Bubble Sort:

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; ++i) {
        for (int j = 0; j < n - i - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

time complexity of O(n^2)

space complexity of O(1).

Optimized Bubble Sort:

```
void optimizedBubbleSort(int arr[], int n) {
    bool swapped;
    for (int i = 0; i < n - 1; ++i) {
        swapped = false;
        for (int j = 0; j < n - i - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }
        if (!swapped)
            break; // If no two elements were swapped in inner loop, the a
    }
}
```

insertion sort in C++:

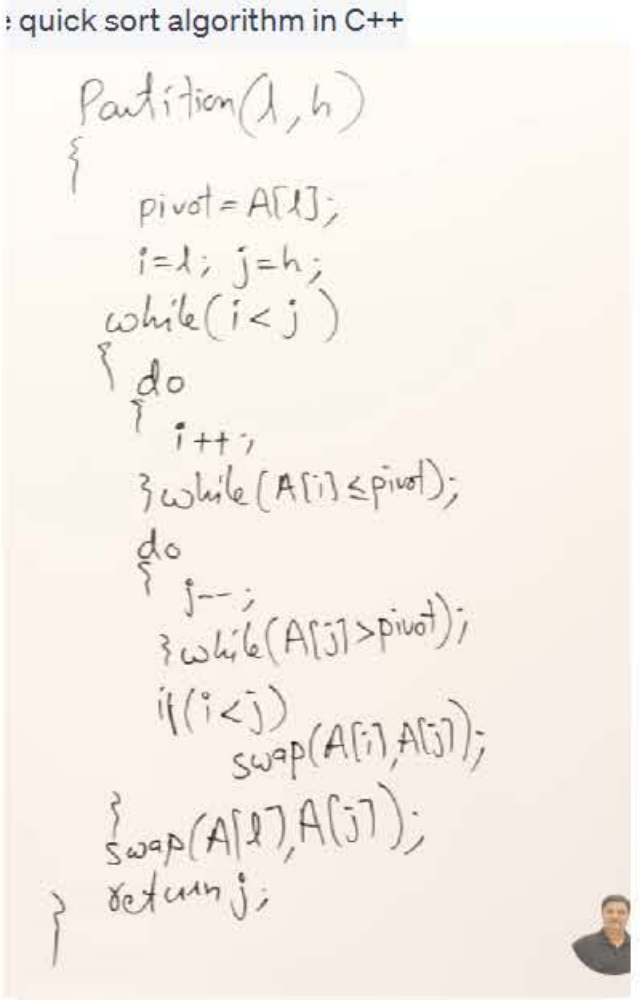
```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }

        arr[j + 1] = key;
    }
}
```

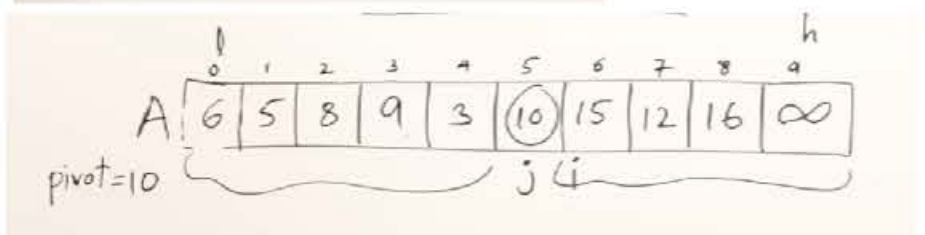
merge sort algorithm in C++:

```
void merge(vector<int>& arr, int left, int middle, int right) {
    int n1 = middle - left + 1;
    int n2 = right - middle;

    vector<int> leftArray(n1);
    vector<int> rightArray(n2);

    for (int i = 0; i < n1; ++i) {
        leftArray[i] = arr[left + i];
    }
    for (int j = 0; j < n2; ++j) {
        rightArray[j] = arr[middle + 1 + j];
    }

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (leftArray[i] <= rightArray[j]) {
            arr[k] = leftArray[i];
            ++i;
        } else {
            arr[k] = rightArray[j];
            ++j;
        }
        ++k;
    }

    while (i < n1) {
        arr[k] = leftArray[i];
        ++i;
        ++k;
    }

    while (j < n2) {
        arr[k] = rightArray[j];
        ++j;
        ++k;
    }
}

void mergeSort(vector<int>& arr, int left, int right) {
    if (left < right) {
        int middle = left + (right - left) / 2;

        mergeSort(arr, left, middle);
        mergeSort(arr, middle + 1, right);

        merge(arr, left, middle, right);
    }
}
```

```
mergeSort(arr, 0, n - 1);
```

quick sort algorithm in C++





Union Of Two Sorted Arrays

Bruteforce → Use a set and push all elements of both the arrays

Optimal → Two pointer approach

```
vector < int > sortedArray(vector < int > a,
    vector < int > b) {
    int n1 = a.size();
    int n2 = b.size();
    int i = 0;
    int j = 0;
    vector<int> unionArr;
    while(i<n1 && j < n2) {
        if(a[i] <= b[j]) {
            if(unionArr.size() == 0 ||
            unionArr.back() != a[i]) {
                unionArr.push_back(a[i]);
            }
            i++;
        }
        else {
            if(unionArr.size() == 0 ||
            unionArr.back() != b[j]) {
                unionArr.push_back(b[j]);
            }
            j++;
        }
    }

    while(j<n2) {
        if(unionArr.size() == 0 ||
            unionArr.back() != b[j]) {
                unionArr.push_back(b[j]);
        }
        j++;
    }

    while(i<n1) {
        if(unionArr.size() == 0 ||
            unionArr.back() != a[i]) {
                unionArr.push_back(a[i]);
        }
        i++;
    }

    return unionArr;
}
```

Intersection Of Two Sorted Arrays

A[] = [1, 2, 2, 3, 3, 4, 5, 6]

B[] = [2, 3, 3, 5, 6, 6, 7]

ans[] → [2, 3, 3, 5, 6]

We can see that duplicacy is allowed here ie we just need to find the common elements between the two arrays without caring about the no of times it occurs

Approach: For every element there should be a correspoding element in the other array

Bruteforce → For every element in array A chech if the element exists in B. Also make a visited array for B to keep track of the elements already selected in B.

```
n1  n2  vis[n2] = {0}

for(i = 0 → n1)
{
    for(j = 0 → n2)
    {
        if(a[i] == b[j] && vis[j] == 0)
        {
            ans.add(a[i]
            vis[j] = 1;
            break;
        }
        if( b[j] > a[i]) break;
    }
}
```

TC → O(n1 x n2)
SC → O(n2)

As the arrays are sorted, if b[j] > a[i] this means the element a[i] will never be found in b

Optimal Siolution → Two pointer approach

```
vector<int> findArrayIntersection(vector<int> &A, int
vector<int> &B, int m)
{
    int i = 0;
    int j = 0;
    vector<int> ans;
    while(i<n && j<m) {
        if(A[i] < B[j]) {
            i++;
        }
        else if(B[j] < A[i]) {
            j++;
        }
        else {
            ans.push_back(A[i]);
            i++;
            j++;
        }
    }
    return ans;
}
```

Missing Number

Given an integer N and an array of size N-1 containing N-1 numbers between 1 to N.
Find the number(between 1 to N), that is not present in the given array.

Bruteforce: For every number n from 1->N, check if n is present in array using linear search

Better: Create a visited array of size N+1 (idx = 0->N). Iterate through the array and mark that element visited.
Iterate through the visited array, if an element is not visited => Missing No

Optimal: We have 2 approaches
1. Sum -> WKT sum of first N natural numbers is N*(N-1)/2.
We iterate through the array and find sum of all elements.
Missing no = N*(N-1)/2 - Sum(array)

TC → O(N)
SC → O(1)

We have a small problem. Say if N=10^5
Sum = N*N(-1)/2 = 10^5 * 10^5 /2 == 10^10
This cannot be store in normal integer, we will reauire a long int
Which is not a big problem but will take a little extra space

2. XOR -> We will use tha property of XOR that XOR of two same no is 0 and any no XOR with 0 is also 0

XOR1 -> XOR of all numbers from 1->N is 1 ^ 2 ^ ..... ^ N
XOR2-> XOR of all numbers of array

XOR1 = 1 ^ 2 ^ 3 ^ 4 ^ 5
XOR2 = 1 ^ 2 ^ 4 ^ 5

So when we perform XOR1 ^ XOR2, the common elements will cancel each other and only the missing element will be left

XOR1 ^ XOR2 = 3

(1^1) ^ (2^2) ^ (3) ^ (4^4)
0 0 0

0 ^ 3 = 3

TC → O(N)
SC → O(1)

The time and space complexity of this approach is same as that of the sum approach, but we can find the ans using int itself.

Find the number that appears once, and the other numbers twice

arr[] = {1, 1, 2, 3, 3, 4, 4}

This is completely uses the same approach as the above solution.

Bruteforce: Linear search to check if every element appears twice

Better: Keep a freq array which will store frequencey. Iterate through this array and if freq=1, then that is the answer

Problem: If an element is -ve or if it is very large no like 10^12 we cannot create array of that size

Solution: use map<long long int, int> ie map<element, freq>

Optimal: XOR method
Create a variable XOR and iterate torough tha array and perform XOR ^ element for each array element.
The number repeated twice will get cancelled with one another and after the iteration, XOR will contain tha answer

Longest Subarray with given Sum K (All Positive elements)

Bruteforce: Generate all subarrays and find their sum. If sum == k, update max length

T=O(N^2)

Better: Create a hashmap to store prefix sum. map<int, int> ie map<sum till idx, idx>


```
int longestSubarrayWithSumK(vector<int> a,
long long k) {
    map<long long, int> preSumMap;
    long long sum = 0;
    int maxLen = 0;
    for(int i = 0;i<a.size();i++) {
        sum += a[i];
        if(sum == k) {
            maxLen = max(maxLen, i+1);
        }
        long long rem = sum - k;
        if(preSumMap.find(rem) != preSumMap.end())
            int len = i - preSumMap[rem];
            maxLen = max(maxLen, len);
        }
        if(preSumMap.find(sum) == preSumMap.end())
            preSumMap[sum] = i;
        }
    }
    return maxLen;
}
```

We maintain a sum variable which will store summ till curr idx.
If sum = K, then 0->i subarray is a subarray with sum = k
else we check if sum-k is present in the map
If present it means that elements after that summed up to k

Note here, we push the currSum if it previously doesnot exist because we want lngest subarray and we want the starting point to be as left as possible

Optimal: Two pointer approach.
Move i forward until sum <= K
Move j forward until sum > K. Update length if sum=k

```
int longestSubarrayWithSumK(vector<int> a, long long k) {
    int n = a.size();
    int i = -1, j = -1;
    long long sum = 0;
    int maxLen = 0;
    while(1){
        while(i < n && sum <= k){
            if(sum == k) maxLen = max(maxLen, i-j);
            sum += a[++i];
        }

        if(i == n) break;
        while(j < i && sum > k){
            sum -= a[++j];
        }
    }
    return maxLen;
}
```

```
longestSubarrayWithSumK(vector<int>{8,15,17,0,11}, 17)
```

Longest Subarray with given Sum K (Positive & Negative elements)

If array contains both positive and negative, brutefore approach is the same as above
Better solution in the above case is the optimal solution for this answer. It cannot be further optimied

Two Sum

Given an array (not sorted) and a target variable. Find 2 elements that sum up to target

Bruteforce: For every element at index i, check if target-arr[i] is present in rest of the array
$T=O(N^2)$

Better: We need to reduce the time complexity of searching target-arr[i].
We can store every element in a set first. And then iterate through the array and for every element num, check if target-num is present in hashmap. If yes that is the answer
$T = O(N \log N)$
$S = O(N)$

Optimal: Sort the array and do 2 pointer approch

```
string read(int n, vector<int> book, int target)
{
    int left = 0, right = n-1;
    sort(book.begin(), book.end());
    while(left < right) {
        int sum = book[left] + book[right];
        if(sum == target) {
            return "YES";
        }
        else if(sum < target) left++;
        else right--;
    }
    return "NO";
}
```

$T = O(N \log N)$
$S = O(1)$

Sort an array of 0's 1's and 2's

Bruteforce: Apply merge sort ot quick sort

$T = O(N \log N)$

$S = O(N)$

Better: Count the no of 0's 1's and 2's

Fill the array with the found count with 0's 1's and 2's

Opimal: Dutch National Flag Algorithm

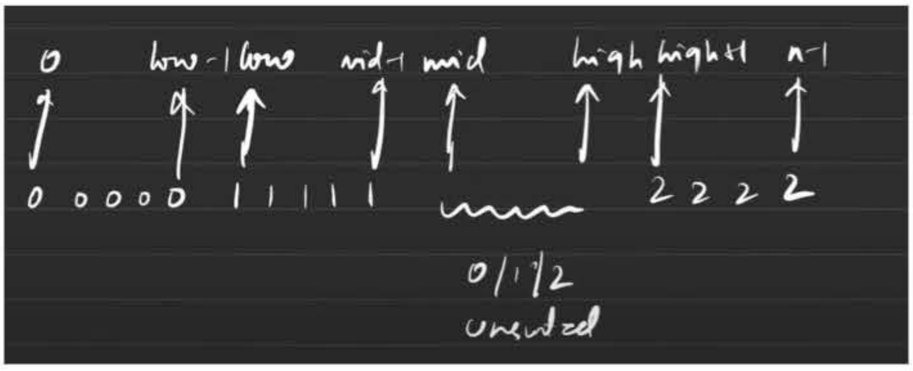
1. We maintain 3 pointers : low, mid and high
2. Assumptions: all elements in range

   [0 -> low-1] -> are 0's

   [low -> mid-1] -> are 1's

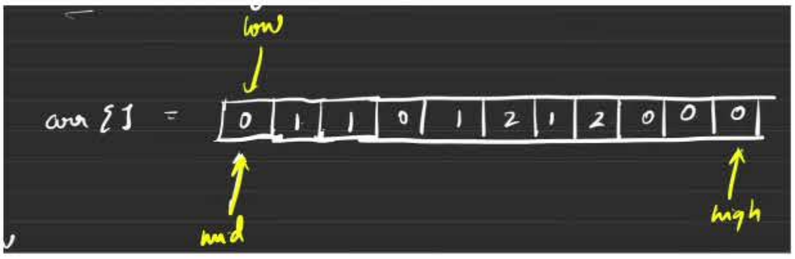
   [high+1 -> n-1] -> are 2's

   => elements in range [mid -> high] are random elements



So elements in range [0 -> mid-1] and [high+1 -> n-1] are sorted. So if we manage to sort [mid+1 -> high] we get the final array

3. Initialize mid=0 and high = n-1 coz initially entire array is unsorted. Also initialize low = 0.



4. Traverse and solve, if

   arr[mid] = 0 => swap(arr[low], arr[mid])
   low++ mid++

   arr[mid] = 1 => mid++

   arr[mid] = 2 => swap(arr[mid], arr[high])
   high--

```cpp
void sortArray(vector<int>& arr, int n)
{
    int low = 0, mid = 0, high = n-1;
    while(mid <= high) {
        if(arr[mid] == 0) {
            swap(arr[low], arr[mid]);
            low++;
            mid++;
        }
        else if(arr[mid] == 1) {
            mid++;
        }
        else {
            swap(arr[mid], arr[high]);
            high--;
        }
    }
}
```

Majority Voting In An Array

Given an array of N elements, find the element whose frequency is greater than N/2

Bruteforce: For every element at idx i, find the count of times it occurs in the rest of array. If count > N/2 print it
T = O(N^2)

Better: Maintain a frequency map ie map<element, freq>. Find the frequency of elements. Now iterate through the map and if freq > N/2 print it
T = O(N)
S = O(N)

Optimal: 1. Moores Voting Algorithm

1. Initialize 2 variables:
   Count - for tracking the count of element
   Element - for which element we are counting
2. Traverse through the given array from index 1->N-1
   If Count is 0 then store the current element of the array as Element.
   If the current element and Element are the same increase the Count by 1.
   If they are different decrease the Count by 1.
3. The integer present in Element should be the result we are expecting

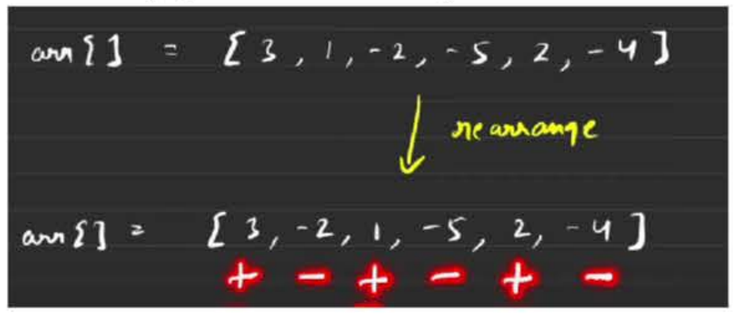
2. Validation

Now iterate through the array again and find its frequency, If it is greater than N/2, then return the 'Element'.

```cpp
int majorityElement(vector<int> v) {
    int cnt = 0;
    int el;
    for(int i = 0;i<v.size();i++) {
        if(cnt == 0) {
            cnt=1;
            el=v[i];
        }
        else if(v[i] == el) {
            cnt++;
        }
        else {
            cnt--;
        }
    }
    int cnt1 = 0;
    for(int i =0;i<v.size();i++) {
        if(v[i] == el) cnt1++;
    }
    if(cnt1 > (v.size() / 2)) {
        return el;
    }
    return -1;
}
```

Rearrange Array Elements by Sign Variety-1

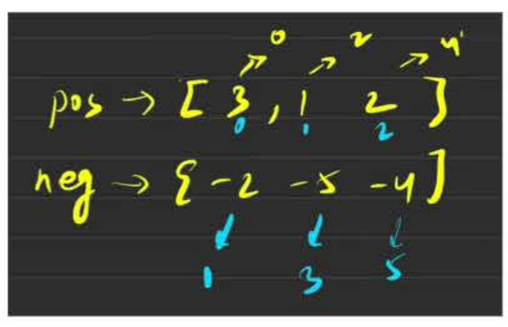
There's an array 'A' of size 'N' with an equal number of positive and negative elements. Without altering the relative order of positive and negative elements, you must return an array of alternately positive and negative values. Note: Start the array with positive elements.

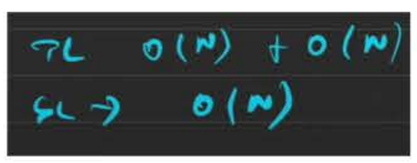


Bruteforce: Create two arrays of size n/2 -> positive and negative. Itereate through the array and fill the positive and negative arrays. Run a loop from 0->n/2 and do the following

arr[2*i] = positive[i]

arr[2*i+1]= negative[i]





Optimal: We cannot reduce the space complexity here, what we can do is we can try to solve it in one iteration. We will create a new array of size n and two pointers in it for pos and neg elements

```cpp
vector<int> rearrangeArray(vector<int>& nums) {
    int n = nums.size();
    vector<int> ans(n ,0);
    int posIndex = 0, negIndex = 1;
    for(int i = 0;i<n;i++) {
        if(nums[i] < 0) {
            ans[negIndex] = nums[i];
            negIndex += 2;
        }
        else {
            ans[posIndex] = nums[i];
            posIndex += 2;
        }
    }
    return ans;
```

Variety-2

There's an array 'A' of size 'N' with positive and negative elements (not necessarily equal). Without altering the relative order of positive and negative elements, you must return an array of alternately positive and negative values. The leftover elements should be placed at the very end in the same order as in array A. Note: Start the array with positive elements.

This question is almost same as the variety1, the only diffference is that th no of positive and negative elements can beanything.

```cpp
vector<int> alternateNumbers(vector<int>&a) {
    vector<int> pos, neg;
    int n = a.size();
    for(int i = 0; i < n;i++) {
        if(a[i] > 0) {
            pos.push_back(a[i]);
        }
        else {
            neg.push_back(a[i]);
        }
    }

    if(pos.size() > neg.size()) {
        for(int i = 0;i<neg.size();i++) {
            a[2*i] = pos[i];
            a[2*i+1] = neg[i];
        }
        int index = neg.size() * 2;
        for(int i = neg.size();i<pos.size();i++) {
            a[index] = pos[i];
            index++;
        }
    }
    else {
        for(int i = 0;i<pos.size();i++) {
            a[2*i] = pos[i];
            a[2*i+1] = neg[i];
        }
        int index = pos.size() * 2;
        for(int i = pos.size();i<neg.size();i++) {
            a[index] = neg[i];
            index++;
        }
    }
    return a;
}
```

This is just a liltle modification of the above bruteforce approach

Next Permutation

Given an array Arr[] of integers, rearrange the numbers of the given array into the lexicographically next greater permutation of numbers. If such an arrangement is not possible, it must rearrange to the lowest possible order (i.e., sorted in ascending order).

This simply states that grnerate all the permutaions of an array in lexicographical order

Bruteforce: 1. Generate all permutaions
2. Return the next permutation on every request

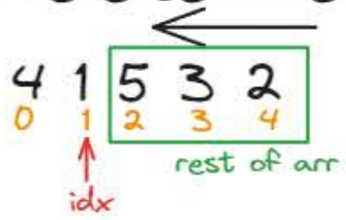
$T = O(N * N!)$

Better: Use STL function

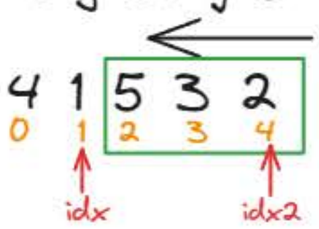
```
vector<int> nextGreaterPermutation(vector<int> &A)
    next_permutation(A.begin(), A.end());
    return A;
}
```

This will rearrange the array to next permutation

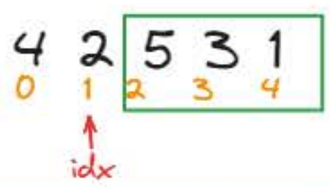
Optimal: 1. Traverse the array in reverse order and find the index at which arr[i] < arr[i+1]

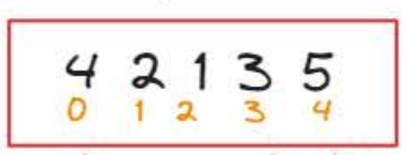


2. Find the element which is just greater than arr[idx] in rest of array



3. Swap arr[idx] and arr[idx2] and reverse the rest of array





next permutation

```
vector<int> nextPermutation(vector<int> &arr){
    if(arr.size() == 1) return arr;

    int n = arr.size();
    int idx = -1;
    for(int i = n-1; i >= 0; i--){ //Finding idx
        if(arr[i] < arr[i+1]){
            idx = i;
            break;
        }
    }

    if(idx == -1){
        reverse(arr.begin(), arr.end());
        return arr;
    }

    for(int i = n-1; i > idx; i++){ //Finding idx2 and swap
        if(arr[i] > arr[idx]){
            swap(arr[i], arr[idx]);
            break;
        }
    }

    reverse(arr.begin()+idx+1, arr.end());//Reverse ROA
    return arr;
}
```