DSA Problems:

(15-11-2024)

1)Remove Duplicates Sorted array :

Given a sorted array arr. Return the size of the modified array which contains only distinct elements.

Note:

1. Don't use set or HashMap to solve the problem.

2. You must return the modified array size only where distinct elements are present and modify the original array such that all the distinct elements come at the beginning of the original array.

Examples :

Input: arr = [2, 2, 2, 2, 2]
Output: [2]
Explanation: After removing all the duplicates only one instance of 2 will remain i.e. [2] so modified array will contains 2 at first position and you should return 1 after modifying the array, the driver code will print the modified array elements.

Input: arr = [1, 2, 4]
Output: [1, 2, 4]

Explation: As the array does not contain any duplicates so you should return 3.

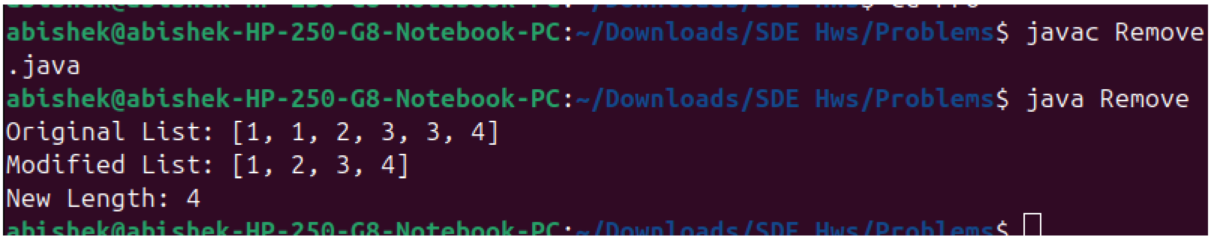
Program :

```
import java.util.List;

import java.util.ArrayList;


class Solution {
```

```
public int remove_duplicate(List<Integer> arr) {
    if (arr.size() == 0) {
        return 0;
    }
    int j = 1;
    for (int i = 1; i < arr.size(); i++) {
        if (!arr.get(i).equals(arr.get(i - 1))) {
            arr.set(j, arr.get(i));
            j++;
        }
    }
    return j;
}
public static void main(String[] args) {
    Solution solution = new Solution();
    List<Integer> arr = new ArrayList<>();
    arr.add(1);
    arr.add(1);
    arr.add(2);
    arr.add(3);
    arr.add(3);
    arr.add(4);
    System.out.println("Original List: " + arr);

    int newLength = solution.remove_duplicate(arr);
```

```
         System.out.println("Modified List: " + arr.subList(0, newLength));
        System.out.println("New Length: " + newLength);
    }
}
```

Output :



```
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$ javac Remove
.java
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$ java Remove
Original List: [1, 1, 2, 3, 3, 4]
Modified List: [1, 2, 3, 4]
New Length: 4
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$
```

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

2) Stock Buy and Sell:

The cost of stock on each day is given in an array A[] of size N. Find all the segments of days on which you buy and sell the stock such that the sum of difference between sell and buy prices is maximized. Each segment consists of indexes of two elements, first is index of day on which you buy stock and second is index of day on which you sell stock.

Note: Since there can be multiple solutions, the driver code will print 1 if your answer is correct, otherwise, it will return 0. In case there's no profit the driver code will print the string "No Profit" for a correct solution.

Example 1:
Input:
N = 7
A[] = {100,180,260,310,40,535,695}
Output:
1
Explanation:
One possible solution is (0 3) (4 6)
We can buy stock on day 0,

and sell it on 3rd day, which will
give us maximum profit. Now, we buy
stock on day 4 and sell it on day 6.

Example 2:

Input:
N = 5
A[] = {4,2,2,2,4}
Output:
1
Explanation:
There are multiple possible solutions.
one of them is (3 4)
We can buy stock on day 3,
and sell it on 4th day, which will
give us maximum profit.

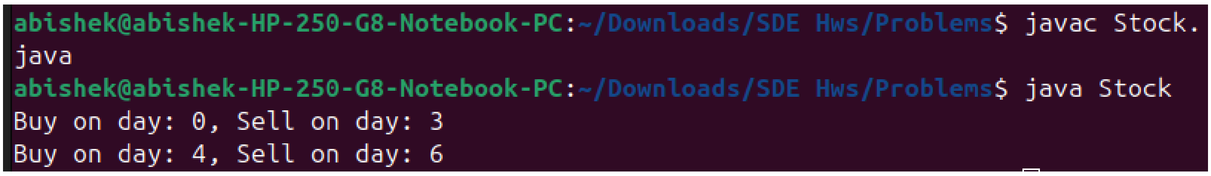
Your Task:

The task is to complete the function stockBuySell() which takes an array of A[] and N as input parameters and finds the days of buying and selling stock. The function must return a 2D list of integers containing all the buy-sell pairs i.e. the first value of the pair will represent the day on which you buy the stock and the second value represent the day on which you sell that stock. If there is No Profit, return an empty list.

Program :

```
import java.util.ArrayList;

class Stock {

   ArrayList<ArrayList<Integer>> stockBuySell(int A[], int n) {
        ArrayList<ArrayList<Integer>> res = new ArrayList<>();
        int i = 0;
        while (i < n - 1) {
            while (i < n - 1 && A[i + 1] <= A[i]) {
                i++;
            }
            if (i == n - 1) break;
            int buy = i++;
            while (i < n && A[i] >= A[i - 1]) {
                i++;
            }
            int sell = i - 1;
            ArrayList<Integer> transaction = new ArrayList<>();
            transaction.add(buy);
            transaction.add(sell);
            res.add(transaction);
        }
        return res;
    }
    public static void main(String[] args) {
        Stock Stock = new Stock();
        int[] stockPrices = {100, 180, 260, 310, 40, 535, 695};
        int n = stockPrices.length;

        ArrayList<ArrayList<Integer>> result = Stock.stockBuySell(stockPrices,
n);

        if (result.size() == 0) {

            System.out.println("No profit can be made.");

        } else {

            for (ArrayList<Integer> transaction : result) {

                System.out.println("Buy on day: " + transaction.get(0) + ", Sell on
day: " + transaction.get(1));

            }

        }

    }

}
```

Output :



```
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$ javac Stock.
java
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$ java Stock
Buy on day: 0, Sell on day: 3
Buy on day: 4, Sell on day: 6
```

Time Complexity : O(n)

Space Complexity : O(1)

3) Coin Change (Count Ways) :

Given an integer array coins[ ] representing different denominations of currency and an integer sum, find the number of ways you can make sum by using different combinations from coins[ ].

Note: Assume that you have an infinite supply of each type of coin. And you can use any coin as many times as you want.

Answers are guaranteed to fit into a 32-bit integer.

Examples:

Input: coins[] = [1, 2, 3], sum = 4

Output: 4

Explanation: Four Possible ways are: [1, 1, 1, 1], [1, 1, 2], [2, 2], [1, 3].

Input: coins[] = [2, 5, 3, 6], sum = 10

Output: 5

Explanation: Five Possible ways are: [2, 2, 2, 2, 2], [2, 2, 3, 3], [2, 2, 6], [2, 3, 5] and [5, 5].

Input: coins[] = [5, 10], sum = 3

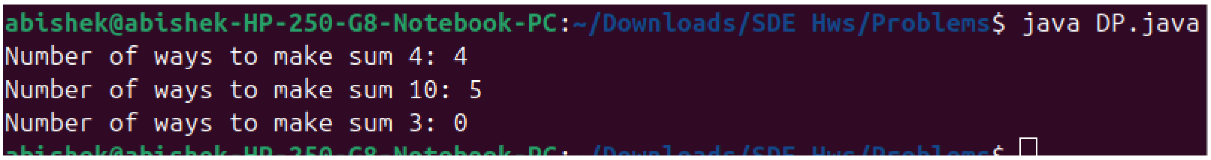
Output: 0

Explanation: Since all coin denominations are greater than sum, no combination can make the target sum.

Program :

```
class DP {
    public int count(int coins[], int sum) {
        int[] dp = new int[sum + 1];
        dp[0] = 1;
        for (int coin : coins) {
            for (int i = coin; i <= sum; i++) {
                dp[i] += dp[i - coin];
            }
        }
        return dp[sum];
    }
    public static void main(String[] args) {
        DP DP = new DP();
        int[] coins1 = {1, 2, 3};
        int sum1 = 4;
        System.out.println("Number of ways to make sum " + sum1 + ": " +
DP.count(coins1, sum1));

        int[] coins2 = {2, 5, 3, 6};
        int sum2 = 10;

        System.out.println("Number of ways to make sum " + sum2 + ": " + DP.count(coins2, sum2));


        int[] coins3 = {5, 10};

        int sum3 = 3;

        System.out.println("Number of ways to make sum " + sum3 + ": " + DP.count(coins3, sum3));

    }

}
```

Output :



```
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$ java DP.java
Number of ways to make sum 4: 4
Number of ways to make sum 10: 5
Number of ways to make sum 3: 0
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$
```

Time Complexity : O(n)

Space Complexity : O(1)

4) First and Last Occurence :

Given a sorted array arr with possibly some duplicates, the task is to find the first and last occurrences of an element x in the given array.

Note: If the number x is not found in the array then return both the indices as -1.

Examples:

Input: arr[] = [1, 3, 5, 5, 5, 5, 67, 123, 125], x = 5

Output: [2, 5]

Explanation: First occurrence of 5 is at index 2 and last occurrence of 5 is at index 5

Input: arr[] = [1, 3, 5, 5, 5, 5, 7, 123, 125], x = 7

Output: [6, 6]

Explanation: First and last occurrence of 7 is at index 6

Input: arr[] = [1, 2, 3], x = 4

Output: [-1, -1]

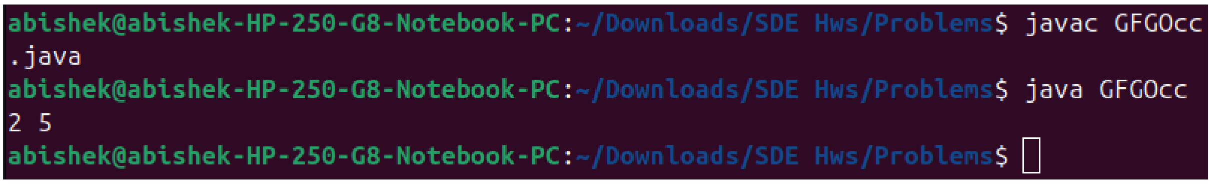
Explanation: No occurrence of 4 in the array, so, output is [-1, -1]

Program :

```
import java.util.ArrayList;

class GFGOcc {
```

```
public ArrayList<Integer> find(int[] arr, int x) {

    ArrayList<Integer> result = new ArrayList<>();

    int first = findFirstOccurrence(arr, x);

    int last = findLastOccurrence(arr, x);

    result.add(first);

    result.add(last);

    return result;

}

private int findFirstOccurrence(int[] arr, int x) {

    int left = 0, right = arr.length - 1;

    int firstOccurrence = -1;

    while (left <= right) {

        int mid = left + (right - left) / 2;

        if (arr[mid] == x) {

            firstOccurrence = mid;

            right = mid - 1; // Move left to find earlier occurrence

        } else if (arr[mid] < x) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return firstOccurrence;
}
private int findLastOccurrence(int[] arr, int x) {
    int left = 0, right = arr.length - 1;
    int lastOccurrence = -1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == x) {
            lastOccurrence = mid;
            left = mid + 1; // Move right to find later occurrence
        } else if (arr[mid] < x) {
            left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return lastOccurrence;
    }
    public static void main(String[] args) {
        GFGOcc ob = new GFGOcc();
        int[] arr = {1, 3, 5, 5, 5, 5, 67, 123, 125};
        int x = 5;
        ArrayList<Integer> ans = ob.find(arr, x);
        System.out.println(ans.get(0) + " " + ans.get(1));
    }
}
```

Output :



```
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$ javac GFGOcc
.java
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$ java GFGOcc
2 5
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$ 
```

Time Complexity : O(logn)

Space Complexity : O1(1)

5)First Repeating Element :

Given an array arr[], find the first repeating element. The element should occur more than once and the index of its first occurrence should be the smallest.

Note:- The position you return should be according to 1-based indexing.

Examples:

Input: arr[] = [1, 5, 3, 4, 3, 5, 6]

Output: 2

Explanation: 5 appears twice and its first appearance is at index 2 which is less than 3 whose first the occurring index is 3.

Input: arr[] = [1, 2, 3, 4]

Output: -1

Explanation: All elements appear only once so answer is -1.

Constraints:

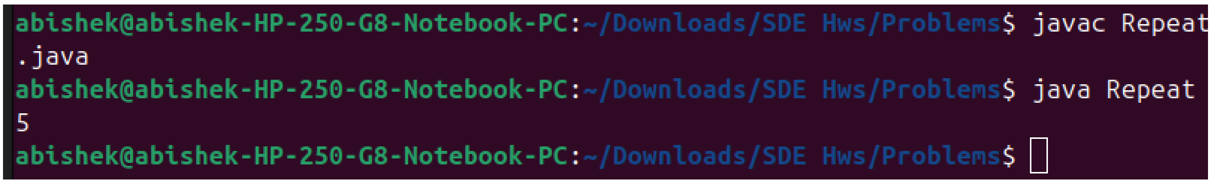
1 <= arr.size <= 106

0 <= arr[i]<= 106

Program :

```
import java.util.HashSet;
class Repeat {
    public int firstRepeating(int arr[]) {
        HashSet<Integer> seen = new HashSet<>();

        for (int i = 0; i < arr.length; i++) {
            if (seen.contains(arr[i])) {
                return i + 1;  // Convert 0-based index to 1-based index
            }
            seen.add(arr[i]);
        }

        return -1;
    }

    public static void main(String[] args) {
        Repeat ob = new Repeat();

        int arr[] = {1, 5, 3, 4, 3, 5, 6};
        int result = ob.firstRepeating(arr);
        System.out.println(result);  // Output should be 2
    }
}
```

Output :



```
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$ javac Repeat
.java
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$ java Repeat
5
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$ 
```

Time Complexity : O(n)

Space Complexity : O(n)

6)Find Transition Point

Given a sorted array, arr[] containing only 0s and 1s, find the transition point, i.e., the first index where 1 was observed, and before that, only 0 was observed. If arr does not have any 1, return -1. If array does not have any 0, return 0.

Examples:

Input: arr[] = [0, 0, 0, 1, 1]

Output: 3

Explanation: index 3 is the transition point where 1 begins.

Input: arr[] = [0, 0, 0, 0]

Output: -1

Explanation: Since, there is no "1", the answer is -1.

Input: arr[] = [1, 1, 1]

Output: 0

Explanation: There are no 0s in the array, so the transition point is 0, indicating that the first index (which contains 1) is also the first position of the array.

Input: arr[] = [0, 1, 1]

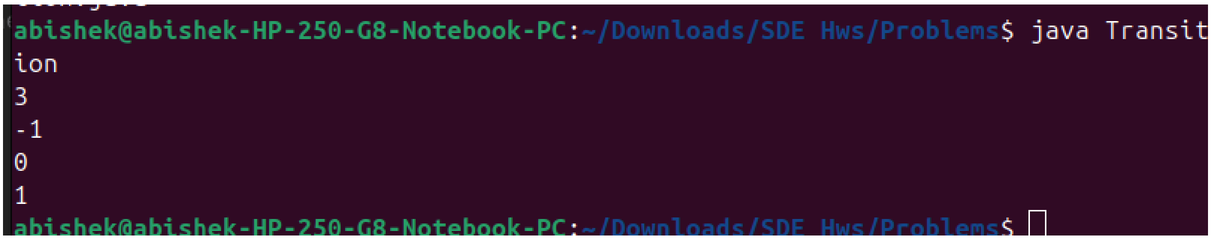
Output: 1

Explanation: Index 1 is the transition point where 1 starts, and before it, only 0 was observed.

Program :

```
class Transition {
    public int findTransitionPoint(int arr[]) {
        int low = 0;
        int high = arr.length - 1;
        if (arr[0] == 1) {
            return 0;
        }
        while (low <= high) {
            int mid = low + (high - low) / 2;

            if (arr[mid] == 1) {
                if (mid == 0 || arr[mid - 1] == 0) {
                    return mid;
                }
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        Transition ob = new Transition();

        int arr1[] = {0, 0, 0, 1, 1};
        System.out.println(ob.findTransitionPoint(arr1));  // Output: 3

        int arr2[] = {0, 0, 0, 0};
        System.out.println(ob.findTransitionPoint(arr2));  // Output: -1

        int arr3[] = {1, 1, 1};
        System.out.println(ob.findTransitionPoint(arr3));  // Output: 0

        int arr4[] = {0, 1, 1};
        System.out.println(ob.findTransitionPoint(arr4));  // Output: 1
    }
}
```

Output :



```
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$ java Transit
ion
3
-1
0
1
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$ 
```

Time Complexity : O(logn)

Space Complexity : O(1)

7)Maximum Index

Given an array arr of positive integers. The task is to return the maximum of j - i subjected to the constraint of arr[i] ≤ arr[j] and i ≤ j.

Examples:

Input: arr[] = [1, 10]

Output: 1

Explanation: arr[0] ≤ arr[1] so (j-i) is 1-0 = 1.

Input: arr[] = [34, 8, 10, 3, 2, 80, 30, 33, 1]

Output: 6

Explanation: In the given array arr[1] < arr[7] satisfying the required condition(arr[i] ≤ arr[j]) thus giving the maximum difference of j - i which is 6(7-1).

Expected Time Complexity: O(n)

Expected Auxiliary Space: O(n)

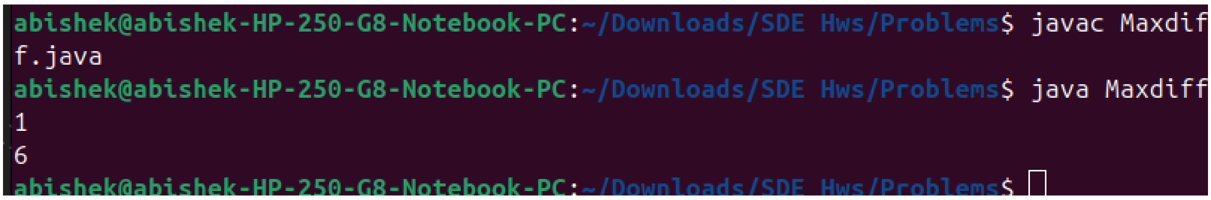
Program :

```
class Maxdiff {

    public int maxDifference(int[] arr) {

        int n = arr.length;

        int[] leftMin = new int[n];
```

```
int[] rightMax = new int[n];

leftMin[0] = arr[0];
for (int i = 1; i < n; i++) {
    leftMin[i] = Math.min(arr[i], leftMin[i - 1]);
}
rightMax[n - 1] = arr[n - 1];
for (int j = n - 2; j >= 0; j--) {
    rightMax[j] = Math.max(arr[j], rightMax[j + 1]);
}
int i = 0, j = 0;
int maxDiff = -1;
while (i < n && j < n) {
    if (leftMin[i] < rightMax[j]) {
        maxDiff = Math.max(maxDiff, j - i);
        j++;
    } else {
```

```
                i++;

            }

        }



        return maxDiff;

    }



    public static void main(String[] args) {

        Maxdiff ob = new Maxdiff();

        int arr1[] = {1, 10};

        System.out.println(ob.maxDifference(arr1));  // Output: 1

        int arr2[] = {34, 8, 10, 3, 2, 80, 30, 33, 1};

        System.out.println(ob.maxDifference(arr2));  // Output: 6

    }

}
```

Output :



```
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$ javac Maxdif
f.java
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$ java Maxdiff
1
6
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$ 
```

Time Complexity: O(n)

Space Complexity : O(n)

8)Wave Array

Given a sorted array arr[] of distinct integers. Sort the array into a wave-like array(In Place). In other words, arrange the elements into a sequence such that arr[1] >= arr[2] <= arr[3] >= arr[4] <= arr[5].....

If there are multiple solutions, find the lexicographically smallest one.

Note: The given array is sorted in ascending order, and you don't need to return anything to change the original array.

Examples:

Input: arr[] = [1, 2, 3, 4, 5]

Output: [2, 1, 4, 3, 5]

Explanation: Array elements after sorting it in the waveform are 2, 1, 4, 3, 5.

Input: arr[] = [2, 4, 7, 8, 9, 10]

Output: [4, 2, 8, 7, 10, 9]

Explanation: Array elements after sorting it in the waveform are 4, 2, 8, 7, 10, 9.

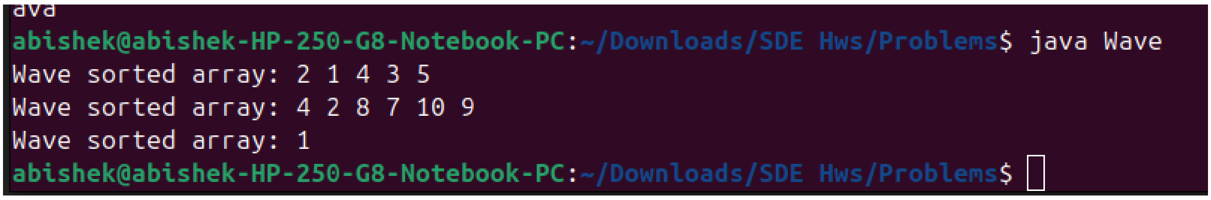
Input: arr[] = [1]

Output: [1]

Program :

```
class GFG {
    public void waveSort(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n - 1; i += 2) {
            if (arr[i] < arr[i + 1]) {
                int temp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = temp;
            }
            if (i - 1 >= 0 && arr[i - 1] > arr[i]) {
                int temp = arr[i - 1];
                arr[i - 1] = arr[i];
                arr[i] = temp;
            }
        }
    }
    public static void main(String[] args) {
        GFG ob = new GFG();
        int arr1[] = {1, 2, 3, 4, 5};
        ob.waveSort(arr1);
        System.out.print("Wave sorted array: ");
        for (int i : arr1) {
            System.out.print(i + " ");
        }
        System.out.println();
        int arr2[] = {2, 4, 7, 8, 9, 10};
        ob.waveSort(arr2);
        System.out.print("Wave sorted array: ");
        for (int i : arr2) {
            System.out.print(i + " ");
        }
        System.out.println();
        int arr3[] = {1};
        ob.waveSort(arr3);
        System.out.print("Wave sorted array: ");
        for (int i : arr3) {
            System.out.print(i + " ");
        }
        System.out.println();
    }
}
```

Output :



```
ava
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$ java Wave
Wave sorted array: 2 1 4 3 5
Wave sorted array: 4 2 8 7 10 9
Wave sorted array: 1
abishek@abishek-HP-250-G8-Notebook-PC:~/Downloads/SDE Hws/Problems$ 
```

Time Complexity : O(n)

Space Complexity : O(1)