ARRAYS

- **Stock Buy Sell to Maximize Profit :-**
  The cost of a stock on each day is given in an array, find the max profit that you can make by buying and selling in those days. For example, if the given array is {100, 180, 260, 310, 40, 535, 695}, the maximum profit can earned by buying on day 0, selling on day 3. Again buy on day 4 and sell on day 6. If the given array of prices is sorted in decreasing order, then profit cannot be earned at all.

  If we are allowed to buy and sell only once, then we can use following algorithm. Maximum difference between two elements. Here we are allowed to buy and sell multiple times.
  Following is algorithm for this problem.

  1) Find the local minima and store it as starting index. If not exists, return.
  2) Find the local maxima. and store it as ending index. If we reach the end, set the end as ending index.
  3) Update the solution (Increment count of buy sell pairs)
  4) Repeat the above steps if end is not reached.

```
void stockBuySell(int price[], int n)
{
    if (n == 1)
        return;

    int count = 0;
    Interval sol[n / 2 + 1];
    int i = 0;
    while (i < n - 1) {
        while ((i < n - 1) && (price[i + 1] <= price[i]))
            i++;
          if (i == n - 1)
            break;

        sol[count].buy = i++;

        while ((i < n) && (price[i] >= price[i - 1]))
            i++;
        sol[count].sell = i - 1;
        count++;
    }
    if (count == 0)
        cout << "There is no day when buying"
             << " the stock will make profitn";
    else {
        for (int i = 0; i < count; i++)
            cout << "Buy on day: " << sol[i].buy
                 << "\t Sell on day: " << sol[i].sell << endl;
    }

    return;
}
```

IB solution :-

```
class Solution {
    public:
        int maxProfit(vector<int> &prices) {
            int total = 0, sz = prices.size();
            for (int i = 0; i < sz - 1; i++) {
                if (prices[i+1] > prices[i]) total += prices[i+1] - prices[i];
            }
            return total;
        }
};
```

- **Maximum profit by buying and selling a share at most twice** :- In a daily share trading, a buyer buys shares in the morning and sells it on same day. If the trader is allowed to make at most 2 transactions in a day, where as second transaction can only start after first one is complete (Sell->buy->sell->buy). Given stock prices throughout day, find out maximum profit that a share trader could have made.

  ```
  Max profit with at most two transactions =
         MAX {max profit with one transaction and subarray price[0..i] +
              max profit with one transaction and aubarray price[i+1..n-1]  }
  i varies from 0 to n-1.
  ```

  We can do this O(n) using following Efficient Solution. The idea is to store maximum possible profit of every subarray and solve the problem in following two phases.

  1) Create a table profit[0..n-1] and initialize all values in it 0.

  2) Traverse price[] from right to left and update profit[i] such that profit[i] stores maximum profit achievable from one transaction in subarray price[i..n-1]

  3) Traverse price[] from left to right and update profit[i] such that profit[i] stores maximum profit such that profit[i] contains maximum achievable profit from two transactions in subarray price[0..i].

  4) Return profit[n-1]

  To do step 1, we need to keep track of maximum price from right to left side and to do step 2, we need to keep track of minimum price from left to right. Why we traverse in reverse directions? The idea is to save space, in second step, we use same array for both purposes, maximum with 1 transaction and maximum with 2 transactions. After an iteration i, the array profit[0..i] contains maximum profit with 2 transactions and profit[i+1..n-1] contains profit with two transactions.

- **Stock transactions for at most k times (DP solution)**
  Let profit[t][i] represent maximum profit using at most t transactions up to day i (including day i). Then the relation is:

  ```
  profit[t][i] = max(profit[t][i-1], max(price[i] – price[j] + profit[t-1][j]))
            for all j in range [0, i-1]
  ```

profit[t][i] will be maximum of –

1) profit[t][i-1] which represents not doing any transaction on the ith day.
2) Maximum profit gained by selling on ith day. In order to sell shares on ith day, we need to purchase it on any one of [0, i – 1] days. If we buy shares on jth day and sell it on ith day, max profit will be price[i] – price[j] + profit[t-1][j] where j varies from 0 to i-1. Here profit[t-1][j] is best we could have done with one less transaction till jth day.

```
int maxProfit(int price[], int n, int k)
{
    int profit[k + 1][n + 1];
    for (int i = 0; i <= k; i++)
        profit[i][0] = 0;
    for (int j = 0; j <= n; j++)
        profit[0][j] = 0;

    for (int i = 1; i <= k; i++) {
        for (int j = 1; j < n; j++) {
            int max_so_far = INT_MIN;

            for (int m = 0; m < j; m++)
                max_so_far = max(max_so_far,
                                 price[j] - price[m] + profit[i - 1][m]);

            profit[i][j] = max(profit[i][j - 1], max_so_far);
        }
    }
    return profit[k][n - 1];
}
```

Optimized Solution:
The above solution has time complexity of O(k.n2). It can be reduced if we are able to calculate maximum profit gained by selling shares on ith day in constant time.

```
profit[t][i] = max(profit [t][i-1], max(price[i] – price[j] + profit[t-1][j]))
                            for all j in range [0, i-1]
```

If we carefully notice,

```
max(price[i] – price[j] + profit[t-1][j])
for all j in range [0, i-1]

can be rewritten as,
= price[i] + max(profit[t-1][j] – price[j])
for all j in range [0, i-1]
= price[i] + max(prevDiff, profit[t-1][i-1] – price[i-1])
where prevDiff is max(profit[t-1][j] – price[j])
for all j in range [0, i-2]
```

So, if we have already calculated max(profit[t-1][j] – price[j]) for all j in range [0, i-2], we can calculate it for j = i – 1 in constant time. In other words, we don’t have to look back in range [0, i-1] anymore to find out best day to buy. We can determine that in constant time using below revised relation.

```
profit[t][i] = max(profit[t][i-1], price[i] + max(prevDiff, profit [t-1][i-1]
- price[i-1])
where prevDiff is max(profit[t-1][j] - price[j]) for all j in range [0, i-2]
int maxProfit(int price[], int n, int k)
{
    int profit[k + 1][n + 1];

    for (int i = 0; i <= k; i++)
        profit[i][0] = 0;
    for (int j = 0; j <= n; j++)
        profit[0][j] = 0;

    for (int i = 1; i <= k; i++) {
        int prevDiff = INT_MIN;
        for (int j = 1; j < n; j++) {
            prevDiff = max(prevDiff,
                           profit[i - 1][j - 1] - price[j - 1]);
            profit[i][j] = max(profit[i][j - 1],
                               price[j] + prevDiff);
        }
    }
    return profit[k][n - 1];
}
```

Time complexity of above solution is O(kn) and space complexity is O(nk). Space complexity can further be reduced to O(n) as we uses the result from last transaction.

- **Merge K sorted Arrays :-**
  - Use Min Heap https://www.geeksforgeeks.org/merge-k-sorted-arrays/ (good for different sized arrays)
  - Divide and Conquer Approach (good for similar sized arrays)

- **Merge two sorted arrays with O(1) extra space :-**
  This task is simple and O(m+n) if we are allowed to use extra space. But it becomes really complicated when extra space is not allowed and doesn't look possible in less than **O(m*n)** worst case time.

```
void merge(int ar1[], int ar2[], int m, int n)
{
    for (int i=n-1; i>=0; i--)
    {
        int j, last = ar1[m-1];
        for (j=m-2; j >= 0 && ar1[j] > ar2[i]; j--)
            ar1[j+1] = ar1[j];
        if (j != m-2 || last > ar2[i])
        {
            ar1[j+1] = ar2[i];
            ar2[i] = last;
        }
    }
}
```

- **Kth largest element in an array**:-
    - Using Max Heap O(n + klogn)
      Insert all n elements in heap.
      Extract Maximum k times.
    - Using Min Heap (k + (n-k)Logk + kLogk)
      Insert first k elements
      Then iterate over the rest. If arr[i] > min , remove root and insert arr[i].
      The root will be the kth largest element.
    - QuickSelect Method O(n) Average , O(n*n) worst

```
int kthSmallest(int arr[], int l, int r, int k)
{
    if (k > 0 && k <= r - l + 1)
    {
     int pos = partition(arr, l, r); // same as quicksort, randomise to
     get O(n) in average

        if (pos-l == k-1)
            return arr[pos];
        if (pos-l > k-1)
            return kthSmallest(arr, l, pos-1, k);
        return kthSmallest(arr, pos+1, r, k-pos+l-1);
    }
    return INT_MAX;
}
```

      https://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-2-expected-linear-time/

- **K maximum sum combinations from two arrays** :-
  Given two equally sized arrays (A, B) and N (size of both arrays).
  A sum combination is made by adding one element from array A and another element of array B. Display the maximum K valid sum combinations from all the possible sum combinations.

  Instead of brute forcing through all the possible sum combinations we should find a way to limit our search space to possible candidate sum combinations.
  1. Sort both arrays array A and array B.
  2. Create a max heap i.e priority_queue in C++ to store the sum combinations along with the indices of elements from both arrays A and B which make up the sum. Heap is ordered by the sum.
  3. Initialize the heap with the maximum possible sum combination i.e (A[N – 1] + B[N – 1] where N is the size of array) and with the indices of elements from both arrays (N – 1, N – 1). The tuple inside max heap will be (A[N-1] + B[N – 1], N – 1, N – 1). Heap is ordered by first value i.e sum of both elements.
  4. Pop the heap to get the current largest sum and along with the indices of the element that make up the sum. Let the tuple be (sum, i, j).
  4.1. Next insert (A[i – 1] + B[j], i – 1, j) and (A[i] + B[j – 1], i, j – 1) into the max heap but make sure that the pair of indices i.e (i – 1, j) and (i, j – 1) are not
  already present in the max heap. To check this we can use set in C++.
  4.2 Go back to 4 until K times.

```
void KMaxCombinations(vector<int>& A, vector<int>& B, int K)
{
    sort(A.begin(), A.end());
    sort(B.begin(), B.end());

    int N = A.size();
    priority_queue<pair<int, pair<int, int> > > pq;
    set<pair<int, int> > my_set;

    pq.push(make_pair(A[N - 1] + B[N - 1], make_pair(N-1, N-1)));

    my_set.insert(make_pair(N - 1, N - 1));
    for (int count=0; count<K; count++) {
        pair<int, pair<int, int> > temp = pq.top();
        pq.pop();
        cout << temp.first << endl;

        int i = temp.second.first;
        int j = temp.second.second;
        int sum = A[i - 1] + B[j];
        pair<int, int> temp1 = make_pair(i - 1, j);
        if (my_set.find(temp1) == my_set.end()) {
            pq.push(make_pair(sum, temp1));
            my_set.insert(temp1);
        }
        sum = A[i] + B[j - 1];
        temp1 = make_pair(i, j - 1);
        if (my_set.find(temp1) == my_set.end()) {
            pq.push(make_pair(sum, temp1));
            my_set.insert(temp1);
        }
    }
}
```

- **K maximum sums of non-overlapping contiguous sub-arrays** :- Given an Array of Integers and an Integer value k, find out k non-overlapping sub-arrays which have k maximum sums.

  Kadane’s algorithm finds out only the maximum subarray sum, but using the same algorithm we can find out k maximum non-overlapping subarray sums. The approach is:

  1) Find out the maximum subarray in the array using Kadane’s algorithm. Also find out its starting and end indices. Print the sum of this subarray.
  2) Fill each cell of this subarray by -infinity.
  3) Repeat process 1 and 2 for k times.

```
void kmax(int arr[], int k, int n) {
    for(int c = 0; c < k; c++){
        int max_so_far = numeric_limits<int>::min();
        int max_here = 0;
        int start = 0, end = 0, s = 0;
        for(int i = 0; i < n; i++)
        {
            max_here += arr[i];
            if (max_so_far < max_here)
            {
                max_so_far = max_here;
                start = s;
                end = i;
            }
            if (max_here < 0)
            {
                max_here = 0;
                s = i + 1;
            }
        }
        cout << "Maximum non-overlapping sub-array sum"
             << (c + 1) << ": "<< max_so_far
             << ", starting index: " << start
             << ", ending index: " << end << "." << endl;
        for (int l = start; l <= end; l++)
            arr[l] = numeric_limits<int>::min();
    }
    cout << endl;
}
```

- **K maximum sums of overlapping contiguous sub-arrays** :- Given an Array of Integers and an Integer value k, find out k sub-arrays(may be overlapping) which have k maximum sums.

  Method for k-maximum sub-arrays:

```
1. Calculate the prefix sum of the input array.
2. Take cand, maxi and mini as arrays of size k.
3. Initialize mini[0] = 0 for the same reason as previous.
4. for each value of the prefix_sum[i] do
       (i). update cand[j] value by prefix_sum[i] - mini[j]
       (ii). maxi will be the maximum k elements of maxi and cand
       (iii). if prefix_sum is minimum than all values of mini then
              include it in mini and remove maximum element form mini
       // After the ith iteration mini holds k minimum prefix sum upto
       // index i and maxi holds k maximum overlapping sub-array sums
       // upto index i.
5. return maxi
```

  Mini :- to store the k smallest prefix sum till i
  Maxi :- to store the k largest subarray sums
  Cand :- for every i k candidates are calculated by subtracting k mini[j] from prefix sum till i

```
void maxMerge(vector<int>& maxi, vector<int> cand)
{
    // Here cand and maxi arrays are in non-increasing
    // order beforehand. Now, j is the index of the
    // next cand element and i is the index of next
    // maxi element. Traverse through maxi array.
    // If cand[j] > maxi[i] insert cand[j] at the ith
    // position in the maxi array and remove the minimum
    // element of the maxi array i.e. the last element
    // and increase j by 1 i.e. take the next element
    // from cand.
    int k = maxi.size();
    int j = 0;
    for (int i = 0; i < k; i++) {
        if (cand[j] > maxi[i]) {
            maxi.insert(maxi.begin() + i, cand[j]);
            maxi.erase(maxi.begin() + k);
            j += 1;
        }
    }
}

// Insert prefix_sum[i] to mini array if needed
void insertMini(vector<int>& mini, int pre_sum)
{
    // Traverse the mini array from left to right.
    // If prefix_sum[i] is less than any element
    // then insert prefix_sum[i] at that position
    // and delete maximum element of the mini array
    // i.e. the rightmost element from the array.
    int k = mini.size();
    for (int i = 0; i < k; i++) {
        if (pre_sum < mini[i]) {
            mini.insert(mini.begin() + i, pre_sum);
            mini.erase(mini.begin() + k);
            break;
        }
    }
}

// Function to compute k maximum overlapping sub-
// array sums
void kMaxOvSubArray(vector<int> arr, int k)
{
    int n = arr.size();
    vector<int> pre_sum = prefix_sum(arr, n);
    vector<int> mini(k, numeric_limits<int>::max());
    mini[0] = 0;

    vector<int> maxi(k, numeric_limits<int>::min());
    vector<int> cand(k);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < k; j++) {
             if(pre_sum[i] < 0 && mini[j]==numeric_limits<int>::max())
                cand[j]=(-pre_sum[i])-mini[j]; // only to prevent overflow
in pre_sum[i] - mini[j];
             else
                cand[j] = pre_sum[i] - mini[j];
        }
        maxMerge(maxi, cand);
        insertMini(mini, pre_sum[i]);
    }
    for (int ele : maxi)
        cout << ele << " ";
    cout << endl;
}
```

- **Given an array arr[], find the maximum j – i such that arr[j] > arr[i] :-**
  To solve this problem, we need to get two optimum indexes of arr[]: left index i and right index j. For an element arr[i], we do not need to consider arr[i] for left index if there is an element smaller than arr[i] on left side of arr[i]. Similarly, if there is a greater element on right side of arr[j] then we do not need to consider this j for right index. So we construct two auxiliary arrays LMin[] and RMax[] such that LMin[i] holds the smallest element on left side of arr[i] including arr[i], and RMax[j] holds the greatest element on right side of arr[j] including arr[j]. After constructing these two auxiliary arrays, we traverse both of these arrays from left to right. While traversing LMin[] and RMa[] if we see that LMin[i] is greater than RMax[j], then we must move ahead in LMin[] (or do i++) because all elements on left of LMin[i] are greater than or equal to LMin[i]. Otherwise we must move ahead in RMax[j] to look for a greater j – i value.

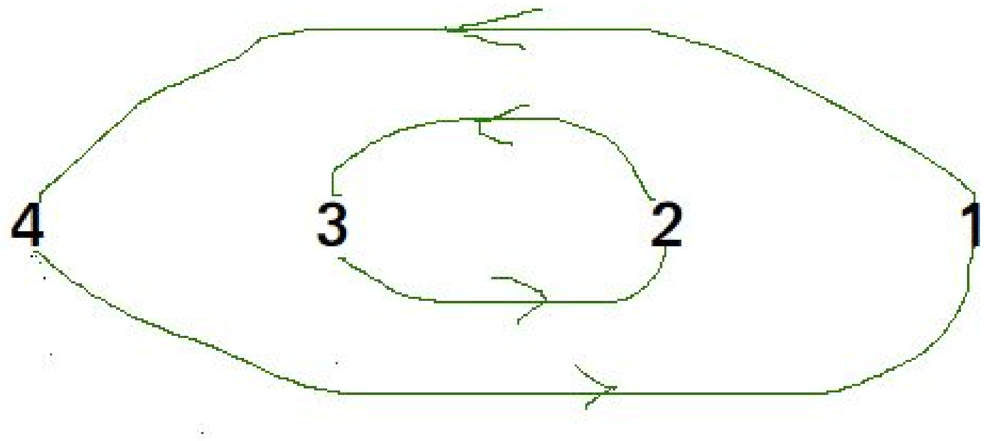
```
int maxIndexDiff(int arr[], int n)
{
    int maxDiff;
    int i, j;

    int *LMin = new int[(sizeof(int) * n)];
    int *RMax = new int[(sizeof(int) * n)];

    LMin[0] = arr[0];
    for (i = 1; i < n; ++i)
        LMin[i] = min(arr[i], LMin[i - 1]);

    RMax[n - 1] = arr[n - 1];
    for (j = n - 2; j >= 0; --j)
        RMax[j] = max(arr[j], 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;
}
```

- **Median of Stream of Running Integers using STL :-**
  The idea is to use max heap and min heap to store the elements of higher half and lower half. Max heap and min heap can be implemented using priority_queue in C++ STL. Below is the step by step algorithm to solve this problem.
  Algorithm:

1) Create two heaps. One max heap to maintain elements of lower half and one min heap to maintain elements of higher half at any point of time..
2) Take initial value of median as 0.
3) For every newly read element, insert it into either max heap or min heap and calulate the median based on the following conditions:
   a) If the size of max heap is greater than size of min heap and the element is less than previous median then pop the top element from max heap and insert into min heap and insert the new element to max heap else insert the new element to min heap. Calculate the new median as average of top of elements of both max and min heap.
   b) If the size of max heap is less than size of min heap and the element is greater than previous median then pop the top element from min heap and insert into max heap and insert the new element to min heap else insert the new element to max heap. Calculate the new median as average of top of elements of both max and min heap.
   c) If the size of both heaps are same. Then check if current is less than previous median or not. If the current element is less than previous median then insert it to max heap and new median will be equal to top element of max heap. If the current element is greater than previous median then insert it to min heap and new median will be equal to top element of min heap.

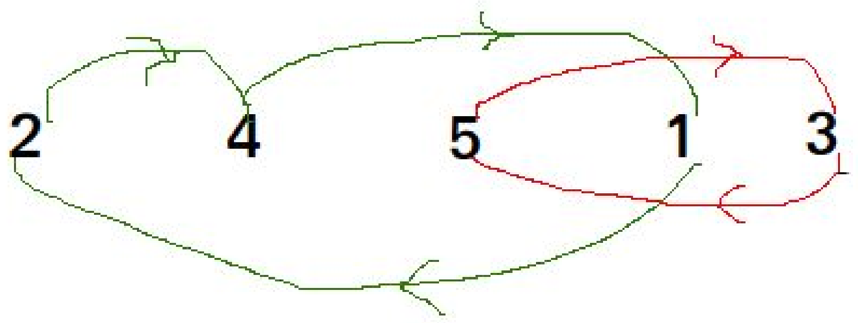
```
void printMedians(double arr[], int n)
{
    priority_queue<double> s;
    priority_queue<double,vector<double>,greater<double> > g;
    double med = arr[0];
    s.push(arr[0]);
    cout << med << endl;
    for (int i=1; i < n; i++)
    {
        double x = arr[i];
        if (s.size() > g.size())
        {
            if (x < med)
            {
                g.push(s.top());
                s.pop();
                s.push(x);
            }
            else
                g.push(x);

            med = (s.top() + g.top())/2.0;
        }
        else if (s.size()==g.size())
        {
            if (x < med)
            {
                s.push(x);
                med = (double)s.top();
            }
            else
            {
                g.push(x);
                med = (double)g.top();
            }
        }
        else
        {
            if (x > med)
            {
                s.push(g.top());
                g.pop();
                g.push(x);
            }
            else
                s.push(x);

            med = (s.top() + g.top())/2.0;
        }
        cout << med << endl;
    }
}
```

- **Minimum number of swaps required to sort an array :-**
  https://www.geeksforgeeks.org/minimum-number-swaps-required-sort-array/
  This can be easily done by visualizing the problem as a graph. We will have **n** nodes and an edge directed from node **i** to node **j** if the element at i'th index must be present at j'th index in the sorted array.



Graph for {4, 3, 2, 1}

The graph will now contain many non-intersecting cycles. Now a cycle with 2 nodes will only require 1 swap to reach the correct ordering, similarly a cycle with 3 nodes will only require 2 swap to do so.



Graph for {4, 5, 2, 1, 5}

Hence,

    ans = **Σi = 1k**(cycle_size - 1)
where **k** is the number of cycles

Better implementation of the above idea :-
https://www.geeksforgeeks.org/minimum-swap-required-convert-binary-tree-binary-search-tree/

- **Number of swaps to sort when only adjacent swapping allowed (Count Inversions):-**
  Given an array arr[] of non negative integers. We can perform a swap operation on any two adjacent elements in the array. Find the minimum number of swaps needed to sort the array in ascending order.

  There is an interesting solution to this problem. It can be solved using the fact that number of swaps needed is equal to number of inversions. So we basically need to count inversions in array.

  The fact can be established using below observations:

  1) A sorted array has no inversions.

  2) An adjacent swap can reduce one inversion. Doing x adjacent swaps can reduce x inversions in an array.

```
int merge(int arr[], int temp[], int left, int mid, int right)
{
    int inv_count = 0;
    int i = left; /* i is index for left subarray*/
    int j = mid;  /* i is index for right subarray*/
    int k = left; /* i is index for resultant merged subarray*/

    while ((i <= mid - 1) && (j <= right))
    {
        if (arr[i] <= arr[j])
            temp[k++] = arr[i++];
        else
        {
            temp[k++] = arr[j++];
            inv_count = inv_count + (mid - i);
        }
    }
    while (i <= mid - 1)
        temp[k++] = arr[i++];
    while (j <= right)
        temp[k++] = arr[j++];

    for (i=left; i <= right; i++)
        arr[i] = temp[i];

    return inv_count;
}

int _mergeSort(int arr[], int temp[], int left, int right)
{
    int mid, inv_count = 0;
    if (right > left)
    {
        mid = (right + left)/2;
        inv_count  = _mergeSort(arr, temp, left, mid);
        inv_count += _mergeSort(arr, temp, mid+1, right);
        inv_count += merge(arr, temp, left, mid+1, right);
    }
    return inv_count;
}
int countSwaps(int arr[], int n)
{
    int temp[n];
    return _mergeSort(arr, temp, 0, n - 1);
}
```

- **Find the maximum subset XOR of a given set :-**
  https://www.geeksforgeeks.org/find-maximum-subset-xor-given-set/

  The idea is based on below facts:

  Number of bits to represent all elements is fixed which is 32 bits for integer in most of the compilers.
  If maximum element has Most Significant Bit MSB at position i, then result is at least 2i

```
1. Initialize index of chosen elements as 0. Let this index be
   'index'
2. Traverse through all bits starting from most significant bit.
   Let i be the current bit.
......(a) Find the maximum element with i'th bit set.  If there
          is no element with i'th bit set, continue to smaller
          bit.
......(b) Let the element with i'th bit set be maxEle and index
          of this element be maxInd. Place maxEle at 'index' and
          (by swapping set[index] and set[maxInd])
......(c) Do XOR of maxEle with all numbers having i'th  bit as set.
......(d) Increment index
3. Return XOR of all elements in set[]. Note that set[] is modified
   in step 2.c.
```

→ **How does this work?**

Let us first understand a simple case when all elements have Most Significant Bits (MSBs) at different positions. The task in this particular case is simple, we need to do XOR of all elements.
If input contains multiple numbers with the same MSB, then it’s not obvious which of them we should choose to include in the XOR. What we do is reduce the input list into an equivalent form that doesn’t contain more than one number of the same length. By taking the maximum element, we know that the MSB of this is going to be there in output. Let this MSB be at position i. If there are more elements with i’th set (or same MSB), we XOR them with the maximum number so that the i’th bit becomes 0 in them and problem reduces to i-1 bits.

→ **Illustration:**

Let the input set be : {9, 8, 5}

We start from 31st bit [Assuming Integers are 32 bit long]. The loop will continue without doing anything till 4'th bit.

The 4th bit is set in set[0] i.e. 9 and this is the maximum element with 4th bit set. So we choose this element and check if any other number has the same bit set. If yes, we XOR that number with 9. The element set[1], i.e., 8 also has 4'th bit set. Now set[] becomes {9, 1, 5}. We add 9 to the list of chosen elements by incrementing 'index'

We move further and find the maximum number with 3rd bit set which is set[2] i.e. 5 No other number in the array has 3rd bit set. 5 is also added to the list of chosen element.

We then iterate for bit 2 (no number for this) and then for 1 which is 1. But numbers 9 and 5 have the 1st bit set. Thus we XOR 9 and 5 with 1 and our set becomes (8, 1, 4)

Finally, we XOR current elements of set and get the result as 8 ^ 1 ^ 4 = 13.

```
int maxSubarrayXOR(int set[], int n)
{
    int index = 0;
    for (int i = INT_BITS-1; i >= 0; i--)
    {
        int maxInd = index;
        int maxEle = INT_MIN;
        for (int j = index; j < n; j++)
        {
            if ( (set[j] & (1 << i)) != 0
                     && set[j] > maxEle )
                maxEle = set[j], maxInd = j;
        }
        if (maxEle == INT_MIN)
        continue;

        swap(set[index], set[maxInd]);
        maxInd = index;
        for (int j=0; j<n; j++)
        {
            if (j != maxInd &&
                (set[j] & (1 << i)) != 0)
                 set[j] = set[j] ^ set[maxInd];
        }
        index++;
    }
    int res = 0;
    for (int i = 0; i < n; i++)
        res ^= set[i];
    return res;
}
```

- **Find Subarray with given sum :-**
  - For non-negative numbers (2 pointer technique):-
    https://www.geeksforgeeks.org/find-subarray-with-given-sum/
    Initialize a variable curr_sum as first element. curr_sum indicates the sum of current subarray. Start from the second element and add all elements one by one to the curr_sum. If curr_sum becomes equal to sum, then print the solution. If curr_sum exceeds the sum, then remove trailing elements while curr_sum is greater than sum.

    ```
    int subArraySum(int arr[], int n, int sum)
    {
        int curr_sum = arr[0], start = 0, i;
        for (i = 1; i <= n; i++)
        {
            while (curr_sum > sum && start < i-1)
            {
                curr_sum = curr_sum - arr[start];
                start++;
            }
            if (curr_sum == sum)
            {
                printf ("Sum found between indexes %d and %d", start,i-1);
                return 1;
            }
            if (i < n)
              curr_sum = curr_sum + arr[i];
        }
        printf("No subarray found");
        return 0;
    }
    ```

    ```
    My algo :-
    int i = 0, j = 0;
    while(j < n)
         while(j < n && curr_sum < sum)
              Curr_sum += arr[j++];
         if(curr_sum == sum)
              Return 1;
         while(i < j && curr_sum > sum)
              Curr_sum -= arr[i++];
    ```

  - Also handles negative numbers (Using hashmap) :-

https://www.geeksforgeeks.org/find-subarray-with-given-sum-in-array-of-integers/
An efficient way is to use a map. The idea is to maintain sum of elements encountered so far in a variable (say curr_sum). Let the given number is sum. Now for each element, we check if curr_sum – sum exists in the map or not. If we found it in the map that means, we have a subarray present with given sum, else we insert curr_sum into the map and proceed to next element. If all elements of the array are processed and we didn't find any subarray with given sum, then subarray doesn't exists.

```
void subArraySum(int arr[], int n, int sum)
{
    unordered_map<int, int> map;
    int curr_sum = 0;
    for (int i = 0; i < n; i++)
    {
        curr_sum = curr_sum + arr[i];
        if (curr_sum == sum)
        {
            cout << "Sum found between indexes "
                 << 0 << " to " << i << endl;
            return;
        }
        if (map.find(curr_sum - sum) != map.end())
        {
            cout << "Sum found between indexes "
                 << map[curr_sum - sum] + 1
                 << " to " << i << endl;
            return;
        }

        map[curr_sum] = i;
    }
    cout << "No subarray with given sum exists";
}
```

-> These can also be used to print all the subarrays with given sum and questions where sum is 0.

- **Check if array elements are consecutive numbers :-**
    - Use hashmap, visited array, sorting
    - O(n) | O(1) solution :-
        - If only positive numbers
          https://www.geeksforgeeks.org/check-if-array-elements-are-consecutive/
          The idea is to traverse the array and for each index i (where 0 ≤ i < n), make arr[arr[i] – min]] as a negative value. If we see a negative value again then there is repetition.

```
bool areConsecutive(int arr[], int n)
{
    if ( n <  1 )
        return false;
    int min = getMin(arr, n);
    int max = getMax(arr, n);
```

```
                if (max - min  + 1 == n)
                {
                    int i;
                    for(i = 0; i < n; i++)
                    {
                        int j;
                        if (arr[i] < 0)
                            j = -arr[i] - min;
                        else
                            j = arr[i] - min;

                        if (arr[j] > 0)
                            arr[j] = -arr[j];
                        else
                            return false;
                    }
                    return true;
                }
                return false; // if (max - min  + 1 != n)
            }
```

- ■ For negative numbers also
  An important assumption here is elements are distinct.

  Find the sum of the array.
  If given array elements are consecutive that means they are in AP.
  So, find min element i.e. first term of AP then calculate ap_sum = n/2 * [2a +(n-1)*d] where d = 1. So, ap_sum = n/2 * [2a +(n-1)]
  Compare both sums. Print Yes if equal, else No.

- **Majority Element :-**
  https://www.geeksforgeeks.org/majority-element/
  A majority element in an array A[] of size n is an element that appears more than n/2 times (and hence there is at most one such element).

  → Moore’s Voting Algorithm O(n) :-
  This is a two step process.

  NOTE : This Method only works when we are given that majority element do exist in the array , otherwise this method won’t work , as in the problem definition we said that majority element may or may not exist but for applying this approach you can assume that majority element do exist in the given input array
  1. Finding a Candidate :
  The algorithm for first phase that works in O(n) is known as Moore’s Voting Algorithm. Basic idea of the algorithm is that if we cancel out each occurrence of an element e with all the other elements that are different from e then e will exist till end if it is a majority element.

```
    findCandidate(a[], size)
    1.  Initialize index and count of majority element
         maj_index = 0, count = 1
    2.  Loop for i = 1 to size - 1
        (a) If a[maj_index] == a[i]
               count++
        (b) Else
```

```
              count--;
          (c) If count == 0
                maj_index = i;
                count = 1
      3.  Return a[maj_index]
```

Above algorithm loops through each element and maintains a count of a[maj_index]. If the next element is same then increment the count, if the next element is not same then decrement the count, and if the count reaches 0 then changes the maj_index to the current element and set the count again to 1. So, the first phase of the algorithm gives us a candidate element.
In the second phase we need to check if the candidate is really a majority element. Second phase is simple and can be easily done in O(n). We just need to check if count of the candidate element is greater than n/2.

2. Check if the element obtained in step 1 is majority element or not

- **N/3 Repeat Number** :- You're given a read only array of n integers. Find out if any integer occurs more than n/3 times in the array in linear time and constant additional space.

```
int Solution::repeatedNumber(const vector<int> &A) {
    int count1=0;
    int count2=0;
    int a=INT_MAX;
    int b=INT_MAX;
    for(int i=0;i<A.size();i++)
    {
        if(a==A[i])
        {
            count1++;
        }
        else if(b==A[i])
        {
            count2++;
        }
        else if(count1==0)
        {
            count1=1;
            a=A[i];
        }
        else if(count2==0)
        {
            count2=1;
            b=A[i];
        }
        else
        {
            count1--;
            count2--;
        }
    }
    count1=0;
    count2=0;
    for(int i=0;i<A.size();i++)
    {
        if(A[i]==a)
        {
            count1++;
        }
        if(A[i]==b)
        {
            count2++;
        }
    }
    int chk=A.size()/3;
    chk++;
    if(count1>=chk)
    {
        return a;
    }
    else if(count2>=chk)
    {
        return b;
    }
    else
    {
        return -1;
    }
}
```

- **Given an array of size n and a number k, find all elements that appear more than n/k times** :- Given an array of size n, find all elements in array that appear more than n/k times.

  First, sort all elements using a O(nLogn) algorithm. Once the array is sorted, we can find all required elements in a linear scan of array. So overall time complexity of this method is O(nLogn) + O(n) which is O(nLogn).

  Following is an interesting O(nk) solution:
  We can solve the above problem in O(nk) time using O(k-1) extra space. Note that there can never be more than k-1 elements in output (Why?). There are mainly three steps in this algorithm.

  1) Create a temporary array of size (k-1) to store elements and their counts (The output elements are going to be among these k-1 elements). Following is structure of temporary array elements.

  ```
  struct eleCount {
      int element;
      int count;
  };
  struct eleCount temp[];
  ```
  This step takes O(k) time.

  2) Traverse through the input array and update temp[] (add/remove an element or increase/decrease count) for every traversed element. The array temp[] stores potential (k-1) candidates at every step. This step takes O(nk) time.

3) Iterate through final (k-1) potential candidates (stored in temp[]). or every element, check if it actually has count more than n/k. This step takes O(nk) time.

The main step is step 2, how to maintain (k-1) potential candidates at every point? The steps used in step 2 are like famous game: Tetris. We treat each number as a piece in Tetris, which falls down in our temporary array temp[]. Our task is to try to keep the same number stacked on the same column (count in temporary array is incremented).

```
void moreThanNdK(int arr[], int n, int k)
{
    // k must be greater than 1 to get some output
    if (k < 2)
       return;

    /* Step 1: Create a temporary array (contains element
       and count) of size k-1. Initialize count of all
       elements as 0 */
    struct eleCount temp[k-1];
    for (int i=0; i<k-1; i++)
        temp[i].c = 0;

    /* Step 2: Process all elements of input array */
    for (int i = 0; i < n; i++)
    {
        int j;

        /* If arr[i] is already present in
           the element count array, then increment its count */
        for (j=0; j<k-1; j++)
        {
            if (temp[j].e == arr[i])
            {
                 temp[j].c += 1;
                 break;
            }
        }

        /* If arr[i] is not present in temp[] */
        if (j == k-1)
        {
            int l;

            /* If there is position available in temp[], then place
              arr[i] in the first available position and set count as 1*/
            for (l=0; l<k-1; l++)
            {
                if (temp[l].c == 0)
                {
                    temp[l].e = arr[i];
                    temp[l].c = 1;
                    break;
                }
            }
            /* If all the position in the temp[] are filled, then
               decrease count of every element by 1 */
            if (l == k-1)
                for (l=0; l<k; l++)
                    temp[l].c -= 1;
        }
    }

    /*Step 3: Check actual counts of potential candidates in temp[]*/
    for (int i=0; i<k-1; i++)
    {
        // Calculate actual count of elements
        int ac = 0;  // actual count
        for (int j=0; j<n; j++)
            if (arr[j] == temp[i].e)
                ac++;

        // If actual count is more than n/k, then print it
        if (ac > n/k)
           cout << "Number:" << temp[i].e
                << " Count:" << ac << endl;
    }
}
```

- **Find the length of largest subarray with 0 sum :-**
  We can Use Hashing to solve this problem in O(n) time. The idea is to iterate through the array and for every element arr[i], calculate sum of elements form 0 to i (this can simply be done as sum += arr[i]). If the current sum has been seen before, then there is a zero sum array. Hashing is used to store the sum values, so that we can quickly store sum and find out whether the current sum is seen before or not.

```
int maxLen(int arr[], int n)
{
    unordered_map<int, int> presum;

    int sum = 0;        // Initialize the sum of elements
    int max_len = 0;
    for(int i=0; i<n; i++)
    {
        sum += arr[i];
        if (arr[i]==0 && max_len==0)
            max_len = 1;
        if (sum == 0)
            max_len = i+1;
        if(presum.find(sum) != presum.end())
        {
            max_len = max(max_len, i-presum[sum]);
        }
        else
        {
            presum[sum] = i;
        }
    }
    return max_len;
}
```

- **Find the smallest positive integer value that cannot be represented as sum of any subset of a given array :-**
  Given a sorted array (sorted in non-decreasing order) of positive numbers, find the smallest positive integer value that cannot be represented as sum of elements of any subset of given set.
  Expected time complexity is O(n).

  A Simple Solution is to start from value 1 and check all values one by one if they can sum to values in the given array. This solution is very inefficient as it reduces to subset sum problem which is a well known NP Complete Problem.

  We can solve this problem in O(n) time using a simple loop. Let the input array be arr[0..n-1]. We initialize the result as 1 (smallest possible outcome) and traverse the given array. Let the smallest element that cannot be represented by elements at indexes from 0 to (i-1) be 'res', there are following two possibilities when we consider element at index i:

  1) We decide that 'res' is the final result: If arr[i] is greater than 'res', then we found the gap which is 'res' because the elements after arr[i] are also going to be greater than 'res'.

  2) The value of 'res' is incremented after considering arr[i]: The value of 'res' is incremented by arr[i] (why? If elements from 0 to (i-1) can represent 1 to 'res-1', then elements from 0 to i can represent from 1 to 'res + arr[i] - 1' be adding 'arr[i]' to all subsets that represent 1 to 'res')

```
int findSmallest(int arr[], int n)
{
   int res = 1;
   for (int i = 0; i < n && arr[i] <= res; i++)
       res = res + arr[i];
   return res;
}
```

- **Find maximum of minimum for every window size in a given array :-**

```
Input:  arr[] = {10, 20, 30, 50, 10, 70, 30}
Output:         70, 30, 20, 10, 10, 10, 10
```

  Step 1: Find indexes of next smaller and previous smaller for every element. Next smaller is the nearest smallest element on right side of arr[i].
  Similarly, previous smaller element is the nearest smallest element on left side of arr[i].
  If there is no smaller element on right side, then next smaller is n.If there is no smaller on left side, then previous smaller is -1.

  For input {10, 20, 30, 50, 10, 70, 30}, array of indexes of next smaller is {7, 4, 4, 4, 7, 6, 7}.
  For input {10, 20, 30, 50, 10, 70, 30}, array of indexes of previous smaller is {-1, 0, 1, 2, -1, 4, 4}

This step can be done in O(n) time using the approach discussed in next greater element.

Step 2: Once we have indexes of next and previous smaller, we know that arr[i] is a minimum of a window of length “right[i] – left[i] – 1”. Lengths of windows for which the elements are minimum are {7, 3, 2, 1, 7, 1, 2}. This array indicates, first element is minimum in window of size 7, second element is minimum in window of size 3, and so on.

Create an auxiliary array ans[n+1] to store the result. Values in ans[] can be filled by iterating through right[] and left[]

```
    for (int i=0; i < n; i++)
    {
        int len = right[i] - left[i] - 1;
        ans[len] = max(ans[len], arr[i]);
    }
```

We get the ans[] array as {0, 70, 30, 20, 0, 0, 0, 10}. Note that ans[0] or answer for length 0 is useless.

Step 3:Some entries in ans[] are 0 and yet to be filled. For example, we know maximum of minimum for lengths 1, 2, 3 and 7 are 70, 30, 20 and 10 respectively, but we don’t know the same for lengths 4, 5 and 6.
Below are few important observations to fill remaining entries
a) Result for length i, i.e. ans[i] would always be greater or same as result for length i+1, i.e., ans[i+1].
b) If ans[i] is not filled it means there is no direct element which is minimum of length i and therefore either the element of length ans[i+1], or ans[i+2], and so on is same as ans[i]
So we fill rest of the entries using below loop.

```
    for (int i=n-1; i>=1; i--)
        ans[i] = max(ans[i], ans[i+1]);
```

```
void printMaxOfMin(int arr[], int n)
{
    stack<int> s;
    int left[n+1];
    int right[n+1];
    for (int i=0; i<n; i++)
    {
        left[i] = -1;
        right[i] = n;
    }
    for (int i=0; i<n; i++)
    {
        while (!s.empty() && arr[s.top()] >= arr[i])
            s.pop();
        if (!s.empty())
            left[i] = s.top();
        s.push(i);
    }
    while (!s.empty())
        s.pop();
    for (int i = n-1 ; i>=0 ; i-- )
    {
        while (!s.empty() && arr[s.top()] >= arr[i])
            s.pop();
        if(!s.empty())
            right[i] = s.top();
        s.push(i);
    }
    int ans[n+1];
    for (int i=0; i<=n; i++)
        ans[i] = 0;

    for (int i=0; i<n; i++)
    {
        int len = right[i] - left[i] - 1;
        ans[len] = max(ans[len], arr[i]);
    }
    for (int i=n-1; i>=1; i--)
        ans[i] = max(ans[i], ans[i+1]);

    for (int i=1; i<=n; i++)
        cout << ans[i] << " ";
}
```

- **Find the maximum repeating number in O(n) time and O(1) extra space :-**
  Given an array of size n, the array contains numbers in range from 0 to k-1 where k is a positive integer and k <= n. Find the maximum repeating number in this array. For example, let k be 10 the given array be arr[] = {1, 2, 2, 2, 0, 2, 0, 2, 3, 8, 0, 9, 2, 3}, the maximum repeating number would be 2. Expected time complexity is O(n) and extra space allowed is O(1). Modifications to array are allowed.

  Following is the O(n) time and O(1) extra space approach. Let us understand the approach with a simple example where arr[] = {2, 3, 3, 5, 3, 4, 1, 7}, k = 8, n = 8 (number of elements in arr[]).

  1) Iterate though input array arr[], for every element arr[i], increment arr[arr[i]%k] by k (arr[] becomes {2, 11, 11, 29, 11, 12, 1, 15 })

  2) Find the maximum value in the modified array (maximum value is 29). Index of the maximum value is the maximum repeating element (index of 29 is 3).

  3) If we want to get the original array back, we can iterate through the array one more time and do arr[i] = arr[i] % k where i varies from 0 to n-1.

  How does the above algorithm work? Since we use arr[i]%k as index and add value k at the index arr[i]%k, the index which is equal to maximum repeating element will have the maximum value in the end. Note that k is added maximum number of times at the index equal to maximum repeating element and all array elements are smaller than k.

  Note :- This is similar to storing 2 numbers at a single index. Here the count is stored as num multiplied by k (which is the maximum element in the array).

- **Minimum number of elements to add to make median equals x :-**

Better approach is to count all the elements equal to x(that is e), greater than x(that is h) and smaller than x(that is l). And then -
if l is greater than h then, the ans will be (l - h) + 1 - e;
And if h is greater than l then, ans will be (h - l - 1) + 1 - e;

- **Find zeroes to be flipped so that number of consecutive 1's is maximized :-**
  Given a binary array and an integer m, find the position of zeroes flipping which creates maximum number of consecutive 1's in array.

  A Better Solution is to use auxiliary space to solve the problem in O(n) time.

  For all positions of 0's calculate left[] and right[] which defines the number of consecutive 1's to the left of i and right of i respectively.

  For example, for arr[] = {1, 1, 0, 1, 1, 0, 0, 1, 1, 1} and m = 1, left[2] = 2 and right[2] = 2, left[5] = 2 and right[5] = 0, left[6] = 0 and right[6] = 3.

  left[] and right[] can be filled in O(n) time by traversing array once and keeping track of last seen 1 and last seen 0. While filling left[] and right[], we also store indexes of all zeroes in a third array say zeroes[]. For above example, this third array stores {2, 5, 6}

  Now traverse zeroes[] and for all consecutive m entries in this array, compute the sum of 1s that can be produced. This step can be done in O(n) using left[] and right[].

  An Efficient Solution can solve the problem in O(n) time and O(1) space. The idea is to use Sliding Window for the given array. The solution is taken from here.
  Let us use a window covering from index wL to index wR. Let the number of zeros inside the window be zeroCount. We maintain the window with at most m zeros inside.

  The main steps are:
  - While zeroCount is no more than m: expand the window to the right (wR++) and update the count zeroCount.
  - While zeroCount exceeds m, shrink the window from left (wL++), update zeroCount;
  - Update the widest window along the way. The positions of output zeros are inside the best window.

```
void findZeroes(int arr[], int n, int m)
{
    int wL = 0, wR = 0;
    int bestL = 0, bestWindow = 0;
    int zeroCount = 0;
    while (wR < n)
    {
        if (zeroCount <= m)
        {
            if (arr[wR] == 0)
              zeroCount++;
            wR++;
        }
        if (zeroCount > m)
        {
            if (arr[wL] == 0)
              zeroCount--;
            wL++;
        }
        if ((wR-wL > bestWindow) && (zeroCount<=m))
        {
            bestWindow = wR-wL;
            bestL = wL;
        }
    }
    for (int i=0; i<bestWindow; i++)
    {
        if (arr[bestL+i] == 0)
           cout << bestL+i << " ";
    }
}
```

- **Find the smallest positive number missing from an unsorted array :-**
  A O(n) time and O(1) extra space solution:
  The idea is similar to this post. We use array elements as index. To mark presence of an element x, we change the value at the index x to negative. But this approach doesn’t work if there are non-positive (-ve and 0) numbers. So we segregate positive from negative numbers as first step and then apply the approach.

  Following is the two step algorithm.
  1) Segregate positive numbers from others i.e., move all non-positive numbers to left side. In the following code, segregate() function does this part.
  2) Now we can ignore non-positive elements and consider only the part of array which contains all positive elements. We traverse the array containing all positive numbers and to mark presence of an element x, we change the sign of value at index x to negative. We traverse the array again and print the first index which has positive value. In the following code, findMissingPositive() function does this part. Note that in findMissingPositive, we have subtracted 1 from the values as indexes start from 0 in C.

- **Find the minimum distance between two numbers :-**
  Given an unsorted array arr[] and two numbers x and y, find the minimum distance between x and y in arr[]. The array might also contain duplicates. You may assume that both x and y are different and present in arr[].
  1) Traverse array from left side and stop if either x or y is found. Store index of this first occurrence in a variable say prev
  2) Now traverse arr[] after the index prev. If the element at current index i matches with either x or y then check if it is different from arr[prev]. If it is different then update the minimum distance if needed. If it is same then update prev i.e., make prev = i.

- **Find the Increasing subsequence of length three with maximum product :-**
  LSL: The largest smaller element on left of given element
  LGR: The largest greater element on right of given element.

  We can do this in O(nLogn) time. For simplicity, let us first create two arrays LSL[] and LGR[] of size n each where n is number of elements in input array arr[]. The main task is to fill two arrays LSL[] and LGR[]. Once we have these two arrays filled, all we need to find maximum product LSL[i]*arr[i]*LGR[i] where 0 < i < n-1 (Note that LSL[i] doesn't exist for i = 0 and LGR[i] doesn't exist for i = n-1). We can fill LSL[] in O(nLogn) time. The idea is to use a Balanced Binary Search Tree like AVL. We start with empty AVL tree, insert the leftmost element in it. Then we traverse the input array starting from the second element to second last element. For every element currently being traversed, we find the floor of it in AVL tree. If floor exists, we store the floor in LSL[], otherwise we store NIL. After storing the floor, we insert the current element in the AVL tree.

We can fill LGR[] in O(n) time. The idea is similar to this post. We traverse from right side and keep track of the largest element. If the largest element is greater than current element, we store it in LGR[], otherwise we store NIL.

Finally, we run a O(n) loop and return maximum of LSL[i]*arr[i]*LGR[i]

- **Maximum Sum Path in Two Arrays :-**

Given two sorted arrays such the arrays may have some common elements. Find the sum of the maximum sum path to reach from beginning of any array to end of any of the two arrays. We can switch from one array to another array only at common elements. Note that the common elements do not have to be at same indexes.

Expected time complexity is O(m+n) where m is the number of elements in ar1[] and n is the number of elements in ar2[].

The idea is to do something similar to merge process of merge sort. We need to calculate sums of elements between all common points for both arrays. Whenever we see a common point, we compare the two sums and add the maximum of two to the result. Following are detailed steps.

1) Initialize result as 0. Also initialize two variables sum1 and sum2 as 0. Here sum1 and sum2 are used to store sum of element in ar1[] and ar2[] respectively. These sums are between two common points.

2) Now run a loop to traverse elements of both arrays. While traversing compare current elements of ar1[] and ar2[].

2.a) If current element of ar1[] is smaller than current element of ar2[], then update sum1, else if current element of ar2[] is smaller, then update sum2.

2.b) If current element of ar1[] and ar2[] are same, then take the maximum of sum1 and sum2 and add it to the result. Also add the common element to the result.

```
int maxPathSum(int ar1[], int ar2[], int m, int n)
{
    int i = 0, j = 0;
    int  result = 0, sum1 = 0, sum2 = 0;
    while (i < m && j < n)
    {
        if (ar1[i] < ar2[j])
            sum1 += ar1[i++];
        else if (ar1[i] > ar2[j])
```

```
            sum2 += ar2[j++];
        else  // we reached a common point
        {
            result += max(sum1, sum2);
            sum1 = 0, sum2 = 0;
            while (i < m &&  j < n && ar1[i] == ar2[j])
            {
                result = result + ar1[i++];
                j++;
            }
        }
    }
    while (i < m)
        sum1  +=  ar1[i++];
    while (j < n)
        sum2 +=  ar2[j++];
    result +=  max(sum1, sum2);
    return result;
}
```

- **Program for array rotation :-**
  - **Using temp array :-**

    ```
    Input arr[] = [1, 2, 3, 4, 5, 6, 7], d = 2, n =7
    1) Store d elements in a temp array
       temp[] = [1, 2]
    2) Shift rest of the arr[]
       arr[] = [3, 4, 5, 6, 7, 6, 7]
    3) Store back the d elements
       arr[] = [3, 4, 5, 6, 7, 1, 2]
    ```

    Time complexity : O(n)
    Auxiliary Space : O(d)

  - **Rotate one by one :-**

    ```
    void leftRotatebyOne(int arr[], int n)
    {
        int temp = arr[0], i;
        for (i = 0; i < n - 1; i++)
            arr[i] = arr[i + 1];

        arr[i] = temp;
    }
    void leftRotate(int arr[], int d, int n)
    {
        for (int i = 0; i < d; i++)
            leftRotatebyOne(arr, n);
    }
    ```

    Time complexity : O(n * d)
    Auxiliary Space : O(1)

  - **A Juggling Algorithm :-**
    This is an extension of method 2. Instead of moving one by one, divide the array in different sets
    where number of sets is equal to GCD of n and d and move the elements within sets.

If GCD is 1 as is for the above example array (n = 7 and d =2), then elements will be moved within one set only, we just start with temp = arr[0] and keep moving arr[I+d] to arr[I] and finally store temp at the right place.

```
void leftRotate(int arr[], int d, int n)
{
    int g_c_d = gcd(d, n);
    for (int i = 0; i < g_c_d; i++) {
        int temp = arr[i];
        int j = i;

        while (1) {
            int k = (j + d)%n;
            if (k == i)
                break;
            arr[j] = arr[k];
            j = k;
        }
        arr[j] = temp;
    }
}
```

- **Reversal algorithm for array rotation :-**
  Algorithm :

```
rotate(arr[], d, n)
  reverse(arr[], 1, d) ;
  reverse(arr[], d + 1, n);
  reverse(arr[], 1, n);
```

Let AB are the two parts of the input array where A = arr[0..d-1] and B = arr[d..n-1]. The idea of the algorithm is :

Reverse A to get ArB, where Ar is reverse of A.
Reverse B to get ArBr, where Br is reverse of B.
Reverse all to get (ArBr) r = BA.

- **Block swap algorithm for array rotation :-**
  Algorithm :

Initialize A = arr[0..d-1] and B = arr[d..n-1]
1) Do following until size of A is equal to size of B

  a) If A is shorter, divide B into Bl and Br such that Br is of same length as A. Swap A and Br to change ABlBr into BrBlA. Now A is at its final place, so recur on pieces of B.

  b) If A is longer, divide A into Al and Ar such that Al is of same length as B Swap Al and B to change AlArB into BArAl. Now B is at its final place, so recur on pieces of A.

2) Finally when A and B are of equal size, block swap them.

```
void leftRotate(int arr[], int d, int n)
{
  if(d == 0 || d == n)
    return;
  if(n-d == d)
  {
    swap(arr, 0, n-d, d);
    return;
  }
  if(d < n-d)
  {
    swap(arr, 0, n-d, d);
    leftRotate(arr, d, n-d);
  }
  else /* If B is shorter*/
  {
    swap(arr, 0, d, n-d);
    leftRotate(arr+n-d, 2*d-n, d); /*This is tricky*/
  }
}
```

- **Maximum sum of i*arr[i] among all rotations of a given array :-**
  (Efficient Solution : O(n))
  The idea is to compute value of a rotation using value of previous rotation.
  When we rotate an array by one, following changes happen in sum of i*arr[i].
  1) Multiplier of arr[i-1] changes from 0 to n-1, i.e., arr[i-1] * (n-1) is added to current value.
  2) Multipliers of other terms is decremented by 1. i.e., (cum_sum – arr[i-1]) is subtracted from current value where cum_sum is sum of all numbers.

  ```
  next_val = curr_val - (cum_sum - arr[i-1]) + arr[i-1] * (n-1);

  int maxSum(int arr[], int n)
  {
      int cum_sum = 0;
      for (int i=0; i<n; i++)
          cum_sum += arr[i];

      int curr_val = 0;
      for (int i=0; i<n; i++)
          curr_val += i*arr[i];
      int res = curr_val;
      for (int i=1; i<n; i++)
      {
          int next_val = curr_val - (cum_sum - arr[i-1])
                         + arr[i-1] * (n-1);
          curr_val = next_val;
          res = max(res, next_val);
      }
      return res;
  }
  ```

  OR
  Using pivot: O(n)
  Ans will be max when the greatest element will be at pos n-1;

- **Duplicates in an array in O(n) and by using O(1) extra space :-**

Given an array of n elements which contains elements from 0 to n-1, with any of these numbers appearing any number of times. Find these repeating numbers in O(n) and using only constant memory space.

1- Traverse the given array from i= 0 to n-1 elements
   Go to index **arr[i]%n (this element might be already storing a count)** and increment its value by n.
3- Now traverse the array again and print all those
   indexes i for which arr[i]/n is greater than 1.

This approach works because all elements are in range from 0 to n-1 and arr[i]/n would be greater than 1 only if a value "i" has appeared more than once.

- **Find the minimum element in a sorted and rotated array :-**

```
int findMin(int arr[], int low, int high)
{
    if (high < low) return arr[0];
    if (high == low) return arr[low];

    int mid = low + (high - low)/2;
    if (mid < high && arr[mid + 1] < arr[mid])
           return arr[mid + 1];

    if (mid > low && arr[mid] < arr[mid - 1])
           return arr[mid];

    if (arr[high] > arr[mid])
           return findMin(arr, low, mid - 1);
    return findMin(arr, mid + 1, high);
}
```

How to handle duplicates?
It turned out that duplicates can't be handled in O(Logn) time in all cases. The special cases that cause problems are like {2, 2, 2, 2, 2, 2, 2, 2, 0, 1, 1, 2} and {2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2}. It doesn't look possible to go to left half or right half by doing constant number of comparisons at the middle. So the problem with repetition can be solved in O(n) worst case.

- **Search an element in a sorted and rotated array :-**

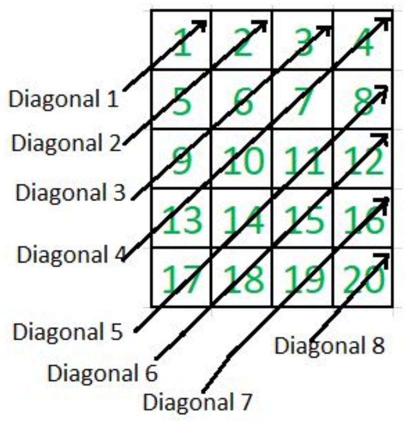
```
int search(int arr[], int l, int h, int key)
{
    if (l > h) return -1;

    int mid = (l+h)/2;
    if (arr[mid] == key) return mid;

    if (arr[l] <= arr[mid])
    {
        if (key >= arr[l] && key <= arr[mid])
             return search(arr, l, mid-1, key);
        return search(arr, mid+1, h, key);
    }
    if (key >= arr[mid] && key <= arr[h])
        return search(arr, mid+1, h, key);
    return search(arr, l, mid-1, key);
}
```

- **Find element at given index after a number of rotations :-**
  An array consisting of N integers is given. There are several Right Circular Rotations of range[L..R] that we perform. After performing these rotations, we need to find element at a given index.

  Examples :

```
Input : arr[] : {1, 2, 3, 4, 5}
        ranges[] = { {0, 2}, {0, 3} }
        index : 1
Output : 3

int findElement(int arr[], int ranges[][2], int rotations, int index)
{
    for (int i = rotations - 1; i >= 0; i--) {
        int left = ranges[i][0];
        int right = ranges[i][1];

        if (left <= index && right >= index) {
            if (index == left)
                index = right;
            else
                index--;
        }
    }
    return arr[index];
}
```

- **Zigzag (or diagonal) traversal of Matrix :-**



```
void diagonalOrder(int matrix[][COL])
{
    for (int line=1; line<=(ROW + COL -1); line++)
    {
        int start_col =  max(0, line-ROW);
         int count = min(line, (COL-start_col), ROW);
        for (int j=0; j<count; j++) .
            printf("%5d ", matrix[min(ROW, line)-j-1][start_col+j]);
        printf("\n");
    }
}
```

- **Minimum number of jumps to reach end** :- Given an array of integers where each element represents the max number of steps that can be made forward from that element. Write a function to return the minimum number of jumps to reach the end of the array (starting from the first element). If an element is 0, then we cannot move through that element

  Simple traversal / DP :- O(n*n)
  BFS/DFS or **single traversal method** = O(n)

  Implementation:
  Variables to be used:

  maxReach The variable maxReach stores at all time the maximal reachable index in the array.
  step The variable step stores the number of steps we can still take(and is initialized with value at index 0,i.e. initial number of steps)
  jump jump stores the amount of jumps necessary to reach that maximal reachable position.

```
int minJumps(int arr[], int n)
{
    if (n <= 1)
        return 0;
    if (arr[0] == 0)
        return -1;
    int maxReach = arr[0];  // stores all time the maximal reachable index in
the array.
    int step = arr[0];      // stores the number of steps we can still take
    int jump =1;

    int i=1;
    for (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;
}
```

- **Longest Span with same Sum in two arrays** :- Given two binary arrays arr1[] and arr2[] of same size n. Find length of the longest common span (i, j) where j >= i such that arr1[i] + arr1[i+1] + …. + arr1[j] = arr2[i] + arr2[i+1] + …. + arr2[j].

Just make an array with diff of arr1[i] - arr2[i], now the problem reduces to finding the largest subarray with sum 0.

```
int longestCommonSum(bool arr1[], bool arr2[], int n)
{
    int arr[n];
    for (int i=0; i<n; i++)
      arr[i] = arr1[i] - arr2[i];
    unordered_map<int, int> hM;

    int sum = 0;
    int max_len = 0;
    for (int i = 0; i < n; i++)
    {
        sum += arr[i];

        if (sum == 0)
            max_len = i + 1;
        if (hM.find(sum) != hM.end())
          max_len = max(max_len, i - hM[sum]);

        else
            hM[sum] = i;
    }
    return max_len;
}
```

- **Smallest subarray with sum greater than a given value** :- Given an array of integers and a number x, find the smallest subarray with sum greater than the given value.

```
int smallestSubWithSum(int arr[], int n, int x)
{
    int curr_sum = 0, min_len = n+1;
    int start = 0, end = 0;
    while (end < n)
    {
        while (curr_sum <= x && end < n)
            curr_sum += arr[end++];
        while (curr_sum > x && start < n)
        {
            if (end - start < min_len)
                min_len = end - start;
            curr_sum -= arr[start++];
        }
    }
    return min_len;
}
```

How to handle negative numbers?
The above solution may not work if input array contains negative numbers. For example arr[] = {- 8, 1, 4, 2, -6}. To handle negative numbers, add a condition to ignore subarrays with negative sums.

```
int smallestSubWithSum(int arr[], int n, int x)
```

```
{
    int curr_sum = 0, min_len = n+1;
    int start = 0, end = 0;
    while (end < n)
    {
        while (curr_sum <= x && end < n)
        {
            if (curr_sum <= 0 && x > 0)
            {
                start = end;
                curr_sum = 0;
            }
            curr_sum += arr[end++];
        }
        while (curr_sum > x && start < n)
        {
            if (end - start < min_len)
                min_len = end - start;
            curr_sum -= arr[start++];
        }
    }
    return min_len;
}
```

- ## Meet in the middle :-

Given a set of n integers where n <= 40. Each of them is at most 1012, determine the maximum sum subset having sum less than or equal S where S <= 10^18.

Here Dp can’t be applied as the sum is very large.

Meet in the middle is a search technique which is used when the input is small but not as small that brute force can be used. Like divide and conquer it splits the problem into two, solves them individually and then merge them. But we can’t apply meet in the middle like divide and conquer because we don’t have the same structure as the original problem.

Split the set of integers into 2 subsets say A and B. A having first n/2 integers and B having rest.
Find all possible subset sums of integers in set A and store in an array X.
Similarly calculate all possible subset sums of integers in set B and store in array Y. Hence, Size of each of the array X and Y will be at most 2n/2.
Now merge these 2 subproblems. Find combinations from array X and Y such that their sum is less than or equal to S.
One way to do that is simply iterate over all elements of array Y for each element of array X to check the existence of such a combination. This will take O((2n/2)2) which is equivalent to O(2n).
To make it less complex, first sort array Y and then iterate over each element of X and for each element x in X use binary search to find maximum element y in Y such that x + y <= S.
Binary search here helps in reducing complexity from 2nto 2n/2log(2n/2)which is equivalent to 2n/2n.
Thus our final running time is O(2n/2n).

```
void calcsubarray(ll a[], ll x[], int n, int c)
```

```
{
    for (int i=0; i<(1<<n); i++)
    {
        ll s = 0;
        for (int j=0; j<n; j++)
            if (i & (1<<j)) // bet i be something 0101010….(n digits). Those
array elements with corresponding set bits will be taken.
                s += a[j+c];
        x[i] = s;
    }
}
ll solveSubsetSum(ll a[], int n, ll S)
{
    calcsubarray(a, X, n/2, 0);
    calcsubarray(a, Y, n-n/2, n/2);

    int size_X = 1<<(n/2);
    int size_Y = 1<<(n-n/2);

    sort(Y, Y+size_Y);
    ll max = 0;
    for (int i=0; i<size_X; i++)
    {
        if (X[i] <= S)
        {
            // lower_bound() returns the first address
            // which has value greater than or equal to
            // S-X[i].
            int p = lower_bound(Y, Y+size_Y, S-X[i]) - Y;

            // If S-X[i] was not in array Y then decrease
            // p by 1
            if (p == size_Y || Y[p] != (S-X[i]))
                p--;

            if ((Y[p]+X[i]) > max)
                max = Y[p]+X[i];
        }
    }
    return max;
}
```

- **Trapping Rain Water** :- Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

```
int findWater(int arr[], int n)
{
    int left[n];
    int right[n];
    int water = 0;

    left[0] = arr[0];
    for (int i = 1; i < n; i++)
       left[i] = max(left[i-1], arr[i]);
    right[n-1] = arr[n-1];
    for (int i = n-2; i >= 0; i--)
       right[i] = max(right[i+1], arr[i]);

    for (int i = 0; i < n; i++)
       water += min(left[i],right[i]) - arr[i];
    return water;
}
```

- **Minimum swaps required to bring all elements less than or equal to k together** :- Given an array of n positive integers and a number k. Find the minimum number of swaps required to bring all the numbers less than or equal to k together.

  A simple approach is to use two pointer technique and sliding window.

  1) Find count of all elements which are less than or equals to 'k'. Let's say the count is 'cnt'
  2) Using two pointer technique for window of length 'cnt', each time keep track of how many elements in this range are greater than 'k'. Let's say the total count is 'bad'.
  3) Repeat step 2, for every window of length 'cnt' and take minimum of count 'bad' among them. This will be the final answer.

- **Maximize sum of consecutive differences in a circular array** :- Given an array of n elements. Consider array as circular array i.e element after an is a1. The task is to find maximum sum of the difference between consecutive elements with rearrangement of array element allowed i.e after rearrangement of element find |a1 – a2| + |a2 – a3| + …… + |an – 1 – an| + |an – a1|.

  The idea is to use Greedy Approach and try to bring elements having greater difference closer.
  Consider the sorted permutation of the given array a1, a1, a2,…., an – 1, an such that a1 < a2 < a3…. < an – 1 < an.
  Now, to obtain the answer having maximum sum of difference between consecutive element, arrange element in following manner:
  a1, an, a2, an-1,…., an/2, a(n/2) + 1
  We can observe that the arrangement produces the optimal answer, as all a1, a2, a3,….., a(n/2)-1, an/2 are subtracted twice while a(n/2)+1, a(n/2)+2, a(n/2)+3,….., an – 1, an are added twice.

- **Find Index of 0 to be replaced with 1 to get longest continuous sequence of 1s in a binary array** :- Given an array of 0s and 1s, find the position of 0 to be replaced with 1 to get longest continuous sequence of 1s. Expected time complexity is O(n) and auxiliary space is O(1).

  Using an Efficient Solution, the problem can solved in O(n) time. The idea is to keep track of three indexes, current index (curr), previous zero index (prev_zero) and previous to previous zero index (prev_prev_zero). Traverse the array, if current element is 0, calculate the difference between curr and prev_prev_zero (This difference minus one is the number of 1s around the prev_zero). If the difference between curr and prev_prev_zero is more than maximum so far, then update the maximum. Finally return index of the prev_zero with maximum difference.

```
int maxOnesIndex(bool arr[], int n)
{
    int max_count = 0;  // for maximum number of 1 around a zero
    int max_index;  // for storing result
    int prev_zero = -1;  // index of previous zero
    int prev_prev_zero = -1; // index of previous to previous zero

    for (int curr=0; curr<n; ++curr)
    {
        if (arr[curr] == 0)
        {
            if (curr - prev_prev_zero > max_count)
            {
                max_count = curr - prev_prev_zero;
                max_index = prev_zero;
            }
            prev_prev_zero = prev_zero;
            prev_zero = curr;
        }
    }
    // Check for the last encountered zero
    if (n-prev_prev_zero > max_count)
       max_index = prev_zero;

    return max_index;
}
```

- **Three way partitioning of an array around a given range** :- Given an array and a range [lowVal, highVal], partition the array around the range such that array is divided in three parts.

```
void threeWayPartition(int arr[], int n, int lowVal, int highVal)
{
    int start = 0, end = n-1;
    for (int i=0; i<=end;)
    {
        if (arr[i] < lowVal)
            swap(arr[i++], arr[start++]);
        else if (arr[i] > highVal)
            swap(arr[i], arr[end--]);

        else
            i++;
    }
}
```

- **Maximum Product Subarray** :- Given an array that contains both positive and negative integers, find the product of the maximum product subarray. Expected Time complexity is O(n) and only O(1) extra space can be used.

```
Input: arr[] = {6, -3, -10, 0, 2}
Output:   180  // The subarray is {6, -3, -10}

int maxSubarrayProduct(int arr[], int n)
{
    int max_ending_here = 1;
    int min_ending_here = 1;
    int max_so_far = 1;
    int flag = 0;
    /* Traverse through the array. Following values are
    maintained after the i'th iteration:
    max_ending_here is always 1 or some positive product
                    ending with arr[i]
    min_ending_here is always 1 or some negative product
                    ending with arr[i] */
    for (int i = 0; i < n; i++) {
        /* If this element is positive, update max_ending_here.
        Update min_ending_here only if min_ending_here is
        negative */
        if (arr[i] > 0) {
            max_ending_here = max_ending_here * arr[i];
            min_ending_here = min(min_ending_here * arr[i], 1);
            flag = 1;
        }

        /* If this element is 0, then the maximum product
        cannot end here, make both max_ending_here and
        min_ending_here 0
        Assumption: Output is always greater than or equal
                    to 1. */
        else if (arr[i] == 0) {
            max_ending_here = 1;
            min_ending_here = 1;
        }

        /* If element is negative. This is tricky
        max_ending_here can either be 1 or positive.
        min_ending_here can either be 1 or negative.
        next min_ending_here will always be prev.
        max_ending_here * arr[i] next max_ending_here
        will be 1 if prev min_ending_here is 1, otherwise
        next max_ending_here will be prev min_ending_here *
        arr[i] */
        else {
            int temp = max_ending_here;
            max_ending_here = max(min_ending_here * arr[i], 1);
            min_ending_here = temp * arr[i];
        }
        if (max_so_far < max_ending_here)
            max_so_far = max_ending_here;
    }
    if (flag == 0 && max_so_far == 1)
        return 0;
    return max_so_far;
}
```

- **Sorted merge in one array** :- Given two sorted arrays, A and B, where A has a large enough buffer at the end to hold B. Merge B into A in sorted order.

One way is to merge the two arrays by inserting the smaller elements to front of A, but the issue with this approach is that we have to shift every element to right after every insertion.

So, instead from comparing which one is smaller element, we can compare which one is larger and then inserting that element to end of A.

```
void sortedMerge(int a[], int b[], int n, int m) {
  int i = n - 1;
  int j = m - 1;
  int lastIndex = n + m - 1;
  while (j >= 0) {
    if (i >= 0 && a[i] > b[j]) {
      a[lastIndex] = a[i]; // Copy Element
      i--;
    } else {
      a[lastIndex] = b[j]; // Copy Element
      j--;
    }
    lastIndex--; // Move indices
  }
}
```

- **k smallest elements in same order using O(1) extra space :-** You are given an array of n-elements you have to find k smallest elements from the array but they must be in the same order as they are in given array and we are allowed to use only O(1) extra space.

```
void printSmall(int arr[], int n, int k)
{
    for (int i = k; i < n; ++i)
    {
        int max_var = arr[k-1];
        int pos = k-1;
        for (int j=k-2; j>=0; j--)
        {
            if (arr[j] > max_var)
            {
                max_var = arr[j];
                pos = j;
            }
        }
        if (max_var > arr[i])
        {
            int j = pos;
            while (j < k-1)
            {
                arr[j] = arr[j+1];
                j++;
            }
            arr[k-1] = arr[i];
        }
    }
    for (int i=0; i<k; i++)
        cout << arr[i] <<" ";
}
```

- **Shuffle array {a1, a2, .. an, b1, b2, .. bn} as {a1, b1, a2, b2, a3, b3, ……, an, bn} without using extra space** :- Given an array of 2n elements in the following format { a1, a2, a3, a4, ….., an, b1, b2, b3, b4, …., bn }. The task is shuffle the array to {a1, b1, a2, b2, a3, b3, ……, an, bn } without using extra space.
  - Divide and conquer O(NlogN) :-
    The idea is to use Divide and Conquer Technique. Divide the given array into half (say arr1[] and arr2[]) and swap second half element of arr1[] with first half element of arr2[]. Recursively do this for arr1 and arr2.

    Let us explain with the help of an example.

    1) Let the array be a1, a2, a3, a4, b1, b2, b3, b4
    2) Split the array into two halves: a1, a2, a3, a4 : b1, b2, b3, b4
    3) Exchange element around the center: exchange a3, a4 with b1, b2 correspondingly.
       you get: a1, a2, b1, b2, a3, a4, b3, b4
    4) Recursively spilt a1, a2, b1, b2 into a1, a2 : b1, b2
       then split a3, a4, b3, b4 into a3, a4 : b3, b4.
    5) Exchange elements around the center for each subarray we get:
       a1, b1, a2, b2 and a3, b3, a4, b4.

    Note: This solution only handles the case when n = 2i where i = 0, 1, 2, …etc.

    ```
    void shufleArray(int a[], int f, int l)
    {
        if (l - f == 1)
            return;
        int mid = (f + l) / 2;
        int temp = mid + 1;
        int mmid = (f + mid) / 2;
        for (int i = mmid + 1; i <= mid; i++)
            swap(a[i], a[temp++]);
        shufleArray(a, f, mid);
        shufleArray(a, mid + 1, l);
    }
    ```

  - Store 2 elements at an index O(n)

- **Longest subsequence whose average is less than K** :- Given an array of N positive integers and Q queries consisting of an integer K, the task is to print the length of the longest subsequence whose average is less than K.

  An efficient approach is to sort the array elements and find the average of elements starting from the left. Insert the average of elements computed from the left into the container(vector or arrays). Sort the container's element and then use binary search to search for the number K in the container. The length of the longest subsequence will thus be the index number which upper_bound() returns for every query.

- **Rearrange an array in maximum minimum form** :- Given a sorted array of positive integers, rearrange the array alternately i.e first element should be maximum value, second minimum value, third second max, fourth second min and so on.

    - Using auxiliary array and sort it and arrange the elements.
    - Storing 2 numbers at an index.

- **Rearrange array in alternating positive & negative items with O(1) extra space** :- Given an array of positive and negative numbers, arrange them in an alternate fashion such that every positive number is followed by negative and vice-versa maintaining the order of appearance.
  Number of positive and negative numbers need not be equal. If there are more positive numbers they appear at the end of the array. If there are more negative numbers, they too appear in the end of the array.

```
void rearrange(int arr[], int n)
{
    int outofplace = -1;
    for (int index = 0; index < n; index ++)
    {
        if (outofplace >= 0)
        {
            // find the item which must be moved into the out-of-place
            // entry if out-of-place entry is positive and current
            // entry is negative OR if out-of-place entry is negative
            // and current entry is negative then right rotate
            //
            // [...-3, -4, -5, 6...] -->   [...6, -3, -4, -5...]
            //      ^                          ^
            //      |                          |
            //     outofplace      -->      outofplace
            //
            if (((arr[index] >= 0) && (arr[outofplace] < 0))
                || ((arr[index] < 0) && (arr[outofplace] >= 0)))
            {
                rightrotate(arr, n, outofplace, index);

                // the new out-of-place entry is now 2 steps ahead
                if (index - outofplace > 2)
                    outofplace = outofplace + 2;
                else
                    outofplace = -1;
            }
        }
        if (outofplace == -1)
        {
            if (((arr[index] >= 0) && (!(index & 0x01)))
                || ((arr[index] < 0) && (index & 0x01)))
            {
                outofplace = index;
            }
        }
    }
}
```

  OR

  The idea is to process the array and shift all negative values to the end in O(n) time. After all negative values are shifted to the end, we can easily rearrange array in alternating positive & negative items. We basically swap next positive element at even position from next negative element in this step.

- **Rotate a matrix by 90 degree without using any extra space** :-

    -
      ```
      void rotateMatrix(int mat[][N])
      {
          for (int x = 0; x < N / 2; x++)
          {
              for (int y = x; y < N-x-1; y++)
              {
                  int temp = mat[x][y];
                  mat[x][y] = mat[y][N-1-x];
                  mat[y][N-1-x] = mat[N-1-x][N-1-y];
                  mat[N-1-x][N-1-y] = mat[N-1-y][x];
                  mat[N-1-y][x] = temp;
              }
          }
      }
      ```

    - There are two steps :

      Find transpose of matrix.
      Reverse columns of the transpose.

      Illustration of above steps :

      Let the given matrix be
      ```
      1  2  3  4
      5  6  7  8
      9  10 11 12
      13 14 15 16
      ```

      First we find transpose.
      ```
      1 5 9 13
      2 6 10 14
      3 7 11 15
      4 8 12 16
      ```

      Then we reverse elements of every column.
      ```
      4 8 12 16
      3 7 11 15
      2 6 10 14
      1 5  9 13
      ```

- **Sum of XOR of all subarrays** :-
  A better solution will be using a prefix array i.e. for every index 'i' of the array 'arr[]', create a prefix array to store the XOR of all the elements from left end of the array 'arr[]' up to the ith element of 'arr[]'. Creating a prefix array will take a time of O(N).
  Now, using this prefix array, we can find the XOR value of any sub-array in O(1) time.

  We can find the XOR from index l to r using the formula:

```
if l is not zero
    XOR = prefix[r] ^ prefix[l-1]
else
    XOR = prefix[r].
```
After this, all we have to do is, to sum up the XOR values of all the sub-arrays.

Since, total number of sub-arrays are of the order (N2), the time-complexity of this approach will be O(N2).

Best solution : For the sake of better understanding, let's assume any bit of an element is represented by the variable 'i' and the variable 'sum' is used to store the final sum.

The idea here is, we will try to find the number of XOR values with ith bit set. Let us suppose, there are 'Si' number of sub-arrays with ith bit set. For, ith bit, sum can be updated as sum += (2i * S) .

So, the question is how to implement the above idea?

We will break the task to multiple steps. At each step, we will try to find the number of XOR values with ith bit set.
Now, we will break each step to sub-steps. In each sub-step, we will try to find the number of sub-arrays staring from an index 'j'(where j varies between 0 to n – 1) with ith bit set in there XOR value. For, ith bit is to be set, odd number of elements of the sub-array should have there ith bit set.
For all the bits, in a variable c_odd, we will store the count of the number of sub-arrays starting from j = 0 with ith bit set in odd number of elements.
Then, we will iterate through all the elements of the array updating the value of c_odd when needed. If we reach an element 'j' with ith bit set, we will update c_odd as c_odd = (n – j – c_odd). Its because, since we encountered a set bit, number of sub-arrays with even number of elements with ith bit set will switch to number of sub-arrays with odd number of elements with ith bit set.

```
int findXorSum(int arr[], int n)
{
    int sum = 0;
    int mul = 1;

    for (int i = 0; i < 30; i++) {

        int c_odd = 0;
        bool odd = 0;

        for (int j = 0; j < n; j++) {
            if ((arr[j] & (1 << i)) > 0)
                odd = (!odd);
            if (odd)
                c_odd++;
        }

        for (int j = 0; j < n; j++) {
            sum += (mul * c_odd);
            if ((arr[j] & (1 << i)) > 0)
                c_odd = (n - j - c_odd);
        }

        mul *= 2;
    }
    return sum;
}
```

- **Search element in a sorted matrix** :- Given a sorted matrix mat[n][m] and an element 'x'. Find position of x in the matrix if it is present, else print -1. Matrix is sorted in a way such that all elements in a row are sorted in increasing order and for row 'i', where 1 <= i <= n-1, first element of row 'i' is greater than or equal to the last element of row 'i-1'. The approach should have O(log n + log m) time complexity.

  ```
  1) Perform binary search on the middle column
     till only two elements are left or till the
     middle element of some row in the search is
     the required element 'x'. This search is done
     to skip the rows that are not required
  2) The two left elements must be adjacent. Consider
     the rows of two elements and do following
     a) check whether the element 'x' equals to the
        middle element of any one of the 2 rows
     b) otherwise according to the value of the
        element 'x' check whether it is present in
        the 1st half of 1st row, 2nd half of 1st row,
        1st half of 2nd row or 2nd half of 2nd row.

  Note: This approach works for the matrix n x m
        where 2 <= n. The algorithm can be modified
        for matrix 1 x m, we just need to check whether
        2nd row exists or not
  ```

- **Search in a row wise and column wise sorted matrix** :- Given an n x n matrix and a number x, find the position of x in the matrix if it is present in it. Otherwise, print "Not Found". In the given matrix, every row and column is sorted in increasing order. The designed algorithm should have linear time complexity.

  ```
  int search(int mat[4][4], int n, int x)
  {
      if (n == 0)
          return -1;
      int smallest = a[0][0], largest = a[n - 1][n - 1];
      if (x < smallest || x > largest)
          return -1;

      int i = 0, j = n - 1;
      while (i < n && j >= 0) {
          if (mat[i][j] == x) {
              cout << "n Found at "
                   << i << ", " << j;
              return 1;
  ```

```
        }
        if (mat[i][j] > x)
            j--;
        else // if mat[i][j] < x
            i++;
    }
    cout << "n Element not found";
    return 0; // if ( i==n || j== -1 )
}
```

- **Find whether a subarray is in form of a mountain or not :-** We are given an array of integers and a range, we need to find whether the subarray which falls in this range has values in form of a mountain or not. All values of the subarray are said to be in form of a mountain if either all values are increasing or decreasing or first increasing and then decreasing.
  We can solve this problem by first some preprocessing then we can answer for each subarray in the constant amount of time. We maintain two arrays left and right where left[i] stores the last index on left side which is increasing i.e. greater than its previous element and right[i] will store the first index on the right side which is decreasing i.e. greater than its next element. Once we maintained these arrays we can answer each subarray in constant time. Suppose range [l, r] is given then only if right[l] >= left[r], the subarray will be in form of a mountain otherwise not because the first index in decreasing form (i.e. right[l]) should come later than last index in increasing form (i.e. left[r]).

```
int preprocess(int arr[], int N, int left[], int right[])
{
    left[0] = 0;
    int lastIncr = 0;
    for (int i = 1; i < N; i++)
    {
        if (arr[i] > arr[i - 1])
            lastIncr = i;
        left[i] = lastIncr;
    }
    right[N - 1] = N - 1;
    int firstDecr = N - 1;
    for (int i = N - 2; i >= 0; i--)
    {
        if (arr[i] > arr[i + 1])
            firstDecr = i;
        right[i] = firstDecr;
    }
}
bool isSubarrayMountainForm(int arr[], int left[],int right[], int L, int R)
{
    return (right[L] >= left[R]);
}
```