Majority Element II(Majority Element (>N/3 times))

Given an integer array of size `n`, find all elements that appear more than ⌊ `n/3` ⌋ times.

Example 1:

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

Output: [3]
```

Example 2:

```
Input: nums = [1]

Output: [1]
```

Example 3:

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

Output: [1,2]
```

Approach 1: Naïve

Traverse the array and count the elements occurrence

Time : O(n^2)

Space= O(1)

Approach 2 : HashMap

1. Add all elements with their frequencies into map
2. Traverse each element in the array and check its frequency is >n/3

```
public List<Integer> majorityElement(int[] nums) {

    int n=nums.length;

    HashMap<Integer,Integer> map=new HashMap<>();

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

    {

       map.put(nums[i],map.getOrDefault(nums[i],0)+1);

    }

    List<Integer> res=new ArrayList<>();

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

   {

      int count=map.get(nums[i]);
```

```
         if(count>n/3)

         {

            res.add(nums[i]);

            map.put(nums[i],0);

         }

      }

      return res;


   }
```

Time: O(n)

Space: O(n)

Approach 3: Moore's Voting algorithm

The idea is based on Moore's Voting algorithm. We first find two candidates. Then we check if any of these two candidates is actually a majority. Below is the solution for above approach.

```
static int appearsNBy3(int arr[], int n)
    {
        int count1 = 0, count2 = 0;

        // take the integers as the maximum
        // value of integer hoping the integer
        // would not be present in the array
        int first =  Integer.MIN_VALUE;;
        int second = Integer.MAX_VALUE;

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

            // if this element is previously
            // seen, increment count1.
            if (first == arr[i])
                count1++;

            // if this element is previously
            // seen, increment count2.
            else if (second == arr[i])
                count2++;

            else if (count1 == 0) {
                count1++;
                first = arr[i];
            }

            else if (count2 == 0) {
                count2++;
                second = arr[i];
            }

            // if current element is different
            // from both the previously seen
            // variables, decrement both the
            // counts.
            else {
                count1--;
                count2--;
            }
        }

        count1 = 0;
        count2 = 0;

        // Again traverse the array and
        // find the actual counts.
        for (int i = 0; i < n; i++) {
            if (arr[i] == first)
                count1++;

            else if (arr[i] == second)
                count2++;
        }

        if (count1 > n / 3)
            return first;

        if (count2 > n / 3)
            return second;

        return -1;
    }
```

Complexity Analysis:

- **Time Complexity**: **O(n)**
  First pass of the algorithm takes complete traversal over the array contributing to O(n) and another O(n) is consumed in checking if count1 and count2 is greater than floor(n/3) times.
- **Space Complexity: O(1)**
  As no extra space is required so space complexity is constant

Majority Element

Given an array `nums` of size `n`, return *the majority element*.

The majority element is the element that appears more than `⌊n / 2⌋` times. You may assume that the majority element always exists in the array.

Example 1:

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

Output: 3
```

Example 2:

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

Output: 2
```

Approach 1: Brute Force

Intuition

We can exhaust the search space in quadratic time by checking whether each element is the majority element.

Algorithm

The brute force algorithm iterates over the array, and then iterates again for each number to count its occurrences. As soon as a number is found to have appeared more than any other can possibly have appeared, return it.

```
public int majorityElement(int[] nums) {
    int majorityCount = nums.length/2;


    for (int num : nums) {
        int count = 0;
        for (int elem : nums) {
            if (elem == num) {
                count += 1;
            }
        }
```

```
            if (count > majorityCount) {
                return num;
            }

        }
        return -1;
    }
```

Complexity Analysis

- Time complexity : O(n^2)
- Space complexity : O(1)

Approach 2: HashMap

Intuition

We know that the majority element occurs more n}{2} \ $\lfloor 2n \rfloor$ times, and a `HashMap` allows us to count element occurrences efficiently.

Algorithm

We can use a `HashMap` that maps elements to counts in order to count occurrences in linear time by looping over `nums`. Then, we simply return the key with maximum value.

Complexity Analysis

- Time complexity : O(n)
- Space complexity : O(n)

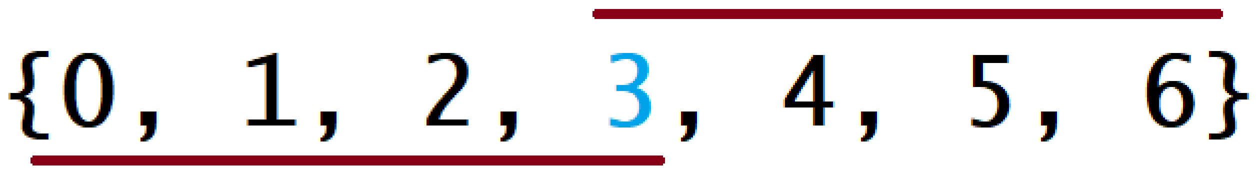
Approach 3: Sorting

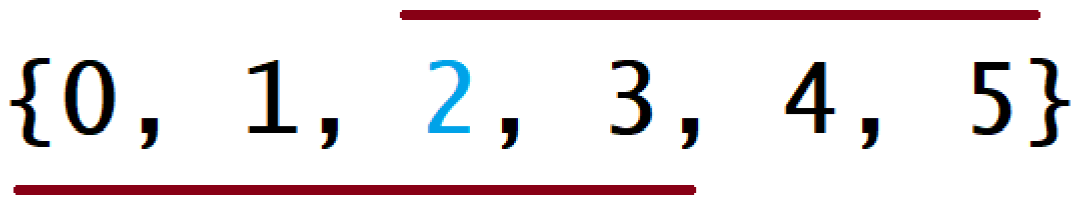
Intuition

If the elements are sorted in monotonically increasing (or decreasing) order, the majority element can be found at index {n}/{2 (and also {n}/{2} - 1, if n is even).

Algorithm

For this algorithm, we simply do exactly what is described: sort `nums`, and return the element in question. To see why this will always return the majority element (given that the array has one), consider the figure below (the top example is for an odd-length array and the bottom is for an even-length array):





```
class Solution {
   public int majorityElement(int[] nums) {
      Arrays.sort(nums);
      return nums[nums.length/2];
   }
}
```

Complexity Analysis

- Time complexity : $O(n\lg n)$
- Space complexity : $O(1)$ or ($O(n)$

Approach 4: Boyer-Moore Voting Algorithm

Intuition

If we had some way of counting instances of the majority element as $+1$ and instances of any other element as -1 summing them would make it obvious that the majority element is indeed the majority element.

```
public int majorityElement(int[] nums) {
       int count=1;
```

```
        int majorityElement=nums[0];

        for(int i=1;i<nums.length;i++)

        {

            if(count==0)

            {

                majorityElement=nums[i];

            }

            if(nums[i]==majorityElement)

            {

                count++;

            }

            else

            {

                count--;

            }

        }

        return majorityElement;

    }
```

Complexity Analysis

- Time complexity : O(n)
- Space complexity : O(1)

Pow(x, n)

Implement pow(x, n), which calculates `x` raised to the power `n` (i.e., x^n).

Example 1:

```
Input: x = 2.00000, n = 10

Output: 1024.00000
```

Example 2:

```
Input: x = 2.10000, n = 3

Output: 9.26100
```

Example 3:

```
Input: x = 2.00000, n = -2

Output: 0.25000

Explanation: 2^-2 = 1/2^2 = 1/4 = 0.25
```

- $-100.0 < x < 100.0$
- $-2^{31} <= n <= 2^{31}-1$
- $-10^4 <= x^n <= 10^4$

Navie solution:

```
public double myPow(double x, int n) {

     double res=1;

    long t=n;

     t=Math.abs(t);

     for(int i=1;i<=t;i++)

     {

         res=res*x;

     }

    return n>0?res:(1/res);

}
```

Time Complexity: O(n)
Space Complexity: O(1)

Efficient Approach:

Binary exponentiation

```
public double myPow(double x, int n) {
        double res=1;
       long t=n;
        t=Math.abs(t);
        while(t>0)
        {
            if((t&1)==1)
            {
                res=res*x;
            }
            x=x*x;
            t=t>>1;
        }
               return n>0?res:(1/res);

    }
```

Time Complexity: O(log|n|)
Auxiliary Space: O(1)

Reverse pairs

Given an integer array `nums`, return *the number of **reverse pairs** in the array*.

A reverse pair is a pair `(i, j)` where `0 <= i < j < nums.length` and `nums[i] > 2 * nums[j]`.

Example 1:

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

Output: 2
```

Example 2:

```
Input: nums = [2,4,3,5,1]

Output: 3
```

Solution 1: Brute Force Approach

Intuition :

As we can see from the given question that **i < j**, So we can just use 2 nested loops and check for the given condition which is arr [i] > 2* arr[j]**.**

Approach:

- We will be having 2 nested For loops the outer loop having i as pointer
- The inner loop with j as pointer and we will make sure that 0 <= i < j < arr.length() and also arr[i] > 2*arr[j] condition must be satisfied.

```
static int reversePairs(int arr[]) {

    int Pairs = 0;

    for (int i = 0; i < arr.length; i++) {

      for (int j = i + 1; j < arr.length; j++) {

        if (arr[i] > 2 * arr[j]) Pairs++;

      }

    }

    return Pairs;  }
```

Time Complexity: O (N^2) (Nested Loops)

Space Complexity: O(1)

Approach 2: Using Merge sort

Solution 2: Optimal Solution

Intuition:

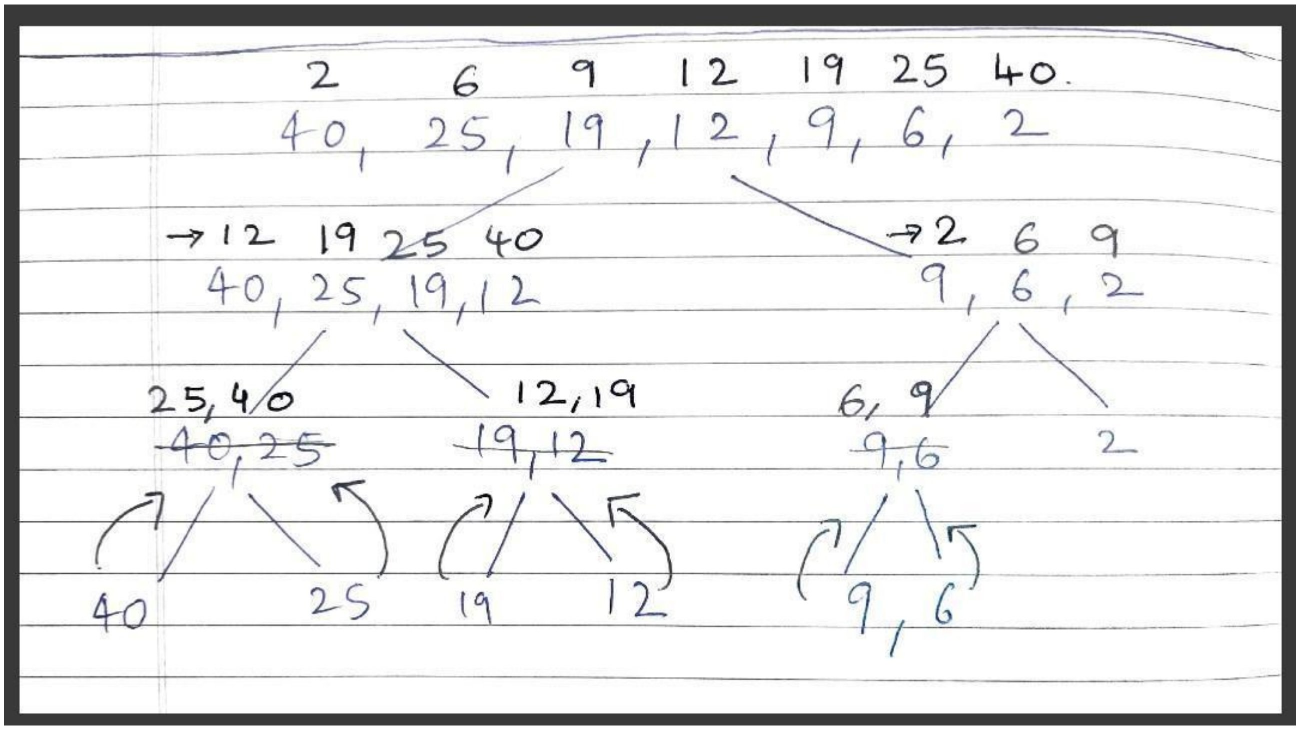
-> We will be using the Merge Sort Algorithm to solve this problem. We split the whole array into 2 parts creating a Merge Sort Tree-like structure. During the conquer step we do the following task :

-> We take the left half of the Array and Right half of the Array, both are sorted in themselves.

-> We will be checking the following condition arr[i] <= 2*arr[j] where i is the pointer in the Left Array and j is the pointer in the Right Array.

-> If at any point arr[i] <= 2*arr[j] , we add 1 to the answer as this pair has a contribution to the answer.

-> If Left Array gets exhausted before Right Array we keep on adding the distance j pointer traveled as both the arrays are Sorted so the next ith element from Left Subarray will equally contribute to the answer.



-> The moment when both Arrays get exhausted we perform a merge operation. This goes on until we get the original array as a Sorted array.

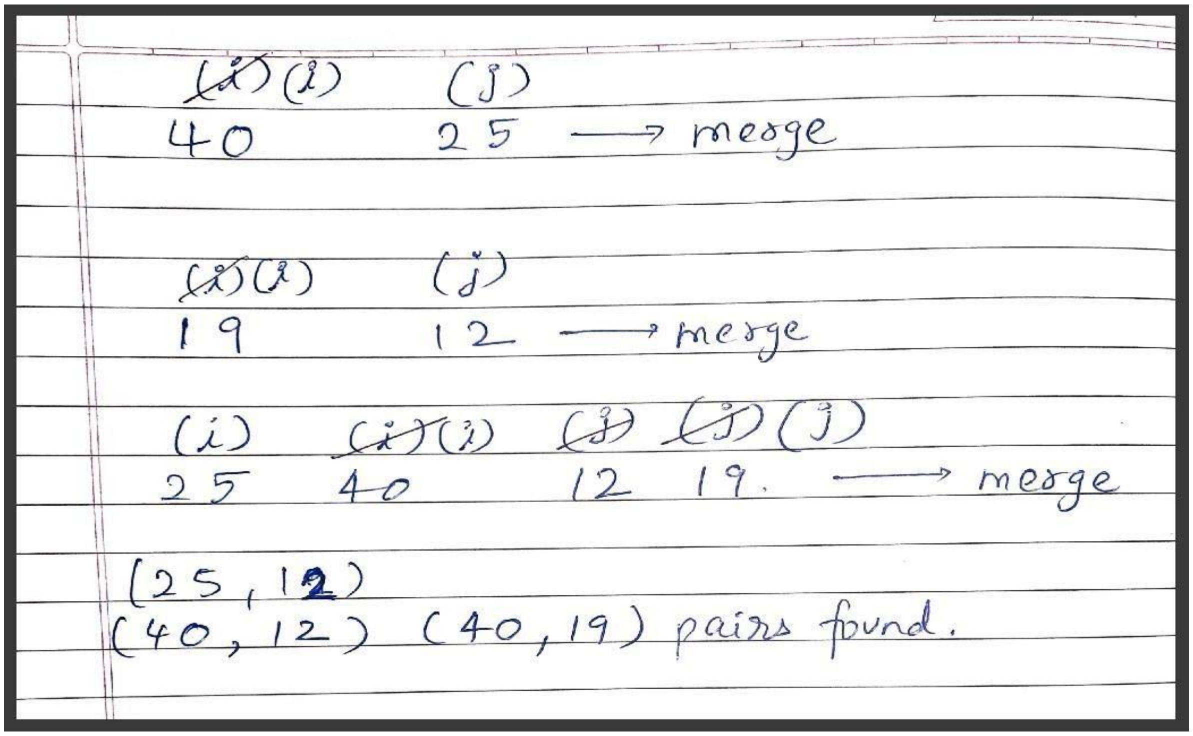
Approach :

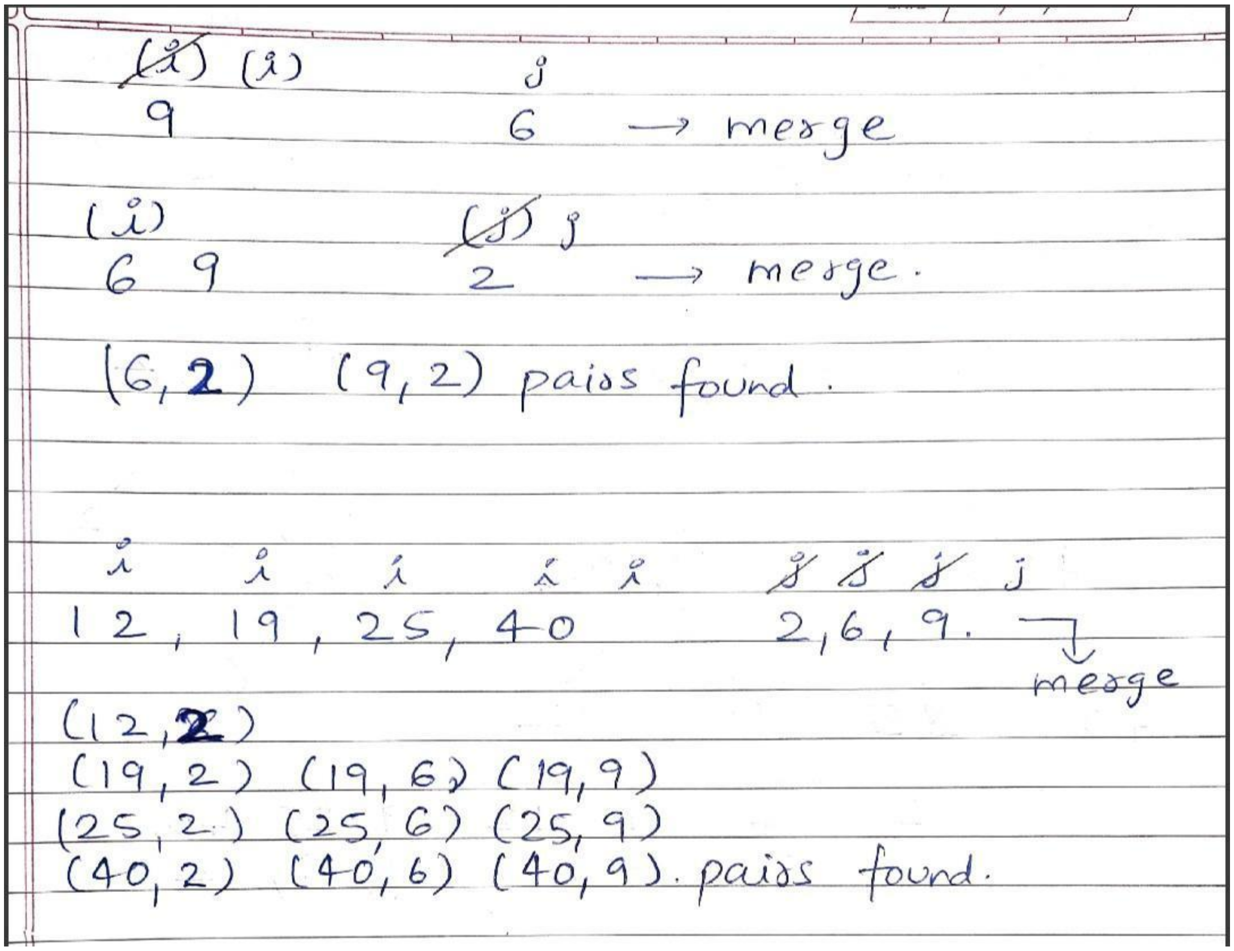
-> We first of all call a Merge Sort function, in that we recursively call Left Recursion and Right Recursion after that we call Merge function in order to merge both Left and Right Calls we initially made and compute the final answer.

-> In the Merge function, we will be using low, mid, high values to count the total number of inversion pairs for the Left half and Right half of the Array.

-> Now, after the above task, we need to Merge the both Left and Right sub-arrays using a temporary vector.

-> After this, we need to copy back the temporary vector to the Original Array. Then finally we return the number of pairs we counted.





~~(i)~~ (i) j

9 6 → merge

(i) ~~(j)~~ j

6 9 2 → merge.

(6, 2) (9, 2) pairs found.

i i i i i ~~j~~ ~~j~~ ~~j~~ j

12, 19, 25, 40 2, 6, 9. → merge

(12, 2)

(19, 2) (19, 6) (19, 9)

(25, 2) (25, 6) (25, 9)

(40, 2) (40, 6) (40, 9). pairs found.

Time Complexity : O(N log N) + O (N) + O (N)

Reason : O(N) – Merge operation , O(N) – counting operation (at each iteration of i , j doesn't start from 0 . Both of them move linearly)

Space Complexity : O(N)

Reason : O(N) – Temporary vector

Use the merge sort technique.

```
class Solution {
    public int reversePairs(int[] nums) {
        return merge(nums,0,nums.length-1);
    }
    public int merge(int[] arr,int l,int r)
    {
        int count=0;
        if(l<r)
        {
            int mid=(l+r)/2;
            count+=merge(arr,l,mid);
            count+=merge(arr,mid+1,r);
            count+=mergesort(arr,l,mid,r);

        }
        return count;
    }
    public int mergesort(int[] arr,int l,int mid,int r)
    {
        int j=mid+1;
        int count=0;
        for(int i=l;i<=mid;i++)
        {
            while(j<=r&&((long)arr[i]>(2*(long)arr[j])))
            {
                j++;
            }
// if a[i] and a[j] form a pair then a[i] and {a[mid+1] to a[j-1]} should also form a pair because
the elements are sorted due to mergesort
```

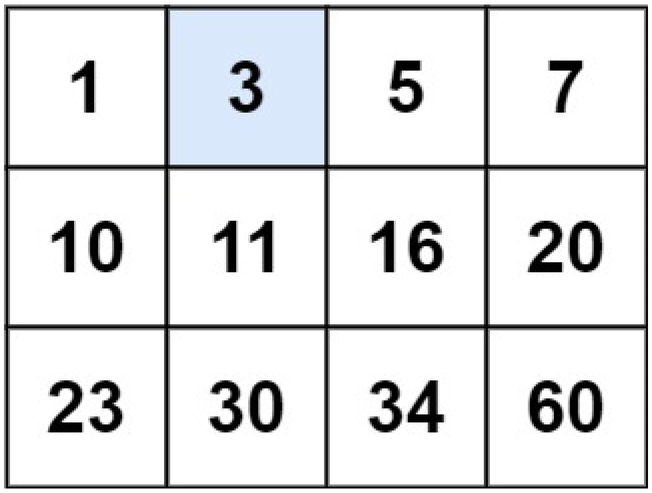
```
    count+= (j-(mid+1));
}
int[] left=Arrays.copyOfRange(arr,l,mid+1);
int[] right=Arrays.copyOfRange(arr,mid+1,r+1);
int i=0,k=l;
j=0;

while(i<left.length&&j<right.length)
{
    if(left[i]<right[j])
    {
        arr[k++]=left[i++];
    }
    else
    {
        arr[k++]=right[j++];
    }
}
while(i<left.length)
{
    arr[k++]=left[i++];
}
while(j<right.length)
{
    arr[k++]=right[j++];
```

```
        }
        return count;
    }
}
```

Search a 2D Matrix

Write an efficient algorithm that searches for a value `target` in an `m x n` integer matrix `matrix`. This matrix has the following properties:

- Integers in each row are sorted from left to right.
- The first integer of each row is greater than the last integer of the previous row.
- **Example 1:**



1	3	5	7
10	11	16	20
23	30	34	60

```
• Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3
• Output: true

Naive approach: The simple idea is to traverse the array and to search elements one by one.

Algorithm:

Run a nested loop, outer loop for row and inner loop for the column

Check every element with x and if the element is found then print "element found"

If the element is not found, then print "element not found".
```

```java
static boolean search(int[][] mat, int x)
 {
   for (int i = 0; i < mat.length; i++) {
     for (int j = 0; j < mat[i].length; j++)
       // if the element is found
       if (mat[i][j] == x) {
         return true;
       }
   }
   return false;
 }
```

Time Complexity: O(n2)

Auxiliary Space: O(1)

Efficient approach: The simple idea is to remove a row or column in each comparison until an element is found. Start searching from the top-right corner of the matrix. There are three possible cases.

1. **The given number is greater than the current number:** This will ensure that all the elements in the current row are smaller than the given number as the pointer is already at the right-most elements and the row is sorted. Thus, the entire row gets eliminated and continues the search for the next row. Here, elimination means that a row needs not be searched.

2. **The given number is smaller than the current number:** This will ensure that all the elements in the current column are greater than the given number. Thus, the entire column gets eliminated and continues the search for the previous column, i.e. the column on the immediate left.

3. **The given number is equal to the current number:** This will end the search.

Algorithm:

1. Let the given element be x, create two variable *i = 0, j = n-1* as index of row and column

2. Run a loop until i = n

3. Check if the current element is greater than x then decrease the count of j. Exclude the current column.

4. Check if the current element is less than x then increase the co

Time Complexity: O(n), Only one traversal is needed, i.e, i from 0 to n and j from n-1 to 0 with at most 2*n steps. The above approach will also work for m x n matrix (not only for n x n). Complexity would be O(m + n).
Auxiliary Space: O(1), No extra space is required.

```
public boolean searchMatrix(int[][] matrix, int target) {

    int n=matrix.length;

    int m=matrix[0].length;

    int i=n-1;

    int j=0;

    while(i>=0&&j<m)

    {

      if(matrix[i][j]==target)

      {

        return true;

      }

      else if(matrix[i][j]>target)

      {
            i--;
        }
        else
        {
           j++;
        }
    }
    return false;
}
```

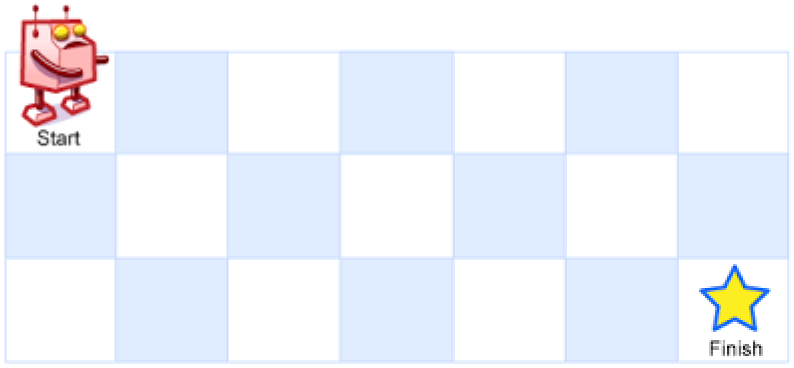
Unique Paths

There is a robot on an `m x n` grid. The robot is initially located at the **top-left corner** (i.e., `grid[0][0]`). The robot tries to move to the **bottom-right corner** (i.e., `grid[m - 1][n - 1]`). The robot can only move either down or right at any point in time.

Given the two integers `m` and `n`, return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

The test cases are generated so that the answer will be less than or equal to `2 * 10`^{9}.

Example 1:



```
Input: m = 3, n = 7

Output: 28
```

Approach 1:Brute force:

Try all the possible ways recursively

Time : O(2^n)

Space: O(m+n)

Approach 2: Dynamic programming

```
class Solution {

   public int uniquePaths(int m, int n) {

      int[][] map=new int[m+1][n+1];

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

      {

         Arrays.fill(map[i],-1);

      }
```

```
        int s=solve(m,n,1,1,map);

        return s;

    }
    public int solve(int m,int n,int i,int j,int[][] map)
    {
        if(i==m&&j==n)
        {
            return 1;
        }
        if(map[i][j]!=-1)
        {
            return map[i][j];
        }
        int count=0;
        if(i<m)
        {
            count+=solve(m,n,i+1,j,map);
        }
        if(j<n)
        {
            count+=solve(m,n,i,j+1,map);
        }
        return map[i][j]=count;
    }
}
```

Time : O(m*n)

Space: O(m*n)

Approcach 3:Combinatrics

In this approach We have to calculate m+n-2 C n-1 here which will be (m+n-2)! / (n-1)! (m-1)!

Now, let us see how this formula is giving the correct answer (Reference 1) (Reference 2) **read reference 1 and reference 2 for a better understanding**
m = number of rows

n = number of columns

Total number of moves in which we have to move down to reach the last row = m – 1 (m rows, since we are starting from (1, 1) that is not included)

Total number of moves in which we have to move right to reach the last column = n – 1 (n column, since we are starting from (1, 1) that is not included)

Down moves = (m – 1)
Right moves = (n – 1)
Total moves = Down moves + Right Moves = (m – 1) + (n – 1)
Now think moves as a string of 'R' and 'D' character where 'R' at any ith index will tell us to move 'Right' and 'D' will tell us to move 'Down'
Now think of how many unique strings (moves) we can make where in total there should be (n – 1 + m – 1) characters and there should be (m – 1) 'D' character and (n – 1) 'R' character?
Choosing positions of (n – 1) 'R' characters, results in automatic choosing of (m – 1) 'D' character positions

Calculate ^nC_r
Number of ways to choose positions for (n – 1) 'R' character = $^{\text{Total positions}}C_{n-1}$ = $^{\text{Total positions}}C_{m-1}$ = (n – 1 + m – 1) !=
Another way to think about this problem: Count the Number of ways to make an **N digit Binary String** (String with 0s and 1s only) with **'X' zeros** and **'Y' ones** (here we have replaced 'R' with '0' or '1' and 'D' with '1' or '0' respectively whichever suits you better)

```
static int numberOfPaths(int m, int n)
    {
        // We have to calculate m+n-2 C n-1 here
        // which will be (m+n-2)! / (n-1)! (m-1)!
        int path = 1;
        for (int i = n; i < (m + n - 1); i++) {
            path *= i;
            path /= (i - n + 1);
        }
        return path;
    }
```
Time:O(m-1) or O(n-1)

Space: O(1);