Q1. Swap Alternate.

Approach: Given an array, say [1,2,3,4,5], we will run a for loop from 0 to n-2 and swap arr[i] with arr[i+1]. If we reach the last element, we will stop, so check if $(i+1)^{th}$ element < size of array.

Dry Run: arr[] : {1,2,3,4,5}

① {1,2,3,4,5} ⇒ {2,1,3,4,5}

② {2,1,3,4,5} ⇒ {2,1,4,3,5}

③ {2,1,4,3,5} Stop

Code:

```
void swapAlternate(int arr[], int size) {

    for(int i = 0; i<size; i+=2 ) {
        if(i+1 < size) {
            swap(arr[i], arr[i+1]);
        }
    }

}
```

SWAP FUNCTION:

If a = 5, b = 10, I can't simply do a = b, b = a to swap them. Suppose I did this:

```
int a = 5, b = 10;
a = b; // a becomes 10
b = a; // b becomes 10
```

a = 10 b = 10

Use a third variable to temporarily store the value of a and then change a to b.

```
int a = 5, b = 10;
int temp = a; // temp = 5.
a = b; // a becomes 10
b = temp; // b becomes 5.
```

a = 10 b = 5

```
void swap(int *a, int *b) {
    // pointers to numbers are given
    int temp = *a; // Store the value at address a
    *a = *b; // Change the value at address a to the value at address b
    *b = temp; // Change the value at address b to the original value of a
}
```

Q2. Find Unique Element.

Approach: Since all numbers except any one occur twice, thus if there are m numbers that occur twice and one that occurs once, I will have 2m+1 numbers.

The XOR of same numbers is 0. Thus 5^1^5 = 1.

We can find XOR of all numbers, which will be the only element occuring once.

Dry Run: arr[] : {1, 2, 8, 2, 1, 7, 7}

1^2^8^2^1^7^7 = 0^8 = 8

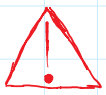
Code:

```
int findUnique(int *arr, int size)
{
    int ans = 0;

    for(int i = 0; i<size; i++) {
        ans = ans^arr[i];
    }
    return ans;
}
```

Q3 Unique Number of Occurances. (Homework)

We can use an unordered map that uses the concept of

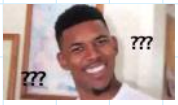


We can use an unordered map that uses the concept of hashing. Learn about it and then use it by yourself.

```cpp
bool uniqueOccurrences(vector<int>& arr) {
    vector<int> occurance(2001, 0);
    // ith index is occurance of (i-1000)

    for(int i=0; i<arr.size(); i++) {
        occurance[arr[i]+1000]++;
    }
    // check if all non-zero elements are unique
    sort(occurance.begin(), occurance.end());
    // if any adjacenet non-zero neighbours are equal, then we have repeating occurances
    for(int i=0;i<2000;i++) {
        if(occurance[i] != 0 && occurance[i] == occurance[i+1])
            return false;
    }
    return true;
}
```

```cpp
bool uniqueOccurrences(vector<int>& arr) {
    unordered_map<int, int> m;
    for(int i=0;i<arr.size();i++) {
        m[arr[i]]++;
    }
    // insert the occurances into a set that will store them in increasing order
    set<int> occ;
    for(auto x: m) {
        occ.insert(x.second);
    }
    // if the set has the same number of elements as the map then there were all unique occurances
    return occ.size() == m.size();
}
```



Q4 Find Duplicates

Approach 1 : XOR all the elements of the array with $1, 2, 3, 4, \ldots, n-1$.

Approach 2 : Subtract the sum of first n-1 natural nos. from the sum of all elements of the array.

Dry Run : arr[] : {4, 2, 1, 3, 1}

n-1 Natural Numbers : 1, 2, 3, 4

XOR : $(4 \wedge 2 \wedge 1 \wedge 3 \wedge 1) \wedge (1 \wedge 2 \wedge 3 \wedge 4) = \boxed{1}$

sum(arr) = 4 + 2 + 1 + 3 + 1 = 11. — (1)

sum of (n-1) natural nos. = 1 + 2 + 3 + 4 = 10 — (2)

$\left(\frac{(n-1)*n}{2}\right)$

eq.(1) - eq.(2) :

$$
\begin{array}{r}
4+2+1+3+1 \\
-\ 1+2+3+4 \\
\hline
1 \\
\hline
\end{array}
$$

Code:

```
int findDuplicate(vector<int> &arr)
{
    int ans = 0;

    for(int i = 0; i<arr.size(); i++ ) {
        ans = ans^arr[i];
    }

    for(int i = 1; i<arr.size();i++ ) {
        ans = ans^i;
    }
    return ans;
}
```

Homework: Find all duplicates in an array.

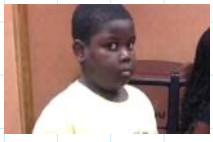
Approaches:

1. For every element, check if there exists a duplicate element in arr[i+1....n-1], if yes then store it in answer array.

2. Sort the array and check for adjacent elements if they are equal.

3. Track frequency using an unordered map (Hash Map) and traverse it to get all the repeating elements.

4. Treat the element at arr[i] as the next index to jump on and mark this new element as negative to show that you have already been to arr[i].
   If you visit an element and found it negative, then you have visited it twice.



Most Efficient Approach's Dry Run:

```
          i
arr[]: {4, 3, 2, 5, 2}
       0  1  2  3  4
```

arr[i] = 4 for i = 0, visit arr[4-1] = arr[3] = 5

and mark it -ve

i

{4, 3, 2, -5, 3}

arr[i] = 3 for i = 1, visit arr[3-1] = arr[2] = 2
and mark it -ve.

{4, 3, -2, -5, 3}

arr[i] = 2 for i = 2, visit arr[2-1] = arr[1] = 3
and mark it -ve.

{4, -3, -2, -5, 3}

arr[i] = 5 for i = 3, visit arr[5-1] = arr[4] = 3
and mark it -ve.

{4, 3, -2, -5, -3}

arr[i] = 3 for i = 4, visit arr[3-1] = arr[2] = -2
Since it's already negative, we have visited arr[2] twice and thus 3 has been used twice to check for arr[3-1]

∴ 3 is occuring twice.

Code :

```
vector<int> findDuplicates(vector<int>& nums) {
    vector<int> ans;
    for(int i=0; i<nums.size(); i++) {
        // visit nums[abs(nums[i]) - 1]
        // abs is used because the current element might be negative
        if(nums[abs(nums[i]) - 1] < 0) {
            ans.push_back(abs(nums[i]));
        }
        else {
            nums[abs(nums[i]) - 1] = -nums[abs(nums[i]) - 1];
        }
    }
    return ans;
}
```

Q5 Find Intersection.

Approach : Maintain two trackers i and j for the first and second array respectively.
At any point of time if :

① arr[i] < arr[j] → increment i
② arr[i] > arr[j] → increment j
③ arr[i] == arr[j] → include in answer and increment both i and j.

Code:

```
vector<int> findArrayIntersection(vector<int> &arr1, int n, vector<int> &arr2, int m)
{
    vector<int> ans;
    int i = 0, j = 0;
    while(i < arr1.size() && j < arr2.size()) {
        if(arr1[i] < arr2[j]) {
            i++;
        }
        else if(arr1[i] > arr2[j]) {
            j++;
        }
        else {
            ans.push_back(arr1[i]);
            i++;
            j++;
        }
    }
    return ans;
}
```

Q6. Pair Sum.

Approach: For the i^{th} element, check if its sum with all $i+1, i+2, \ldots, n-1$ elements == sum. If no then continue, if yes then push the i^{th} & j^{th} elements in an answer vector.

Dry Run: arr[] : {1, 2, 3, 4}
sum = 5.

① arr[] : {1, 2, 3, 4} (i)
Check arr[j] for all j = 1 to 3.
arr[i] = 1 , 1 + arr[j] == 5.
This is satisfied for j = 3, 1 + 4 = 5.
⇒ ans.push_back({1,4})

② arr[] : {1, 2, 3, 4} (i)
Check arr[j] for all j = 2 to 3.
arr[i] = 2 , 2 + arr[j] == 5.
This is satisfied for j = 2, 2 + 3 = 5.
→

⇒ ans.push_back({2,3})

③ arr[] : {1, 2, 3, 4}

Check arr[j] for all j = 3 to 3.

arr[i] = 3 , 3 + arr[j] == 5.

This is not satisfied for any pair.

Code :

```
vector<vector<int> > pairSum(vector<int> &arr, int s){
   vector< vector<int> > ans;

   for(int i=0;i<arr.size(); i++ )
   {
              for(int j = i+1; j<arr.size(); j++) {
              if(arr[i] +arr[j] == s)
          {
              vector<int> temp;
              temp.push_back(min(arr[i], arr[j]));
              temp.push_back(max(arr[i], arr[j]));
                            ans.push_back(temp);
          }
       }
   }
   sort(ans.begin(), ans.end());
   return ans;
}
```

Homework : Find triplet sum.

Same Approach as Find Pair Sum.

Code :

```
#include <set>
vector<vector<int>> findTriplets(vector<int>arr, int n, int K) {
    vector<vector<int>> ans;
    set<vector<int>> s;
    for(int i=0;i<n;i++) {
        for(int j = i + 1; j < n; j++) {
            for(int k = j + 1; k < n; k++) {
                if(arr[i]+arr[j]+arr[k] == K) {
                    vector<int> temp;
                    temp.push_back(arr[i]);
                    temp.push_back(arr[j]);
                    temp.push_back(arr[k]);
                    sort(temp.begin(), temp.end());
                    s.insert(temp);
                }
            }
        }
    }
    for(auto x: s) {
        ans.push_back(x);
    }
    return ans;
}
```

Q7. Sort 0, 1

Approach: ① Traverse array & update count of 0 and 1. Again, traverse the array and fill 0s for the first numZero indices and 1 for the remaining indices.

② Maintain 2 pointing variables i and j, i=0 & j=n-1. In any condition, if:

i) arr[i] == 0 → i++
ii) arr[j] == 1 → j--
iii) arr[i] == 1 && arr[j] == 0 → swap them
i++, j--

Dry Run: arr[]: {1,0,0,1,1,1,0,1}

{1,0,0,1,1,1,0,1} → decrement j;
{1,0,0,1,1,1,0,1} → swap(arr[i++], arr[j--]);
{1,0,0,1,1,1,0,1} → increment i;
{1,0,0,1,1,1,0,1} → increment i;
{1,0,0,1,1,1,0,1} → decrement j;
{1,0,0,1,1,1,0,1} → decrement j;
{1,0,0,1,1,1,0,1} →

Code :

```
void printArr(int arr[], int n) {
    for(int i=0;i<n;i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

void sortOneTwo(int arr[], int n) {
    int i = 0, j = n-1;
    while(i < j) {
        if(arr[i] == 0) {
            i++;
        }
        else if(arr[j] == 1) {
            j--;
        }
        else if(arr[i] == 1 && arr[j] == 0) {
            swap(arr[i], arr[j]);
            i++;
            j--;
        }
    }
}
```

```
8
1 1 0 0 0 0 1 0

Output.txt
0 0 0 0 0 1 1 1
```

Homework : Sort 0 1 2.

Approach : Take 3 pointing variables i, j and k where i, j and k are such that :

arr[0.... i-1] = 0
arr[k+1 .... n-1] = 2

i = 0, j = 0 and k = n-1.

arr[] : {0, 1, 1, 0, 2, 1, 0}
i j k

If arr[j] == 0, swap(arr[i], arr[j]) and then i++, j++.

If arr[j] == 1, increment j.

If arr[j] == 2, swap(arr[j], arr[k]) and then k--

{0, 1, 1, 0, 2, 1, 0} → swapping arr[i], arr[j] & i++, j++.
i j k

{0, 1, 1, 0, 2, 1, 0} → j++
i j k

{0, 1, 1, 0, 2, 1, 0} → j++
i j k

{0, 1, 1, 0, 2, 1, 0} → j++
 i j   K

{0, 1, 1, 0, 2, 1, 0} → swapping arr[i], arr[j] & i++, j++.
 i j   K

{0, 0, 1, 1, 2, 1, 0} → Swapping arr[j], arr[K] & K--
    i j   K

{0, 0, 1, 1, 0, 1, 2} → swapping arr[i], arr[j] & i++, j++.
    i j K

{0, 0, 0, 1, 1, 1, 2} → j++
       i j K

{0, 0, 0, 1, 1, 1, 2} →
       i K j

Observe how we are mainting arr[0....i-1] = 0 and

arr[K+1 ... n-1] = 2.

Why are we not incrementing j when swapping (arr[j], arr[K]) ?

arr[K] might be 0. If I swap arr[j] with arr[K], then we will get arr[j] = 0, we want swap this with arr[i] in the next step so we should not omit arr[j] by incrementing j.

Code :

```
void sort012(int *arr, int n)
{
    int i = 0, j = 0, k = n-1;
    while(j <= k) {
        if(arr[j] == 1)
            j++;
        else if(arr[j] == 0) {
            swap(arr[i++], arr[j++]);
        }
        else {
            swap(arr[j], arr[k--]);
        }
    }
}
```