Search in a sorted 2D matrix

Problem Statement: Given an m*n 2D matrix and an integer, write a program to find if the given integer exists in the matrix.
Given 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:

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

target = 3

Output: true

Explanation: As the given integer(target) exists in the given 2D matrix, the function has returned true.
```

Example 2:

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

target = 13

Output: false

Explanation: As the given integer(target) does not exist in the given 2D matrix, the function has returned false.
```

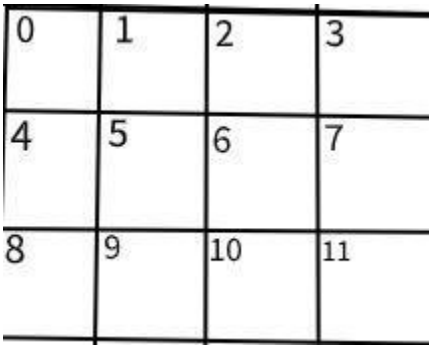
Solution

***Disclaimer**: Don't jump directly to the solution, try it out yourself first.*

Solution 1: Naive approach
Approach: We can traverse through every element that is present in the matrix and return true if we found any element in the matrix is equal to the target integer. If the traversal is finished we can directly return false as we did not find any element in the matrix to be equal to the target integer.
Time complexity: O(m*n)
Space complexity: O(1)
Solution 2: [Efficient] – Binary search

Intuition: As it is clearly mentioned that the given matrix will be row-wise and column-wise sorted, we can see that the elements in the matrix will be in a **monotonically increasing order.** So we can apply **binary search** to search the matrix. Consider the 2D matrix as a 1D matrix having indices from **0 to (m*n)-1** and apply binary search**.** Below the available image is the visual representation of the indices of 3*4 matrix.

Array Part III



Approach:

i) Initially have a low index as the first index of the considered 1D matrix(i.e: 0) and high index as the last index of the considered 1D matrix(i.e: (m*n)-1).

```
int low = 0;
int  high = (m*n)-1;
```

ii) Now apply binary search. Run a while loop with the condition **low<=high**. Get the middle index as **(low+high)/2**.We can get the element at middle index using **matrix[middle/m][middle%m]**.

```
while(low<=high)
    int middle = (low+high)/2;
```

iii) If the element present at the middle index is greater than the target, then it is obvious that the target element will not exist beyond the middle index. So shrink the search space by updating the **high index to middle-1**.

```
if(matrix[middle/m][middle%m]<target)
    high = middle-1;
```

iv) If the middle index element is lesser than the target, shrink the search space by updating the **low index to middle+1**.

```
if(matrix[middle/m][middle%m]>target)
    low = middle+1;
```

v) If the middle index element is equal to the target integer, return true.

```
if(matrix[middle/m][middle%m]==target)
    return true;
```

vi) Once the **loop terminates we can directly return false** as we did not find the target element.

Code:

```
bool searchMatrix(vector<vector<int>>& matrix, int target) {
    int lo = 0;
    if(!matrix.size()) return false;
    int hi = (matrix.size() * matrix[0].size()) - 1;

    while(lo <= hi) {
        int mid = (lo + (hi - lo) / 2);
        if(matrix[mid/matrix[0].size()][mid % matrix[0].size()] == target) {
            return true;
        }
        if(matrix[mid/matrix[0].size()][mid % matrix[0].size()] < target) {
            lo = mid + 1;
        }
        else {
            hi = mid - 1;
        }
    }
```

```
        return false;
    }
```

Time complexity: O(log(m*n))
Space complexity: O(1)

Implement Pow(x,n) | X raised to the power N

Problem Statement: Given a double x and integer n, calculate x raised to power n. Basically Implement pow(x, n).
Examples:
Example 1:

Input: x = 2.00000, n = 10

Output: 1024.00000

Explanation: You need to calculate 2.00000 raised to 10 which gives ans 1024.00000

Example 2:

Input: x = 2.10000, n = 3

Output: 9.26100

Explanation: You need to calculate 2.10000 raised to 3 which gives ans 9.26100

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*
Solution 1: Brute force approach

Approach: Looping from 1 to n and keeping a ans(double) variable. Now every time your loop runs, multiply x with ans. At last, we will return the ans.
Now if n is negative we must check if n is negative, if it is negative divide 1 by the and.
Code:

```
#include<bits/stdc++.h>
using namespace std;
double myPow(double x, int n) {
      double ans = 1.0;
      for (int i = 0; i < n; i++) {
        ans = ans * x;
      }
      return ans;
}
int main()
{
    cout<<myPow(2,10)<<endl;
}
```

Output: 1024
Time Complexity: O(N)
Space Complexity: O(1)

Solution 2: Using Binary Exponentiation

Approach: Initialize ans as 1.0 and store a duplicate copy of n i.e nn using to avoid overflow
Check if nn is a negative number, in that case, make it a positive number.
Keep on iterating until nn is greater than zero, now if nn is an odd power then multiply x with ans ans reduce nn by 1.
Else multiply x with itself and divide nn by two.
Now after the entire binary exponentiation is complete and nn becomes zero, check if n is a negative value we know the answer will be 1 by and.
Code:

```
#include<bits/stdc++.h>
using namespace std;
double myPow(double x, int n) {
  double ans = 1.0;
  long long nn = n;
  if (nn < 0) nn = -1 * nn;
  while (nn) {
    if (nn % 2) {
      ans = ans * x;
      nn = nn - 1;
    } else {
      x = x * x;
      nn = nn / 2;
    }
  }
  if (n < 0) ans = (double)(1.0) / (double)(ans);
  return ans;
}
int main() {
  cout << myPow(2, 10) << endl;
}
```

Output : 1024
Time Complexity: O(log n)
Space Complexity: O(1)

Find the Majority Element that occurs more than N/2 times

Problem Statement: Given an array of **N integers**, write a program to return an element that occurs more than **N/2** times in the given array. You may consider that such an element always exists in the array.
Example 1:
Input Format: N = 3, nums[] = {3,2,3}

Result: 3

Explanation: When we just count the occurrences of each number and compare with half of the size of the array, you will get 3 for the above solution.

nums = {3,2,3}
N = 3, so N/2 = 1 (floor value is taken)

Count of '3' in array = 2 -> Answer
Count of '2' in array = 1

Example 2:
Input Format: N = 7, nums[] = {2,2,1,1,1,2,2}

Result: 2

Explanation: After counting the number of times each element appears and comparing it with half of array size, we get 2 as result.

nums = {2,2,1,1,1,2,2}
N = 7, so N/2 = 3 (floor value is taken)

Count of '2' in array = 4 -> Answer
Count of '1' in array = 3

Example 3:
Input Format: N = 10, nums[] = {4,4,2,4,3,4,4,3,2,4}

Result: 4

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1 (Brute Force):
Check the count of occurrences of all elements of the array one by one. Start from the first element of the array and count the number of times it occurs in the array. If the *count* is greater than the floor of N/2 then return that element as the answer. If not, proceed with the next element in the array and repeat the process.
Time Complexity: O(N^2)
Space Complexity: O(1)
Solution 2 (Better):
Intuition: Use a better data structure to reduce the number of look up operations and hence the time complexity. Moreover, we have been calculating the count of the same element again and again – so we have to reduce that also.

Approach:

1. Use a hashmap and store as *(key,value)* pairs. (Can also use frequency array based on size of nums)
2. Here the *key* will be the element of the array and *value* will be the number of times it occurs.
3. Traverse the array and update the value of the key. Simultaneously check if the value is greater than **floor of N/2**.
    1. If yes, return the key
    2. Else iterate forward.

Time Complexity: O(N)-> Frequency array or O(N log N) -> HashMap

Space Complexity: O(N)

Solution 3 (Optimal):
Moore's Voting Algorithm

Intuition: The question clearly states that the *nums* array has a majority element. Since it has a majority element we can say definitely the count is more than **N/2**.
Majority element count **= N/2 + x;**
Minority/Other elements **= N/2 – x;**
Where x is the number of times it occurs after reaching the minimum value **N/2**.
Now, we can say that count of minority elements and majority element are equal upto certain point of time in the array. So when we traverse through the array we try to keep track of the count of elements and which element we are tracking. Since the majority element appears more than **N/2** times, we can say that at some point in array traversal we find the majority element.

Approach:

1. Initialize 2 variables:
    1. **Count** – for tracking the count of element
    2. **Element** – for which element we are counting
2. Traverse through *nums* array.
    1. If **Count** is *0* then initialize the current traversing integer of array as **Element**
    2. If the traversing integer of array and **Element** are same increase **Count** by 1
    3. If they are different decrease **Count** by 1
3. The integer present in **Element** is the result we are expecting
4.

Code:

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int count = 0;
        int candidate = 0;

        for (int num : nums) {
            if (count == 0) {
                candidate = num;
            }
            if(num==candidate) count += 1;
            else count -= 1;
        }

        return candidate;
    }
};
```

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

Majority Elements(>N/3 times) | Find the elements that appears more than N/3 times in the array

Problem Statement: Given an array of N integers. Find the elements that appears more than **N/3** times in the array. If no such element exists, return an empty vector.

Example 1:

Input: N = 5, array[] = {1,2,2,3,2}

Ouput: 2

Explanation: Here we can see that the Count(1) = 1, Count(2) = 3 and Count(3) = 1.Therefore, the count of 2 is greater than N/3 times. Hence, 2 is the answer.

Example 2:

Input: N = 6, array[] = {11,33,33,11,33,11}

Output: 11 33

Explanation: Here we can see that the Count(11) = 3 and Count(33) = 3. Therefore, the count of both 11 and 33 is greater than N/3 times. Hence, 11 and 33 is the answer.

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1: Brute-Force

Approach: Simply count the no. of appearance for each element using nested loops and whenever you find the count of an element greater than N/3 times, that element will be your answer.

Code:

```
#include <bits/stdc++.h>
using namespace std;
vector < int > majorityElement(int arr[], int n) {
  vector < int > ans;
  for (int i = 0; i < n; i++) {
    int cnt = 1;
    for (int j = i + 1; j < n; j++) {
      if (arr[j] == arr[i])
        cnt++;
    }

    if (cnt > (n / 3))
      ans.push_back(arr[i]);
  }

  return ans;
}

int main() {
  int arr[] = {1,2,2,3,2};
  vector<int> majority;
  majority = majorityElement(arr, 5);
```

```
  cout << "The majority element is" << endl;
  set < int > s(majority.begin(), majority.end());
  for (auto it: s) {
    cout << it << " ";
  }
}
```

Output:
The majority element is 2
Time Complexity: O(n^2)
Space Complexity: O(1)
Solution 2: Better Solution
Approach: Traverse the whole array and store the count of every element in a map. After that traverse through the map and whenever you find the count of an element greater than N/3 times, store that element in your answer.
Dry Run: Lets take the example of arr[] = {10,20,40,40,40}, n=5.
First, we create an unordered map to store the count of each element.
Now traverse through the array

1. Found 10 at index 0, increase the value of key 10 in the map by 1.
2. Found 20 at index 1, increase the value of key 20 in the map by 1.
3. Found 40 at index 2, increase the value of key 40 in the map by 1.
4. Found 40 at index 3, increase the value of key 40 in the map by 1.
5. Found 40 at index 4, increase the value of key 40 in the map by 1.

Now, Our map will look like this:
10 -> 1
20 ->1
40 ->3
Now traverse through the map,
We found that the value of key 40 is greater than 2 (N/3). So, 40 is the answer.
Code:

```
#include <bits/stdc++.h>
using namespace std;
vector < int > majorityElement(int arr[], int n) {
  unordered_map < int, int > mp;
  vector < int > ans;
  for (int i = 0; i < n; i++) {
    mp[arr[i]]++;
  }
  for (auto x: mp) {
    if (x.second > (n / 3))
      ans.push_back(x.first);
  }
  return ans;
}
int main() {
  int arr[] = {1,2,2,3,2};
  vector < int > majority;
  majority = majorityElement(arr, 5);
  cout << "The majority element is " << ;

  for (auto it: majority) {
    cout << it << " ";
  }}
```

Output: The majority element is: 2
Time Complexity: O(n)

Space Complexity: O(n)
Solution 3: Optimal Solution (Extended Boyer Moore's Voting Algorithm)
Approach + Intuition: In our code, we start with declaring a few variables:

- num1 and num2 will store our currently most frequent and second most frequent element.
- c1 and c2 will store their frequency relatively to other numbers.
- We are sure that there will be a max of 2 elements which occurs > N/3 times because there cannot be if you do a simple math addition.

Let, ele be the element present in the array at any index.

- if ele == num1, so we increment c1.
- if ele == num2, so we increment c2.
- if c1 is 0, so we assign num1 = ele.
- if c2 is 0, so we assign num2 = ele.
- In all the other cases we decrease both c1 and c2.

In the last step, we will run a loop to check if num1 or nums2 are the majority elements or not by running a for loop check.

Intuition: Since it's guaranteed that a number can be a majority element, hence it will always be present at the last block, hence, in turn, will be on nums1 and nums2. For a more detailed explanation, please watch the video below.
Code:

```
#include <bits/stdc++.h>
using namespace std;
vector < int > majorityElement(int nums[], int n) {
  int sz = n;
  int num1 = -1, num2 = -1, count1 = 0, count2 = 0, i;
  for (i = 0; i < sz; i++) {
    if (nums[i] == num1)
      count1++;
    else if (nums[i] == num2)
      count2++;
    else if (count1 == 0) {
      num1 = nums[i];
      count1 = 1;
    } else if (count2 == 0) {
      num2 = nums[i];
      count2 = 1;
    } else {
      count1--;
      count2--;
    }
  }
  vector < int > ans;
  count1 = count2 = 0;
  for (i = 0; i < sz; i++) {
    if (nums[i] == num1)
      count1++;
    else if (nums[i] == num2)
      count2++;
  }
  if (count1 > sz / 3)
    ans.push_back(num1);
  if (count2 > sz / 3)
    ans.push_back(num2);
  return ans;
}
```

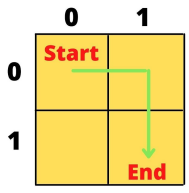
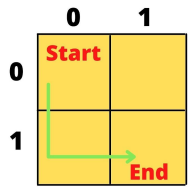
```
int main() {
  int arr[] = {1,2,2,3,2};
  vector < int > majority;
  majority = majorityElement(arr, 5);
  cout << "The majority element is ";

  for (auto it: majority) {
    cout << it << " ";
  }
}
```

Output:
The majority element is 2
Time Complexity: O(n)
Space Complexity: O(1)

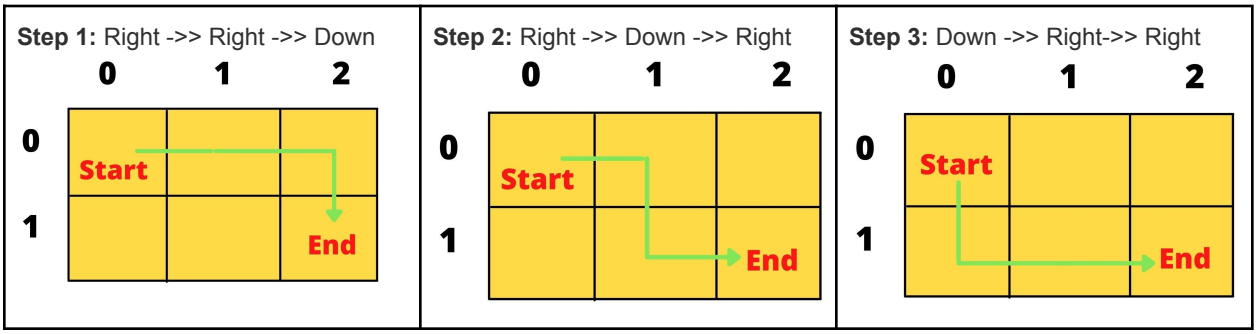
Grid Unique Paths | Count paths from left-top to the right bottom of a matrix

Problem Statement: Given a matrix **m X n**, count paths from left-top to the right bottom of a matrix with the constraints that from each cell you can either only move to the rightward direction or the downward direction.
Example 1:
Input Format: m = 2, n= 2
Output: 2

Explanation: From the top left corner there are total 2 ways to reach the bottom right corner:

Step 1: Right ->> Down	**Step 2:** Down ->> Right	
		

Example 2:
Input Format: m = 2, n= 3
Ouput: 3
Explanation: From the top left corner there is a total o**f 3** ways to reach the bottom right corner.

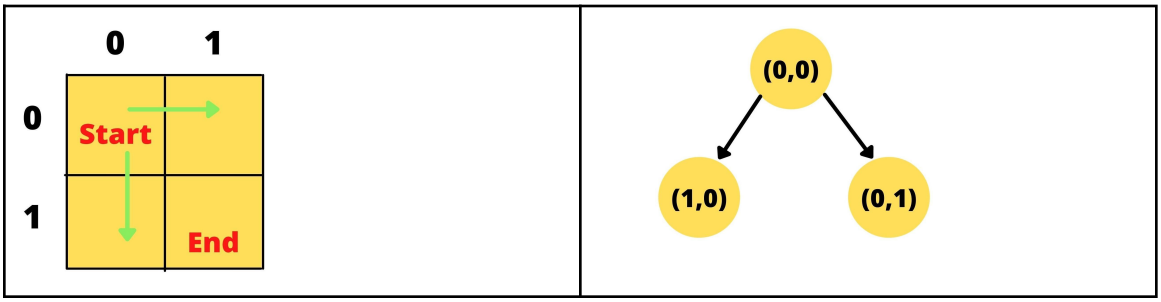


Solution

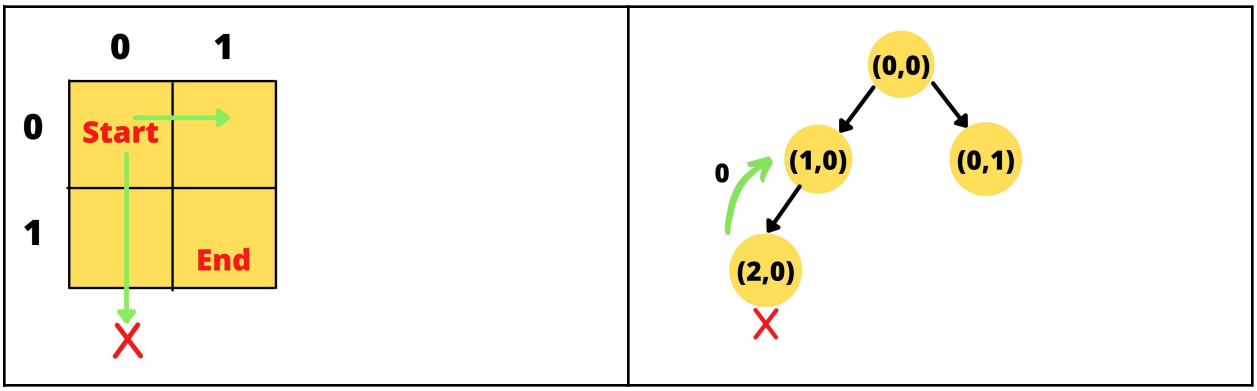
Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Approach: The problem states that we can either move rightward or downward direction. So we recursively try out all the possible combinations.

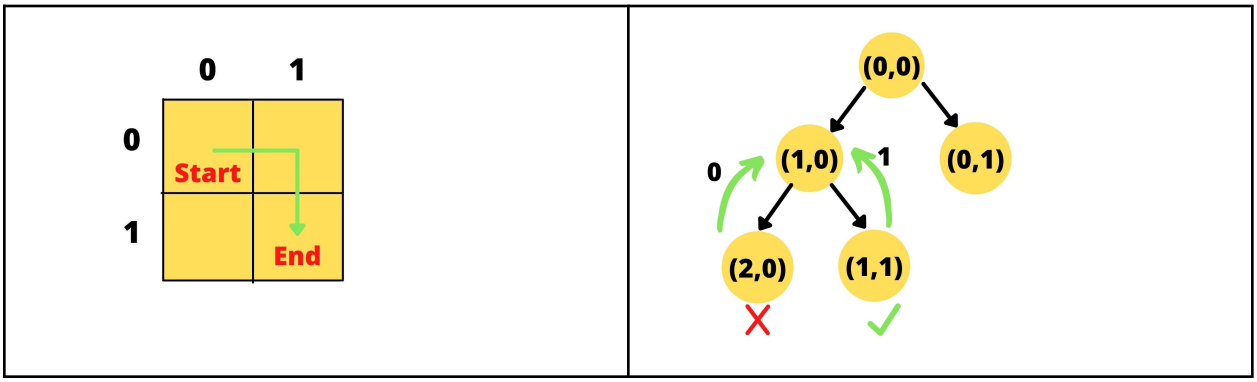
1. At first, we are at the **(0,0)** index let's assume this state as **(i,j).** From here we can move towards the bottom as well as towards the right and we recursively move until we hit the base case.



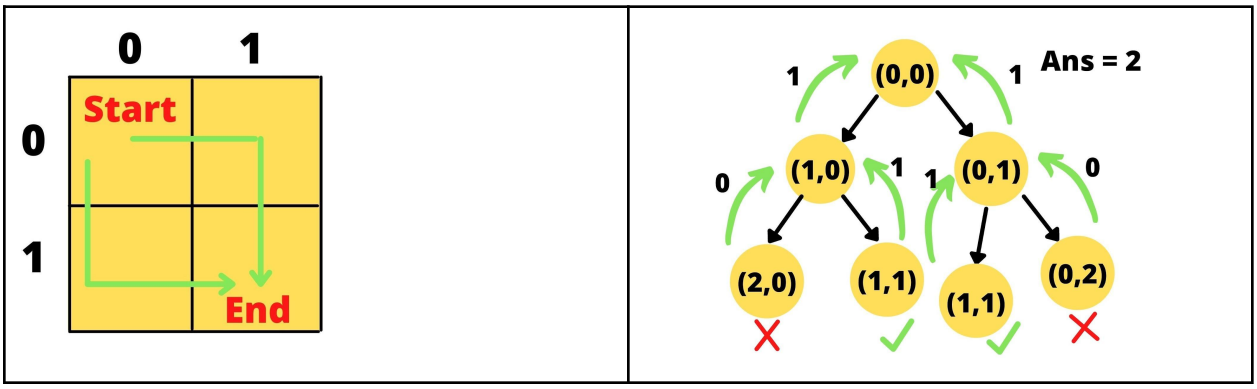
2. At any point of time when the recursive call goes out of the matrix boundary **(example: let's assume m = 2, n= 2, and the current position of i and j is (2,0) which is out of matrix boundary)**, we'll return zero because from here there are no possible paths beyond and that is the first base case.



3. Whenever the recursive call reaches the end we'll return **1** because we have found one possible path to reach the end.



4. In the recursive tree what result we have got from the left transition and the right transition will sum it up and return the answer.



Code:

```
#include<bits/stdc++.h>
using namespace std;
class Solution {
public:
     int countPaths(int i,int j,int n,int m)
    {
        if(i==(n-1)&&j==(m-1)) return 1;
        if(i>=n||j>=m) return 0;
        else return countPaths(i+1,j,n,m)+countPaths(i,j+1,n,m);
    }
    int uniquePaths(int m, int n) {
       return countPaths(0,0,m,n);
    }
};
int main()
{
    Solution obj;
    int totalCount= obj.uniquePaths(3,7);
    cout<<"The total number of Unique Paths are "<<totalCount<<endl;
}
```

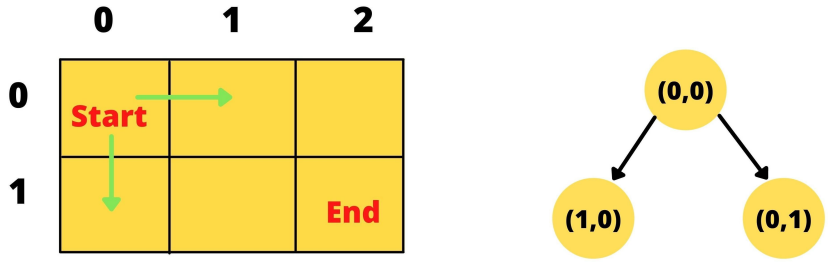
Output:
The total number of Unique Paths are 28

Time Complexity: The time comp[lexity of this recursive solution is exponential.

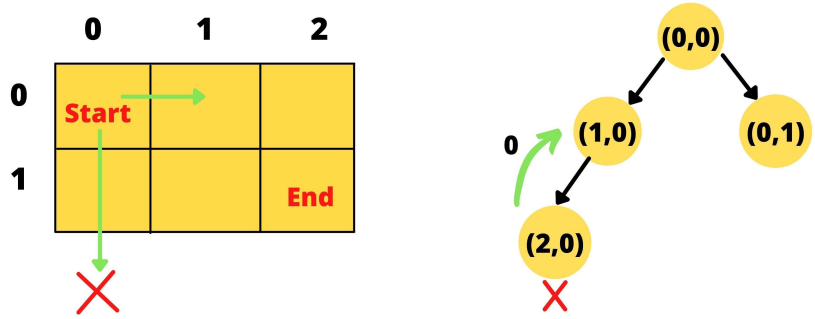
Space Complexity: As we are using stack space for recursion so the space complexity will also be exponential.

Solution 2: Dynamic Programming Solution
Intuition: The dynamic programming solution of this problem is a bit extension of the recursive solution. In recursive solutions, there are many overlapping subproblems. In this solution instead of traversing all the possible paths, whenever we get the result we'll store it in a matrix for future use. Whenever we encounter the same subproblem we directly get the value from the matrix instead of recomputing it. By this memorization technique, we can avoid the recomputation and the time complexity will be drastically reduced. This is the main intuition behind this dynamic programming solution.

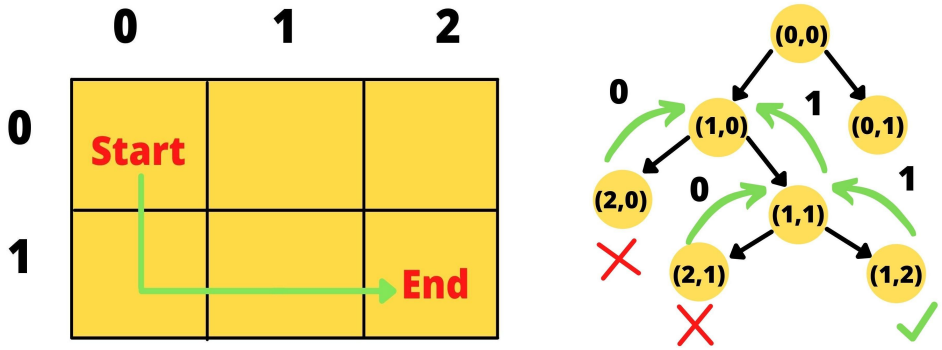
Approach:
Step 1: Take a dummy matrix **A[ ][ ]** of size **m X n** and fill it with **-1.**
Step 2: At first, we are at the **(0,0)** index let's assume this state as **(i,j).** From here we can move towards the bottom as well as towards the right and we recursively move until we hit the base case.



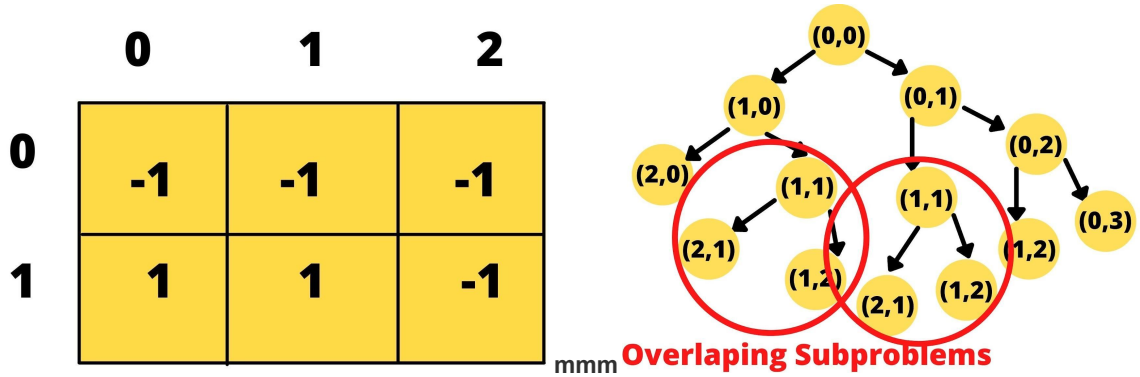
Step 3: At any point of time when the recursive call goes out of the boundary **(example: let's assume m = 2, n= 3, and the current position of i and j is (2,0) which is out of matrix boundary)**, we will return zero because from here there are no possible paths beyond and that is the first base case.



Step 4: Whenever the recursive call reaches the end we'll return **1** because we have found one possible path to reach the end.



Step 5: The only change in the dynamic programming solution is whenever we are returning answers we store them in the matrix **A[i][j]** and wherever we are making a recursive call we simply check if that state is already visited or not in other words we'll check if **A[i][j]** is **-1** or not if it is not **-1** that means that there is a subproblem which is repeating. Now instead of recomputing the subproblem, we'll return the value at **A[i][j].**



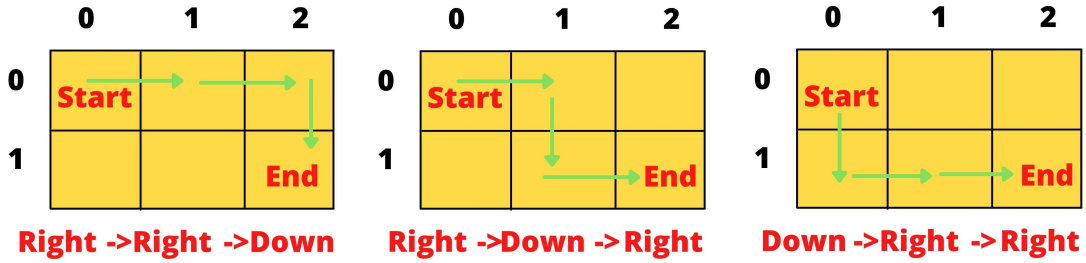
Code:

```
#include<bits/stdc++.h>
using namespace std;
class Solution {
public:
    int countPaths(int i,int j,int n,int m,vector<vector<int>> &dp)
    {
        if(i==(n-1)&&j==(m-1)) return 1;
        if(i>=n||j>=m) return 0;
        if(dp[i][j]!=-1) return dp[i][j];
     else return dp[i][j]=countPaths(i+1,j,n,m,dp)+countPaths(i,j+1,n,m,dp);

    }
    int uniquePaths(int m, int n) {
        vector<vector<int>> dp(m+1,vector<int>(n+1,-1));

        //dp[1][1]=1;
       int num=countPaths(0,0,m,n,dp);
        if(m==1&&n==1)
```

```
            return num;
        return dp[0][0];
    }
};
int main()
{
    Solution obj;
    int totalCount= obj.uniquePaths(3,7);
    cout<<"The total number of Unique Paths are "<<totalCount<<endl;}
```

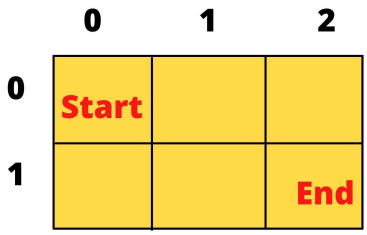
Output:
The total number of Unique Paths are 28
Time Complexity: The time complexity of this solution will be **O(n*m)** because at max there can be **n*m** number of states.
Space Complexity: As we are using extra space for the dummy matrix the space complexity will also be **O(n*m).**

Solution 3: Combinatorics Solution
Intuition: If we observe examples there is a similarity in paths from start to end. Each time we are taking an exactly **m+n-2** number of steps to reach the end.

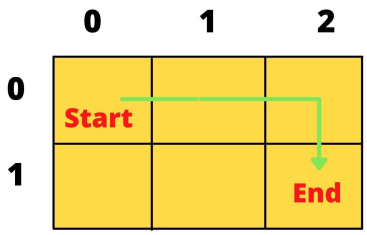


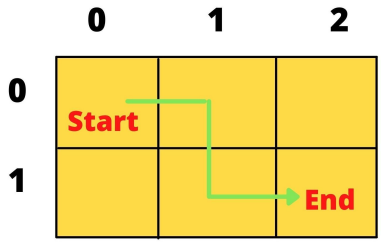
From start to reach the end we need a certain number of rightward directions and a certain number of downward directions. So we can figure out we need **n-1** no. of rightward direction and **m-1** no. of downward direction to reach the endpoint.
Since we need an **m+n-2** number of steps to reach the end among those steps if we choose **n-1** rightward direction or **m-1** downward direction and calculate the combinations **(ie: m+n-2Cn-1** or **m+n-2Cm-1**) we'll get the total number of paths.
Approach: The approach of this solution is very simple just use a for loop to calculate the **m+n-2Cn-1** or **m+n-2Cm-1**

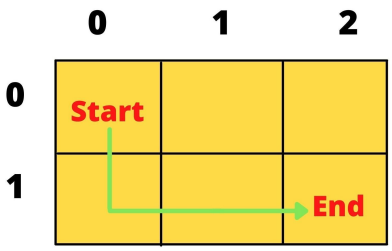
Dry Run: We'll take the input m = 2 and n = 3



According to our observation, we need an **m+n-2** number of steps to reach the end. So we need **3** steps and in every step, we need an **n-1** number of rightward direction and **m-1** number of downward direction. Among **3** steps if we choose **2** rightward directions then the result will be **3 (3C2)** or among **3** steps if we choose **1** downward direction then the result will also be **3 (3C1)**.







Code:

```
#include<bits/stdc++.h>
using namespace std;
class Solution {
public:
    int uniquePaths(int m, int n) {
            int N = n + m - 2;
            int r = m - 1;
            double res = 1;

            for (int i = 1; i <= r; i++)
                res = res * (N - r + i) / i;
            return (int)res;
    }
};
int main()
{
    Solution obj;
    int totalCount= obj.uniquePaths(2,3);
    cout<<"The total number of Unique Paths are "<<totalCount<<endl;
}
```

Output:
The total number of Unique Paths are 3
Time Complexity: The time complexity of this solution will be **O(n-1)** or **O(m-1)** depending on the formula we are using.
Space Complexity: As we are not using any extra space the space complexity of the solution will be **O(1).**

Count Reverse Pairs

Problem Statement: Given an array of numbers, you need to return the count of reverse pairs. **Reverse Pairs** are those pairs where i<j and arr[i]>2*arr[j].

Examples:

Example 1:

Input: N = 5, array[] = {1,3,2,3,1)

Output: 2

Explanation: The pairs are (3, 1) and (3, 1) as from both the pairs the condition arr[i] > 2*arr[j] is satisfied.

Example 2:

Input: N = 4, array[] = {3,2,1,4}

Output: 1

Explaination: There is only 1 pair (3 , 1) that satisfy the condition arr[i] > 2*arr[j]

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

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.

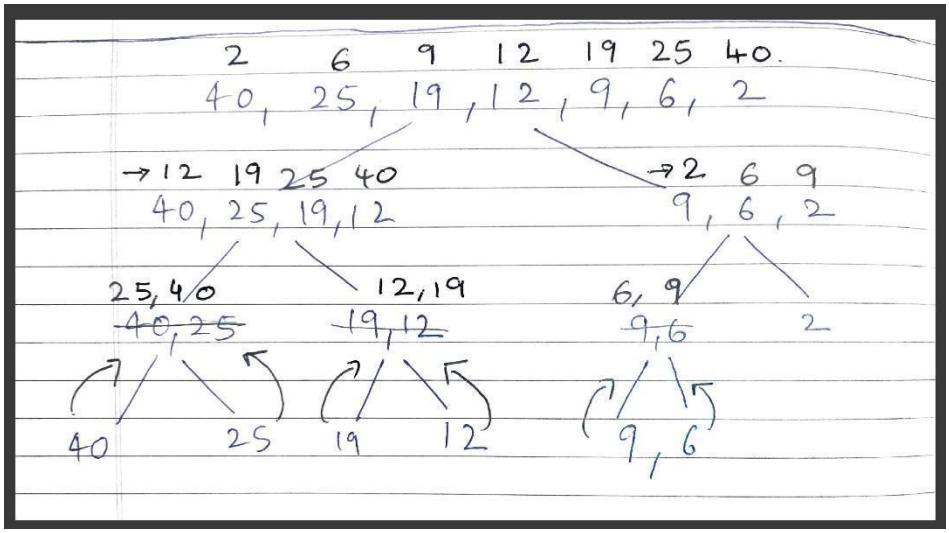
Code:

```
#include<bits/stdc++.h>
using namespace std;
 int reversePairs(vector < int > & arr) {
      int Pairs = 0;
      for (int i = 0; i < arr.size(); i++) {
        for (int j = i + 1; j < arr.size(); j++) {
          if (arr[i] > 2 * arr[j]) Pairs++;
        }
      }
      return Pairs;
    }


int main()
{
    vector<int> arr{1,3,2,3,1};
```

```
    cout<<"The Total Reverse Pairs are "<<reversePairs(arr);
}
```

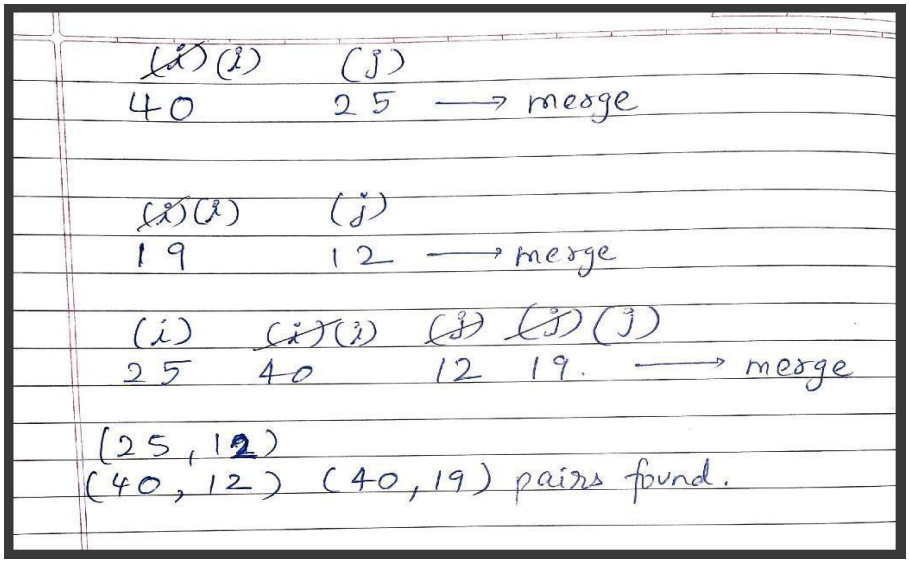
Output: The Total Reverse Pairs are 2
Time Complexity: O (N^2) (Nested Loops)
Space Complexity: O(1)

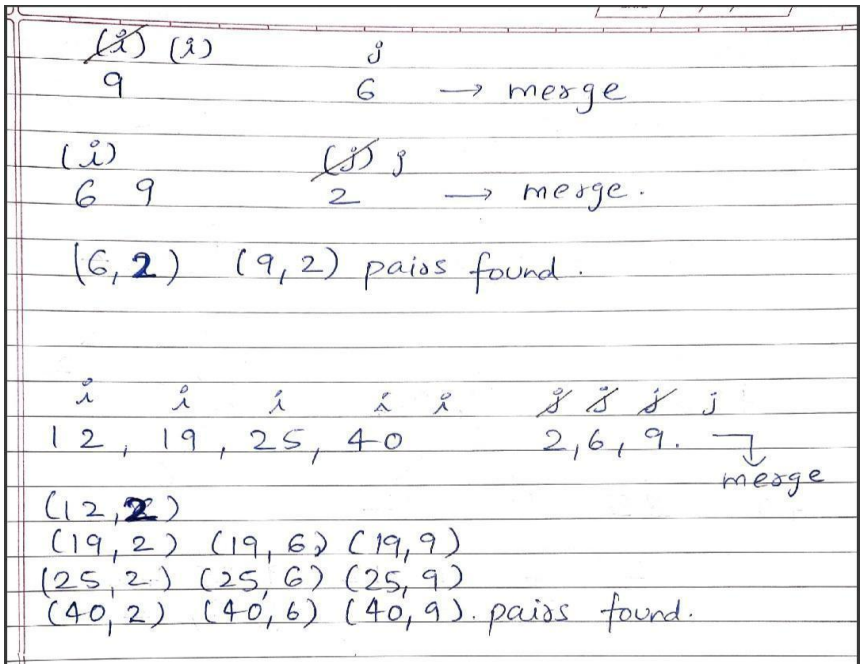
Solution 2: Optimal Solution
Intuition:
-> We will be using the Merge Sort Algorithm to solve this problem. We split the whole array into 2 parts creating a Merge Sort Tree-like structure. During the conquer step we do the following task :
-> We take the left half of the Array and Right half of the Array, 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.





Code:

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

using namespace std;
int Merge(vector < int > & nums, int low, int mid, int high) {
  int total = 0;
  int j = mid + 1;
  for (int i = low; i <= mid; i++) {
    while (j <= high && nums[i] > 2 LL * nums[j]) {
      j++;
    }
    total += (j - (mid + 1));
  }

  vector < int > t;
```

```
  int left = low, right = mid + 1;

  while (left <= mid && right <= high) {

    if (nums[left] <= nums[right]) {
      t.push_back(nums[left++]);
    } else {
      t.push_back(nums[right++]);
    }
  }

  while (left <= mid) {
    t.push_back(nums[left++]);
  }
  while (right <= high) {
    t.push_back(nums[right++]);
  }

  for (int i = low; i <= high; i++) {
    nums[i] = t[i - low];
  }
  return total;
}
int MergeSort(vector < int > & nums, int low, int high) {

  if (low >= high) return 0;
  int mid = (low + high) / 2;
  int inv = MergeSort(nums, low, mid);
  inv += MergeSort(nums, mid + 1, high);
  inv += Merge(nums, low, mid, high);
  return inv;
}
int reversePairs(vector < int > & arr) {
  return MergeSort(arr, 0, arr.size() - 1);
}

int main() {
  vector<int> arr{1,3,2,3,1};
  cout << "The Total Reverse Pairs are " << reversePairs(arr);
}
```

Output: The Total Reverse Pairs are 2
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