using Greedy
8	Week 8	Tree traversals, MST using Kruskal's algorithm
9	Week 9	Longest common subsequence
10	Week 10	N queen's problem
11	Week 11	Travelling salesman problem
12	Week 12	BFS and DFS implementation with array
13	Week 13	Randomized quick sort
14	Week 14	String matching algorithms
15	Week 15	Implement any problem statement with two different algorithm design strategy that you have learnt. Compare and contrast with its time complexity analysis. Submit the same as report.

WEEK 1	LINEAR SEARCH

Searching:

Searching is the algorithmic process of finding a particular item in a collection of items. If the value is present in the array, then the searching is said to be successful and the searching process gives the location of that value in the array. However, if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful.

There are two popular methods for searching an array elements: linear search and binary search.

Linear Search:

Linear search, also called as sequential search, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found. Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted).

Pseudocode:

```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3: Repeat Step 4 while I<=N Step 4:
            IF A[I] = VAL
                SET POS = I PRINT POS
                Go to Step 6
            [END OF IF]
            SET I = I + 1
        [END OF LOOP]
Step 5: IF POS = –1
        PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

WEEK 2	BUBBLE SORT, INSERTION SORT

Sorting:

Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending. That is, if A is an array, then the elements of A are arranged in a sorted order (ascending order) in such a way that A [0] < A [1] < A [2] ……. < A [N].

For example, if we have an array that is declared and initialized as

int A[] = {21, 34, 11, 9, 1, 0, 22};

Then the sorted array (ascending order) can be given as:

A[] = {0, 1, 9, 11, 21, 22, 34;

2.a.Bubble sort

Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (in case of arranging elements in ascending order). In bubble sorting, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts.

Pseudocode:

```
      BUBBLE_SORT(A, N)


      Step 1: Repeat Step 2 For 1 = to N-1
      Step 2: Repeat For J = to N -  I
      Step 3:           IF A[J] > A[J + 1]
                              SWAP A[J] and A[J+1] [
                   END OF INNER LOOP]
                [END OF OUTER LOOP]
     Step 4: EXIT
```

2.b. Insertion sort

Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time. We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards while playing bridge. The main idea behind insertion sort is that it inserts each item into its proper place in the final list. To save memory, most implementations of the insertion sort algorithm work by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place.

Pseudocode:

```
      INSERTION-SORT (ARR, N)


      Step 1: Repeat Steps 2 to 5 for K = 1 to N – 1
      Step 2: SET TEMP = ARR[K]
```

```
Step 3: SET J = K – 1
Step 4: Repeat while TEMP <= ARR[J]
                    SET ARR[J + 1] = ARR[J]
                    SET J = J – 1
             [END OF INNER LOOP]
 Step 5:                SET ARR[J + 1] = TEMP
         [END OF LOOP]
Step 6: EXIT
```

WEEK 3	MERGE SORT

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then it merges the two sorted halves. **The merge () function** is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

Pseudocode :

• Declare left variable to 0 and right variable to n-1

• Find mid by medium formula. mid = (left+right)/2

• Call merge sort on (left,mid)

• Call merge sort on (mid+1,rear)

• Continue till left is less than right

• Then call merge function to perform merge sort.

Algorithm:

```
Step 1: Start
Step 2: Declare an array and left, right, mid variable
Step 3: Perform merge function.
    mergesort(array,left,right)
    mergesort (array, left, right)
    if left > right
    return
    mid= (left+right)/2
    mergesort(array, left, mid)
    mergesort(array, mid+1, right)
    merge(array, left, mid, right)
Step 4: Stop
```

Program:

```
// C program for Merge Sort
#include <stdio.h>
#include <stdlib.h>

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l,
                    int m, int r)
{
        int i, j, k;
int n1 = m - l + 1;
int n2 = r - m;

// Create temp arrays
int L[n1], R[n2];

// Copy data to temp arrays
// L[] and R[]
for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

// Merge the temp arrays back
// into arr[l..r]
// Initial index of first subarray
i = 0;

// Initial index of second subarray
j = 0;

// Initial index of merged subarray
k = l;
while (i < n1 && j < n2)
{
        if (L[i] <= R[j])
        {
                arr[k] = L[i];
                i++;
        }
        else
        {
                arr[k] = R[j];
                j++;
        }
        k++;
}

// Copy the remaining elements
// of L[], if there are any
while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
}
```

```
        // Copy the remaining elements of
        // R[], if there are any
        while (j < n2)
        {
                arr[k] = R[j];
                j++;
                k++;
        }
}

// l is for left index and r is
// right index of the sub-array
// of arr to be sorted
void mergeSort(int arr[],
                        int l, int r)
{
        if (l < r)
        {
                // Same as (l+r)/2, but avoids
                // overflow for large l and h
                int m = l + (r - l) / 2;

                // Sort first and second halves
                mergeSort(arr, l, m);
                mergeSort(arr, m + 1, r);

                merge(arr, l, m, r);
        }
}

// UTILITY FUNCTIONS
// Function to print an array
void printArray(int A[], int size)
{
        int i;
        for (i = 0; i < size; i++)
                printf("%d ", A[i]);
        printf("\n");
}

// Driver code
int main()
{
        int arr[] = {12, 11, 13, 5, 6, 7};
        int arr_size = sizeof(arr) / sizeof(arr[0]);
        printf("Given array is \n");
        printArray(arr, arr_size);

        mergeSort(arr, 0, arr_size - 1);

        printf("\nSorted array is \n");
        printArray(arr, arr_size);
        return 0;
}
```

WEEK 4	QUICK SORT, BINARY SEARCH

4.a. Aim:

Implement a divide and conquer algorithm to perform Quicksort in an array

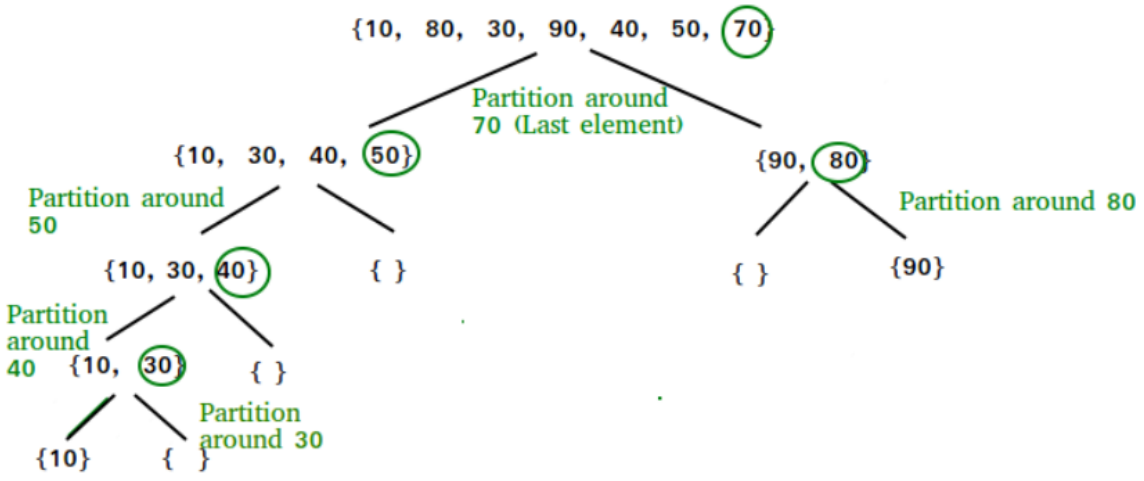
Procedure:

QuickSort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot.
- Pick median as the pivot.

The key process in **quickSort** is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Example:



Algorithm:

```
/* low  –> Starting index,  high  –> Ending index */
quickSort(arr[], low, high) {
   if (low < high) {
      /* pi is partitioning index, arr[pi] is now at right place */
```

```
        pi = partition(arr, low, high);

        quickSort(arr, low, pi – 1);  // Before pi

        quickSort(arr, pi + 1, high); // After pi

    }

}
```

/* This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot */

```
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low – 1)  // Index of smaller element and indicates the
    // right position of pivot found so far

    for (j = low; j <= high- 1; j++){

        // If current element is smaller than the pivot
        if (arr[j] < pivot){
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Code:

```
/* C++ implementation of QuickSort */
#include <bits/stdc++.h>
using namespace std;

// A utility function to swap two elements
void swap(int* a, int* b)
{
        int t = *a;
        *a = *b;
        *b = t;
}
```

```
/* This function takes last element as pivot, places the pivot element at its correct position
in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater
elements to right of pivot */
int partition(int arr[], int low, int high)
{
        int pivot = arr[high]; // pivot
        int i = (low- 1);
// Index of smaller element and indicate the right position of pivot found so far

        for (int j = low; j <= high - 1; j++) {
                // If current element is smaller than the pivot
                if (arr[j] < pivot) {
                        i++; // increment index of smaller element
                        swap(&arr[i], &arr[j]);
                }
        }
        swap(&arr[i + 1], &arr[high]);
        return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
        if (low < high) {
                /* pi is partitioning index, arr[p] is now
                at right place */
                int pi = partition(arr, low, high);

                // Separately sort elements before
                // partition and after partition
                quickSort(arr, low, pi - 1);
                quickSort(arr, pi + 1, high);
        }
}


/* Function to print an array */
```

```
void printArray(int arr[], int size)
{
        int i;
        for (i = 0; i < size; i++)
                cout << arr[i] << " ";
        cout << endl;
}

// Driver Code
int main()
{
        int arr[] = { 10, 7, 8, 9, 1, 5 };
        int n = sizeof(arr) / sizeof(arr[0]);
        quickSort(arr, 0, n - 1);
        cout << "Sorted array: \n";
        printArray(arr, n);
        return 0;
}
```

<u>Sample Input and output:</u>

```
Sorted array:
1 5 7 8 9 10
```

<u>Algorithm Analysis:</u>

Time taken by QuickSort, in general, can be written as follows.

$T(n) = T(k) + T(n-k-1) + \quad (n)$

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements that are smaller than the pivot.
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

Worst Case:

The worst case occurs when the partition process always picks the greatest or smallest element as the pivot. If we consider the above partition strategy where the last element is always picked as a pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for the worst case.
$T(n) = T(0) + T(n-1) + \quad (n)$ which is equivalent to $T(n) = T(n-1) + \quad (n)$

The solution to the above recurrence is $O(n^2)$.
Best Case:

The best case occurs when the partition process always picks the middle element as the pivot. The following is recurrence for the best case.

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

The solution for the above recurrence is O(nLogn).

4.b.Aim:

Implement a divide and conquer algorithm to perform Binary search in an array

Procedure:

Given a sorted array **arr[]** of **n** elements, write a function to search a given element **x** in **arr[]** and return the index of x in the array. Consider array is 0 base index.

Algorithm:

Binary Search Algorithm can be implemented in the following two ways

1. Iterative Method
2. Recursive Method

1. Iteration Method

```
binarySearch(arr, x, low, high)
    repeat till low = high
           mid = (low + high)/2
               if (x == arr[mid])
               return mid
               else if (x > arr[mid]) // x is on the right
                   low = mid + 1
               else              // x is on the left side
                   high = mid – 1
```

2. Recursive Method (The recursive method follows the divide and conquer approach)

```
binarySearch(arr, x, low, high)
       if low > high
           return False
```

```
    else
        mid = (low + high) / 2
            if x == arr[mid]
            return mid

        else if x > arr[mid]   // x is on the right side
            return binarySearch(arr, x, mid + 1, high)

        else                   // x is on the left side
            return binarySearch(arr, x, low, mid - 1)
```

Code:

```
// C++ program to implement recursive Binary Search
#include <bits/stdc++.h>
using namespace std
// A recursive binary search function. It returns
// location of x in given array arr[l..r] is present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
            int mid = l + (r - l) / 2;
            // If the element is present at the middle
            // itself
            if (arr[mid] == x)
                    return mid;
```

```
                // If element is smaller than mid, then

                // it can only be present in left subarray

                if (arr[mid] > x)

                        return binarySearch(arr, l, mid - 1, x);

                // Else the element can only be present in right subarray

                return binarySearch(arr, mid + 1, r, x);

        }

        // We reach here when element is not present in array

        return -1;

    }

    int main(void)

    {

        int arr[] = { 2, 3, 4, 10, 40 };

        int x = 10;

        int n = sizeof(arr) / sizeof(arr[0]);

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

        (result == -1)

                ? cout << "Element is not present in array"

                : cout << "Element is present at index " << result;

        return 0;

    }
```

<u>Sample input and output:</u>

Input: arr[] = {10, 20, 30, 50, 60, 80, 110, 130, 140, 170}, x = 110
Output: 6
Explanation: Element x is present at index 6.
Input: arr[] = {10, 20, 30, 40, 60, 110, 120, 130, 170}, x = 175
Output: -1

Explanation: Element x is not present in arr[].

Algorithm Analysis:

Time Complexity: O (log n)

WEEK 5	**STRASSEN MATRIX MULTIPLICATION, MAXIMUM SUBARRAY SUM**

5.a.Aim:

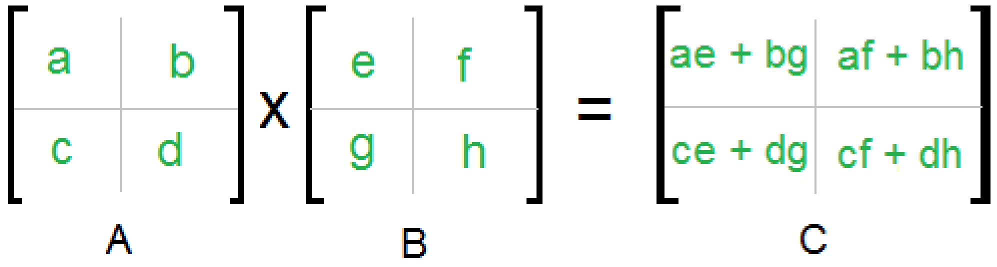
To implement a divide and conquer algorithm to perform strassen matrix multiplication.

Procedure:

Given two square matrices A and B of size n x n each, find their multiplication matrix.

Following is simple Divide and Conquer method to multiply two square matrices.

1. Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.
2. Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.



A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

Code:

```
#include <bits/stdc++.h>
using namespace std;

#define ROW_1 4
#define COL_1 4

#define ROW_2 4
#define COL_2 4

void print(string display, vector<vector<int> > matrix,
                int start_row, int start_column, int end_row,
                int end_column)
{
        cout << endl << display << " =>" << endl;
        for (int i = start_row; i <= end_row; i++) {
                for (int j = start_column; j <= end_column; j++) {
                        cout << setw(10);
                        cout << matrix[i][j];
                }
                cout << endl;
        }
```

```
		cout << endl;
		return;
	}

	void add_matrix(vector<vector<int> > matrix_A,
					vector<vector<int> > matrix_B,
					vector<vector<int> >& matrix_C,
					int split_index)
	{
		for (auto i = 0; i < split_index; i++)
			for (auto j = 0; j < split_index; j++)
				matrix_C[i][j]
					= matrix_A[i][j] + matrix_B[i][j];
	}

	vector<vector<int> >
	multiply_matrix(vector<vector<int> > matrix_A,
					vector<vector<int> > matrix_B)
	{
		int col_1 = matrix_A[0].size();
		int row_1 = matrix_A.size();
		int col_2 = matrix_B[0].size();
		int row_2 = matrix_B.size();

		if (col_1 != row_2) {
			cout << "\nError: The number of columns in Matrix "
					"A must be equal to the number of rows in "
					"Matrix B\n";
			return {};
		}

		vector<int> result_matrix_row(col_2, 0);
		vector<vector<int> > result_matrix(row_1,
		result_matrix_row);

		if (col_1 == 1)
			result_matrix[0][0]
				= matrix_A[0][0] * matrix_B[0][0];
		else {
			int split_index = col_1 / 2;

			vector<int> row_vector(split_index, 0);
			vector<vector<int> > result_matrix_00(split_index,

		row_vector);
			vector<vector<int> > result_matrix_01(split_index,
row_vector);
        vector<vector<int> > result_matrix_10(split_index,

row_vector);
        vector<vector<int> > result_matrix_11(split_index,

row_vector);

        vector<vector<int> > a00(split_index, row_vector);
        vector<vector<int> > a01(split_index, row_vector);
        vector<vector<int> > a10(split_index, row_vector);
        vector<vector<int> > a11(split_index, row_vector);
        vector<vector<int> > b00(split_index, row_vector);
        vector<vector<int> > b01(split_index, row_vector);
        vector<vector<int> > b10(split_index, row_vector);
        vector<vector<int> > b11(split_index, row_vector);

        for (auto i = 0; i < split_index; i++)
                for (auto j = 0; j < split_index; j++) {
                        a00[i][j] = matrix_A[i][j];
                        a01[i][j] = matrix_A[i][j + split_index];
                        a10[i][j] = matrix_A[split_index + i][j];
                        a11[i][j] = matrix_A[i + split_index]
                                                        [j + split_index];
                        b00[i][j] = matrix_B[i][j];
                        b01[i][j] = matrix_B[i][j + split_index];
                        b10[i][j] = matrix_B[split_index + i][j];
                        b11[i][j] = matrix_B[i + split_index]
                                                        [j + split_index];
                }

        add_matrix(multiply_matrix(a00, b00),
                        multiply_matrix(a01, b10),
                        result_matrix_00, split_index);
        add_matrix(multiply_matrix(a00, b01),
                        multiply_matrix(a01, b11),
                        result_matrix_01, split_index);
        add_matrix(multiply_matrix(a10, b00),
                        multiply_matrix(a11, b10),
                        result_matrix_10, split_index);
        add_matrix(multiply_matrix(a10, b01),
                        multiply_matrix(a11, b11),
                        result_matrix_11, split_index);

        for (auto i = 0; i < split_index; i++)
```

```
                    for (auto j = 0; j < split_index; j++) {
                        result_matrix[i][j]
                            = result_matrix_00[i][j];
                        result_matrix[i][j + split_index]
                            = result_matrix_01[i][j];
                        result_matrix[split_index + i][j]
                            = result_matrix_10[i][j];
                        result_matrix[i + split_index]
                                    [j + split_index]
                            = result_matrix_11[i][j];
                    }

                result_matrix_00.clear();
                result_matrix_01.clear();
                result_matrix_10.clear();
                result_matrix_11.clear();
                a00.clear();
                a01.clear();
                a10.clear();
                a11.clear();
                b00.clear();
                b01.clear();
                b10.clear();
                b11.clear();
            }
            return result_matrix;
        }

        int main()
        {
            vector<vector<int> > matrix_A = { { 1, 1, 1, 1 },
                                              { 2, 2, 2, 2 },
                                              { 3, 3, 3, 3 },
                                              { 2, 2, 2, 2 } };

            print("Array A", matrix_A, 0, 0, ROW_1 - 1, COL_1 - 1);

            vector<vector<int> > matrix_B = { { 1, 1, 1, 1 },
                                              { 2, 2, 2, 2 },
                                              { 3, 3, 3, 3 },
                                              { 2, 2, 2, 2 } };

            print("Array B", matrix_B, 0, 0, ROW_2 - 1, COL_2 - 1);

            vector<vector<int> > result_matrix(
                multiply_matrix(matrix_A, matrix_B));
```

```
                    print("Result Array", result_matrix, 0, 0, ROW_1 - 1,
                            COL_2 - 1);
            }
```

Sample input and output:

```
Array A =>
      1       1       1       1
      2       2       2       2
      3       3       3       3
      2       2       2       2


Array B =>
      1       1       1       1
      2       2       2       2
      3       3       3       3
      2       2       2       2


Result Array =>
      8       8       8       8
     16      16      16      16
     24      24      24      24
     16      16      16      16
```

Algorithm analysis:

Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as
$T(N) = 7T(N/2) + O(N^2)$

From Master's Theorem, time complexity of above method is
$O(N^{Log7})$ which is approximately $O(N^{2.8074})$

WEEK 6	**FINDING MAXIMUM AND MINIMUM IN AN ARRAY, CONVEX HULL PROBLEM**

6.a. Aim:

Implement a divide and conquer algorithm to find a maximum and minimum element in an array

Procedure:

Given an array of size **N.** The task is to find the maximum and the minimum element of the array using the minimum number of comparisons.

Algorithm:

```
Pair MaxMin(array, array_size)
    if array_size = 1
        return element as both max and min
    else if arry_size = 2
        one comparison to determine max and min
         return that pair
    else    /* array_size  > 2 */
        recur for max and min of left half
        recur for max and min of right half
        one comparison determines true max of the two candidates
        one comparison determines true min of the two candidates
        return the pair of max and min
```

Code:

```
// C++ program of above implementation
#include <iostream>
using namespace std;

// structure is used to return
// two values from minMax()
struct Pair {
            int min;
            int max;
};

struct Pair getMinMax(int arr[], int low, int high)
{
            struct Pair minmax, mml, mmr;
            int mid;

            // If there is only one element
            if (low == high) {
                        minmax.max = arr[low];
```

```
                minmax.min = arr[low];
                return minmax;
        }

        // If there are two elements
        if (high == low + 1) {
                if (arr[low] > arr[high]) {
                        minmax.max = arr[low];
                        minmax.min = arr[high];
                }
                else {
                        minmax.max = arr[high];
                        minmax.min = arr[low];
                }
                return minmax;
        }

        // If there are more than 2 elements
        mid = (low + high) / 2;
        mml = getMinMax(arr, low, mid);
        mmr = getMinMax(arr, mid + 1, high);

        // Compare minimums of two parts
        if (mml.min < mmr.min)
                minmax.min = mml.min;
        else
                minmax.min = mmr.min;

        // Compare maximums of two parts
        if (mml.max > mmr.max)
                minmax.max = mml.max;
        else
                minmax.max = mmr.max;

        return minmax;
}

// Driver code
int main()
{
        int arr[] = { 1000, 11, 445, 1, 330, 3000 };
        int arr_size = 6;

        struct Pair minmax = getMinMax(arr, 0, arr_size - 1);

        cout << "Minimum element is " << minmax.min << endl;
```

```
                cout << "Maximum element is " << minmax.max;

                return 0;
        }
```

Sample input and output:

Input: arr[] = {3, 5, 4, 1, 9}
Output: Minimum element is: 1
          Maximum element is: 9
Input: arr[] = {22, 14, 8, 17, 35, 3}
Output: Minimum element is: 3
          Maximum element is: 35

Algorithm analysis:

T(n) = T(floor(n/2)) + T(ceil(n/2)) + 2

T(2) = 1

T(1) = 0

If n is a power of 2, then we can write T(n) as:
T(n) = 2T(n/2) + 2

After solving the above recursion, we get
T(n) = 3n/2 -2

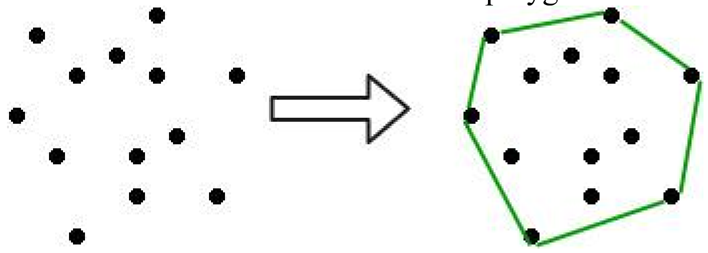
Thus, the approach does 3n/2 -2 comparisons if n is a power of 2. And it does more than 3n/2 -2 comparisons if n is not a power of 2.

6.b. Aim:

To implement a divide and conquer algorithm to construct a convex hull.

Procedure:

A convex hull is the smallest convex polygon containing all the given points.



Input is an array of points specified by their x and y coordinates. The output is the convex hull of this set of points.

Algorithm:

The QuickHull algorithm is a Divide and Conquer algorithm similar to QuickSort. Let a[0…n-1] be the input array of points. Following are the steps for finding the convex hull of these points.

1. Find the point with minimum x-coordinate lets say, min_x and similarly the point with maximum x-coordinate, max_x.
2. Make a line joining these two points, say **L**. This line will divide the whole set into two parts. Take both the parts one by one and proceed further.
3. For a part, find the point P with maximum distance from the line L. P forms a triangle with the points min_x, max_x. It is clear that the points residing inside this triangle can never be the part of convex hull.
4. The above step divides the problem into two sub-problems (solved recursively). Now the line joining the points P and min_x and the line joining the points P and max_x are new lines and the points residing outside the triangle is the set of points. Repeat point no. 3 till there no point left with the line. Add the end points of this point to the convex hull.

Code:

```
// C++ program to implement Quick Hull algorithm
// to find convex hull.
#include<bits/stdc++.h>
using namespace std;

// iPair is integer pairs
#define iPair pair<int, int>

// Stores the result (points of convex hull)
set<iPair> hull;

// Returns the side of point p with respect to line
// joining points p1 and p2.
int findSide(iPair p1, iPair p2, iPair p)
{
    int val = (p.second - p1.second) * (p2.first - p1.first) -
                (p2.second - p1.second) * (p.first - p1.first);

    if (val > 0)
        return 1;
    if (val < 0)
        return -1;
    return 0;
}

// returns a value proportional to the distance
// between the point p and the line joining the
// points p1 and p2
int lineDist(iPair p1, iPair p2, iPair p)
{
    return abs ((p.second - p1.second) * (p2.first - p1.first) -
                (p2.second - p1.second) * (p.first - p1.first));
}

// End points of line L are p1 and p2. side can have value
// 1 or -1 specifying each of the parts made by the line L
void quickHull(iPair a[], int n, iPair p1, iPair p2, int side)
{
    int ind = -1;
    int max_dist = 0;

    // finding the point with maximum distance
    // from L and also on the specified side of L.
    for (int i=0; i<n; i++)
    {
        int temp = lineDist(p1, p2, a[i]);
        if (findSide(p1, p2, a[i]) == side && temp > max_dist)
        {
            ind = i;
            max_dist = temp;
        }
    }

    // If no point is found, add the end points of L to the convex hull.
    if (ind == -1)
    {
        hull.insert(p1);
        hull.insert(p2);
```

```
            return;
    }

    // Recur for the two parts divided by a[ind]
    quickHull(a, n, a[ind], p1, -findSide(a[ind], p1, p2));
    quickHull(a, n, a[ind], p2, -findSide(a[ind], p2, p1));
}

void printHull(iPair a[], int n)
{
    // a[i].second -> y-coordinate of the ith point
    if (n < 3)
    {
            cout << "Convex hull not possible\n";
            return;
    }

    // Finding the point with minimum and maximum x-coordinate
    int min_x = 0, max_x = 0;
    for (int i=1; i<n; i++)
    {
            if (a[i].first < a[min_x].first)
                    min_x = i;
            if (a[i].first > a[max_x].first)
                    max_x = i;
    }

    // Recursively find convex hull points on one side of line joining a[min_x] and a[max_x]
    quickHull(a, n, a[min_x], a[max_x], 1);

    // Recursively find convex hull points on other side of line joining a[min_x] and
 a[max_x]
    quickHull(a, n, a[min_x], a[max_x], -1);

    cout << "The points in Convex Hull are:\n";
    while (!hull.empty())
    {
            cout << "(" <<( *hull.begin()).first << ", "
                    << (*hull.begin()).second << ") ";
            hull.erase(hull.begin());
    }
}

// Driver code
int main()
{
        iPair a[] = {{0, 3}, {1, 1}, {2, 2}, {4, 4},
                        {0, 0}, {1, 2}, {3, 1}, {3, 3}};
        int n = sizeof(a)/sizeof(a[0]);
        printHull(a, n);
        return 0;
    }
```

Sample input and output:

```
Input : points[] = {(0, 0), (0, 4), (-4, 0), (5, 0),
               (0, -6), (1, 0)};
Output : (-4, 0), (5, 0), (0, -6), (0, 4)
Input : points[] = {{0, 3}, {1, 1}, {2, 2}, {4, 4},
                {0, 0}, {1, 2}, {3, 1}, {3, 3}};
Output :  The points in convex hull are:
        (0, 0) (0, 3) (3, 1) (4, 4)

Input : points[] = {{0, 3}, {1, 1}
Output : Not Possible
```

There must be at least three points to form a hull.

Algorithm analysis:

The analysis is similar to Quick Sort. On average, we get time complexity as O(n Log n), but in worst case, it can become O(n)

WEEK 7	HUFFMAN CODING, KNAPSACK USING GREEDY

7.a. Aim :

To implement a greedy algorithm to perform encoding mechanism using huffman code method

Procedure:

Huffman Coding-

- Huffman Coding is a famous Greedy Algorithm.
- It is used for the lossless compression of data.
- It uses variable length encoding.
- It assigns variable length code to all the characters.
- The code length of a character depends on how frequently it occurs in the given text.
- The character which occurs most frequently gets the smallest code.
- The character which occurs least frequently gets the largest code.
- It is also known as **Huffman Encoding**.

Prefix Rule-

- Huffman Coding implements a rule known as a prefix rule.
- This is to prevent the ambiguities while decoding.
- It ensures that the code assigned to any character is not a prefix of the code assigned to any other character.

Major Steps in Huffman Coding-

There are two major steps in Huffman Coding-

1. Building a Huffman Tree from the input characters.
2. Assigning code to the characters by traversing the Huffman Tree.

Huffman Tree-

The steps involved in the construction of Huffman Tree are as follows-

Step-01:

- Create a leaf node for each character of the text.
- Leaf node of a character contains the occurring frequency of that character.

Step-02:

- Arrange all the nodes in increasing order of their frequency value.

Step-03:

Considering the first two nodes having minimum frequency,

- Create a new internal node.
- The frequency of this new node is the sum of frequency of those two nodes.
- Make the first node as a left child and the other node as a right child of the newly created node.

Step-04:

- Keep repeating Step-02 and Step-03 until all the nodes form a single tree.
- The tree finally obtained is the desired Huffman Tree.

Time Complexity-

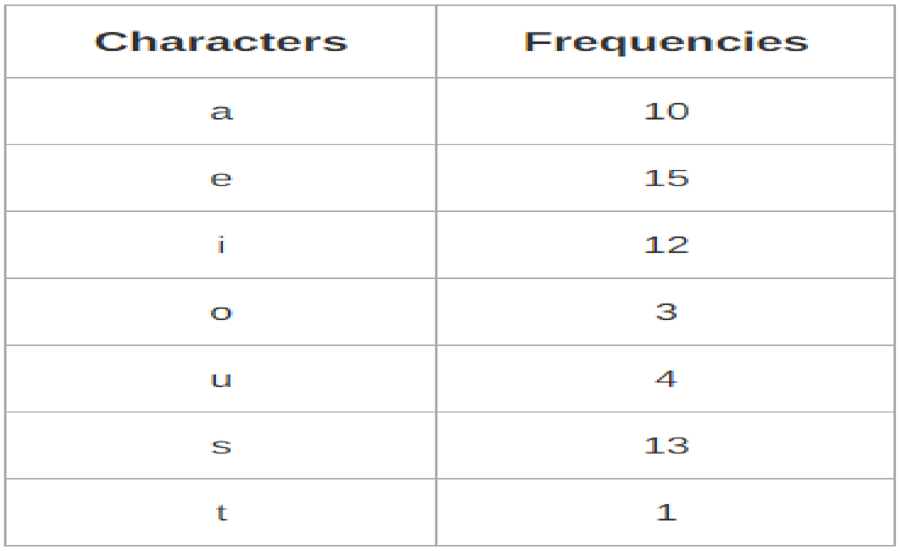
The time complexity analysis of Huffman Coding is as follows-

- extractMin( ) is called 2 x (n-1) times if there are n nodes.
- As extractMin( ) calls minHeapify( ), it takes O(logn) time.

Thus, Overall time complexity of Huffman Coding becomes **O(nlogn)**.

Here, n is the number of unique characters in the given text.

PRACTICE PROBLEM BASED ON HUFFMAN CODING-

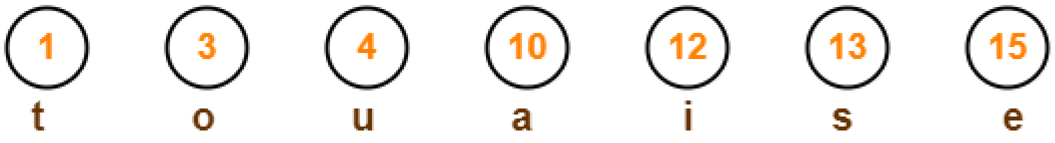


Characters	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1

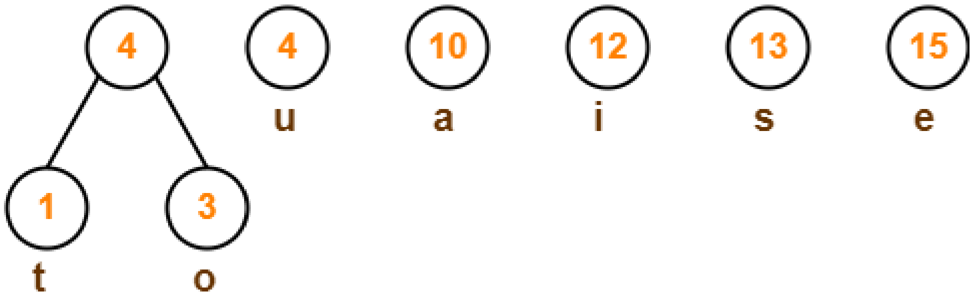
Solution-
First let us construct the Huffman Tree.

Huffman Tree is constructed in the following steps-

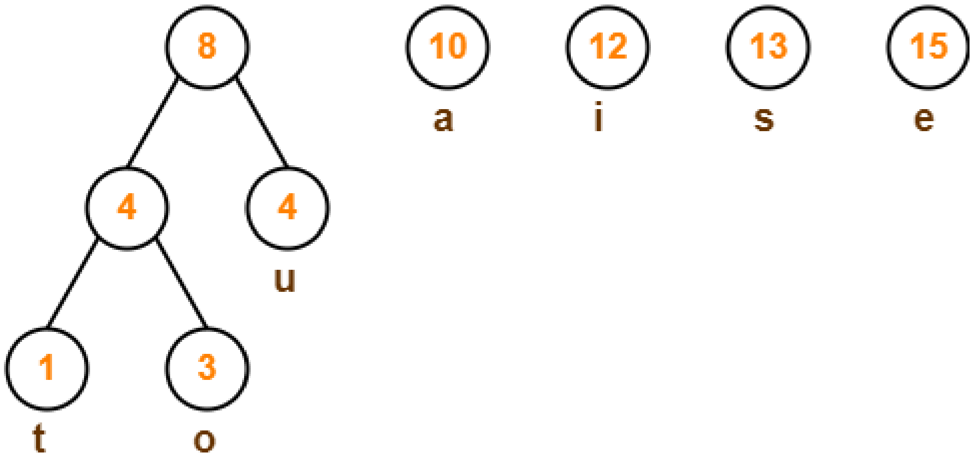
Step-01:



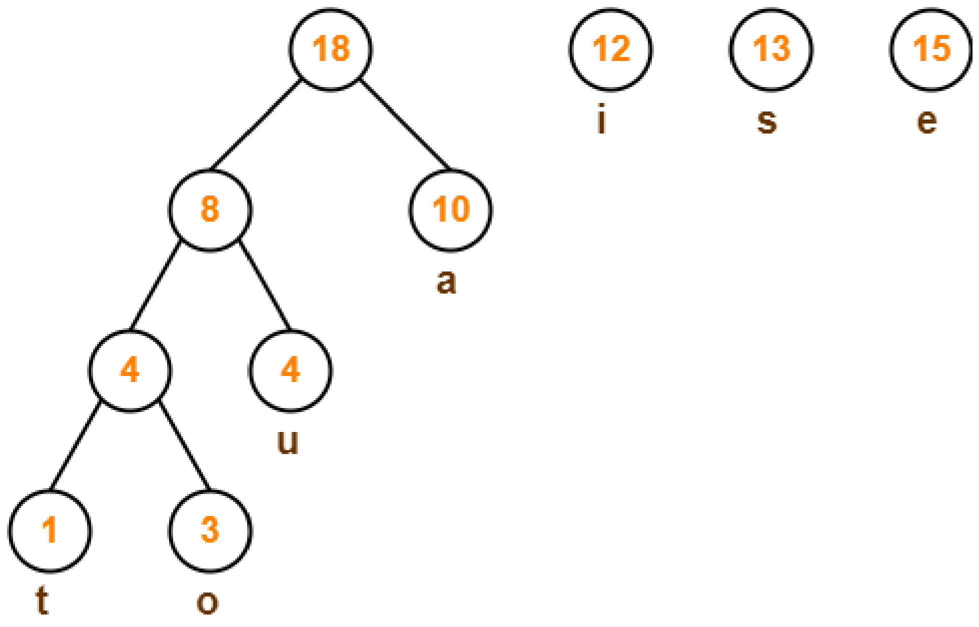
Step-02:



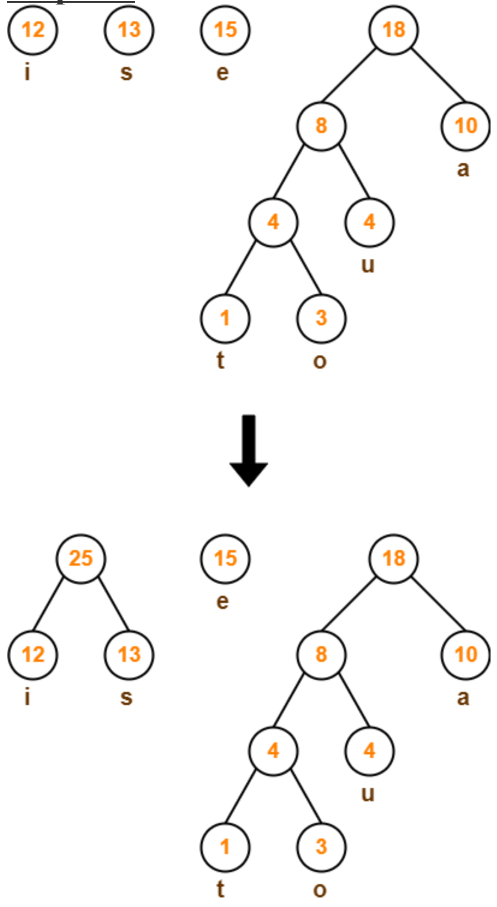
Step-03:



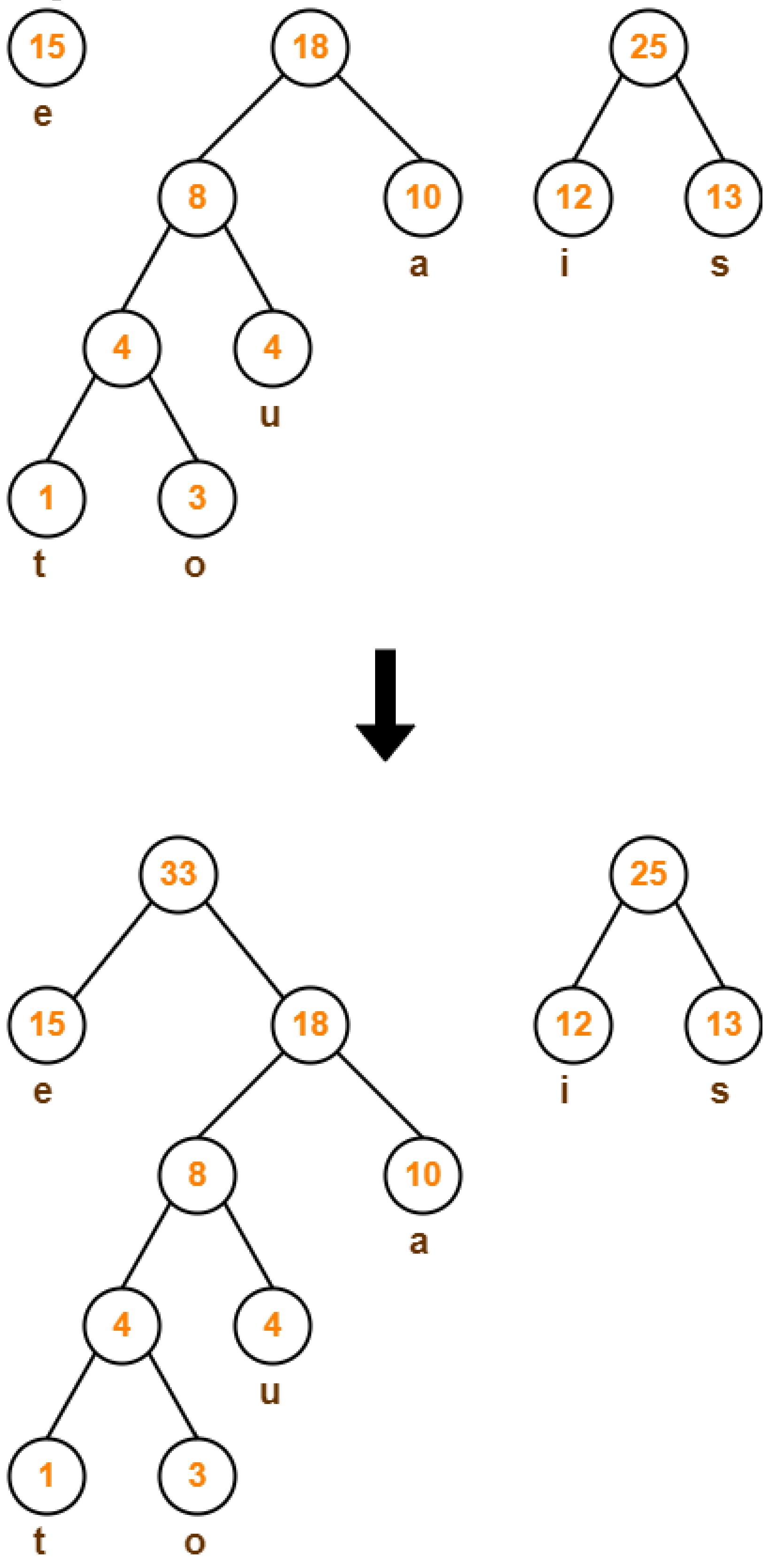
Step-04:


18
12
i
13
s
15
e
8
10
a
4
4
u
1
t
3
o

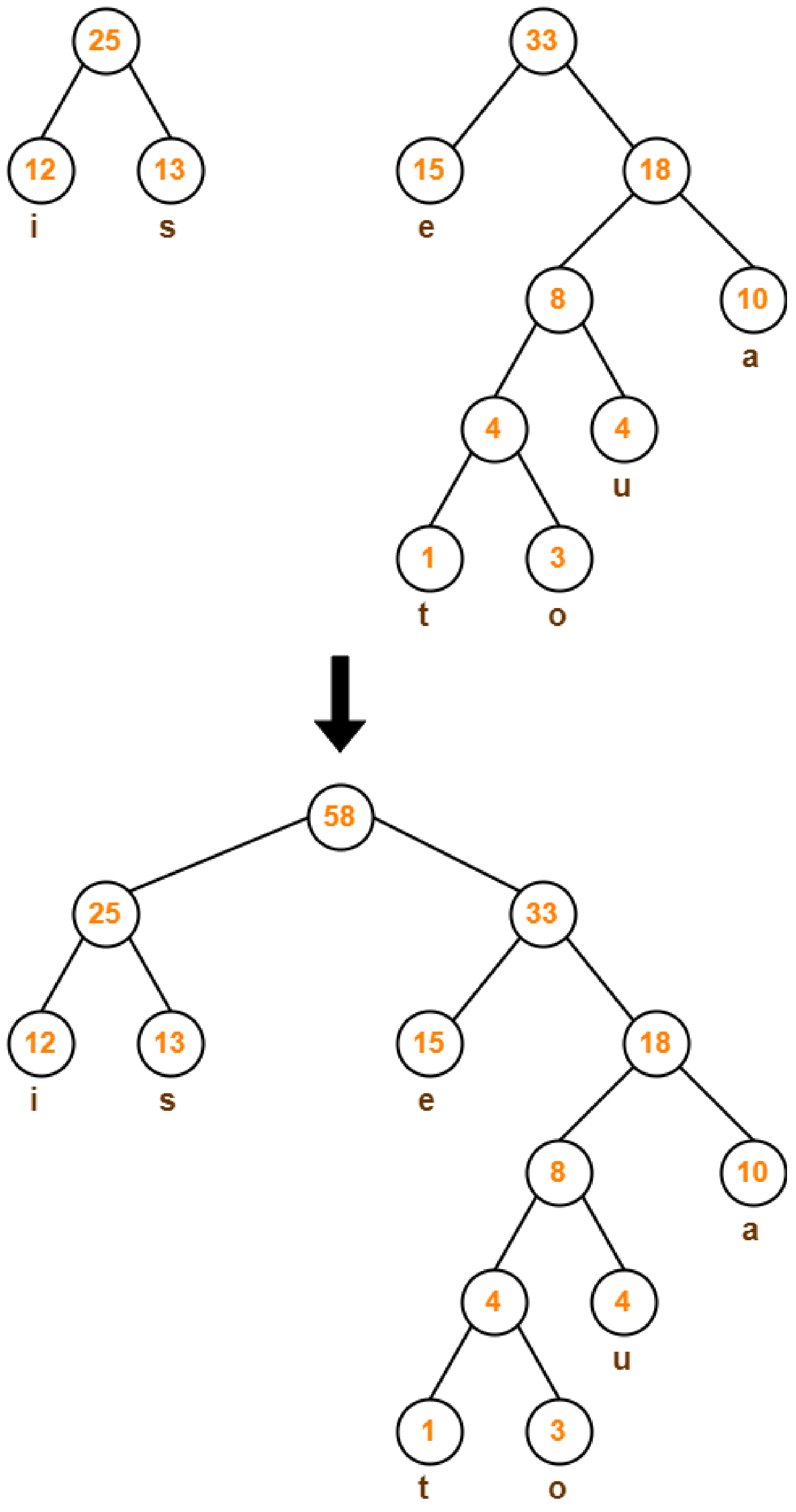
Step-05:


12
i
13
s
15
e
18
8
10
a
4
4
u
1
t
3
o
25
12
i
13
s
15
e
18
8
10
a
4
4
u
1
t
3
o

Step-06:


15
e
18
8
10
a
4
4
u
1
t
3
o
25
12
i
13
s
33
15
e
18
8
10
a
4
4
u
1
t
3
o
25
12
i
13
s

Step-07:



Huffman Tree

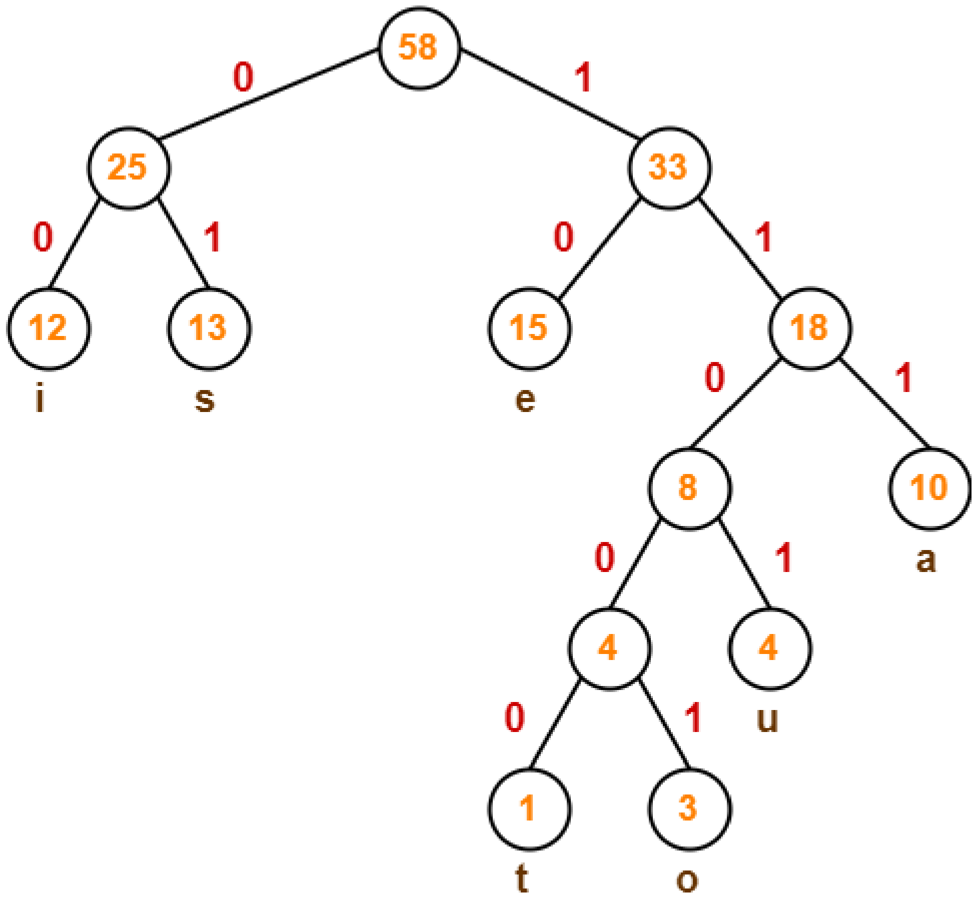
Now,

- We assign weight to all the edges of the constructed Huffman Tree.
- Let us assign weight '0' to the left edges and weight '1' to the right edges.

Rule

- If you assign weight '0' to the left edges, then assign weight '1' to the right edges.
- If you assign weight '1' to the left edges, then assign weight '0' to the right edges.
- Any of the above two conventions may be followed.
- But follow the same convention at the time of decoding that is adopted at the time of encoding.

After assigning weight to all the edges, the modified Huffman Tree is-



Huffman Tree

Now, let us answer each part of the given problem one by one-

Huffman Code For Characters-

-

To write Huffman Code for any character, traverse the Huffman Tree from root node to the leaf node of that character.

Following this rule, the Huffman Code for each character is-

- a = 111
- e = 10
- i = 00
- o = 11001
- u = 1101
- s = 01
- t = 11000

From here, we can observe-

- Characters occurring less frequently in the text are assigned the larger code.
- Characters occurring more frequently in the text are assigned the smaller code.

Huffman Coding Algorithm

```
create a priority queue Q consisting of each unique character.
sort then in ascending order of their frequencies.
for all the unique characters:
    create a newNode
    extract minimum value from Q and assign it to leftChild of newNode
    extract minimum value from Q and assign it to rightChild of newNode
    calculate the sum of these two minimum values and assign it to the value of newNode
    insert this newNode into the tree
return rootNode
```

```
// Huffman Coding in C

#include <stdio.h>
#include <stdlib.h>

#define MAX_TREE_HT 50

struct MinHNode {
  char item;
  unsigned freq;
  struct MinHNode *left, *right;
};

struct MinHeap {
  unsigned size;
  unsigned capacity;
  struct MinHNode **array;
};

// Create nodes
struct MinHNode *newNode(char item, unsigned freq) {
  struct MinHNode *temp = (struct MinHNode *)malloc(sizeof(struct MinHNode));

  temp->left = temp->right = NULL;
  temp->item = item;
  temp->freq = freq;

  return temp;
}

// Create min heap
struct MinHeap *createMinH(unsigned capacity) {
  struct MinHeap *minHeap = (struct MinHeap *)malloc(sizeof(struct MinHeap));

  minHeap->size = 0;

  minHeap->capacity = capacity;

    minHeap->array = (struct MinHNode **)malloc(minHeap->capacity * sizeof(struct
MinHNode *));
  return minHeap;
}

// Function to swap
void swapMinHNode(struct MinHNode **a, struct MinHNode **b) {
  struct MinHNode *t = *a;
  *a = *b;
  *b = t;
}

// Heapify
void minHeapify(struct MinHeap *minHeap, int idx) {
  int smallest = idx;
  int left = 2 * idx + 1;
  int right = 2 * idx + 2;
```

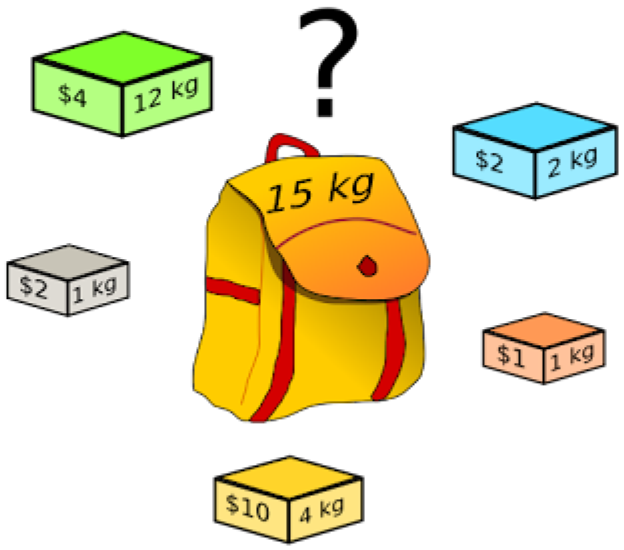
```
 if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]-
>freq)
    smallest = left;

      if (right < minHeap->size && minHeap->array[right]->freq < minHeap-
>array[smallest]->freq)
    smallest = right;

  if (smallest != idx) {
    swapMinHNode(&minHeap->array[smallest], &minHeap->array[idx]);
    minHeapify(minHeap, smallest);
  }
}

// Check if size if 1
int checkSizeOne(struct MinHeap *minHeap) {
  return (minHeap->size == 1);
}

// Extract min
struct MinHNode *extractMin(struct MinHeap *minHeap) {
  struct MinHNode *temp = minHeap->array[0];
  minHeap->array[0] = minHeap->array[minHeap->size - 1];

  --minHeap->size;
  minHeapify(minHeap, 0);

  return temp;
}

// Insertion function
void insertMinHeap(struct MinHeap *minHeap, struct MinHNode *minHeapNode) {
  ++minHeap->size;
  int i = minHeap->size - 1;

  while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {
    minHeap->array[i] = minHeap->array[(i - 1) / 2];
    i = (i - 1) / 2;
  }
  minHeap->array[i] = minHeapNode;
}

void buildMinHeap(struct MinHeap *minHeap) {
  int n = minHeap->size - 1;
  int i;
  for (i = (n - 1) / 2; i >= 0; --i)
    minHeapify(minHeap, i);
}

int isLeaf(struct MinHNode *root) {
  return !(root->left) && !(root->right);
}

struct MinHeap *createAndBuildMinHeap(char item[], int freq[], int size) {
  struct MinHeap *minHeap = createMinH(size);

  for (int i = 0; i < size; ++i)
    minHeap->array[i] = newNode(item[i], freq[i]);

  minHeap->size = size;
  buildMinHeap(minHeap);

  return minHeap;
}

struct MinHNode *buildHuffmanTree(char item[], int freq[], int size) {
  struct MinHNode *left, *right, *top;
  struct MinHeap *minHeap = createAndBuildMinHeap(item, freq, size);

  while (!checkSizeOne(minHeap)) {
    left = extractMin(minHeap);
    right = extractMin(minHeap);

    top = newNode('$', left->freq + right->freq);

    top->left = left;
    top->right = right;

    insertMinHeap(minHeap, top);
  }
  return extractMin(minHeap);
}

void printHCodes(struct MinHNode *root, int arr[], int top) {
  if (root->left) {
    arr[top] = 0;
    printHCodes(root->left, arr, top + 1);
  }
  if (root->right) {
    arr[top] = 1;
    printHCodes(root->right, arr, top + 1);
  }
  if (isLeaf(root)) {
    printf("  %c   | ", root->item);
    printArray(arr, top);
  }
}

// Wrapper function
void HuffmanCodes(char item[], int freq[], int size) {
  struct MinHNode *root = buildHuffmanTree(item, freq, size);

  int arr[MAX_TREE_HT], top = 0;

  printHCodes(root, arr, top);
}

// Print the array
void printArray(int arr[], int n) {
  int i;
  for (i = 0; i < n; ++i)
    printf("%d", arr[i]);

  printf("\n");
}

int main() {
  char arr[] = {'A', 'B', 'C', 'D'};
  int freq[] = {5, 1, 6, 3};

  int size = sizeof(arr) / sizeof(arr[0]);

  printf(" Char | Huffman code ");
  printf("\n--------------------\n");

  HuffmanCodes(arr, freq, size);
}
```

7.b. Aim:

To implement a greedy algorithm to implement knapsack method

Procedure:

//Program to implement knapsack problem using greedy method



What actually Problem Says ?

1. Given a set of items, each with a **weight and a value**.
2. Determine the **number of each item** to include in a collection so that the total weight is less than a given limit and the total value is as large as possible.
3. It derives its name from the problem faced by someone who is constrained by a **fixed-size knapsack** and must fill it with the most useful items.

Program:

```
# include<stdio.h>

void knapsack(int n, float weight[], float profit[], float capacity) {
float x[20], tp = 0;
int i, j, u;
u = capacity;

for (i = 0; i < n; i++)
x[i] = 0.0;
```

```
for (i = 0; i < n; i++) {
if (weight[i] > u)
break;
else {
x[i] = 1.0;
tp = tp + profit[i];
u = u - weight[i];
}
}

if (i < n)
x[i] = u / weight[i];

tp = tp + (x[i] * profit[i]);

printf("\nThe result vector is:- ");
for (i = 0; i < n; i++)
printf("%f\t", x[i]);

printf("\nMaximum profit is:- %f", tp);

}

int main() {
float weight[20], profit[20], capacity;
int num, i, j;
float ratio[20], temp;

printf("\nEnter the no. of objects:- ");
scanf("%d", &num);

printf("\nEnter the wts and profits of each object:- ");
for (i = 0; i < num; i++) {
scanf("%f %f", &weight[i], &profit[i]);
}

printf("\nEnter the capacityacity of knapsack:- ");
scanf("%f", &capacity);

for (i = 0; i < num; i++) {
ratio[i] = profit[i] / weight[i];
}

for (i = 0; i < num; i++) {
for (j = i + 1; j < num; j++) {
if (ratio[i] < ratio[j]) {
```

```
            temp = ratio[j];
            ratio[j] = ratio[i];
            ratio[i] = temp;

            temp = weight[j];
            weight[j] = weight[i];
            weight[i] = temp;

            temp = profit[j];
            profit[j] = profit[i];
            profit[i] = temp;
    }
    }
    }

    knapsack(num, weight, profit, capacity);
    return(0);
}
```

```
Enter the no. of objects:- 7

Enter the wts and profits of each object:-
2 10
3 5
5 15
7 7
1 6
4 18
1 3

Enter the capacity of knapsack:- 15

The result vector is:- 1.000000     1.000000     1.000000     1.000000
   1.000000     0.666667     0.000000

Maximum profit is:- 55.333332
```

Sorting of n items (or objects) in decreasing order of the ratio Pj/Wj takes O (n log n) time. Since this is the lower bound for any comparison-based sorting algorithm. Therefore, the total time including sort is O(n log n).

WEEK 8	**TREE TRAVERSALS, MST USING KRUSKAL'S ALGORITHM**

8.a. Aim:

To implement a greedy algorithm to perform Tree Traversals

Procedure:

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

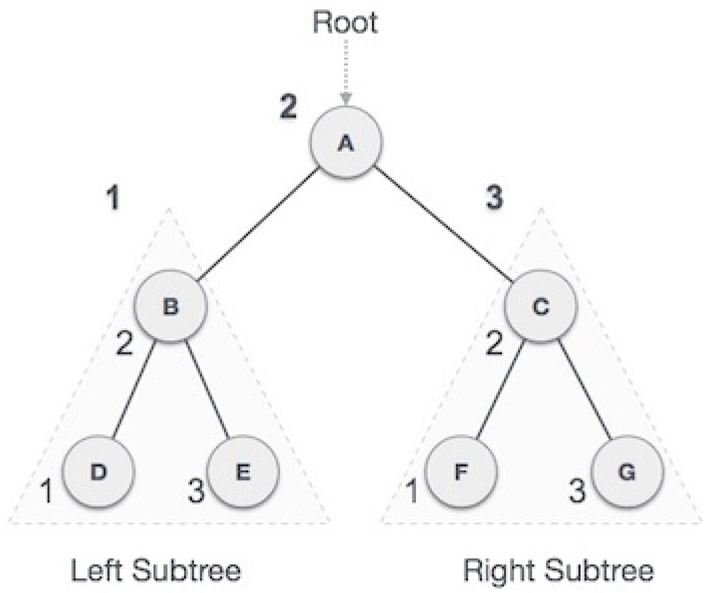
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

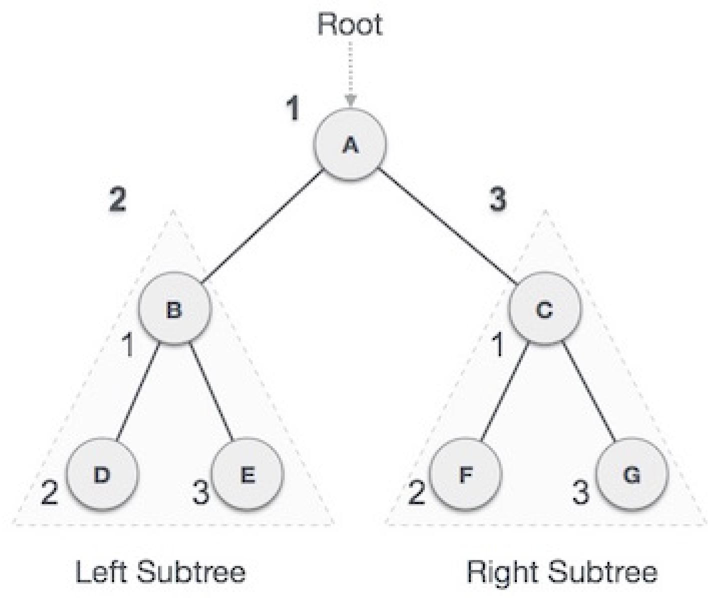
D → B → E → A → F → C → G

Algorithm

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

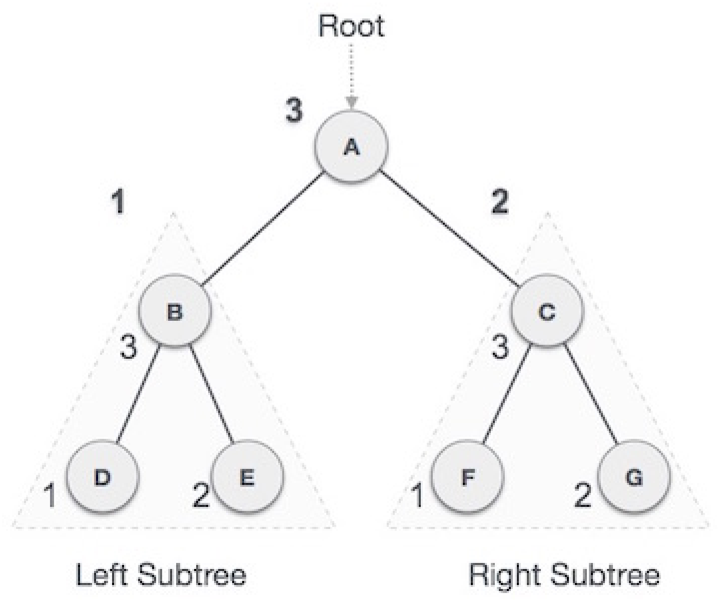
A → B → D → E → C → F → G

Algorithm

Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

D → E → B → F → G → C → A

Algorithm

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;

   struct node *leftChild;
   struct node *rightChild;
};

struct node *root = NULL;

void insert(int data) {
   struct node *tempNode = (struct node*) malloc(sizeof(struct node));
   struct node *current;
   struct node *parent;

   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;
   //if tree is empty
   if(root == NULL) {
      root = tempNode;
   } else {
      current = root;
      parent = NULL;

      while(1) {
         parent = current;

         //go to left of the tree
         if(data < parent->data) {
            current = current->leftChild;

            //insert to the left
            if(current == NULL) {
               parent->leftChild = tempNode;
               return;
            }
         }  //go to right of the tree
         else {
            current = current->rightChild;

            //insert to the right
            if(current == NULL) {
               parent->rightChild = tempNode;
               return;
            }
         }
      }
   }
}

struct node* search(int data) {
   struct node *current = root;
   printf("Visiting elements: ");

   while(current->data != data) {
      if(current != NULL)
         printf("%d ",current->data);

      //go to left tree
      if(current->data > data) {
         current = current->leftChild;
      }
      //else go to right tree
      else {
```

```
      current = current->rightChild;
    }

    //not found
    if(current == NULL) {
      return NULL;
    }
  }

  return current;
}

void pre_order_traversal(struct node* root) {
  if(root != NULL) {
    printf("%d ",root->data);
    pre_order_traversal(root->leftChild);
    pre_order_traversal(root->rightChild);
  }
}

void inorder_traversal(struct node* root) {
  if(root != NULL) {
    inorder_traversal(root->leftChild);
    printf("%d ",root->data);
    inorder_traversal(root->rightChild);
  }
}

void post_order_traversal(struct node* root) {
  if(root != NULL) {
    post_order_traversal(root->leftChild);
    post_order_traversal(root->rightChild);
    printf("%d ", root->data);
  }
}

int main() {
  int i;
  int array[7] = { 27, 14, 35, 10, 19, 31, 42 };

  for(i = 0; i < 7; i++)
    insert(array[i]);

  i = 31;
  struct node * temp = search(i);

  if(temp != NULL) {
    printf("[%d] Element found.", temp->data);
    printf("\n");
  }else {
    printf("[ x ] Element not found (%d).\n", i);
  }

  i = 15;
  temp = search(i);

  if(temp != NULL) {
    printf("[%d] Element found.", temp->data);
    printf("\n");
  }else {
    printf("[ x ] Element not found (%d).\n", i);
  }

  printf("\nPreorder traversal: ");
  pre_order_traversal(root);

  printf("\nInorder traversal: ");
  inorder_traversal(root);

  printf("\nPost order traversal: ");
  post_order_traversal(root);

  return 0;
}
```

Output

```
Visiting elements: 27 35 [31] Element found.
Visiting elements: 27 14 19 [ x ] Element not found (15).

Preorder traversal: 27 14 10 19 35 31 42
Inorder traversal: 10 14 19 27 31 35 42
Post order traversal: 10 19 14 31 42 35 27
```

Time Complexity Analysis:

In general, if we want to analyze the time complexity of a tree traversal then we have to think in the terms of the number of nodes visited.

Hence, if a tree has n nodes, then each node is visited only once in inorder, preorder or post order traversal and hence the complexity of the inorder traversal of the binary tree is O(n) .

8.b. Aim:

Implement a greedy algorithm to find minimum spanning tree using kruskal's algorithm.

Procedure:

Kruskal algorithm is used to generate a minimum spanning tree for a given graph. But, what exactly is a minimum spanning tree? A minimum spanning tree is a subset of a graph with the same number of vertices as the graph and edges equal to the number of vertices -1. It also has a minimal cost for the sum of all edge weights in a spanning tree.

Kruskal's algorithm sorts all the edges in increasing order of their edge weights and keeps adding nodes to the tree only if the chosen edge does not form any cycle. Also, it picks the edge with a minimum cost at first and the edge with a maximum cost at last. Hence, you can say that the Kruskal algorithm makes a locally optimal choice, intending to find the global optimal solution. That is why it is called a Greedy Algorithm.

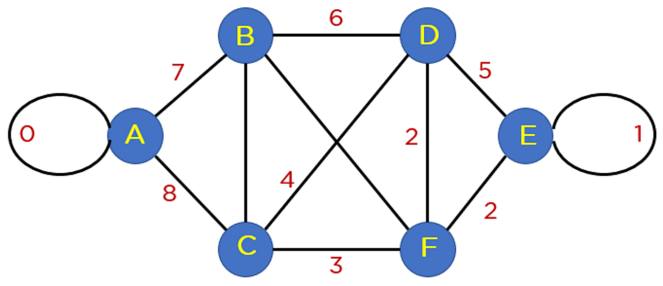
Creating Minimum Spanning Tree Using Kruskal Algorithm

First look into the steps involved in Kruskal's Algorithm to generate a minimum spanning tree:

- Step 1: Sort all edges in increasing order of their edge weights.
- Step 2: Pick the smallest edge.
- Step 3: Check if the new edge creates a cycle or loop in a spanning tree.
- Step 4: If it doesn't form the cycle, then include that edge in MST. Otherwise, discard it.
- Step 5: Repeat from step 2 until it includes |V| - 1 edges in MST.

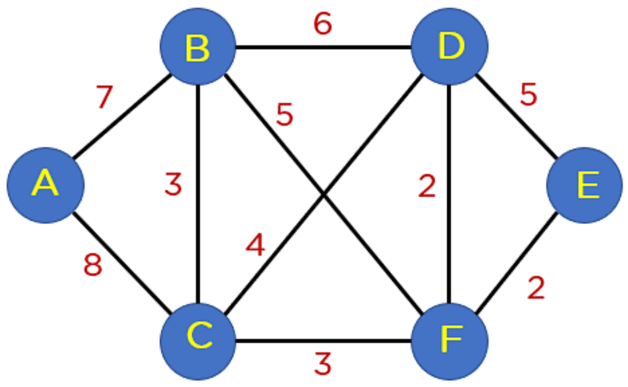
Using the steps mentioned above, you will generate a minimum spanning tree structure. So, now have a look at an example to understand this process better.

The graph G(V, E) given below contains 6 vertices and 12 edges. And you will create a minimum spanning tree T(V', E') for G(V, E) such that the number of vertices in T will be 6 and edges will be 5 (6-1).



Graph G(V, E)

If you observe this graph, you'll find two looping edges connecting the same node to itself again. And you know that the tree structure can never include a loop or parallel edge. Hence, primarily you will need to remove these edges from the graph structure.



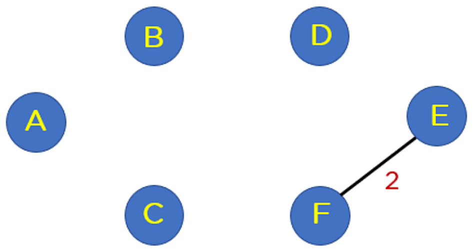
Removing parallel edges or loops from the graph.

The next step that you will proceed with is arranging all edges in a sorted list by their edge weights.

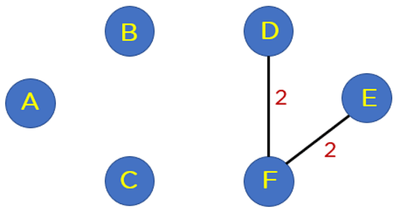
The Edges of the Graph	Edge Weight

Source Vertex	Destination Vertex	
E	F	2
F	D	2
B	C	3
C	F	3
C	D	4
B	F	5
B	D	6
A	B	7
A	C	8

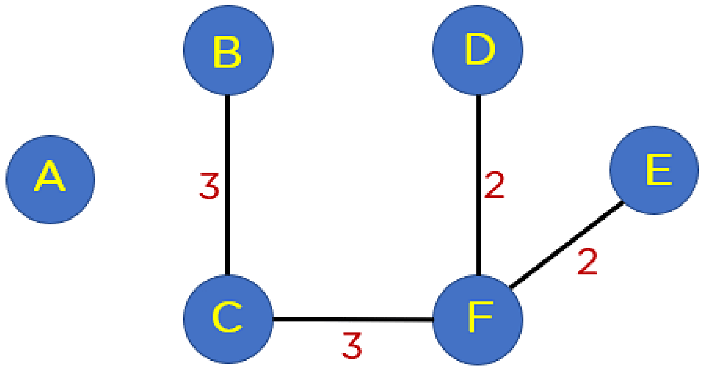
After this step, you will include edges in the MST such that the included edge would not form a cycle in your tree structure. The first edge that you will pick is edge EF, as it has a minimum edge weight that is 2.



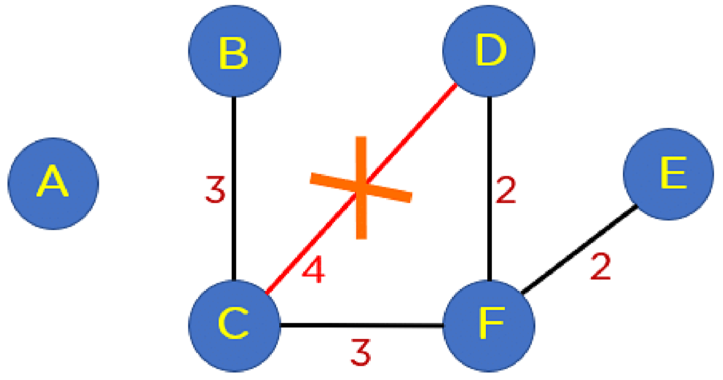
Add edge FD to the spanning tree.



Add edge BC and edge CF to the spanning tree as it does not generate any loop.

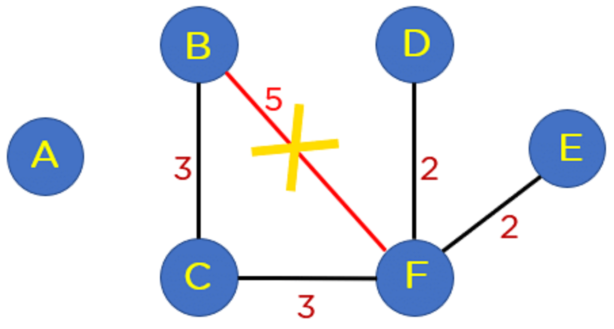


Next up is edge CD. This edge generates the loop in Your tree structure. Thus, you will discard this edge.



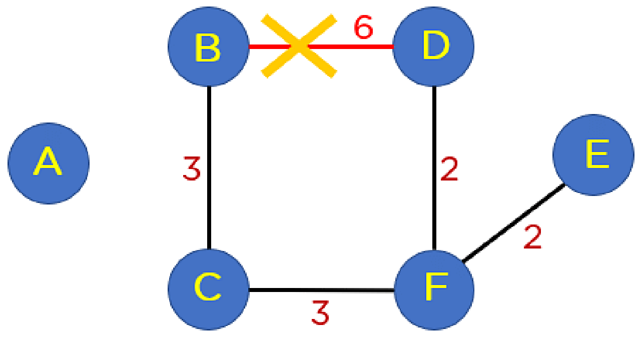
Edge CD should be discarded, as it creates loop.

Following edge CD, you have edge BF. This edge also creates the loop; hence you will discard it.



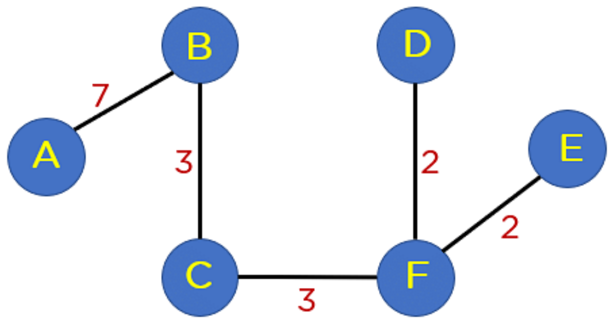
Edge BF should be discarded.

Next up is edge BD. This edge also formulates a loop, so you will discard it as well.



Edge BD should be discarded.

Next on your sorted list is edge AB. This edge does not generate any cycle, so you need not include it in the MST structure. By including this node, it will include 5 edges in the MST, so you don't have to traverse any further in the sorted list. The final structure of your MST is represented in the image below:



Minimum Spanning Tree.

The summation of all the edge weights in MST T(V', E') is equal to 17, which is the least possible edge weight for any possible spanning tree structure for this particular graph.

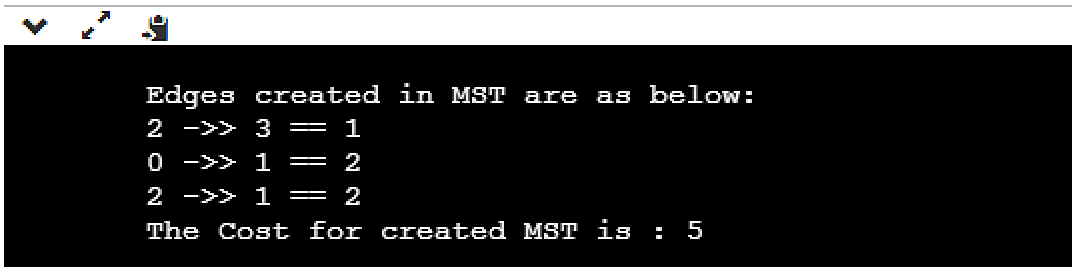
The C program to implement Kruskal's algorithm using above mentioned strategy is as follows:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
//structure that denotes a weighted edge
struct Edge {
int source, destination, weight;
};
//structure that denotes a weighted, undirected and connected graph
struct Graph {
int Node, E;
struct Edge* edge;
};
//allocates memory for storing graph with V vertices and E edges
struct Graph* GenerateGraph(int Node, int E)
{
struct Graph* graph = (struct Graph*)(malloc(sizeof(struct Graph)));
graph->Node = Node;
graph->E = E;
graph->edge = (struct Edge*)malloc(sizeof( struct Edge));
return graph;
}
//subset for Union-Find
struct tree_maintainance_set {
int parent;
int rank;
};
//finds the set of chosen element i using path compression
int find_DisjointSet(struct tree_maintainance_set subsets[], int i)
{
    //find root and make root as parent of i
if (subsets[i].parent != i)
subsets[i].parent
= find_DisjointSet(subsets, subsets[i].parent);
```

```
return subsets[i].parent;
}
//Creates the Union of two sets
void Union_DisjointSet(struct tree_maintainance_set subsets[], int x, int y)
{
int xroot = find_DisjointSet(subsets, x);
int yroot = find_DisjointSet(subsets, y);
  //connecting tree with lowest rank to the tree with highest rank
if (subsets[xroot].rank < subsets[yroot].rank)
subsets[xroot].parent = yroot;
else if (subsets[xroot].rank > subsets[yroot].rank)
subsets[yroot].parent = xroot;
  //if ranks are same, arbitrarily increase the rank of one node
else
{
subsets[yroot].parent = xroot;
subsets[xroot].rank++;
}
}
//function to compare edges using qsort() in C programming
int myComp(const void* a, const void* b)
{
struct Edge* a1 = (struct Edge*)a;
struct Edge* b1 = (struct Edge*)b;
return a1->weight > b1->weight;
}
//function to construct MST using Kruskal’s approach
void KruskalMST(struct Graph* graph)
{
int Node = graph->Node;
struct Edge
result[Node];
int e = 0;
int i = 0;
    //sorting all edges
qsort(graph->edge, graph->E, sizeof(graph->edge[0]),
myComp);
    //memory allocation for V subsets
struct tree_maintainance_set* subsets
= (struct tree_maintainance_set*)malloc(Node * sizeof(struct tree_maintainance_set));
    //V subsets containing only one element
for (int v = 0; v < Node; ++v) {
subsets[v].parent = v;
subsets[v].rank = 0;
}
    //Edge traversal limit: V-1
while (e < Node - 1 && i < graph->E) {
struct Edge next_edge = graph->edge[i++];
int x = find_DisjointSet(subsets, next_edge.source);
int y = find_DisjointSet(subsets, next_edge.destination);
if (x != y) {
result[e++] = next_edge;
Union_DisjointSet(subsets, x, y);
}
}
    //printing MST
printf(
"Edges created in MST are as below: \n");
int minimumCost = 0;
for (i = 0; i < e; ++i)
{
printf("%d -- %d == %d\n", result[i].source,
result[i].destination, result[i].weight);
minimumCost += result[i].weight;
}
printf("The Cost for created MST is : %d",minimumCost);
return;
}
int main()
{
int Node = 4;
int E = 6;
```

```
struct Graph* graph = GenerateGraph(Node, E);
    //Creating graph with manual value insertion
// add edge 0-1
graph->edge[0].source = 0;
graph->edge[0].destination = 1;
graph->edge[0].weight = 2;
}
// add edge 0-2
graph->edge[1].source = 0;
graph->edge[1].destination = 2;
graph->edge[1].weight = 4;
// add edge 0-3
graph->edge[2].source = 0;
graph->edge[2].destination = 3;
graph->edge[2].weight = 4;
// add edge 1-3
graph->edge[3].source = 1;
graph->edge[3].destination = 3;
graph->edge[3].weight = 3;
// add edge 2-3
graph->edge[4].source = 2;
graph->edge[4].destination = 3;
graph->edge[4].weight = 1;
// add edge 1-2
graph->edge[5].source = 1;
graph->edge[5].destination = 2;
graph->edge[5].weight = 2;
KruskalMST(graph);
return 0;
}
```

Output:



```
Edges created in MST are as below:
2 ->> 3 == 1
0 ->> 1 == 2
2 ->> 1 == 2
The Cost for created MST is : 5
```

You can verify this output's accuracy by comparing it with the MST structure shown above. The overall cost for this MST is 5.

The time complexity of this algorithm is O(E log E) or O(E log V), where E is a number of edges and V is a number of vertices.

WEEK 9	LONGEST COMMON SUBSEQUENCE

9.Aim:

To implement a dynamic algorithm to find longest common subsequence

<u>Procedure:</u>

The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

If S1 and S2 are the two given sequences then, Z is the common subsequence of S1 and S2 if Z is a subsequence of both S1 and S2. Furthermore, Z must be a **strictly increasing sequence** of the indices of both S1 and S2.
In a strictly increasing sequence, the indices of the elements chosen from the original sequences must be in ascending order in Z.
If

```
S1 = {B, C, D, A, A, C, D}
```

Then, {A, D, B} cannot be a subsequence of S1 as the order of the elements is not the same (ie. not strictly increasing sequence).

Let us understand LCS with an example.

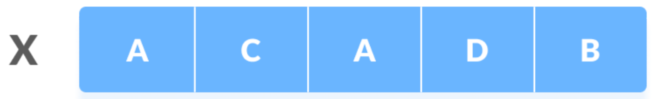
If

```
S1 = {B, C, D, A, A, C, D}

S2 = {A, C, D, B, A, C}
```

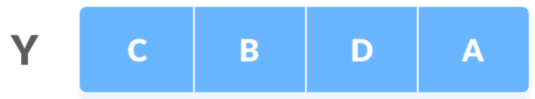
Then, common subsequences are {B, C}, {C, D, A, C}, {D, A, C}, {A, A, C}, {A, C}, {C, D}, ... Among these subsequences, {C, D, A, C} is the longest common subsequence. We are going to find this longest common subsequence using dynamic programming.

Using Dynamic Programming to find the LCS

Let us take two sequences:



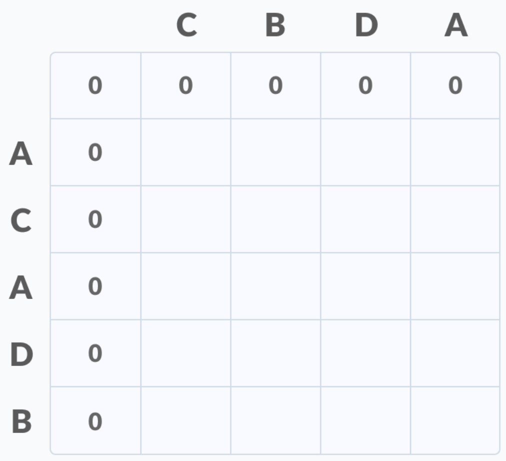
The first sequence



Second Sequence

The following steps are followed for finding the longest common subsequence.

1. Create a table of dimension n+1*m+1 where n and m are the lengths of X and Y respectively. The first row and the first column are filled with zeros.



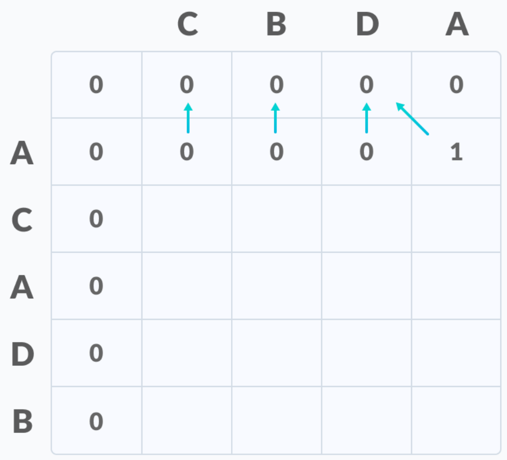
		C	B	D	A
	0	0	0	0	0
A	0				
C	0				
A	0				
D	0				
B	0				

Initialise a table

2. Fill each cell of the table using the following logic.

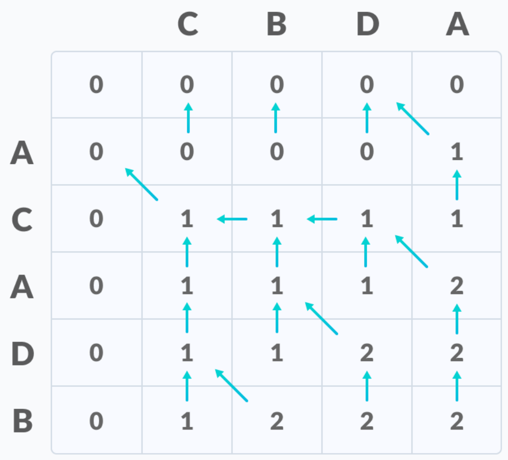
3. If the character correspoding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.

4. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.



		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0				
A	0				
D	0				
B	0				

Fill the values

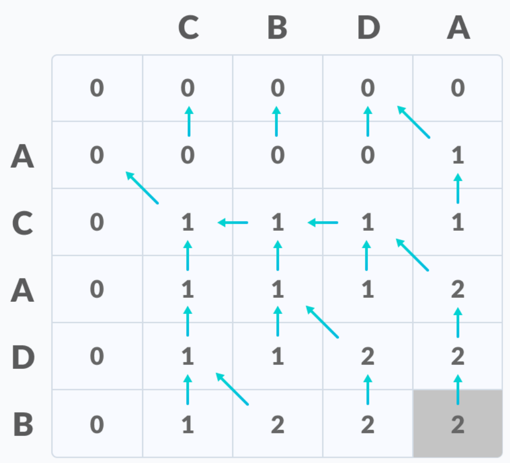


		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0	1	1	1	1
A	0	1	1	1	2
D	0	1	1	2	2
B	0	1	2	2	2

5. **Step 2** is repeated until the table is filled.

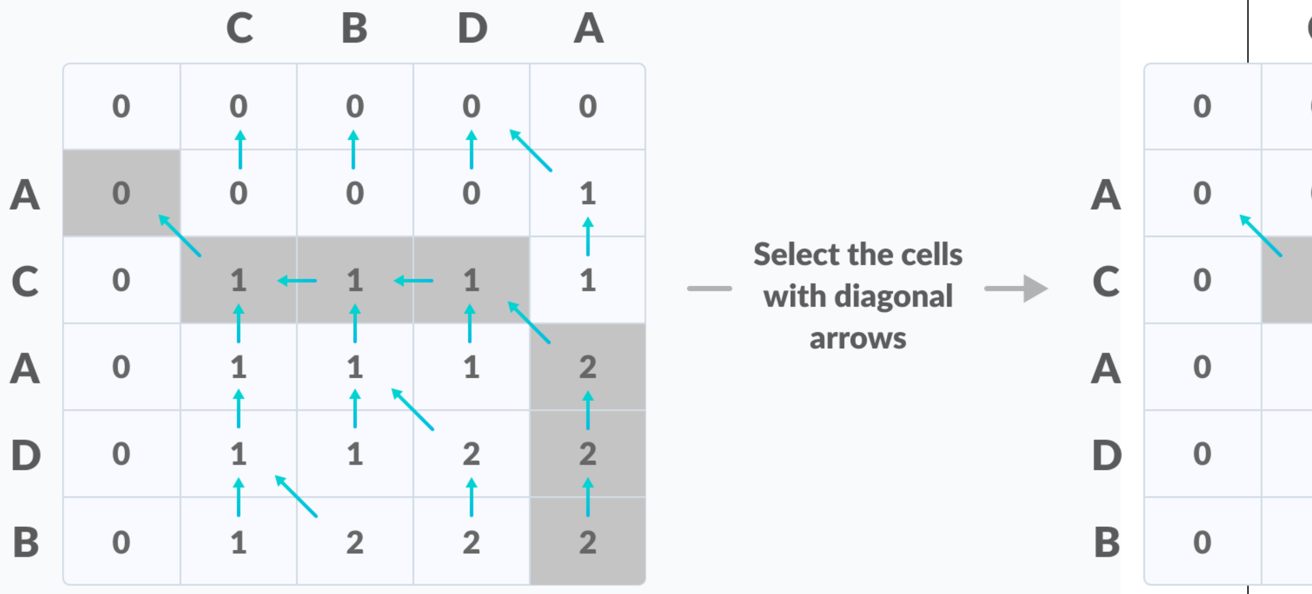
   Fill all the values

6. The value in the last row and the last column is the length of the longest



common subsequence. The bottom right corner is the length of the LCS

7. In order to find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to () symbol form the longest common subsequence.



Create a path according to the arrows

Thus, the longest common subsequence is CA.

LCS

How is a dynamic programming algorithm more efficient than the recursive algorithm while solving an LCS problem?

The method of dynamic programming reduces the number of function calls. It stores the result of each function call so that it can be used in future calls without the need for redundant calls.

In the above dynamic algorithm, the results obtained from each comparison between elements of X and the elements of Y are stored in a table so that they can be used in future computations.

So, the time taken by a dynamic approach is the time taken to fill the table (ie. O(mn)). Whereas, the recursion algorithm has the complexity of $2^{max(m, n)}$.

Longest Common Subsequence Algorithm

```
X and Y be two given sequences

Initialize a table LCS of dimension X.length * Y.length

X.label = X

Y.label = Y

LCS[0][] = 0

LCS[][0] = 0
```

```
Start from LCS[1][1]

Compare X[i] and Y[j]

  If X[i] = Y[j]

    LCS[i][j] = 1 + LCS[i-1, j-1]

    Point an arrow to LCS[i][j]

  Else

    LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])

    Point an arrow to max(LCS[i-1][j], LCS[i][j-1])
```

```c
// The longest common subsequence in C

#include <stdio.h>
#include <string.h>

int i, j, m, n, LCS_table[20][20];
char S1[20] = "ACADB", S2[20] = "CBDA", b[20][20];

void lcsAlgo() {
  m = strlen(S1);
  n = strlen(S2);

  // Filling 0's in the matrix
  for (i = 0; i <= m; i++)
    LCS_table[i][0] = 0;
  for (i = 0; i <= n; i++)
    LCS_table[0][i] = 0;

  // Building the mtrix in bottom-up way
  for (i = 1; i <= m; i++)
    for (j = 1; j <= n; j++) {
      if (S1[i - 1] == S2[j - 1]) {
        LCS_table[i][j] = LCS_table[i - 1][j - 1] + 1;
      } else if (LCS_table[i - 1][j] >= LCS_table[i][j - 1]) {
        LCS_table[i][j] = LCS_table[i - 1][j];
      } else {
        LCS_table[i][j] = LCS_table[i][j - 1];
      }
    }

  int index = LCS_table[m][n];
  char lcsAlgo[index + 1];
  lcsAlgo[index] = '\0';

  int i = m, j = n;
  while (i > 0 && j > 0) {
    if (S1[i - 1] == S2[j - 1]) {
      lcsAlgo[index - 1] = S1[i - 1];
      i--;
      j--;
      index--;
    }

    else if (LCS_table[i - 1][j] > LCS_table[i][j - 1])
      i--;
    else
      j--;
  }

  // Printing the sub sequences
  printf("S1 : %s \nS2 : %s \n", S1, S2);
  printf("LCS: %s", lcsAlgo);
}

int main() {
  lcsAlgo();
  printf("\n");
}
```

Time Complexity

- Worst case time complexity: **O(n*m)**
- Average case time complexity: **O(n*m)**
- Best case time complexity: **O(n*m)**
- Space complexity: **O(n*m)**

Since we are using two for loops for both the strings ,therefore the time complexity of finding the longest common subsequence using dynamic programming approach is **O(n * m)** where n and m are the lengths of the strings.

WEEK 10	N QUEEN's PROBLEM

N-Queen

N - Queens problem is to find an arrangement of N queens on a n x n chess board, such that no queen can attack any other queens on the board. The chess queens can attack in any direction as horizontal, vertical, horizontal, and diagonal way. A binary matrix is used to display the positions of N Queens, where no queens can attack other queens.

Pseudo Code:

```
Place (k, i)
  {
   For j ← 1 to k - 1
    do if (x [j] = i)
     or (Abs x [j]) - i) = (Abs (j - k))
   then return false;
    return true;
}

N - Queens (k, n)
{
  For i ← 1 to n
      do if Place (k, i) then
  {
    x [k] ← i;
    if (k ==n) then
      write (x [1....n]);
    else
    N - Queens (k + 1, n);
  }
}
```

Application of N-Queen Problem:

It is used in VLSI testing, traffic control, parallel memory storage schemes, and deadlock prevention.

WEEK 11	TRAVELLING SALESMAN PROBLEM

Travelling Salesman Problem Using Branch and Bound

Travelling Salesman Problem is based on a real-life scenario, where a salesman from a company must start from his own city and visit all the assigned cities exactly once and return to his home till the end of the day. The exact problem statement goes like this,
"Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Pseudo Code:

```
reduce_row():
/// row[i] is the minimum value for the ith row
for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            if (matrix_reduced[i][j] < row[i])
                row[i] = matrix_reduced[i][j];

/// subtracting the minimum value and reducing the matrix
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        if (matrix_reduced[i][j] != INF && row[i] != INF)
            matrix_reduced[i][j] -= row[i];

reduce_cloumn():
fill_n(col, N, INF);

/// row[i] is the minimum value for the ith row
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        if (matrix_reduced[i][j] < col[j])
            col[j] = matrix_reduced[i][j];

///for reducing the matrix
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        if (matrix_reduced[i][j] != INF && col[j] != INF)
            matrix_reduced[i][j] -= col[j];
Cost_calculation():
int cost_calculation(int matrix_reduced[N][N])
{
    /// setting the initial cost as 0
    int cost = 0;

    /// reducing the row
    int row[N];
    reduce_row(matrix_reduced, row);
    /// reducing the column
    int col[N];
    reduce_column(matrix_reduced, col);

    /// main cost calculation
    for (int i = 0; i < N; i++)
        cost += (row[i] != INT_MAX) ? row[i] : 0,
            cost += (col[i] != INT_MAX) ? col[i] : 0;

    return cost;
}
```

Application of TSP:

It is used in network design, and transportation route design. The objective is to minimize the distance.

WEEK 12	BFS AND DFS IMPLEMENTATION WITH ARRAY

1. DFS

The Depth First Search (DFS) is an algorithm for traversing or searching tree or graph data structures which uses the idea of backtracking. It explores all the nodes by going forward if possible or uses backtracking. DFS can be implemented with recursion to do the traversal or implemented iteratively with a stack.

Pseudo Code:

```
DFS(G,v)   ( v is the vertex where the search starts )
      Stack S := {};   ( start with an empty stack )
      for each vertex u, set visited[u] := false;
      push S, v;
      while (S is not empty) do
         u := pop S;
         if (not visited[u]) then
            visited[u] := true;
            for each unvisited neighbour w of u
               push S, w;
         end if
      end while
   END DFS()
```

Applications of DFS: Detecting cycle in a graph

A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.

2. BFS

Breadth **First Search** (BFS) is an algorithm for traversing or searching tree or graph data structures. It explores all the nodes at the present depth before moving on to the nodes at the next depth level. Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

Pseudo Code:

```
vector<int> bfsOfGraph(int V, vector<int> adj[])
{
    vector<int> bfs_traversal;
    vector<bool> vis(V, false);
    for (int i = 0; i < V; ++i) {

        // To check if already visited
        if (!vis[i]) {
            queue<int> q;
            vis[i] = true;
            q.push(i);

            // BFS starting from ith node
            while (!q.empty()) {
                int g_node = q.front();
                q.pop();
                bfs_traversal.push_back(g_node);
                for (auto it : adj[g_node]) {
                    if (!vis[it]) {
                        vis[it] = true;
                        q.push(it);
                    }
                }
            }
        }
    }
    return bfs_traversal;
}
```

Application of BFS:

Broadcasting - Networking makes use of what we call as packets for communication. These packets follow a traversal method to reach various networking nodes. It is being used as an algorithm that is used to communicate broadcasted packets across all the nodes in a network.

WEEK 13	RANDOMIZED QUICK SORT

Randomized Quick Sort

Randomized quicksort solves this problem by first randomly shuffling the values in the array, and then running quicksort, using the first value as the pivot every time. Now, shuffling might produce a sorted array, and this pivot choice would then result in a slow O(n^2) quicksort. But, shuffling almost never produces sorted data. In fact, since you can re-arrange n values in n! ways (n factorial), the probability that one re-arrangement gives you sorted data is 1/n!, which is effectively zero for n bigger than 10.

Pseudo Code:

```
// Sorts an array arr[low..high]
randQuickSort(arr[], low, high)

1. If low >= high, then EXIT.

2. While pivot 'x' is not a Central Pivot.
  (i)   Choose uniformly at random a number from [low..high].
        Let the randomly picked number number be x.
  (ii)  Count elements in arr[low..high] that are smaller
        than arr[x]. Let this count be sc.
  (iii) Count elements in arr[low..high] that are greater
        than arr[x]. Let this count be gc.
  (iv)  Let n = (high-low+1). If sc >= n/4 and
        gc >= n/4, then x is a central pivot.

3. Partition arr[low..high] around the pivot x.

4. // Recur for smaller elements
   randQuickSort(arr, low, sc-1)

5. // Recur for greater elements
   randQuickSort(arr, high-gc+1, high)
```

Application of Quick Sort:

In an operating system, particularly a real-time one, you may wish to schedule tasks by their priorities and some other properties such as relative importance, expected execution time, etc.

WEEK 14	STRING MATCHING ALGORITHMS

String Matching algorithm

Rabin-Karp algorithm is an algorithm used for searching/matching patterns in the text using a hash function. Unlike Naive string-matching algorithm, it does not travel through every character in the initial phase rather it filters the characters that do not match and then performs the comparison. A sequence of characters is taken and checked for the possibility of the presence of the required string. If the possibility is found then, character matching is performed.

Pseudo Code:

```
    n = t.length
    m = p.length
    h = dm-1 mod q
    p = 0
    t0 = 0
    for i = 1 to m
  p = (dp + p[i]) mod q
  t0 = (dt0 + t[i]) mod q
 for s = 0 to n - m
   if p = ts
     if p[1.....m] = t[s + 1..... s + m]
        print "pattern found at position" s
  If s < n-m
     ts + 1 = (d (ts - t[s + 1]h) + t[s + m + 1]) mod q
```

Applications:

String matching strategies or algorithms provide key role in various real-world problems or applications. A few of its imperative applications are Spell Checkers, Spam Filters, Intrusion Detection System, Search Engines, Plagiarism Detection, Bioinformatics, Digital Forensics and Information Retrieval Systems.

WEEK 15	➢ **Implement any problem statement with two different algorithm design strategy that you have learnt.** ➢ **Compare and contrast with its time complexity analysis.** ➢ **Submit the same as report.**