Arrays :

```
int arr[] = {12, 11, 13, 5, 6, 7};
int size = sizeof(arr) / sizeof(arr[0]);
// When we have a static array
```

Given an array of integers, return the second smallest number

```
int largest = a[0];
int second_largest = INT_MIN;

for(int i = 1;i < n;i++) {
    if(a[i] > largest) {
        second_largest = largest;
        largest = a[i];
    }
    else if(a[i] > second_largest && a[i] != largest) {
    // IT IS IMP WE MAKE SURE TO HANDLE DUPLICARTES
        second_largest = a[i];
    }
}
```

Given an array with duplicity allowed, change the array so only unique elements are stored along with the length of it

```
int removeDuplicates(vector<int>& nums) {
    int index = 1;
    int n = nums.size();

    for (int i = 1; i < n; i++) {
        if (nums[i] != nums[index - 1]) {
        // TWO POINTER APPROACH
            nums[index] = nums[i];
            index++;
        }
    }
    return index;
}
```

Given an array right rotate it k steps

```
// SOLUTION 1
// NO EXTRA SPACE O(2nlogn)
```

```
void rotate(vector<int>& nums, int k) {
  k = k % nums.size();
  if(k == 0) return;

  reverse(nums.end()-k, nums.end());
  reverse(nums.begin(), nums.end()-k);
  reverse(nums.begin(), nums.end());

  return;
}
```

```
// SOLUTION 2
// EXTRA SPACE O(2n)

void rotate(vector<int>& nums, int k) {
  int n = nums.size();
  k = k % n;
  if (k == 0) return;

  vector<int> temp(k);

  for (int i = n - k; i < n; i++) {
      temp[i - (n - k)] = nums[i];
  }

  for (int i = n - k - 1; i >= 0; i--) {
  // IF I START FROM 0 OVERWRITING HAPPENS ON THE ELEMENTS WHICH IS AHEAD.
  // HENCE DO IT BACKWARD
      nums[i + k] = nums[i];
  }

  for (int i = 0; i < k; i++) {
      nums[i] = temp[i];
  }

  return;
}
```

Given an array having some zeroes, move all of them to end

```
void moveZeroes(vector<int>& nums) {
  int n = nums.size();
  int left = 0;

  for(int right = 0;right < n;right++) {
      if(nums[right] != 0) {
```

```
            swap(nums[right], nums[left]);
            // WHY THIS WORKS ?
            // 0, 5 WHEN SWAPPED LEFT BECOMES 0. OR 0, 0, 5
            // WHEN SWAPPED LEFT STILL BECOMES A ZERO.
            // HENCE IN BOTH CASES LEFT ALWAYS POINTS TO A ZERO
            left++;
        }
    }

    return;
}
```

Given two sorted array where duplicity is allowed, return union of these two ensuring there are only unique elements in it

```
// BRUTE FORCE
// USING ORDERED SET
set<int> st;

for(auto i : arr1) {
    st.insert(i);
}

for(auto i : arr2) {
    st.insert(i);
} // THEN CREATE A VECTOR PUSH_BACK
  // ALL ELEMENTS FROM ST AND RETURN
INSERTION IN OREDERED SET TAKES logn TIME
  // why ? It is maintained using a BST
```

```
// TWO POINTER APPROACH\
// O(n1 + n1)
vector<int> sortedArray(vector<int> a, vector<int> b) {
    vector<int> ans;
    int i = 0, j = 0;
    int n1 = a.size(), n2 = b.size();

    while (i < n1 && j < n2) { SIMILAR TO MERGE SORT
        if (a[i] < b[j]) {
            if (ans.empty() || ans.back() != a[i]) { NO DUPLICACY
                ans.push_back(a[i]);
            }
            i++;
        } else if (a[i] > b[j]) {
            if (ans.empty() || ans.back() != b[j]) {
                ans.push_back(b[j]);
```

```
            }
            j++;
        } else { // a[i] == b[j]
            if (ans.empty() || ans.back() != a[i]) {
                ans.push_back(a[i]);
            }
            i++;
            j++;
        }
    }

    // Process remaining elements in a
    while (i < n1) {
        if (ans.empty() || ans.back() != a[i]) {
            ans.push_back(a[i]);
        }
        i++;
    }

    // Process remaining elements in b
    while (j < n2) {
        if (ans.empty() || ans.back() != b[j]) {
            ans.push_back(b[j]);
        }
        j++;
    }

    return ans;
}
```

Similar Question to previous where return intersection where duplicity is allowed ?

```
// BRUTE FORCE
// keep a visited array for arr2 {as duplicity is allowed}
/*
    for each element in arr1 check if it exists in arr2 and it is not visited. DONE!
*/
```

```
// TWO POINTER SOLUTOION
vector<int> findArrayIntersection(vector<int> &a, int n,
                                                    vector<int> &b, int m)
{
    int left = 0;
    int right = 0;
    vector<int> ans;
```

```
    while(left < n && right < m) {
        if(a[left] < b[right]) left++;
        else if(b[right] < a[left]) right++;
        else {
            ans.push_back(a[left]);
            // if duplicacy is not allowed
            // ans.empty() || ans.back() != a[left] add this condition
            left++;
            right++;
        }
    }

    return ans;
}
```

Given an array of size n-1

- There are elements in that array from 1 to n, but one number is missing return it.

```
// Brute force
// from 1 -> n check if that element is present
                        // in arr if not return it O(n*2)
// Better approach
// keep a visited array {Hashing} then traverse and
// check which one is missing T -> O(2n) and S -> O(n)
// optimal
// return n*(n+1) / 2 - sum of numbers in array :
// but will cause overflow and int can't handle it and
// going to long makes more space consumption
// optimal with no overflowing
// XOR operation
```

1. XOR Properties:
    - Identity: $x \oplus 0 = x$
    - Self-Cancellation: $x \oplus x = 0$
    - Commutativity and Associativity: $x \oplus y = y \oplus x$ and $(x \oplus y) \oplus z = x \oplus (y \oplus z)$

```
int findMissingNumber(vector<int>& nums) {
    int n = nums.size();
    int xorFull = 0;
    int xorArray = 0;

    for (int i = 0; i < n; ++i) {
    // COULD HAVE DONE IN TWO DIFFERENT LOOPS BUT SAVE TIME
```

```
        xorArray ^= nums[i];
        xorFull ^= i+1;
    }
    xorFull ^= n;

    // Missing number is the XOR of xorFull and xorArray
    return xorFull ^ xorArray;
}
```

Given an array `[1, 2, 4, 5]`, the missing number is `3`.

1. Compute XOR of numbers from `1` to `5`:
   - $1 \oplus 2 \oplus 3 \oplus 4 \oplus 5$
2. Compute XOR of array elements:
   - $1 \oplus 2 \oplus 4 \oplus 5$
3. XOR the results of the above two computations to find the missing number.

- use the commutative property and we will be left with 0^3 = 3

Given an array where elements appear twice, except one element return that

- return the XOR of all elements

Two sum : {Keep hashing elements and find if we have already stored target - currNum}

DNF Algo : used for sorting an array with three distinct values. It is particularly useful for problems where you need to sort an array into three groups or categories.

Time → O(n)

```
void sortColors(vector<int>& nums) {
      int n = nums.size();
      int low = 0;
      int mid = 0;
      int high = n-1;
      // EVERYTHING BETWEEN MID AND HIGH IS UNSORTED

      while(mid <= high) {
          if(nums[mid] == 0) {
              swap(nums[low], nums[mid]);
              low++; // SHRINK FROM LEFT
              mid++;
          }
          else if(nums[mid] == 1) {
              mid++;
          }
          else {
              swap(nums[mid], nums[high]);
```

```
            high--;
            // SHRINK FROM RIGHT.
            // NO CHANGE IN MID AS WE SWAPPED IT WITH ELEMENT IN
            // HIGH IT COULD BE ANYTHING
        }
    }
}
```

Majority Element :

Moore's Voting Algorithm is an efficient algorithm used to find the majority element in an array. A majority element is defined as an element that appears more than n / 2 times in an array of size n.

```
int majorityElement(vector<int>& nums) {
    int n = nums.size();
    int element = 0;  // Candidate for majority element
    int cnt = 0;      // Counter for the candidate

    // First pass to find the candidate
    for (auto num : nums) {
        if (cnt == 0) {
            element = num;
            cnt = 1;
        } else if (num == element) {
            cnt++;
        } else {
            cnt--;
        }
    }

    // Second pass to validate the candidate
    // {why do we need to validate this ?}
    // => Moore's Voting Algorithm does indeed have a "greedy" flavor to it.
    // In algorithms, a "greedy" approach typically makes a sequence of choices,
    // each of which seems the best at the moment,
    // without worrying about the consequences of those choices in the future.
    int check = 0;
    for (auto num : nums) {
        if (num == element) {
            check++;
        }
    }

    // Return the majority element if it is indeed the majority
    return check > n / 2 ? element : -1;
}
```

maxSubArray : Kadane's Algorithm{greedy approach}

```
int maxi = nums[0];
for(int i = 1, curr = nums[0];i < nums.size();i++)
{
    curr = max(nums[i], nums[i] + curr);
    maxi = max(maxi, curr);
}
return maxi;
```

We basically just check if it is better to let the new guy go alone, or add the curr to it and continue.

- say we are asked to return the index, then whenever the curr_sum < 0 make curr_sum = 0 {so for sure it would be better to let new guy go alone}
- when ever now you update the maxi {keep track of when curr_sum == 0 and maxi was updated!}

Given an array having both +ve and -ve elements, rearrange them such a way we have alternate + and -

```
vector<int> rearrangeArray(vector<int>& nums) {
    int n = nums.size();
    vector<int> ans(n);
    int pos = 0;
    int neg = 1; // it is given n >= 2

    for(int i = 0; i < n; i++) {
        if(nums[i] > 0) {
            ans[pos] = nums[i];
            pos += 2;
        } else {
            ans[neg] = nums[i];
            neg += 2;
        }
    }

    return ans;
}
```

Given an array or string we can get the next permutation using a std next_permutation() func.

But How does it work ?

12345 next would be 12354 then, 12435

- Basically we find a dip coming from right and then swap the dipped element with least bigger number than the dipped element and then reverse the numbers after peak and we are done!

```
void nextPermutation(vector<int>& nums) {
  int ind = -1;
  int n = nums.size();
```

```
  // Step 1: Find the first decreasing element from the end.
  for(int i = n - 2; i >= 0; i--) {
      if(nums[i] < nums[i + 1]) {
          ind = i;
          break;
      }
  }

  // Step 2: If the sequence is non-increasing, reverse it.
  if(ind == -1) {
      reverse(nums.begin(), nums.end());
      // IT IS DESCENDING ORDER WE HAVR TO MAKE IT ASCENDING
      // {AS NEXT BIGGER DOESN'T EXIST}
  }
  else {
      // Step 3: Find the smallest element larger than nums[ind] and swap.
      for(int i = n - 1; i > ind; i--) {
          if(nums[i] > nums[ind]) {
              swap(nums[i], nums[ind]);
              // EVEN AFTER SWAPPING THE ELEMENTS AFTER ind WILL BE IN
              // DECREASING ORDER SO JUST REVERSE THEM
              // {make it as small as possible to get next permutation}
              break;
          }
      }
      // Step 4: Reverse the sequence after the ind.
      reverse(nums.begin() + ind + 1, nums.end());
  }
}
```

Given an array return the leaders, which is greater than all elements to the right.

```
vector<int> superiorElements(vector<int> &a) {
    int n = a.size();
    vector<int> ans;

    int maxi = INT_MIN;
    for(int i = n - 1; i >= 0; i--) {
        if(a[i] > maxi) {
            ans.push_back(a[i]);
            maxi = a[i];
        }
    }
```

```
    return ans;
}
```

unordered_set → is implemented using a hash table : lookup ⇒ O(1)

ordered_set → is implemented using a red-black tree : lookup ⇒ O(logn)

Given an array return the longest consecutive sequence :

```
int longestConsecutive(vector<int>& nums) {
  unordered_set<int> st;

  // Insert all elements into the unordered_set {WHY ?} -> above given reasons
  for(auto num : nums) st.insert(num);

  int ans = 0;

  // Iterate over the elements in the set
  for(auto num : st) {
      // Check if this number is the start of a sequence
      if(st.find(num - 1) == st.end()) {
          int currentNum = num;
          int count = 0;

          // Count the length of the sequence
          while(st.find(currentNum) != st.end()) {
              count++;
              currentNum++;
          }

          // Update the maximum sequence length
          ans = max(ans, count);
      }
  }

  return ans;
}
```

When dealing with 2D array, a space complexity of n^2 is minimum expected. Try optimizing other things like space in these type of cases

Set Matrix Zeroes : Given a 2D matrix, if an element is 0 set its row and column to 0.

```
// BRUTE FORCE O(N^3)
Traverse the matrix, when found a 0 set all the elements in the
row and column -1 {0 will create complecations}
then in next traversal make those -1 => 0
// BETTER SOLUTION O(N^2) + SPACE
```

```
Create one extra column and row and use them as markers
// OPTIMAL
Use first row and column as markers
```

```
void setZeroes(vector<vector<int>>& matrix) {
  int rows = matrix.size();
  int cols = matrix[0].size();
  bool firstRowZero = false;
  bool firstColZero = false;

  // Step 1: Determine if the first row
  // or first column should be zero
  for(int i = 0; i < rows; i++) {
      if(matrix[i][0] == 0) {
          firstColZero = true;
          break;
      }
  }

  for(int j = 0; j < cols; j++) {
      if(matrix[0][j] == 0) {
          firstRowZero = true;
          break;
      }
  }

  // Step 2: Use first row and first column as markers
  // {i = 0 and j = 0 will be dealt later}
  for(int i = 1; i < rows; i++) {
      for(int j = 1; j < cols; j++) {
          if(matrix[i][j] == 0) {
              matrix[i][0] = 0;
              matrix[0][j] = 0;
          }
      }
  }

  // Step 3: Zero out cells based on the markers
  for(int i = 1; i < rows; i++) {
      for(int j = 1; j < cols; j++) {
          if(matrix[i][0] == 0 || matrix[0][j] == 0) {
              matrix[i][j] = 0;
          }
      }
  }

  // Step 4: Handle the first row and first column
```

```
  if(firstColZero) {
      for(int i = 0; i < rows; i++) {
          matrix[i][0] = 0;
      }
  }

  if(firstRowZero) {
      for(int j = 0; j < cols; j++) {
          matrix[0][j] = 0;
      }
  }
}
```

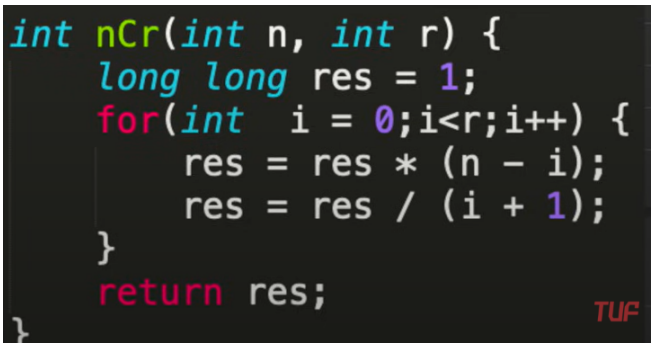
Given a 2D matrix rotate it by 90 degree

```
// BRUTE FOREC -> Create a new copy, do changes
// OPTIMAl
void rotate(vector<vector<int>>& matrix) {
  int n = matrix.size();

  // Step 1: Transpose the matrix
  for(int i = 0; i < n; i++) {
      for(int j = i + 1; j < n; j++) {
          swap(matrix[i][j], matrix[j][i]);
      }
  }

  // Step 2: Reverse each row
  for(int i = 0; i < n; i++) {
      reverse(matrix[i].begin(), matrix[i].end());
  }
} // YOU COULD HAVE DONE STEP 1 AND 2 IN SAME ITERATION ONCE THE INNER j
  // LOOP IS COMPLETED TO SAVE TIME
```

CHAKRI :)

```
vector<int> spiralOrder(vector<vector<int>>& matrix) {
  int rows = matrix.size();
  int cols = matrix[0].size();

  vector<int> ans;

  int top = 0;
  int bottom = rows - 1;
  int left = 0;
  int right = cols - 1;
```

```
  while (top <= bottom && left <= right) {
      // Traverse from left to right across the top row
      for (int j = left; j <= right; j++) {
          ans.push_back(matrix[top][j]);
      }
      top++;

      // Traverse from top to bottom down the right column
      for (int i = top; i <= bottom; i++) {
          ans.push_back(matrix[i][right]);
      }
      right--;

      // Traverse from right to left across the bottom row
      if (top <= bottom) {  // Ensure there's a row to traverse
          for (int j = right; j >= left; j--) {
              ans.push_back(matrix[bottom][j]);
          }
          bottom--;
      }

      // Traverse from bottom to top up the left column
      if (left <= right) {  // Ensure there's a column to traverse
          for (int i = bottom; i >= top; i--) {
              ans.push_back(matrix[i][left]);
          }
          left++;
      }
  }

  return ans;
}
```



in a pascals triangle, the element at ith row and jth column is {i-1}C{j-1}

- To get a row in pascal triangle, we need to always apply ncr it is redundant

```
vector<int> generateRow(int row) {
    long long ans = 1;
    vector<int> ansRow;
    ansRow.push_back(1);
    for(int col = 1;col<row;col++) {
        ans = ans * (row - col);
        ans = ans / (col);
        ansRow.push(ans);
    }
    return ansRow;
}
```

- Then use this to get whole triangle

```
vector<vector<int>> pascalTriangle(int N) {
    vector<vector<int>> ans;
    for(int i = 1;i<=N;i++) {
        ans.push_back(generateRow(i));
    }
    return ans;
}
```

- rows are started from 1 but cols are 0

Given an array, return the elements which appear more than (floor of n / 3) times

- First observation we can do by simple trail and error is that there would be at-max two elements present
- Can we modify Moore's algo and modify it a bit ?

```
// As we are aiming to get two elements
int majorityElements(vector<int>& nums) {
    int n = nums.size();
    int element1 = -1, element2 = -1;
    // Candidate for majority element
    int cnt1 = 0, cnt 2 = 0;
    // Counter for the candidate

    // First pass to find the candidates
    for (auto num : nums) {
        if (cnt1 == 0 && num != element2) {
        // NO INTERFERENCE AS WE ARE SURE THERE WILL AT-MAX TWO
            element1 = num;
            cnt1 = 1;
        }
        else if (cnt2 == 0 num != element1) {
                element2 = num;
            cnt2 = 1;
```

```
        }
        else if (num == element1) {
            cnt1++;
        }
        else if (num == element2) {
            cnt2++;
        }
        else {
            cnt1--;
            cnt2--;
        }
    }

    // Second pass to validate the candidates
    int check1 = 0, check2 = 0;
    for (auto num : nums) {
        if (num == element1) {
            check1++;
        }
        else if (num == element2) {
            check2++;
        }
    }

    // Return the majority elements if it is indeed the majority
      // =? return only those whose count is >= n / 3
}
```

Given an array return the triplets whose sum equals zero. Insure there are no duplicates.

```
// BRUTE FORCE O(n^3)
vector<vector<int>> threeSum(vector<int>& nums) {
  int n = nums.size();
  set<vector<int>> st;
  for(int i = 0;i < n;i++) {
      for(int j = i + 1;j < n;j++) {
          for(int k = j + 1;k < n;k++) {
          // GET ALL POSSIBLE TRIPLETS
              if(nums[i] + nums[j] + nums[k] == 0) {
                  vector<int> temp = {nums[i], nums[j], nums[k]};
                  sort(temp.begin(), temp.end());
                  st.insert(temp);
                  // WE ARE PUSHING THEIR SORTED TRIPLET
                  // IN A SET TO CHECK FOR DUPLICATES
              }
          }
      }
```

```
  }

  vector<vector<int>> ans(st.begin(), st.end());
  // LEARN THIS METHOD OF CREATNG AN ARRAY
  return ans;
}
```

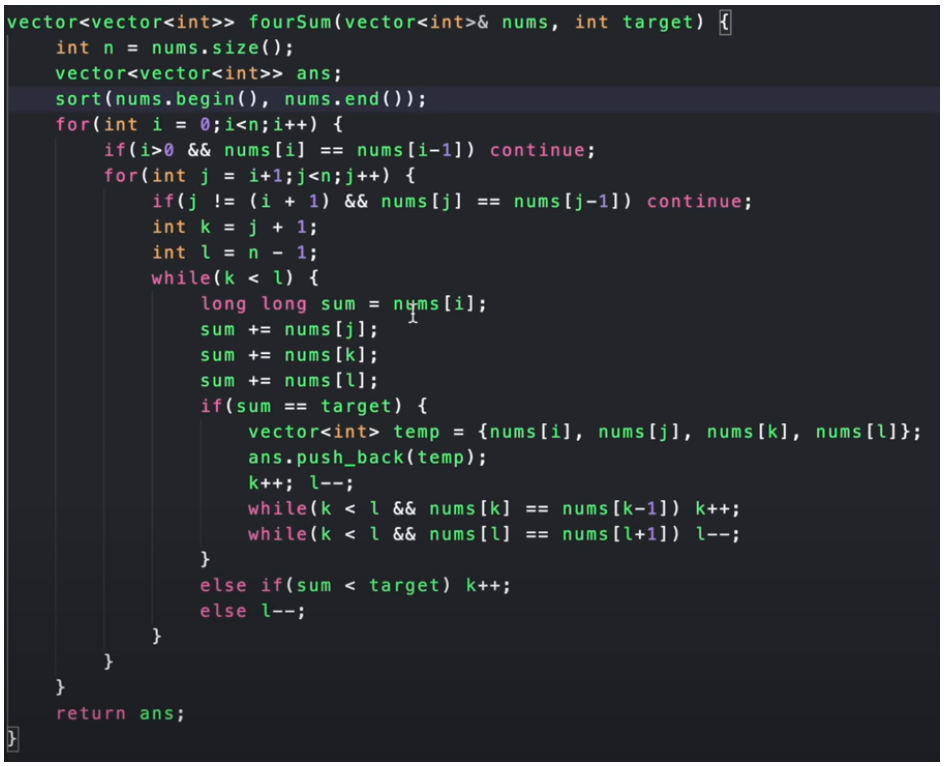
- Try remembering the two sum problem, apply it here ?

```
// BETTER SOLUTION O(n^2)
vector<vector<int>> threeSum(vector<int>& nums) {
  int n = nums.size();
  set<vector<int>> st;

  for(int i = 0; i < n; i++) {
      unordered_map<int, int> mp;
      for(int j = i + 1; j < n; j++) {
          int target = -(nums[i] + nums[j]);
          if(mp.find(target) != mp.end()) {
              vector<int> temp = {nums[i], nums[j], target};
              sort(temp.begin(), temp.end());
              st.insert(temp);
          }
          mp[nums[j]]++; // first checking and then adding
      }
  }

  vector<vector<int>> ans(st.begin(), st.end());
  return ans;
}
```

```
// OPTIMAL SOLUTION -> get rid of set
// IT IS STILL O(n^2)
vector<vector<int>> threeSum(vector<int>& nums) {
  vector<vector<int>> ans;
  int n = nums.size();
  sort(nums.begin(), nums.end());
  // Sorting the array

  for(int i = 0; i < n - 2; i++) {
      if(i > 0 && nums[i] == nums[i-1]) continue;
      // Skip duplicates for the first element

      int j = i + 1;
      int k = n - 1;

      while(j < k) {
```

```
            int sum = nums[i] + nums[j] + nums[k];

            if(sum > 0) { // AS IT IS SORTED
                k--;
            } else if(sum < 0) {
                j++;
            } else {
                ans.push_back({nums[i], nums[j], nums[k]});
                j++;
                k--;
                // Skip duplicates for the second element
                while(j < k && nums[j] == nums[j-1]) j++;
                // Skip duplicates for the third element
                while(j < k && nums[k] == nums[k+1]) k--;
            }
        }
    }
    return ans;
}
```

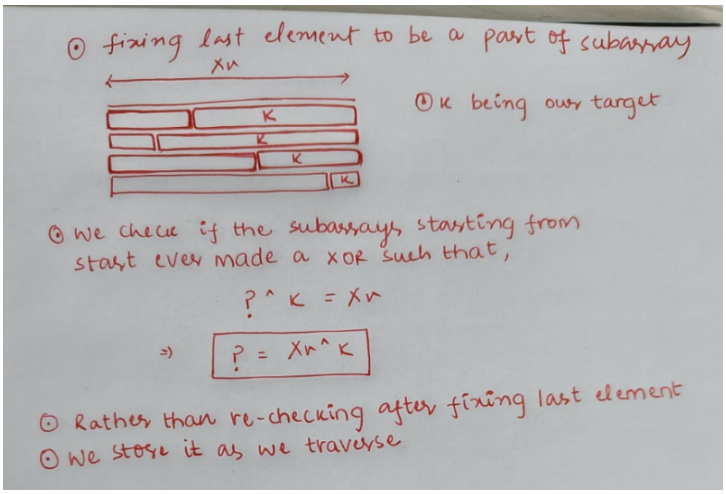
4 SUM !



```
vector<vector<int>> fourSum(vector<int>& nums, int target) {
    int n = nums.size();
    vector<vector<int>> ans;
    sort(nums.begin(), nums.end());
    for(int i = 0;i<n;i++) {
        if(i>0 && nums[i] == nums[i-1]) continue;
        for(int j = i+1;j<n;j++) {
            if(j != (i + 1) && nums[j] == nums[j-1]) continue;
            int k = j + 1;
            int l = n - 1;
            while(k < l) {
                long long sum = nums[i];
                sum += nums[j];
                sum += nums[k];
                sum += nums[l];
                if(sum == target) {
                    vector<int> temp = {nums[i], nums[j], nums[k], nums[l]};
                    ans.push_back(temp);
                    k++; l--;
                    while(k < l && nums[k] == nums[k-1]) k++;
                    while(k < l && nums[l] == nums[l+1]) l--;
                }
                else if(sum < target) k++;
                else l--;
            }
        }
    }
    return ans;
}
```

- for j we need to make sure it is not the current i's second next element before we skip duplicate
- Also, while adding make sure not to do it in one go as it might cause overflow!

Given an array return number of subarrays whose xor makes a target = k




```
int subarrays_With_XOR_K(vector<int> a, int k) {
    int xr = 0;
    unordered_map<int, int> mpp;
    mpp[xr]++; // {0, 1) -> {As the beginning element might be the target
                          // itself then this count will add up}
    int cnt = 0;

    for(int i = 0; i < a.size(); i++) {
        xr = xr ^ a[i]; // Calculate the current prefix XOR
        int x = xr ^ k;  // The required XOR for a valid subarray {?}
        cnt += mpp[x];   // Add the count of such subarrays {COULD BE MULTIPLE}
        mpp[xr]++;       // Update the map with the current XOR
    }

    return cnt;
}
```

Merge overlapping intervals

```
vector<vector<int>> mergeOverlappingIntervals(vector<vector<int>> &arr) {
    int n = arr.size();
    sort(arr.begin(), arr.end());
    // Sorting based on the starting time of intervals
    vector<vector<int>> ans;

    for(int i = 0; i < n; i++) {
        if(ans.empty() || arr[i][0] > ans.back()[1]) {
            ans.push_back(arr[i]);
            // Add new interval if it does not overlap
        } else {
```

```
            ans.back()[1] = max(ans.back()[1], arr[i][1]);
            // Merge overlapping intervals
        }
    }

    return ans;
}
```

Given two sorted arrays, return the net sorted without using extra space as arr1 and arr2

- SHELL SORT

```
class Solution {
private:
    void swapIfGreater(long long arr1[], long long arr2[], int ind1, int ind2) {
        if (arr1[ind1] > arr2[ind2]) {
            swap(arr1[ind1], arr2[ind2]);
        }
    }

public:
    // Function to merge the arrays.
    void merge(long long arr1[], long long arr2[], int n, int m) {
        int len = (n + m);
        int gap = (len / 2) + (len % 2); // ceil

        while (gap > 0) {
            int left = 0;
            int right = left + gap;

            while (right < len) {
                // If both pointers are in the first array.
                if (left < n && right < n) {
                    swapIfGreater(arr1, arr1, left, right);
                }
                // If left pointer is in the first array and right pointer
                // is in the second array.
                else if (left < n && right >= n) {
                    swapIfGreater(arr1, arr2, left, right - n);
                }
                // If both pointers are in the second array.
                else {
                    swapIfGreater(arr2, arr2, left - n, right - n);
                }
                left++;
                right++;
            }
```

```
            if (gap == 1) break;
            // it will run if gap is 1 for the first then it will break
            gap = (gap / 2) + (gap % 2);
        }
    }
};
```

- We could use merge sort but needs extra space, heap sort also need to maintain a DS.
- insertion sort requires entire array not chunks of it.
- One other thing we could do was

```
void merge(int arr1[], int arr2[], int m, int n) {
    int i = m - 1;
    int j = 0;

    // Traverse the arrays {last elements of first array and
    // first elements of second array being swapped and then individual sorting happe
    while (i >= 0 && j < n) {
        // If the current element of arr1 is greater than arr2's element
        if (arr1[i] > arr2[j]) {
            // Swap the elements
            swap(arr1[i], arr2[j]);

            // Sort arr2 since it might have been disturbed
            sort(arr2, arr2 + n);
        }

        // Move to the next element
        i--;
        j++;
    }
}
```

Given an array of size n. there are elements from 1-n but one element is repeated return missing and duplicate as a pair

- create an array with elements 1-n.
- find difference of sum of both arrays ⇒ X - Y = c1
- find difference of sum of squares of both arrays ⇒ X^2 - Y^2 = c2
- solve both and get X, Y

```
// XOR approach
pair<int, int> findMissingAndRepeating(const vector<int>& arr) {
    int n = arr.size();
    int xor1 = 0;
```

```
    // Step 1: XOR all elements of the array and numbers from 1 to n
    for (int i = 0; i < n; i++) {
        xor1 ^= arr[i];
    }
    for (int i = 1; i <= n; i++) {
        xor1 ^= i;
    }

    // Step 2: xor1 now contains x ^ y
    // (missing ^ repeating)

    // Step 3: Find the rightmost set bit.
    // WHY ? {This is the bit that differertiates the missing and duplicate number}
    int setBit = xor1 & ~(xor1 - 1);

    int x = 0, y = 0;

    // Step 4: Divide elements into two groups based on the set bit
    // {As there mupltiple possible combinations who would differ in the same bit}
    for (int i = 0; i < n; i++) {
        if (arr[i] & setBit)
            x ^= arr[i];
        else
            y ^= arr[i];
    }
    for (int i = 1; i <= n; i++) {
        if (i & setBit)
            x ^= i;
        else
            y ^= i;
    }

    // Step 5: Determine which is missing and which is repeating
    for (int i = 0; i < n; i++) {
        if (arr[i] == x)
            return {y, x}; // y is missing, x is repeating
    }
    return {x, y}; // x is missing, y is repeating
}
```

Given an array return the number of pairs (first, second) such that first comes before second and if greater than second → INVERSE PAIRS

▼ Changes in merge sort to get the result

```
int ans = 0;

void merge(vector<int> &arr, int low, int mid, int high) {
```

```
    vector<int> temp; // temporary array
    int left = low;      // starting index of left half of arr
    int right = mid + 1;   // starting index of right half of arr

    //storing elements in the temporary array in a sorted manner//

    while (left <= mid && right <= high) {
        if (arr[left] <= arr[right]) {
            temp.push_back(arr[left]);
            left++;
        }
        else {
            temp.push_back(arr[right]);
            ans += (mid - left + 1); // left ka starting is greater => all the ele
            right++;
        }
    }

    // if elements on the left half are still left //

    while (left <= mid) {
        temp.push_back(arr[left]);
        left++;
    }

    //  if elements on the right half are still left //
    while (right <= high) {
        temp.push_back(arr[right]);
        right++;
    }

    // transfering all elements from temporary to arr //
    for (int i = low; i <= high; i++) {
        arr[i] = temp[i - low];
    }
}

void mergeSort(vector<int> &arr, int low, int high) {
    if (low >= high) return;
    int mid = (low + high) / 2 ;
    mergeSort(arr, low, mid);  // left half
    mergeSort(arr, mid + 1, high); // right half
    merge(arr, low, mid, high);  // merging sorted halves
}
```

In the previous question what if the condition given is a[i] > 2 * a[j] ? → REVERSE PAIRS

▼ Changes in merge sort to get the result

```
int ans = 0;

void merge(vector<int>& nums, int low, int mid, int high) {
    int j = mid + 1;
    // Count reverse pairs {The technique we used in the previous question wont wo
    for (int i = low; i <= mid; i++) {
        while (j <= high && nums[i] > 2LL * nums[j]) {
            j++;
        }
        ans += (j - (mid + 1));
    }

    // Standard merge operation
    vector<int> temp;
    int left = low, right = mid + 1;
    while (left <= mid && right <= high) {
        if (nums[left] <= nums[right]) {
            temp.push_back(nums[left++]);
        } else {
            temp.push_back(nums[right++]);
        }
    }
    while (left <= mid) temp.push_back(nums[left++]);
    while (right <= high) temp.push_back(nums[right++]);

    // Copy the sorted subarray back to nums
    for (int i = low; i <= high; i++) {
        nums[i] = temp[i - low];
    }
}

void mergeSort(vector<int>& nums, int low, int high) {
    if (low >= high) return;
    int mid = low + (high - low) / 2;
    mergeSort(nums, low, mid);
    mergeSort(nums, mid + 1, high);
    merge(nums, low, mid, high);
}
```

Given an array with integers, return me the maximum possible product with subarrays

- take all elements ?
- if even -ve we are good
- if odd -ve we omit one of it
- if we encounter a zero in mid way make the current product = 1

```
int subarrayWithMaxProduct(vector<int> &arr) {
    int pre = 1, suff = 1; // one from front the other from end
    int ans = INT_MIN;
    int n = arr.size();
    for(int i = 0; i < n; i++) {
        if(pre == 0) pre = 1;
        if(suff == 0) suff = 1;

        pre = pre * arr[i];
        suff = suff * arr[n - i - 1];
        ans = max(ans, max(pre, suff));
    }
    return ans;
}
```