Array

1. Set Matrix Zero
   1. Time Complexity: O((N\*M)(N+M)) 🡪 O(N\*M) for traversing through each element and (N+M) for traversing to row and column of elements having value 0.

Space Complexity: O(1)

* 1. Time Complexity : O(N\*M)

Space Complexity: O(1)

1. Pascals Triangle
   1. Time Complexity: O(numRows2)
   2. Space Complexity : O(numRows2)

Algo:

For(I = 0 🡪numRows-1 ){

r[i].resize(i+1)

r[i][0] = r[i][i] = 1;

for(j = 1;j<I;j++){

r[i][j] = r[i-1][j-1] + r[i-1][j]

}

}

return r;

1. Next Permutation:
   1. Time Complexity: O(N)
   2. Space Complexity: O(1)
      1. Algo:

int i;

for(I = n-2;i>=0;i--){

if(nums[i] < nums[i+1]){

break;

}

}

If(I >= 0){

For(j = n-1;j>I;--j){

Swap(nums[i],nums[j]);

Break;

}

}

Reverse(nums.begin()+i+1,nums.end());

1. Kadane’s Algorithm
   * + 1. Naïve Approach
          1. Time Complexity: O(N^3)
          2. Space Complexity : O(1)

Algo:

If(n==1){

Return nums[0];

}

Int I,j;

For(I = 0; i<= n-1;i++){

For(j = i 🡪 n-1;j++){

Int sum =0 ;

For( k =I;k<=j;k++){

Sum +=nums[k];

}

If(sum > max\_sum){

Subarray.clear();

Max\_sum = sum;

Subarray.push\_back(i);

Subarray.push\_back(j);

}

}

}

return max\_sum;

* + - 1. Efficient approach:
         1. Time Complexity: O(n);
         2. Space Complexity: O(1);

Algo:

Int msf = INT\_MIN, meh = 0;

Int s = 0;

For(I = 0 🡪 nums.size()-1;i++){

Meh += nums[i];

If(meh > msf){

Subarray.clear();

Msf = meh;

Subarray.push\_back(s);

Subarray.push\_back(i);

}

If(meh < 0){

Meh = 0;

S = i+1;

}

}

Return msf;

1. Sort 0,1, & 2
   1. Time Complexity: O(n)
   2. Space Complexity: O(1)
      1. Algo:

Int lo =0 ,mid = 0, high = size;

While(mid <= hi){

Switch (nums[mid]){

Case 0:

Swap(nums[lo++], nums[mid++]);

Break;

Case 1:

Mid++;

Break;

Case 2:

Swap(nums[mid],nums[hi--]);

Break;

}

}

1. Stock Buy and Sell
   1. Naïve Solution:
      1. Time Complexity: O(n2)
      2. Space Complexity: O(1)
         1. Algo:

Int maxPro = 0;

Int n = size;

For(I 🡪 0 to n-1; i++){

For(j = i+1 🡪 n-1;j++){

If(arr[j] > arr[i]){

maxPro = max(arr[j]-arr[i], maxPro);

}

}

}

Return maxPro;

* 1. Efficient Solution:
     1. Time Complexity: O(n);
     2. Space Complexity: O(1);

Algo:

Int maxPro = 0;

Int n = size;

Int minPrice = INT\_MAX;

For(I = 0 🡪 n-1, i++){

minPrice = min(minPrice, arr[i]);

maxPro = max(maxPro, arr[i]-minPrice);

}

Return maxPro;

1. Rotate Matrix
   1. Naïve Efficient:
      1. Time Complexity: O(n\*n);
      2. Space Complexity: O(n\*n);

Algo:

Vector<vector<int>> roasted(n, vector<int>(n,0));

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

For(int j = o;j<n;j++){

Roasted[j][n-i-1] = matrix[i][j];

}

}

Return roasted;

* 1. Efficient Approach:
     1. Time Complexity: O(n\*n)+O(n\*n)
     2. Space Complexity: O(1)
        1. Algo:

For(int I = 0;i<n;i++){

For(int j = 0;j<I;j++){

Swap(matrix[i][j], matrix[j][i]);

}

}

For(int I =0;i<n;i++){

Reverse(matrix[i].begin(),matrix[i].end());

}

1. Merge Overlapping
   1. Naive approach
      1. Time Complexity: O(Nlog(n))+O(n)
      2. Space Complexity: O(n)
         1. Algo:

Sort(arr.begin(),arr.end());

For(I = 0🡪n-1;i++){

Start = arr[i].first, end = arr[i].second;

If(!ans.empty()){

If(start<=ans.back().second){

Continue;

}

}

For(j = i+1🡪 n-1;j++){

If(arr[j].first <= end){

End = max(end,arr[j].second);

}

}

End = max(end,arr[i].second);

Ans.push\_back({start,end});

}

Return ans;

* 1. Efficient Approach
     1. Time Complexity: O(nlogn)+O(n);
     2. Space Complexity: O(n);

1. Merge sorted Array
   1. Time Complexity: O(nlogn)+O(n);
   2. Space Complexity: O(1);
2. Duplicate Integer in an array of n1 integer
   1. Naïve Approach:
      1. Time Complexity : O(nlogn+n)
      2. Space Complexity: O(1);

Algo:

N = arr.size();

Sort(arr.begin(),arr.end());

For(I = 0 🡪 n-2;i++){

If(arr[i] == arr[i+1]){

Return arr[i];

}

}

* 1. Efficient approach (Floyds algorithm):
     1. Time Complexity: O(n);
     2. Space Complexity: O(1);

Algo:

Slow = nums[0];

Fast = nums[0];

do{

slow = nums[slow];

fast = nums[nums[fast]];

}while(slow != fast);

Fast = nums[0];

While(slow!=fast){

Slow = nums[slow];

Fast = nums[fast];

}

Return slow;

1. Inversion of array
   1. Using Merge sort
      1. Time Complexity: O(nlogn)
      2. Space Complexity: O(n)

Algo:

int merge(int arr[],int temp[],int left,int mid,int right)

{

int inv\_count=0;

int i = left;

int j = mid;

int k = left;

while((i <= mid-1) && (j <= right)){

if(arr[i] <= arr[j]){

temp[k++] = arr[i++];

}

else

{

temp[k++] = arr[j++];

inv\_count = inv\_count + (mid - i);

}

}

while(i <= mid - 1)

temp[k++] = arr[i++];

while(j <= right)

temp[k++] = arr[j++];

for(i = left ; i <= right ; i++)

arr[i] = temp[i];

return inv\_count;

}

int merge\_Sort(int arr[],int temp[],int left,int right)

{

int mid,inv\_count = 0;

if(right > left)

{

mid = (left + right)/2;

inv\_count += merge\_Sort(arr,temp,left,mid);

inv\_count += merge\_Sort(arr,temp,mid+1,right);

inv\_count += merge(arr,temp,left,mid+1,right);

}

return inv\_count;

}

1. Searching in a 2d matrix
   1. Naïve Approach
      1. Time Complexity: O(m\*n);
      2. Space Complexity: O(1);
   2. Effcient Approach
      1. Time Complexity: log(m\*n);
      2. Space Complexity: O(1);

Algo:

If(!matrix.size())

Return false;

Intlo = 0;

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;

1. Power (x,n):
   1. Naïve approach:
      1. Time Complexity: O(n);
      2. Space Complexity: O(1);
         1. Algo:

Double ans = 1.0;

For(I =0 🡪n-1;i++){

Ans = ans\*x;

}

Return ans;

* 1. Efficient Approach (Binary Exponentiation):
     1. Time Complexity: O(logN);
     2. Space Complexity: O(1)

Algo :

Double ans = 1.0;

Long long nn = n;

If(nn < 0){

-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;

1. Reverse Pairs
   1. Naïve approach
      1. Time Complexity: O(n2)
      2. Space Complexity: O(1)

Algo:

int pairs = 0;

for(I = 0 🡪 arr.size()-1;i++){

for(j=i+1🡪arr.size()-1;j++){

if(arr[i] > 2\*arr[j]){

pairs++;

}

}

}

Return pairs;

* 1. Efficient approach:
     1. Time Complexity: O(nlogn)+O(n)+O(n)
     2. Space Complexity: O(N)
        1. Algo:

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;

}

1. Grid Unique Paths
   1. Naïve Approach:
      1. Time Complexity: O(exponential)
      2. Space Complexity: exponential

Algo:

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);

}

* 1. Efficient Approach (Dynamic Programming):
     1. Time Complexity: O(m\*n)
     2. Space Complexity: O(1)

Algo:

For( I =1;i<m;i++){

For(j = 1🡪n-1;j++){

Dp[j] += dp[j-1];

}

}

Return dp[n-1];

1. Majority Elements (>N/2)
   1. Naïve Approach :
      1. Time Complexity: O(n)
      2. Space Complexity: O(n)
   2. Efficient Approach:
      1. Time Complexity: O(N)
      2. Space Complexity: O(1)

Algo :

Initialize count ,candidate = 0;

For(int num : nums){

If(count == 0){

Candidate = num;

}

If(num == candidate) count++;

Else{

Count--;

}

}

Return candidate;

1. Majority Elements (>N/3 elements)
   1. Naïve Approach :
      1. Time Complexity : O(n)
      2. Space Complexity : O(n)
   2. Efficient approach:
      1. Time Complexity: O(n)
      2. Space Complexity: O(1)

Algo:

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;

1. 2-sum Problem
   1. Naïve Approach
      1. Time Complexity: O(n2)
      2. Space Complexity:O(1)
   2. Efficient Approach:
      1. Time Complexity: O(nlogn)
      2. Space Complexity: O(1)
2. 4sum Problem
   1. Naïve approach:
      1. Time Complexity: O(nlogn+n^3logn)
      2. Space Complexity: O(1)
   2. Efficient approach:
      1. Time Complexity: O(n^3)
      2. Space Complexity: O(1)

Algo:

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

        int n = nums.size();

        vector<vector<int>> ans;

        for(int i = 0;i<n;i++){

            for(int j = i+1;j<n;j++){

                int start = j+1 , end = n-1;

                long long remain = (long long)target - nums[i] - nums[j];

                while(start < end){

                    if(nums[start]+nums[end] == remain){

                        ans.push\_back({nums[i],nums[j],nums[start],nums[end]});

                        start++;

                        end--;

                        while(start<end && nums[start-1] == nums[start]){

                            start++;

                        }

                    }else if(nums[start]+nums[end] > remain){

                        end--;

                    }else{

                        start++;

                    }

                }

                while(j+1 < n && nums[j] == nums[j+1]){

                    j++;

                }

            }

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

                i++;

            }

        }

        return ans;

1. Longest Consecutive Sequence
   1. Naïve Approach:
      1. Time Complexity: O(nlogn)
      2. Space Complexity: O(1)

Algo:

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

Int ans = 1;

Int prev = nums[0];

Int cur = 1;

For(I = 1🡪nums.size()-1;i++){

If(nums[i] == prev + 1){

Cur++;

}else if(nums[i] != prev){

Cur = 1;

}

Prev= nums[i];

Ans = max(ans,cur);

}

Return ans;

* 1. Efficient approach:
     1. Time Complexity: O(n)
     2. Space Complexity: O(1)

Algo:

Set<int> HashSet;

For(int num:nums){

HashSet.insert(num);

}

Int longestStreak = 0;

For(int num:nums){

If(!hashSet.count(num-1)){

Int currentNum = num;

Int currentStreak = 1;

While(hashSet.count(currentNum+1)){

currentNum += 1;

currentStreak += 1;

}

longestStreak = max(longestStreak,currentStreak);

}

}

Return longestStreak;

1. Length of longest subarray of sum zero
   1. Naïve approach
      1. Time Complexity: O(n^2)
      2. Space Complexity: O(1)

Algo:

Int Max\_nu = 0;

For(int I = 0🡪 n-1;i++){

Sum= 0;

For(j = i🡪n-1;i++){

Sum +=a[j];

If(sum == 0){

Max\_nu = max(Max\_nu,j-i+1);

}

}

}

Return Max\_nu;

* 1. Efficient Approach:
     1. Time Complexity: O(N)
     2. Space Complexity: O(n)

Algo:

Unordered\_map<int,int> mpp;

Int maxi = 0;

Int sum = 0;

For(I = 0 🡪 n-1;i++){

Sum+=a[i];

If(sum==0){

Maxi = i+1;

}else{

If(mpp.find(sum) != mpp.end()){

Maxi = max(maxi,i-mpp[sum]);

}else{

Mpp[sum] = i;

}

}

}

Return maxi;

1. Length of longest substring without any repeating character.
   1. Naïve Approach
      1. Time Complexity: O(n2)
      2. Space Complexity: O(n)

Algo:

If(str.size() == 0){

Return 0;

}

Int maxans = INT\_MIN;

For(I = 0 🡪 n-1;i++){

Unordered\_set<int> set;

For(j = I🡪n-1;i++){

If(set.find(str[j] != set.end()){

Maxans = max(maxans,j-i);

Break;

}

Set.insert(str[j]);

}

}

Return maxans;

* 1. Efficient approach
     1. Time Complexity: O(n)
     2. Space Complexity: O(n)

ALgo:

Vector<int> mpp(256,-1);

Int left = 0,right = 0;

Int n = s.size();

Int len = 0;

While(right < n){

If(mpp[s[right]] != -1){

Left = max(mpp[s[right]]+1,left);

}

Mpp[s[right]] = right;

Len = max(len,right-left+1);

Right++;

}

Return len;

1. 3 sum
   1. Naïve Approach
      1. Time Complexity: O(n3)
      2. Space Complexity: O(1)

Algo:

For(I =0 🡪 n-3;i++){

For(j = 0🡪n-2;j++){

For(k = j+1🡪n;k++){

Temp.clear();

If(nums[i]+nums[j]+nums[k] == 0){

Temp.push\_back(nums[i]);

Temp.push\_back(nums[j]);

Temp.push\_back(nums[k]);

}

If(temp.size() != 0){

Ans.push\_back(temp);

}

}

}

}

Return ans;

* 1. Efficient Approach
     1. Time Complexity: O(n2)
     2. Space Complexity: O(3k)

Algo:

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

// moves for a

for (int i = 0; i < (int)(num.size())-2; i++) {

if (i == 0 || (i > 0 && num[i] != num[i-1])) {

int lo = i+1, hi = (int)(num.size())-1, sum = 0 - num[i];

while (lo < hi) {

if (num[lo] + num[hi] == sum) {

vector<int> temp;

temp.push\_back(num[i]);

temp.push\_back(num[lo]);

temp.push\_back(num[hi]);

res.push\_back(temp);

while (lo < hi && num[lo] == num[lo+1]) lo++;

while (lo < hi && num[hi] == num[hi-1]) hi--;

lo++; hi--;

}

else if (num[lo] + num[hi] < sum) lo++;

else hi--;

}

}

}

return res;

1. Trapping Rain Water
   1. Naïve Approach
      1. Time Complexity: O(n\*n)
      2. Space Complexity: O(1)

Algo:

Int n = arr.size();

Int waterTrapped = 0;

For(int I = 0🡪n-1;i++){

Int j = I;

Int leftmax = 0, rightMax = 0;

While(j >= 0){

leftMax = max(leftMax,arr[j]);

j--;

}

J = I;

While(j < n){

rightMax = max(rightMax,arr[j]);

j++;

}

waterTrapped += min(leftMax,rightMax)-arr[i];

}

Return waterTrapped;

* 1. Efficient Approach
     1. Time Complexity: O(3\*n)
     2. Space Complexity: O(n)

Algo:

Int n = arr.size();

Int prefix[n], suffix[n];

Prefix[0] = arr[0];

For(int I = 1🡪 n-1;i++){

Prefix[i] = max(prefix[i-1],arr[i]);

}

Suffix[n-1] = arr[n-1];

For(int I = n-2🡪0;i--){

Suffix[i] = max(suffix[i+1],arr[i]);

}

Int waterTrapped = 0;

For(I = 0🡪n-1;i++){

waterTrapped += min(prefix[i],suffix[i])-arr[i];

}

Return waterTrapped;

1. Remove Duplicate from Sorted Arrays
   1. Naïve Approach (Using set):
      1. Time Complexity: O(nlogn + n);
      2. Space Complexity: O(n)
   2. Efficient Approach:
      1. Time Compleity: O(n);
      2. Space Complexity: O(1)

Algo:

int i = 0;

for (int j = 1; j < n; j++) {

if (arr[i] != arr[j]) {

i++;

arr[i] = arr[j];

}

}

return i + 1;

1. Remove Duplicate from Sorted Arrays
   1. Naïve approach:
      1. Time Complexity: O(n)
      2. Space Complexity: O(1)

Algo:

Int Cnt = 0,maxi = 0;

For(I = 0🡪 n-1;i++){

Cnt = 0;

Maxi = 0;

For(I = 0🡪n-1;i++){

If(nums[i] == 1){

Cnt++;

}else{

Cnt = 0;

}

Maxi = max(maxi,cnt);

}

}

1. Subset Sums
   1. Naïve Approach
      1. Time Complexity: O(2n+2nlog2n)
      2. Space Complexity: O(2n)

Algo:

Vector<int> ans;

subsetSumHelper(0,arr,n,ans,0);

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

Sort(

If(ind == n){

Ans.push\_back(sum);

Return;

}

subsetSumHelper(ind+1,arr,n,ans,sum+arr[ind]);

subsetSumHelper(inde+1,arr,n,ans,sum);

return ans;

1. Print all unique subset of a array
   1. Naïve approach
      1. Time Complexity: O(2n(klog(x)));
      2. Space Complexity: O(2nk);

Algo:

Vector<vector<int>> ans;

Set<vector<int>> res;

Vector<int> ds;

Fun(nums,0,ds,res);

For(auto it = res.begin(); it!=res.end();it++){

Ans.push\_back(\*it);

}

Return ans;

**Fun(function)**

If(index == nums.size()){

Sort(ds.begin(),ds.end());

Res.insert(ds);

}

Ds.push\_back(nums[index]);

Fun(nums,index+1,ds,res);

Ds.pop\_back();

Fun(nums,index+1,ds,res);

* 1. Efficient Approach
     1. Time Complexity: O(2n);
     2. Space Complexity: O(2nk);

Algo:

**nonDuplicate(Function)**

ans.push\_back(temp);

for(I = index ;i<n;i++){

if(i!= index && nums[i] == nums[i-1]){

continue;

}

}

Temp.push\_back(nums[i]);

nonDuplicate(i+1,nums,temp,ans,n);

temp.pop\_back();

**subSets(Function)**

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

vector<vector<int>> ans;

vector<int> temp;

int n = nums.size();

nonDuplicates(0,nums,temp,ans,n);

return ans;

1. Combination Sum -1
   1. Naive Approach
      1. Time Complxeity: O(2tk);
      2. Space Complexity: O(kx);

Algo:

**Function1:**

If(index == n){

If(target == 0){

Ans.push\_back(temp);

}

Return;

}

If(candidates[index] <= target){

Temp.push\_back(candidates[index]);

findCombination(index,candidates,target-candidates[index],ans,n,temp);

temp.pop\_back();

}

findCombination(index+1,candidates,target,ans,n,temp);

**Function 2:**

int n = nums.size();

vector<vector<int>> ans;

vector<int> temp;

findCombination(0,candidates,target,ans,n,temp);

return ans;

1. Combination -2
   1. Time Complexity: O(2nk);
   2. Space Complexity: O(kx);

Algo:

If(target == 0){

Ans.push\_back(temp);

Return;

}

For(int I = index;i<candidates.size();i++){

If(i>index && candidates[i] == candidates[i-1]){

Continue;

}

If(candidates[i]>target){

Break;

}

Temp.push\_back(candidates[i]);

findCombination(i+1,candidates,target-candiates[i],ans,temp,n);

temp.pop\_back();

}

**Function 2**

Sort(canddiates.begin(),candidates.end());

Int n = candidates,size();

Vector<vector<int>> ans;

Vector<int> temp;

findCombination(0,candiadates,target,ans,temp,n);

return ans;

1. Palindrome Partitioning
   1. Naïve Approach
      1. Time Complexity: **O( (2^n) \*k\*(n/2) )**
      2. **Space Complexity: O(k\*x)**
         1. **Reason:** k is the average length of the list of palindromes and if we have x such list of palindromes in our final answer

ALGO:

**Function 1**

bool isPalin(string s,int start,int end){

while(start <= end){

if(s[start++] == s[end--]){

return false;

}

}

}

**Function 2**

if(index == s.size()){

ans.push\_back(path);

return;

}

For( i=index🡪n-1;i++){

If(isPalin(s,index,i)){

Path.push\_back(s.substr(index,i-index+1));

Partition(i+1,s,ans,path);

Path.pop\_back();

}

}

**Function 3:**

Vector<vector<string>> ans;

Vector<string> path;

Partition(0,s,ans,path);

Return ans;

1. Kth Permutation Sequence
   1. Naïve Approach
      1. Time Complexity:  **O(N! \* N) +O(N! Log N!)**
      2. **Space Complexity: O(n)**

**Algo:**

void permutationHelper(string & s, int index, vector < string > & res) {

if (index == s.size()) {

res.push\_back(s);

return;

}

for (int i = index; i < s.size(); i++) {

swap(s[i], s[index]);

permutationHelper(s, index + 1, res);

swap(s[i], s[index]);

}

}

string getPermutation(int n, int k) {

string s;

vector < string > res;

for (int i = 1; i <= n; i++) {

s.push\_back(i + '0');

}

permutationHelper(s, 0, res);

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

auto it = res.begin() + (k - 1);

return \*it;

}

* 1. Efficient Approach
     1. Time Complexity: O(N2)
     2. Space Complexity: O(N)

Algo:

string getPermutation(int n, int k) {

int fact = 1;

vector < int > numbers;

for (int i = 1; i < n; i++) {

fact = fact \* i;

numbers.push\_back(i);

}

numbers.push\_back(n);

string ans = "";

k = k - 1;

while (true) {

ans = ans + to\_string(numbers[k / fact]);

numbers.erase(numbers.begin() + k / fact);

if (numbers.size() == 0) {

break;

}

k = k % fact;

fact = fact / numbers.size();

}

return ans;

}

1. All Permutation String and Array
   1. Naïve Approach
      1. Time Complexity: O(N!\*N)
      2. Space Complexity: O(n)

Algo:

**Function 1:**

If(ds.size() == nums.size()){

Ans.push\_back(ds);

Return;

}

For(I = 0🡪n-1;i++){

If(!freq[i]){

Ds.push\_back(nums[i]);

freq[i] = 1;

recurPermute(ds,nums,ans,freq);

freq[i] = 0;

ds.pop\_back();

}

}

**Function 2:**

Vector<vector<int>> ans;

Vector<int> ds;

Int freq[nums.size()];

For(int I = 0;i<nums.size();i++){

Freq[i] = 0;

}

recurPermute(ds,nums,ans,freq);

return ans;

1. N Queens Problem
   1. Naïve Approach:
      1. Time Compleity: O(n!\*n)
      2. Space Complexity: O(n2)

Algo:

**Function 1:**

Int duprow = row;

Int dupcol = col;

While(row>=0 &&col>=0){

If(board[row][col] == ‘Q’)

Return false;

Row--;

Col--;

}

Col = dupcol;

Row = duprow;

While(col>=0){

If(board[row][col] == ‘Q’)

Return false;

Col--;

}

Row = duprow;

Col = dupcol;

While(row < n && col >= 0){

If(board[row][col] == ‘Q’)

Return false;

Row++;

Col--;

}

Return true;

**Function 2:**

If(col == n){

Ans.push\_back(board);

Return;

}

For(row = 0🡪n-1;row++){

If(isSafe(row,col,board,n)){

Board[row][col] = ‘Q’;

Solve(col+1,board,ans,n);

Board[row][col] = ‘.’;

}

}

**Function 3:**

Vector<vector<string>> ans;

Vector<string> board(n);

String s(n,’.’);

For(int I = 0;i<n;i++){

Board[i] = s;

}

Solve(0,board,ans,n);

Return ans;

* 1. Efficient Approach:
     1. Time Complexity: O(n!\*n)
     2. Space Complexity: O(n)

Algo:

void solve(int col, vector < string > & board, vector < vector < string >> & ans, vector < int > & leftrow, vector < int > & upperDiagonal, vector < int > & lowerDiagonal, int n) {

if (col == n) {

ans.push\_back(board);

return;

}

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

if (leftrow[row] == 0 && lowerDiagonal[row + col] == 0 && upperDiagonal[n - 1 + col - row] == 0) {

board[row][col] = 'Q';

leftrow[row] = 1;

lowerDiagonal[row + col] = 1;

upperDiagonal[n - 1 + col - row] = 1;

solve(col + 1, board, ans, leftrow, upperDiagonal, lowerDiagonal, n);

board[row][col] = '.';

leftrow[row] = 0;

lowerDiagonal[row + col] = 0;

upperDiagonal[n - 1 + col - row] = 0;

}

}

}

vector < vector < string >> solveNQueens(int n) {

vector < vector < string >> ans;

vector < string > board(n);

string s(n, '.');

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

board[i] = s;

}

vector < int > leftrow(n, 0), upperDiagonal(2 \* n - 1, 0), lowerDiagonal(2 \* n - 1, 0);

solve(0, board, ans, leftrow, upperDiagonal, lowerDiagonal, n);

return ans;

}

1. Sudoku Solver
   1. Approach 1:
      1. Time Complexity: O(9n^2);
      2. Space Complexity: O(1);

Algo:

**Function 1:**

For(I = 0🡪8;i++){

If(board[i][col] == c)

Return false;

If(board[row][i] == c)

Return false

If(board[3\*(row/3)+i/3][3\*(col/3)+i%3] == c){

Return false

}

Return true;

**Function 2:**

For(I = 0🡪board.size()-1;i++){

For(j= 0🡪board[0].size()-1;i++){

If(board[i][j] == ‘.’){

For(char c = ‘1’;c<= ‘9’;c++){

If(isValid(board,I,j,c)){

Board[i][j] = c;

If(solveSudoku(board)){

Return true;

}else{

Board[i][j] = ‘.’;

}

}

}

Return false;

}

}

}

Return true;

1. Rat in a maze
   1. Time Complexity: O(4(m\*n));
   2. Space Complexity: O(m\*n)

Algo:

**Function 1**:

Vector<string> ans;

Vector<vector<int>> vis(n, vector<int> (n,0));

If(m[0][0] == 1){

findPathHelper(0,0,m,n,ans, “”, vis);

return ans;

**Function 2**:

If(I == n-1 && j == n-1){

Ans.push\_back(move);

Return;

}

If(I + 1<n && !vis[i+1][j] && a[i+1][j] == 1){

Vis[i][j] = 1;

findpathHelper(i+1,j,a,n,ans,move+ ‘D’,vis);

vis[i][j] = 0;

}

If(j-1 >= 0 && !vis[i][j-1] && a[i][j-1] == 1){

Vis[i][j] = 1;

findPathHelper(I,j-1,a,n,ans,move+ ‘L’,vis);

vis[i][j] = 0;

}

If(j+1 < n && !vis[i][j+1] && a[i][j+1] == 1){

Vis[i][j] = 1;

findPathHelper(I,j+1,a,n,ans,move+ ‘R’,vis);

vis[i][j] = 0;

}

If(i-1 >= 0 && !vis[i-1][j] && a[i-1][j] == 1){

Vis[i][j] = 1;

findPathHelper(i-1,j,ans,move+ ‘U’,vis);

vis[i][j] = 0;

}

1. Nth root of an integer
   1. Time Complexity: O(n\*log(m\*10d));
   2. Space Complexity: O(1)

Algo:

**Function 1**:

Double ans = 1.0;

For(I = 1🡪n;i++){

Ans = ans\*number;

}

Return ans;

**Function 2**:

Double low = 1;

Double high = m;

Double eps = 1e-7;

While((high – low) > eps){

Double mid = (low+high) >>1;

If(multiply(mid,n) < m){

Low = mid;

}else{

High = mid;

}

}

1. Median of row wise sorted matrix
   1. Naïve Approach:
      1. Time Complexity: O(r\*c(log(r\*c)));
      2. Space Complexity: O(r\*c);

Algo:

int Findmedian(int arr[3][3], int row, int col)

{

int median[row \* col];

int index = 0;

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

{

for (int j = 0; j < col; j++)

{

median[index] = arr[i][j];

index++;

}

}

return median[(row \* col) / 2];

}

* 1. Efficient Approach (Binary S):
     1. Time Complexity: O(r\*log c);
     2. Space complexity: O(1);

Algo:

**Function 1:**

Int l = 0,h = row.size()-1;

While(l < = h){

Int md = (l+h)>>1;

If(row[md] <= mid)

{

L = md+1;

}else{

H = md-1;

}

}

Return l;

**Function 2:**

int low = 1;

int high = 1e9;

int n = A.size();

int m = A[0].size();

while(low <= high)

{

Int mid = (low + high) >>1;

Int cnt = 0;

For( I =0🡪n-1;i++){

Cnt += countSmallerThanMid(A[i],mid);

}

If(cnt <= (n\*m)/2)

Low = mid + 1;

Else

High = mid-1;

}

Return low;

1. Single Element in sorted Array
   1. Naïve Approach
      1. Time Complexity: O(n)
      2. Space Complexity: O(1)

Algo:

int n = nums.size();

int elem = 0;

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

elem = elem ^ nums[i];

}

return elem;

* 1. Efficient Approach:
     1. Time Complexity: O(log(n))
     2. Space Complexity: O(1)

Algo:

int low = 0;

int high = n - 2;

while (low <= high) {

int mid = (low + high) / 2;

if (mid % 2 == 0) {

if (nums[mid] != nums[mid + 1])

high = mid - 1; //Shrinking the right half

else

low = mid + 1; //Shrinking the left half

} else {

if (nums[mid] == nums[mid + 1])

high = mid - 1; //Shrinking the right half

else

low = mid + 1; //Shrinking the left half

}

}

return nums[low];

* 1. Efficient Approach 2:
     1. Time Complexity: Olog(n));
     2. Space Complexity: O(1);

Algo:

int singleNonDuplicate(vector<int>& nums) {

int low = 0,high = nums.size()-2;

while(low <= high){

int mid = (low+high)>>1;

if(nums[mid] == nums[mid^1]){

low = mid+1;

}else{

high = mid-1;

}

}

return nums[low];

}

1. Search Element in sorted in rotated & sorted Array
   1. Naïve Approach
      1. Time Complexity: O(n)
      2. Space Complexity: O(1)
   2. Efficient Approach
      1. Time Complexity: O(log(n))
      2. Space Complexity: O(1)

Algo:

Int low = 0,high = nums.size()-1;

While(low <= high){

Int mid = (low + high)>>1;

If(nums[mid] == target){

Return mid;

}

If(nums[low] <= nums[mid]){

If(nums[low] <= target && nums[mid]>= target){

High = mid-1;

}else{

Low = mid+1;

}

}else{

If(nums[mid] <= target && target <= nums[high]){

Low = mid+1;

}else{

High = mid-1;

}

}

}

Return -1;

1. Median of two sorted arrays
   1. Naive Approach
      1. Time Complexity: O(m+n)
      2. Space Complexity: O(m+n)

Algo:

* + - 1. Create a vector of size m+n and store all the value in sorted manner and find the median.
      2. If(n%2==1)
         1. Return finalArray[((n+1)/2)-1];
      3. Else
         1. Return ((float)finalArray[(n/2)-1]+(float)finalArray[(n/2)])/2
  1. Efficient Approach:
     1. Time complexity: O(log(min(m,n)));
     2. Space Complexity: O(1)

Algo:

if(m>n)

return median(nums2,nums1,n,m);

int low=0,high=m,medianPos=((m+n)+1)/2;

while(low<=high) {

int cut1 = (low+high)>>1;

int cut2 = medianPos - cut1;

int l1 = (cut1 == 0)? INT\_MIN:nums1[cut1-1];

int l2 = (cut2 == 0)? INT\_MIN:nums2[cut2-1];

int r1 = (cut1 == m)? INT\_MAX:nums1[cut1];

int r2 = (cut2 == n)? INT\_MAX:nums2[cut2];

if(l1<=r2 && l2<=r1) {

if((m+n)%2 != 0)

return max(l1,l2);

else

return (max(l1,l2)+min(r1,r2))/2.0;

}

else if(l1>r2) high = cut1-1;

else low = cut1+1;

}

return 0.0;

1. Kth Element of two sorted arrays
   1. Naïve Approach
      1. Time Complexity: O(k)
      2. Space Complexity: O(1)

Algo:

Function1:

Int p1 = 0,p2 = 0,counter = 0,answer = 0;

While(p1<m && p2<n){

If(counter == k) break;

Else if(array[p1] < array2[p2]){

Answer = array1[p1];

++p1;

}else{

Answer = array[p2];

++p2;

}

++counter;

}

If(counter != k){

If(p1 != m-1)

Answer = array1[k-counter];

Else

Answer = array2[k-counter];

}

Return answer;

* 1. Efficient Approach
     1. Time Complexity: O(log(min(m,n)))
     2. Space Complexity: O(1)

Algo:

If(m>n){

Return kthElement(arr2,arr1,n,m,k);

}

Int low = max(0,k-m),high = min(k,n);

While(low<=high){

Int cut1 = (low+high)>>1;

Int cut2 = k-cut1;

Int l1 = cut1 == 0? INT\_MIN:arr1[cut1-1];

Int l2 = cut2 ==0? INT\_MIN:arr2[cut2-1];

Int r1 = cut1 == n?INT\_MAX:arr1[cut1];

Int r2 = cut2 == m?INT\_MAX:arr2[cut2];

If(l1 <= r2 && l2<=r1)

Return max(l1,l2);

Else if(l1>r2){

High = cut1-1;

}else{

Low = cut1+1;

}

}

Return 1;

1. Allocate minimum number of pages
   1. Naïve Approach
      1. Time Complexity: O(nlogn)
      2. Space Complexity: O(1)

Algo:

Function 1:

Int cnt = 0;

Int sumAllocated = 0;

For(I = 0🡪n-1;i++)

If(sumAllocated + A[i] > pages)

Cnt++;

sumAllocated = A[i];

if(sumAllocated > pages) return false;

}else{

sumAlocated += A[i];

}

}

If(cnt < students) return true;

Return false;

Function 2:

If(B > A.size()) return -1;

Int low = A[0];

Int high = 0;

For(i=0🡪n-1){

High = high + a[i]

Low = min(low,A[i]);

}

While(low <= high){

Int mid = (low + high)>>1;

If(isPossible(A,mid,B)){

High = mid-1;

}

Else{

Low = mid+1;

}

}

Return low;

1. Aggressive Cows
   1. Naïve Approach
      1. Time Complexity: O(n\*m)
      2. Space Cmplexity: O(1)

Algo:

Int n = inp.size();

Int k = inp[0];

Cows--;

For(I = 1🡪n-1;i++){

If(inp[i]-k >= dist){

Cows--;

If(!cows){

Return true;

}

K = inp[i];

}

}

Return false;

Sort(a,a+n);

maxD = inp[n-1]-inp[0];

ans = INT\_MIN;

for(d = 1🡪maxD;d++){

bool possible = isCompatible(inp,d,m);

if(possible){

ans = max(ans,d);

}

}

* 1. Efficient Approach:
     1. Time Complexity: O(n\*log(m));
     2. Space Complexity: O(1)

Algo:

Int cntCows = 1;

Int lastPlacedCows = a[0];

For(I = 1🡪n-1;i++){

If(a[i]-lastPlacedCow >= minDist){

cntCows++;

lastPlacedCows = a[i]

}

}

If(cntCows >= cows) return true;

Return false;

**FUNCTION MAIN:**

Sort(a,a+n);

Low = 1,high = a[n-1]-a[0];

While(low <= high){

Mid = (low+high)>>1;

If(possible(a,n,cows,mid)){

Low = mid+1;

}else{

High = mid-1;

}

}