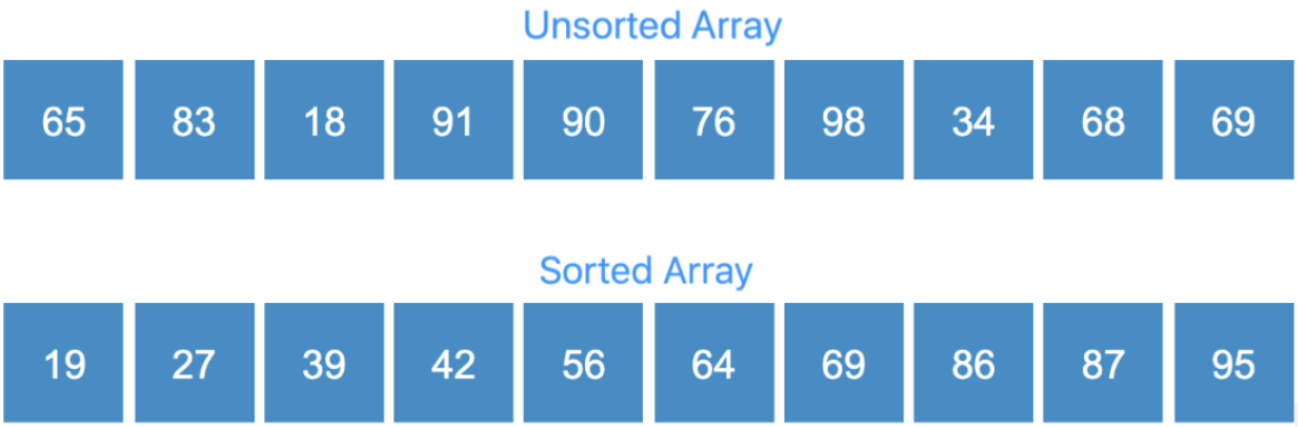
Lesson Plan

Basic Sorting Algorithms

PW SKILLS

What is Sorting?

A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.



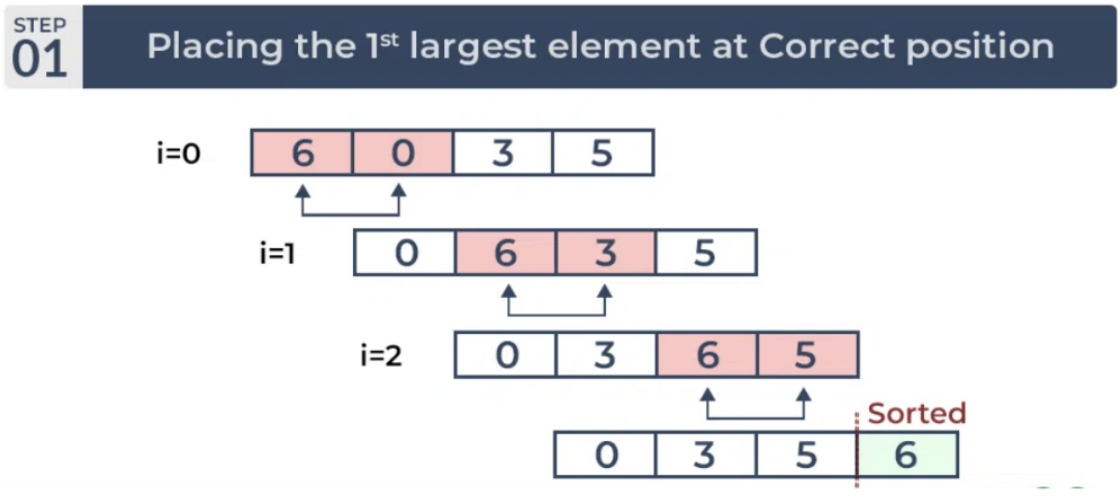
Bubble Sort Algorithm

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

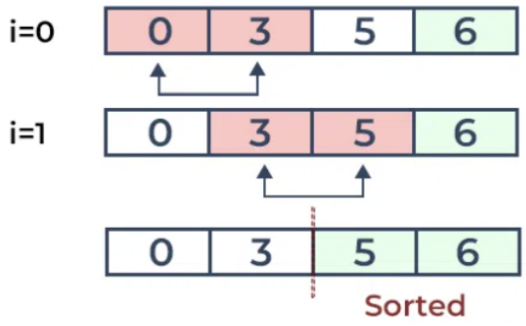
In Bubble Sort algorithm,

- traverse from left and compare adjacent elements and the higher one is placed at right side.
- In this way, the largest element is moved to the rightmost end at first.
- This process is then continued to find the second largest and place it and so on until the data is sorted.

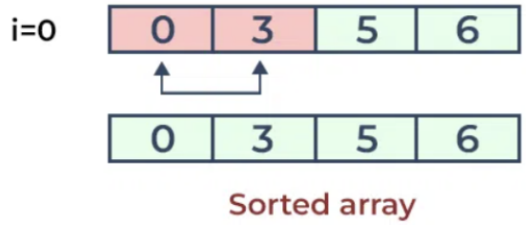
Let us understand the working of bubble sort with the help of the following illustration:
Input: arr[] = {6, 3, 0, 5}



STEP 02 Placing 2nd largest element at Correct position



STEP 03 Placing 3rd largest element at Correct position



Bubble sort Code:

```
static void bubbleSort(int arr[], int n)
    {
        int i, j, temp;
        boolean swapped;
        for (i = 0; i < n - 1; i++) {
                        for (j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {

                    // Swap arr[j] and arr[j+1]
                    temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                  }
            }

        }
    }
```

Time Complexity: O(N2), it compares every element with every other element in the worst-case scenario

Average Case O(n2), to compare and swap elements multiple times, especially when the list is unsorted or partially sorted

Best Case O(n),When the list is already sorted, requiring just one pass to confirm the sorted order. (after optimization)

Worst Case O(n2),When the list is sorted in reverse order, leading to maximum comparisons and swaps in each iteration.

Auxiliary Space: O(1), As are not using any extra space

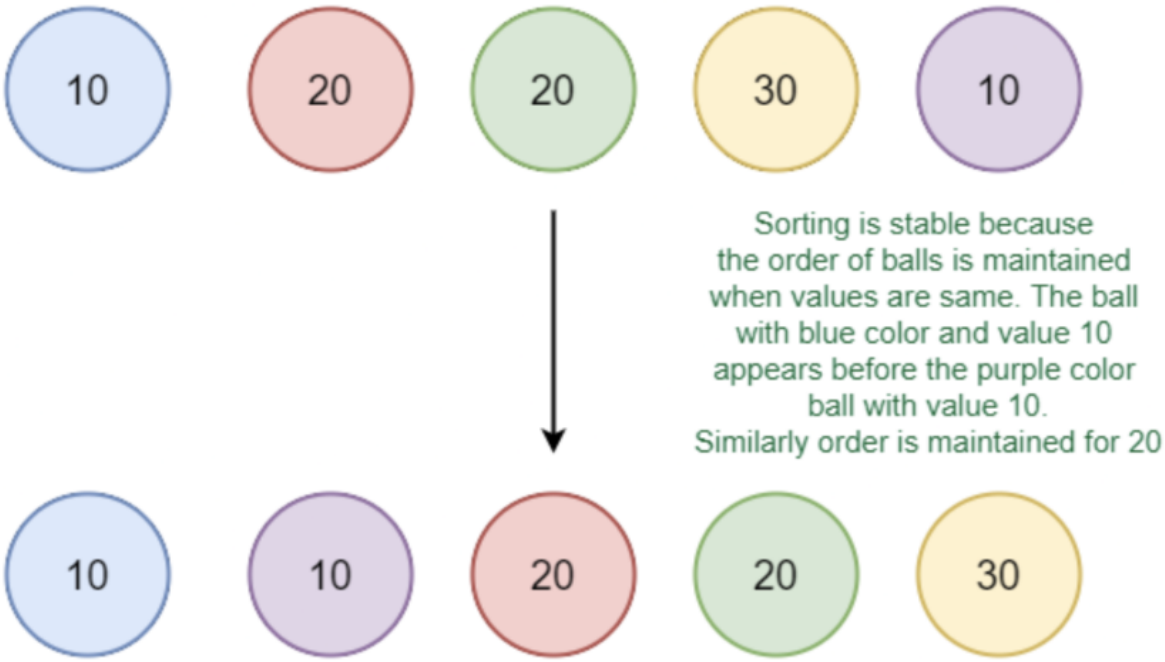
Optimization: It can be optimized by stopping the algorithm if the inner loop didn't cause any swap.

```
//optimized code
static void bubbleSort(int arr[], int n)
    {
        int i, j, temp;
        boolean swapped;
        for (i = 0; i < n - 1; i++) {
            swapped = false;
            for (j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {

                    // Swap arr[j] and arr[j+1]
                    temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = true;
                }
            }

            // If no two elements were
            // swapped by inner loop, then break
            if (swapped == false)
                break;
        }
    }
```

Now after optimization best case time complexity will become O(n)

Stable and Unstable sort

Stable sort: A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input data set
Unstable Sort: An unstable sort does not guarantee the preservation of the original order of equal elements



Example of stable sort

Some Sorting Algorithms are stable by nature, such as Bubble Sort, Insertion Sort, Merge Sort, Count Sort, etc

Unstable Sorting Algorithms: Quick Sort, Heap Sort, Selection Sort

Q. How much maximum swaps are needed to sort array of length 6 ?

Ans: Maximum swaps will be needed case the array is sorted in reverse order
Consider = {6,5,4,3,2,1}
//in first iteration 6 will swap with 5 then 4 upto 1 total 5 swaps
//in second 5 will swap with 4,3,2,1 -> 4
//third=>3swap,2nd=>2swaps,last=>1 swap
Total 5+4+3+2+1= 15 swaps will be required

//this can be generalised in worst case for an array of length n,
n-1+n-2+ .. +1 = (n)(n-1)/2 swpas are needed

Code:

```
public class BubbleSort {
    public static int bubbleSortWithSwapsCount(int[] arr) {
        int n = arr.length;
        int swaps = 0;
        boolean swapped;

        for (int i = 0; i < n - 1; i++) {
            swapped = false;
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
```

```
                    // Swap elements
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swaps++;
                    swapped = true;
                }
            }
            // If no swaps occurred, array is already sorted
            if (!swapped) {
                break;
            }
        }
        return swaps;
    }

    public static void main(String[] args) {
        int[] myArray = {6, 5, 4, 3, 2, 1};
        int numSwaps = bubbleSortWithSwapsCount(myArray);

        System.out.print("Sorted array: ");
        for (int value : myArray) {
            System.out.print(value + " ");
        }
        System.out.println();
        System.out.println("Number of swaps: " + numSwaps);
    }
}
```

Q. Sort a String in decreasing order of values associated after removal of values smaller than X.

Input: X = 79, str = "Even 78 Bob 99 Suzy 88 Alice 86"
Output: for 99 Geeks 88 Gfg 86

Explanation: "Even 78" is removed, since the number associated to it is smaller than X(= 79) Now, the reordered string string based on decreasing order of numbers associated is "Bob 99 Suzy 88 Alice 86".

Code:

```
// Java code for the above approach

import java.util.ArrayList;
import java.util.List;

public class Main {

 // Function to split the input
 // string into a list
```

```
 public static List<String> tokenizer(String Str) {
  List<String> List = new ArrayList<>();
  for (String s : Str.split("\\s+")) {
   List.add(s);
  }
  return List;
 }

 // Function to sort the given list based
 // on values at odd indices of the list
 public static String SortListByOddIndices(List<String>
List, int x) {

  int l = List.size();

  // Function to remove the values
  // less than the given value x
  for (int i = l - 1; i >= 0; i -= 2) {
   if (Integer.parseInt(List.get(i)) < x) {
    List.remove(i - 1);
    List.remove(i - 1);
   }
  }

  l = List.size();

  for (int i = 1; i < l; i += 2) {
   for (int j = 1; j < l - i; j += 2) {

    // Compares values at odd indices of
    // the list to sort the list
    if (List.get(j).compareTo(List.get(j + 2)) < 0
      || (List.get(j - 1).compareTo(List.get(j + 1)) > 0
      && List.get(j).equals(List.get(j + 2)))) {

     String temp1 = List.get(j);
     String temp2 = List.get(j - 1);

     List.set(j, List.get(j + 2));
     List.set(j + 2, temp1);

     List.set(j - 1, List.get(j + 1));
     List.set(j + 1, temp2);
    }
   }
  }

  return String.join(" ", List);
 }
```

```
 // Driver Code
 public static void main(String[] args) {

  String Str = "Axc 78 Czy 60";
  int x = 77;

  // Function call
  List<String> List = tokenizer(Str);

  Str = SortListByOddIndices(List, x);

  System.out.println(Str);
 }
}
```

Code Explanation: The idea is to use the Bubble Sort technique. Follow the steps below to solve the problem:

Split the string in to a list, then remove the entries that are less than the given value i.e. X.
Sort the list based on the number associated with it using bubble sort method.
If the numbers are not equal, sort the numbers in decreasing order and simultaneously sort the names.
If the numbers are equal then sort them lexicographically.
Swap both the strings and the number together in order to keep them together.

Time Complexity: O(N2) ,Bubble Sort takes O(N2)

Auxiliary Space: O(N), since are creating a new list to store the splitted string

Q. Push zeroes to end while maintaining the relative order of other elements.

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

Code:

```
static void pushZerosToEnd(int arr[], int n)
    {
        int count = 0;  // Count of non-zero elements


        for (int i = 0; i < n; i++)
            if (arr[i] != 0)
                arr[count++] = arr[i];
        while (count < n)
            arr[count++] = 0;
    }
```

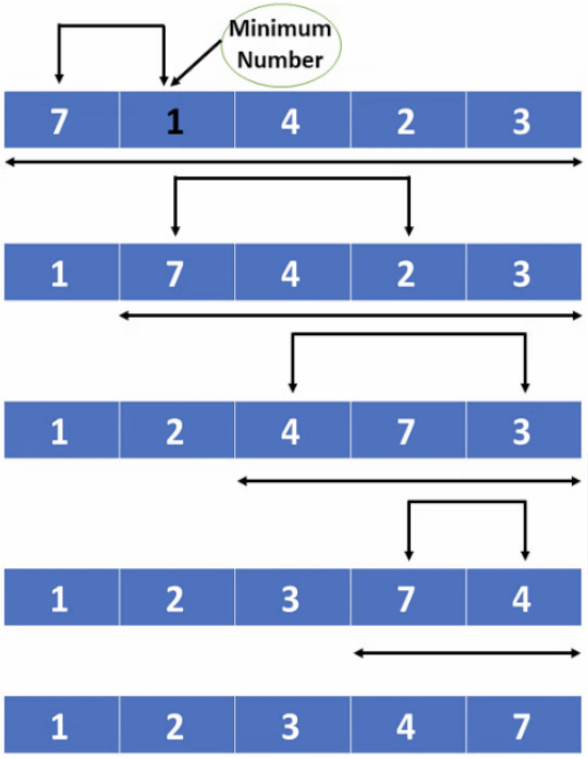
Explanation: Traverse the given array 'arr' from left to right. While traversing, maintain count of non-zero elements in array. Let the count be 'count'. For every non-zero element arr[i], put the element at 'arr[count]' and increment 'count'. After complete traversal, all non-zero elements have already been shifted to front end and 'count' is set as index of first 0. Now all we need to do is run a loop that makes all elements zero from 'count' till end of the array.

Time Complexity: O(n) ,traversing the array only once

Auxiliary Space: O(1), not using any extra space

Selection Sort: Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

Dry Run of selection sort:




```
void selectionSort(int arr[])
    {
        int n = arr.length;

        // One by one move boundary of unsorted subarray
        for (int i = 0; i < n-1; i++)
        {
            // Find the minimum element in unsorted array
            int min_idx = i;
            for (int j = i+1; j < n; j++)
                if (arr[j] < arr[min_idx])
                    min_idx = j;
```

```
            // Swap the found minimum element with the first
            // element
            int temp = arr[min_idx];
            arr[min_idx] = arr[i];
            arr[i] = temp;
        }
    }
```

Time Complexity: The time complexity of Selection Sort is O(N2) as there are two nested loops

Auxiliary Space: O(1) as the only extra memory used is for temporary variables

Insertion Sort: To sort an array of size N in ascending order iterate over the array and compare the current element (key) to its predecessor, if the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

The simple steps of achieving the insertion sort are listed as follows -

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step2 - Pick the next element, and store it separately in a key.

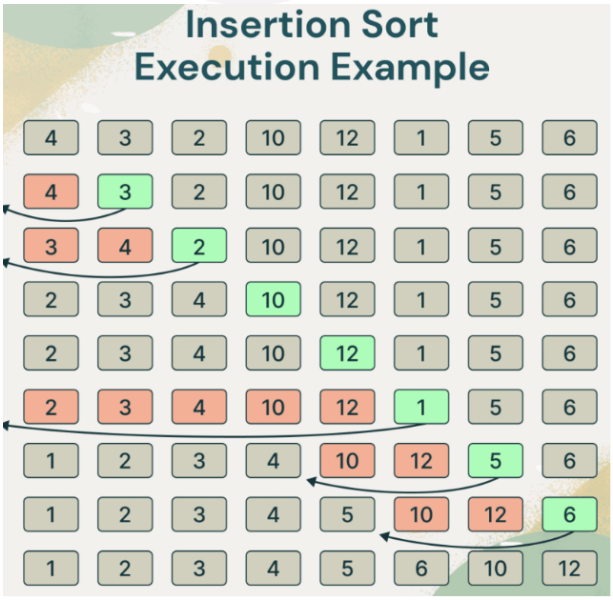
Step3 - Now, compare the key with all elements in the sorted array.

Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.

Dry Run:



Code:

```
void insertionSort(int arr[])
    {
        int n = arr.length;
        for (int i = 1; i < n; ++i) {
            int key = arr[i];
            int j = i - 1;

            /* Move elements of arr[0..i-1], that are
               greater than key, to one position ahead
               of their current position */
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j = j - 1;
            }
            arr[j + 1] = key;
        }
    }
```

Time Complexity: O(N^2), in the worst case, each element requires comparison and potential shifting with every preceding element before finding its correct position in the sorted sequence

The worst-case time complexity of the Insertion sort is O(N^2)

The average case time complexity of the Insertion sort is O(N^2)

The time complexity of the best case is O(N).

Auxiliary Space: O(1),no extra space used

Insertion sort is stable, as it maintains the relative order of equal elements while sorting. However, selection sort is unstable and might change the relative positions of elements with the same value during the sorting process.

Q. What will the array look like after the first iteration of selection sort [2,3,1,6,4]

[1,2,3,6,4]
[1,3,2,4,6]
[1,3,2,6,4]
[2,3,1,4,6]

Ans. [1,3,2,6,4], as in first run minimum element will be swapped by first element

Q. Which sorting technique is used here?
A player is sorting a deck of cards numbered from 1 to 52. She first picks one card then picks the next card and puts it after the first card if if it is bigger or before the first card if it is smaller, then she picks another card and puts it into its proper position.

a. Bubble sort
b. Insertion sort
c. Selection sort
d. None of these

Ans. Insertion Sort,each card is placed in its correct position by comparing it sequentially with the sorted cards and shifting elements as needed, resulting in a progressively sorted deck.

Q. Which of the following is not a stable sorting algorithm?

a) Insertion sort
b) Selection sort
c) Bubble sort
d) None of these

Ans. Selection Sort, Selection Sort is not stable because it doesn't ensure the preservation of the original order of equal elements.During the selection and swapping process, elements with the same value might change their relative positions in the sorted sequence compared to their original order in the input

Q. Majority Element (Leetcode 169)
Given an array nums of size n, return the majority element.
The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

Input: nums = [2,2,1,1,1,2,2]
Output: 2

Code:

```
class Solution {
    public int majorityElement(int[] nums) {
        Arrays.sort(nums);
        int n = nums.length;
        return nums[n/2];
    }
}
```

Explanation: Sort the array using any algorithm in non-dereasing order, since majority element occurs more than n/2 times it will always be present at middle position in sorted array so return element at index n/2

Time Complexity: O(nlogn) sorting , or O(n2) depends on algorihtm of sorting

Space: O(1),no space used

Q. Given an array with N distinct elements, convert the given array to a form where all elements are in the range from 0 to N-1. The order of elements is the same, i.e., 0 is placed in the place of the smallest element, 1 is placed for the second smallest element, ... N-1 is placed for the largest element.

Input: arr[] = {10, 40, 20}
Output: arr[] = {0, 2, 1}

```
public static void convert(int arr[], int n)
    {
        // Create a temp array and copy contents
        // of arr[] to temp
        int temp[] = arr.clone();

        // Sort temp array
        Arrays.sort(temp);

        // Create a hash table.
        HashMap<Integer, Integer> umap = new HashMap<>();

        // One by one insert elements of sorted
        // temp[] and assign them values from 0
        // to n-1
        int val = 0;
        for (int i = 0; i < n; i++)
            umap.put(temp[i], val++);

        // Convert array by taking positions from
        // umap
        for (int i = 0; i < n; i++)
            arr[i] = umap.get(arr[i]);
    }
```

Explanation: The idea is to sort the given array and use an unordered map to store the reduced form of each value of array then update the whole array to its reduced form using values from unordered map.

Time complexity: O(N * log N),bcz of soring

Auxiliary Space: O(N), for hashmap

Q. Assign Cookies (Leetcode 455)
Assume you are an awesome parent and want to give your children some cookies. But, you should give each child at most one cookie.

Each child i has a greed factor g[i], which is the minimum size of a cookie that the child will be content with; and each cookie j has a size s[j]. If s[j] >= g[i], we can assign the cookie j to the child i, and the child i will be content. Your goal is to maximize the number of your content children and output the maximum number.

Input: g = [1,2,3], s = [1,1]
Output: 1

Explanation: You have 3 children and 2 cookies. The greed factors of 3 children are 1, 2, 3.
And even though you have 2 cookies, since their size is both 1, you could only make the child whose greed factor is 1 content.
You need to output 1.

Code:

```
public int findContentChildren(int[] g, int[] s) {
      int i = 0;
      Arrays.sort(g);
      Arrays.sort(s);
      for(int j = 0; i< g.length && j < s.length; j++) {
        if(s[j] >= g[i]) i++;
      }
      return i;
    }
```

Explanation: sort both the array , now using greedy approach check if the current cookie satisfy the current children sicne after sorting this will child minimum satisfaction cost
If not move to next cookie, if yes increase the count then return the ans

Time complexity: O(N * log N),bcz of soring

Auxiliary Space: O(1), no extra space used expect of a const space int i

Q. Given an array, arr[] containing 'n' integers, the task is to find an integer (say K) such that after replacing each and every index of the array by |ai – K| where (i [1, n]), results in a sorted array. If no such integer exists that satisfies the above condition then return -1.

Input: arr[ ] = [10, 5, 4, 3, 2, 1]
Output: 8

Explanation: Upon performing the operation |ai-8|, we get [2, 3, 4, 5, 6, 7] which is sorted.

Code:

```
public static void find(int[] arr, int n)
    {
        // Initializing the two variables
        int l = 0;
        int r = 1000000000;
        for (int i = 0; i < n - 1; i++) {

            // If (a[i]<a[i+1]) then take minimum and store
in variable)

            if (arr[i] < arr[i + 1]) {
                r = Math.min(r, (arr[i] + arr[i + 1]) / 2);
            }

            // If (a[i]>a[i+1]) then take maximum and store
in separate variable)

            else if (arr[i] > arr[i + 1]) {
                l = Math.max(l,
                             (arr[i] + arr[i + 1] + 1) / 2);
```

```
            }
        }

        if (l > r) {
            System.out.println(-1);
        }
        else {
            System.out.println(l);
        }
    }
```

Explanation: For the array to be sorted each pair of adjacent elements should be sorted. That means few cases arise if we take care for particular ai and ai+1 and those are as follows:

Let (ai < ai+1), so the following inequality arise:
If (K <ai) then upon (ai – K < ai+1 – K) the elements will be as it is (ai < ai+1).
If (K > ai) then upon (K – ai > K – ai+1) the elements will be as it is (ai > ai+1).
So, K should be midway between ai and ai+1 that is K should be K = (ai + ai+1)/2 .
Similarly for (ai > ai+1) the value of k would be K = (ai + ai+1)/2
Finally we will take the minimum of all for which (K < ai) and maximum of all for which (K > ai).

Time Complexity: O(n), iterating the loop for once only.

Auxiliary Space: O(1), no extra space is used.

PW SKILLS

THANK YOU !

PW SKILLS