```
FIND TRAILING ZEROES

public class trailingZeroes {
  public static void main(String[] args) {
    // 5! -> 120  => 1 trailing zero
    int n = 10;
    int ans = 0;
    for (int i = 5; i <= n; i=i*5) {
      ans += n/i;
    }
    System.out.println(ans);
  }
}
```

```
Find max. difference of arr[j]-arr[i], s.t j > i
e.g: arr = [2,3,10,6,4,8,1]
     o/p: 8

public class MaxDifference {
  //Naive Approach: O(n^2)
  static void findMaxDifference(int arr[]) {
    int n = arr.length;
    int max_diff = Integer.MIN_VALUE;
    for (int i = 0; i < n-1; i++) {
      for (int j = i+1; j < n; j++) {
        int diff = arr[j]-arr[i];
        max_diff = Math.max(max_diff,diff);
      }
    }
    System.out.println(max_diff);
  }

  // Optimised: O(n) approach
  // max. diff can be obtained if arr[j] is maximum and arr[i] is
minimum possible
  // so maintain the min. element found till ith index
  static void findMaxDifference_2(int arr[]) {
    int min_element = arr[0];
```

```
    int diff = 0;
    for (int i = 1; i < arr.length; i++) {
      diff = Math.max(diff,arr[i]-min_element);
      min_element = Math.min(min_element,arr[i]);
    }
    System.out.println(diff);
  }

  public static void main(String[] args) {
    // int arr[] = {2,3,10,6,4,8,1};
    int arr[] = {30,10,8,2};
    findMaxDifference(arr);
    findMaxDifference_2(arr);
  }
}
```

```
Question: given an array. find freq of all the elements in the
array
e.g: arr = [10,10,10,25,30,30]
      10 - 3
      25 - 1
      30 - 2

import java.util.*;
public class FrequencySortedArr {

  private static void findFreq(int arr[]) {
    for (int i = 0; i < arr.length; i++) {
      int count = 0;
      for (int j = i; j < arr.length; j++) {
        if (arr[i] != -1 && arr[i] == arr[j]) {
          count++;
          if (i != j)
            arr[j] = -1;
        }
      }
      if (arr[i] != -1)
```

```
        System.out.println(arr[i] + "-> " + count);
    }
  }

  //using hash map in O(n)
  private static void findFreq2(int arr[]) {
    HashMap<Integer,Integer> m = new HashMap<>();
    for (int i = 0; i < arr.length; i++) {
      if (!m.containsKey(arr[i]))
        m.put(arr[i],1);
      else
        m.put(arr[i], m.get(arr[i])+1);
    }
    //iterate over the map and print the count of the keys
    Set<Integer> keys = m.keySet();
    for (Integer k : keys) {
      System.out.println(k + " " + m.get(k));
    }
  }

  public static void main(String[] args) {
    int arr[] = {10,10,10,25,30,30};
    // int arr[] = {10,10,10,10};
    // findFreq(arr);
    findFreq2(arr);
  }
}
```

```
Find Leaders in an array
leaders: if for a particular element, all the elements on its
right are smaller than it
    e.g: arr=[7,10,4,3,6,5,2]
    o/p: 10,6,5,2

public class FindLeaders
{

    //Approach1: O(n^2)
    static void findLeaders(int arr[]) {
        int n = arr.length;
        for (int i = 0; i < n; i++) {
            boolean isLeader = true;
            for (int j = i+1; j < n; j++) {
                if (arr[j] > arr[i]) {
                    isLeader = false;
                    break;
                }
            }
            if (isLeader)
                System.out.print(arr[i] + " ");
        }
    }

    //Approach2: O(n) -> The idea is to start traversing from the
right
    // but the problem is it will print the leader from the last
    // so, we can use an array to store the elements and then
reverse it to display the ans
    // space : O(n) and time = O(n) for reversing

    static void findLeaders2(int arr[]) {
        // the last element is always a leader, so print it
        int n = arr.length;
        int curr_leader = arr[n-1];
        System.out.print(curr_leader + " ");
        for (int i = n-2; i >= 0; i--) {
```

```
            if (arr[i] > curr_leader) {
                curr_leader = arr[i];
                System.out.print(curr_leader + " ");
            }
        }
    }

    public static void main(String[] args) {
        int arr[] = {7,10,4,3,6,5,2};
        findLeaders(arr);
        System.out.println();
        findLeaders2(arr);
    }
}
```

```
Question: given an array representing the stocks of n days in
advance
we can buy and sell the stock on any day. maximize the profit
arr = [1,5,3,8,12]
o/p = 13 (5-1 + 12-3)

public class StockBuySell {

  // take the bottom and the peak value and add it to the total
profit
  // while the stock value is increasing i.e if it greater than
the prev. day stock price, keep it adding to the profit and
return it
  static int solve(int arr[]) {
    int profit = 0;
    for (int i = 1; i < arr.length; i++) {
      if (arr[i] > arr[i-1])
        profit += arr[i] - arr[i-1];
    }
    return profit;
  }

  public static void main(String[] args) {
    int arr[] = {1,5,3,8,12};
```

```
    System.out.println(solve(arr));
  }
}
```

```
Trapping Rain Water problem
  arr of non-negative elements
  bars of different heights are given
  how much units of water can be collected b/w the bars

  e.g: arr = [3,0,1,2,5]
      o/p : 6
    arr = [1,2,3]
      o/p: 0
    arr = [3,2,1]
      o/p: 0

public class RainWaterTrap {
  /* Naive Approach
    first thing to notice is that no water can be trapped for the
1st and last bar as there is no bar for support on the left and
right respectively
    find left max bar -> max left bar so that we can store max
water
    find rMax -> right max bar
  */

  static int findWaterTrapped(int arr[]) {
    int ans = 0;
    for (int i = 1; i < arr.length-1; i++) {
      int lMax = arr[i];
      for (int j = 0; j < i; j++)
        lMax = Math.max(lMax,arr[j]);

      int rMax = arr[i];
      for (int j = i+1; j < arr.length; j++)
        rMax = Math.max(rMax,arr[j]);

      ans += Math.min(rMax,lMax) - arr[i];
```

```
    }
    return ans;
  }

  //Optimized O(n): preCompute lMax and rMax so that we don't
have to calculate it for every index
  static int findWaterTrapped2(int arr[]) {
    int n = arr.length;
    int lMax[] = new int[n];
    int rMax[] = new int[n];
    lMax[0] = arr[0];
    for (int i = 1; i < n; i++) {
      lMax[i] = Math.max(arr[i],lMax[i-1]);
    }

    rMax[n-1] = arr[n-1];
    for (int i = n-2; i >= 0; i--) {
      rMax[i] = Math.max(arr[i],rMax[i+1]);
    }
    int ans = 0;
    for (int i = 1; i < n-1; i++) {
      ans += Math.min(lMax[i],rMax[i]) - arr[i];
    }
    return ans;
  }

  public static void main(String[] args) {
    int arr[] = {3,0,1,2,5};
    System.out.println(findWaterTrapped2(arr));
  }
}
```

```
Question: Given a binary array i.e containing 0 and 1
    find max no. of consecutive 1's
    e.g:
        arr = [1,0,1,1,1,1,0,1,1]
            o/p: 4
         arr = [1,1,1,1]
            o/p: 4

public class MaxConsecutive1s {
  /*naive approach: O(n^2)
    traverse through every element, and for every element we
count how many consecutive 1's are there
    we take a ans variable which we update to get the max
consective 1's
  */
  static int countMax1(int arr[]) {
    int ans = 0;
    for (int i = 0; i < arr.length; i++) {
      int count = 0;
      for (int j = i; j < arr.length; j++) {
        if (arr[j] == 1)
          count++;
        else
          break;
      }
      ans = Math.max(ans,count);
    }
    return ans;
  }

  /*optimized: O(n)
    traverse from left to right, whenever we encounter a 0, reset
the current count
    otherwise increment the count and keep on updating the max
consective 1's find till now
  */
  static int countMax2(int arr[]) {
    int ans = 0, curr = 0;
```

```
    for (int i = 0; i < arr.length; i++) {
      if (arr[i] == 0)
        curr = 0;
      else
        curr++;

      ans = Math.max(ans,curr);
    }
    return ans;
  }

  public static void main(String[] args) {
    int arr[] = {0,1,1,0,1,1,1};
    System.out.println(countMax2(arr));
  }
}
```

```
Question: Given an array. find max sum of a subarray
  subarray: contiguous elements picked from an array
  e.g:
    arr = [2,3,-8,7,-1,2,3]
      o/p: 11
    arr = [5,8,3]
      o/p: 16

public class MaxSumSubarray {
  /* Naive: O(n2)
    try all the possible subarrays and find sum of them. keep on
updating the maximum sum find till now
  */
  static int findMaxSubarray1(int arr[]) {
    int ans = Integer.MIN_VALUE;
    for (int i = 0; i < arr.length; i++) {
      int temp = 0;
      for (int j = i; j < arr.length; j++) {
        temp += arr[j];
        ans = Math.max(temp,ans);
      }
```

```
    }
    return ans;
  }

  // the idea is to extend the prev arr or start a new subarray
  // we already have the maximum sum for prev. elements
  //Kadane's algorithm: find's max sum of subarray
  static int findMaxSubarray2(int arr[]) {
    int ans = arr[0];
    int max_ending_here = arr[0];
    for (int i = 1; i < arr.length; i++) {
      max_ending_here = Math.max(max_ending_here + arr[i],
arr[i]);
      ans = Math.max(max_ending_here,ans);
    }
    return ans;
  }


  public static void main(String[] args) {
    int arr[] = {2,3,-8,7,-1,2,3};
    // int arr[] = {-6,-1,-8};
    System.out.println(findMaxSubarray2(arr));
  }
}
```

```
Question: Find max. length even-odd subarray
  e.g:
    arr = [10,12,14,7,8]
      o/p: 3

    arr = [7,10,13,14]
      o/p: 4

public class EvenOddLength {
  static int findMaxEvenOddLength(int arr[]) {
    int res = 1;
    for (int i = 0; i < arr.length-1; i++) {
      int curr = 1;
```

```
      for (int j = i+1; j < arr.length; j++) {
        if (arr[j] % 2 == 0 && arr[j-1] % 2 != 0 || arr[j] % 2 !=
0 && arr[j-1] % 2 == 0)
          curr++;
        else
          break; //bcoz subarray elements need to be contiguous
      }
      res = Math.max(res,curr);
    }
    return res;
  }

  // use kadane's algorithm: O(n)
  static int findMaxEvenOddLength2(int arr[]) {
    int res = 1;
    int curr = 1;
    for (int i = 1; i < arr.length; i++) {
      if (arr[i] % 2 == 0 && arr[i-1] % 2 != 0 || arr[i] % 2 != 0
&& arr[i-1] % 2 == 0)
        curr++;

      // if the array till i is not alternating(even-odd), start
a new subarray with i
      else
        curr = 1;

      res = Math.max(res,curr); //update the max. subarray of
alternating odd-even length
    }
    return res;
  }

  public static void main(String[] args) {
    int arr[] = {7,10,13,14};
    // int arr[] = {10,12,14,7,8};
    System.out.println(findMaxEvenOddLength2(arr));
  }
}
```

```
Find Majority element
  majority element: an element which appears more than n/2 times
(where n is size of array)
  e.g:
    arr = [8,3,4,8,8]
      o/p: 0 or 3 or 4 => return the index of majority element
      else return -1 if it doesn't exist

public class MajorityElement {
  //Naive Approach: O(n^2)
  static int findMajorityElement(int arr[]) {
    int n = arr.length;
    for (int i = 0; i < n-1; i++) {
      int count = 1;
      for (int j = i+1; j < n; j++) {
        if (arr[j] == arr[i])
          count++;
      }
      if (count > n/2)
        return i;
    }
    return -1;
  }

  //Moore's Voting algorithm: O(n)
  static int findMajorityElement2(int arr[]) {
    int n = arr.length;
    //phase1: find a candidate
    int count = 1, res = 0;
    for (int i = 1; i < n; i++) {
      if (arr[i] == arr[res])
        count++;
      else
        count--;

      if (count == 0) {
        res = i;
```

```
        count = 1;
      }
    }

    //phase2: check if candidate is actually a majority
    count = 0;
    for (int i = 0; i < n; i++) {
      if (arr[i] == arr[res])
        count++;
    }
    if (count <= n/2)
      return -1;
    else
      return res;
  }

  public static void main(String[] args) {
    // int arr[] = {8,3,4,8,8};
    int arr[] = {8,8,6,6,4,6};
    System.out.println(findMajorityElement2(arr));
  }
}
```

```
Question: Find max sum of K consecutive elements
    e.g: arr = [1,8,30,-5,20,7]
         k = 3
          o/p : 45

public class WindowSliding {
  //naive approach: O(n*k)
  static int maxConsecutiveSum(int arr[], int k) {
    int res = Integer.MIN_VALUE;
    for (int i = 0; i <= arr.length-k; i++) {
      int curr_sum = 0;
      for (int j = 0; j < k; j++) {
        curr_sum += arr[i+j];
      }
```

```
      res = Math.max(curr_sum,res);
    }
    return res;
  }

  /*use window sliding technique: O(n)
    compute the sum of first k elements
    now, to calculate sum of next k elements, we will use the
already computed sum of first k elements
    how? we will add the next element and then delete the element
of the previous window or subarray
  */
  static int maxConsecutiveSum2(int arr[], int k) {
    int curr_sum = 0;
    for (int i = 0; i < k; i++)
      curr_sum += arr[i];

    int max_sum = curr_sum;
    for (int i = k; i < arr.length; i++) {
      curr_sum += arr[i] - arr[i-k];
      max_sum = Math.max(max_sum, curr_sum);
    }
    return max_sum;
  }

  public static void main(String[] args) {
    int arr[] = {1,8,30,-5,20,7};
    int k = 3;
    System.out.println(maxConsecutiveSum2(arr, k));
  }
}
```

```
Question: Given array of non-negative no. find whether there
exists a subarray with the given sum or not
e.g: arr = [1,4,20,3,10,5]
     sum = 33
      o/p: Yes, [20,3,10]

public class SubarrayWithGivenSum {
  //naive approach: try all the possible subarrays and keep
checking if sum == given_sum or not
  static boolean findSubarraySum(int arr[], int sum) {
    for (int i = 0; i < arr.length; i++) {
      int curr_sum = 0;
      for (int j = i; j < arr.length; j++) {
        curr_sum += arr[j];
        if (curr_sum == sum)
          return true;
        else if (curr_sum > sum)
          break;
      }
    }
    return false;
  }

  //window sliding technique with window of variable size: O(n)
  static boolean findSubarraySum2(int arr[], int sum) {
    //initial window size is 0
    int s = 0;
    int curr_sum = 0;
    for (int e = 0; e < arr.length; e++) {
      curr_sum += arr[e];
      // while curr_sum comes out to be greater than given sum,
then there is no need of inc. the subarray and taking it's sum as
it will never be equal to given sum. so, we will consider a new
subarray/window and hence increment the variable s.
      while (curr_sum > sum) {
        curr_sum -= arr[s];
        s++;
```

```
      }
      if (curr_sum == sum)
        return true;
    }
    return false;
  }

  public static void main(String[] args) {
    int arr[] = {1,4,20,3,10,5};
    System.out.println(findSubarraySum(arr,33));
  }
}
```

```
Question: given an array and q queries. for each query, you are
given start index and end index.
find the the subarray sum from start to end
e.g: arr = [2,8,3,9,6,5,4]
      q = 3
        l = 0, r = 2
          o/p: sum = 13
        l = 1, r = 3
          o/p: 20


/*
Naive approach:
  for each and every query, run a loop from start to end index
and calculate the sum
  it will take, O(q * n) time
*/

public class PrefixSum {

  // Use prefix sum : precompute the sum of array
  static int computePrefixSum(int arr[], int l, int r) {
    //create a new array and initilize it
    int A[] = new int[arr.length];
    A[0] = arr[0];
```

```
    for (int i = 1; i < arr.length; i++) {
      A[i] = A[i-1] + arr[i];
    }

    if (l == 0)
      return A[r];
    return A[r] - A[l-1];
  }

  public static void main(String[] args) {
    int arr[] = {2,8,3,9,6,5,4};
    System.out.println(computePrefixSum(arr,1,3));
  }
}
```

```
Question: given an array. return the index of any equilibrium
point that exists
  equilbrium point: if sum of elements before it and after it are
same, it is equilbrium point
  e.g: arr = [3,4,8,-9,20,6]
       o/p: true (point is 20)

public class EquilibriumPoint {
  //use precomputation technique
  static int checkEquiPoint(int arr[]) {
    int n = arr.length;
    int sum = 0;
    for (int i = 0; i < n; i++)
      sum += arr[i];

    int leftSideSum = 0;
    int rightSideSum = sum;

    for (int i = 0; i < n; i++) {
      rightSideSum -= arr[i];
      if (leftSideSum == rightSideSum)
        return i;
      leftSideSum += arr[i];
```

```
    }
    return -1; //if no equilibrium point found
  }

  public static void main(String[] args) {
    int arr[] = {3,4,8,-9,20,6};
    System.out.println(checkEquiPoint(arr));
  }
}
```