Q-1) Count the number of digits in the given number?

```
void count_digits_1(int num){
   int ct=0;
   while(num){
      num/=10;
      ct++;
   }
   cout<<ct<<endl;
}
```

We divide the number by 10 and simultaneously increment the counter until the number becomes zero. The final value in the counter will have the number of digits in the original number.

Complexity :

Time : O(n) where "n" is number of digits

Space: O(1)

```
void count_digits_2(int num){
   string n = to_string(num);
   cout<<n.length();
}
```

In the above mentioned approach, we convert the number to string and then use inbuilt .length() function to calculate the length of the string which in this case would be the total number of digits in the number.

Complexity :

Time : O(1)

Space: O(1)

```
void count_digits_3(int num){
   cout<<floor(log10f(num))+1<<endl;
}
```

Log of any number "n" to the base "x" denotes how many times x multiply to itself to become n. For example Log(100) to base 10 will be 2 as 10^2 = 100. Similarly log(1000) to base 10 will be 3. This means log of any number between 100 - 1000 to base 10 will be between(2,3) and the floor of that value + 1 will be 3 which is the number of digits in each of all the numbers from 100 to 999.

Complexity :

Time : O(1)

Space: O(1)

Q-2) Reverse the given number.

```
void reverse(int x)
{
   long long int rev = 0;
   while(x){
      rev = rev*10 + x%10;
      x/=10;
   }
   // if rev exceeds the signed 32-bit integer range [-(2^31), 2^31 - 1] then it
will have a garbage value
   // hence we use the below code to return 0 in that case
   cout<<(rev<pow(2,31)-1 && rev>-pow(2,31))*rev<<endl;
   //cout<<(rev<INT_MAX && rev>INT_MIN)*rev<<endl;
   // INT_MAX and INT_MIN can be used instead of pow(2,31)-1 and -
pow(2,31)

   /*
   Complexity
   time: O(n) where "n" is number of digits
   space: O(1)
   */

}
```

We take the last digit of number and store it in the variable "rev", then we multiply the rev with 10 and add the 2nd last digit to it and store the result in rev. This loop goes on until the number becomes 0.

Complexity:

Time : O(n) where "n" is number of digits

Space: O(1)

Q-3) Check if given number is palindrome or not?

```
string check_palindrome(int x)
{
   if (x<0) return "Not a Palindrome";
   long long int rev = 0, num=x;
   while(x>rev){
      rev = rev*10 + x%10;
      x/=10;
   }
   // the above while loop reverses only the last half of the integer
   if(x==rev || x==rev/10) // we use 2nd condition in case of odd num of
digits
      return "Palindrome";
   else
      return "Not a Palindrome";
   // works only for x in the signed 32-bit integer range [-(2^31), 2^31 - 1]
   /*
   Complexity
   time: O(n/2) where "n" is number of digits
   space: O(1)
   */
}
```

Brute force approach : Reverse whole number and check if it equals the original number or not.

Optimize approach: A number is considered as a palindrome if it reads the same forwards and backwards. Hence if the reverse of the 2nd half of a number equals the 1st half then the number is palindrome. Hence we run the loop and at each iteration we keep extracting the last digit of number and at the same time form the reverse of 2nd half: Refer (Ques -2 of Basic Maths section). When the current value in "rev" becomes >= current value of x, we break the loop. Next if there were even number of digits in the original number, then x must be equal to rev else x = rev/10 for a number to be palindrome.

Complexity:

Time: O(n/2) where "n" is number of digits

Space: O(1)

Q-4) Calculate the GCD of given two numbers.

```
void gcd_brute_force(int n, int m){
   if(n<m){
      n = n + m;
      m = n - m;
      n = n - m;
   }
   int gcd = m;
   while(gcd>0){
      if(n%gcd==0 && m%gcd==0){
         cout<<gcd<<endl;
         break;
      }
      gcd--;
   }
   /*
   we all know that the GCD value of two numbers cannot exceed the smallest of the two numbers
   hence we intiate the gcd from smallest number and decrement it while we get a number that divides both the number
   complexity
   worst case that is for gcd(23,21), it will be O(m) where m is the smallest number

   Complexity
   time: O(min(n,m)) where is number of digits
   space: O(1)
   */
}
```

we all know that the GCD value of two numbers cannot exceed the smallest of the two numbers hence we intiate the gcd from smallest number and decrement it while we get a number that divides both the number.

Complexity:

Time: O(min(n,m))

Space: O(1)

```
void gcd_optimized(int n, int m){
   n+=m;
   do{
      n-=m;
      if(n<m){
```

```
        n = n + m;
        m = n - m;
        n = n - m;
        }
    }while(n%m);
    cout<<m<<endl;
}
```

1) find the smallest of two numbers
2)check whether the smallest divides biggest
3) if not then decrement the value of biggest number by smallest number
4) go back to 1
5) if yes then print the current smallest number

gcd(60,315)
60 = 3x5x4
315 = (3^2)x5x7
gcd is 15
so we can rewrite it as
60 = 15*4
315 = 15*21
if we repeatedly deduct the smallest value from biggest number and update the numbers every time 15 will come out as common 15(21-4), 15(17-4), 15(13-4),...15(5-4), after this we will have two numbers 15 and 60, 15 divides both 15 and 60 hence gcd is 15

Complexity:
Time : O(n/m)
Space: O(1)

```
void gcd_super_optimized(int n,int m){
    int rem;
    if(n<m){
        n = n + m;
        m = n - m;
        n = n - m;
    }
    while(n%m){
        rem = n%m;
        n = m;
        m = rem;
    }
    /*
    */
    cout<<m<<endl;

    /*
```

```
   */

}
check euclidean method for finding gcd for understanding the logic
      gcd(35,9) -> 35%9 = 8 -> 9%8 = 1 -> 8%1 -> 0
      the last nonzero remainder is the gcd which is in this case is 1
Complexity:
      Time : O(n/m)
      Space: O(1)
```

Q-5) Check if the given number is Armstrong number or not.

```
void is_amstrong(int n){
   int k=0,sum=0,m=n;
   while(n){
      k++;
      n/=10;
   }
   n=m;
   while(n){
      sum+=pow(n%10,k);
      n/=10;
   }
   if(sum==m) cout<<"It is an amstrong number"<<endl;
   else cout<<"It is not an amstrong number"<<endl;
}
```

A number is said to be armstrong number if sum of each of its digits raised to power: total number of digits in the number is equal to the number itself.
We calculate the total number of digits in the number and store it in "k" and then we extract the last digit and raise it to power k and add that to "sum".
Complexity:

Time : O(n)+O(n) where "n" is number of digits
Space: O(1)

Q-6) Print all the divisors of the given number.

```
void divisors(int num){
   int i = 1;
   while(i<num){
      if(num%i==0){
         cout<<i<<" ";
      }
      i++;
   }
   cout<<num<<endl;
   /*
   time complexity: O(n)
   advantage: output is sorted
   */
}
```

Run the loop from 1 to num, if any number divides num then print it.
Complexity:
Time : O(n) where n is number
Space: O(1)

```
void divisors_optimized(int num){
   int i = 1, j=0;
   for(int i=1; i<=sqrt(num); i++){
      if(num%i==0){
         j = num/i;
         if(i==j){
            cout<<i<<endl;
         }
         else
            cout<<i<<" "<<j<<" ";
      }
   }
   /*
   */
}
```

all the divisors are paired to another divisor that when multiplied together makes a number so if we find 1 dividor, then the quotient is also a divisor.
the square root of the number splits the pairs

```
   36: 1, 2, 3, 4, 6, 9, 12, 18, 36
        |  |  |  |  ^  |  |  |  |
        |  |  |   -------  |  |  |
        |  |  |------------ |  |  |
```

Complexity:

Time : O(sqrt(n))

Space: O(1)

Disadvantage: output is not sorted

Q-7) Check if given number is prime or not.

```
bool isPrime(int n)
{
   if(n==1) return false;
   for(int i=2;i<=sqrt(n);i++){
      if(n%i==0){
         return false;
      }
   }
   return true;
   /*
   */
}
```

If there are no divisors till sqrt(n), that means there are no more divisors as the sqrt(n) splits the pairs of divisors. Refer Ques-6 in Basic Maths section for better understanding.

Complexity:

Time : O(sqrt(n))

Space: O(1)

Q-1) Print name "n" times.

```
#include<bits/stdc++.h>
using namespace std;
void printNtimes(string a, int n){
   if(n==0){
      return;
   }
   cout<<a<<" ";
   printNtimes(a,n-1);
}
int main()
{
   int num;
   string name;
   cout<<"Enter name and number: ";
   cin>>name>>num;
   printNtimes(name,num);
}
```

Complexity:

Time : O(n)

Space: O(n) -> worst case which will happen if the recursion stack becomes full.

Q-2) Print 1 to N using Recursion.

```cpp
#include<bits/stdc++.h>
using namespace std;

// a backtracking approach aka backward recursion: first goes to the depth and then print and comes out
void print1ton(int n)
{
   if(n==0){
      return;
   }
   print1ton(n-1);
   cout<<n<<" ";
}

// forward recursion: first, number is printed and then function is called
void print1ton_2(int i, int n){
   if(i>n){
      return;
   }
   cout<<i<<" ";
   print1ton_2(i+1,n);
}
int main()
{
   print1ton(5);
   cout<<endl;
   print1ton_2(1,3);
   return 0;
}
```

Complexity:

Time : O(N) -> Since the function is being called n times, and for each function, we have only one printable line that takes O(1) time, so the cumulative time complexity would be O(N)

Space: O(N) -> In the worst case, the recursion stack space would be full with all the function calls waiting to get completed and that would make it an O(N) recursion stack space

Q-3) Print N to 1 using Recursion

```
#include<bits/stdc++.h>
using namespace std;

// forward recursion: first, number is printed and then function is called
void printnto1(int n)
{
   if(n==0){
      return;
   }
   cout<<n<<" ";
   printnto1(n-1);
}

// a backtracking approach aka backward recursion: first goes to the depth and then print and comes out
void printnto1_2(int i, int n){
   if(i>n){
      return;
   }
   printnto1_2(i+1,n);
   cout<<i<<" ";
}
int main()
{
   printnto1(7);
   cout<<endl;
   printnto1_2(1,4);
   return 0;
}
```

Complexity:

Time : O(N) -> Since the function is being called n times, and for each function, we have only one printable line that takes O(1) time, so the cumulative time complexity would be O(N)

Space: O(N) -> In the worst case, the recursion stack space would be full with all the function calls waiting to get completed and that would make it an O(N) recursion stack space

Q-4) Calculate sum of first N numbers.

```
#include<bits/stdc++.h>
using namespace std;

int sum_1ton(int s){
   if(s==0){
      return 0;
   }
   s += sum_1ton(s-1);
   return s;
}

int sum_1ton_2(int i, int sum){
   if(i<1){
      return 0;
   }
   sum = i + sum_1ton_2(i-1,sum);
   return sum;
}

int main()
{
   int num;
   cin>>num;
   cout<<sum_1ton_2(num,0);
   return 0;
}
```

We can also use formula: sum of 1 to n = n(n+1)/2 but just for practice we are using recursion.

Complexity:

Time : O(n)

Space: O(n) worst case if the recursion stack is full

Q-5) Factorial of given number.

```
#include<bits/stdc++.h>
using namespace std;
int factorial(int n){
   if(n<1){
      return 1;
   }
   return n*factorial(n-1);
}

int main()
{
   int num;
   cin>>num;
   cout<<factorial(num);
   return 0;
}
```

Complexity:

Time : O(N) -> Since the function is being called n times, and for each function, we have only one printable line that takes O(1) time, so the cumulative time complexity would be O(N)

Space: O(N) -> In the worst case, the recursion stack space would be full with all the function calls waiting to get completed and that would make it an O(N) recursion stack space

Q-6) Reverse an array.

```
void reverse_array(int arr[], int i, int j){
   if(i>=j){
      return;
   }
   int temp;
   temp = arr[j];
   arr[j] = arr[i];
   arr[i] = temp;
   reverse_array(arr,i+1,j-1);
}
```

for swapping we can also use swap(arr[i],arr[j])
or

```
arr[i] = arr[i]+arr[j];
arr[j] = arr[i]-arr[j];
arr[i] = arr[i]-arr[j];
```

this will save extra space
we can also use library function reverse(arr,arr+n)
Complexity:

Time : O(n)
Space: O(1)

Q-7) Check if given string is palindrome or not.

```
bool is_alphanum(char c){
   return (c>='A' && c<='Z') || (c>='a' && c<='z') || (c>='0' && c<='9');
}

char to_lower(char c){
   return (c>='A' && c<='Z')?c+32:c;
}

// recursive approach
bool check_palindrome(string s, int i, int j, int l){
   if(i>=j) return true;
   while(!(is_alphanum(s[i]))){
      i++;
      if(i==l) break;
   }
   while(!(is_alphanum(s[j]))){
      j--;
      if(j<0) break;
   }
   if(i==l && j<0) return true;
   if(to_lower(s[i])==to_lower(s[j])){
      return true && check_palindrome(s,i+1,j-1,l);
   }
   else return false;
}

bool check_palindrome_without_recursion(string s){
   int i=0, l=s.size();
   int j=l-1;
   while(i<j){
      while(!(is_alphanum(s[i]))){
         i++;
         if(i==l) break;
      }
      while(!(is_alphanum(s[j]))){
         j--;
         if(j<0) break;
      }
      if(i==l && j==0) return true;
      if(!(to_lower(s[i++])==to_lower(s[j--]))) return false;
   }
```

```
    return true;
}
```

We use two pointer approach. i points to 0th index and j points to last. We increment i and decrement j until we find the alphanumeric character at both ends. Then we check if both the chars are same or not irrespective of upper case or lower case.
If in case all the letters are non-alphanumeric then i will reach the rightmost end and j will reach the leftmost end and in this case the string will be considered as a palindrome.
Complexity:

Time : O(n) -> actually takes n/2 time

Space: O(1)

Q-8) Fibonacci Number

```
// recursive approach
int fib(int n) {
   if(n==0) return 0;
   if(n==1) return 1;
   return fib(n-1) + fib(n-2);
}
Complexity:
      Time : O(2^n)
      Space: O(n) --> if stack overflows

int fib_without_recursion(int n){
   if(n==0) return 0;
   else if(n==1) return 1;
   int i=0, j=1;
   while(n>=2){
      j+=i;
      i=j-i;
      n--;
   }
   return j;
}
Complexity:
      Time : O(n)
      Space: O(1)
```

Q-1) Count Frequency

Given an array of integers: [1, 2, 1, 3, 2] and we are given some queries: [1, 3, 4, 2, 10].
For each query, we need to find out how many times the number appears in the array.
For example, if the query is 1 our answer would be 2, and if the query is 4 the answer will be 0.
Brute Force:
take each from query arr and compare it with each in int arr, if match then increase the counter.
takes O(Q*N)
Optimized approach (using hashing):
1) Assume or find the highest possible number in query array
2) Pre-storing : Declare hash array of the size equal to highest number + 1(for ease of referencing)
3) Fetching : for each num in query array, fetch hash[num] and increase the value by 1

```
// Array Hashing
vector<int> count_num_freq(int arr[], int n){
   vector<int> hash(13,0); // we assume 12 is the highest number
   for(int i=0; i<n; i++){
      hash[arr[i]]++;
   }
   return hash;
}
vector<int> count_char_freq(string s, int n){
   vector<int> hash(26,0); // if s contains both upper case and lower case then adjust size accordingly
   for(int i=0; i<n; i++){
      hash[s[i]-'a']++; // if s contains only upper case then hash[s[i]-'A'], if it contains both then simply hash[s[i]]
   }
   return hash;
}

// hashing using map (for hashing numbers like 10^9 or higher)
map<int, int> count_num_freq_map(int arr[], int n){
   map<int, int> freq_map;
   for(int i=0; i<n; i++){
      freq_map[arr[i]]++;
   }
```

```
    return freq_map;
}
map<char, int> count_char_freq_map(string s, int n){
    map<char, int> freq_map;
    for(int i=0; i<n; i++){
        freq_map[s[i]]++;
    }
    return freq_map;
}
```

If we want to fetch the value of a key that does not exist in the map, the map will always return 0 in C++ and null in Java.

The total time complexity will be O(N * time taken by map data structure).

Storing (i.e. insertion) and fetching (i.e. retrieval) in a C++ map, both take always O(logN) time complexity, where N = the size of the map. But the unordered_map in C++ and HashMap in Java, both take O(1) time complexity to perform storing (i.e. insertion) and fetching (i.e. retrieval). Now, it is valid for the best case and the average case. But in the worst case, this time complexity will be O(N) for unordered_map.

Now, the worst case occurs very very rarely. It almost never happens and most of the time, we will be using unordered_map. Our first priority will be always to use unordered_map and then map. If unordered_map gives a time limit exceeded error(TLE), we will then use the map.
The time complexity in the worst case is O(N) because of the internal collision.

In the map data structure, the data type of key can be anything like int, double, pair<int, int>, etc. But for unordered_map the data type is limited to integer, double, string, etc.
We cannot have an unordered_map whose key is pair<int, int>.

Q-2) Find the highest and lowest frequency element.

Given an array 'v' of 'n' numbers. Your task is to find and return the highest and lowest frequency elements.
If there are multiple elements that have the highest frequency or lowest frequency, pick the smallest element.

```
vector<int> getFrequencies(vector<int>& v) {
   unordered_map<int, int> hash;
   vector<int> ans(2);
   for(auto it: v) hash[it]++;
   int max=INT_MIN, min=INT_MAX;
   for(auto it: hash){
      if( it.second>max || (it.second==max && it.first<ans[0]) ){
         max = it.second;
         ans[0] = it.first;
      }
      if( it.second<min || (it.second==min && it.first<ans[1]) ){
         min = it.second;
         ans[1] = it.first;
      }
   }
   return ans;
}
```

Complexity:
Time : O(n), where n = size of the array. The insertion and retrieval operation in the map takes O(1) time.
Space: O(n), where n = size of the array. It is for the map we are using.

Selection Sort

```
void selection_sort(int *arr, int n){
   int min_idx, temp;
   for(int i=0; i<n-1; i++){
      min_idx = i;
      for(int j=i+1; j<n; j++){
         if(arr[j]<arr[min_idx]) min_idx = j;
      }
      temp = arr[min_idx];
      arr[min_idx] = arr[i];
      arr[i] = temp;
   }
}
```

Complexity:

Time : O(n^2) (where n = size of the array), for the best, worst, and average cases.

Space: O(1)

Bubble Sort

```
void bubble_sort(int *arr, int n){
   int temp;
   bool swap=true;
   for(int i=n-1; i>=0; i--){
      if(swap){
         swap = false;
         for(int j=0; j<=i-1; j++){
            if(arr[j]>arr[j+1]){
               temp = arr[j];
               arr[j] = arr[j+1];
               arr[j+1] = temp;
               swap = true;
            }
         }
      }
      else break;
   }
}
```

Complexity:

Time : O(n^2) for the worst and average cases and O(n) for the best case.

Space: O(1)

Insertion Sort

```
void insertion_sort(int *arr, int n){
   int temp,j;
   for(int i=1; i<n; i++){
      j=i;
      while(arr[j]<arr[j-1] && j>0){
         temp = arr[j];
         arr[j] = arr[j-1];
         arr[j-1] = temp;
         j--;
      }
   }
}
```

Complexity:

Time : O(n^2), (where n = size of the array), for the worst, and average cases.

Space: O(1)

Best Case Time Complexity:

The best case occurs if the given array is already sorted. And if the given array is already sorted, the outer loop will only run and the inner loop will run for 0 times. So, our overall time complexity in the best case will boil down to O(N),where N = size of the array.

Merge Sort

```
void merge(int *arr, int l, int m, int h){
   int i=l, j=m+1, temp[h-l+1], k=0;
   while(i<=m && j<=h){
      if(arr[i]<arr[j]){
         temp[k] = arr[i];
         i++;
      }
      else{
         temp[k] = arr[j];
         j++;
      }
      k++;
   }
   while(i<=m){
      temp[k] = arr[i];
      k++;
      i++;
   }
   while(j<=h){
      temp[k] = arr[j];
      k++;
      j++;
   }
   for(int t=0; t<=h-l; t++) arr[l+t]=temp[t];
}
void merge_sort(int *arr, int l, int h){
   if(l>=h) return;
   int m = (l+h)/2;
   merge_sort(arr,l,m);
   merge_sort(arr,m+1,h);
   merge(arr,l,m,h);
   return;
}
```

Complexity:

Time : O(nlog(n))

Reason: At each step, we divide the whole array, for that logn and we assume n steps are taken to get sorted array, so overall time complexity will be nlogn

Space: O(n)

Reason: We are using a temporary array to store elements in sorted order

Quick Sort

```
int partition(int *arr, int l, int h){
   int p=arr[h],j=l-1;
   for(int i=l; i<h; i++){
      if(arr[i]<p){
         j++;
         swap(arr[i],arr[j]);
      }
   }
   swap(arr[j+1],arr[h]);
   return j+1;
}
int partition_2(int *arr, int l, int h){
   int p=arr[h],i=l,j=h;
   while(i<j){
      while(arr[i]<=p && i<h) i++;
      while(arr[j]>=p && j>l) j--;
      if(i<j)
         swap(arr[i],arr[j]);
   }
   swap(arr[i],arr[h]);
   return i;
}
void quick_sort(int *arr, int l, int h){
   if(l>=h) return;
   int p = partition_2(arr, l, h);
   quick_sort(arr,l,p-1);
   quick_sort(arr,p+1,h);
}
```

Complexity:

Time : O(nlog(n))

At each step, we divide the whole array, for that logn and n steps are taken for the partition() function, so overall time complexity will be n*logn.

Worst Case – This case occurs when the pivot is the greatest or smallest element of the array. If the partition is done and the last element is the pivot, then the worst case would be either in the increasing order of the array or in the decreasing order of the array.

Worst Case Time complexity: $O(n^2)$

Best Case – This case occurs when the pivot is the middle element or near to middle element of the array.

Time Complexity for the best and average case: O(n*logn)

Space: O(1) + O(N) auxiliary stack space.

1) Find the largest element in an array.

```
int brute_force_approach(int arr[], int n){
   sort(arr,arr+n);
   return arr[n-1];
}
/*
   Time Complexity: O(N*log(N))
   Space Complexity: O(1)
*/

int optimal_approach(int arr[], int n){
   int max_elem = arr[0];
   for(int i=0; i<n; i++) if(arr[i]>max_elem) max_elem=arr[i];
   return max_elem;
}
/*
   Time Complexity: O(N)
   Space Complexity: O(1)
*/
int main()
{
   int n;
   cin>>n;
   int arr[n];
   for(int i=0; i<n; i++) cin>>arr[i];
   cout<<brute_force_approach(arr,n)<<endl;
   cout<<optimal_approach(arr, n);
   return 0;
}
```

2) Find 2nd largest element.

```
int brute_force_approach(int arr[], int n){
   sort(arr,arr+n);
   int j = n-2;
   while(arr[j]==arr[n-1] && j>=0) j--;
   if(j>=0) return arr[j];
   return arr[n-1];
}
/*
   Time Complexity: O(N*log(N))
   Space Complexity: O(1)
*/
int better_approach(int arr[], int n){
   int max_ele=arr[0];
   for(int i=0; i<n; i++) max_ele = max(max_ele,arr[i]);
   int s_max=arr[0];
   for(int i=0; i<n; i++) if(arr[i]!=max_ele) s_max = max(s_max,arr[i]);
   return s_max;
}
/*
   Time Complexity: O(N) double pass
   Space Complexity: O(1)
*/

int optimal_approach(int arr[], int n){
   int max=arr[0],s_max = arr[0];
   for(int i=0; i<n; i++){
      if(arr[i]>max){
         s_max = max;
         max = arr[i];
      }
      else if(arr[i]>s_max || s_max==max) s_max = arr[i];
   }
   return s_max;
}
/*
   Time Complexity: O(N) single pass
   Space Complexity: O(1)
*/
int main()
{
   int n;
```

```
    cin>>n;
    int arr[n];
    for(int i=0; i<n; i++) cin>>arr[i];
    cout<<brute_force_approach(arr,n)<<endl;
    cout<<optimal_approach(arr, n)<<endl;
    cout<<better_approach(arr,n);
    return 0;
}
```

3) Check if array is sorted.

```
bool brute_force(int arr[], int n){
   int i=0, j=0, k=0;
   while(i<n-1){
      if(arr[i]<arr[i+1]) j++;
      else if(arr[i]>arr[i+1]) k++;
      else
      {
         j++;
         k++;
      }
      i++;
   }
   if(j==n-1 || k==n-1)
      return true;
   return false;
}
/*
   Time complexity: O(N) best/avg/worst
   Space complexity: O(1)
*/
bool better_approach(int arr[], int n){
   int last_dir=0, curr_dir=0 ,i=0;
   while(i<n-1){
      last_dir = curr_dir;
      if(arr[i]<arr[i+1]) curr_dir=1;
      else if(arr[i]>arr[i+1]) curr_dir=-1;
      else curr_dir=last_dir;
      if(curr_dir!=last_dir && last_dir!=0) return false;
      i++;
   }
   return true;
}
/*
   Time complexity: O(N) avg/worst(when all the elements are sorted)
   Best case occurs in case where the order is not maintained within inital(3 to
4) elements
   Space complexity: O(1)
*/
```

4) Remove Duplicates from sorted array

```
void better_approach(int *arr, int& n){
   for(int i=0; i<n-1; i++){
      int key=arr[i];
      while(arr[i+1]==key && i<n-1){
         arr[i+1]=INT_MIN;
         i++;
      }
   }
   int i,j=0;
   for(i=0; i<n; i++){
      while(arr[j]!=INT_MIN && j<n) j++;
      i=j;
      while(arr[i]==INT_MIN && i<n) i++;
      if(i<n)
         swap(arr[i],arr[j]);
   }
   n=j;
}
/*
   Time complexity: O(N) + O(N)
   Space complexity: O(1)
*/
void brute_force_approach(int *arr, int& n){
   set<int> s;
   for(int i=0; i<n; i++) s.insert(arr[i]);
   int i=0;
   for(auto it: s){
      arr[i] = it;
      i++;
   }
   n = i;
}
/*
   Time complexity: O(nlog(n)) + O(n)
   Space complexity: O(n)
*/
void optimal_approach(int *arr, int& n){
   int i=0;
   for(int j=1; j<n; j++){
      if(arr[j]!=arr[i]){
```

```
            i++;
            arr[i] = arr[j];
        }
    }
    n=i+1;
}
/*
    Time Complexity: O(n)
    Space Complexity: O(1)
*/
```

5) Left rotate an array by one place.

```
void brute_force_approach(int *arr, int n){
   int a = arr[0];
   for(int i=1;i<n;i++) arr[i-1]=arr[i];
   arr[n-1] = a;
}
/*
   Time Complexity: O(n), as we iterate through the array only once.
   Space Complexity: O(1) as no extra space is used
*/
```

6) Left rotate an array by D places

```
void rotate_right_brute_force(int *arr, int n, int k){
   for(int i=1; i<=k; i++){
      int temp = arr[n-1],j;
      for(j=n-1; j>0; j--)
         arr[j]=arr[j-1];
      arr[j] = temp;
   }
}
/*
   Time Complexity: O(N*K)
   Space Complexity: O(1) - no extra space used
*/
void rotate_right_better_approach(int *arr, int n, int k){
   int a[n];
   for(int i=0; i<n; i++){
      a[(i+k)%n] = arr[i];
   }
   for(int i=0; i<n; i++) arr[i]=a[i];
}
/*
   k = 2, n = 6
   0 1 2 3 4 5
   4 5 0 1 2 3
   4-->0 (4+2)%6 = 0
   5-->1 (5+2)%7 = 1
   (i+k)%n = new_idx;
*/
/*
   Time Complexity: O(N) + O(N)
   Space Complexity: O(N)
*/
void reverse(int *arr, int s, int e){
   while(s<=e){
      swap(arr[s],arr[e]);
      s++;
      e--;
   }
}
void rotate_right_optimal_approach(int *arr, int n, int k){
   k=k%n;
```

```
    reverse(arr, 0, n-k-1);
    reverse(arr, n-k, n-1);
    reverse(arr,0,n-1);
}
/*
    1 2 3 4 5
    k = 2, n = 5
    reversing first n-k elements
    2 1 3 4 5
    reversing last k elements
    2 1 5 4 3
    reversing whole array
    3 4 5 1 2
    Time complexity: O(N) + O(N)
    Space Complexity: O(1)
*/
void rotate_left_optimal_approach(int *arr, int n, int k){
    k =  k%n;
    rotate_right_optimal_approach(arr,n,n-k);
}
/*
    rotating array left by k times is equivalent to rotating array right n-k
times;
    Time complexity: O(N) + O(N)
    Space Complexity: O(1)
*/
```

7) Move Zeros to end

```
void brute_force_approach(int *arr, int n){
   int a[n]={0},j=0;
   for(int i=0; i<n; i++){
      if(arr[i]){
         a[j]=arr[i];
         j++;
      }
   }
   for(int i=0; i<n; i++) arr[i]=a[i];
}
/*
   TC: O(N)+O(N)
   SC: O(N)
*/
void better_approach(int *arr, int n){
   int i=0, j=0;
   while(i<n && j<n){
      while(arr[j]!=0 && j<n) j++;
      while(arr[i]==0 && i<n) i++;
      if(i>j && i<n && j<n)
         swap(arr[i],arr[j]);
      else if(i<j) i++;
   }
}
/*
   TC: O(N)
   SC: O(1)
*/


void optimal_approach(vector<int>& nums) {
   int n = nums.size();
   int i=0;
   for(int j=0;j<n;j++){
      if(nums[j]){
         swap(nums[i],nums[j]);
         i++;
      }
   }
}
```

```
/*
   TC: O(N)
   SC: O(1)
*/
```

8) Linear Search

```
int brute_force_approach(int arr[], int n, int k){
    for(int i=0; i<n; i++) if(arr[i]==k) return i;
    return -1;
}
/*
    TC: O(N)+O(N)
    SC: O(N)
*/
```

9) Find the Union

```
/*
Problem Statement: Given two sorted arrays, arr1, and arr2 of size n and m.
Find the union of two sorted arrays.
The union of two arrays can be defined as the common and distinct elements
in the two arrays.
NOTE: Elements in the union should be in ascending order.
*/

void brute_force(int a[], int b[], int n, int m, int *arr){
   for(int i=0; i<n+m; i++) arr[i] = INT_MAX;
   int j=1,isp=0,ct=1;
   arr[0] = a[0];
   for(int i=1; i<n; i++){
      isp=0;
      for(int k=0; k<ct; k++){
         if(arr[k]==a[i]){
            isp=1;
            break;
         }
      }
      if(isp) continue;
      arr[j] = a[i];
      while(arr[j]<arr[j-1] && j>0){
         swap(arr[j],arr[j-1]);
         j--;
      }
      ct++;
      j=ct;
   }
   for(int i=0; i<m; i++){
      isp=0;
      for(int k=0; k<ct; k++){
         if(arr[k]==b[i]){
            isp=1;
            break;
         }
      }
      if(isp) continue;
      arr[j] = b[i];
      while(arr[j]<arr[j-1] && j>0){
```

```
            swap(arr[j],arr[j-1]);
            j--;
        }
        ct++;
        j=ct;
    }
}
/*
    TC: O(N) + O(N^2) + O(N^2) + O(M^2) + O(MN)
    SC: O(N+M)
*/

void better_approach(int a[], int b[], int n, int m, int *arr){
    set<int> s;
    for(int i=0; i<n; i++) s.insert(a[i]);
    for(int i=0; i<m; i++) s.insert(b[i]);
    int i=0;
    for(auto it: s){
        arr[i] = it;
        i++;
    }
}
/*
    TC: O(n*log(n+m) + m*log(m+n) + n + m)
    SC: O(2*(N+M))
*/

void optimal_approach(int a[], int b[], int n, int m, int* arr){
    int i=0,j=0,k=0;
    if(a[i]<=b[j]){
        arr[k] = a[i];
        k++;
        i++;
    }
    else{
        arr[k]=b[j];
        k++;
        j++;
    }
    while(i<n && j<m){
        while(a[i]<=arr[k-1] && i<n) i++;
        while(b[j]<=arr[k-1] && j<m) j++;
        if(i<n && j<m){
            if(a[i]<=b[j]){
```

```
            arr[k] = a[i];
            k++;
            i++;
        }
        else{
            arr[k] = b[j];
            k++;
            j++;
        }
    }

  }
  while(a[i]<=arr[k-1] && i<n) i++;
  while(b[j]<=arr[k-1] && j<m) j++;
  while(a[i]>arr[k-1] && i<n){
    arr[k]=a[i];
    k++;
    i++;
  }
  while(b[j]>arr[k-1] && j<m){
    arr[k]=b[j];
    k++;
    j++;
  }
}
/*
Two Pointer approach (similar to merge algorithm of merge sort)
  TC:O(N+M)
  SC:O(N+M) for worst case i.e. there are no common elements
  Does not work for unsorted arrays
*/
```

10) Find missing number in array.

```
/*
   Problem Statement: 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.
*/
int brute_force(int arr[],int n){
   int is_mis;
   for(int i=1; i<=n; i++){
      is_mis=1;
      for(int j=0; j<n-1; j++){
         if(i==arr[j]){
            is_mis=0;
            break;
         }
      }
      if(is_mis) return i;
   }
   return 0;
}
/*
   TC: O(N^2)
   SC: O(1)
*/
int better_approach(int arr[], int n){
   int a[n+1] = {0};
   for(int i=0; i<n-1; i++){
      a[arr[i]] = 1;
   }
   for(int i=1; i<n+1; i++) if(!a[i]) return i;
   return 0;
}
/*
   TC: O(2N)
   SC: O(N+1)
*/
int optimal_approach_1(int arr[], int n){
   int sum=n*(n+1)/2;
   for(int i=0; i<n-1; i++) sum-=arr[i];
   return sum;
}
/*
   TC: O(N)
   SC: O(1)
```

```
*/
int optimal_approach_2(int arr[], int n){
    int xor1=0,xor2=0;
    for(int i=0; i<n-1; i++){
        xor1 ^=arr[i];
        xor2 ^=(i+1);
    }
    xor2 ^=n;
    return xor1 ^xor2;
}
/*
    XOR Property:
        a^a = 0
        a^0 = a
    TC: O(2N)
    SC: O(1)
*/
```

11) Count max consecutive ones

```
/*
   Problem Statement: Given an array that contains only 1 and 0 return the
count of maximum consecutive ones in the array.
*/
int brute_force(int arr[], int n){
   int sum=0,mx_sum=0;
   for(int i=0; i<n; i++){
      if(arr[i]) sum++;
      else{
         cout<<mx_sum<<" "<<sum<<endl;
         mx_sum=max(mx_sum,sum);
         sum=0;
      }
   }
   return max(mx_sum,sum);
}
/*
   TC: O(N)
   SC: O(1)
*/
```

12) Find the number appearing only once.

```
/*
   Problem Statement: Given a non-empty array of integers arr, every element
appears twice except for one. Find that single one.
*/
int brute_force(int arr[], int n){
   int ct;
   for(int i=0; i<n; i++){
      ct=0;
      for(int j=0; j<n; j++){
         if(arr[i]==arr[j]) ct++;
      }
      if(ct==1) return arr[i];
   }
   return -1;
}
/*
   TC: O(N^2)
   SC: O(1)
*/
int better_approach(int arr[], int n){
   map<int, int> m;
   for(int i=0; i<n; i++) m[arr[i]]++;
   for(auto it: m) if(it.second==1) return it.first;
   return 0;
}
/*
   TC: O(n*log(n))
   SC: O(N)
*/
int optimal_approach(int arr[], int n){
   int a=0;
   for(int i=0; i<n; i++) a^=arr[i];
   return a;
}
/*
   XOR Property:
      a^a = 0
      0^a = a
*/
int optional_approach(int arr[], int n){
   int i=0;
```

```
    for(i=0; i<n-2; i+=2){
        if(arr[i]!=arr[i+1]) return arr[i];
    }
    return arr[i];
}
/*
    TC: O(N/2)
    SC: O(1)
    Works only for sorted array
*/
```

13) Longest Subarray with given sum k (positives)

```
/*
   Problem Statement: Given an array and a sum k, we need to print the length
of the longest subarray that sums to k.
*/
int brute_force_approach(int arr[], int n, int k){
   int mx=0;
   for(int i=0; i<n; i++){
      for(int t=i; t<n; t++ ){
         int sum = 0;
         for(int j=i; j<=t; j++){
            sum += arr[j];
         }
         if(sum==k){
            mx=max(mx,t-i+1);
         }
      }
   }
   return mx;
}
/*
   considering all the subarrays of array and then calculating the sum
   followed by comparing it with k and updating mx value
   TC: O(N^3)
   SC: O(1)
*/
int brute_force_approach_2(int arr[], int n, int k){
   int mx=0;
   for(int i=0; i<n; i++){
      int sum=0;
      for(int t=i; t<n; t++ ){
         sum+=arr[t];
         if(sum==k){
            mx=max(mx,t-i+1);
         }
      }
   }
   return mx;
}
/*
   considering all the subarrays of array and then calculating the sum
   followed by comparing it with k and updating mx value
```

```
    TC: O(N^2)
    SC: O(1)
*/
int better_approach(int arr[], int n, int k){
    map<int, int> m;
    int sum=0,mx=0;
    for(int i=0; i<n; i++){
        sum+=arr[i];
        if(sum==k) mx = i+1;
        int rem = sum-k;
        if(m.find(rem)!=m.end()){
            mx = max(mx,i-m[rem]);
        }
        if(m.find(sum)==m.end()) m[sum]=i;
    }
    return mx;
}
/*
    TC: O(N)
    SC: O(N)
*/
int optimal_approach(int arr[], int n, int k){
    int i=0,j=0, mx=0, sum=arr[0];
    while(j<n){
        while(i<j && sum>k){
            sum-=arr[i];
            i++;
        }
        if(sum==k){
            mx = max(mx,j-i+1);
        }
        j++;
        if(j<n) sum+=arr[j];
    }
    return mx;
}
/*
    The steps are as follows:
        First, we will take two pointers: left and right, initially pointing to the
        index 0.

        The sum is set to a[0] i.e. the first element initially.

        Now we will run a while loop until the right pointer crosses the last index
```

i.e. n-1.

Inside the loop, we will do the following:

We will use another while loop and it will run until the sum is lesser or equal to k.

Inside this second loop, from the sum, we will subtract the element that is pointed by the left pointer and increase the left pointer by 1.

After this loop gets completed, we will check if the sum is equal to k. If it is, we will compare the length of the current subarray i.e. (right-left+1) with the existing one and consider the maximum one.

Then we will move forward the right pointer by 1. If the right pointer is pointing to a valid index(< n) after the increment, we will add the element i.e. a[right] to the sum.

Finally, we will return the maximum length.

Unlike the above two approaches, this does not work if there are any -ve nums in array

Complexity Analysis

Time Complexity: O(2*N), where N = size of the given array.

Reason: The outer while loop i.e. the right pointer can move up to index n-1(the last index). Now, the inner while loop i.e. the left pointer can move up to the right pointer at most. So, every time the inner loop does not run for n times rather it can run for n times in total.

So, the time complexity will be O(2*N) instead of O(N2).

Space Complexity: O(1) as we are not using any extra space.

*/

14) Longest Subarray with given sum

Same as the first 3 approaches of previous ques

1) Two-Sum

```
/*
    Problem Statement: Given an array of integers arr[] and an integer
    target.

    1st variant: Return YES if there exist two numbers such that their sum is
    equal to the target. Otherwise, return NO.

    2nd variant: Return indices of the two numbers such that their sum is
    equal to the target. Otherwise, we will return {-1, -1}.

    Note: You are not allowed to use the same element twice. Example: If the
    target is equal to 6 and num[1] = 3, then nums[1] + nums[1] = target
    is not a solution.
*/



pair<int, int> brute_force(int arr[], int n, int k){
   for(int i=0; i<n-1; i++){
      int a = arr[i];
      for (int j = i + 1; j < n; j++) {
        if (a + arr[j] == k){
          return {i,j};
        }
      }
   }
   return {-1,-1};
}
/*
   TC: O(N^2)
   SC: O(1)
*/
pair<int, int> better_approach(int arr[], int n, int k){
   map<int, int> m;
   for(int i=0; i<n; i++){
      if(m.find(k-arr[i]) != m.end()) return {m[k-arr[i]],i};
      if(m.find(arr[i]) == m.end()) m[arr[i]] = i;
   }
   return {-1,-1};
}
```

```cpp
/*
   TC: O(NlogN) or O(N) depending of map structure used
   SC: O(N)
*/
pair<int, int> better_approach(int arr[], int n, int k){
   map<int, int> m;
   for(int i=0; i<n; i++){
      if(m.find(k-arr[i]) != m.end()) return {m[k-arr[i]],i};
      if(m.find(arr[i]) == m.end()) m[arr[i]] = i;
   }
   return {-1,-1};
}
/*
   TC: O(NlogN) or O(N) depending of map structure used
   SC: O(N)
*/
string optimized_approach(int arr[], int n, int k){
   unordered_map<int, int> m;
   for(int i=0; i<n; i++) m[arr[i]] = i;
   sort(arr,arr+ n);
   int i=0, j=n-1;
   while(i<j){
      if(arr[i]+arr[j]==k) return "YES";
      else if(arr[i]+arr[j]>k) j--;
      else i++;
   }
   return "NO";
}
/*
   TC: O(N) + O(N), in very rare and worst case O(N^2)
   SC: O(N)
   cannot return indices
*/
```

2) Sort_0_1_2s

```
/*
     Problem Statement: Given an array consisting of only 0s, 1s, and 2s.
     Write a program to in-place sort the array without using inbuilt sort
     functions. (Expected: Single pass-O(N) and constant space)
*/
void brute_force(int *arr, int n){
   // bubble, insertion, selection sort : O(N^2)
   // quick, merge sort : O(NlogN)
   sort(arr,arr+n);
}
void better_approach(int *arr, int n)
{
  int count[3]={0},k=0;
  for(int i=0; i<n; i++){
    count[arr[i]]++;
  }
  for(int i=0; i<=2; i++){
    while(count[i]--){
      arr[k]=i;
      k++;
    }
  }
}
/*
   TC: O(N) + O(N)
   SC: O(N)
*/
void optimal_approach(int *arr, int n)
{
  int i=0,j=0,k=n-1;
  while(j<=k){
    if(arr[j]==0){
      swap(arr[i],arr[j]);
      i++;
      j++;
    }
    else if(arr[j]==1){
      j++;
    }
    else{
      swap(arr[k],arr[j]);
```

```
            k--;
        }
    }
}
/*
    Dutch National Flag algo
    arr[0,i) contains all 0s
    arr[i,mid-1) contains all 1s
    arr[high+1,n-1] contains all 2s

    TC: O(N)
    SC: O(1)
*/
```

3) Majority Element > n/2 times

```
/*
    Problem Statement: Given an array of N integers, write a program to
    return an element that occurs more than N/2 times in the given array. You
    may consider that such an element always exists in the array.
*/

int brute_force(int *arr, int n){
    int th=n/2 + n%2, ct;
    for(int i=0; i<n-1; i++){
        ct=0;
        for(int j=i; j<n; j++){
            if(arr[i]==arr[j]) ct++;
            if(ct>=th) return arr[i];
        }
    }
    return arr[0];
}
/*
    TC: O(N^2)
    SC: O(1)
*/

int better_approach(int *arr, int n){
    int th=n/2 + n%2, ct;
    map<int,int> m;
    for (int i = 0; i < n; i++) {
        m[arr[i]]++;
        if(m[arr[i]]>=th) return arr[i];
    }
    return 0;
}
/*
     Time Complexity: O(N*logN), where N = size of the given array.
```

```
    Reason: We are using a map data structure. Insertion in the map takes logN
    time. And we are doing it for N elements. So, it results in the first term
    O(N*logN). If we use unordered_map instead, the first term will be O(N) for
    the best and average case and for the worst case, it will be O(N2).

    Space Complexity: O(N) as we are using a map data structure.
*/
int optimal_approach(int *arr, int n){
    int ele,ct=0;
    for(int i=0; i<n; i++){
    if (ct == 0) {
        ele = arr[i];
        ct = 1;
        }
    else if (ele == arr[i]) ct++;
    else ct--;
    }
    return ele;
}
/*
    Moore's Voting Algorithm
    TC: O(N)
    SC: O(1)
    watch YT for itiution
*/
```

4) Kadane's algo for Subarray with maximum sum

```
/*
    Problem Statement: Given an integer array arr, find the contiguous
    subarray (containing at least one number) which has the largest sum and
    returns its sum and prints the subarray.
*/
int brute_force(int *arr, int n){
    int s=0,e=0;
    long long mx_sm=0, sum;
    for(int i=0; i<n; i++){
        sum=0;
        for(int j=i; j<n; j++){
            sum+=arr[j];
            if(sum>mx_sm){
                s=i;
                e=j;
                mx_sm=sum;
            }
        }
    }
    return mx_sm;
}
/*
    TC: O(N^2)
    SC: O(1)
*/

int optimal_approach(int *arr, int n){
    int max_sum=0,sum=0,s=0,fs=0,e=0;
    for(int i=0; i<n; i++){
        if(sum==0){
            s=i;
        }
        sum+=arr[i];
```

```
            if(sum>max_sum){
                max_sum=sum;
                fs=s;
                e=i;
            }
            if(sum<0) sum=0;
        }
        cout<<"The Subarray with maximum sum is: "<<endl;
        for(int i=fs; i<=e; i++) cout<<arr[i]<<" ";
        cout<<endl;
        return max_sum;
    }
    /*
        Kadane's algo
        we start adding elements of array to sum and update the max_sum
        accordingly
        if at any point i, sum<0 it indicates that sum of whole arr[0..i] is neg and
        the sub array will either exist in the remaining part after i or that before i
        (if we don't find sum in anywhere from arr[i..n] > maxsum we have from
        arr[0..i])
        if there is max_subarray, it always starts at sum=0 so we note down that
        index as start
        if sum>max_sum we note down curr index as last, and start_idx as
        final_start_idx
        TC: O(N)
        SC: O(1)
    */
```

5) Stock buy and sell

```
/*
    Problem Statement: You are given an array of prices where prices[i] is
    the price of a given stock on an ith day.

    You want to maximize your profit by choosing a single day to buy one
    stock and choosing a different day in the future to sell that stock.

    Return the maximum profit you can achieve from this transaction.

    If you cannot achieve any profit, return 0.

*/
int brute_force(int arr[], int n){
    int mx_p=0, p;
    for(int i=0; i<n-1; i++){
        for(int j=i+1; j<n; j++){
            mx_p = max(mx_p,arr[j]-arr[i]);
        }
    }
    return mx_p;
}
/*
    TC: O(N^2)
    SC: O(1)
*/

int optimal_approach(int arr[], int n){
    int mn=arr[0], mx=0;
    for(int i=0; i<n; i++){
        if(arr[i]<=mn){
            mn = arr[i];
        }
        else{
```

```
            mx = max(mx,arr[i]-mn);
        }
    }
    return mx;
}
/*
    TC: O(N)
    SC: O(1)

    we keep the track of the minimum element if current element is
    greater than minimum element then we calculate the difference and
    update the max else we update the min element

*/
```

6) Arrange in alternating Pos and Neg elements

```
/*
    Variety-1
    Problem Statement:

    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.
*/

void brute_force(int *arr, int n){
   for(int i=0; i<n; i++){
      if(i%2==0 && arr[i]<0){
         int j=i;
         while(arr[j]<0 & j<n) j++;
         if(j<n){
            swap(arr[i],arr[j]);
            while (j - 1 > i) {
               swap(arr[j - 1], arr[j]);
               j--;
            }
         }
         else return;
      }
      else if(i%2!=0 && arr[i]>0){
         int j=i;
         while(arr[j]>0 & j<n) j++;
         if(j<n){
            swap(arr[i],arr[j]);
            while (j - 1 > i) {
```

```
                swap(arr[j - 1], arr[j]);
                j--;
            }
        }
        else return;
    }
  }
}
/*
   // can skip this one
   TC: O(N^3)
   SC: O(1)
*/
void better_approach(int *arr, int n){
   vector<int> pos, neg;
   for(int i=0; i<n; i++){
      if(arr[i]>=0) pos.push_back(arr[i]);
      else neg.push_back(arr[i]);
   }
   int j=0,k=0;
   for(int i=0; i<n/2; i++){
      arr[2*i] = pos[i];
      arr[2*i +1] = neg[i];
   }
}
/*
    Segregating pos and neg elements in different vectors and then placing
    them at correct place in final array.

    TC: O(N + N)
    SC: O(N)
*/
void optimal_approach(int *arr, int s){
   int a[s];
   for(int i=0; i<s; i++) a[i]=arr[i];
   int p=0,n=1;
```

```
    for(int i=0; i<s; i++){
        if(a[i]>=0){
            arr[p]=a[i];
            p+=2;
        }
        if (a[i] < 0) {
            arr[n] = a[i];
            n += 2;
        }
    }
}
/*
    Approach:

In this optimal approach, we will try to solve the problem in a single pass and try to arrange the array elements in the correct order in that pass only.

We know that the resultant array must start from a positive element so we initialize the positive index as 0 and negative index as 1 and start traversing the array such that whenever we see the first positive element, it occupies the space at 0 and then posIndex increases by 2 (alternate places).

Similarly, when we encounter the first negative element, it occupies the position at index 1, and then each time we find a negative number, we put it on the negIndex and it increments by 2.

When both the negIndex and posIndex exceed the size of the array, we see that the whole array is now rearranged alternatively according to the sign.

    TC: O(N)
    SC: O(N)

*/
//  the above 3 algos work if there are equal number of positive and negative element
```

```
// variety 2: unequal number of pos and neg elements

void optimal_approach_2(int *arr, int n){
    vector<int> pos, neg;
    for(int i=0; i<n; i++){
        if(arr[i]<0) neg.push_back(arr[i]);
        else pos.push_back(arr[i]);
    }
    if(pos.size()>=neg.size()){
        for(int i=0; i<neg.size(); i++){
            arr[2*i] = pos[i];
            arr[2*i+1] = neg[i];
        }
        int idx = 2*neg.size();
        for(int i=neg.size(); i<pos.size(); i++){
            arr[idx] = pos[i];
            idx++;
        }
    }
    else{
        for(int i=0; i<pos.size(); i++){
            arr[2*i] = pos[i];
            arr[2*i+1] = neg[i];
        }
        int idx = 2*pos.size();
        for(int i=pos.size(); i<neg.size(); i++){
            arr[idx] = neg[i];
            idx++;
        }
    }
}
/*
Time Complexity: O(2*N) { The worst case complexity is O(2*N) which is a
combination of O(N) of traversing the array to segregate into neg and pos
array and O(N) for adding the elements alternatively to the main array}.
```

Explanation: The second O(N) is a combination of O(min(pos, neg)) + O(leftover elements).

There can be two cases: when only positive or only negative elements are present, O(min(pos, neg)) + O(leftover) = O(0) + O(N), and when equal no. of positive and negative elements are present, O(min(pos, neg)) + O(leftover) = O(N/2) + O(0). So, from these two cases, we can say the worst-case time complexity is O(N) for the second part, and by adding the first part we get the total complexity of O(2*N).

Space Complexity: O(N/2 + N/2) = O(N) { N/2 space required for each of the positive and negative element arrays, where N = size of the array A}.
*/

7) Next Permutation

```
/*

Problem Statement: 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).
*/
void generate_permutations(vector<int> a, vector<bool> check,
vector<vector<int>>&b){
    for(int i=1; i<=check.size()-1; i++){
        if(!check[i]){
            check[i] = 1;
            a.push_back(i);
            if(a.size()==check.size()-1){
                b.push_back(a);
                return;
            }
            generate_permutations(a,check,b);
            check[i] = 0;
            a.pop_back();
        }
    }
    return;
}
/*
    TC: O(N*N!)
    SC: O(N*N! + N)
*/

void generate_permutations_using_swap(int p, vector<int> arr,
vector<vector<int>> &ans){
    if(p==arr.size()-1){
        ans.push_back(arr);
```

```
    }
    else{
        for(int i=p; i<arr.size(); i++){
            swap(arr[i],arr[p]);
            generate_permutations_using_swap(p+1,arr,ans);
            swap(arr[i],arr[p]);
        }
    }
}
/*
    TC: O(N*N!)
    SC: O(1)
*/

void using_inbuilt_func(vector<int> &arr, int n){
    next_permutation(arr.begin(),arr.begin()+n);
}

void optimal_approach(vector<int> &arr, int n){
    vector<bool> check(n+1,0);
    int mx=arr[n-1],i;
    for(i=n-1; i>=0; i--){
        check[arr[i]]=1;
        if(arr[i]>=mx){
            mx=arr[i];
        }
        else{
            int j=arr[i]+1;
            while(!check[j]) j++;
            arr[i] = j;
            check[j] = 0;
            j=0;
            while(j<=n){
                while(!check[j] && j<=n) j++;
                if(j==n+1) break;
                arr[++i] = j;
```

```
                check[j]=0;
            }
            return;
        }
    }
    for(int i=0; i<n; i++) arr[i] = i+1;
    return;
}
/*
```

1 3 2 5 4
next permutation = 1 3 4 2 5

we loop through the arr from the end and keep track of the maximum element

if at any point i, the current element is less than maximum element then it means that arr[0..i-1] does not need to be changed as we have elements greater than arr[i] to replace it

meanwhile we also maintain a boolean check array where if check[i] = 1 that means number i is available else locked

once we find that element which is less than the max element (basically we are finding element which is breaking the ascending order of elements in array when it is read backwards)

in given example

1 3 2 5 4
4 < 5 but 5 is not < 2 therefore 2 is our break point. check array at this point will be [0 1 0 1 1]

once we find the breakpoint we replace it with the element in arr[i+1..n-1] just greater than it.

in this case it is 4 and then we mark check[4] = 0

So 1 3 4 5 4 and our check array looks like [0 1 0 0 1]

initialize i to just after break point

now loop through check array, if you find check[j] = 1 than arr[i] = j and increment i & j;

final ans 1 3 4 2 5

another way to do this is to just reverse the arr[i+1..n-1] after swapping. in that case we don't need check array

now what we do is
    TC: O(N)
    SC: O(N)
*/

```
void optimal_approach_2(vector<int> &arr, int n){
    int idx = -1;
    for(int i=n-1; i>=1; i--){
        if(arr[i-1]<arr[i]){
            idx = i-1;
            break;
        }
    }
    if(idx==-1){
        reverse(arr.begin(),arr.end());
        return;
    }
    for(int i=n-1; i>idx; i--){
        if(arr[i]>arr[idx]){
            swap(arr[i],arr[idx]);
            break;
        }
```

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

finding the break point (the point that breaks ascending order from last element i.e. from right to left direction)

if no break point means it is the last permutation then reverse the array

if there is a break point then swap it with the first element from rhs which is greater

now reverse the ascending array i.e. array from [breakpoint+1,n-1]

    TC: O(N + N + N) = O(N)
    SC: O(1)
*/
```

8) Find Leaders in an array.

```
/*
Problem Statement: Given an array, print all the elements which are leaders.
A Leader is an element that is greater than all of the elements on its right side in the array.
*/

vector<int> superiorElements(vector<int>&arr) {
    int n = arr.size();
    vector<int> ans;
    for(int i=n-1; i>=0; i--){
        int j=i+1;
        while(arr[i]>arr[j] && j<n) j++;
        if(j==n) ans.push_back(arr[i]);
    }
    return ans;
}
/*
    TC: O(N^2)
    SC: O(N) used to store ans
*/
vector<int> superiorElements(vector<int>&arr) {
    int n = arr.size();
    int mx=arr[n-1];
    vector<int> ans;
    ans.push_back(mx);
    for(int i=n-2; i>=0; i--){
        if(arr[i]>mx){
            ans.push_back(arr[i]);
            mx = arr[i];
        }
    }
    return ans;
```

```
}
/*

We traverse from the end and keep track of max element

if at any point i, an element arr[i] > max, that means arr[i] is one of the leaders so we add it to leaders array and update the max element

    TC: O(N)
    SC: O(N) used to store ans
*/
```

9) Longest Consecutive Sequence in an array

```
/*

Problem Statement: You are given an array of 'N' integers. You need to find the length of the longest sequence which contains the consecutive elements.

Input:
    A = [5, 8, 3, 2, 1, 4], N = 6

Output:
    5

Explanation:
    The resultant sequence can be 1, 2, 3, 4, 5.
    The length of the sequence is 5.
*/
int brute_force_approach(vector<int>&arr) {
    int n = arr.size();
    int mx=0;
    for(int i=0; i<n; i++){
        int ct = 0;
        int f=1, x=arr[i];
        do{
            f=0;
            for(int j=0; j<n; j++){
                if(arr[j]==x){
                    ct++;
                    x++;
                    f=1;
                    break;
                }
            }
        }while(f==1);
        mx = max(mx,ct);
    }
```

```
    return mx;
}
/*

Algorithm:

To begin, we will utilize a loop to iterate through each element one by one.

Next, for every element x, we will try to find the consecutive elements like x+1, x+2, x+3, and so on using the linear search algorithm in the given array.

Within a loop, our objective is to locate the next consecutive element x+1.

If this element is found, we increment x by 1 and continue the search for x+2.

Furthermore, we increment the length of the current sequence (cnt) by 1.

This process repeats until step 2.2 occurs.

If a specific consecutive element, for example, x+i, is not found, we will halt the search for subsequent consecutive elements such as x+i+1, x+i+2, and so on. Instead, we will take into account the length of the current sequence (cnt).

Among all the lengths we get for all the given array elements, the maximum one will be the answer.

    TC: O(n^2)
    SC: O(1)
*/
int better_approach(vector<int>&arr) {
    int n = arr.size();
    sort(arr.begin(),arr.begin()+n);
    int i=0, j=1, mx=1, ct=1;
```

```
    while(j<n){
        while(arr[j]==arr[i] && j<n) j++;
        if(j==n) break;
        else if(arr[j]-arr[i]==1){
            ct++;
        }
        else{
            mx = max(mx,ct);
            ct=1;
        }
        i=j;
        j++;
    }
    mx = max(ct, mx);
    return mx;
}
/*
    TC: O(nlog(n) + n)
    SC: O(1)
*/
int optimal_approach(vector<int>&arr) {
    int n = arr.size(),mx=0,ct=0;
    unordered_set<int> s;
    for(auto it: arr) s.insert(it);
    for(auto it: s){
        if(s.find(it-1)==s.end()){
            int j=it,ct=1;
            while(s.find(j+1)!=s.end()){
                ct++;
                j++;
            }
            mx = max(ct,mx);
        }
    }
    return mx;
}
```

```
/*

Algorithm:
We will declare 2 variables,
'cnt' → (to store the length of the current sequence),
'longest' → (to store the maximum length).
First, we will put all the array elements into the set data structure.
For every element, x, that can be a starting number(i.e. x-1 does not exist
in the set) we will do the following:

    We will set the length of the current sequence(cnt) to 1.

    Then, again using the set, we will search for the consecutive elements
    such as x+1, x+2, and so on, and find the maximum possible length of
    the current sequence.

    This length will be stored in the variable 'cnt'.

    After that, we will compare 'cnt' and 'longest' and update the variable
    'longest' with the maximum value (i.e. longest = max(longest, cnt)).

    Finally, we will have the answer i.e. 'longest'.

   TC: O(N + 2N)
   SC: O(N)
*/
```

10) Set Matrix Zero

```
/*
Problem Statement: Given a matrix if an element in the matrix is 0 then you will have to set its entire column and row to 0 and then return the matrix.
*/
vector<vector<int>> brute_force_approach(vector<vector<int>> &matrix, int n, int m) {
    vector<vector<int>> mt(matrix);
    for(int i=0; i<n; i++){
        for(int j=0; j<m; j++){
            if(matrix[i][j]==0){
                for(int k=0; k<n; k++) mt[k][j]=0;
                for(int k=0; k<m; k++) mt[i][k]=0;
            }
        }
    }
    return mt;
}
/*
    TC: O(N*M*(N+M)) worst case
    SC: O(N*M)
*/
vector<vector<int>> brute_force_approach_2(vector<vector<int>> &matrix, int n, int m) {
    for(int i=0; i<n; i++){
        for(int j=0; j<m; j++){
            if(matrix[i][j]==0){
                for(int k=0; k<n; k++){
                    if(matrix[k][j]!=0) matrix[k][j]=-1;
                }
                for(int k=0; k<m; k++){
                    if(matrix[i][k]!=0) matrix[i][k]=-1;
                }
            }
```

```
        }
    }
    for(int i=0; i<n; i++){
        for(int j=0; j<m; j++){
            if(matrix[i][j]==-1) matrix[i][j] = 0;
        }
    }

    return matrix;
}
/*
    works only if matrix is assumed to be having only positive elements
    TC: O(N*M*(N+M)) worst case
    SC: O(1)
*/
vector<vector<int>> better_approach(vector<vector<int>> &matrix, int n, int
m) {
    int arr[n]={0},a[m]={0};
    for(int i=0; i<n; i++){
        for(int j=0; j<m; j++){
            if(matrix[i][j]==0){
                arr[i]=1;
                a[j]=1;
            }
        }
    }
    for(int i=0; i<n; i++){
        if(arr[i]){
            for(int j=0; j<m; j++) matrix[i][j]=0;
        }
    }
    for(int i=0; i<m; i++){
        if(a[i]){
            for(int j=0; j<n; j++) matrix[j][i]=0;
        }
    }
```

```
    return matrix;
}
/*
    TC: O(N*M + N*M + N*M)
    SC: O(N + M)
*/
vector<vector<int>> optimal_approach(vector<vector<int>> &matrix, int n,
int m) {
    int col=1;
    for(int i=0; i<n; i++){
        for(int j=0; j<m; j++){
            if(matrix[i][j]==0){
                if(j==0) col=0;
                else matrix[0][j]=0;
                matrix[i][0] = 0;
            }
        }
    }
    for(int i=1; i<n; i++){
        for(int j=1; j<m; j++){
            if(matrix[0][j]==0 || matrix[i][0]==0){
                matrix[i][j]=0;
            }
        }
    }
    if(matrix[0][0]==0){
        for(int i=0; i<m; i++){
            matrix[0][i]=0;
        }
    }
    if(col==0){
        for(int i=0; i<n; i++){
            matrix[i][0]=0;
        }
    }
    return matrix;
```

```
}
/*
    TC: O(N*M + N*M)
    SC: O(1)

First, we will traverse the matrix and mark the proper cells of 1st row and 1st column with 0 accordingly.

The marking will be like this: if cell(i, j) contains 0, we will mark the i-th row i.e. matrix[i][0] with 0 and we will mark j-th column i.e. matrix[0][j] with 0.

If i is 0, we will mark matrix[0][0] with 0 but if j is 0, we will mark the col variable with 0 instead of marking matrix[0][0] again.

After step 1 is completed, we will modify the cells from (1,1) to (n-1, m-1) using the values from the 1st row, 1st column, and col0 variable.

We will not modify the 1st row and 1st column of the matrix here as the modification of the rest of the matrix(i.e. From (1,1) to (n-1, m-1)) is dependent on that row and column.

Finally, we will change the 1st row and column using the values from matrix[0][0] and col0 variable.

Here also we will change the row first and then the column.

If matrix[0][0] = 0, we will change all the elements from the cell (0,1) to (0, m-1), to 0.

If col0 = 0, we will change all the elements from the cell (0,0) to (n-1, 0), to 0.
*/
```

11) Rotate Matrix by 90 degrees

```
/*
You are given an n x n 2D matrix representing an image, rotate the image
by 90 degrees (clockwise).

Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
Output: [[7,4,1],[8,5,2],[9,6,3]]
*/
void brute_force(vector<vector<int>>& matrix) {
    int n = matrix[0].size();
    vector<vector<int>> m(n,vector<int>(n,0));
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            m[j][n-i-1] = matrix[i][j];
        }
    }
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            matrix[i][j] = m[i][j];
        }
    }
}

/*
    Take first row, put it in last column of dummy matrix
    take second row, put it in 2nd last column and so on

    TC: O(N*N + N*N)
    SC: O(N*N)
*/
void optimal(vector<vector<int>>& matrix) {
    int n = matrix[0].size();
    for(int i=0; i<n; i++){
        for(int j=i+1; j<n; j++){
            swap(matrix[i][j],matrix[j][i]);
```

```
        }
    }
    for(int i=0; i<n; i++){
        int k=0, j = n-1;
        while(k<=j){
            swap(matrix[i][k],matrix[i][j]);
            k++;
            j--;
        }
    }
}

/*
    Take transpose of the matrix and reverse each row of the matrix
    TC: O(N*N/2 + N*N/2) = O(N*N)
    SC: O(1)
*/
```

12) Spiral Traversal of Matrix

```
/*

Problem Statement: Given a Matrix, print the given matrix in spiral order.
    Examples:
    Example 1:
    Input: Matrix[][] = { { 1, 2, 3, 4 },
                          { 5, 6, 7, 8 },
                          { 9, 10, 11, 12 },
                          { 13, 14, 15, 16 } }
    Outhput: 1, 2, 3, 4, 8, 12, 16, 15, 14, 13, 9, 5, 6, 7, 11, 10.
    Explanation: The output of matrix in spiral form.
    Example 2:
    Input: Matrix[][] = { { 1, 2, 3 },
                          { 4, 5, 6 },
                          { 7, 8, 9 } }

    Output: 1, 2, 3, 6, 9, 8, 7, 4, 5.
    Explanation: The output of matrix in spiral form.
*/

vector<int> spiralMatrix(vector<vector<int>>&m) {
    int mt=m[0].size(), n=m.size();
    int t = 0, b = n-1, l = 0, r=mt-1, ct=0;
    vector<int> ans;
    while(t<=b && l<=r){
        if(ct%4==0){
            for(int j=l; j<=r; j++){
                ans.push_back(m[t][j]);
            }
            t++;
        }
        else if(ct%4==1){
            for(int i=t; i<=b; i++){
                ans.push_back(m[i][r]);
```

```
            }
            r--;
        }
        else if(ct%4==2){
            for(int j=r; j>=l; j--){
                ans.push_back(m[b][j]);
            }
            b--;
        }
        else if(ct%4==3){
            for(int i=b; i>=t; i--){
                ans.push_back(m[i][l]);
            }
            l++;
        }
        ct++;
    }
    return ans;
}
/*
```

Create and initialize variables top as starting row index, bottom as ending row index left as starting column index, and right as ending column index.

In each outer loop traversal print the elements of a square in a clockwise manner.

Print the top row, i.e. Print the elements of the top row from column index left to right and increase the count of the top so that it will move to the next row.

Print the right column, i.e. Print the rightmost column from row index top to bottom and decrease the count of right.

Print the bottom row, i.e. if top <= bottom, then print the elements of a bottom row from column right to left and decrease the count of bottom

```
Print the left column, i.e. if left <= right, then print the elements of the left column from the bottom row to the top row and increase the count of left.

Run a loop until all the squares of loops are printed.

    TC: O(N*M)
    SC: O(N*M)
*/
```

13) Count Subarray Sum equals K

```
/*

Problem Statement: Given an array of integers and an integer k, return the total number of subarrays whose sum equals k.

A subarray is a contiguous non-empty sequence of elements within an array.
*/
int brute_force_approach(vector<int>& nums, int k){
    int n = nums.size(), ct=0;
    for(int i=0; i<n; i++){
        for(int j=i; j<n; j++){
            int sum = 0;
            for(int t=i; t<=j; t++){
                sum+=nums[t];
            }
            if(sum==k)
                ct++;
        }
    }
    return ct;
}
/*
    TC: O(N^3)
    SC: O(1)
*/

int better_approach(vector<int>& nums, int k) {
    int n = nums.size(), ct=0;
    for(int i=0; i<n; i++){
        int sum = 0;
        for(int j=i; j<n; j++){
            sum+=nums[j];
            if(sum==k)
                ct++;
```

```
        }
    }
    return ct;
}
/*
    TC: O(N^2)
    SC: O(1)
*/
int optimal_approach(vector<int>& nums, int k) {
    int n = nums.size(), ct=0, sum=0;
    map<int,int> m;
    for(int i=0; i<n; i++){
        sum+=nums[i];
        if(sum==k) ct++;
        int rem = sum-k;
        if(m.find(rem)!=m.end()){
            ct+=m[rem];
        }
        if(m.find(sum)==m.end()) m[sum]=1;
        else m[sum]++;
    }
    return ct;
}
/*

Check longest subarray with sum K problem in array easy module to
understand the logic.
    TC: O(N*logN)
    SC: O(N)
*/
```

8) Merge two sorted arrays without using extra space.

```
/*
    Problem statement: Given two sorted arrays arr1[] and arr2[] of sizes n
    and m in non-decreasing order.

    Merge them in sorted order. Modify arr1 so that it contains the first N
    elements and modify arr2 so that it contains the last M elements.

   Example 1:
   Input:

   n = 4, arr1[] = [1 4 8 10]
   m = 5, arr2[] = [2 3 9]
   Output:

   arr1[] = [1 2 3 4]
   arr2[] = [8 9 10]
   Explanation:
   After merging the two non-decreasing arrays, we get, 1,2,3,4,8,9,10.
   Example2:
   Input:

   n = 4, arr1[] = [1 3 5 7]
   m = 5, arr2[] = [0 2 6 8 9]
   Output:

   arr1[] = [0 1 2 3]
   arr2[] = [5 6 7 8 9]
   Explanation:
   After merging the two non-decreasing arrays, we get, 0 1 2 3 5 6 7 8 9.
*/
// Brute force approach is same as that of merge algorithm of merge sort but
it uses extra space

// Leetcode variant:
```

```
void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
    if(m==0){
        nums1.clear();
        nums1.insert(nums1.begin(),nums2.begin(),nums2.end());
        nums2.clear();
        return;
    }
    else if(n==0) return;
    // the above two conditions are of edge cases
    else{
        int i=m-1, j=0;
        while(i>=0 && j<n){
            if(nums1[i]>=nums2[j]) swap(nums1[i],nums2[j]);
            else break;
            i--;
            j++;
        }
        sort(nums1.begin(),nums1.begin()+m);
        sort(nums2.begin(),nums2.end());
        for(int k=0;k<n;k++){
            nums1[m+k] = nums2[k];
        }
    }
}
/*
```

The steps are as follows:

We will declare two pointers i.e. left and right.

The left pointer will point to the last index of the arr1[](i.e. Basically the maximum element of the array).

The right pointer will point to the first index of the arr2[](i.e. Basically the minimum element of the array).

Now, the left pointer will move toward index 0 and the right pointer will move towards the index m-1.

While moving the two pointers we will face 2 different cases like the following:

If arr1[left] > arr2[right]: In this case, we will swap the elements and move the pointers to the next positions.

If arr1[left] <= arr2[right]: In this case, we will stop moving the pointers as arr1[] and arr2[] are containing correct elements.

Thus, after step 2, arr1[] will contain all smaller elements and arr2[] will contain all bigger elements.

Finally, we will sort the two arrays.

Time Complexity: O(min(n, m)) + O(n*logn) + O(m*logm), where n and m are the sizes of the given arrays.

Reason: O(min(n, m)) is for swapping the array elements. And O(n*logn) and O(m*logm) are for sorting the two arrays.

Space Complexity: O(1) as we are not using any extra space.
*/

```
void compare_and_swap(int left, int right, vector<int>& arr1, vector<int>&
arr2){
    if(arr1[left]>arr2[right]) swap(arr1[left],arr2[right]);
    else return;
}
void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
    if(m==0){
        nums1.clear();
        nums1.insert(nums1.begin(),nums2.begin(),nums2.end());
        nums2.clear();
        return;
    }
```

```
    else if(n==0) return;
    else{
        int len=n+m;
        int gap=len/2 + len%2;
        while(gap>0){
            int left=0, right=left+gap;
            while(right<len){
                if(left<m && right<m){
                    compare_and_swap(left,right,nums1,nums1);
                }
                else if(left<m && right>=m){
                    compare_and_swap(left,right-m,nums1,nums2);
                }
                else if(left>=m && right>=m){
                    compare_and_swap(left-m,right-m,nums2,nums2);
                }
                left++;
                right++;
            }
            if (gap==1) break;
            gap = gap/2 + gap%2;
        }
        for(int i=0; i<n; i++){
            nums1[m+i] = nums2[i];
        }
    }
}
/*
```

Algorithm / Intuition

Optimal Approach 2 (Using gap method):

This gap method is based on a sorting technique called shell sort. The intuition of this method is simple.

Intuition:

Similar to optimal approach 1, in this approach, we will use two pointers i.e. left and right, and swap the elements if the element at the left pointer is greater than the element at the right pointer.

But the placing of the pointers will be based on the gap value calculated. The formula to calculate the initial gap is the following:

Initial gap = ceil((size of arr1[] + size of arr2[]) / 2)

Assume the two arrays as a single continuous array and initially, we will place the left pointer at the first index and the right pointer at the (left+gap) index of that continuous array.

Now, we will compare the elements at the left and right pointers and move them by 1 place each time after comparison.

While comparing we will swap the elements if the element at the left pointer > the element at the right pointer.

After some steps, the right pointer will reach the end and the iteration will be stopped.

After each iteration, we will decrease the gap and will follow the same procedure until the

iteration for gap = 1 gets completed. Now, after each iteration, the gap will be the following:

gap = ceil(previous gap / 2)

The whole process will be applied to the imaginary continuous array constructed using arr1[] and arr2[].

The reason we are using right-m, left-m etc in code is that when right = left + gap is the index of imaginary continuous array so if the right or left pointer lies in the second array then we have to reduce length of 1st array from that

in order to access elements in 2nd array.

Complexity Analysis

Time Complexity: O((n+m)*log(n+m)), where n and m are the sizes of the given arrays.

Reason: The gap is ranging from n+m to 1 and every time the gap gets divided by 2. So, the time complexity of the outer loop will be O(log(n+m)). Now, for each value of the gap, the inner loop can at most run for (n+m) times. So, the time complexity of the inner loop will be O(n+m). So, the overall time complexity will be O((n+m)*log(n+m)).

Space Complexity: O(1) as we are not using any extra space.
*/

9) Find repeating and missing numbers

```
/*

Problem Statement: You are given a read-only array of N integers with values also in the range [1, N] both inclusive. Each integer appears exactly once except A which appears twice and B which is missing. The task is to find the repeating and missing numbers A and B where A repeats twice and B is missing.
*/
vector<int> brute_force(vector<int> arr, int n) {
    int rep=0,mis=0;
    for(int i=1; i<=n; i++){
        int ct=0;
        for(int j=0; j<n; j++){
            if(arr[j]==i) ct++;
        }
        if(ct==2) rep=i;
        else if (ct==0) mis=i;
        if(rep && mis) return {rep,mis};
    }
}
/*
    TC: O(N^2)
    SC: O(1)
*/
vector<int> better_approach(vector<int> arr, int n) {
    int rep=0,mis=0;
    int hash[n+1] = {0};
    for(int i=0; i<n; i++){
        hash[arr[i]]++;
    }
    for(int i=1; i<=n; i++){
        if(hash[i]==2) rep=i;
        else if(hash[i]==0) mis=i;
        if(rep && mis) return {rep,mis};
```

```
    }
}
/*
    TC: O(N) + O(N)
    SC: O(N)
*/
vector<int> optimal_approach_1(vector<int> arr, int n) {
    long long N = (long long)n;
    long long SN = (N*(N+1))/2, S2N=(N*(N+1)*(2*N+1))/6;
    long long S=0,S2=0;
    for(int i=0; i<n; i++){
        S += (long long)arr[i];
        S2 += ((long long)arr[i]*(long long)arr[i]);
    }
    long long rep = S2-S2N;
    rep /= (S-SN);
    rep += (S-SN);
    rep /= 2;
    long long mis = rep - (S-SN);
    return {(int)rep,(int)mis};
}
/*
```

Using Mathematics

Sum of 1 to N = N(N+1)/2, Sum of squares from 1 to N = N(N+1)(2N+1)/6

Actual sum of all elements = Val1 , Actual sum of squares of all elements = Val2

if we subtract sum of 1 to N from actual sum, all the elements get cancelled on LHS except for repeating and missing element.

Therefore we get expression: rep - mis = Actual Sum - Sum 1 to N

Similarly we get expression: rep^2 - mis^2 = Actual squared sum - Sum

squared 1 to N

rep^2-mis^2 = (rep-mis)(rep+mis), so we can substitute the value of (rep-mis) in the equation to get (rep+mis)

now since we have two variables and two equations we can solve them to get values of rep and mis.

TC: O(N)

SC: O(1)

*/

```
vector<int> optimal_approach_2(vector<int> arr, int n) {
    int xr=0;
    for(int i=0; i<n; i++) {
        xr^=arr[i];
        xr^=(i+1);
    }
    int a=1;
    while((a&xr) == 0){
        a=a<<1;
    }
    int zero=0,one=0;
    for(int i=0; i<n; i++){
        if(arr[i]&a) one^=arr[i];
        else zero^=arr[i];
        if((i+1)&a) one^=(i+1);
        else zero^=(i+1);
    }
    for(int i=0; i<n; i++) if(arr[i]==zero) return{zero,one};
    return {one,zero};
}
/*
```

We xor all the elements in array and all the numbers from 1 to N.

Now xr will be (repeating number ^ missing number) lets say we get 5 which in binary will be 1 1 0

now there is 1 at 2nd position from right in xor, that means the bit in 2nd postion of both repeating and missing number will be different.

So now from a pool of all the array elements and all elements from 1 to N, we will put a number having bit set (or 1) at 2nd position in a group called "One" and if not then in group called "Zero".

Total number of occurence of repeating number and missing number in the entire pool will be 3,1 repectively and remaining will be 2.

So if we take xor of each group, it will give us repeating number and missing number

Note: we only need to find the first set bit from right for segregation.

For example:

2 3 1 3 1 2 3 4
xor = 3^4 = 7 i.e. 1 1 1

the first set bit from right is at position 1

so group one (with all the elements with set bit at position 1) and group zero with rest will be:

One : 3^1^3^1^3 = 3
Zero: 2^2^4 = 4

now we will take any value let say zero and will check if it exists in array or not. If it exists then it is repeating number and one is storing the missing number else otherwise

TC: O(n)+O(n)+O(n)+O(n) = O(n)
SC: O(1)

```
*/
```