1. **Remove Element**
   When $nums[j]$ equals to the given value, skip this element by incrementing j. As long as $nums[j]$!= val, we copy $nums[j]$ to $nums[i]$ and increment both indexes at the same time. Repeat the process until j reaches the end of the array and the new length is i.

2. **Remove Duplicates from Sorted Array:** Keep two pointers
3. **Given an array of strings, group anagrams together:**

   Maintain a map string -> list where key is the sorted string of all the words

4. **Maximum Product of Word Lengths:**

   Check which letters appear in each word using Bit Manipulation as follows:

   For each word initiate a value with 0 and do:

   Val |= 1 << (words[i][j] - ‘a’)

   where words[i][j] is a letter of a word.

   Then iterate over the ‘val’ array and check if val[i] & val[j] == 0 which means that they have no words in common and calculate the max product of length for such words.

5. Merge Intervals

Sort the given vector of vectors. Check whether the intervals overlap as follows:

- Set minVal and maxVal to start and end of first interval.
- Start iterating from the second element and check:
    - **if(intervals[i][0] <= maxVal)** which means they are overlapping and update the maxVal to max(maxVal, end of this interval).
    - **Else** there is no overlap between the intervals and hence add minVal and maxVal to temp vector and push it to result.
    - Reinitialise minVal and maxVal to current interval’s start and end.
- Add minVal and maxVal to temp vector and push it to result.

6. Search a 2D Matrix:

- **Integers in each row are sorted from left to right.**

- **The first integer of each row is greater than the last integer of the previous row.**

Approach 1: Apply binary search on each row (NlogN)

Approach 2: Treat 2D array as 1D array only

```
int low = 0;
int high = m * n - 1;


while(low != high)
{
    int mid = (low + high - 1) / 2;
    if(matrix[mid / m][mid % m] < target)
    {
        low = mid + 1;
    }
    else
    {
        high = mid;
    }
}
return matrix[high / m][high % m] == target;
```

7. Search a 2D Matrix II

- **Integers in each row are sorted in ascending from left to right.**
- **Integers in each column are sorted in ascending from top to bottom.**

```
int low = 0;
 int high = matrix[0].size() - 1;

 while(high >= 0 && low < matrix.size())
 {
     if(matrix[low][high] == target)
     {
         return true;
     }
     if(matrix[low][high] < target)
     {
         low++;
     }
     else
     {
         high--;
     }
 }
 return false;
```

8. Unique Path II(with obstacles):

```
// Number of ways of reaching the starting cell = 1.

result[0][0] = 1

 // Filling the values for the first column

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

 {

     result[0][i] = (obstacleGrid[0][i] == 0 && result[0][i - 1] == 1) ? 1 : 0;

 }

 // Filling the values for the first row

 for(int i = 1; i < m; i++)

 {

     result[i][0] = (obstacleGrid[i][0] == 0 && result[i - 1][0] == 1) ? 1 : 0;

 }


 for(int i = 1; i < m; i++)

 {

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

     {

         if(obstacleGrid[i][j] == 0)

         {

             result[i][j] = result[i - 1][j] + result[i][j - 1]; //Without obstacle same DP

         }

         else

         {
             result[i][j] = 0;
         }
     }
 }

 return result[m - 1][n - 1];
```

9. Increasing Triplet Subsequence:

```
  int n1 = INT_MAX, n2 = INT_MAX;

    for(int i = 0; i < nums.size(); i++)
    {
        if(nums[i] <= n1)
        {
            n1 = nums[i];
        }
        else if(nums[i] <= n2)
        {
            n2 = nums[i];
        }
        else
        {
            return true;
        }
    }
    return false;
```

10. **Largest Number**

1. Convert each number to string.
2. Then sort using a comparator which compares by placing one number after the other:

```
int mycomp(string a, string b)
{
    string ab = a.append(b);
    string ba = b.append(a);

    return ab.compare(ba) > 0 ? 1 : 0;
}
```

11. **Find Duplicate in Array**

<u>SLOW AND FAST POINTER</u>

```
int slow_ptr = nums[0];
    int fast_ptr = nums[nums[0]];

    while(fast_ptr != slow_ptr)
    {
        slow_ptr = nums[slow_ptr];
        fast_ptr = nums[nums[fast_ptr]];

    }


    fast_ptr = 0;


    while(slow_ptr != fast_ptr)

    {

        fast_ptr = nums[fast_ptr];

        slow_ptr = nums[slow_ptr];

    }


    return slow_ptr;
```

12. Palindrome Number

We can revert the last part of the number "1221" from "21" to "12", and compare it with the first half of the number "12", since 12 is the same as 12, we know that the number is a palindrome.

```
if(x < 0 || (x % 10 == 0 && x != 0))

        {

            return false;

        }


        int result = 0;
```

```
while(x > result)

{

    result = result * 10 + (x % 10);

    x /= 10;

}


return result == x || x == (result / 10);
```

13. **Rearrange an array so that arr[i] becomes arr[arr[i]] with O(1) extra space**

    1) Increase every array element arr[i] by (arr[arr[i]] % n)*n.

    2) Divide every element by n.

    After the increment operation of first step, every element holds both old values and new values.

    Old value can be obtained by arr[i]%n and new value can be obtained by arr[i]/n.

14. Repeat and Missing Number Array

1. Traverse the array. While traversing, use the absolute value of every element as an index and **make the value at this index as negative** to mark it visited. If something is **already marked negative then this is the repeating element**. To find **missing, traverse the array again and look for a positive value.**
2. `Sum(Actual) = Sum(1...N) + A - B`

   `Sum(Actual) - Sum(1...N) = A - B.`

   `Sum(Actual Squares) = Sum(1^2 ... N^2) + A^2 - B^2`

   `Sum(Actual Squares) - Sum(1^2 ... N^2) = (A - B)(A + B)`

```
= (Sum(Actual) - Sum(1...N)) ( A + B)
```

15. Flip: You are given a binary string(*i.e.* with characters `0` and `1`) S consisting of characters S1, S2, …, SN. In a single operation, you can choose two indices L and R such that 1 ≤ L ≤ R ≤ N and flip the characters SL, SL+1, …, SR. By flipping, we mean change character `0` to `1` and vice-versa.

Your aim is to perform ATMOST one operation such that in final string number of `1`s is maximised.

__**Steps**__:

1. Convert 1's to '-1' and 0's to '1'.
2. Apply **Kadane's Theorem.**

16. First Missing Positive: Given an unsorted integer array, find the smallest missing positive integer.

1. Put each number in its right place.

   For example:

   When we find 5, then swap it with A[4].

2. At last, the first place where its number is not right, return the place + 1.

```
int n = A.size();

        int result;



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

        {
            if(A[i] > 0 && A[i] < n && A[i] != A[A[i] - 1])
            {
                swap(A[i], A[A[i] - 1]);
            }
        }

        for(int i = 0; i < n; i++)
        {
            if(A[i] != i + 1)
            {
                result = i + 1;
                break;
            }
        }

        return result;
    }
```

17. Modular Exponentiation (Power in Modular Arithmetic):

```
int res = 1;

x = x % p;
```

```
while(n > 0)

{

    if(n & 1)

        res = (res * x) % p;

    n = n >> 1;

    x = (x * x) % p;

}
```

18. Permutations

SWAP LOGIC:

```
        void helper(vector<int> &nums, int left, int right, vector<vector<int>>
&result)

      {

        if(left >= right)

        {

            result.push_back(nums);

            return;

        }
```

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

    {

        swap(nums[i], nums[left]);

        helper(nums, left + 1, right, result);

        swap(nums[i], nums[left]);

    }

}
```

17. Minimum Characters required to make a String Palindromic

Each index of LPS array contains the longest prefix which is also a suffix. Now **take the string and reverse of a string and combine them with a sentinal character in between them** and **compute the LPS array of this combined string**. Now **take the last value of the LPS array and subtract it with the length of the original string**, This will give us the minimum number of insertions required in the begining of the string .

18. Roman To Integer

- Iterate from end of string and check whether ith element is less than (i + 1)th element:
    - If true, subtract ith element from result
    - Else add it.

```
unordered_map<char, int> romanMap = {{'I', 1}, {'V', 5}, {'X', 10}, {'L', 50}, {'C', 100}, {'D',
500}, {'M', 1000}};


  int n = A.size();


  int result = romanMap[A[n - 1]];


  for(int i = n - 2; i >= 0; i–)
  {
    if(romanMap[A[i]] < romanMap[A[i + 1]])
    {
      result -= romanMap[A[i]];
    }
    else
    {
      result += romanMap[A[i]];
    }
  }
```

```
    return result;
```

19. Zigzag String

Iterate through STRING from left to right, appending each character to the appropriate row. The appropriate row can be tracked using two variables: the current row and the **current direction**.

The current direction changes only when we moved up to the topmost row or moved down to the bottommost row.

```
if(B == 1)
    {
        return A;
    }


    vector<string> temp(min(B, n));

    bool changeDir = false;

    string result = "";


    int i = 0, j = 0, cnt = 0;


    while(cnt < n)
    {
        temp[i] += A[cnt];
        if(i == 0 || i == B - 1)

        {

            changeDir = !changeDir;

        }



        if(changeDir)

        {

            i++;

        }

        else

        {

            i--;

        }



        cnt++;

    }
```

20. Triplets with Sum between given range

- Intialize three variables a,b and c as first 3 values of array.
- Now,iterate from i = 3 to n and check the following:

- Check if sum falls in (1, 2),if it does then return true.
- If not, then check if sum is greater than 2,if so, then replace MAX(a,b,c) to current element arr[i].
- Otherwise sum must be less than 1 then replace MIN(a,b,c) to current element arr[i].
- And finally after coming out of loop check once again for last triplet if sum falls in (1,2) then return true,otherwise return false.

21. Number of 1 Bits

Subtraction of 1 from a number toggles all the bits (from right to left) till the rightmost set bit(including the rightmost set bit). So if we subtract a number by 1 and do bitwise & with itself (n & (n-1)), we unset the rightmost set bit. If we do n & (n-1) in a loop and count the no of times loop executes we get the set bit count.

```
int result = 0;

 while(A)

 {

    A &= (A - 1);

    result++;

 }

 return result;
```

22. Single Number II

Let us look at every bit position.

Every number that occurs thrice will either contribute 3 ‘1’s or 3 ‘0’s to the position.

The number that occurs once X will contribute exactly one 0 or 1 to the position depending on whether it has 0 or 1 in that position.

So:

- If X has 1 in that position, we will have (3x+1) number of 1s in that position.
- If X has 0 in that position, we will have (3x+1) number of 0s in that position.

```
for(int i = 31; i >= 0; i--)

 {

        int temp = 1 << i;

        int count = 0;



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

        {
            if(temp & A[j])

            {

                count++;

            }

        }



        if(count % 3 != 0)

        {

            result |= (1 << i);

        }

    }

return result;
```

23. Different Bits Sum Pairwise

- The idea is to count differences at individual bit positions.
    - We traverse from 0 to 31 and count numbers with i’th bit set. Let this count be ‘count’. There would be “**n-count**” numbers **with i’th bit not set**.
    - So **count of differences at i’th bit** would be “**count * (n-count) * 2**”, the reason for this formula is as every pair having one element which has set bit at i’th position and second element having unset bit at i’th position contributes exactly 1 to sum, therefore total permutation count will be count*(n-count) and multiply by 2 is due to one more repetition of all this type of pair as per given condition for making pair 1<=i,j<=N.

    CODE:

```
for(int i = 31; i >= 0; i--)

  {

     int temp = 1 << i;

     long long count = 0;



     for(int j = 0; j < n; j++)
```

```
    {
        if(temp & A[j])
        {
            count++;
        }
    }


    result += count * (n - count) * 2;
}
return result;
```

24. **Container With Most Water:** Given n non-negative integers a1, a2, ..., an, where each represents a point at coordinate (i, ai). 'n' vertical lines are drawn such that the two endpoints of line i is at (i, ai) and (i, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water

Take 2 pointers(low and high), one at the beginning one at the end. if(A[low] < A[high]) then low++, else high-- and keep calculating area.

```
if(n == 1)

   {

       return 0;

   }



   int low = 0, high = n - 1;

   int maxArea = INT_MIN;



   while(low < high)
```

```
{
    int area = min(A[low], A[high]) * (high - low);
    if(area > maxArea)
    {
        maxArea = area;
    }
    if(A[low] < A[high])
    {
        low++;
    }
    else
    {
        high--;
    }

}

return maxArea;
```

25. **Divide Integers:** Divide two integers without using multiplication, division and mod operator.

Suppose dividend = 15 and divisor = 3, 15 - 3 > 0. We now try to subtract more by *shifting* 3 to the left by 1 bit (6). Since 15 - 6 > 0, shift 6 again to 12. Now 15 - 12 > 0, shift 12 again to 24, which is larger than 15. So we can at most subtract 12 from 15. Since 12 is obtained by shifting 3 to left twice, it is 1 << 2 = 4 times of 3. We add 4 to an answer variable (initialized to be 0). The above process is like 15 = 3 * 4 + 3. We now get part of the quotient (4), with a remaining dividend 3.

Then we repeat the above process by subtracting divisor = 3 from the remaining dividend = 3 and obtain 0. We are done. In this case, no shift happens. We simply add 1 << 0 = 1 to the answer variable.

```
if(A == INT_MIN && B == -1)

    {

        return INT_MAX;

    }
    int temp;

    int result;



    int sign = A > 0 ^ B > 0 ? -1 : 1;

    int

    while(A >= B)

    {

        int count = 0;

        temp = B;

        while(temp <= A)

        {

            temp = temp << 1;
            count++;

        }



        A -= (temp >> 1);

        result += (1 << (count - 1));

    }



    return result;
```

26. **Max Continuous Series of 1s**

The main steps are:

– While zeroCount is no more than m: expand the window to the right (wR++) and update the count zeroCount.

– While zeroCount exceeds m, shrink the window from left (wL++), update zeroCount;

– Update the widest window along the way. The positions of output zeros are inside the best window.

```
int left = 0, right = 0, window = 0, start = 0, count = 0;



    vector<int> result;



    while(right < n)

    {

        if(count <= B)

        {

            if(A[right] == 0)

            {

                count++;

            }
            right++;

        }

if(count > B)

        {

            if(A[left] == 0)

            {

                count--;

            }

            left++;

        }

        if((right - left > window) && count <= B)

        {

            window = right - left;
            start = left;

        }

    }



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

    {

        result.push_back(start + i);

    }
```

27. **Remove Duplicates from Sorted Array II:** Given a sorted array *nums*, remove the duplicates in-place such that duplicates appeared at most *twice* and return the new length.

```
int removeDuplicates(vector<int>& nums) {

        int count = 1;

        int i = 0;
```

```
if(nums.size() <= 2)

{

    return nums.size();

}



for(int j = 1; j < nums.size(); j++)

{

    if(nums[j] == nums[i])

    {

        count++;

        if(count <= 2)

        {

            i++;

            nums[i] = nums[j];
            }
        }
        else
        {
            i++;
            nums[i] = nums[j];
            count = 1;
        }
    }

    return i + 1;
}
```

28. **Diffk: Given an array ‘A’ of sorted integers and another non negative integer k, find if there exists 2 indices i and j such that A[i] - A[j] = k, i != j.**

```
int low = 0, high = 1;



   while(high < n)

   {

      if(A[high] - A[low] == B)

      {

         return 1;

      }

      if(A[high] - A[low] > B)

      {

         low++;

         if(low == high)
         {
            high++;
         }
      }
      else
      {
         high++;
      }
   }
```

29. **Counting Triangles**: You are given an array of **N** non-negative integers, **A0**, **A1** ,…, **AN-1**.

Considering each array element **Ai** as the edge length of some line segment, count the number of triangles which you can form using these array values.

For a triangle to be possible from 3 values, the sum of any two values (or sides) must be greater than the third value (or third side).

- Sort the array in non-decreasing order.
- Initialize two pointers ‘i’ and ‘j’ to first and second elements respectively, and initialize count of triangles as 0.
- Fix ‘i’ and ‘j’ and find the rightmost index ‘k’ (or largest ‘arr[k]’) such that ‘arr[i] + arr[j] > arr[k]’. The number of triangles that can be formed with ‘arr[i]’ and ‘arr[j]’ as two sides is ‘k – j’. Add ‘k – j’ to count of triangles.
- Increment ‘j’ to fix the second element again.
- If ‘j’ has reached end, then increment ‘i’. Initialize ‘j’ as ‘i + 1’, ‘k’ as ‘i+2’ and repeat the steps 3 and 4.

```
sort(A.begin(), A.end());
    int result = 0;


    for(int i = 0; i < n - 2; i++)
    {
        int k = i + 2;


        for(int j = i + 1; j < n; j++)
        {
            while(k < n && A[i] + A[j] > A[k])
            {
```

```
            k++;

        }



        if(k > j)

        {

            result += (k - j - 1);

        }

    }

}



return result;
```

30. **Add Binary Strings**: Given two binary strings, return their sum (also a binary string).

```
string ans = "";

    int ansLen = 0, carry = 0, sum;

    int i = (int)a.length() - 1, j = (int)b.length() - 1;

    while (i >= 0 || j >= 0 || carry) {
          sum = carry;
          if (i >= 0) sum += (a[i] - '0');
          if (j >= 0) sum += (b[j] - '0');
          ans.push_back((char)('0' + sum % 2));
          carry = sum / 2;
          i--;
          j--;
    }
    reverse(ans.begin(), ans.end());
    return ans;
```

31. **Max Distance:** Given an array A of integers, find the maximum of `j - i` subjected to the constraint of `A[i] <= A[j]`.

If there is no solution possible, return `-1`.

```
vector<pair<int, int>> temp;

   int n = A.size();
```

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

{

    temp.push_back(make_pair(A[i], i));

}



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



int result = 0;

int pos = temp[n - 1].second;



for(int i = n - 2; i >= 0; i--)

{

    int t = pos - temp[i].second;

    // cout << t << endl;

    result = max(result, t);

    pos = max(temp[i].second, pos);

}
```

```
    return result;
```

32. **List Cycle**

Step1: Proceed in the usual way, you will use to find the loop, i.e. Have two pointers, increment one in single step and other in two steps, If they both meet in sometime, there is a loop.

Step2: Freeze one pointer where it was and increment the other pointer in one step counting the steps you make and when they both meet again, the count will give you the length of the loop (this is same as counting the number of elements in a circular link list).

Step3: Reset both pointers to the start of the link list, increment one pointer to the length of loop times and then start the second pointer. increment both pointers in one step and when they meet again, it will be the start of the loop (this is same as finding the n^{th} element from the end of the link list).

33.**Remove Nth Node from List End**

Take 2 pointers and increment one pointer ‘n’ times and then start the second pointer. Increment both pointers in one step until second pointer reaches the end.

```
ListNode *ptr1, *ptr2, *prev;

    if(A == NULL)
    {
        return NULL;
    }

    prev = NULL;
    ptr1 = A;
    ptr2 = A;

    while(ptr2 != NULL && B != 0)
    {
        ptr2 = ptr2->next;
        B--;
    }

    if(B > 0)
```

```
{
    A = ptr1->next;
    return A;
}

while(ptr2 != NULL)
{
    prev = ptr1;
    ptr1 = ptr1->next;
    ptr2 = ptr2->next;
}

if(prev != NULL)
    prev->next = ptr1->next;
else
{
    A = ptr1->next;
}
```

```
    return A;
```

34. **Swap List Nodes in pairs**

(RECURSION)

```
ListNode* Solution::swapPairs(ListNode* A) {

    if(A == NULL || A->next == NULL)

    {

        return A;

    }

    ListNode *head = A;



    ListNode *ptr = head->next;

    head->next = swapPairs(head->next->next);

    ptr->next = head;

    return ptr;

}
```

35. **Reorder List**

1. Reverse the LL from the middle element.
2. Now take 2 pointers, one from beginning and other from element next to middle element and perform necessary operations.

36. **Palindrome List(same as previous)**

37. **Sorted Permutation Rank**

```
string temp = A;



    //Sort to know all the cahracters in the string and push into a vector

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



    vector<char> order;

    vector<long long> perms(n, 1);



    for(int i = 0; i < temp.size(); i++)

    {

        order.push_back(temp[i]);

    }
```

```
//Number of permutations ,i.e., n!

for(int i = 1; i < perms.size(); i++)

{

    perms[i] = i*(perms[i-1]%1000003)%1000003;

}




// result = position of element in sorted list * permutation[n - 1]

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

{

    long long pos = find(order.begin(), order.end(), A[i]) - order.begin();

    result += pos * perms[order.size() - 1];



    if(result > 1000003)

    {

        result %= 1000003;

    }
    order.erase(order.begin() + pos);

}



return (result + 1) % 1000003;
```

38. Nearest Smaller Element: Given an array, find the nearest smaller element G[i] for every element A[i] in the array such that the element has an index smaller than i.

1. Push the (last element, index) to the stack.
2. Iterate array from 2nd last element and **push** the element **if element is greater** then stack top. **Else** keep popping the element from stack till A[i] < stack.top() and finally push the element.

```
stack<pair<int, int>> temp;

    int n = A.size();
    vector<int> result(n, -1);

    temp.push(make_pair(A[n - 1], n - 1));

    for(int i = n - 2; i >= 0; i--)
    {
        pair<int,int> t = temp.top();
        if(A[i] < t.first)
        {
            while(!temp.empty() && A[i] < t.first)
            {
                result[t.second] = A[i];
                temp.pop();
                if(!temp.empty())
                    t = temp.top();
            }
        }
        temp.push(make_pair(A[i], i));
    }

    return result;
```

39. **Redundant Braces:** Write a program to validate if the input string has redundant braces?

```
((a + b)) has redundant braces so answer will be 1
(a + (a + b)) doesn't have have any redundant braces so answer will be 0
```

Sol: Check if there is an operator present between an opening brace and closing brace.

```
stack<char> temp;
    int count = 0;

    for(int i = 0; i < A.size(); i++)
    {
        if(A[i] == '+' || A[i] == '-' || A[i] == '*' || A[i] == '/' || A[i] == '(')
        {
            temp.push(A[i]);
        }
        if(A[i] == ')')
        {
            count = 0;
            char t = temp.top();
            while(t != '(')
            {
                if(t == '+' || t == '-' || t == '*' || t == '/' || t == '(')
                {
                    count++;
                }
                temp.pop();
                t = temp.top();
            }

            // cout << count << endl;
            if(count == 0)
            {
                return 1;
            }
            temp.pop();
        }
    }
    return 0;
```

40. **Rain Water Trapped:** https://www.interviewbit.com/problems/rain-water-trapped/

```
vector<int> left(n);
    vector<int> right(n);

    int result = 0;

    left[0] = A[0];
    for(int i = 1; i < n; i++)
    {
        left[i] = max(left[i - 1], A[i]);
    }

    right[n - 1] = A[n - 1];
    for(int i = n - 2; i >= 0; i--)
    {
        right[i] = max(right[i + 1], A[i]);
    }

    for(int i = 0; i < n; i++)
    {
        result += min(left[i], right[i]) - A[i];
    }

    return result;
```

41. **Largest Rectangular Area in a Histogram**

https://www.youtube.com/watch?v=ZmnqCZp9bBs

1. Push 0 in the stack.

2. If element is greater than or equal to the stack top, push it's **index** in the stack.
3. Else pop the stack elements until the stack top is less than or equal to the element.
4. Calculate area as follows whenever an element is removed from the stack:
    a. If stack is empty: area = A[top] * (i - stack.top() - 1)
    b. Else: area = A[top] * i

```
priority_queue<int> temp;
   for(int i = 0; i < B; i++)
   {
      temp.push(A[i]);
   }

   result.push_back(temp.top());
   for(int i = B; i < n; i++)
   {
      int max = temp.top();
      int pos = find(A.begin(), A.end(), max) - A.begin();
      // cout << pos << endl;
      if(i - pos >= B)
      {
         while(i - pos >= B)
         {
            temp.pop();
            max = temp.top();
            pos = find(A.begin(), A.end(), max) - A.begin();
         }
      }
      temp.push(A[i]);
      result.push_back(temp.top());
   }
```

42. **Letter Phone:** Given a digit string, return all possible letter combinations that the number could represent.**(Backtracking)**

Function Call: helper(A, temp, m, 0, result);

```
void helper(string A, string temp, map<char, string> &m, int i, vector<string> &result)
{
   if(temp.size() == A.size())
   {
      result.push_back(temp);
      return;
   }

   for(int j = 0; j < m[A[i]].size(); j++)
   {
      temp += m[A[i]][j];
      helper(A, temp, m, i + 1, result);
      temp.pop_back();
   }
}
```

43. **Combination Sum**

https://www.interviewbit.com/problems/combination-sum/

44. **Next Permutation**

1. Iterate from end and find the first element which is smaller than its next element.
2. If there is no such element, i.e., if i == -1 then reverse the complete array and return.
3. Again iterate from last and find first element that is greater than A[i].(j)
4. Swap A[i] and A[j].
5. Reverse the elements from i + 1 to end.

```
int n = a.size();
   int i = n - 2;

   for(; i >= 0; i--)
   {
      if(a[i] < a[i + 1])
      {
         break;
      }
```

```
    }

    if(i == -1)
    {
        sort(a.begin(), a.end());
        return;
    }

    int j = n - 1;
    while(j > i)
    {
        if(a[j] > a[i])
        {
            break;
        }
        j--;
    }
    // cout << i << " " << j << endl;

    // cout << "Here: " << i << " " << j << endl;
    swap(a[i], a[j]);
    sort(a.begin() + i + 1, a.end());
```

45. Find Permutation: Given a positive integer n and a string s consisting only of letters D or I, you have to find any permutation of first n positive integer that satisfy the given input string.

D means the next number is smaller, while I means the next number is greater.

1. Add numbers from 1 to n in an array.
2. Take 2 pointers: ‘low’ at 0 and high at n - 1
3. If string has a ‘D’ add high to the result and decrease high. Else add low to the result and increase low.

```
vector<int> order(B + 1);
    vector<int> result;

    for(int i = 1; i <= B; i++)
    {
        order[i] = i;
    }

    int low = 1, high = B;

    for(int i = 0; i < A.size(); i++)
    {
        if(A[i] == 'D')
        {
            result.push_back(order[high]);
            high--;
        }
        else
        {
            result.push_back(order[low]);
            low++;
        }
    }

    result.push_back(order[low]);

    return result;
```

46. **Fraction to Recurring Decimal**
https://www.interviewbit.com/problems/fraction/

1. Check the signs of numerator and denominator and make both of them absolute.
2. Divide the numerator with denominator and add the result to the string.
3. Create a map which stores the remainder and **current size of the string**.
4. If remainder is 0, return the result else:
    a. While remainder != 0, check whether the remainder exists in the map already.
    b. If it already exists insert “(“ at the map[rem] position. Add “)” at the end and this string is the result.
    c. Else multiply the remainder by 10, calculate rem / den and add this to string and make rem %= den.

47. **Equal :** Given an array A of integers, find the index of values that satisfy `A + B = C + D`, where `A,B,C & D` are integers values in the array.

https://www.interviewbit.com/problems/equal/

```
map<int, set<pair<int,int>>> m;
Loop i = 1 to N :
    Loop j = i + 1 to N :
        calculate sum
        If in hash table any index already exist for sum then
            try to find out that it is valid solution or not IF Yes Then
update solution
        update hash table
    EndLoop;
EndLoop;
```

48. Generate all Parentheses

```
void helper(int open, int close, string temp, vector<string> &result)

{

    if(open == 0 && close == 0)

    {

        result.push_back(temp);

        return;

    }



    if(close > 0)

    {

        helper(open, close - 1, temp + ")", result);

    }

    if(open > 0)
    {
        helper(open - 1, close + 1, temp + "(", result);
    }
}
vector<string> Solution::generateParenthesis(int A) {
    vector<string> result;

    string temp = "";

    helper(A, 0, temp, result);
    sort(result.begin(), result.end());

    return result;
}
```

49. Permutation Sequence

```
vector<int> totalPerms(10, 1);
    vector<int> nums;
    string result = "";

    for(int i = 0; i < 9; i++)
    {

        nums.push_back(i + 1);

    }


    for(int i = 1; i <= 9; i++)

    {

        totalPerms[i] = i * totalPerms[i - 1];

    }


    while(n != 0)

    {

        int pos = (k - 1) / totalPerms[n - 1];

        result += to_string(nums[pos]);


        nums.erase(nums.begin() + pos);

        k = k - pos * totalPerms[n - 1];

        n--;

    }


    return result;
```

50. **Isomorphic Strings**

The idea is that we need to map a char to another one, for example, "egg" and "add", we need to constract the mapping 'e' -> 'a' and 'g' -> 'd'. Instead of directly mapping 'e' to 'a', another way is to mark them with same value, for example, 'e' -> 1, 'a'-> 1, and 'g' -> 2, 'd' -> 2, this works same.

```
if(s.size() != t.size())

        {

            return false;

        }



        vector<int> v1(128, 0);

        vector<int> v2(128, 0);



        for(int i = 0; i < s.size(); i++)

        {

            if(v1[s[i]] != v2[t[i]])

            {

                return false;

            }

            v1[s[i]] = v2[t[i]] = i + 1;

        }



        return true;
```

51. **Sliding Window Maximum: Given an array *nums*, there is a sliding window of size *k* which is moving from the very left of the array to the very right. You can only see the *k* numbers in the window. Each time the sliding window moves right by one position. Return the max sliding window.**

Method 1: Using priority_queue
Method 2: Using deque

1. Create a Deque, *Qi* of capacity k, that stores only useful elements of current window of k elements. An element is useful if it is in current window and is greater than all other elements on left side of it in current window.
2. We process all array elements one by one and maintain *Qi* to contain useful elements of current window and these useful elements are maintained in sorted order. The element at front of the *Qi* is the largest and element at rear of *Qi* is the smallest of current window.

```
class Solution {
public:
   vector<int> maxSlidingWindow(vector<int>& nums, int k) {
      deque<int> pq;
      vector<int> result;

      if(nums.size() == 0)
      {
         return result;
      }

      int n = nums.size();
```

```
        //For initial 'k' elements
        for(int i = 0; i < k; i++)
        {
            while(!pq.empty() && nums[pq.back()] < nums[i])
            {
                pq.pop_back();
            }

            pq.push_back(i);
        }

        //For rest of the elements
        for(int i = k; i < n; i++)
        {
            result.push_back(nums[pq.front()]);

            while(!pq.empty() && i - pq.front() >= k)
            {
                pq.pop_front();
            }

            while(!pq.empty() && nums[pq.back()] < nums[i])
            {
                pq.pop_back();
            }

            pq.push_back(i);
        }

        result.push_back(nums[pq.front()]);

        return result;
    }
};
```

52. **Combinations:** Given two integers *n* and *k*, return all possible combinations of *k* numbers out of 1 ... *n*.

```
void helper(int i, int k, vector<int> &order, vector<int> &temp, vector<vector<int>>
&result)
   {
      if(temp.size() == k)
      {
         result.push_back(temp);
         return;
      }

      for(int j = i; j < order.size(); j++)
      {
         temp.push_back(order[j]);
         helper(j + 1, k, order, temp, result);
         temp.pop_back();
      }
   }
   vector<vector<int>> combine(int n, int k) {
      vector<vector<int>> result;
      vector<int> order;

      //Create an array with numbers between 1 to n
      for(int i = 1; i <= n; i++)
      {
         order.push_back(i);
      }

      //Push the current element and recursively iterate for the next elements
      for(int i = 0; i <= n - k; i++)
      {
         vector<int> temp;
         temp.push_back(order[i]);
         helper(i + 1, k, order, temp, result);
      }

      return result;
}
```

53. **Largest Continuous Sequence Zero Sum:** Find the largest continuous sequence in an array which sums to zero.

https://www.interviewbit.com/problems/largest-continuous-sequence-zero-sum/

54. **Square Root**

```
 if(x < 2)
{
    return x;
}
int low = 0;
int high = x / 2;

while(low <= high)
{
    int mid = (low + high) / 2;
    long long sq = (long long)mid * mid;

    if(sq == x)
    {
        return mid;
    }

    if(sq > x)
    {
        high = mid - 1;
    }
    else
    {
        low = mid + 1;
    }
}

return low - 1;
```

55. **Balanced Binary Tree**

```
int calcHeight(TreeNode* A)
 {
     if(A == NULL)
     {
         return 0;
     }

    int left = 1 + calcHeight(A->left);
    int right = 1 + calcHeight(A->right);

    return max(left, right);
 }
int Solution::isBalanced(TreeNode* A) {
    if(A == NULL)
    {
        return 1;
    }

    int left = calcHeight(A->left);
    int right = calcHeight(A->right);

    if(isBalanced(A->left) && isBalanced(A->right) && abs(left - right) <= 1)
    {
        return 1;
    }

    return 0;
}
```

56. **Identical Binary Trees**

```
if(!A && !B)
```

```
{
     return 1;
}


if(!A || !B || A->val != B->val)
{
     return 0;
}

return(isSameTree(A->left, B->left) && isSameTree(A->right, B->right));
```

57. **Max Depth of Binary Tree**

```
if(A == NULL)
{
     return 0;
}

return (max(maxDepth(A->left), maxDepth(A->right)) + 1);
```

58. **Min Depth of Binary Tree**

```
   if(!A->left && !A->right)
   {
      return 1;
   }

   int left = INT_MAX;
   int right = INT_MAX;

   if(A->left)
   {
      left = 1 + minDepth(A->left);
   }
   if(A->right)
   {
      right = 1 + minDepth(A->right);
   }

   return (min(left, right));
```

59. **Symmetric Binary Tree**

```
bool helper(TreeNode *left, TreeNode *right)
{
    if(!left && !right)
    {
        return true;
    }
    if(!left || !right)
    {
        return false;
    }

    bool b1 = false;
    bool b2 = false;

    if(left->val == right->val)
    {
        b1 = helper(left->right, right->left);
        b2 = helper(left->left, right->right);
    }

    return b1 && b2;
}
int Solution::isSymmetric(TreeNode* A) {
    return helper(A->left, A->right);
}
```

60. Sorted Array To Balanced BST

```
TreeNode* helper(const vector<int> &A, int low, int high)
{
    if(low > high)
    {
        return NULL;
    }

    int mid = (low + high) / 2;
    TreeNode* t = new TreeNode(A[mid]);
    t->left = helper(A, low, mid - 1);
    t->right = helper(A, mid + 1, high);
    return t;
}
TreeNode* Solution::sortedArrayToBST(const vector<int> &A) {
    int n = A.size();
    TreeNode* root = helper(A, 0, n - 1);

    return root;
}
```

60. Shortest Unique Prefix

Create a TRIE and maintain a variable "**frequency**" for each letter. For each word where the **letter with frequency "1"** appears that is its shortest unique prefix.

61. Populating Next Right Pointers in Each Node II(Not a complete binary tree)

```
void connect(TreeLinkNode *root) {
        while(root != NULL)
        {
            TreeLinkNode* temp = new TreeLinkNode(0);
            TreeLinkNode* current = temp;

            while(root != NULL)
            {
                if(root->left)
                {
                    current->next = root->left;
                    current = current->next;
                }

                if(root->right)
                {
                    current->next = root->right;
                    current = current->next;
                }
                root = root->next;
            }
            root = temp->next;
        }
    }
```

62. Populating Next Right Pointers in Each Node(Complete Binary Tree):

```
void connect(TreeLinkNode *root) {
        if(root == NULL )
        {
            return;
        }

        if(root->left == NULL && root->right == NULL)
        {
            return;
        }

        TreeLinkNode* temp = root;
        while(temp != NULL)
        {
            temp->left->next = temp->right;
            if(temp->next)
            {
                temp->right->next = temp->next->left;
            }

            temp = temp->next;
        }
        connect(root->left);
    }
```

63. Jump Game:

```
        int n = nums.size();
        int maxl = nums[0]; //Max jump from given position

        for(int i = 1; i <= maxl && maxl <= n - 1; i++)
        {
            maxl = max(maxl, nums[i] + i);
        }

        if(maxl >= n - 1)
        {
            return true;
        }

        return false;
```

64. Max Product Subarray

```
int n = A.size();

    int minVal = A[0];
    int maxVal = A[0];
    int maxProd = A[0];

    for(int i = 1; i < n; i++)
    {
        if(A[i] < 0)
        {
            swap(minVal, maxVal);
        }
        maxVal = max(A[i], maxVal * A[i]);
        minVal = min(A[i], minVal * A[i]);

        maxProd = max(maxProd, maxVal);
    }

    return maxProd;
```

65. Seats: There is a row of seats. Assume that it contains `N` seats adjacent to each other. There is a group of people who are already seated in that row randomly. i.e. some are sitting together & some are scattered.
An occupied seat is marked with a character `'x'` and an unoccupied seat is marked with a `dot ('.')`

Now your target is to make the whole group sit together i.e. next to each other, without having any vacant seat between them in such a way that the total number of hops or jumps to move them should be minimum.

```
int Solution::seats(string A) {

    int n = A.size();

    vector<int> filled;

    int result = 0;



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

    {

        if(A[i] == 'x')

        {

            filled.push_back(i);

        }

    }
```

```
if(filled.size() <= 1)

{

    return 0;

}



int mid = filled.size() / 2;

int low = filled[mid] - 1;

int high = filled[mid] + 1;



for(int i = mid - 1; i >= 0; i--)

{

    result += low - filled[i];

    low--;

}



for(int i = mid + 1; i < filled.size(); i++)

{

    result += filled[i] - high;

    high++;

}



return result;
```

```
}
```

66. Ugly Numbers: Given three prime number(`p1, p2, p3`) and an integer k. Find the first(smallest) k integers which have only `p1, p2, p3` or a combination of them as their prime factors.

```
1 Declare an array for ugly numbers:  ugly[n]

2 Initialize first ugly no:  ugly[0] = 1

3 Initialize three array index variables i2, i3, i5 to point to

   1st element of the ugly array:

        i2 = i3 = i5 =0;

4 Initialize 3 choices for the next ugly no:

         next_mulitple_of_2 = ugly[i2]*2;

         next_mulitple_of_3 = ugly[i3]*3

         next_mulitple_of_5 = ugly[i5]*5;

5 Now go in a loop to fill all ugly numbers till k and check if it
hasn't occurred before
```

```
next_ugly_no  = Min(next_mulitple_of_2, next_mulitple_of_3,
next_mulitple_of_5);


    ugly[i] =  next_ugly_no


    if (next_ugly_no  == next_mulitple_of_2)

    {
        i2 = i2 + 1;

        next_mulitple_of_2 = ugly[i2]*2;

    }

    if (next_ugly_no  == next_mulitple_of_3)

    {

        i3 = i3 + 1;

        next_mulitple_of_3 = ugly[i3]*3;

     }

     if (next_ugly_no  == next_mulitple_of_5)

     {

        i5 = i5 + 1;

        next_mulitple_of_5 = ugly[i5]*5;

     }
```

67. Longest Palindromic Substring:

https://www.interviewbit.com/problems/longest-palindromic-substring/

(i) O(n^2) space:

- **Fill all the diagonal elements with 1**
- **for(int i = 0; i < n - 1; i++)**

```
{

     if(A[i] == A[i + 1])

          dp[i][i+1] = 1
```

```
                maxLen = 2
}
```

- **Fill rest of the DP as upper traingular matrix as:**
  - **if(A[i] == A[j])**

```
            dp[i][j] = dp[i + 1][j - 1]
            maxLen = max(maxLen, j - i + 1)
```

(ii) O(n) space:

68. Maximum Absolute Difference: You are given an array of N integers, A1, A2 ,…, AN. Return maximum value of `f(i, j)` for all 1 ≤ *i, j* ≤ N.

`f(i, j)` is defined as `|A[i] - A[j]| + |i - j|`, where |x| denotes absolute value of x.

Sol. f(i, j) = |A[i] – A[j]| + |i – j| can be written in 4 ways (Since we are looking at max value, we don't even care if the value becomes negative as long as we are also covering the max value in some way).

69. Maximum Width Ramp: Given an array A of integers, a *ramp* is a tuple (i, j) for which i < j and A[i] <= A[j]. The width of such a ramp is j - i.

Find the maximum width of a ramp in A. If one doesn't exist, return 0.

1. Create a vector of pairs with number and its position as a pair.
2. Sort the array in ascending array.
3. Whenever we write an index i, we know there was a ramp of width **i - min(indexes_previously_written)**.
4. Return the max of the above value.

70. Edit Distance

Recursion:

```
int helper(string A, string B, int i, int j)
{
    // cout << i << " " << j << endl;
    if(i == A.size() && j == B.size())
    {
        return 0;
    }
    if(i == A.size() && j != B.size())
    {
        return B.size() - j;
    }
    if(i != A.size() && j == B.size())
    {
        return A.size() - i;
    }


    // cout << "Here" << endl;
    if(A[i] == B[j])
    {
        return helper(A, B, i + 1, j + 1);
    }
    else
    {
        return 1 + min(min(helper(A, B, i + 1, j), helper(A, B, i, j + 1)), helper(A, B, i + 1,
j + 1));                              (Insert)                     (Deletion)          (Replace)
    }
}
int Solution::minDistance(string A, string B) {
    return helper(A, B, 0, 0);
}
```

DP:

```
for (int i=0; i<=m; i++)
    {
        for (int j=0; j<=n; j++)
        {
            // If first string is empty, only option is to
            // insert all characters of second string
            if (i==0)
                dp[i][j] = j;  // Min. operations = j

            // If second string is empty, only option is to
            // remove all characters of second string
            else if (j==0)
                dp[i][j] = i; // Min. operations = i

            // If last characters are same, ignore last char
            // and recur for remaining string
            else if (str1[i-1] == str2[j-1])
                dp[i][j] = dp[i-1][j-1];

            // If the last character is different, consider all
            // possibilities and find the minimum
            else
                dp[i][j] = 1 + min(dp[i][j-1],  // Insert
                                   dp[i-1][j],  // Remove
```

```
                                    dp[i-1][j-1]); // Replace

        }

    }
```

71. Bitwise ORs of Subarrays

1. Take two sets, one which stores the previous ORs and one which initially stores the current number.
2. Take the OR of the second set(containing the current number) with all members of the previous ORs.
3. Make prevORs equal to currOrs.

```
class Solution {

public:

    int subarrayBitwiseORs(vector<int>& A) {

        unordered_set<int> prevORs;

        unordered_set<int> currORs;

        unordered_set<int> result;



        int n = A.size();

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

        {
            currORs = {A[i]};



            for(int j : prevORs)
            {
                currORs.insert(A[i] | j);
            }


            prevORs = currORs;
            for(int j : currORs)
                result.insert(j);
        }


        return result.size();
    }
};
```

72. Longest Arithmetic Progression(https://www.interviewbit.com/problems/longest-arithmetic-progression/)

1. For 3 elements, i, j and k, AP is given by j - i = k - j. So the equation can be simplified as:

2j - k = i

2. Take a 2d DP and **initialise all elements by 2**.
3. Make dp[i][i] = 1 as AP of number with itself has length 1.
4. Push the first element into a map with position as its value.
5. Iterate from second element and for all elements check whether 2*j - k is present in the map or not. If it is present, then

   Dp[i][j] = dp[m[temp]][i] + 1

   where m[temp[i]] is the position of element in the map.

   And result = max(result, dp[i][j])

73. Longest Increasing Subsequence:

O(n^2) Solution:

1. Take a 1d vector to store the longest increasing subsequence till that particular element.
2. For each element, iterate for all the elements before it and check if they are smaller.
3. If an element is smaller than current element, then update the value for current element.

```
for(int i = 1; i < n; i++)
    {
        int m = 0;
        for(int j = 0; j < i; j++)
        {
            if(A[i] > A[j])
            {
                m = max(m, inc[j]);
                inc[i] = 1 + m;
            }
        }
    }
```

74. Longest valid Parentheses

1. Take two pointers left and right.
2. Iterate string from left to right:
    a. Increment left if ‘(‘ appears and right if ‘)’ appears.
    b. if(left == right) and left + right > result then update result = left + right.
    c. If right > left then reinitialise right and left to 0.
3. Reinitialise left and right to 0
4. Iterate string from right to left:
    a. Same as ‘a’
    b. Same as ‘b’
    c. If right < left then reinitialise.

75. Jump Game 2:

Let's say the range of the current jump is [curBegin, curEnd], **curFarthest** is the farthest point that all points in [curBegin, curEnd] can reach. Once the current point reaches curEnd, then trigger another jump, and **set the new curEnd with curFarthest**, then keep the above steps, as the following:

Code:

```
int n = nums.size();

int currEnd = 0, currFarthest = 0, jumps = 0;

 int i = 0;

for(; i < n - 1; i++)
```

```
    {

        if (i > currFarthest) return -1;                                             //If end isn't
always reached

        currFarthest = max(currFarthest, i + nums[i]);

        if(i == currEnd)

        {

            jumps++;

          currEnd = currFarthest;


            if(currEnd >= n - 1)

            {

                break;

            }

        }

    }

    return jumps;
```

76. Minimum Moves to Equal Array Elements: Given a non-empty integer array, find the minimum number of moves required to make all array elements equal, where a move is incrementing a selected element by 1 or decrementing a selected element by 1.

1. Take two pointers, start one from 0 and other from (length - 1).
2. Add the difference of two numbers to the result.

Code:

```
long long n = nums.size();
 sort(nums.begin(), nums.end());
  int result = 0;
 int i = 0, j = n - 1;
 while(i < j)
 {
     result += nums[j] - nums[i];
     i++;
     j--;
 }
 return result;
```

77. Flip String to Monotone Increasing

https://leetcode.com/problems/flip-string-to-monotone-increasing/

1. Calculate prefix sum to get number of 1’s at a particular position.
2. For each point, calculate flips required to convert 1’s on its left to 0 and 0’s on its right to 1.
3. Return the min of the values calculated above.

78. Majority Element: O(n) time and O(1) space

```
    int major=num[0], count = 1;
    for(int i=1; i<num.length;i++){
        if(count==0){
            count++;
            major=num[i];

        }else if(major==num[i]){
            count++;

        }else count--;

    }
    return major;
```