1.[**1845. Seat Reservation Manager**](https://leetcode.com/problems/seat-reservation-manager/)

Design a system that manages the reservation state of n seats that are numbered from 1 to n.

Implement the SeatManager class:

* SeatManager(int n) Initializes a SeatManager object that will manage n seats numbered from 1 to n. All seats are initially available.
* int reserve() Fetches the **smallest-numbered** unreserved seat, reserves it, and returns its number.
* void unreserve(int seatNumber) Unreserves the seat with the given seatNumber.

**Example 1:**

**Input**

["SeatManager", "reserve", "reserve", "unreserve", "reserve", "reserve", "reserve", "reserve", "unreserve"]

[[5], [], [], [2], [], [], [], [], [5]]

**Output**

[null, 1, 2, null, 2, 3, 4, 5, null]

**Explanation**

SeatManager seatManager = new SeatManager(5); // Initializes a SeatManager with 5 seats.

seatManager.reserve(); // All seats are available, so return the lowest numbered seat, which is 1.

seatManager.reserve(); // The available seats are [2,3,4,5], so return the lowest of them, which is 2.

seatManager.unreserve(2); // Unreserve seat 2, so now the available seats are [2,3,4,5].

seatManager.reserve(); // The available seats are [2,3,4,5], so return the lowest of them, which is 2.

seatManager.reserve(); // The available seats are [3,4,5], so return the lowest of them, which is 3.

seatManager.reserve(); // The available seats are [4,5], so return the lowest of them, which is 4.

seatManager.reserve(); // The only available seat is seat 5, so return 5.

seatManager.unreserve(5); // Unreserve seat 5, so now the available seats are [5].

**Constraints:**

* 1 <= n <= 105
* 1 <= seatNumber <= n
* For each call to reserve, it is guaranteed that there will be at least one unreserved seat.
* For each call to unreserve, it is guaranteed that seatNumber will be reserved.
* At most 105 calls **in total** will be made to reserve and unreserve.

Approach 1:

1. There are given three Function (seat Manager), reserve, unreserve.
2. In first function seat Manger we crate number of object that is given in function parameter.
3. In second function reserve the smallest number seat.
4. In third function unreserve the seat that is given.
5. So, the approach is to we create a min priority queue that means smallest element is always on top of the priority queue.
6. At first function create a given number of object of priority queue and push into the queue.
7. In second function for reserve we simply remove the top element from the queue. Also check the edge case that is queue is not empty.
8. In third function for unreserve push the element into the queue that the element is unresrve.

Code:

  priority\_queue<int, vector<int, greater<int> > pq;

    SeatManager(int n) {

        for(int i=1;i<=n;i++)

        {

            pq.push(i);

        }

    }

    int reserve () {

       if(pq.size()==0)

       {

           return 0;

       }

       else

       {

           int ans=pq.top();

           pq.pop();

           return ans;

       }

    }

    void unreserve (int seatNumber) {

        pq.push(seatNumber);

    }

Time Complexity: O (N log (N));// log (N) due to push and pop TC of PQ.

Space Complexity: O(N);//for priority queue;

==================================================================================

2.[**345. Reverse Vowels of a String**](https://leetcode.com/problems/reverse-vowels-of-a-string/)

Given a string s, reverse only all the vowels in the string and return it.

The vowels are 'a', 'e', 'i', 'o', and 'u', and they can appear in both lower and upper cases, more than once.

**Example 1:**

**Input:** s = "hello"

**Output:** "holle"

**Example 2:**

**Input:** s = "leetcode"

**Output:** "leotcede"

Approach 1:

* 1. While see this problem the first solution came in mind is to two pointer approach.
  2. We can declare the two variable that is low and high and values which is assign to low in starting index and high in last index.
  3. We can take while loop the check the condition low is less then high and also take another while inside the first while which is also satisfied same condition but addition condition is low is less then size of the string.
  4. Inside the second while loop check if vowel is found break the inner loop.
  5. Similarly in high index case and implement one condition in loop check high is greater than zero.
  6. After end both inner loops swap the element.
  7. And increment low index and decrement the high index.
  8. End.
  9. Return string.

Code:

bool isVowel(char ch)

    {

        if( ch=='A'||ch=='E'||ch=='I'|| ch=='O'||ch=='U'||ch=='a'||ch=='e'||ch=='i'|| ch=='o'||ch=='u')

      {

                    return true;

      }

      return false;

    }

    string reverseVowels(string s) {

        int n=s.length();

        int low=0, high=n-1;

        while(low<high)

        {

            while(low<high&&low<n)

            {

                if(isVowel(s[low]))

                {

                    break;

                }

                low++;

            }

             while(low<high&&high>=0)

            {

                 if(isVowel(s[high]))

                {

                    break;

                }

                high--;

            }

            if(low<high)

            {

                swap(s[low],s[high]);

            }

            low++;

            high--;

        }

        return s;

    }

Time complexity: O (N); // N is length of string.

Space complexity: O (1);

==================================================================================

3.[**2849. Determine if a Cell Is Reachable at a Given Time**](https://leetcode.com/problems/determine-if-a-cell-is-reachable-at-a-given-time/)

You are given four integers sx, sy, fx, fy, and a **non-negative** integer t.

In an infinite 2D grid, you start at the cell (sx, sy). Each second, you **must** move to any of its adjacent cells.

Return true *if you can reach cell*(fx, fy) *after****exactly*** t ***seconds***, *or* false *otherwise*.

A cell's **adjacent cells** are the 8 cells around it that share at least one corner with it. You can visit the same cell several times.

**Example 1:**

**Input:** sx = 2, sy = 4, fx = 7, fy = 7, t = 6

**Output:** true

**Explanation:** Starting at cell (2, 4), we can reach cell (7, 7) in exactly 6 seconds by going through the cells depicted in the picture above.

**Example 2:**

**Input:** sx = 3, sy = 1, fx = 7, fy = 3, t = 3

**Output:** false

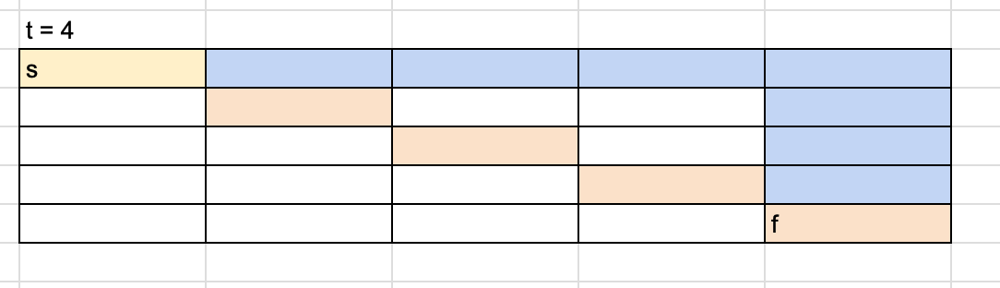
**Explanation:** Starting at cell (3, 1), it takes at least 4 seconds to reach cell (7, 3) by going through the cells depicted in the picture above. Hence, we cannot reach cell (7, 3) at the third second.

**Constraints:**

* 1 <= sx, sy, fx, fy <= 109
* 0 <= t <= 109

Approach 1:

1. If sx fx and sy fy are equal and time not equal to 1 return true else false.
2. Because in problem statement say that you move in each time 1 second so if any cell move its either take 0 or more than 1 not equal to 1.
3. If different start and finish cell.
4. Compute the absolute value of x coordinate and y coordinate.
5. For minimum time to reach always greater than x absolute value of x and y coordinate both.
6. X<=t&&y<=t;
7. In given picture you can see minimum 4 sec to reach finish cell.
8. Similarly, in horizontal and vertical either of x or y coordinate less then equal to time.



Code:

  bool isReachableAtTime(int sx, int sy, int fx, int fy, int t) {

       int miny=abs(fy-sy);

       int minx=abs(fx-sx);

       if(miny==0&&minx==0)

       {

           return t!=1?1:0;

       }

       return minx<=t&&miny<=t;

    }

Time complexity: O (1);

Space complexity: O (1);

==================================================================================

4.[**1759. Count Number of Homogenous Substrings**](https://leetcode.com/problems/count-number-of-homogenous-substrings/)

Given a string s, return the number of ***homogenous*** substrings of s. Since the answer may be too large, return it **modulo** 109 + 7.

A string is **homogenous** if all the characters of the string are the same.

A **substring** is a contiguous sequence of characters within a string.

**Example 1:**

**Input:** s = "abbcccaa"

**Output:** 13

**Explanation:** The homogenous substrings are listed as below:

"a" appears 3 times.

"aa" appears 1 time.

"b" appears 2 times.

"bb" appears 1 time.

"c" appears 3 times.

"cc" appears 2 times.

"ccc" appears 1 time.

3 + 1 + 2 + 1 + 3 + 2 + 1 = 13.

**Example 2:**

**Input:** s = "xy"

**Output:** 2

**Explanation:** The homogenous substrings are "x" and "y".

**Example 3:**

**Input:** s = "zzzzz"

**Output:** 15

Approach 1:

* 1. Step 1 we can compute the sequence of frequency element of each character.
  2. Compute the total number appears of times of each character using the formula.
  3. Like sum of n element ( n\* (n+1))/2;
  4. Finally, sum of all the answer of each character of frequency.
  5. Also compute the sum module division of answer (sum is very large).

Code:

int mod=1000000007;

    int countPossible\_string(long long int n)

    {

        long long int sum=0;

         sum=((n\*(n+1))/2);

        return sum%mod;

    }

    int countHomogenous(string s) {

        int n=s.size();

        int ans=0;

        for(int i=0;i<n;i++)

        {

            long long int count=1;

            for(int j=i+1;j<n;j++)

            {

               if(s[i]==s[j])

                {

                    count++;

                    i++;

                }

                else

                {

                    break;

                }

            }

            ans+=countPossible\_string(count);

        }

        return ans%mod;

    }

Time complexity: O (N);

Space Complexity: O (1);

Approach 2:

1. Similar to the first approach but some change.
2. Compute the frequency and computing the update the resalt.
3. In each character if frequency is greater than 1
4. Compute ans= right – left +1;
5. Else ans +=1;
6. End;

**How it works**

Input: "aaaaa"

left = 0

right = 0

Find "a"

output: 1

left = 0

right = 1

output: 3 (1 + 2)

1 is current output

2 is length of substring (= "aa")

left = 0

right = 2

output: 6 (3 + 3)

(first)3 is current output

(second)3 is length of substring (= "aaa")

left = 0

right = 3

output: 10 (6 + 4)

6 is current output

4 is length of substring (= "aaaa")

left = 0

right = 4

output: 15 (10 + 5)

10 is current output

5 is length of substring (= "aaaaa")

output: 15

Code:

int mod=1000000007;

    int countHomogenous(string s) {

        int n=s.size();

        long long int ans=0;

        int left=0;

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

        {

            if(s[left]==s[right])

            {

                ans+=(right-left+1);

            }

            else

            {

                ans+=1;

                left=right;

            }

        }

        return ans%mod;

    }

Time complexity: O (N);

Space Complexity: O (1);

==================================================================================

5.[**1743. Restore the Array From Adjacent Pairs**](https://leetcode.com/problems/restore-the-array-from-adjacent-pairs/)

There is an integer array nums that consists of n **unique**elements, but you have forgotten it. However, you do remember every pair of adjacent elements in nums.

You are given a 2D integer array adjacentPairs of size n - 1 where each adjacentPairs[i] = [ui, vi] indicates that the elements ui and vi are adjacent in nums.

It is guaranteed that every adjacent pair of elements nums[i] and nums[i+1] will exist in adjacentPairs, either as [nums[i], nums[i+1]] or [nums[i+1], nums[i]]. The pairs can appear **in any order**.

Return *the original array*nums*. If there are multiple solutions, return****any of them***.

**Example 1:**

**Input:** adjacentPairs = [[2,1],[3,4],[3,2]]

**Output:** [1,2,3,4]

**Explanation:** This array has all its adjacent pairs in adjacentPairs.

Notice that adjacentPairs[i] may not be in left-to-right order.

**Example 2:**

**Input:** adjacentPairs = [[4,-2],[1,4],[-3,1]]

**Output:** [-2,4,1,-3]

**Explanation:** There can be negative numbers.

Another solution is [-3,1,4,-2], which would also be accepted.

**Example 3:**

**Input:** adjacentPairs = [[100000,-100000]]

**Output:** [100000,-100000]

**Constraints:**

* nums.length == n
* adjacentPairs.length == n - 1
* adjacentPairs[i].length == 2
* 2 <= n <= 105
* -105 <= nums[i], ui, vi <= 105
* There exists some nums that has adjacentPairs as its pairs.

Approach 1:

Using hash Map.

1. In Map vector in move only on time. (edge case)
2. We can store all the element also store all the neighbour element.
3. Find the stating or end element for which we check the size of map vector if it is one Store into the answer vector.
4. Iterate the loop and take two variable prev and curr for storing neighbouring element and update each iteration.
5. End.

Code:

vector<int> restoreArray(vector<vector<int>>& adjacentPairs) {

       int n=adjacentPairs.size();

       vector<int>ans;

       unordered\_map<int,vector<int>>m;

       for(int i=0;i<n;i++)

       {

           m[adjacentPairs[i][0]].push\_back(adjacentPairs[i][1]);

           m[adjacentPairs[i][1]].push\_back(adjacentPairs[i][0]);

       }

       for(int i=0;i<n;i++)

       {

           if(m[adjacentPairs[i][0]].size()==1)

           {

               ans.push\_back(adjacentPairs[i][0]);

               break;

           }

           if(m[adjacentPairs[i][1]].size()==1)

           {

               ans.push\_back(adjacentPairs[i][1]);

               break;

           }

       }

       int prev=0,curr=ans[0];

       for(int i=0;i<n;i++)

       {

           int p=0;

            for(int j=0;j<m[ans[i]].size();j++)

            {

                 if(m[ans[i]][j]!=prev&&p==0)

                 {

                     prev=ans[i];

                     curr=m[ans[i]][j];

                     ans.push\_back(curr);

                     p++;

                 }

            }

       }

       return ans;

    }

Time complexity: O (N);

Space Complexity: O (N);

Approach 2:

Using Graph:

Code:

1. Create a graph.
2. Find the first or last element.
3. Traversal the graph and value store into answer.
4. Return ans.

 void dfs(map<int,vector<int>> &m,int val,set<int> &s, vector<int> &ans)

     {

         if(s.find(val)!=s.end())return;

         ans.push\_back(val);

         s.insert(val);

         for(auto x:m[val])

         {

             dfs(m,x,s,ans);

         }

     }

    vector<int> restoreArray(vector<vector<int>>& adjacentPairs) {

       int n=adjacentPairs.size();

       vector<int>ans;

       map<int,vector<int>>m;

       for(auto x:adjacentPairs)

       {

           m[x[0]].push\_back(x[1]);

           m[x[1]].push\_back(x[0]);

       }

       int head;

       for(auto x:m)

       {

           if(x.second.size()==1)

           {

               head=x.first;

               break;

           }

       }

       set<int>vis;

       dfs(m,head,vis,ans);

       return ans;

    }

Time complexity: O (N) + O (2\*E);

Space Complexity: O (N);

==================================================================================

Date 14 November 2023 important question

6.[**1930. Unique Length-3 Palindromic Subsequences**](https://leetcode.com/problems/unique-length-3-palindromic-subsequences/)

Given a string s, return *the number of****unique palindromes of length three****that are a****subsequence****of*s.

Note that even if there are multiple ways to obtain the same subsequence, it is still only counted **once**.

A **palindrome** is a string that reads the same forwards and backwards.

A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

* For example, "ace" is a subsequence of "abcde".

**Example 1:**

**Input:** s = "aabca"

**Output:** 3

**Explanation:** The 3 palindromic subsequences of length 3 are:

- "aba" (subsequence of "aabca")

- "aaa" (subsequence of "aabca")

- "aca" (subsequence of "aabca")

**Example 2:**

**Input:** s = "adc"

**Output:** 0

**Explanation:** There are no palindromic subsequences of length 3 in "adc".

**Example 3:**

**Input:** s = "bbcbaba"

**Output:** 4

**Explanation:** The 4 palindromic subsequences of length 3 are:

- "bbb" (subsequence of "bbcbaba")

- "bcb" (subsequence of "bbcbaba")

- "bab" (subsequence of "bbcbaba")

- "aba" (subsequence of "bbcbaba")

**Constraints:**

* 3 <= s.length <= 105
* s consists of only lowercase English letters.

Approach 1: Brute Force approach   
1. We can compute all the possible sub string.

2. we can used one set data structure in this store all the sun sequence string of length three as well as palindrome.

3. return the size of set data structure.

Code:

 void printsubsequence(string input ,string output,unordered\_set<string>&st)

    {

        if(input.size()==0)

        {

            if(output.size()==3&&output[0]==output[2])

            {

                st.insert(output);

            }

            return;

        }

        printsubsequence(input.substr(1),output,st);// not take the value

        printsubsequence(input.substr(1),output+input[0],st); //take the value

    }

    int countPalindromicSubsequence(string s) {

        int n=s.size();

      unordered\_set<string>st;

       printsubsequence(s,"",st);

 return st.size();

}

Time complexity: O (2^n)

Space Complexity: O (N + N)

Approach 2:

Optimize solution:

1. first must we know s.find(‘a’) and s.rfind(‘a’) are function. These function basically return the index value of first time the character ‘a’ is found in string similarly second function that will return the last index appearance of the character in the string.
2. Uses of set data structure.

Code:

 int countPalindromicSubsequence(string s) {

int res = 0;

        unordered\_set<char> uniq;

        for (char c : s) {

            uniq.insert(c);

        }

        for (char c : uniq) {

            int start = s.find(c);

            int end = s.rfind(c);

            if (start < end) {

                unordered\_set<char> charSet;

                for (int i = start + 1; i < end; i++) {

                    charSet.insert(s[i]);

                }

                res += charSet.size();

            }

        }

        return res;

    }

Time Complexity: O (N+ number of unique elements in string);

Space Complexity: O (N);

Approach 3:

Similar to second but logic is different.

Code:

  int countPalindromicSubsequence(string s) {

      int n=s.size();

       unordered\_set<string>st;

       vector<int> left(26),right(26);

       for(int i=0;i<n;i++)

       {

           right[s[i]-'a']++;

       }

       for(int i=0;i<n;i++)

       {

           right[s[i]-'a']--;

           for(int j=0;j<26;j++)

           {

               if(left[j]>0&&right[j]>0){

               char ch='a'+j;

               string temp="";

               temp+=ch;

               temp+=s[i];

               temp+=ch;

               st.insert(temp);

               }

           }

           left[s[i]-'a']++;

       }

       return st.size();}

==================================================================================

7.[**1838. Frequency of the Most Frequent Element**](https://leetcode.com/problems/frequency-of-the-most-frequent-element/)

**Using sliding window approach:**

The **frequency** of an element is the number of times it occurs in an array.

You are given an integer array nums and an integer k. In one operation, you can choose an index of nums and increment the element at that index by 1.

Return *the****maximum possible frequency****of an element after performing****at most***k*operations*.

**Example 1:**

**Input:** nums = [1,2,4], k = 5

**Output:** 3

**Explanation:** Increment the first element three times and the second element two times to make nums = [4,4,4].

4 has a frequency of 3.

**Example 2:**

**Input:** nums = [1,4,8,13], k = 5

**Output:** 2

**Explanation:** There are multiple optimal solutions:

- Increment the first element three times to make nums = [4,4,8,13]. 4 has a frequency of 2.

- Increment the second element four times to make nums = [1,8,8,13]. 8 has a frequency of 2.

- Increment the third element five times to make nums = [1,4,13,13]. 13 has a frequency of 2.

**Example 3:**

**Input:** nums = [3,9,6], k = 2

**Output:** 1

**Constraints:**

* 1 <= nums.length <= 105
* 1 <= nums[i] <= 105
* 1 <= k <= 105

Approach 1: Brute force approach.

* 1. Sort the array list.
  2. Traversal from last index and until the value k is negative.
  3. Simultaneously, count the frequency of element.
  4. Update the ans of Maximum frequency.
  5. Return ans;

Time Complexity: O(N^2)

Space Complexity: O(1)

Code:

But not work properly:

 int maxFrequency(vector<int>& nums, int k) {

        sort(nums.begin(),nums.end());

        int n=nums.size();

  int ans=0, freq=0,temp=0,k=k1;

        if(n==1)

        return 1;

        for(int i=n-1;i>0;i--)

        {

             freq=0;

             int j=i;

             while(1)

             {

                 freq++;

                 int updatekvalues=k-(nums[i]-nums[j]);

                 j--;

                 if(updatekvalues>(nums[i]-nums[j])||j<=0)

                 {

                     break;

                 }

                 k=updatekvalues;

             }

             k=k1;

             ans=max(ans,freq);

        }

        return ans;

Approach 2: sliding window (optimal solution)

1. sort the array
2. take two pointers with the help check the condition given below.
3. (nums[right]\*(right-left+1))<=(tatalsuminbetweenleftandright + k).
4. If above condition is not true then subtract the total sum left index value and increament the left pointer.
5. Maximize the answer.

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

Space Complexity: O(1)

Code:

  int maxFrequency(vector<int>& nums, int k) {

        sort(nums.begin(),nums.end());

        int n=nums.size();

        //using sliding window

        int left=0,right=0;

        long ans=0,total=0;

        while(right<n)

        {

            total+=nums[right];

            while(((long)(right-left+1)\*nums[right])>(k+total))

            {

                total-=nums[left];

                left++;

            }

            ans=max(ans,(long)(right-left+1));

            right++;

        }

       return ans;

}

==================================================================================

8. [**1814. Count Nice Pairs in an Array**](https://leetcode.com/problems/count-nice-pairs-in-an-array/)

You are given an array nums that consists of non-negative integers. Let us define rev(x) as the reverse of the non-negative integer x. For example, rev(123) = 321, and rev(120) = 21. A pair of indices (i, j) is **nice** if it satisfies all of the following conditions:

* 0 <= i < j < nums.length
* nums[i] + rev(nums[j]) == nums[j] + rev(nums[i])

Return *the number of nice pairs of indices*. Since that number can be too large, return it **modulo** 109 + 7.

**Example 1:**

**Input:** nums = [42,11,1,97]

**Output:** 2

**Explanation:** The two pairs are:

- (0,3) : 42 + rev(97) = 42 + 79 = 121, 97 + rev(42) = 97 + 24 = 121.

- (1,2) : 11 + rev(1) = 11 + 1 = 12, 1 + rev(11) = 1 + 11 = 12.

**Example 2:**

**Input:** nums = [13,10,35,24,76]

**Output:** 4

**Constraints:**

* 1 <= nums.length <= 105
* 0 <= nums[i] <= 109

Solution:

Approach 1: optimal

1. Reverse the array of index element and subtraction the original index element and update the index value.
2. Sort the array.
3. Count the frequency of element.
4. Compute the total possible pair using formula (count\*(count-1))/2;
5. Sum of all the pair return answer.

Code:

 int mod=1000000007;

    int countNicePairs(vector<int>& nums) {

        int n=nums.size();

        for(int i=0;i<n;i++)

        {

            int val=nums[i],rev=0;

            while(val>0)

            {

               rev=rev\*10+val%10;

               val=val/10;

            }

            nums[i]=nums[i]-rev;

        }

        sort(nums.begin(),nums.end());

        long long int ans=0;

        for(int i=0;i<n-1;i++)

        {

            long long int count=1;

            while(i<n-1&&nums[i]==nums[i+1])

            {

                count++;

                i++;

            }

            ans=(ans%mod + (count\*(count-1))/2)%mod;

        }

        return (int)ans;

    }

Time Complexity:

O (N+maxNumberOfDigitOfElement) + O(NlonN)(sort)+ O (N)(for checking the unique element)

Space Complexity: O (1);

Approach 2:

Using hash map

 int mod=1000000007;

    int countNicePairs(vector<int>& nums) {

        int n=nums.size();

        unordered\_map<long,long> m;

        for(int i=0;i<n;i++)

        {

            int val=nums[i],rev=0;

            while(val>0)

            {

               rev=rev\*10+val%10;

               val=val/10;

            }

            m[nums[i]-rev]++;

        }

        long long int ans=0;

        for(auto x:m)

        {

            ans=(ans%mod + (x.second\*(x.second-1))/2)%mod;

        }

        return (int)ans;

    }

Time Complexity:

O (N+maxNumberOfDigitOfElement) ;

Space Complexity: O (N);

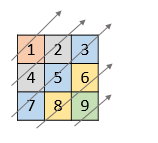
===============================================================================

**Important for interview**

9.[**1424. Diagonal Traverse II**](https://leetcode.com/problems/diagonal-traverse-ii/)

Given a 2D integer array nums, return *all elements of*nums*in diagonal order as shown in the below images*.

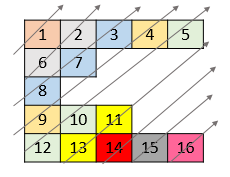
**Example 1:**



**Input:** nums = [[1,2,3],[4,5,6],[7,8,9]]

**Output:** [1,4,2,7,5,3,8,6,9]

**Example 2:**



**Input:** nums = [[1,2,3,4,5],[6,7],[8],[9,10,11],[12,13,14,15,16]]

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

**Constraints:**

* 1 <= nums.length <= 105
* 1 <= nums[i].length <= 105
* 1 <= sum(nums[i].length) <= 105
* 1 <= nums[i][j] <= 105

Approach 1: Brute force approach its go to time limit error.

* 1. Traversal the first half diagonal bottom to top insert into answer vector.
  2. Again traversal the second half.

Code:

vector<int> findDiagonalOrder(vector<vector<int>>& nums) {

        int row=nums.size();

        int maxval=0;

        for(int i=0;i<row;i++)

        {

            int p=nums[i].size();

            maxval=max(maxval,p);

        }

        maxval=max(row,maxval);

        vector<int> ans;

        for(int i=0;i<maxval;i++)

        {

            int p=i,q=0;

            for(int j=p;j>=0&&q<maxval;j--){

               if(j<row&&nums[j].size()>q)

               {

                   ans.push\_back(nums[j][q]);

               }

            q++;

            }

        }

        for(int i=1;i<maxval;i++)

        {

            int p=i,q=maxval-1;

            for(int j=p;j<maxval;j++){

               if(q<row&&nums[q].size()>j)

               {

                   ans.push\_back(nums[q][j]);

               }

            q--;

            }

        }

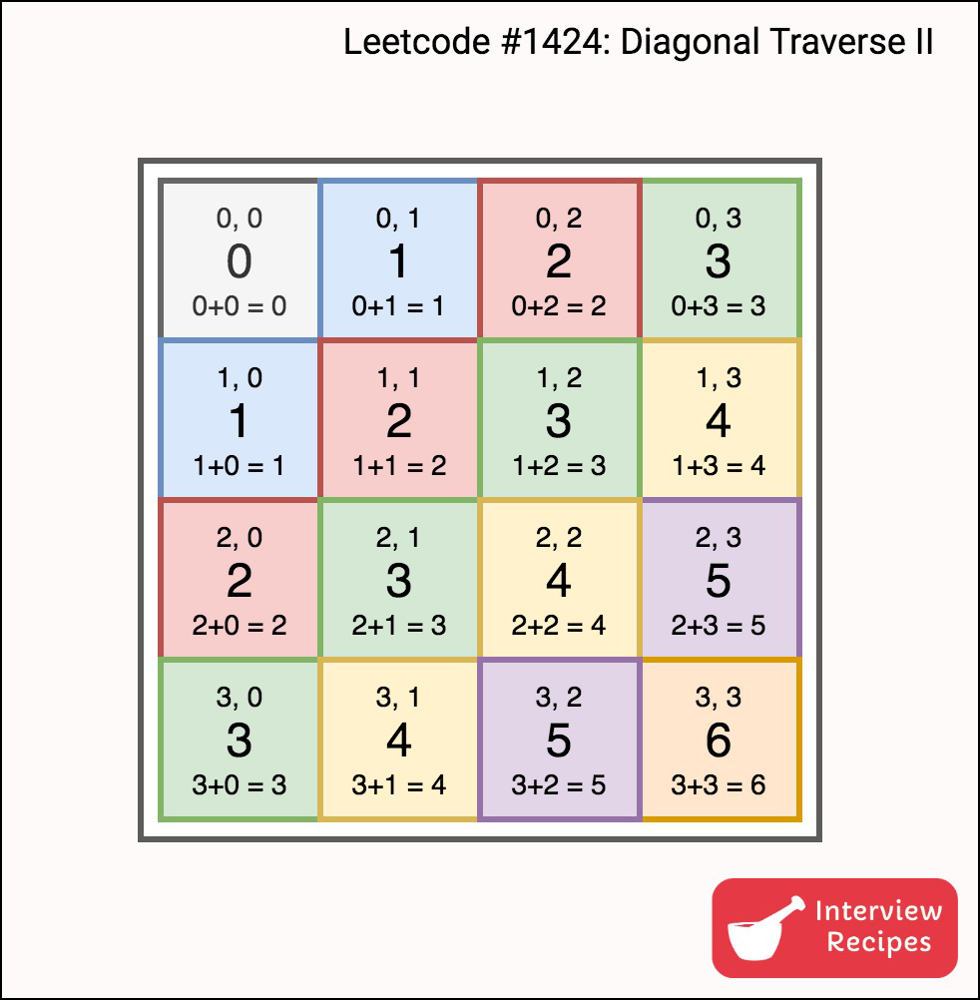
       return ans;}

Time Complexity: O(N^2) where n is the maximum size of either row or column.

Space Complexity: O(1)

Approach 2: Optimal and best using map and traversal from bottom.

**Key Idea**  
In a 2D matrix, elements in the same diagonal have same sum of their indices.



So if we have all elements with same sum of their indices together, then it’s just a matter of printing those elements in order.

**Algorithm**

1. Insert all elements into an appropriate bucket i.e. nums[i][j] in (i+j)th bucket.
2. For each bucket starting from 0 to max, print all elements in the bucket.  
   **Note**: Here, diagonals are from bottom to top, but we traversed the input matrix from first row to last row. Hence we need to print the elements in reverse order.

Code:

 vector<int> findDiagonalOrder(vector<vector<int>>& nums) {

 int n=nums.size();

       unordered\_map<int,vector<int>>m;

       vector<int> ans;

       int maxval=0;

       for(int i=n-1;i>=0;i--)

       {

           for(int j=0;j<nums[i].size();j++)

           {

               m[i+j].push\_back(nums[i][j]);

               maxval= max(maxval,i+j);

           }

       }

       for(int i=0;i<=maxval;i++)

       {

           for(auto x:m[i])

           {

                ans.push\_back(x);

           }

       }

       return ans;

    }

Time Complexity: O(N^2) where n is the maximum size of either row or column.

Space Complexity: O(1)

==================================================================================

[**1727. Largest Submatrix With Rearrangements**](https://leetcode.com/problems/largest-submatrix-with-rearrangements/)

Best question:

[**935. Knight Dialer**](https://leetcode.com/problems/knight-dialer/)

Best question:

[**1611. Minimum One Bit Operations to Make Integers Zero**](https://leetcode.com/problems/minimum-one-bit-operations-to-make-integers-zero/)

Difficult hard