(1) Given a Sorted Array arr[] of size N and an integer k. The task is to check if k is present in the Array or not using ternary Search.

Example :- Input :- N = 5, k = 6

arr[] = {1, 2, 3, 4, 6}

Output :- 1

Explanation :- Since, 6 is present in the array at index 4, output is 1.

⟹ It is similar to Binary Search where we divide the array into two parts but in this algorithm, we divide the given array into three parts by taking mid1 and mid2

$$mid\ 1 = l + (r-l)/3$$

$$mid\ 2 = r - (r-l)/3$$

Steps :- (1) First we Compare the key with the element at mid1. If found equal, return mid1.

(2) If not, then we compare the key with the element at mid2. If found equal, we return mid2.

(3) If not, then we Compare if the key is less than the element at mid1. If yes, then recur to the first part.

(4) If not, the we check whether the key is greater than the element at mid2. If yes, then recur to the third part.

(5) If not, then we recur to the second (middle) part.

★ Recursive Approach :-

```
int T Search_recursive (struct Array arr, int l, int h, int key)
{
    if (h >= l)
    {
        int mid1 = l + (h-l)/3;
        int mid2 = h - (h-l)/3;
        if (arr.A[mid1] == key)
            return mid1;
        if (arr.A[mid2] == key)
            return mid2;
        else if (arr.A[mid1] > key)
            return T Search_recursive (arr, l, mid1-1, key);
        else if (arr.A[mid2] < key)
            return T Search_recursive (arr, mid2+1, h, key);
        else
            return T Search_recursive (arr, mid1+1, mid2-1, key);
    }
    return -1;
}
```

★ Iterative Approach :-

```
int T Search_iterative (struct Array arr, int l, int h, int key)
{
    while (l <= h)
    {
        int mid1 = l + (h-l)/3;
        int mid2 = h - (h-l)/3;
```

```
if (arr.A[mid1] == key)
    return mid1;
if (arr.A[mid2] == key)
    return mid2;
else if (arr.A[mid1] > key)
    h = mid1 - 1;
else if (arr.A[mid2] < key)
    l = mid2 + 1;
  else
   {
      l = mid1 + 1;
      h = mid2 - 1;
   }
}
return -1;
}
```

---

(2) **Kadane's Algorithm** :- Given an Array Arr[] of N integers. Find the Contiguous Sub-array (containing at least one number) which has the maximum sum and return its sum.

⟹ The Simple Idea of Kadane's Algorithm is to look for all positive contiguous segments of the array (max_ending_here is used for this). And keep track of maximum sum contiguous segment among all positive segments (max_sum is used for this). Each time we get a positive-sum compare it with max_sum and update max_sum if it is greater than max_sum.

-2	-3	4	-1	-2	1	5	8	7	-3	2

4 + (-1) + (-2) + 1 + 5 + 8 + 7

Maximum Contiguous Array Sum is 22

Largest Sub Array Sum Problem

Imp

Algorithm :-

```
int Maximum_SubArray_Sum ( struct Array arr)
{
    int max_Sum = 0 ;
    int max_ending_here = 0 ;
      int Start = 0, S = 0 , end = 0 ;
    for (int i = 0 ; i < arr. length ; i++)
    {
        max_ending_here = max_ending_here + arr. A[i];
        if (max_ending_here > max_Sum)
            {
                max_Sum = max_ending_here;
                    Start = S ;
                    end = i ;
            }
        if (max_ending_here < 0)
            {
                max_ending_here = 0 ;
                    S = i+1 ;
            }
    }
    printf (" Starting Index : %d \n", Start) ;
    printf (" Ending Index : %d \n", end) ;
    Return max_Sum ;
}
```

3) **Josephus Problem** :- Given the total no. of persons 'n' and a number k which indicates that k-1 persons are skipped and the k^{th} person is killed in the circle in a fixed Direction.

The task is to choose the safe place in the circle so that when you perform these operations starting from 1st place in the circle, you are the last one remaining and survive.

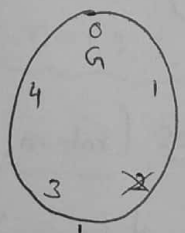
Example :- Input , n = 5, k = 3

Output = 4

Explanation :- There are 5 persons so skipping 2 person ie. 3rd person will be killed. Thus the safe position is 4

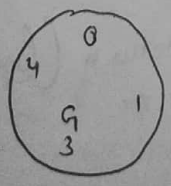
⟹ We will divide this problem into sub problem as we always do in recursion.

n = 5, k = 3

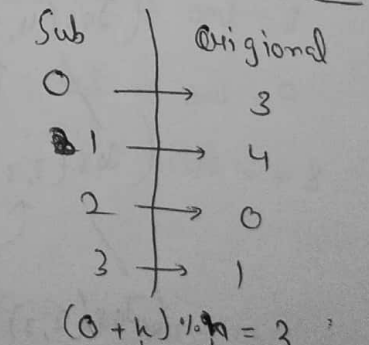


n = 4, k = 3 (original)

n = 4 , k = 3 (sub problem)



Relative position



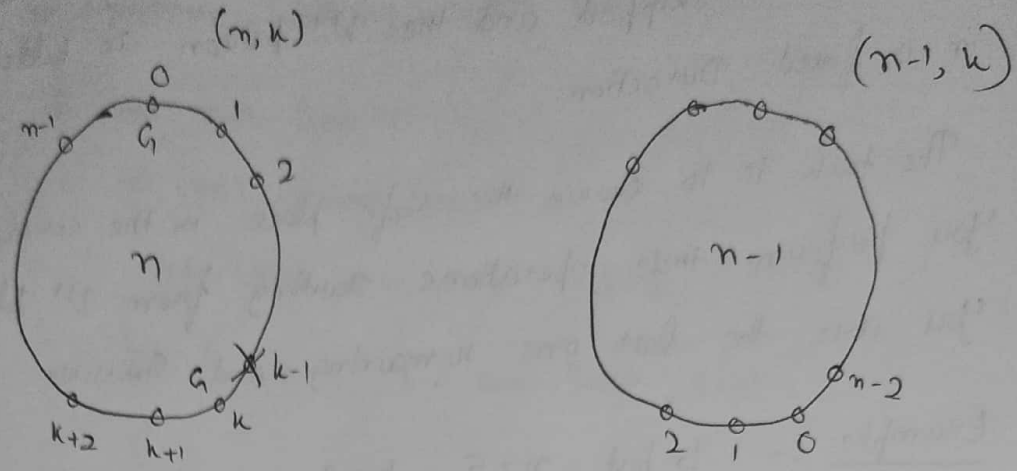
(0 + k) % n = 3

3 5

Now, After killing one person among five or if we start with 4 persons, no. of persons will be same but position of gun will be different

So, J (n, k) = J (n-1, k) + ...

Now, to generalize the solution for converting sub problem into original.



$0 \rightarrow (k) \% n$

$1 \rightarrow (k+1) \% n$

$2 \rightarrow (k+2) \% n$

⋮

Note :- '% n' is used for bcz we do this activity in circular fashion so there will be a chance of overflow.

$$f(n,k) = \left((f(n-1,k) + k) \% n \right)$$

Program :-

Jos (5, 3) = 3

(Jos(4, 3) + 3) % 5 = 3

(Jos(3, 3) + 3) % 4 = 0

(Jos(2, 3) + 3) % 3 = 1

(Jos(1, 3) + 3) % 2 = 1

0

```
int Jos (int n, int k)
{
    if (n == 1)
        return 0;
    return ((Jos (n-1, k) + k) % n);
}
```

(4) **Special keyboard** :- Imagine you have a special keyboard with the following keys :-

key 1 :- Print 'A' on Screen
key 2 :- (Ctrl-A) : Select Screen
key 3 :- (Ctrl-c) : Copy selection to buffer
key 4 :- (Ctrl-v) : print buffer on screen appending it after what has already been printed.

Find maximum No. of A's that can be produced by pressing keys on the special keyboard N times.

Explanation :- Input N = 7 , output = 9

We can almost get 9 A's on Screen by pressing following key sequence.

A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V

⇒ Now, we can see that if $n < 7$, than output is N itself.

So, For n = 1, 2, 3, 4, 5, 6 ⇒ ans = n (1, 2, 3, 4, 5, 6)

For n = 7 ⇒ AAAA → Ctrl A → Ctrl C → Ctrl V ⇒ ans = 8
AAA → Ctrl A → Ctrl C → Ctrl V → Ctrl ⇒ ans = 9
AA → Ctrl A → Ctrl C → Ctrl V → Ctrl V → Ctrl V ⇒ 8
A ⇒ ans ⇒ 7

For n = 8 ⇒ AAAAA → Ctrl A → Ctrl C → Ctrl V ⇒ ans = 10
AAAA → For (n = 7) + no. of A's in pair = 12
AAA → For (n = 7) + no. of A's in pair = 12
AA → For (n = 7) + no. of A's in pair = 10
A → 8

So, we observe that the sequence of N keystrokes which produces an optimal string length will end with a suffix of ctrl-A, ctrl-C, followed by only ctrl-V's (for $N > 6$)

The Task is to find out the break point after which we get the above suffix of keystrokes. Breakpoint is that instance after which we need to only press ctrl-A, ctrl-C once and the only ctrl-V's afterwards to generate the optimal length. If we loop from N-3 to 1 and choose each of these values for the breakpoint, and Compute that optimal string they would produce. Once the loop ends, we will have the maximal of the optimal lengths for various breakpoints, thereby giving us the optimal length for N keystrokes.

Program

```
int keyboard (int N)
{
  if (N < 7)
    return N;
  else
    {
      int max = 0;
      int b;
      for (b = N-3; b >= 1; b--)
      {
        int curr = (N-b-1) * keyboard(b);
        if (curr > max)
          max = curr;
      }
      return max;
}
```

Now, if $n = 19$:-

Step-1 :- $b = 16$

$$\text{Cum} = (19 - 16 - 1) * \text{keyb}(16)$$

$$= 2 * \text{keyb}(16)$$

$$= 2 * 2 * \text{key}(13)$$

$$= 2 * 2 * 2 * \text{key}(10)$$

$$= 2 * 2 * 2 * 2 * \text{key}(8)$$

$$= 2 * 2 * 2 * 2 * 2 * \text{keyb}(6)$$

$$= 2 * 2 * 2 * 2 * 2 * 6$$

$$= 192$$

and will do this until $b >= 1$

And will get maximum value.

(5) Jump Game :- Given a positive integer N and a list of N integers A[]. Each element in the array denotes the maximum length of jump you can cover. Find out if you can make it to the last index if you start from first index of the list.

Explanation :- Input : N = 6.

A[] = {1, 2, 0, 3, 0, 0}

Output : 1

explanation :- Jump 1 step from first index to second index. Then jump 2 steps to reach 4th index and now Jump 2 steps to reach end.

$1 \longrightarrow 2 \rightarrow 3 \longrightarrow 0$

$\Longrightarrow$ Implementation :-

(1) maxReach :- The variable maxReach stores at all time the maximal reachable index in the array.

(2) Jump :- Jump stores the amount of jumps necessary to reach the maximal reachable position, it also indicates the current jump we are making in the array.

(3) Step :- The variable step stores the no. of steps we can still take in the current jump 'jump' (and is initialized with value at index 0, ie initial no. of steps)

Now, lets we have an Array, Arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9}

- maxReach = arr[0]; // arr[0] = 1, so max. index we can reach at that moment is 1.
  step = arr[0]; // arr[0] = 1, the no. of steps we can take is still 1.
  jump = 1; // we are currently making our first jump.

• Now starting iteration from index 1, the above values are updated as follows:

① First we test whether we have reached the end of the array, in that case we just need to return the jump variable.

```
if (i == arr.length - 1)
    return jump;
```

② Next we update the maxReach. This is equal to the maximum of maxReach and i + arr[i] (the no. of steps we can take from current position)

```
maxReach = Math.max (maxReach, i + arr[i]);
```

③ We used up a step to get to the current index, so steps have to be decreased

```
step --;
```

④ If no more steps are remaining (ie. steps = 0, then we must have used a jump). Therefore increase jump. Since we know that it is possible somehow to reach maxReach, we again initialize the steps to the no. of steps to reach maxReach from position i. But before re-initializing step, we also check whether a step is becoming zero or negative. In this case, it is not possible to reach further.

```
if (step == 0)
{
    jump++;
    if (i >= maxReach)
        return -1;
    step = maxReach - i;
}
```

Code :-

```
int max (int x, int y)
{
    return (x>y) ? x : y;
}

int minJumps (int arr[], int n)
{
    if (n<=1)
        return 0;
    if (arr[0]==0)
        return -1;
    int maxReach = arr[0];
    int step = arr[0];
    int jump = 1;
    for (int i=1 ; i<n ; i++)
    {
        if (i==n-1)
            return jump;
        maxReach = max (maxReach, i+arr[i]);
        step--;
        if (step==0)
        {
            jump++;
            if (i>=maxReach)
                return -1;
            step = maxReach - i;
        }
    }
    return -1;
}
```

(6) Given An array A[] of N positive integers. The task is to find the maximum of j-i subjected to the constraint of A[i] ≤ A[j] and i ≤ j

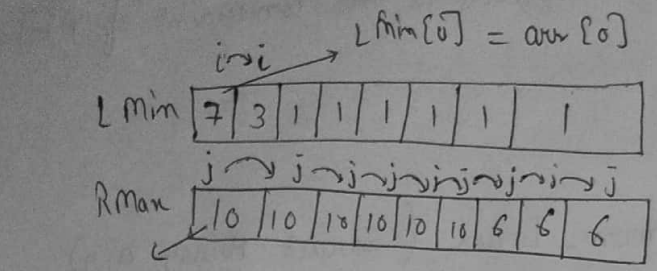
Approach 1 :-

```
int Maximum_Index ( Struct Array arr)
{
    int i, j;
    int max = 0;
    int diff = 0;
    for ( i=0 ; i< arr.length -1; i++)
    {
        for ( j = i+1 ; j< arr.length ; j++)
        {
            if (A[i] <= A[j])
                diff = j - i;
            if (diff > max)
        }   max = diff;
    }
}   return max;
```

Time :- $O(n^2)$

Approach - 2 :- It will take O(n) time, but with O(n) space, In this method we will take two Auxillary arrays LMin[] and RMax[] such that LMin[i] holds smallest element on left side and RMax[] holds greatest element on right side, we traverse both these arrays and if LMin[i] > RMax[j] then do i++ bcz all elements on left side of LMin[i] are greater or equal to LMin[i]. otherwise we must move ahead in RMax[j] to look for greater j-i value.

Arr = { 7, 3, 1, 8, 9, 10, 4, 5, 6 }



RMax[0] = arr[0]; So, j = 8 & i = 1

⇒ j - i ⇒ 8 - 1 = 7

Program :-

```
int maxIndexDiff (int Arr[], int n)
{
    int maxDiff = 0;
    int i, j;
    int LMin[n], int RMax[n];
    LMin[0] = Arr[0];
    RMax[n-1] = Arr[n-1];
    for (i=1; i<n; i++)
    {
        LMin[i] = min (Arr[i], LMin[i-1]);
    }
    for (j=n-2; j>=0; j--)
    {
        RMax[j] = max (Arr[i], RMax[j+1]);
    }
    i=0, j=0, maxDiff = -1;
    while (j<n && i<n)
    {
        if (LMin[i] <= RMax[j])
        {
            maxDiff = max(maxDiff, j-i);
            j = j+1
        }
        else
            i = i+1;
    }
    return maxDiff;
}
```

(7) Maximum Circular SubArray Sum :- Given an array Arr[] of N integers arranged in a circular fashion. your task is to find the maximum Contigous Subarray Sum.

Input :- arr[] = { 8, -8, 9, -9, 10, -11, 12 }

output : 22

Starting from 12 and ends on 10

⇒ Now we know, If we want maximum circular SubArray Sum We'll always use kadane's Algorithm, But here Array is circular So, we will use kadanas Algorithm but with some modifications also, So Lets understand.

So, here Array is 8, -8, 9, -9, 10, -11, 12

So, Maximum SubArray Sum = 12+8-8+9-9+10 = 22

So, We can get this by doing in such a way, We will get first Sum of whole array and Subtract the minimum SubArray Sum from it, like

8 -8 9 -9 10 [-11] 12

↓ minimum SubArray

To get minimum SubArray Sum, We will change Signs of every elemant in the array and then get maximum Sum from that Array

-8 8 -9 9 -10 11 -12

So, maximum Sum from that Array = 11

then reverse its Sign and Subtract from whole Array and this Value with reversed Sign, is the minimum value for orisional Array ⇒ -11 and Now Subtract it

So, find result ⇒ 8 -8 +9 -9 +10 -11 +12 -(-11)

⇒ 11+11

⇒ 22

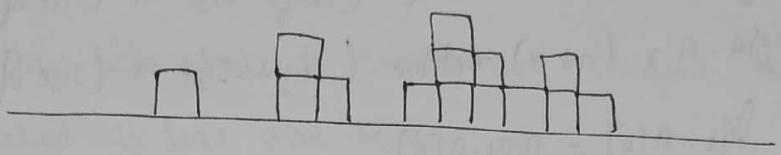
and also perform Simple kadane Algorithm and then find the maximum from both simple and modified kadane Algorithm.

```
int kadanes (Struct Array arr)
{
    int max_Sum = 0;
    int max_ending_here = 0;
    for (int i=0 ; i< arr.length ; i++)
    {
        max_Ending_here = max_ending_here + arr.A[i];
        if (max_ending_here > max_Sum)
            max_Sum = max_ending_here;
        if (max_ending_here < 0)
            max_ending_here = 0;
    }
    return max_Sum;
}

int Maximum_Circular_SubArray_Sum (Struct Array * arr, Struct Array arr2
{
    int Sum = 0;
    for (int i=0 ; i< arr -> length ; i++)
    {
        Sum = Sum + arr -> A[i];
        arr -> A[i] = -(arr -> A[i]);
    }
    int reverse_max_Sum_SubArray = kadanes (arr2);
    int answer = Sum - (- reverse_max_Sum_SubArray);
    return answer;
}
```

(8) Trapping Rain Water :- Given an Array arr[] of N non-negative integers representing the height of blocks. If width of each block is 1, Compute how much water can be trapped b/w the blocks during the rainy season.

Example :- arr[] = {0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1}

Output = 6



Now, To Solve this Question, we can see that water will only store if for that position there must be a taller building on the left and right side for that position

Now, we have to see how much water can be stored on that position, so will do Preprocessing ie. we'll take two auxillary arrays and it will left and Right, Left will fill is for that index which is largest height in left side and similar concept for right

left → 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3

3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 1 ← Right

Now, How much water is stored at that position

Formula (i) = Min (left[i], Right[i]) - A[i]

```
int Trapped_Water (struct Array arr)
{
    struct Array left;
    struct Array right;
    left.length = left.size = arr.length;
    left.size = right.size = arr.size;
    left.A = (int *) malloc (left.size * (sizeof(int)));
    right.A = (int *) malloc (right.size * (sizeof(int)));
    left.A[0] = arr.A[0];
    right.A[arr.length-1] = arr.A[arr.length-1];
    for (int i=1; i< left.length; i++)
    {
        left.A[i] = max (arr.A[i], left.A[i-1]);
    }
    for (int i = arr.length-2; i>=0; i--)
    {
        right.A[i] = max (arr.A[i], right.A[i+1]);
    }
    int answer = 0;
    for (int i=0; i< left.length; i++)
    {
        answer += min (left.A[i], right.A[i]) - arr.A[i];
    }
    return answer;
}
```

(9) Allocate minimum number of Pages :- you are given N number of books. Every i^{th} book has Ai number of Pages.

You have to allocate Contagious books to m number of students. There can be many ways or Permutations to do so. In each permutation, one of the m students will be allocated the maximum no. of Pages. out of all these permutations, the task is to find that particular permutation in which the maximum no. of Pages allocated to a student is the minimum of those in all the other permutations and print this minimum value.

Each book will be allocated to exactly one student. Each student has to be allocated at least one book.

Note :- Return -1 if a valid assignment is not possible, and allotment should be in Contigous order.

Example 1 :- Input : N = 4

A[] = {12, 34, 67, 90}

m = 2

output : 113

Explanation :- Allocation can be done in following ways:

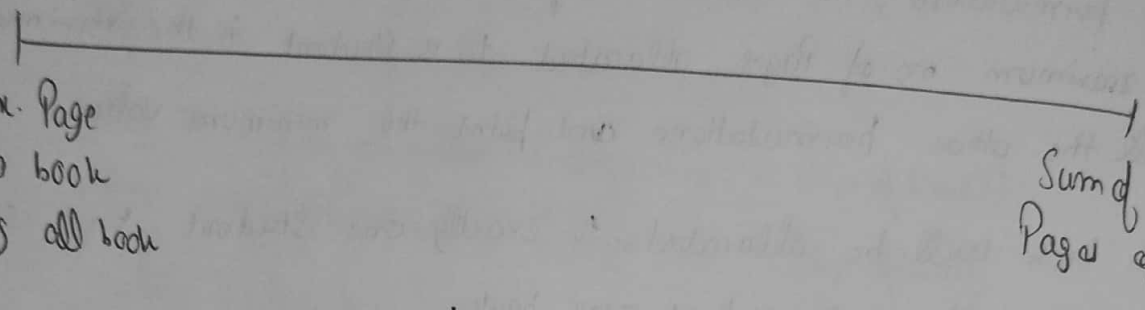
{12} and {34, 67, 90}

Maximum Pages = 191 {12, 34} and {67, 90}

Maximum Pages = 157 {12, 34, 67} and {90}

Maximum Pages = 113, Therefore the minimum of these Cases is 113, which is selected as the output.

like if there is only one student for all book ⇒ 12+34+67+90 = 203
like if there are equal no. of students for equal no. of books
ie. no. of students = no. of books
then, a student can read a book which has maximum no. of Pages, So the range will be from



and, we have to find b/w them, depending upon no. of students, so the idea is to use Binary Search. We fix a value for the number of Pages as mid of current minimum and maximum. We initialize minimum as maximum as 0 and Sum-of-all Pages respectively. If a current mid can be a solution, than we search on the lower half, else we search in higher half.
Now the ques. arises, how to check mid value is feasible or not? Basically, we need to check if we can assign Pages to all students in a way that the maximum no. doesnot exceed current value. To do this, we sequentially assign Pages to every student while the current number of assigned Pages doesn't exceed the value. In this Process, if the number of students becomes more than m, than the Solution is not feasible. Else feasible.

```
# include <stdbool.h> (for boolean return)

bool Possible(struct Array arr, int m,
              int curr_mid)
{
  int curr_sum = 0;
  int Student_Reluired = 1;

for (int i=0 ; i< arr.length ; i++)
  {
    if (arr.A[i] > curr_mid)
        return false;
    if (curr_sum + arr.A[i] > curr_mid)
    {
       Student_Reluired ++;
       Curr_Sum = arr.A[i];
       if (Student_Reluired > m)
           return false;
    }
    else
    {
      Curr_sum = Curr_Sum + arr.A[i];
    }
  }
} return true;
```

```
int Allocate_minimum_No_of_Pages
   (struct Array arr, int m)
{
    int Sum = 0;
    int max = 0;
  for(int i=0 ; i< arr.length ; i++)
  {
     Sum = Sum + arr.A[i];
     if (max < arr.A[i])
         max= arr.A[i];
  }
  int start = max;
  int end = Sum;
  int result = 0;
  while (start <= end)
  {
     int mid = (start + end)/2;
     if (Possible (arr, m, mid))
     {
        result = mid;
        end = mid-1;
     }
     else
     {
        start = mid+1;
     }
  }
  return result;
}
```

(10) Counter the Number of Possible Triangles :- Given an unsorted array arr[] of n positive integers. Find the number of triangles that can be formed with three different array elements as lengths of three sides of triangles.

Example :- Input : n = 5

arr[] = {6, 4, 9, 7, 8}

output : 10

⟹ Basic Condition for a triangle is $a+b>c$ && $a+c>b$ && $b+c>a$

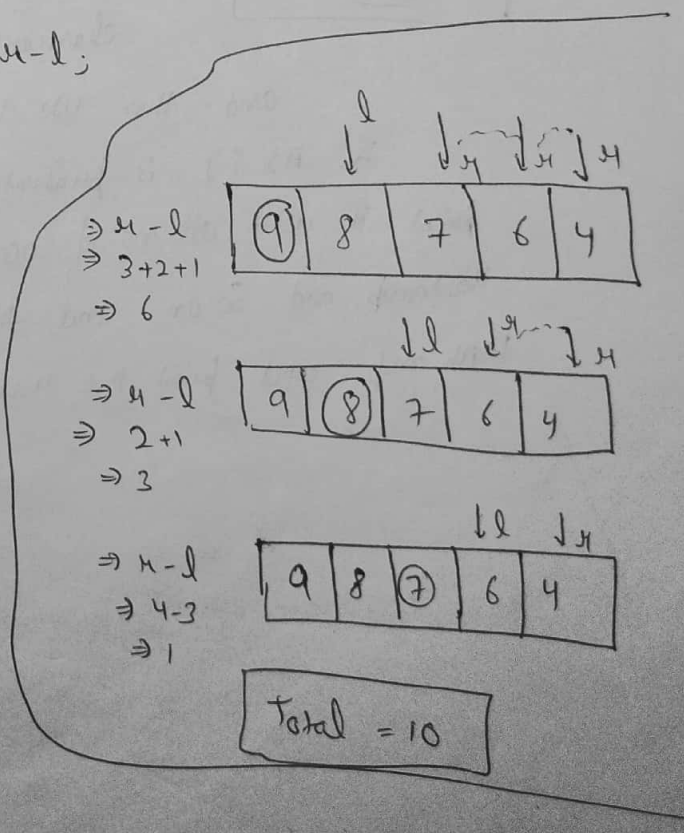
So, **Approach -1** :- In this approach we will simply use three Nested loops.

```
int Maximum_Number_of_Triangles (struct Array arr)
{
    int Count = 0;
    for (int i = 0; i < arr.length - 2; i++)
    {
        for (int j = i+1; j < arr.length - 1; j++)
        {
            for (int k = j+1; k < arr.length; k++)
            {
                if (arr.A[i] + arr.A[j] > arr.A[k] &&
                    arr.A[j] + arr.A[k] > arr.A[i] &&
                    arr.A[i] + arr.A[k] > arr.A[j])
                    Count++;
            }
        }
    }
    return Count;
}
```

Time :- $O(n^3)$

Approach-2 :- In this Approach, We will see an optimal Solution of $O(n^2)$, In this approach we will sort the array in descending order, then fix one larger element in out most loop in every iteration and take two pointers as shown.

```
int Maximum_Number_of_Triangles(struct Array arr)
{
    int Count = 0;
    for (int i=0 ; i< arr.length - 2 ; i++)
        {
            int l = i+1;
            int r = arr.length -1;
            While ( l <= r)
            {
                if (arr.A[l] + arr.A[r] > arr.A[i])
                    {
                        Count += r-l;
                        r--;
                    }
                else
                    l++;
            }
        }
    return Count;
}
```



(11) Sort an Array According to other :- Given two Integer arrays A1[] and A2[] of size N and M respectively. Sort the first array A1[] such that all the relative positions of the elements in the first Array are the same as the elements in the second array A2[]

Note :- If elements are repeated in the second array, Consider their first occurence only.

Example :- Input :- N = 11
M = 4
A1[] = {2,1,2,5,7,1,9,3,6,8,8}
A2[] = {2,1,8,3}

output :- 2 2 1 1 8 8 3 5 6 7 9

⇒ Approach - 1 :- We can use Hashmap, we will store every element in Hashmap for A1 with its frequency and then we will check for A2[] that an element in A2[] is present in A1 or not if it is present then print it with all no. of occurances and make it count as zero in Hashmap and so on and then after this we will scan through Hash map and print the remaining elements.

```
void Sort1 (struct Array *arr1, struct Array *arr2)
{
    int max = 0;
    for (int i = 0; i < arr1 -> length ; i++)
    {
        if (max < arr1 -> A[i])
    }       max = arr1 -> A[i];
    Struct Array hash;
    hash. length = hash. size = max+1;
    hash. A = (int *) malloc (hash. size * (sizeof (int)));
    for (int i=0; i< hash. length ; i++)
    {
        hash. A[i] = 0;
    }
    for (int i=0; i< arr1 -> length ; i++)
    {
        hash. A[arr1 -> A[i]]++;
    }
    for (int i=0; i< arr2 -> length ; i++)
    {
        if (hash. A [arr2 -> A [i]] > 0)
        {
            for (int j = 0; j< hash. A [arr2 -> A[i]]; j++)
            {
                printf ("%d _", arr2 -> A[i]);
            }
            hash. A [arr2 -> A[i]] = 0;
        }
    }
    for (int i =0; i < hash. length ; i++)
    {
        if (hash. A[i] > 0)
        {
            for (int j=0 ; j< hash. A [i] ; j++)
            {
                printf ("%d _", i);
            }
        }
        hash. A[i] = 0;
    }
```

⇒ **Approach - 2** :- Use Sorting and Binary Search

temp

2	1	2	5	7	1	9	3	6	8	8

A_2

2	1	8	3

Vis

ø1	ø1	ø1	ø1	0	0	0	0	ø1	ø1	0

Sorted temp

1	1	2	2	3	5	6	7	8	8	9

```
Void Sort (int A[], int n)
 {
    for (int i=0 ; i <n-1 ; i++)
      {
        int flag =0;
        for(int j= 0; j <n -1-i ; j ++)
          {
            if (A[j] > A[j+1])
              {
                swap (&A[j], &A[j+1])
                flag =1;
              }
          }
        if (flag ==0)
          break;
      } }
int Search (int A[], int start, int end, int key)
  {
    while (end > start)
      { int mid = (start + end)/2;
        if (A[mid] ==key)
            return mid;
        else if ( key > A[mid])
            return Search (A, mid +1, end, key);
        else if (key< A[mid])
            return Search (A, start, mid-1, key);
  } }
```

```
void SortAccording(int A[], int B[], int m, int n)
{
    int tamp[n], visited[n];
    for (int i=0; i<n; i++)
    {
        tamp[i] = A[i];
        visited[i] = 0;
    }
    Sort(tamp, n);
    int ind = 0;
    for (int i=0; i<m; i++)
    {
        int f = search(tamp, 0, n-1, B[i]);
        if (f==-1)
            continue;
        for (int j=f; (j<n && tamp[j]==B[i]); j++)
        {
            A[ind++] = tamp[j];
            visited[j] = 1;
        }
    }
    for (int i=0; i<n; i++)
    {
        if (visited[i]==0)
            A[ind++] = tamp[i];
    }
    for (int i=0; i<n; i++)
    {
        printf("%d ", A[i]);
    }
}
```

(12)

Minimum Platforms :- Given arrival and departure times of all trains that reach a railway station. Find the minimum No. of platforms required for the railway station so that no train is kept waiting.

Consider that all the trains arrive on the same day and leave on the same day. Arrival and departure time can never be the same for a train but we can have arrival time of one train equal to departure time of the other. At any given instance of time, same platform can not be used for both departure of a train and arrival of another train. In such cases we need different platforms

Example :- Input, n = 6

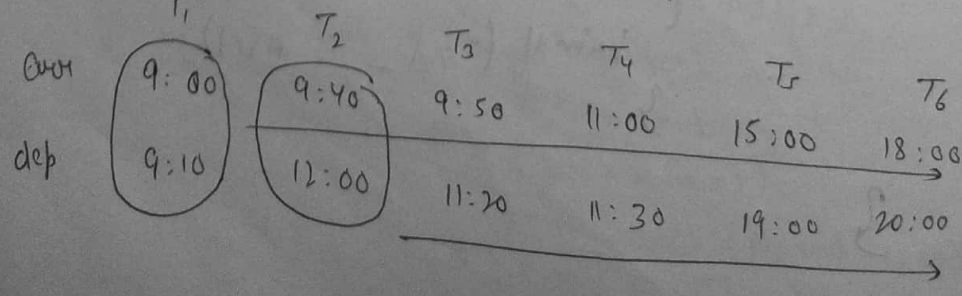
arr[] = { 0900, 0940, 0950, 1100, 1500, 1800 }

dep[] = { 0910, 1200, 1120, 1130, 1900, 2000 }

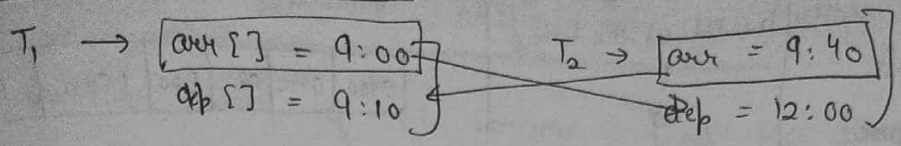
Output = 3

Explanation :- Minimum 3 platforms are required to safely arrive and depart all trains

⇒ **Approach - I** :- In this approach, we can simply one Train at a time and compare it with other trains time, if it coincide, then we must need a platform, for this we can use two nested loops.



Suppose Take T_1 and T_2

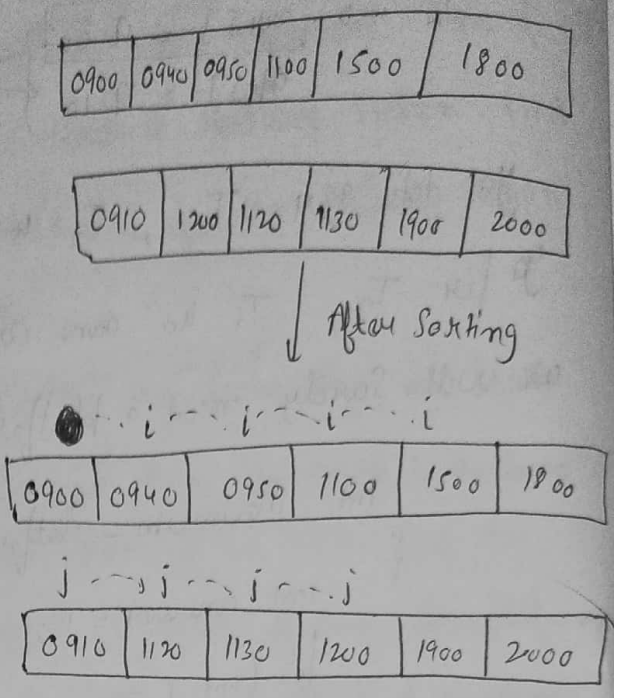


ya toh agar T_1, T_2 ke Jane ya aane ke beech mein aaye ya fir T_2, T_1 ke aane aur Jaane ke beech main aaye, then we will surely need a platform.

```
int Maximum_Platforms (struct Array arr1, struct Array arr2)
{
    int answer = 1;
    for (int i = 0; i < arr.length - 1; i++)
    {
        int platform_needed = 1;
        for (int j = i+1; j < arr.length; j++)
        {
            if (arr1.A[i] >= arr1.A[j] && arr1.A[i] <= arr2.A[j]
                || arr1.A[j] >= arr1.A[i] && arr1.A[j] <= arr2.A[i])
                platform_needed++;
        }
        answer = max (answer, platform_needed);
    }
    Return answer;
}
```

⇒ **Approach - 2** :- In this approach, we will sort the arrival and departure of all the trains, Total platforms at any time can be obtained by subtracting total departures from total arrivals by that that time

Time	Event Type	Total Platform needed at this time
9:00	Arrival	1
9:10	Departure	0
9:40	Arrival	1
9:50	Arrival	2
11:00	Arrival	3
11:20	Departure	2
11:30	Departure	1
12:00	Departure	0
15:00	Arrival	1
18:00	Arrival	2
19:00	Departure	1
20:00	Departure	0




```
int Minimum_Platform (Struct Array arr1, Struct Array arr2)
{
    Sort (arr1);
    Sort (arr2);

    int Platform_needed = 1, result = 1;
    int i = 1, j = 0;

    while (i < arr1.length && j < arr2.length)
    {
        if (arr1.A[i] <= arr2.A[j])
            Platform_needed ++;
            i++;
        }
        else if (arr1.A[i] > arr2.A[j])
        {
            Platform_needed --;
            j--;
        }
        if (Platform_need > result)
            result = Platform_needed;
    }
    return result;
}
```

(13) Longest Common Prefix in an Array :-

Given a array of N strings, find the longest Common Prefix among all strings present inthe array

Example :- N = 4

arr[] = { geeksforgeeks, geeks, geek, geezer }

output : gee

Explanation : "gee" is the longest common prefix in all the given strings.

```
char * findPrefix ( char * Prefix, char * check)
{
  int length1 = 0;
  int length2 = 0;
  char * Prefix_1 ;
  Prefix_1 = (char *) malloc (100 * sizeof(char));
  for(int i =0; Prefix[i] != '\0' ; i++)
     length1 ++;
  for (int i=1 ; check[i] != '\0' ; i++)
     length2 ++;
  int i, j ;
for(i=0, j=0 ; i<length1-1 && j< length2-1;
                                  i++, j++)
  {
    if (Prefix[i] != check[i])
       break ;
    Prefix_1[i] = Prefix[i];
  }
return Prefix_1 ;
}
```

```
char * Common_Prefix (char * exp[], int n)
{
   char * Prefix = exp[0];
   for (int i=1 ; i<n ; i++)
   {
     Prefix = findPrefix ( Prefix, exp[i]);
   }
   return Prefix;
}
```

Note :- Input for array of strings :-

```
int n=4
char *exp[n] ;
for (int i=0 ; i<n ; i++)
{ exp[i] = malloc (100 * sizeof (char));
  printf ("Enter word: ");
  scanf ("%99s", exp[i]);
}
```

↓ This is to overcome overflow.

(14) Roman [redacted] Number to Integer :- Given a string in Roman no format (s). your task is to Convert it into an integer. Various Symbols and their values given below.

I 1
V 5
X 10
L 50
C 100
D 500
M 1000

⇒ I Place before V or X indicates one less, So four is IV (one less than 5) and 9 is IX (one less than 10).

⇒ Similarly XL = 40 ; L = 50
XC = 90 ; C = 100
CD = 400 ; D = 500
CM = 900 ; M = 1000

Algorithm :-
- Split the Roman Numeral string into Roman Symbols
- Convert each Symbol of Roman Numerals into the value it represents.
- Take the Symbol one by one from starting from index 0.

① If the current value of Symbol is greater than or equal to the value of next Symbol, than add this value to running total.
② else Substract this value by adding the value of next Symbol to running total.

Code :-

```
int roman_to_int (char * string)
{
    int answer = 0;
  for (int i=0 ; string[i] != '\0' ; i++)
    {
      int length = strlen (string);
      int s1 = value (string[i]);
        if (i+1 < length)
        {
          int s2 = value (string[i+1]);
            if (s1 >= s2)
            answer = answer + s1;
            else
            {
            answer = answer + s2 - s1;
              i++;
            }
        }
      else
      answer = answer + s1;
    }
  return answer;
}
```

```
int value (char x)
{
    if (x == 'I')
      return 1;
    if (x == 'V')
      return 5;
    if (x == 'X')
      return 10;
    if (x == 'L')
      return 50;
    if (x == 'C')
      return 100;
    if (x == 'D')
      return 500;
    if (x == 'M')
      return 1000;
    return -1;
}
```

(15) **Find the Element that Appears once** :- Given a Sorted Array A[] of N positive integers having all the numbers occuring exactly twice, except for one number which will occur only once. Find the number occuring only once.

Example - 1 :- Input : N = 5
A = {1, 1, 2, 5, 5}

output : 2

Explanation :- Since 2 occurs once, while other numbers occur twice, 2 is the answer.

⇒ **Approach - 1** :- We can use Hashmap and will see where the Counter is equal to 1. (ie. 1 < Count < 2)

→ **Approach - 2** :- As it is Sorted array, we can traverse from right to left and can easily figure out required element.

```
int ans = -1;
for (int i = 0; i < n; i += 2)
{
    if (arr[i] != arr[i+1])
    {
        ans = arr[i];
    } break;
}
if (arr[n-2] != arr[n-1])
    ans = arr[n-1];
printf("%d", ans);
```

⇒ Approach-3 :-

We know there is a property of XOR

(i) $n \wedge n = 0$

(ii) $0 \wedge n = n$

We will use this property

So, Lets suppose $A = \{5, 4, 1, 4, 3, 5, 1\}$

int res = 0

res = $5 \wedge 4 \wedge 1 \wedge 4$ [$\because n \wedge n = 0$]

= $5 \wedge 1 \wedge 3 \wedge 5$

= $4 \wedge 3 \wedge 4$ [$\because n \wedge 0 = n$]

= 3

```
int main ()
{
    int A[] = { 5, 4, 1, 3, 3, 5, 1 };
    int res = 0;
    for (int i = 0 ; i < 7 ; i++)
    {
        res = res ^ A[i];
    }
    printf ("%d ", res);
}
```

(18) **Array Pair Sum Divisibility Problem** :- Given an array of integers and a number k, write a function that returns true if given array can be divided into pairs such that sum of every pair is divisible by k.

Example :- Input : arr = [9, 5, 7, 3], k = 6

Output : True

Explanation : {(9, 3), (5, 7)} is a possible Solution.

9+3 = 12 is divisible by 6 as well as 5+7 = 12

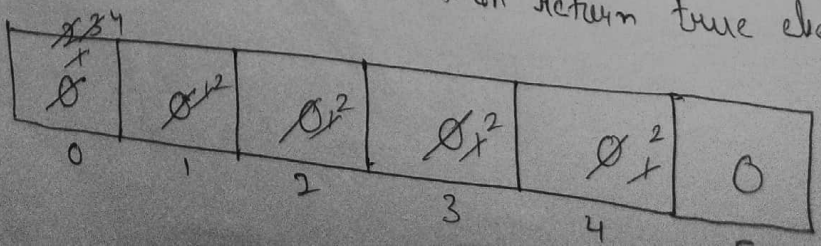
⟹ The first thing, if n is odd, then we can't make pairs, so in that case return false, Now, let us understand a case

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20

Now, Take a hashmap equal to size of 'k', Now, we can see that n is even and pairs can be (1,4) (2,3) (5,10) (6,9) (7,8) (15,20)

So, if we can see (1,4) → 1 % 5 = 1, 4 % 5 = 4] = 5 { 2 % 5 = 2, 3 % 5 = 3] 5 { 5 % 5 = 0, 10 % 5 = 0

So, what we will do, we will increment counter in hashmap for all the remainders and will check like for (1,4) it is (1, k-1) (2, k-2) if the count for these value are same than return true else false



```
bool Pair_Sum (int A[] , int n, int k)
{
    int hash[k];
    for (int i=0 ; i<k ;i++)
    {
        hash[i] = 0;
    }
    for (int i=0 ; i<n; i++)
    {
        hash [ (A[i] +k) %k]++ ;  // To Handle negative Cases
    }
    if (hash [0] %2 != 0)  return false;
    for (int i=1 ; i<k ; i++)
    {
        if (hash [i] != hash [k-i])
        {
            return false;
        }
    }
    return true;
}
```

(17) **SubArrays With Sum k** :- Given an Array Containing N integers and a positive integer k, find the length of the longest Sub Array with the Sum of the elements divisible by the given value k.

Example 1 :- Input : A[] = {2, 7, 6, 1, 4, 5}

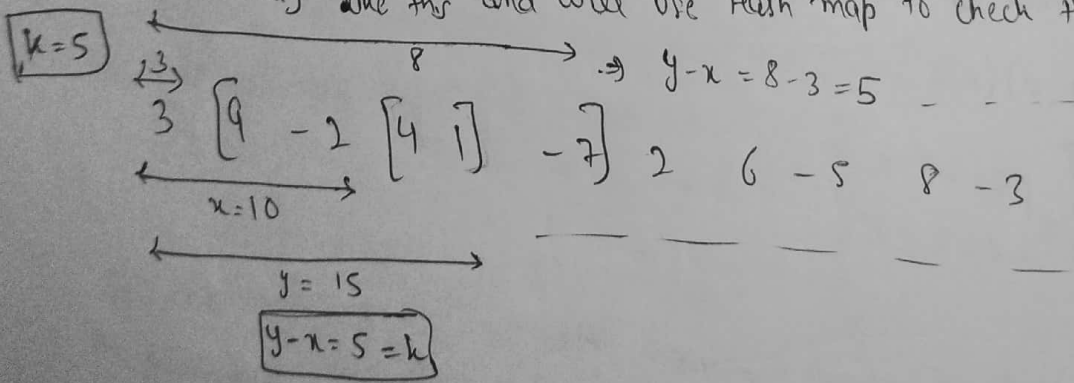
k = 3

Output : 4

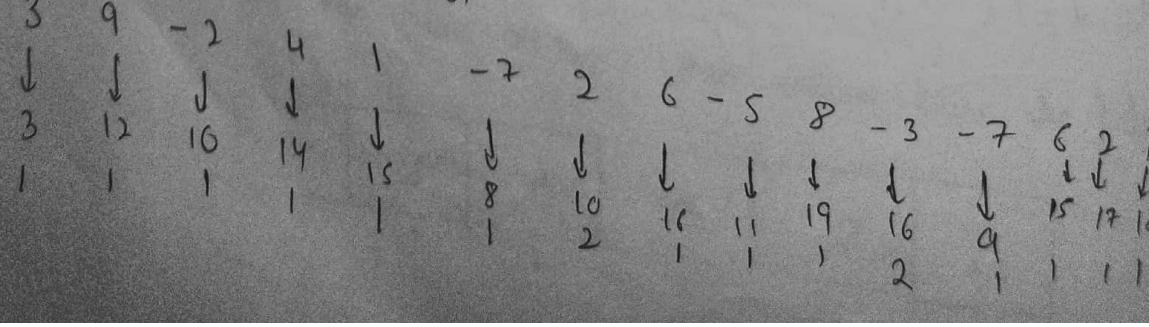
Explanation :- The SubArray is {7, 6, 1, 4} with Sum 18, which is divisible by 3

⇒ **Approach-1** :- In this approach we can just use two nested loops and if we get sum equals to required sum than we will just simply increase the SubArray Counter.

⇒ **Approach-2** :- In this approach, we will see that we will increase the Counter in such a way by using prefix sum array like this and will use Hash map to check this difference.



So, this is our approach



Sub Arrays in this Case :-

① (4, 1) ② (9, -2, 4, 1, -7) ③ (1, -7, 2, 6, -5, 8) ④ (8, -3)

⑤ (4, +1, -7, 2, 6, -5, 8, -3, -7, 6) ⑥ (6, -5, 8, -3, -7, 6)

⑦ (-7, 4, 1, -7, 2, 6, -5, 8, -3, -7, 6, 2)

```
int SubArrays_with_Sum (int A[], int n, int k)
{
    int hash[100] = {0};
    int res = 0;
    int curr_sum = 0;
    for (int i=0 ; i<n; i++)
    {
        curr_sum = curr_sum + A[i];
        if (curr_sum == k)
            res++;
        if (hash[curr_sum - k] > 0)
        {
            res += hash[curr_sum - k];
            hash[curr_sum]++;
        }
        else
            hash[curr_sum] ++;
    }
    return res;
}
```

(18) **Longest SubArray with Sum divisible by k** :- Given an array containing N integers and a positive integer k, find the length of the longest sub Array with sum of the elements divisible by given value k.

Example :- Input :- A[] = {2, 7, 6, 1, 4, 5}

k = 3

output :- 4

Explanation :- The SubArray is {7, 6, 1, 4} with sum 18, which is divisible by 3

⟹ Approach - 1 :- By using two nested loops

```
int Basic_Longest (int A[], int n, int k)
{
    int max_length = 0;
    for (int i=0 ; i<n-1 ; i++)
    {
        int Sum = 0;
        for (int j = i ; j<n; j++)
        {
            Sum = Sum + A[j];
            if (Sum%k == 0)
            {
                int length = j-i+1;
                if (max_length < length)
                max_length = length;
            }
        }
    }
    return max_length;
}
```

Approach-2 :- Using Hashmap :- Will store remainders & whenever remainder is same sum upto that will be longest

mod_arr

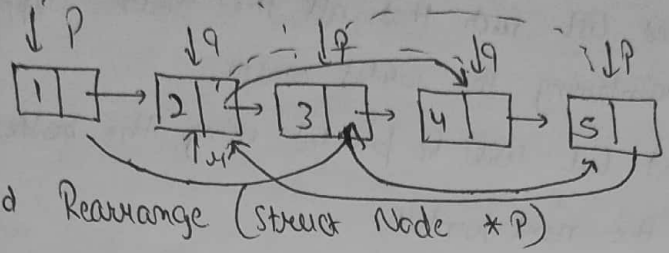
2	0	0	1	2	1
0	1	2	3	4	5

```
int Advanced_longest (int A[], int n, int k)
{
    int mod_arr[n];
    int max = 0;
    int ind;
    int Curr_Sum = 0;
    for (int i=0 ; i<n ; i++)
    {
        Curr_Sum = Curr_Sum + A[i];
        mod_arr[i] = (((Curr_Sum % k) + k) % k); // for negative input
    }
    for (int i=0 ; i<n ; i++)
    {
        if (mod_arr[i] == 0)
            max = i+1;
        else
        {
            for (int j= n-1 ; j >=0 ; j--)
            {
                if (mod_arr[i] == mod_arr[j])
                {
                    ind = j - i;
                } break;
            }
        }
        if (ind > max)
            max = ind;
    }
    return max;
}
```

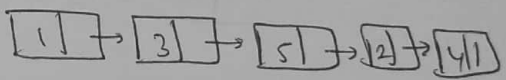
(19) **Rearrange a Linked List** :- Given a Singly Linked List, the task is to re-arrange it in a way that all odd position nodes are together and all even positions node are together.

Example 1 :- Input : 1 → 2 → 3 → 4 → 5

Output : 1 → 3 → 5 → 2 → 4




```
Void Rearrange (Struct Node *P)
{
    Struct Node *q = P→next;
    Struct Node *H = P→next; // it is used to connect all odd
                             position nodes with even positioned
    while (1)
    {
        if (q→next == NULL)
        {
            P→next = H;
            break;
        }
        P→next = q→next;
            P = q→next;
        if (P→next == NULL)
            {
                q→next = NULL;
                P→next = H;
            } break;

        q→next = P→next;
        q = P→next;
    }
}
```



(20) Given a Linked List of Size N, where every Node represents a Sub-Linked-list and Contains two pointers

(i) a next pointer to next node

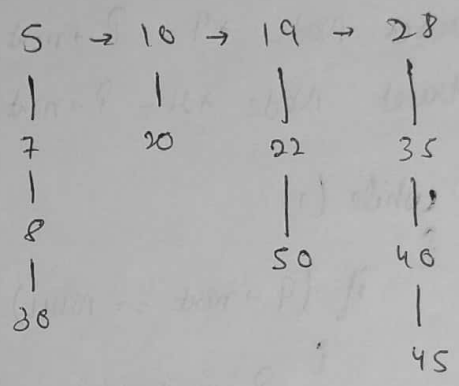
(ii) a bottom pointer to a linked list where this Node is head.

Each of the Sub Linked list is in Sorted order.

Flatten the linked list such that all the nodes appear in a single level while maintaining the sorted order.

Note :- The flattened List will be printed using the bottom pointer instead of the next pointer.

Example :- Input :-



Output :- 5 → 7 → 8 → 10 → 19 → 20 → 22 → 28 → 30 → 35 → 40 → 45 → 50

⇒ In this Ques, we will also understand, how to create that type of linked list and use Merge sort for ~~this~~ linked list Concept.

```
# include <stdio.h>
# include <stdlib.h>
# define SIZE(arr) (sizeof(arr) / sizeof(arr[0]))
Struct Node
{
    int data;
    Struct Node * next;
    Struct Node * down;
} * first = NULL;
```

```
void Push (struct Node ** headRef, int data)
{
    struct Node * newNode;
    newNode = (struct Node *) malloc (sizeof (struct Node));
    newNode → data = data;
    newNode → next = NULL;
    newNode → down = * headRef;
    * headRef = newNode;
}

void CreateVerticalList (struct Node ** head, int arr[], int n)
{
    for (int i=0; i<n; i++)
        Push (head, arr[i]);
}

int main ()
{
    int arr1[] = {8, 6, 4, 1};
    int arr2[] = {7, 3, 2};
    int arr3[] = {9, 5};
    int arr4[] = {12, 11, 10};
    CreateVerticalList (&(first), arr1, SIZE (arr1));
    CreateVerticalList (&(first → next), arr2, SIZE (arr2));
    CreateVerticalList (&(first → next → next), arr3, SIZE (arr3));
    CreateVerticalList (&(first → next → next → next), arr4, SIZE (arr4));
    void flatten (struct Node **first);
    flatten (&first);
    void display (struct Node * p);
    display (first);
    return 0;
}
```

```
Struct Node * Merge (Struct Node * a, Struct Node
{
    Struct Node * answer = NULL;
    if (a == NULL)
        return b;
    else if (b == NULL)
        return a;
    if (a -> data <= b -> data)
    {
        answer = a;
        answer -> down = Merge (a -> down, b);
    }
    else
    {
        answer = b;
        answer -> down = Merge (a, b -> down);
    }
    answer -> next = NULL;
    return answer;
}

Void flatten (Struct Node ** first)
{
    Struct Node * temp = * first;
    Struct Node * temp1 = temp -> next;
    if (temp == NULL || temp -> next == NULL)
        return;
    flatten (& temp1);
    * first = Merge (temp, temp -> right);
}
```

(22)

Stock Span Problem :-

The stock span Problem is a financial problem where we have a series of n daily price quotes for a stock and we need to calculate the span of stock price for all n days. The span Si of the stocks price on a given day I is defined as the maximum number of consecutive days just before the given day, for which the price of the stock on the current day is less than or equal to its price on the given day.

For Example, if an array of 7 days prices is given as {100, 80, 60, 70, 60, 75, 85}, then the span values for corresponding 7 days are {1, 1, 1, 2, 1, 4, 6}.

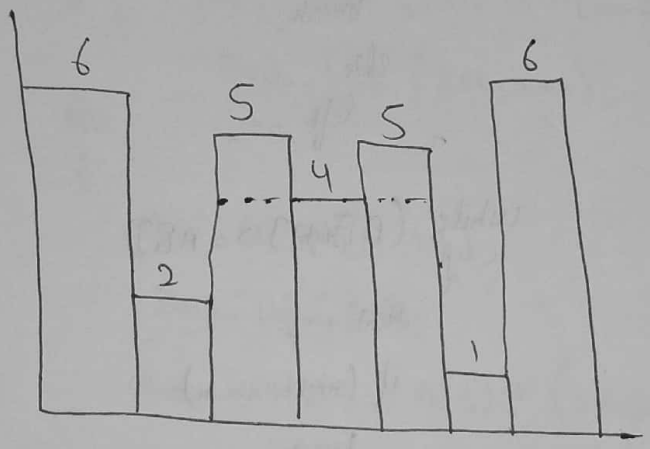
```
int B[n];
Void Stock_Spain (int A[], int n)
{
    int j=1;
    push(A[0]);
    for (int i=1 ; i<n; i++)
    {
        int span = 1;
        if (top → data > A[i])
        {
            B[i++] = span;
            push(A[i]);
        }
        else
        {
            struct Node *t = top;
            while (t → data < A[i])
            {
                t = t → next;
                span++;
            }
            B[i++] = span;
            push(A[i]);
    } } }
```

(23) **Maximum Rectangular Area in a Histogram** :- Find the largest rectangular area possible in a given histogram where the largest rectangle can be made of a number of contigous bars. For simplicity, assume that all bars have the same width and width is 1 unit, there will be N bars height of each bar will be given by the Array arr.

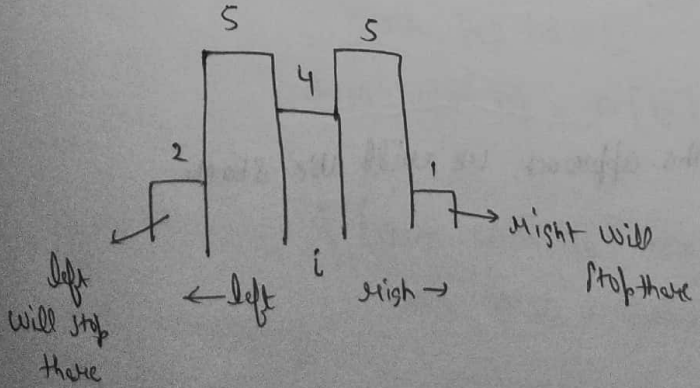
Example :- Input : N = 7, Arr[] = {6, 2, 5, 4, 5, 1, 6}

Output : 12

Explanation :-



⟹ **Approach - 1** :- In this approach, we will see the Daugas buildings on the left and right for all the indices by running a for loop and calculate the area for every bar and which one is max., will stored and return it.



area = A[i] * (right - left - 1);

```
int maximum (int A[], int n)
{
    int max = -1;
    for (int i=0; i<n; i++)
    {
        int curr_area = 0;
        int left = i;
        int right = i;
        while (A[left] >= A[i])
        {
            if (left == 0)
                break;
            else
                left--;
        }
        while (A[right] >= A[i])
        {
            right++;
            if (right == n)
                break;
        }
        curr_area = curr_area + (A[i] * (right - left - i));
        if (max < curr_area)
            max = curr_area;
    }
    return max;
}
```

⇒ **Approach - 2** :- In this approach, we will use stack

```
int Maximum_Area_Histogram (int A[], int n)
{
    Struct Stack st;
    Create Stack (&st, 100);
    int max_area = 0;
    int tp;
    int area_with_top = 0;
    int i = 0;
    while (i<n)
    {
        if ( isEmpty_Stack (&st) || A[St.Q[st.top]] <= A[i])
            Stack_push (&st, i++);
        else
        {
            tp = St.Q [st.top];
            Stack_pop (&st);
            area_with_top = A[tp] * (isEmpty_Stack (&st) ? i :
                                     i - St.Q[st.top] - 1);
            if (area_with_top > max_area)
                max_area = area_with_top;
        }
    }
    while (! isEmpty_Stack (&st))
    {
        tp = St.Q [st.top];
        Stack_pop (&st);
        area_with_top = A [tp] * (isEmpty_Stack(&st) ? i :
                                  i - St.Q[st.top] - 1);
        if (area_with_top > max_area)
            max_area = area_with_top;
    }
    return max_area;
}
```

(24) 132 Creeky Buildings :- There are N buildings in Linear land, They appear in a Linear line one after the other and their heights are given inthe array arr[]. Greek want to select three buildings in Linear land and remodel them as recreational spots. The third of the selected Building must be taller than the first and shorter than second. Can Greek build the three-building recreational zone?

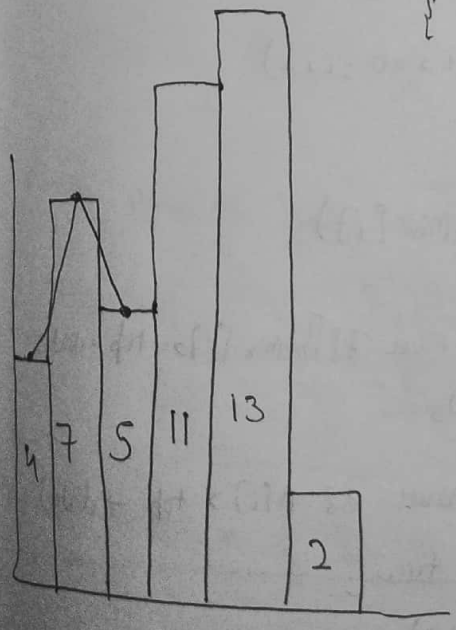
Example :- N = 6

arr[] = {4, 7, 11, 5, 13, 2}

output :- True

Explanation :- [4, 7, 5] fits the Condition.

⇒ Approach 1 :- By running 3 nested loops, simple




```
bool Creeky_buildings (int A[], int n)
{
    for (int i=0 ; i<n-2 ; i++)
    {
        for (int j = i+1 ; j<n-1 ; j++)
        {
            for (int k= j+1 ; k<n ; k++)
            {
                if (A[i] < A[j] && A[k] > A[i]
                    && A[k] < A[j])
                    return true;
            }
        }
    }
    return false;
}
```

⇒ Approach-2 :- Using Stack, while Using Stacke we have to see this Condition is [redacted]

a < c
c > b
b > a

So, To Do this, we will take an auxillary array in which we will store the height of building (ie. smaller when Comparing with all other) and take Stack with implemantation as below.

```
bool Greedy_Buildings (int A[], int n)
{
    if (n<3)
      return false;
    int PreMin[n];
    PreMin[0] = A[0];
    for (int i=1 ; i<n ; i++)
    {
      PreMin[i] = min (PreMin[i-1], A[i]);
    }
    for (int i=n-1 ; i>=0 ; i++)
    {
      int x=0;
      if (A[i] > PreMin[i])
      {
        while (top!=NULL && PreMin[i]>=top->data)
          x=POP();
        if (top!=NULL && A[i]> top->data)
          return true;
      } Push (A[i]);
    }
    return false;
}
```

BIT MASKING

This topic is very imp. bcz our Computer understands Binary language (0 & 1) and we work in decimal no., so to fast our Computation we will directly use Bit operations to reduce time Complexity.

Bitwise operators :-

1. & (And)
2. ^ (XOR)
3. | (OR)
4. ~ (Invert)

For a Decimal no. :- $(274)_{10} \rightarrow \overbrace{2 \times 10^2}^{200} + \overbrace{7 \times 10^1}^{70} + \overbrace{4 \times 10^0}^{4} = 274$

For a Binary no. :- $(101)_2 \rightarrow \underbrace{1 \times 2^2}_{4} + \underbrace{0 \times 2^1}_{0} + \underbrace{1 \times 2^0}_{1} = 5$

So, Binary representation of 5 is 101

Now, How to find a Binary representation of a binary no.

2	14	0
2	7	1
2	3	1
	1	

$\Rightarrow$ So, $(14)_{10} = (1110)_2$

Now, Addition of Binary no. :-

$$\begin{array}{r} {\scriptstyle 0\ \ 1\ \ 0} \\ 1\ 0\ 1 \\ +\ 1\ 1\ 1 \\ \hline 1\ 1\ 0\ 0 \end{array}$$

Now for Subtraction of a binary no.:- $\begin{array}{r} 9 \\ -\,7 \\ \hline 2 \end{array}$

We can write it as $9 + (-7)$

So, we will find Negative inverse of 7 and then add it with 9 to find Subtraction of 9

and we can find negative inverse by using 2's Complement.

(1) Invert all bits

(2) Add one

For eg:- $5 \rightarrow 101$

$101 \rightarrow 010$

$+\,1$

$011 \rightarrow -5$

So, Now, $12 - 5 = 7$

$$\begin{array}{r} 1100 \\ +\,1011 \\ \hline (111)_2 = 7 \end{array}$$

Ye Wale

(1) aaya bcz $(5)_2 = 101$

but it is 0000--101

So, its inverse will be

111--011

thats why we take 2's Complement of 5 as 1011

★ Bit Wise operators :-

① and (&), or (|), XOR (^), inverse (1's complement) (~)

a	b	&	|	^
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

~ : It will inverse all the bits of that digit

$5 \rightarrow 101$

$\sim 5 \rightarrow 010$

Note :- For ^ :- Jab dono bits same hogi, then it will be 0 else 1.

Now, >> :- 12 >>2 (shifting bits of 12 two times right side

12 → 1100

After shifting → 0011 → 3

Now, << :- 12 <<2 (shifting bits of 12 two times left side)

12 → 1100

After shifting :- 110000 → 48

```
int a = 5;
int b = a>>1; // b = 2 (whenever you shift by 1, >>1, it means you always divide this no. by 2).
```

5 → 101

After shift → 010 = 2

After shift → 001 = 1

After shift → 000 = 0

Note :- Whenever we want to divide a no. by 2, we can do >>1 for faster Computation. (mainly in loops).

Now,

```
int a = 3
int b = a<<1; // b = 6 (whenever you shift left by 1, <<1, it means you always multiply this no. by 2)
```

3 → 011

After shift → 110 = 6

After shift → 1100 = 12

Imp Note :- Whenever want to multiply a no. by 2, we can do <<1 for faster Computation. (mainly in loops).

Imp Note :- Whenever we want to check a no. is odd/even than we use % operation, but for fast Computation, we can use masking like with 1, so if (a&1 == 1) than it will be odd and if (a&1 == 0) than it will be even.

Now for Swapping we can do :-

```
int a = 5;
int b = 7;
```

a = a ^ b = 2

b = 0 ^ b = 5

a = a ^ b = 7

```
  101
^ 111
-----
  010 → 2
^ 111
-----
  101 → 5
```

```
  010
^ 101
-----
  111
-----
```

Note :- We can swap two no. without using any third variable ;

★ Find i^{th} bit :-

$n \rightarrow 1\ 0\ 0\ \underset{5^{th}}{1}\ 1\ 0\ 1\ 0\ 1$

mask → 0 0 0 1 0 0 0 0

n & mask → Non zero → 1

↳ 0

So, to find i^{th} bit is 0 or 1, then we can use masking

mask = 1 << i

if (n & mask == 0) then i^{th} bit = 0

else i^{th} bit = 1

★ Set i^{th} bit :-

$n \rightarrow$ (8) 1 (7) 0 (6) 0 (5) 1 (4) 1 (3) 0 (2) 1 (1) 0 (0) 1

mask = 3 << i

mask → 0 0 0 0 0 1 0 0 0

Now (n | mask == 1) | Operator will always set 0th bit to 1

☆ clear ith bit :- To make a bit 1 to 0

	8	7	6	5	4	3	2	1	0
n →	1	0	0	1	1	0	1	0	1
(~(mask)) → &	1	1	1	1	0	1	1	1	1
	1	0	0	1	0	0	1	0	1

Now, We have to find mask, so, first shift i times

mask = 1 << i

then invert it to operate & operation ~(mask)

⇒ Find number of bits to change to convert a to b.

→

a → 1 0 1 1 0

b → 1 1 0 1 1

111 → 3

Now, we'll do XOR their a ^ b → 0 1 1 0 1, Now we have to see how many times 1 occur to get no. of bits which are not same (or we can how many set bits are there)

```
int a = 13;
a = a>>1
```

→

```
while (a != 0);
{
    a = a >> 1
    Count ++;
}
```

Time Complexity :- depending upon no. of bits (ie. O(n))

Optimized Solution :- we'll use a trick `n & (n-1)`, it will make least significant bit as 0. (ie. Set bit)

1100 → 1000 → 0100 → 0000

Now, iterations are less bcz time Complexity depends upon Set bits only.

☆ Find the two non-repeating elements in an array where every element repeats twice

$A = \{5, 4, 1, 4, 3, 5, 1, 2\}$

$res = 5 \wedge 4 \wedge 1 \wedge 4$

$= 5 \wedge 1 \wedge 3 \wedge 5$

$= 4 \wedge 3 \wedge 4 \wedge 2$

$= 3 \wedge 2$

Now, $res = 3 \wedge 2$ (these are two non-repeating elements)

Now, $3 \wedge 2 \Rightarrow$

$$\begin{array}{r} 1\ 1 \\ 1\ 0 \\ \hline 0\ 1 \end{array} \longrightarrow 0^{th} \text{ bit is one.}$$

In these ques, we have to check where the right most bit is 1 or at which position and then divide the array into two buckets having 0 bit and 1 bit at that position and take XOR for both buckets and remaining will be the two answers.

```
int main()
{
int A[] = {5,4,1,3,3,5,1,2};
int res = 0;
for (int i = 0; i < 8; i++)
    res = res ^ A[i];
int Pos = (res & ~(res - 1));
int temp = res;
int b[10] = {0}, c[10] = {0}
int mask = 1 << (Pos - 1);
```

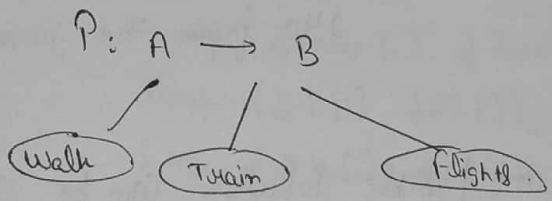
```
for (int i = 0; i < 8; i++)
{
  if ((A[i] & mask))
    b[i] = A[i];
  else
    c[i] = A[i];
}
int ans1 = 0, ans2 = 0
for (int i = 0; i < 10; i++)
{
  ans1 = ans1 ^ res ^ c[i];
  res = 0;
}
for (int i = 0; i < 10; i++)
{
  ans2 = ans2 ^ temp ^ b[i];
```

Result

GREEDY ALGORITHMS

It is one of the strategy or approach to solve problems, this method is to used to solve optimize problems, ie. minimum result or maximum result.

like we have to travel from A to B, we can travel by walk, car, train or flights but with minimum cost or with in given 10 hours.



So, a solution which is satisfying the conditions will be a feasible solution though there may be many solutions. now, we can go via train or flight but with train their will be a minimum cost so it will be a optimal solution and there will be only one optimal solution.

Rough Idea :-

```
Algorithm Greedy (a, n)
{
    for (i=1 to n) do
    {
        x = Select (a);
        if x = feasible then
            Solution = Solution + x;
    }
}
```

n=5

a | a_1 | a_2 | a_3 | a_4 | a_5 |
1 2 3 4 5

★ **Knapsack Problem** :- Given weights and value of N items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

Example - 1 :- Input : N = 3, W = 50

Value[] = {60, 100, 120}

Weight[] = {10, 20, 30}

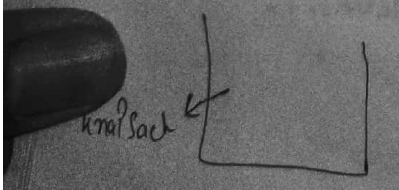
output : 240

Explanation :- Total maximum value of item we can have is 240 from the given capacity of sack

⟹ Now, one feasible solution can be, is, we can take those values which has maximum profit and put them in the bag and another feasible solution can be, is, we can take those values have least weight, so that we can make 3 more items.

But the optimal solution will be to choose maximum value by weight ratio items to get maximum total value and we can also take the products acc. to our need like if we want 5 kg from 10 kg weight given, we can take it.

Value →	60	100	120
Weight →	10	20	30
Value / Weight →	6	5	4



50 - 10 = 40
40 - 20 = 20
20 - 20 = 0

Weight = 10 + 20 + $\frac{2}{3}$ × 30 = (3) =

Value = 60 + 100 + $\frac{2}{3}$ × 120 = 240

```
float knapsack (float P[], float w[], int n, float m)
{
    float Ratio[n];
    for (int i=0 ; i< n ; i++)
    {
        Ratio[i] = (P[i] / w[i]);
    }
    for (int i=0 ; i< n-1 ; i++)
    {
        int flag = 0;
        for (int j = i+1 ; j< n ; j++)
        {
            if (Ratio[i] < Ratio[j])
            {
                swap (&Ratio[i], &Ratio[j]);
                swap (&P[i], &P[j]);
                swap (&w[i], &w[j]);
                flag = 1;
            }
        }
        if ( flag == 0)
            break;
    }
    float Profit = 0; float weight = 0; int measure = 0;
    while (weight + w[measure] <= m)
    {
        Profit = Profit + P[measure];
        weight = weight + w[measure++];
    }
    printf ("%f ", Profit);
    printf ("%f ", weight);
    float extra = ((m - weight) / (w[measure])) * P[measure];
    Profit = Profit + extra;
    return Profit;
}
```

★ **Coin Piles** :- There are N piles of coins each containing A_i ($1 <$ coins. Find the minimum number of coins to be removed such that the absolute difference of coins in any two piles is at most k.

Note :- You can also remove a pile by removing all the coins of that pile.

Example :- Input :- N=4, k=0

arr[] = {2, 2, 2, 2}

Output :- 0

Explanation :- For any two piles the difference in the number of coins is <=0. So no need to remove any coins.

⇒ Now, Acc. to Questions we have to find minimum no. of coins to be removed so that for any two piles we take the difference will atmost k, so will use two nested loops & Count by seeing the difference as follows :

```
int minimum-Coins (int A[], int n, int k)
{
    static int Count = 0;
    for (int i = 0; i < n-1; i++)
    {
        for (int j = i+1; j < n; j++)
        {
            if (A[j] > A[i])
            {
                int P = A[j] - (A[i] + k);
                if (P > 0)
                {
                    Count = Count + P;
                    A[j] = A[j] - P;
                }
            }
        }
    }
```

* **Minimize the Heights** :- Given an Array arr[] denoting heights of N towers and a positive integer k, you have to modify the height of each tower either by increasing or decreasing them by k only once. After modifying, height should be a non-negative integer.

Find out the minimum possible difference of the height of the shortest and longest towers after you have modified each tower.

Note :- It is Compulsory to increase or decrease (if possible) by k to each tower.

⇒ Example :- Input :- k = 2, N = 4

Arr[] = {1, 5, 8, 10}

Output :- 5

Explanation :- The array can be modified as {3, 3, 6, 8}

The difference b/w the largest & the Smallest is 8 - 3 = 5

⇒

```
int minimize (int A[], int n, int k)
{
    int Smallest = A[0]+k;
    int largest = A[n-1]-k;
    int mi, ma, m;
    int ans = A[n-1] - A[0];
    for (int i=1 ; i<n; i++)
    {
        mi = min (Smallest, A[i]-k);
        ma = max (largest, A[i-1]+k);
        if (mi<0)
            Continue;
        ans = min (ans, ma-mi);
    }
    return ans;
}
```

★ **Police and Theives** :- Given an array of size n such that each element Contains either a 'P' for policeman or a 'T' for the

find the maximum no. of theives that can be caught by the police. Keep in mind the following Conditions.

1) Each Policeman Can Catch only one theif.

2) A Policeman Cannot Catch a theif who is more than k units away from h

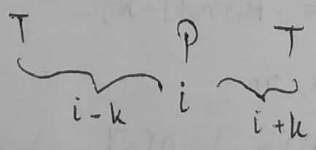
Example :- Input :- n = 5 , k = 1

arr[] = { P, T, T, P, T }

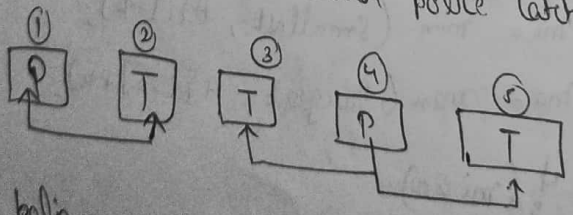
Output :- 2

Explanation :- Maximum 2 theives Can be caught. First policeman Catch first theif and second police man can Catch either second or third theif.

⇒ a Police can Cover k units from left & right of its position to Catch a theif



and if a police Catch a theif we will change 'T' to 'C' (ie Catch) and will Count ++ whenever police Catch a theif



policeman ① and ④ will Catch a theif and P_1 Can Catch a Theif 2 and P_4 Can Catch T_3 or T_2 anyone.

```
int catch_them(char A[], int n, int k)
{
    int ans = 0;
    for (int i=0 ; i<n ; i++)
    {
        int flag = 0;
        if (A[i] == 'P')
        {
            int j = max(0, i-k);
            for (int u=j ; u<i ; u++)
            {
                if (A[u] == 'T')
                {
                    A[u] = 'C';
                    ans++;
                    flag = 1;
                    break;
                }
            }
            if (flag == 0)
            {
                int p = min(i+k, n-1);
                for (int u=i+1 ; u<=p ; u++)
                {
                    if (A[u] == 'T')
                    {
                        A[u] = 'C';
                        ans++;
                        flag = 1;
                        break;
                    }
                }
            }
        }
    }
    return ans;
}
```

★ **Water The Plants** :- A Gallery with plants is divided into n parts, numbered : 0, 1, 2, 3, .. n-1. There are provisions for attaching water sprinklers at every partition. A sprinkler with range x at partition i can water all partitions from i-x to i+x.

Given an array Gallery consisting of n integers, where Gallery[i] is the range of sprinkler at partition i (power == -1) indicates no sprinkler attached, return the minimum number of sprinklers that need to be turned on to water the Complete Gallery.

If there is no possible way to water the full length using the given sprinklers, print -1.

⇒ To Do this We will traverse the whole array and will make all the indices 0, which can be sprinkled by sprinkler and at last will check if there is any -1, than return true else false

```
int sprinkler (int A[], int n)
{
    for (int i=0; i<n; i++)
    {
        if (A[i] != -1 && A[i] != 0)
        {
            int m;
            int p = max (A[i], i+A[i]);
            if (i - A[i] < 0)
            m = A[i];
            else
            m = min (A[i], i-A[i]);
            for (int j = i ; j <= p; j++)
                A[j] = 0;
            for (int k = m ; k >= 0; k--)
                A[k] = 0;
            for (int i=0 ; i<n; i++)
            {
                if (A[i] != 0)
                    return -1;
            }
    }
    return 1;
}
```

★ **Job Sequencing** :- Given a set of N jobs where each job$_i$ has a deadline and profit associated with it. Each job takes 1 unit of time to complete and only one job can be scheduled at a time. We earn the profit associated with job if and only if the job is completed by its deadline.

Find the number of jobs done and the maximum profit.

Note :- Jobs will be given in the form (Job id, Deadline, Profit) associated with that job.

Example :- Input :- N = 4

Jobs = {(1,4,20), (2,1,10), (3,1,40), (4,1,30)}

output :- 2 60

Explanation :- Job$_1$ and Job$_3$ can be done with maximum profit of 60 (20+40).

⟹

```
int Maximum_Profit (int P[], int D[], int n, int k)
{
    int Profit = 0;
    for (int i = 0; i < n-1; i++)
    {
        for (int j = i; j < n; j++)
        {
            if (P[j] > P[i])
            {
                swap (&P[i], &P[j]);
                swap (&D[j], &D[i]);
            }
        }
    }
    for (int i = 0; i < n; i++)
    {
        static int j = 0;
        if (D[i] > j && j < k)
        {
            Profit = Profit + P[i];
            j++;
        }
    }
    return Profit;
}
```

(25) Suppose there is a circle. There are N petrol pumps on that circle. You will be given two cell of data.

a) The amount of Petrol that every Petrol Pump has.

b) Distance from that Petrol Pump to the next petrol pump.

Find a Starting pt. where the truck can start to get through the Complete circle without exhausting its petrol in between.

Note :- Assume for 1 litre Petrol, the truck can go 1 unit of distance.

Example :- N = 4

Petrol = 4 6 7 4

Distance = 6 5 3 5

Output : 1

Explanation :- There are 4 Petrol pumps with amount of petrol and distance to next Petrol pump value pairs as {4,6}, {6,5}, {7,3}, {4,5}. The first point from where truck can make a circular tour is 2nd Petrol pump. output in this case is 1 (index of 2nd Petrol pump).

⇒ This Question is in a circular fashion, so we know whenever there will be a circular fashion then. then will use % (modular) function.

So we will take two pointers start and end and we will iterate over all the petrol pumps and check if the tour is completed or not and whenever the current petrol < 0, then will remove that petrol-pump and update the start pointer and if current petrol > 0 then will always update end pointer and if we reach the start again, then will return start pointer.

```
int Circular_Tour (int P[], int D[], int n)
{
    int start = 0;
    int end = 1;
    int curr_Petrol = P[start] - D[start];
    while (start != end || curr_Petrol < 0)
    {
        while (start != end && curr_Petrol < 0)
        {
            curr_Petrol = curr_Petrol - (P[start] - D[start]);
            start = (start + 1) % n;
            if (start == 0)
                return -1;
        }
        curr_Petrol = curr_Petrol + (P[end] - D[end]);
        end = (end + 1) % n;
    }
    return start;
}
```

28) **Maximum of All SubArrays of Size k** :- Given an Array arr[] of Size N and an integer k. Find the maximum for each and every contigous subArray of size k.

Example :- Input :- N = 9, k = 3

arr[] = {1, 2, 3, 1, 4, 5, 2, 3, 6}

output :- 3 3 4 5 5 5 6

Explanation :-

1st	Contigous SubArray	= {1 2 3}	max = 3
2nd	"	= {2 3 1}	max = 3
3rd	"	= {3 1 4}	max = 4
4th	"	= {1 4 5}	max = 5
5th	"	= {4 5 2}	max = 5
6th	"	= {5 2 3}	max = 5
7th	"	= {2 3 6}	max = 6

⇒ **Approach 1** :- We will simply do by using two nested loops. outer loop will run from 0 to n-k and inner loop will from (outer index +1 to outer index + k)

```
Void Maximum_of_All_Sub_Arrays (int A[], int n, int k)
{
    for (int i=0 ; i<= n-k ; i++)
    {
        int max = A[i];
        for (int j = i+1 ; j<k+i ; j++)
        {
            if (A[j] > max)
                max = A[j];
        }
        printf ("%d ", max);
    }
}
```

(27) **Count Inversions** :- Given an array of integers. Find the inversion Count in the array.

Inversion Count :- For an array, inversion Count indicates how far (or close) the array is from being sorted. If array is already sorted then the inversion Count is 0. If an array is sorted in the reverse order then inversion Count is the maximum.

Formally, two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j.

Example :- Input : N = 5, arr[] = {2, 4, 1, 3, 5}
Output : 3
Explanation :- The sequence 2, 4, 1, 3, 5 has three inversions (2, 1), (4, 1), (4, 3)

Approach - 1 :- We will use two nested loops to check if a[i] > a[j]

```
int inversion (int A[], int n)
{
    int Count = 0;
    for (int i = 0; i < n-1; i++)
    {
        for (int j = i+1; j < n; j++)
        {
            if (A[i] > A[j])
                Count++;
        }
    }
    return Count;
}
```

Approach-2 :- Using Merge Sort

```
int Inversion_Count(int A[], int l, int h)
{
    int inv = 0;
    int Merge(int A[], int l, int mid, int h)
    if (l < h)
    {
        int mid = floor((l+h)/2);
inv = inv + Inversion_Count(A, l, mid);
inv = inv + Inversion_Count(A, mid+1, h);
    inv = inv + Merge(A, l, mid, h);
    }
    return inv;
}
```

```
int Merge(int A[], int l, int mid, int
{
    int i = l, j = mid+1, k = l;
    int b[100];
    int Count = 0;
    while (i<=mid && j<=h)
    {
        if (A[i] < A[j])
            b[k++] = A[i++];
        else
        {
            b[k++] = A[j++];
            Count = Count + (mid+1-i);
        }
    }
for( ; i<=mid ; i++)
    b[k++] = A[i];
for( ; j<=h ; j++)
    b[k++] = A[j];
for(int p=l ; p<=h ; p++)
    {
        A[p] = b[p];
    }
    return Count;
}
```

28) Next Higher Palindromic Number Using Same Set of digits :-

Given a palindromic no. N in the form of string. The task is to find the smallest palindromic no. greater than N using the same set of digits as in N.

Example :- Input :- N = "35453"

Output :- 53435

Explanation :- Next Higher Palindromic no. is 53435.

```
Void Swap ( char * x, char * y)
{
    char temp = *x;
     *x = *y;
     *y = temp;
}

Void reverse (char num[], int i, int j)
  {
     while (i < j)
     {
        Swap (&num[i], &num[j]);
          i++;
          j--;
     }
  }
```

```
void Palindrome (char num[], int n)
{
    if (n <= 3)
    {
        printf ("Not possible");
        return;
    }
    int mid = n/2 - 1;
    int i, j;
    for (i = mid-1 ; i >= 0 ; i--)
    {
        if (num[i] < num[i+1])
            break;
    }
    if (i<0)
    {
        printf ("Not possible");
        return;
    }
    int smallest = i+1;
    for ( j = i+2; j <= mid ; j++)
    {
        if (num[j] > num[i] && num[smallest] >= num[j])
            smallest = j;
    }
    Swap (&num[i], &num[smallest]);
    Swap (&num[n-1-i], &num[n-smallest-1]);
    Reverse (num, i+1, mid);
    if (n%2 == 0)
        Reverse (num, mid+1, n-i-2);
    else
        Reverse (num, mid+2, n-i-2);
    printf ("Next highest Palindrome number :\n");
    for (int i=0 ; num[i] != '\0' ; i++)
        printf ("%c", num[i]);
}
```

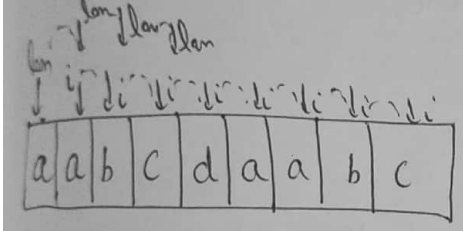
29 **Longest Prefix Suffix** :- Given a string of characters, find the length of the longest proper prefix which is also a proper suffix.

Note :- Prefix and Suffix can be overlapping but they should not be equal to the entire string.

Example :- Input :- S = "abab"

Output :- 2

Explanation :- "ab" is the longest proper prefix and Suffix.




```
int length_of_Pref_Suffix (char str[], int n)
{
  if (n<2)
    return 0;
  int len = 0;
    int i = 1;
  while (i<n)
  {
    if (str[i] == str[len])
      {
        len++;
        i++;
      }
    else
    {
      i = i-len+1;
      len = 0;
    }
  }
  return len;
}
```

30

Smallest Window in a String

Given two String S and P. Find the Smallest Window in the String S Consisting of all the characters (including duplicates) of the String P. Return "-1" in case there is no such window present. In case there are multiple such windows of same length, return the one with the least starting index;

Example :- Input :- S = "timetopractice"
P = "toc".

Output :- toprac

Explanation :- "toprac" is the Smallest Substring in which "toc" can be found.

Approach - 1 :- Brute force Approach

```
void Smallest (char s[], char P[])
{
  int i, j, k;
  char * ans = (char *) malloc (sizeof (char));
  int len = 100;
  for (i=0; s[i] != '\0'; i++)
  {
    for (j=i; s[j] != '\0'; j++)
    {
      int m = 0;
      for (k=i; k<=j; k++)
      {
        ans[m++] = s[k];
      }
      ans[m] = '\0';
      int count = 0;
      int l = 0;
      for (int p = 0; ans[p] != '\0'; p++)
      {
        if (ans[p] == P[l])
        {
          count++;
          l++;
        }
      }
      if (count == strlen (P))
      {
        int length = strlen (ans);
        if (length < len)
        {
          for (int m = 0; ans[m] != '\0'
          {
            printf ("%c", ans[m]);
          }
          len = length;
          printf ("\n");
        }
      }
    }
  }
```

Approach-2 :- Using Hashing

```
void Smallest_window_hash(char s[], char p[])
{
    int l1 = strlen(s);
    int l2 = strlen(p);
    if (l2 > l1)
    {
        printf("Not possible \n");
        return;
    }
    int hash_str[256] = {0};
    int hash_pat[256] = {0};
    for (int i=0; i<l2; i++)
    {
        hash_pat[p[i]]++;
    }
    int start=0, start_index=-1;
    int min_length = _INT_MAX_;
    int count=0;
    for (int j=0; s[j]!='\0'; j++)
    {
        hash_str[s[j]]++;
        if (hash_str[s[j]] == hash_pat[s[j]])
            count++;
        if (count == l2)
        {
            while ((hash_str[s[start]] > hash_pat[s[start]]) ||
                   hash_pat[s[start]]==0)
            {
                if (hash_str[s[start]] > hash_pat[s[start]])
                    hash_str[s[start]]--;
                start++;
            }
            int len_window = j-start+1;
            if (len_window < min_length)
            {
                min_length = len_window;
                start_index = start;
            }
        }
    }
    if (start_index == -1)
    {
        printf("Not possible");
        return;
    }
    while(s[start_index]!='\0' && min_length>0)
    {
        printf("%c", s[start_index]);
        start_index++;
        min_length--;
    }
}
```

31) **Merge Sort for Linked List** :- Given Pointer / Reference to the head of the linked list, the task is to sort the given linked list using merge sort.

Note :- If the length of the linked list is odd, then the extra node should go in the first list while splitting.

Example :- Input :- 5
value [] = { 3, 5, 2, 4, 1 }
Output :- 1 2 3 4 5

Explanation :- After sorting the given linked list, the resultant matrix will be $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$

$\Longrightarrow$ We will follow the approach as we see in arrays, that will always break the list from middle until each & every node will be a list itself.

```
Void find_Middle (struct Node ** first, struct Node ** a,
                                        struct Node **b)
{
    struct Node * slow = first;
    struct Node * fast = first -> next;
    while (fast)
    {
        fast = fast -> next;
        if (fast)
        {
            slow = slow -> next;
            fast = fast -> next;
        }
    }
    *a = first;
    *b = first -> next;
    slow -> next = NULL;
}
```

```
struct Node * merge (struct Node *a, struct Node *b)
{
    struct Node * answer = NULL;
    if (a == NULL)
        return b;
    if (b == NULL)
        return a;
    if (a -> data <= b -> data)
    {
        answer = a;
        answer -> next = merge (a -> next, b);
    }
    else
    {
        answer = b;
        answer -> next = merge (a, b -> next);
    }
    return answer;
}

void merge_sort (struct Node **first)
{
    struct Node *a, *b;
    struct Node * temp = *first;
    if ((temp == NULL) || (temp -> next == NULL))
        return;
    find_middle (temp, &a, &b);
    merge_sort (&a);
    merge_sort (&b);
    *first = merge (a, b);
}
```

32 **Remove k Digits** :- Given a non-integer S represented as a string, remove k digits from the number so that the new number is the smallest possible

Note :- The given num does not contain any leading zero.

Example :- Input :- S = "149811", k = 3

Output :- 111

Explanation :- Remove the three digits 4, 9 and 8 to form the new number 111 which is smallest.

```
void Remove_kdigits (char str[], int n, int k)
{
  char *ans = (char *) malloc ((n-k) * sizeof
                               (char));

  for (int i=0; str[i] != '\0'; i++)
  {
    while (top && k>0 && (str[i]-'0') <
                         (topchar() - '0'))
    {
        POP();
        k--;
    }
    if (top || str[i] != '0')
        push (str[i]);
  }
  while (top && k--)
    POP();

  if (top == NULL)
    Return;

  for (int i=0; i< n-k; i++)
    {
      ans[i] = '0';
    }
  while (top)
  {
    ans[P-1] = top -> data;
      POP();
      n--;
      P--;
  }
  for (int i=n; ans[i] != '\0'; i++)
    {
      if (ans[i] == '0')
          Continue;
      printf ("%c", ans[i]);
    }
}
```

(33) **Boundary Traversal of Binary Tree** :- Given a Binary Tree, find its boundary traversal. The Traversal should be in the following order:-

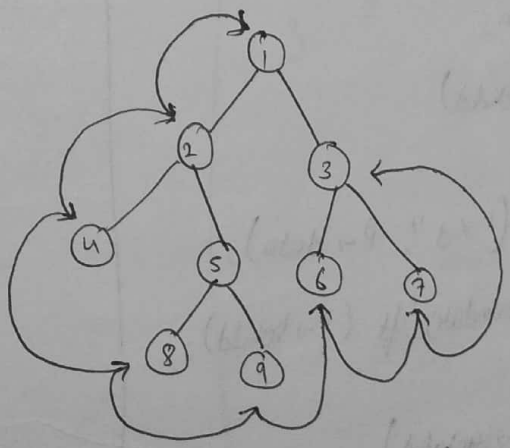
1) **Left Boundary Nodes** :- Defined as the path from the root to the left-most node ie. the leaf node you could reach when you always travel preffering the left subtree over the right subtree.

2) **Leaf Nodes** :- All the leaf nodes expects for the ones that are part of left or right boundary.

3) **Reverse Right Boundary Nodes** :- Defined as the path from right-most-node to the root. The right-most node is the leaf node you could reach when you always travel preffering the right subtree over the left subtree. Exclude the root from this as it was already included in the traversal of left boundary nodes.

Note :- if the root doesn't have a left subtree or right subtree, then the root itself is the left or right boundary.

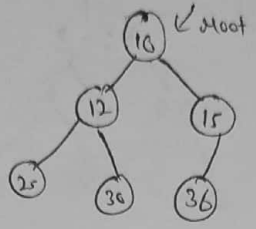
Example :- **Input** :-

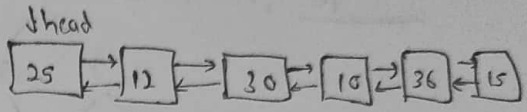


Output :- 1 2 4 8 9 6 7 3

```
Void Boundary_Trav(struct Node *P)
{
    if (P == NULL)
        return;
    printf("%d", P->data);
    Print Boundary left (P->lchild);
    Print Leaf (P->lchild);
    Print Leaf (P->rchild);
    Print Boundary Right (P->rchild);
}

Void Print Boundary left (struct Node *P)
{
    if (P == NULL)
        return;
    if (P->lchild)
    {
        printf("%d", P->data);
        Print Boundary left (P->lchild);
    }
    else if (P->rchild)
    {
        printf("%d", P->data);
        Printf Boundary left (P->rchild);
    }
}

Void Print Leaf (struct Node *P)
{
    if (P == NULL)
        return;
    Print Leaf (P->lchild);
    if ((!P->lchild) && (!P->rchild))
        printf("%d", P->data);
    Print Leaf (P->rchild);
}

Void Print Boundary Right (Struct Node *P)
{
    if (P == NULL)
        return;
    if (P->rchild)
    {
        Print Boundary Right (P->rchild);
        printf("%d", P->data);
    }
    else if (P->lchild)
    {
        Print Boundary Right (P->lchild);
        printf("%d", P->data);
    }
}
```

34 **Binary Tree to Doubly Linked List** :- Given a Binary Tree (BT), Convert it to a Doubly Linked List (DDL). The left and right pointers in nodes are to be used as previous and next pointers respectively in Converted DDL. The order of Nodes in DDL must be same as inorder of given Binary Tree. The first node of Inorder traversal (leftmost node in BT) must be the head of the DDL.





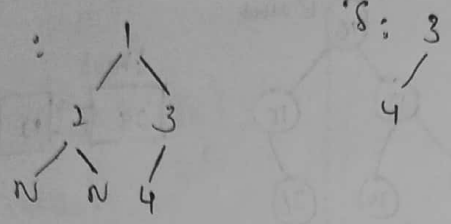

```
Struct Node * Convert (Struct Node *P)
{
    if (P == NULL)
        return NULL;
    Convert (P→lchild);
    if (Prev → Global == NULL)
        head = P;
    else
    {
        P→lchild = Prev;
        Prev→rchild = P;
    }
    Prev = P;
    Convert (P→rchild);
}
```

```
Void DDL Traverse (Struct Node *P)
{
    while (P != NULL)
    {
        printf ("%d ", P→data);
        P = P→rchild;
    }
}
```

35 **Check If Sub Tree** :- Given two Binary Trees with head reference of T and S having atmost N nodes. The task is to check if S is present as subtree in T.

A Subtree of a tree T_1 is a tree T_2 consisting of node in T_1 and all of its descendants in T_1.

Example :-

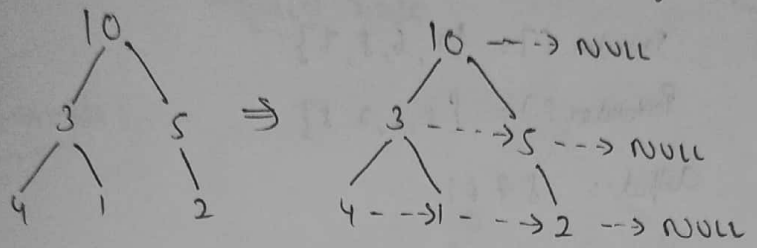


S is present in T

```
bool Sub_Tree (struct Node *t, struct Node *s)
{
    if (t == NULL)
        return false;
    if (s == NULL)
        return true;
    if (isIdentical(t, s))
        return true;
    if (Sub_Tree (t->lchild, s))
        return true;
    if ((Sub_Tree (t->rchild, s))
        return true;
}
```

```
bool isIdentical (struct Node *t, struct Node
{
    if (s == NULL && t == NULL)
        return true;
    if (t == NULL || s == NULL)
        return false;
    if ((t->data == s->data) && isIdentical
        (t->lchild, s->lchild) && isIdentical
        (t->rchild, s->rchild))
        return true;
    else
        return false;
}
```

Connect Nodes at Same Level :- Given a Binary tree, Connect the nodes that are at same level. You will be given an additional nextRight pointer for the same. Initially all the nextRight Pointers points to garbage values. your function should set these pointers to point next Right for each node.




```
void ConnectNodes (struct Node *p)
{
    struct Queue q;
    CreateQueue (&q, 100);
    struct Node *t;
    enQueue (&q, root);
    while (isEmpty (q))
    {
        t = deQueue (&q);
        if (t->lchild && t->rchild)
        {
            t->lchild->nextRight = t->rchild;
            enQueue (&q, t->lchild);
        }
        enQueue (&q, t->rchild);
        if (t->lchild)
            enQueue (&q, t->lchild);
        if (t->rchild)
            enQueue (&q, t->rchild);
    }
}
```

(37) Generate Tree from Preorder and Inorder Traversal :-

Given 2 Arrays of Inorder & Preorder traversal. Construct a tree and print the postorder traversal.

Example :- Input :- N = 4

Inorder [] = {1, 6, 8, 7}

Preorder [] = {1, 6, 7, 8}

Output :- 8 7 6 1

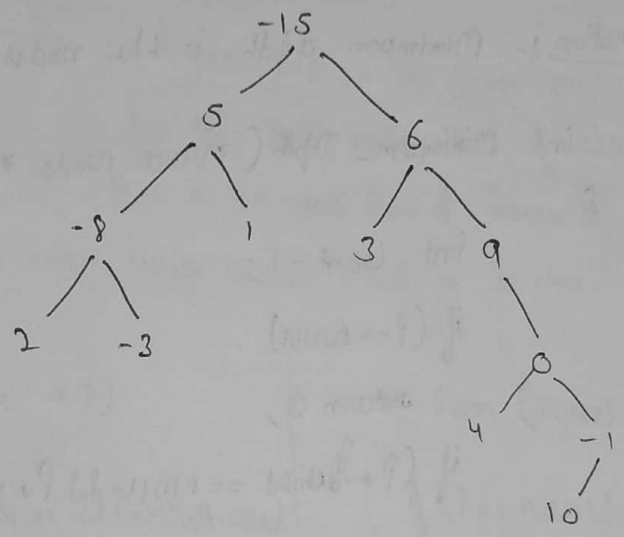
```
Struct Node * Build (int A[], int B[], int Start, int end)
{
    Static int PreIndex = 0;
    if (Start > end)
        return NULL;
    Struct Node *P = (Struct Node *) malloc (sizeof (Struct Node));
    P->data = B[PreIndex++];
    P->lchild = P->rchild = NULL;
    if (Start == end)
        return P;
    int inIndex = Search (A, Start, end, P->data);
    P->lchild = Build (A, B, Start, inIndex - 1);
    P->rchild = Build (A, B, inIndex + 1, end);
    return P;
}
```

```
int Search (int B[], int Start, int end,
            int key)
{
    for (int i = Start; i <= end; i++)
    {
        if (key == B[i])
            return i;
    }
}
```

Maximum Path Sum between 2 Leaf Nodes :- Given a Binary Tree in which each node element contains a number. Find the maximum possible path sum from one leaf node to another leaf node.

Note :- Here Leaf node is a node which is directly connected to one different node.

Example :-



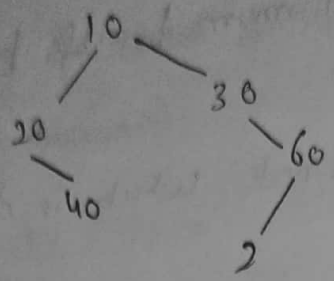
output :- 27

Explanation :- The maximum possible sum from one leaf node to another is (3 + 6 + 9 + 0 + (-1) + 10) = 27

```
int Maximum_Sum_Leaf (struct Node *P, int *res)
{
    if (P == NULL)
        return 0;
    if (P->lchild == NULL && P->rchild == NULL)
        return P->data;
    int * temp = res;
    int ls = Maximum_Sum_leaf (P->lchild, temp);
    int rs = Maximum_Sum_leaf (P->rchild, temp);
    if (P->lchild && P->rchild)
    {
        *res = max (*res, ls + rs + P->data);
        return (max(ls, rs) + P->data);
    }
}
```

39 **Minimum Depth** :- Given a Binary Tree, find its minimum depth.

Example :-



output :- 3

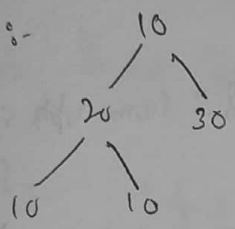
Explanation :- Minimum depth is b/w nodes 10, 20 and 40.

```
int Minimum_Depth ( struct Node *P)
{
    int Count = 1;
    if (P == NULL)
        Return 0;
    if (P→ lchild == NULL && P→ rchild == NULL)
        Return 1;
    int x = Minimum_Depth (P→ lchild);
    int y = Minimum_Depth (P→ rchild);
    if (x > y)
        Return y+1;
    else
        Return x+1;
}
```

40 **Sum Tree** :- Given a Binary tree. Return true, if for every node X in the tree other the leaves, its value is equal to sum of its left subtree's value and its right subtree's value. Else return false.

An Empty tree is also a sum tree as the sum of an empty tree can be considered to be 0. A leaf node is also considered a sum tree.

Example :-



Output :- 0

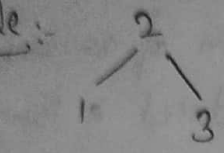
Explanation :- The given tree is not a sum tree, for the root node, sum of elements in left sub tree is 40 and sum of elements in right sub tree is 30, Root element = 10 which is not equal to 30+40.

```
int check_Sum (struct Node * P)
{
  if (P==NULL || P→lchild == NULL && P→rchild==NULL)
     return true ;
  int x= Sum (P→lchild);
  int y = Sum (P→rchild);
  if((x+y== P→data) && (check_Sum (P→lchild) &&
     (check_Sum (P→rchild)))
     return true;
  return false;
}
```

```
int Sum (struct Node *P)
{
   if (P == NULL)
      return 0;
   else
   return P→data + Sum (P→lchild) +
          Sum (P→rchild);
}
```

(41) **Find a Pair with given target in BST** :- Given a Binary tree and a target sum. check whether there's pair of Nodes in the BST with value summing up to the target sum.

Example :-



Sum = 5

Output :- 1 (Node with value 2 & 3 sum upto 5)

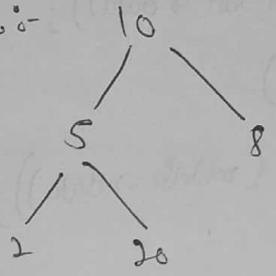
```
bool check_sum (struct Node *P, int sum, int hash[])
{
    if (P == NULL)
        return false;
    if (check_sum (P→lchild, sum, hash))
        return true;
    hash [P→data] = 1;
    for (int i=0; i< 200; i++)
    {
        if (hash [sum - P→data] == 1)
        {   return true;
        }
    return check_sum (P→rchild, sum, hash);
    return
        false;
}
```

Fixing two Nodes of a BST :- You are given the root of a Binary Search tree (BST), where exactly two nodes were swapped by mistake. Fix (or Correct) the BST by swapping them back. Do not change the structure of the tree.

Note :- it is guaranteed that the given input will form BST, except for 2 Nodes that will be wrong. All changes must be reflected in the original linked list.

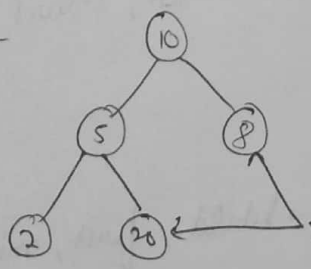
Example1 :-

Input :-

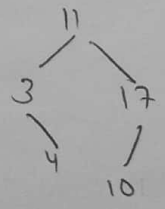


Output :- 1

Explanation :-



Example 2 :- **Input** :-



Output :- 1

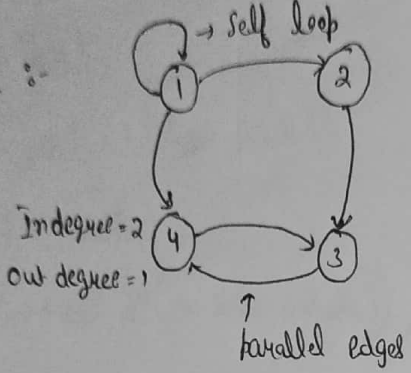
Explanation :- By swapping node 11 and 10, the BST can be fixed.

```
Void Swap (struct Node *t)
{
    struct Node * Prev = NULL, *first = NULL, * last = NULL, * middle = NULL;
Void Swap_Numbers (struct Node *t, struct Node **first, struct Node **middle,
                   struct Node ** last, struct Node ** Prev);
    Swap_Numbers (root, &first, &middle, &last, &Prev);
    if (first && last)
    Swap (&(first -> data), &(last -> data));
    else if (first && middle)
    Swap (&(first -> data), &(middle -> data));
}
Void Swap_Numbers (struct Node *t, struct Node **first, struct Node **middle,
                   struct Node ** last, struct Node ** Prev)
{
    if (t)
    {
    Swap_Numbers (t -> lchild, first, middle, last, Prev);
    if (*Prev && (*Prev) -> data > t -> data)
      {
          if (!*first)
          {
            *first = *Prev;
          *middle = t;
          }
          else
            *last = t;
      }
      *Prev = t;
      Swap_Numbers (t -> rchild, first, middle, last, Prev);
} }
```

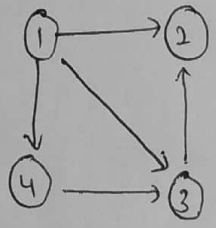
GRAPHS

It is a collection of vertices and edges, $G = (V, E)$, where V = Set of vertices & E = set of edges

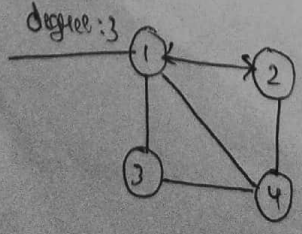
(1) **Directed Graph** :-



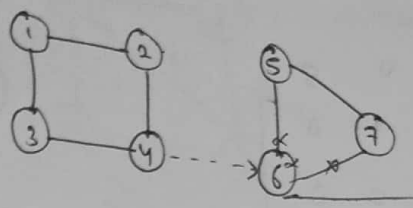
(2) **Simple Digraph** :- A Graph without self loop and parallel edges is known as Simple Digraph.



(3) **Non-Directed Graph** :- A Graph where edges are not directed, they can be or may be directed in both directions as there are not any parallel edges, not any indegree / outdegree, there could be just a degree and these are also known as Graphs and the Graph which have Directed edges are known as Di Graph.



④ Non-Connected Graphs :- Where the Graphs are not Connected as we can splitted into different Components are known as Non-Connected Graphs

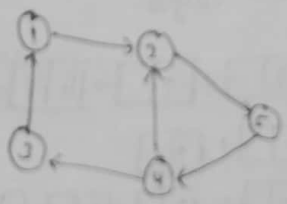


Articulation point

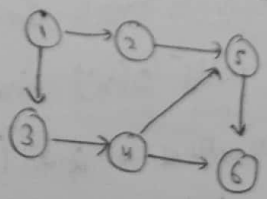
2 Components

if we Connect (4) with (6), than it will become Connected Graph but further if i Remove (6) along with its edges than it will again become non-Connected Graph, Hence, the point by which a graph can become non-Connected by Removal of that point, we can say that pt is Articulation pt.

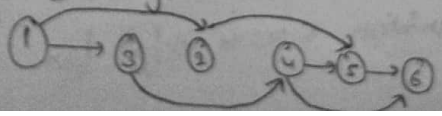
⑤ Strongly Connected :- If a Directed Graph, Satisfies the Condition that from any Vertex, We can Reach any Vertex than it will be a Strongly Connected Graph



⑥ Directed Acyclic Graph (DAG) :- In a Directed Graph, if you Cant Reach the Starting pt. again, it means Cycles are not formed for any Vertex or every Vertex, such a Graph are known as DAG.

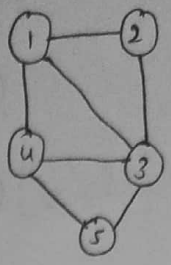


If we can arrange the edges in forward direction only of DAG, than it will be known as Topological ordering.



★ Representation of Undirected Graph :-

(1) Adjacency Matrix :-



$G = (V, E)$

$|V| = n = 5$

$|E| = e = 7$

if $(i, j) \rightarrow A[i, j] = 1$

else $A[i, j] = 0$

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \\ 5 \times 5 \end{array}$$

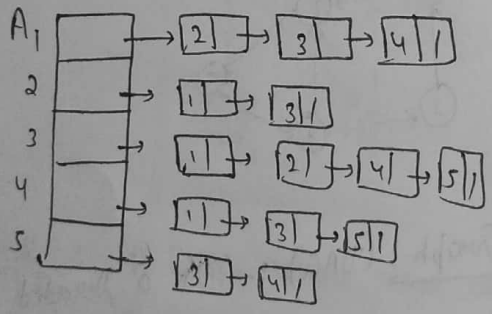
$n \times n = n^2$

$O(n^2)$

if there is an edge b/w i & j, marked / fill it with 1 else 0

(2) Adjacency List :-

In this Array of pointers (→ List) we take, in which array represents Vertices and the remaining List will represent edges.



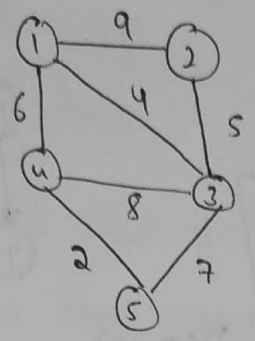
Note :- We can see that, in Matrix representation, if we want to access any vertex, then we have to visit every vertex to check if there is a path or not, which will $O(n^2)$ time but in List representation it depends upon no. of vertices + 2 * (no. of edges)

$|V| + 2|E|$

$V + e \Rightarrow O(V+e)$ Space & time

"That's why List representation is more useful for analysation" Consumption

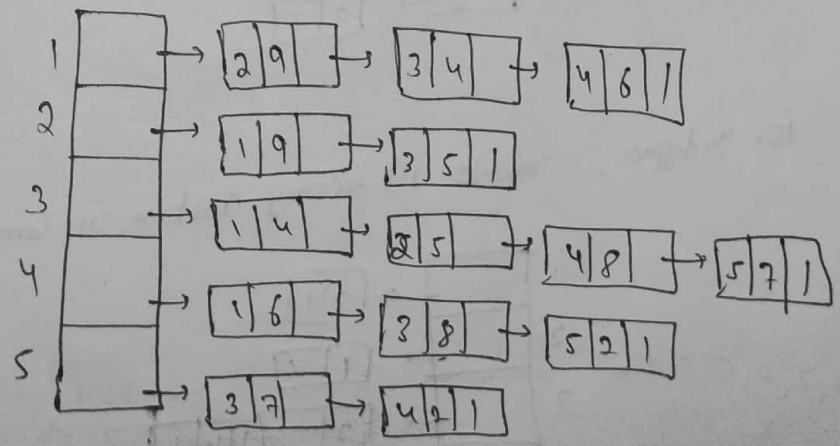
In Case of Weighted Graph, it means if the weights or Cost of the edge is given, then



In form of Adjacency Matrix, and Now this matrix is known as Cost Adjacency Matrix.

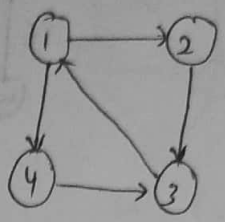
	1	2	3	4	5
1	0	9	4	6	0
2	9	0	5	0	0
3	4	5	0	8	7
4	6	0	8	0	2
5	0	0	7	2	0

In form of Adjacency List, and Now this List is known as Cost Adjacency List.



★ Representation of Directed Graph :- Analysation is same as in Undirected Graph.

(1) Adjacency Matrix :-



$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

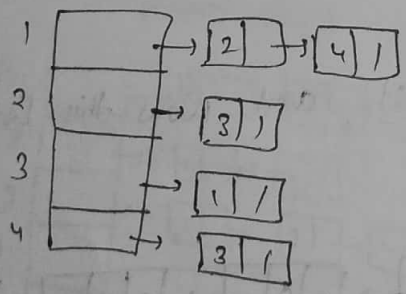
$n \times n = n^2$

$O(n^2)$

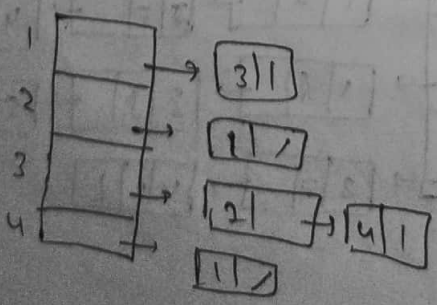
Now, if we want see Outdegree we can just simply watch a row if it is 1, than there is a edge out but if we want to see Indegree, than visit Column for that vertex if it is 1 than there will be edge in.

(2) Adjacency matrix | Inverse Adjacency matrix :-

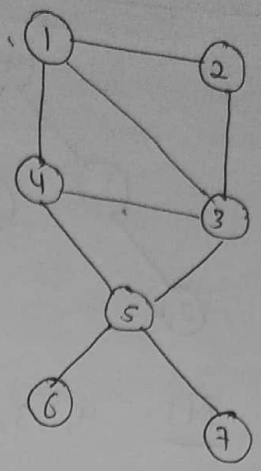
" for Outdegree, Adjacency matrix is come into play "



" for Indegree, Inverse Adjacency matrix is come into play "

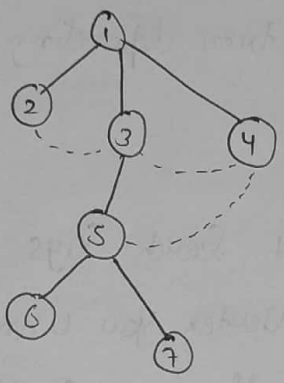


* Breadth First Search :-



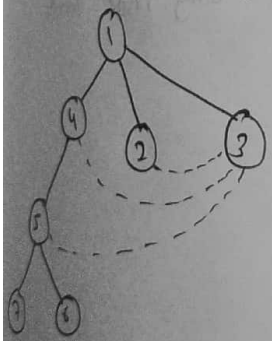
It is similar to Level order traversal in a Binary tree.

Now, lets try to make a tree of that and will use dotted lines because otherwise it will be a graph

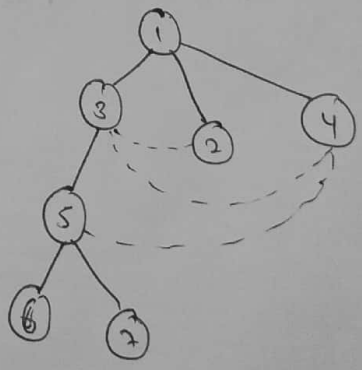


BFS :- 1, 2, 3, 4, 5, 6, 7 (level order)

Now, this is not the only BFS, We can take any BFS as there can be multiple trees formation depending upon the order of adjacent vertices



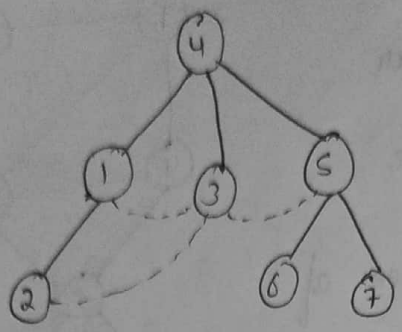
BFS :- 1, 4, 2, 3, 5, 7, 6 (level order)



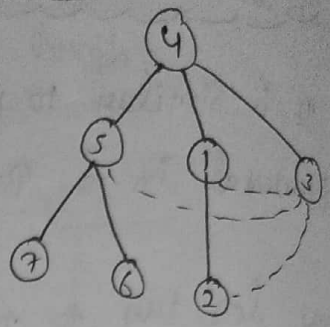
BFS :- 1, 3, 2, 4, 5, 6, 7

So, this is how we can form multiple Trees and having multiple BFS.

Now, We can also start from other than 1



BFS :- 4, 1, 3, 5, 2, 6, 7

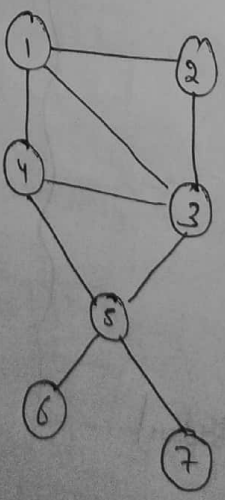


BFS :- 5, 4, 1, 3, 7, 6, 2

and can form many more trees depending upon adjacent vertices we are taking.

⇒ Summarize :- Breadth First Search says that you can start from any starting vertex you want and when you visit any vertex, you have explore it completely, explore means you have visit to all its adjacent vertices and exploration can be done in any order.

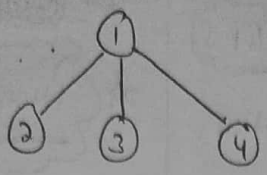
Now, To Do BFS, We need Queue, So, first visit any node and then complete its exploration



Now, Let us start from 1, drop it in the Queue & explore it

BFS : 1, 2, 3, 4

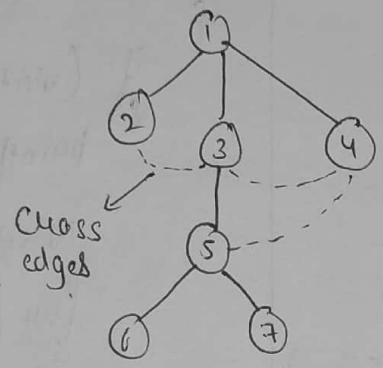
Queue : ~~1~~, 2, 3, 4



Now, take out from the Queue & explore and continue till Queue will be empty

BFS : 1, 2, 3, 4, 5, 6, 7

Queue : ~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~5~~, ~~6~~, ~~7~~



$\Longrightarrow$ BFS Spanning Tree

In this way, we can form multiple BFS but the main thing is we have to visit and explore, each & every vertex.

Note :- Cross edges are connected to the adjacent level of vertices.

Time Complexity :- $O(n)$ bcz Analytically we visited all the nodes

Program :-

q | 1 | 2 | 3 | 4 | | | |

visited | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
0 1 2 3 4 5 6 7

① Using Adjacency matrix :- BFS

n = no. of vertices

```
Void BFS (int i)
{
  printf ("%d", i);
  visited[i] = 1;
  enqueue (q, i);
  while (! isEmpty ())
  {
    int u = dequeue (q);
    for (v = 1; v <= n; v++)
    {
      if (A[u][v] == 1 && visited[v] == 0)
      {
        printf ("%d", v);
        visited[v] = 1;
        enqueue (q, v);
      }
    }
  }
}
```

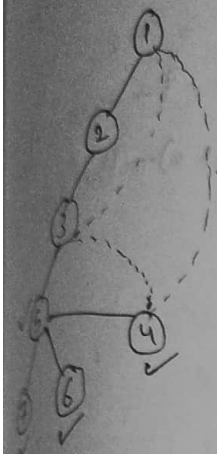
★ Program for DFS :-

visited | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
0 1 2 3 4 5 6 7

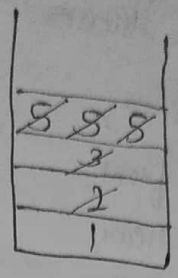
```
Void DFS (int u)
{
  if (visited[u] == 0)
    printf ("%d", u);
  visited[u] = 1;
  for (v = 1; v <= n; v++)
  {
    if (A[u][v] == 1 && visited[v] == 0)
      DFS (v);
  }
}
```

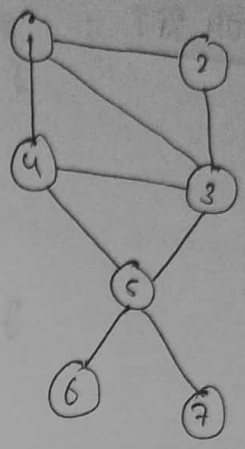
Depth First Search :- It is similar to Preorder traversal in Binary trees.

DFS : 1, 2, 3, 5, 7, 6, 4



→ back edges





⇒ DFS spanning Tree

- Now, start visiting from any vertex (let us say 1)
- Now, start exploring By taking any adjacent vertex (Let us say 2)

Now, we don't have to further explore (1), stop and suspended (1) and start exploring (2) and push (1) into the stack so that in future we can explore it.

Now, go on (3), suspend (2) and start exploring (3) and push (2) in the stack.

So the approach is when we get the new vertex, we will suspend the old vertex and start exploring new vertex.

and when the exploration of a node is done, then go back to previous node by popping it out from stack.

Time Complexity :- It depends upon no. of vertices bcz we are visiting once thats why it is $O(n)$.

but its Analytically, while code, it may take more time.

☆ Code for BFS and DFS :-

① For BFS :-

```
struct Queue
{
    int size;
    int front;
    int rear;
    int *Q;
};
```

```
void createQueue (struct Queue *q, int size)
{
    q->size = size;
    q->front = q->rear = -1;
    q->Q = (int *) malloc (size * sizeof(int));
}
```

```
void enQueue (struct Queue *q, int x)
{
    if (q->rear == q->size - 1)
        printf ("Queue is full");
    else
    {
        q->rear++;
        q->Q[q->rear] = x;
    }
}
```

```
int deQueue (struct Queue *q)
{
    int x = -1;
    if (q->front == q->rear)
        printf ("Queue is Empty");
    else
    {
        q->front++;
        x = q->Q[q->front];
    }
    return x;
}
```

```
int isEmpty (struct Queue *q)
{
    if (q->front == q->rear)
        return 1;
    return 0;
}
```

```
int main ()
{   int A[7][7] = {
                  {0,0,0,0,0,0,0},
                  {0,0,1,1,0,0,0},
                  {0,1,0,0,1,0,0},
                  {0,1,0,0,1,0,0},
                  {0,0,1,1,0,1,1},
                  {0,0,0,0,1,0,0},
                  {0,0,0,0,1,0,0}
                  };
    Void BFS (int A[][7], int i);
    BFS (A,4);
    Return 0;
}
Void BFS ( int A[][7], int i)
{
  struct queue q;
  Create Queue ( &q,7);
  int visited [7] = {0};
  int u;
  printf ("%d", i);
  visited [i] = 1;
  enqueue ( &q, i);
  While ( ! is Empty (&q))
    {
      int u = dequeue (&q);
      for (int v = 1 ; v <=7 ; v++)
        {
          printf ("%d", v);
          visited [v] = 1;
          enqueue (&q, v);
        }
    }
}
```

if (A[u][v] == 1 && visited[v] == 0)

② For DFS :-

```
int main()
{
    int A[7][7] = {
                    {0,0,0,0,0,0,0},
                    {0,0,1,1,0,0,0},
                    {0,1,0,0,1,0,0},
                    {0,1,0,0,1,0,0},
                    {0,0,1,1,0,1,1},
                    {0,0,0,0,1,0,0},
                    {0,0,0,0,1,0,0}
                  };
    Void DFS (int A[][7], int i, int m[7]);
    int visited[7] = {0};
    DFS (A, 1, visited);
    Return 0;
}

Void DFS (int A[][7], int i, int m[7])
{
    if (m[i] == 0)
    {
        printf ("%d", i);
        m[i] = 1;
        for (int v = 0; v <= 7; v++)
        {
            if (A[i][v] = 1 && m[v] == 0)
                DFS (A, v, m);
        }
    }
}
```

43 **Length of the Longest Substring** :- Given a string S, find the length of the longest substring without repeating characters.

Example :- Input :- S = "geeksforgeeks"

Output :- 7

Explanation :- Longest substring is "eksforg"

Approach-2 :- **Brute-force approach**

```
int Longest (char str[])
{
   int len = strlen (str);
  int hash_visited [256] = {0};
  int max = 0;
   int j;
for(int i=0 ; i<len ; i++)
{
   for (int j=i ; j<len ; j++)
   {
      for(int k= i; k<=j ; k++)
       {
         hash_visited [str[k]] ++;
       }
      if (hash_visited [str[j]] >1)
         break;
      for (int P=i ; P<=j ; P++)
       {
         hash_visited [str[P]] --;
       }
      if ((j-i) > max)
         max = j-i+1;
   }
  for (int m=i ; m<=j ; m++)
    {
      hash_visited [str[m]] --;
    }
  if ((j-i) > max)
      max = j-i;
}
  return max;
}
```

Approach-2

: Acquire and Release method

```
int Longest (char str[])
{
    char hash [256] = {0};
    int max = 0;
    int j = -1;
    for (int i=0 ; str[i] != '\0' ; i++)
    {
        // ---- Acquire -----
        hash [str[i]] ++;
        // --- Release ---
        if (hash [str[i]] > 1)
        {
            while (j < i)
            {
                j++;
                hash [str[j]] --;
                if (hash [str[j]] == 1)
                    break;
            }
            if ((i - j) > max)
                max = i - j;
        }
    }
    return max;
}
```

Longest k unique characters substring :- Given a string need to print the s
of the longest possible substring that has exactly k unique cha
if there is no possible substring then print -1.

Example :- S = "aabacbebebe" , k = 3

output :- 7

Explanation :- "cbebebe" is the longest substring with k dis
characters

Approach - 1 : Brute force approach

Approach-2 :- Using Acquire & Release method

```
int longest_k_unique (char str[], int k)
{
    char hash[256] = {0};
    int max = -1;
    int j = -1;
    int count = 0;
    for (int i=0; str[i] != '\0'; i++)
    {
        if (hash[str[i]] == 0)
            count++;
        has[str[i]]++;
        if (count > k)
        {
            while (j < i)
            {
                j++;
                hash[str[j]]--;
                if (hash[str[j]] == 0)
                    break;
            }
            count--;
        }
        if (count == k)
        {
            if ((i-j) > max)
                max = i-j;
        }
    }
    return max;
}
```