Advanced Programming Lab

WEEK - 0

Aim:

1.) Write a program to print the Pascal's Triangle

Description:

In mathematics, Pascal's triangle is a triangular array of the binomial coefficients that arises in probability theory, combinatorics, and algebra. Pascal's triangle is an array of binomial coefficients. The top row is numbered as n=0, and in each row are numbered from the left beginning with k = 0. Each number is found by adding two numbers which are residing in the previous row and exactly top of the current cell.It is a triangular array of the binomial coefficients. Write a function that takes an integer value n as input and prints first n lines of the Pascal's triangle. Following are the first 6 rows of Pascal's Triangle.

To solve the problem, we need to first create an array of size N or numRows (input value). This array is used to store each of the rows expected in the output, so, for example, array[1] = [1,1]. In this array, the number of columns (say, numCols) is equal to the number of the i-th row + 1 (Since, 0-indexed), i.e., for 0-th row, numCols = 1. So, the number of columns is different for each row.

Next, we need to run a loop from i = 0 to numRows – 1 (inclusive) in order to store each row in our array. For each of iteration of this loop, we follow the below steps:

1.Create an array of size (i + 1) (For some languages such as C++, you need to create a 2D array at the start of the program and resize array[i] to (i + 1)).

2.Set the first and last value of array[i] to 1.

3.Run another loop from j = 1 to i – 1 (inclusive) and for each iteration put array[i][j] = array[i – 1][j – 1] + array[i – 1][j].

Program:

```
#include <iostream>
using namespace std;
int factorial(int n)
{
 if(n==0)
 {
    return 1;
 }
 else if(n==1)
 {
    return 1;
 }
 else
 {
    return (n*factorial(n-1));
 }
}
int main()
{
   int n,i,j;
   cout<<"Enter Number of lines : ";
   cin>>n;
   for(i=0;i<n;i++)
   {
      for(j=0;j<n-i;j++)
      {
         cout<<" ";
      }
      for(j=0;j<i+1;j++)
      {
         cout<<factorial(i)/(factorial(j)*factorial(i-j))<<" ";
      }
      cout<<endl;
   }
   return 0;
}
```

OUTPUT:

```
Enter Number of lines : 5
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

Aim:

2.) Write a program to print median ,when two arrays are given sort them individually and merge them then for the resulted array find out the median

Description:

Given two sorted arrays, a[] and b[], the task is to find the median of these sorted arrays, in O(log n + log m) time complexity, when n is the number of elements in the first array, and m is the number of elements in the second array.

This is an extension of the median of two sorted arrays of equal size problem. Here we handle arrays of unequal size also.

Median:

Median is the middle most value of the given array if the given array consists of an even number of elements then there are two medians of that array if the number of elements in the array is odd then the median is only one that is the middle most value.

Median is a statistical measure that determines the middle value of a dataset listed in ascending order (i.e., from smallest to largest value). The measure divides the lower half from the higher half of the dataset. Along with mean and mode, median is a measure of central tendency.

Although the mean is the most commonly used measure of central tendency for quantitative data, the median can be used instead if the data contains large outliers. The outliers generally skew the mean, while the median is not affected by extreme values. Sometimes the two measures are used simultaneously to determine the value that describes the central value the best.

Program:

We have many ways to find but let us discuss about 2 ways

Program:1.)

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

void print_vector(vector<int> vector1)
{
  for(auto i=vector1.begin();i!=vector1.end();i++)
  {
     cout<<*i<<" ";
  }
}

int main()
{
  int mid,n1,n2,i,a;
  vector<int> v1;
  vector<int> v2;
  cout<<"size of first vector: ";
  cin>>n1;
  cout<<"size of second vector: ";
  cin>>n2;
  cout<<"Enter First Vector Elements: ";
  for(i=0;i<n1;i++)
  {
    cin>>a;
    v1.push_back(a);
  }
  cout<< "Enter Second Vector Elements: ";
  for(i=0;i<n2;i++)
  {
    cin>>a;
    v2.push_back(a);
  }
  sort(v1.begin(), v1.end());
  cout<<"First Vector Elements:";
  print_vector(v1);
  sort(v2.begin(), v2.end());
  cout<< "\nSecond Vector Elements: ";
  print_vector(v2);
  cout<<endl;

  vector<int> vect1(v1.size() + v2.size());
  merge(v1.begin(),v1.end(),v2.begin(),v2.end(),vect1.begin());
  cout<<"Vector elements after merging ";
  print_vector(vect1);
  cout<<endl;

  if((n1+n2)%2!=0)
  {
  nth_element(vect1.begin(), vect1.begin() + vect1.size()/2, vect1.end());
   cout << "The median is " << vect1[vect1.size()/2] << '\n';
  }
  else
  {
   nth_element(vect1.begin(), vect1.begin() + (vect1.size()/2)-1, vect1.end());
   cout << "The medians are "<<vect1[(vect1.size()/2)-1]<<" ";
   nth_element(vect1.begin(), vect1.begin() + vect1.size()/2, vect1.end());
   cout<< vect1[vect1.size()/2]<< '\n';
  }
   return 0;
}
```

OUTPUT:

1.)

```
size of first vector: 3
size of second vector: 4
Enter First Vector Elements: 1 6 8
Enter Second Vector Elements: 2 5 7 9
First Vector Elements:1 6 8
Second Vector Elements: 2 5 7 9
Vector elements after merging 1 2 5 6 7 8 9
The median is 6
```

2.)

```
size of first vector: 4
size of second vector: 4
Enter First Vector Elements: 1 5 9 6
Enter Second Vector Elements: 2 3 4 7
First Vector Elements:1 5 6 9
Second Vector Elements: 2 3 4 7
Vector elements after merging 1 2 3 4 5 6 7 9
The medians are 4 5
```

OR

Program:2.)

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

int getMedian(int ar1[], int ar2[], int n, int m)
{
        int i = 0;
        int j = 0;
        int count;
        int m1 = -1, m2 = -1;
        for (count = 0; count <= (m + n)/2; count++)
        {
                m2=m1;
                if(i != n && j != m)
                {
                        m1 = (ar1[i] > ar2[j]) ? ar2[j++] : ar1[i++];
                }
                else if(i < n)
                {
                        m1 = ar1[i++];
                }
                else
                {
                        m1 = ar2[j++];
                }
        }

        if((m + n) % 2 == 1){
                return m1;
        }
```

```
	else{
		return (m1+m2)/2;
	}
}

int main()
{
	int ar1[] = {900};
	int ar2[] = {5,8,10,20};
	int n1 = sizeof(ar1)/sizeof(ar1[0]);
	int n2 = sizeof(ar2)/sizeof(ar2[0]);
	cout << getMedian(ar1, ar2, n1, n2);
}
```

OUTPUT:

```
10
```

WEEK - 1

Aim:

1.)A permutation of an array of integers is an arrangement of its members into a sequence or linear order.

- For example, for arr = [1,2,3], the following are considered permutations of arr: [1,2,3], [1,3,2], [3,1,2], [2,3,1].

The next permutation of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the next permutation of that array is the permutation that follows it in the sorted container. If such an arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

- For example, the next permutation of arr = [1,2,3] is [1,3,2].
- Similarly, the next permutation of arr = [2,3,1] is [3,1,2].
- While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.

Given an array of integers nums, find the next permutation of nums.

The replacement must be in place and use only constant extra memory.

Description:

Lexicographic order

In mathematics, the lexicographic or lexicographical order is a generalization of the alphabetical order of the dictionaries to sequences of ordered symbols or, more generally, of elements of a totally ordered set. There are several variants and generalizations of the lexicographical ordering

lexicographically nth permutation

Idea behind printing n-th permutation is quite simple:

we should use STL (explained in above link) for finding the next permutation and do it till the nth permutation. After n-th iteration, we should break from the loop and then print the string which is our nth permutation.

lexicographically next permutation

The lexicographically next permutation is basically the greater permutation. For example, the next of “ACB” will be “BAC”. In some cases, the lexicographically next permutation is not present, like “BBB” or “DCBA” etc. In C++ we can do it by using a library function called next_permutation().

Program 1:

```
//Next permutation

#include <iostream>
#include <algorithm>
using namespace std;
int main()
{
   int arr[10], n,i;
   cin>>n;
   for(i=0;i<n;i++)
   {
      cin>>arr[i];
   }
   cout<<"Entered list:";
   cout<<"[";
   for (int i = 0; i < n; i++)
        {
           if(i<n-1)
           {
                cout << arr[i]<<",";
           }
           else
           {
           cout << arr[i];
           }
        }
        cout<<"]"<<endl;
```

```
    next_permutation(arr,arr+n);
    cout<<"Output : ";
    cout<<"[";
        for (int i = 0; i < n; i++)
        {
            if(i<n-1)
            {
                cout << arr[i]<<",";
            }
            else
            {
            cout << arr[i];
            }
        }
        cout<<"]";

        return 0;
}
```

OUTPUT:

```
3
1 3 2
Entered list:[1,3,2]
Output : [2,1,3]
```

Program 2:

```
//Nth permutation

#include <iostream>
#include <algorithm>
using namespace std;
string firstpermutation(string &str)
{
string s;
sort(str.begin(), str.end());
s=str;
return s;
}
int main()
{
   string first, s;
   cout<<"Enter a permutation: ";
   cin>>s;
   int key;
   cout<<"Enter the n value : ";
   cin>>key;
   first=firstpermutation(s);
   cout<<first<<endl;
   for(int i = 1; i<=key; i++)
   {
      bool val = next_permutation(first.begin(), first.end());
      if (val == false)
      {
        cout<<"No such permutation";
         break;
      }
      else
      {
      if(i==key-1)
      {
      cout <<"nth permutation :"<<first<< endl;
      }
      }
   }
   return 0;
}
```

OUTPUT:

```
Enter a permutation: 231
Enter the n value : 5
nth permutation :312
```

WEEK - 2

Aim:
1.) Given an input string s and a pattern p, implement regular expression matching with support for '.' and '*' where:

- '.' Matches any single character.
- '*' Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

Description:

It is a standardized way to express patterns to be matched against sequences of characters. some of typical regex parameters are as shown below −
1.Target sequence (subject) − It is used to search for the sequence pattern.
2.Regular expression (pattern) − It is used to search for in the target sequence.
3.Matches array − Matches information is stored in one of the special match_results array types (such as cmatch or smatch).
4.Replacement strin − This operation replaces the matches.

SNO	Regex operation and description
1	regex_matchIt is a match sequence.
2	Regex_search It is a search sequence.
3	Regex_replace It is a replace matched sequence.

These classes encapsulate a regular expression and the results of matching a regular expression within a target sequence of characters.
basic_regex: Regular expression object (class template)
sub_match: Identifies the sequence of characters matched by a sub-expression (class template)
match_results: Identifies one regular expression match, including all sub-expression matches (class template)

Program:

```
#include <iostream>
#include <regex>

using namespace std;
int main()
{
   string s,p;
   cout<<"Enter string: ";
   cin>>s;
   cout<<"Enter pattern : ";
   cin>>p;
   regex  b( p );

   if ( regex_match(s,b) )
   {
      cout << "TRUE";
   }
   else
   {
      cout<<"FALSE";
   }
   return 0;
}
```

OUTPUT:

```
Enter string: mahita
Enter pattern : ma.*
TRUE
```

WEEK - 3

Aim:

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

Example:
Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 9
Output: True
There is a subset (4, 5) with sum 9.

Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 30
Output: False
There is no subset that add up to 30.

Description:

Method 1:

Recursion.

Approach: For the recursive approach we will consider two cases.

1. Consider the last element and now the required sum = target sum – value of ‘last’ element and number of elements = total elements – 1
2. Leave the ‘last’ element and now the required sum = target sum and number of elements = total elements – 1

Following is the recursive formula for the isSubsetSum() problem.

Let’s take a look at the simulation of above approach-:

set[]={3, 4, 5, 2}

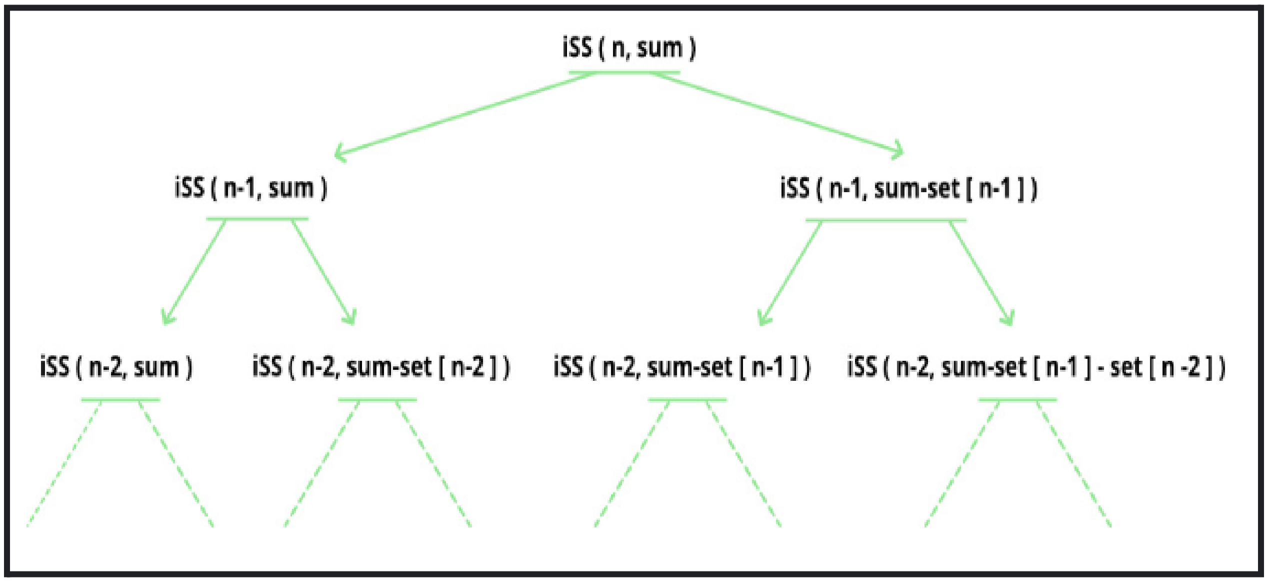
sum=9

(x, y)= 'x' is the left number of elements,

'y' is the required sum

```
               (4, 9)
              {True}
            /        \
         (3, 6)       (3, 9)

        /     \        /   \
     (2, 2)   (2, 6)   (2, 5)  (2, 9)
     {True}
     /   \
  (1, -3) (1, 2)
{False}  {True}
         /     \
       (0, 0)   (0, 2)
       {True}  {False}
```


iSS (n, sum)
iSS (n-1, sum)
iSS (n-1, sum-set [n-1])
iSS (n-2, sum)
iSS (n-2, sum-set [n-2])
iSS (n-2, sum-set [n-1])
iSS (n-2, sum-set [n-1] - set [n -2])

Program:

```
#include <iostream>
using namespace std;

bool isSubsetSum(int set[], int n, int sum)
{

        if (sum == 0)
                return true;
        if (n == 0)
```

```
                return false;

        if (set[n - 1] > sum)
                return isSubsetSum(set, n - 1, sum);

        return isSubsetSum(set, n - 1, sum)|| isSubsetSum(set, n - 1, sum - set[n -1]);
}

int main()
{
        int set[20];
        int i,n,sum;
        cout<<"Enter the size of array :";
        cin>>n;
        cout<<"Enter array elements :";
        for(i=0;i<n;i++)
        {
           cin>>set[i];
        }
        cout<<"Enter a number as sum :";
        cin>>sum;
        if (isSubsetSum(set, n, sum) == true)
                cout <<"True";
        else
                cout <<"False";
        return 0;
}
```

OUTPUT:

1.)

```
Enter the size of array :5
Enter array elements :3 4 5 1 2
Enter a number as sum :6
True
```

2.)

```
Enter the size of array :6
Enter array elements :3 34 4 12 5 2
Enter a number as sum :9
True
```

WEEK - 4

Aim:

Given a rod of length n inches and an array of prices that includes prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces. For example, if the length of the rod is 8 and the values of different pieces are given as the following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6)

length	1	2	3	4	5	6	7	8
price	1	5	8	9	10	17	17	20

And if the prices are as following, then the maximum obtainable value is 24 (by cutting in eight pieces of length 1)

length	1	2	3	4	5	6	7	8
price	3	5	8	9	10	17	17	20

Description:

A naive solution to this problem is to generate all configurations of different pieces and find the highest-priced configuration. This solution is exponential in terms of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem and can efficiently be solved using Dynamic Programming.

1) Optimal Substructure:

We can get the best price by making a cut at different positions and comparing the values obtained after a cut. We can recursively call the same function for a piece obtained after a cut.

Let cutRod(n) be the required (best possible price) value for a rod of length n. cutRod(n) can be written as follows.

cutRod(n) = max(price[i] + cutRod(n-i-1)) for all i in {0, 1 .. n-1}

2) Overlapping Subproblems

The following is a simple recursive implementation of the Rod Cutting problem.

The implementation simply follows the recursive structure mentioned above.

Output

```
Maximum Obtainable Value is 22
```

Considering the above implementation, the following is the recursion tree for a Rod of length 4.

```
cR() ---> cutRod()

                        cR(4)

                 /       /

                 /       /

          cR(3)        cR(2)     cR(1)   cR(0)

          / |          /        |

         /  |    /       |

      cR(2) cR(1) cR(0) cR(1) cR(0) cR(0)

     /   |       |

    /    |       |

  cR(1) cR(0) cR(0)         cR(0)

   /

 /

CR(0)
```

In the above partial recursion tree, cR(2) is solved twice. We can see that there are many subproblems that are solved again and again. Since the same subproblems are called again, this problem has the Overlapping Subproblems property. So the Rod Cutting problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of the same subproblems can be avoided by constructing a temporary array val[] in a bottom-up manner.

Program:

```
#include <bits/stdc++.h>
#include <iostream>
#include <math.h>
using namespace std;
 int max(int a, int b)
{ return (a > b) ? a : b;
}

int cutRod(int price[], int index, int n)
{

   if (index == 0)
  {
      return n * price[0];
   }

   int notCut = cutRod(price,index - 1,n);
   int cut = INT_MIN;
   int rod_length = index + 1;

   if (rod_length <= n)
      cut = price[index]
            + cutRod(price,index,n - rod_length);

   return max(notCut, cut);
}
int main()
{
   int arr[] = { 1, 5, 8, 9, 10, 17, 17, 20 };
```

```
    int size = sizeof(arr) / sizeof(arr[0]);
    cout << "Maximum Obtainable Value is "<< cutRod(arr, size - 1, size);
    getchar();
    return 0;
}
```

OUTPUT:

```
Maximum Obtainable Value is 22
```

WEEK - 5

Aim:

Divide and Conquer Strategy :

Find the minimum and maximum element in the array using divide and conquer with recursion.

Description:

Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

Divide and Conquer algorithm consists of a dispute using the following three steps.

Divide the original problem into a set of subproblems.

Conquer: Solve every subproblem individually, recursively.

Combine: Put together the solutions of the subproblems to get the solution to the whole problem.

Recursion:

Recursion is a method,which calls itself directly or indirectly until a suitable condition is met. In this method, we repeatedly call the function within the same function, and it has a base case and a recursive condition. The recursive condition helps in the repetition of code again and again, and the base case helps in the termination of the condition.

We can easily solve this problem by using Divide and Conquer. The idea is to recursively divide the array into two equal parts and update the maximum and minimum of the whole array in recursion by passing minimum and maximum variables by reference. The base conditions for the recursion will be when the subarray is of length 1 or 2.

Given an integer array, find the minimum and maximum element present in it by making minimum comparisons by using the divide-and-conquer technique.

Program:

```
#include <iostream>

#include <vector>

#include <climits>

using namespace std;

void findMinAndMax(vector<int> const &nums, int low, int high, int &min, int
&max)
{
 if (low == high)
 {
      if (max < nums[low])
      {
         max = nums[low];
      }
 if (min > nums[high])
      {
         min = nums[high];
      }
return;
   }
 if (high - low == 1)
   {
      if (nums[low] < nums[high])
        {
         if (min > nums[low])
         {
            min = nums[low];
         }

         if (max < nums[high])
         {
            max = nums[high];
         }
}
```

```
        else {
            if (min > nums[high])
          {
                min = nums[high];
            }

            if (max < nums[low])
            {
                max = nums[low];
            }
        }
        return;
    }
  int mid = (low + high) / 2;
   findMinAndMax(nums, low, mid, min, max);
    findMinAndMax(nums, mid + 1, high, min, max);
}

int main()
{
    vector<int> nums = { 7, 2, 9, 3, 1, 6, 7, 8, 4 };
     int max = INT_MIN, min = INT_MAX;
    int n = nums.size();
    findMinAndMax(nums, 0, n - 1, min, max);

    cout << "The minimum array element is " << min << endl;
    cout << "The maximum array element is " << max << endl;
    return 0;
}
```

OUTPUT:

```
The minimum array element is 1
The maximum array element is 9
```

WEEK - 6

Aim:

Write a python program to find the repetitive substrings in the given string.

Description:

The bag-of-words model is a simplifying representation used in natural language processing and information retrieval. In this model, a text is represented as the bag of its words, disregarding grammar and even word order but keeping multiplicity. The bag-of-words model has also been used for computer vision.

In this article, we are going to discuss a Natural Language Processing technique of text modeling known as Bag of Words model. Whenever we apply any algorithm in NLP, it works on numbers. We cannot directly feed our text into that algorithm. Hence, Bag of Words model is used to preprocess the text by converting it into a bag of words, which keeps a count of the total occurrences of most frequently used words.

This model can be visualized using a table, which contains the count of words corresponding to the word itself.

Bag of Words:

Bag-of-Words is one of the most fundamental methods to transform tokens into a set of features. The BoW model is used in document classification, where each word is used as a feature for training the classifier. For example, in a task of review based sentiment analysis, the presence of words like ‘fabulous’, ‘excellent’ indicates a positive review, while words like ‘annoying’, ‘poor’ point to a negative review .

Bag of words is the technique used to extract the features from the features of what we want from a text or a document.

Program:

```
from collections import Counter

s=input("Enter the string :").split()

FreqDict=dict(Counter(s))

print("The repeated words in the given string are:")

print(list(filter(lambda x:FreqDict[x]>1,list(FreqDict.keys()))))
```

OR

```
string=input().split()
l=[]
lc=[]
for i in string:
   l.append(i)
for i in l:
   y=l.count(i)
   lc.append(y)
#print(lc,l)
ls=[]
for i in range(0,len(lc)):
   if(lc[i]>1):
      ls.append(l[i])
s=set(ls)
ss=list(s)
print(ss)
```

OUTPUT:

```
Enter the string :

Find the repetitive sub string in the given string ?

The repeated words in the given string are:

['the', 'string']
```

WEEK - 7

Aim:

Given an array of strings ,group of anagrams together. You can return the answer in any order.An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase ,typically using all the original letters exactly once.

Description:

A simple method is to create a Hash Table. Calculate the hash value of each word in such a way that all anagrams have the same hash value. Populate the Hash Table with these hash values. Finally, print those words together with the same hash values. A simple hashing mechanism can be a modulo sum of all characters. With modulo sum, two non-anagram words may have the same hash value. This can be handled by matching individual characters.

Following is another method to print all anagrams together. Take two auxiliary arrays, index array, and word array. Populate the word array with the given sequence of words. Sort each individual word of the word array. Finally, sort the word array and keep track of the corresponding indices. After sorting, all the anagrams cluster together. Use the index array to print the strings from the original array of strings.

Let us understand the steps with the following input Sequence of Words:

"cat", "dog", "tac", "god", "act"

1) Create two auxiliary arrays index[] and words[]. Copy all given words to words[] and store the original indexes in index[]

index[]: 0   1   2   3   4

words[]: cat dog tac god act

2) Sort individual words in words[]. Index array doesn’t change.

index[]:   0   1   2   3   4

words[]: act dgo act dgo act

3) Sort the words array. Compare individual words using strcmp() to sort

```
index:     0    2    4    1    3
words[]:  act  act  act  dgo  dgo
```

4) All anagrams come together. But words are changed in the words array. To print the original words, take the index from the index array and use it in the original array. We get

"cat tac act dog god"

Program:

```
#include <bits/stdc++.h>
using namespace std;
void printAnagrams(string arr[], int size)
{
        unordered_map<string, vector<string> > map;
        for (int i = 0; i < size; i++)
              {
                string word = arr[i];
                char letters[word.size() + 1];
                strcpy(letters, word.c_str());
                sort(letters, letters + word.size() + 1);
                string newWord = "";
                for (int i = 0; i < word.size() + 1; i++)
                 {
                        newWord += letters[i];
                }
                if (map.find(newWord) != map.end())
                 {
                        map[newWord].push_back(word);
                }
        else {
            vector<string> words;
            words.push_back(word);
            map[newWord] = words;
        }
    }
    unordered_map<string, vector<string> >::iterator it;
    for (it = map.begin(); it != map.end(); it++) {
        vector<string> values = map[it->first];
        if (values.size() > 1) {
            cout << "[";
            for (int i = 0; i < values.size() - 1; i++) {
                cout << values[i] << ", ";
            }
            cout << values[values.size() - 1];
            cout << "]";
        }
    }
}

int main()
{
    string arr[] = { "cat", "dog", "tac", "god", "act" };
    int size = sizeof(arr) / sizeof(arr[0]);

    printAnagrams(arr, size);

    return 0;
}
```

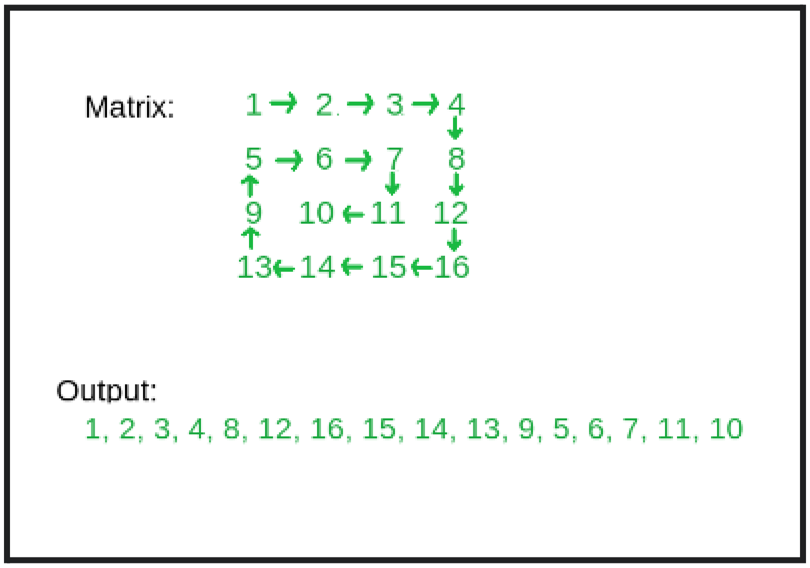
OUTPUT:

```
[dog, god][cat, tac, act]
```

WEEK - 8

Aim:

Given an m*n matrix, return all elements of the matrix in spiral order.



```
Output:
  1, 2, 3, 4, 8, 12, 16, 15, 14, 13, 9, 5, 6, 7, 11, 10
```

Description:

Let the array have R rows and C columns. seen[r] denotes that the cell on the r-th row and c-th column was previously visited. Our current position is (r, c), facing direction di, and we want to visit R x C total cells.

As we move through the matrix, our candidate's next position is (cr, cc). If the candidate is in the bounds of the matrix and unseen, then it becomes our next position; otherwise, our next position is the one after performing a clockwise turn.

The Spiral Matrix problem takes a 2-Dimensional array of N-rows and M-columns as an input, and prints the elements of this matrix in spiral order.

The spiral begins at the top left corner of the input matrix, and prints the elements it encounters, while looping towards the center of this matrix, in a clockwise manner.

Program:

```
#include <bits/stdc++.h>
using namespace std;
#define R 4
#define C 4

void spiralPrint(int m, int n, int a[R][C])
{
        int i, k = 0, l = 0;

        while (k < m && l < n) {
                for (i = l; i < n; ++i) {
                        cout << a[k][i] << " ";
                }
                k++;

                for (i = k; i < m; ++i) {
                        cout << a[i][n - 1] << " ";
                }
                n--;

                if (k < m) {
                        for (i = n - 1; i >= l; --i) {
                                cout << a[m - 1][i] << " ";
                        }
                        m--;
                }

                if (l < n) {
                        for (i = m - 1; i >= k; --i) {
                                cout << a[i][l] << " ";
```

```
                    }
                    l++;
            }
        }
}

int main()
{
        int a[R][C] = {{1, 2, 3, 4},
                            {5, 6, 7, 8},
                            {9, 10, 11, 12},
                            {13, 14, 15, 16}};

        spiralPrint(R, C, a);
        return 0;
}
```

OUTPUT:

```
1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10
```

WEEK - 9

Aim:

Given a sorted array of distinct integers and a target value, return the index if the target is found.ifnot, return the index where it would be if it were inserted in order.
You must write an algorithm with O(log n) runtime complexity.

Description:

Naive Approach: Follow the steps below to solve the problem:

1. Iterate over every element of the array arr[] and search for integer K.
2.If any array element is found to be equal to K, then print index of K.
3.Otherwise, if any array element is found to be greater than K, print that index as the insert position of K. If no element is found to be exceeding K, K must be inserted after the last array element.

A simple solution is to write our own custom routine for finding the index of the first occurrence of an element. The idea is to perform a linear search on the given array for determining the index.

Given a sorted array arr[] consisting of N distinct integers and an integer K, the task is to find the index of K, if it's present in the array arr[]. Otherwise, find the index where K must be inserted to keep the array sorted.

Examples:

Input: arr[] = {1, 3, 5, 6}, K = 5

Output: 1

Explanation: Since 5 is found at index 2 as arr[2] = 5, the output is 2.

Input: arr[] = {1, 3, 5, 6}, K = 2

Output: 2

Explanation: Since 2 is not present in the array but can be inserted at index 1 to make the array sorted.

There is another method to solve the problem i.e.,To find the position of an element in an array, you use the indexOf() method. This method returns the index of the first occurrence of the element that you want to find, or -1 if the element is not found. The following illustrates the syntax of the indexOf() method.

Program:

```
#include <iostream>
using namespace std;
int main()
{       int arr[10],i,n,k;
        cout<<"Enter the array size :";
        cin>>n;
        cout<<"Enter array elements"<<endl;
        for(i=0;i<n;i++)
        {
           cin>>arr[i];
        }
        cout<<"enter target value:"<<endl;
        cin>>k;
        for (int i = 0; i < n; i++)
        {       if (arr[i] == k)
                {
                        cout<<"The position is :"<<i<<endl;
                        break;
                }
                else if (arr[i] > k)
                {
                   cout<<"The position is :"<<i<<endl;
                   break;
                }
                else if( k>arr[n-1])
                {
                   cout<<"The position is :"<<n<<endl;
                   break;
                }
        }
                return 0;
}
```

OUTPUT:

```
Enter the array size :5
Enter array elements
1 4 6 8 9
enter target value:
6
The position is :2

Enter the array size :5
Enter array elements
1 2 3 4 9
enter target value:
8
The position is :4

Enter the array size :5
Enter array elements
1 2 3 5 9
enter target value:
11
The position is :5
```