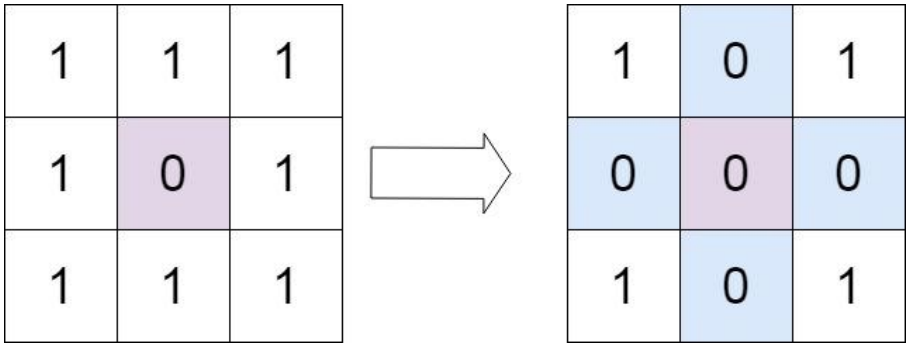
DAY 1

1. **Set matrix zeroes**

   Given an `m x n` integer matrix, if an element is `0`, set its entire row and column to `0`'s.

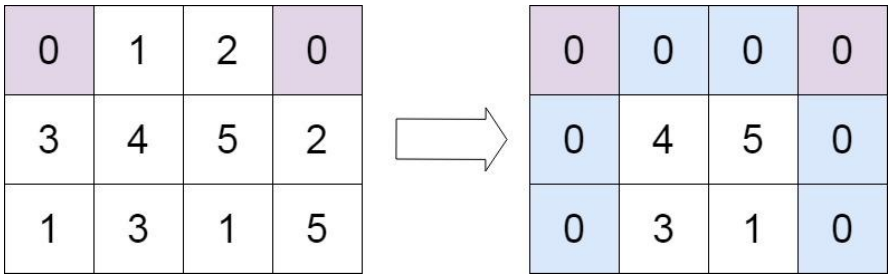
   You must do it in place.

   Example 1:




```
Input: matrix = [[1,1,1],[1,0,1],[1,1,1]]
Output: [[1,0,1],[0,0,0],[1,0,1]]
```

   Example 2:



```
Input: matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]
Output: [[0,0,0,0],[0,4,5,0],[0,3,1,0]]
```

Constraints:

- `m == matrix.length`
- `n == matrix[0].length`
- `1 <= m, n <= 200`
- `-2^31 <= matrix[i][j] <= 2^31 - 1`

Follow up:

- A straightforward solution using `O(mn)` space is probably a bad idea.
- A simple improvement uses `O(m + n)` space, but still not the best solution.
- Could you devise a constant space solution?

APPROACH 1

→ space complexity O(m+n)
→time complexity O(n*m+n*m)

```
class Solution {
public:
    void setZeroes(vector<vector<int>>& matrix) {

        int m=matrix.size();
        int n=matrix[0].size();
        vector<int> rows(m,1);
        vector<int> col(n,1);




        for(int i=0;i<m;i++){
            for(int j=0;j<n;j++){
                if(matrix[i][j]==0){
                    rows[i]=0;
                    col[j]=0;
                }
            }
        }

        for(int i=0;i<m;i++){
            for(int j=0;j<n;j++){
                matrix[i][j]*=rows[i]*col[j];
            }
        }

    }
};
```

APPROACH 2

→ Time complexity O(2*(m*n))
→ Space complexity O(1)

```
#include <bits/stdc++.h>
```

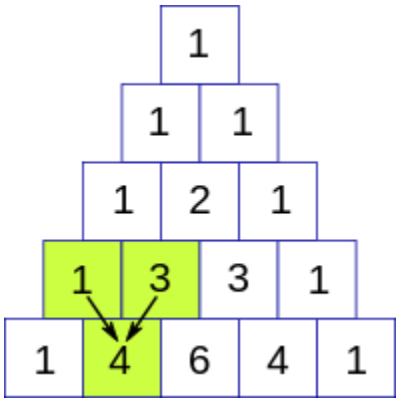
```
void setZeros(vector<vector<int>> &matrix)
{
        // Write your code here.
    int m=matrix.size();
    int n=matrix[0].size();
    int col=1;
    for(int i=0;i<m;i++)
        if(matrix[i][0]==0)
            col=0;

    for(int i=0;i<m;i++){
        for(int j=1;j<n;j++){
            if(matrix[i][j]==0){
                matrix[i][0]=0;
                matrix[0][j]=0;
            }
        }
    }
    for(int i=m-1;i>=0;--i){
        for(int j=n-1;j>=0;--j){
            if(j==0)
                matrix[i][j]*=col;
            else
                if(matrix[i][0]==0|| matrix[0][j]==0)
                    matrix[i][j]=0;
        }
    }


}
```

2. **Pascals Triangle**

Problem Statement

You are given an integer N. Your task is to return a 2-D ArrayList containing the pascal’s triangle till the row N.

A Pascal's triangle is a triangular array constructed by summing adjacent elements in preceding rows. Pascal's triangle contains the values of the binomial coefficient. For example in the figure below.



For example, given integer N= 4 then you have to print.

1

1 1

1 2 1

1 3 3 1

Here for the third row, you will see that the second element is the summation of the above two-row elements i.e. 2=1+1, and similarly for row three 3 = 1+2 and 3 = 1+2.

Input Format :

The first line of input contains an integer ‘T’ denoting the number of test cases.

The first line of each test case contains a single integer N denoting the row till which you have to print the pascal’s triangle.

Output Format :

For each test case, return the 2-D array/list containing the pascal’s triangle till the row N.

Note:

You do not need to print anything; it has already been taken care of. Just implement the given function.

Constraints:

1 <= T <= 40

1 <= N <= 50

Time Limit: 1 sec

Sample Input 1 :

3

1

2

3

Sample Output 1 :

1

1

1 1

1

1 1

1 2 1

APPROACH

Time Complexity O(n^2)

Space Complexity O(n^2)

```
class Solution {
public:
    vector<vector<int>> generate(int numRows) {
         vector<vector<int>> result(numRows);
    result[0].push_back(1);
    for(int i=1;i<numRows;i++)
    {
        result[i].push_back(1);
        for(int j=1;j<i;j++)
            result[i].push_back(result[i-1][j-1]+result[i-1][j]);
        result[i].push_back(1);
    }
    return result;


    }
};
```

OTHER PROBLEMS ON PASCAL TRIANGLE

→ Return the value at row ‘n’ of Pascal’s Triangle

Formula used: value of rth element in nth row= nCr

Space complexity O(N)

Time complexity O(N)

```
class Solution {
public:
    vector<int> getRow(int rowIndex) {


        vector <int> result;
        result.push_back(1);
        long long int x=0;
        for(int i=1;i<=rowIndex;i++)
        {
            x=result[i-1];
            x*=rowIndex-i+1;
            x/=i;
            result.push_back(x);
        }
```

```
            return result;
        }
    };
```

→ Return the value at row ‘n’ and column ‘m’ of Pascal’s Triangle

Using the formula : $^{\text{(row index)}}\mathbf{C}_{\text{(column index)}}$

$$^{N-1}C_{M-1}$$

3. **Next Permutation**

   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 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.

Example 1:

```
Input: nums = [1,2,3]
Output: [1,3,2]
```

Example 2:

```
Input: nums = [3,2,1]
Output: [1,2,3]
```

Example 3:

```
Input: nums = [1,1,5]
Output: [1,5,1]
```

Constraints:

- `1 <= nums.length <= 100`
- `0 <= nums[i] <= 100`

APPROACH 1

→ Space Complexity O(1)

→ Time Complexity O(n log n)

```
#include <bits/stdc++.h>
void swap(vector<int> &per, int ind1,int ind2)
{
    int temp=per[ind1];
    per[ind1]=per[ind2];
    per[ind2]=temp;
}

vector<int> nextPermutation(vector<int> &permutation, int n)
{
    //  Write your code here.
    int ind=-1;
    int nextInd=n-1;
    for(int i=n-1;i>0;i--)
    {
        if(permutation[i-1]<permutation[i])
        {
            ind=i;
            while(permutation[i-1]>=permutation[nextInd])
                nextInd--;
            swap(permutation,i-1,nextInd);
            break;
```

```
        }
    }
    if(ind==-1)
        ind=0;
    sort(permutation.begin()+ind, permutation.end());
    return permutation;
}
```

APPROACH 2

Instead of sorting reverse the elements

→ Space Complexity O(1)

→ Time Complexity O(n)

```
class Solution {
public:
    void swap(vector<int> &per, int ind1,int ind2)
    {
        int temp=per[ind1];
        per[ind1]=per[ind2];
        per[ind2]=temp;
    }
```

```
void reverse(vector <int> &v, int i,int j)
{
    while(i<j)
    {
        swap(v,i++,j--);
    }
}

void nextPermutation(vector<int>& permutation) {

//  Write your code here.
    int n=permutation.size();
    int ind=-1;
    int nextInd=n-1;
    for(int i=n-1;i>0;i--)
    {
        if(permutation[i-1]<permutation[i])
        {
            ind=i;
            while(permutation[i-1]>=permutation[nextInd])
                nextInd--;
            swap(permutation,i-1,nextInd);
```

```
                break;
            }
        }
        if(ind==-1)
            ind=0;
        reverse(permutation,ind,n-1);
    }
};
```

4. **Maximum Subarray Sum (Kadane's Algorithm)**

   Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return *its sum*.

   A subarray is a contiguous part of an array.

   Example 1:

   ```
   Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
   Output: 6
   Explanation: [4,-1,2,1] has the largest sum = 6.
   ```

   Example 2:

   ```
   Input: nums = [1]
   Output: 1
   ```

   Example 3:

   ```
   Input: nums = [5,4,-1,7,8]
   Output: 23
   ```

   Constraints:

- `1 <= nums.length <= 105`
- `-104 <= nums[i] <= 104`

Follow up: If you have figured out the `O(n)` solution, try coding another solution using the divide and conquer approach, which is more subtle.

BRUTE FORCE APPROACH:

```
#include <bits/stdc++.h>
long long maxSubarraySum(int arr[], int n)
{
    long long sum=0;
    long long mx=arr[0];
    for(int i=0;i<n;i++)
    {

        sum=0;
        for(int j=i;j<n;j++)
        {
            sum+=arr[j];
            mx=max(mx,sum);
        }
    }
    if(mx<0)
        mx=0;
    return mx;
}
```

APPROACH 2 (Kadane's Algorithm):

Time complexity O(n)

Space complexity O(1)

```
class Solution {

public:

    int maxSubArray(vector<int>& nums) {

        int n=nums.size();

        int sum=0;

        int mx=nums[0];  //if subarray should have at least 1 element.

        for(int i=0;i<n;i++)

        {

            sum+=nums[i];

            mx=max(sum,mx);

            if(sum<0)

                sum=0;

        }

        return mx;

    }

};
```

→ to print the subarray with maximum sum

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int n=nums.size();
        int sum=0;
        int mx=nums[0];
        int beg1=0,end1=0,tb=0,te=0;
        for(int i=0;i<n;i++)
        {
            te=i;
            sum+=nums[i];
            if(mx<sum)
            {
                beg1=tb;
                end1=te;
                mx=sum;
            }
            if(sum<0)
            {
```

```
                sum=0;
                tb=i+1;
            }
        }
        for(int i=beg1;i<=end1;i++)
            cout<<nums[i]<<" ";

        return mx;
    }
};
```

5. **Sort 0 1 2**

   Given an array `nums` with `n` objects colored red, white, or blue, sort them in-place so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

   We will use the integers `0`, `1`, and `2` to represent the color red, white, and blue, respectively.

   You must solve this problem without using the library's sort function.

   Example 1:

   ```
   Input: nums = [2,0,2,1,1,0]
   Output: [0,0,1,1,2,2]
   ```

   Example 2:

   ```
   Input: nums = [2,0,1]
   Output: [0,1,2]
   ```

   Constraints:

   - `n == nums.length`

- `1 <= n <= 300`
- `nums[i]` **is either** `0`**,** `1`**, or** `2`**.**

Follow up: Could you come up with a one-pass algorithm using only constant extra space?

APPROACH 1 (similar to dutch national flag algorithm)

Time complexity O(N)

Space complexity O(1)

```
class Solution {
public:
    void swap(vector<int> &arr,int a, int b)
    {
        int t=arr[a];
        arr[a]=arr[b];
        arr[b]=t;
    }
    void sortColors(vector<int>& arr) {
        int n=arr.size();
        int p1=0,p2=n-1;
        while(arr[p2]==2&&p1<p2)
                p2--;
        while(arr[p1]==0&&p1<p2)
                p1++;
        for(int i=p1;i<=p2;++i){
            while(arr[p2]==2&&i<p2)
                p2--;
            while(arr[p1]==0&&p1<i)
                p1++;
            //FOR FASTER OUTPUT
```

```
            if(arr[i]==2)
            {
                swap(arr,i,p2);
                i--;
                p2--;
            }

           else if(arr[i]==0)
            {
                swap(arr,p1,i);
                p1++;
            }
            else
                ;
        }
    }
};
```

6. **Stock buy and sell**

   You are given an array `prices` where `prices[i]` is the price of a given stock on the `ith` day.

   You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

   Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return `0`.

   Example 1:

   ```
   Input: prices = [7,1,5,3,6,4]
   Output: 5
   Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
   Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.
   ```

   Example 2:

   ```
   Input: prices = [7,6,4,3,1]
   Output: 0
   Explanation: In this case, no transactions are done and the max profit = 0.
   ```

Constraints:

- `1 <= prices.length <= 105`
- `0 <= prices[i] <= 104`

APPROACH 1 (BRUTE FORCE)

→Time Complexity O(N^2)

→Space Complexity O(1)

```
#include <bits/stdc++.h>
int maximumProfit(vector<int> &prices){
    // Write your code here.
    int n=prices.size();
    int mxp=0;


    int t;
    for(int i=0;i<n;++i)
    {
       for(int j=i+1;j<n;++j)
       {
           t=prices[j]-prices[i];
           //cout<<t<<"  ";
           mxp=max(mxp,t);
       }
        //cout<<endl;


  }
```

```
    return mxp;
}
```

APPROACH 2 (Similar to Kadane's Algorithm Approach)

→ Time Complexity O(N)

→ Space Complexity O(1)

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int n=prices.size();
        int mxp=0,temp=0;
        int buy=prices[0],sell;
        for(int i=1;i<n;++i)
        {
            sell=prices[i];
            temp=sell-buy;
            mxp=max(mxp,temp);
            if(temp<0)
            {
                buy=prices[i];
```

```
            }
          }
        return mxp;
    }
};
```

APPROACH 3 (Finding the minimum value for buying and calculating accordingly the profits made)

→ Time Complexity O(N)

→ Space Complexity O(1)

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int n=prices.size();
        int mxp=0;
        int buy=prices[0];
        for(int i=1;i<n;++i)
        {
            buy=min(buy,prices[i]);
            mxp=max(mxp,(prices[i]-buy));
```

```
            //cout<<endl;



        }

         return mxp;

    }

};
```