**]]]1- Reverse the array**

Input : arr[] = {1, 2, 3}

Output : arr[] = {3, 2, 1}

Input : arr[] = {4, 5, 1, 2}

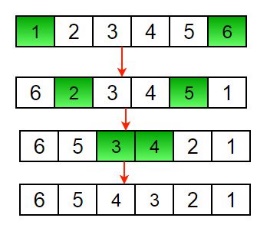
Output : arr[] = {2, 1, 5, 4}

**Algo:**

1) Initialize start and end indexes as start = 0, end = n-1

2) In a loop, swap arr[start] with arr[end] and change start and end as follows : ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]

start = start +1, end = end – 1



public class ReverseArrayIterative {  
 static void rvereseArray(int arr[], int start, int end) {  
 int temp;  
 while (start < end) {  
 temp = arr[start];  
 arr[start] = arr[end];  
 arr[end] = temp;  
 start++;  
 end--;  
 }  
 }  
  
 static void printArray(int arr[], int size) {  
 for (int i = 0; i < size; i++)  
 System.*out*.print(arr[i] + " ");  
 System.*out*.println();  
 }  
  
 public static void main(String args[]) {  
 int arr[] = {1, 2, 3, 4, 5, 6};  
 *printArray*(arr, 6);  
 *rvereseArray*(arr, 0, 5);  
 System.*out*.print("Reversed array is \n");  
 *printArray*(arr, 6);  
  
 }  
}

public class ReverseArrayRecurssive {  
 static void rvereseArray(int arr[], int start, int end) {  
 int temp;  
 if (start >= end)  
 return;  
 temp = arr[start];  
 arr[start] = arr[end];  
 arr[end] = temp;  
 *rvereseArray*(arr, start + 1, end - 1);  
 }  
  
 static void printArray(int arr[], int size) {  
 for (int i = 0; i < size; i++)  
 System.*out*.print(arr[i] + " ");  
 System.*out*.println();  
 }  
  
 public static void main(String args[]) {  
 int arr[] = {1, 2, 3, 4, 5, 6};  
 *printArray*(arr, 6);  
 *rvereseArray*(arr, 0, 5);  
 System.*out*.print("Reversed array is \n");  
 *printArray*(arr, 6);  
 }  
}

**1 2 3 4 5 6**

**Reversed array is**

**6 5 4 3 2 1**

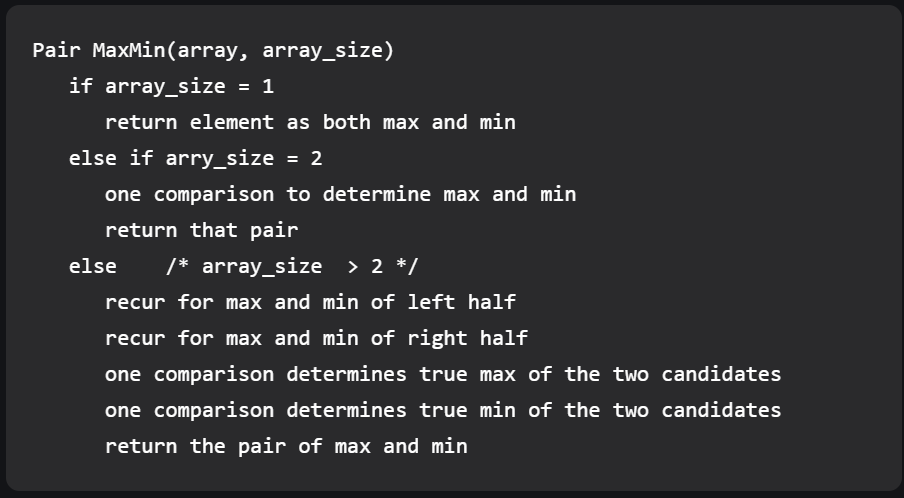
**2-Find the maximum and minimum element in an array**

**METHOD 1 (Simple Linear Search)**

* Initialize values of min and max as minimum and maximum of the first two elements respectively.
* Starting from 3rd, compare each element with max and min, and change max and min accordingly
* (i.e., if the element is smaller than min then change min, else if the element is greater than max then change max, else ignore the element)

**METHOD 2 (Tournament Method) – Merge sort – Divide part 😊**

* Divide the array into two parts and compare the maximums and minimums of the two parts to get the maximum and the minimum of the whole array.



public class MinMaxOfArray {  
 static class Pair {  
 int min;  
 int max;  
 }  
  
 static Pair getMinMax(int arr[], int low, int high) {  
 Pair minMax = new Pair();  
 Pair mmLeft = new Pair();  
 Pair mmRight = new Pair();  
 int mid;  
  
 // If there is only one element  
 if (low == high) {  
 minMax.max = arr[low];  
 minMax.min = arr[low];  
 return minMax;  
 }  
  
 /\* If there are two elements \*/  
 if (high == low + 1) {  
 if (arr[low] > arr[high]) {  
 minMax.max = arr[low];  
 minMax.min = arr[high];  
 } else {  
 minMax.max = arr[high];  
 minMax.min = arr[low];  
 }  
 return minMax;  
 }  
  
 /\* If there are more than 2 elements \*/  
 mid = (low + high) / 2;  
 mmLeft = *getMinMax*(arr, low, mid);  
 mmRight = *getMinMax*(arr, mid + 1, high);  
  
 /\* compare minimums of two parts\*/  
 if (mmLeft.min < mmRight.min) {  
 minMax.min = mmLeft.min;  
 } else {  
 minMax.min = mmRight.min;  
 }  
  
 /\* compare maximums of two parts\*/  
 if (mmLeft.max > mmRight.max) {  
 minMax.max = mmLeft.max;  
 } else {  
 minMax.max = mmRight.max;  
 }  
 return minMax;  
 }  
  
 public static void main(String args[]) {  
 int arr[] = {1000, 11, 445, 1, 330, 3000};  
 int arr\_size = 6;  
 Pair minmax = *getMinMax*(arr, 0, arr\_size - 1);  
 System.*out*.printf("\nMinimum element is %d", minmax.min);  
 System.*out*.printf("\nMaximum element is %d", minmax.max);  
 }  
}

Minimum element is 1

Maximum element is 3000

**3- Find the "Kth" max and min element of an array**

**Method 1 (Simple Solution)**

* A simple solution is to sort the given array using a O(N log N) sorting algorithm like Merge Sort, Heap Sort, etc, and return the element at index k-1 in the sorted array.
* Time Complexity of this solution is **O(N Log N)**

**Method 3 (Using Min Heap – HeapSelect)**

* We can find k’th smallest element in time complexity better than O(N Log N).
* A simple optimization is to create a Min Heap of the given n elements and call extractMin() k times.
* Time Complexity of this solution is **O(N Log K)**

public class KthSmallestNumber {  
 // Function to find the k'th smallest element in an array using min-heap  
 public static int findKthSmallest(List<Integer> input, int k) {  
 // base case  
 if (input == null || input.size() < k) {  
 System.*exit*(-1);  
 }  
  
 // create an empty min-heap and initialize it with all input elements  
 PriorityQueue<Integer> pq = new PriorityQueue<>(input);  
  
 // pop from min-heap exactly `k-1` times  
 while (--k > 0) {  
 pq.poll();  
 }  
  
 // return the root of min-heap  
 return pq.peek();  
 }  
  
 public static void main(String[] args) {  
 List<Integer> input = Arrays.*asList*(7, 4, 6, 3, 9, 1);  
 int k = 3;  
 System.*out*.println("k'th smallest array element is " + *findKthSmallest*(input, k));  
 }  
}

**4- Given an array which consists of only 0, 1 and 2. Sort the array without using any sorting algo**

**Algo (Dutch national flag problem)**

* Mid iterates through all the array
* If mid points to 0 => swap it with low
* If mid points to 1 => Leave it as it is and move forward. it’s a fixed pole 😊
* If mid points to 2 => swap it with high



public class SortZerosOnesTwos {  
 public static void main(String[] args) {  
 int arr[] = {0, 1, 1, 0, 1, 2, 1, 2, 0, 0, 0, 1};  
 int arr\_size = arr.length;  
 *sort012*(arr, arr\_size);  
 System.*out*.println("Array after seggregation ");  
 *printArray*(arr, arr\_size);  
 }  
  
 static void sort012(int input[], int arr\_size) {  
 int low = 0, mid = 0;  
 int high = arr\_size - 1;  
 while (mid <= high) {//mid iterates whole array from low to high  
 switch (input[mid]) {  
 case 0: {  
 input = *swap*(input, low, mid);  
 low++;  
 mid++;  
 break;  
 }  
 case 1:  
 mid++;  
 break;  
 case 2: {  
 input = *swap*(input, mid, high);  
 high--;  
 break;  
 }  
 }  
 }  
 }  
  
 static void printArray(int arr[], int arr\_size) {  
 int i;  
 for (i = 0; i < arr\_size; i++)  
 System.*out*.print(arr[i] + " ");  
 System.*out*.println("");  
 }  
  
 static int[] swap(int arr[], int index1, int index2) {  
 int temp = arr[index1];  
 arr[index1] = arr[index2];  
 arr[index2] = temp;  
 return arr;  
 }  
}

**5- Move all the negative elements to one side of the array (with O(1) extra space)**

**Two Pointer Approach:** The idea is to solve this problem with constant space and linear time is by using a two-pointer or two-variable approach where we simply take two variables like left and right which hold the 0 and N-1 indexes. Just need to check that:

* Check If the left and right pointer elements are negative then simply increment the left pointer.
* Otherwise, if the left element is positive and the right element is negative then simply swap the elements, and simultaneously increment and decrement the left and right pointers.
* Else if the left element is positive and the right element is also positive then simply decrement the right pointer.
* Repeat the above 3 steps until the left pointer ≤ right pointer.

public class ShiftNegativeNumbersLeft {  
 static void shiftAll(int[] arr, int left, int right) {  
 // Loop to iterate over the array from left to the right  
 while (left <= right) {  
 if (arr[left] < 0 && arr[right] < 0)  
 left++;  
 else if (arr[left] > 0 && arr[right] < 0) {  
 int temp = arr[left];  
 arr[left] = arr[right];  
 arr[right] = temp;  
 left++;  
 right--;  
 } else if (arr[left] > 0 && arr[right] > 0)  
 right--;  
 else {  
 left++;  
 right--;  
 }  
 }  
 }  
  
 static void display(int[] arr, int right) {  
 for (int i = 0; i <= right; ++i)  
 System.*out*.print(arr[i] + " ");  
 System.*out*.println();  
 }  
  
 public static void main(String[] args) {  
 int[] arr = {-12, 11, -13, -5, 6, -7, 5, -3, 11};  
 int arr\_size = arr.length;  
 *shiftAll*(arr, 0, arr\_size - 1);  
 *display*(arr, arr\_size - 1);  
 }  
}

**-12 -3 -13 -5 -7 6 5 11 11**

**6- Find the Union and Intersection of the two sorted arrays.**

* Use two index variables i and j, initial values i = 0, j = 0
* If arr1[i] is smaller than arr2[j] then print arr1[i] and increment i.
* If arr1[i] is greater than arr2[j] then print arr2[j] and increment j.
* If both are same then print any of them and increment both i and j.
* Print remaining elements of the larger array.

public class UnionOfTwoSortedArrays {  
 /\* m is the number of elements in arr1[]  
 n is the number of elements in arr2[] \*/  
 static void printUnion(int arr1[], int arr2[], int m, int n) {  
 int i = 0, j = 0;  
 while (i < m && j < n) {  
 if (arr1[i] < arr2[j])  
 System.*out*.print(arr1[i++] + " ");  
 else if (arr2[j] < arr1[i])  
 System.*out*.print(arr2[j++] + " ");  
 else {  
 System.*out*.print(arr2[j++] + " ");  
 i++;  
 }  
 }  
 /\* Print remaining elements of the larger array \*/  
 while (i < m)  
 System.*out*.print(arr1[i++] + " ");  
 while (j < n)  
 System.*out*.print(arr2[j++] + " ");  
 }  
  
 public static void main(String args[]) {  
 int arr1[] = {1, 2, 4, 5, 6};  
 int arr2[] = {2, 3, 5, 7};  
 int m = arr1.length;  
 int n = arr2.length;  
 *printUnion*(arr1, arr2, m, n);  
 }  
}

1 2 3 4 5 6 7

Intersection:

* Use two index variables i and j, initial values i = 0, j = 0
* If arr1[i] is smaller than arr2[j] then increment i.
* If arr1[i] is greater than arr2[j] then increment j.
* If both are same then print any of them and increment both i and j.

Last step is not needed itselt!

**7- Merge 2 sorted arrays without using Extra space.**

1) Initialize i,j,k as 0,0,n-1 where n is size of arr1

2) Iterate through every element of arr1 and arr2 using two pointers i and j respectively

if arr1[i] is less than arr2[j]

increment i

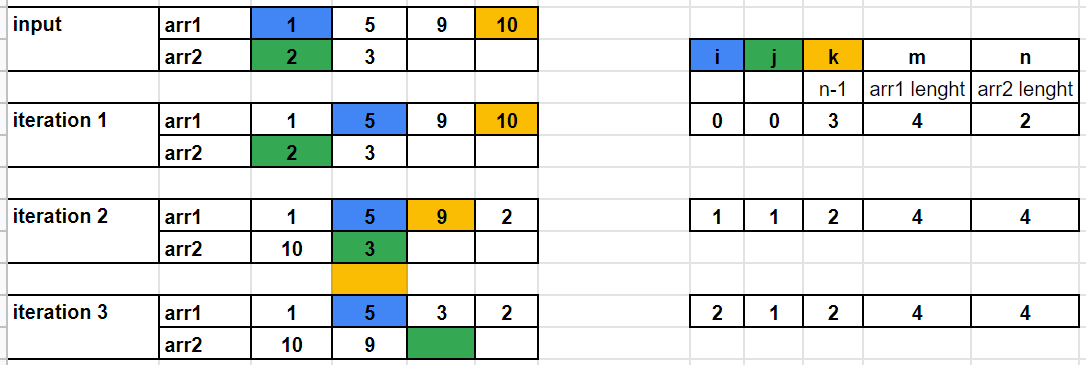
else

swap the arr2[j] and arr1[k]

increment j and decrement k

3) Sort both arr1 and arr2

class MergeTwoSortedArraysWithoutSpace {  
 static int *arr1*[] = new int[]{1, 5, 9, 10};  
 static int *arr2*[] = new int[]{2, 3};  
  
 static void merge(int m, int n) {  
 int i = 0, j = 0, k = m - 1;  
 while (i <= k && j < n) {  
 if (*arr1*[i] < *arr2*[j])  
 i++;  
 else {  
 int temp = *arr2*[j];  
 *arr2*[j] = *arr1*[k];  
 *arr1*[k] = temp;  
 j++;  
 k--;  
 }  
 }  
 Arrays.*sort*(*arr1*);  
 Arrays.*sort*(*arr2*);  
 }  
  
 public static void main(String[] args) {  
 *merge*(*arr1*.length, *arr2*.length);  
 System.*out*.print("After Merging \nFirst Array: ");  
 System.*out*.println(Arrays.*toString*(*arr1*));  
 System.*out*.print("Second Array: ");  
 System.*out*.println(Arrays.*toString*(*arr2*));  
 }  
}



**Complexities**:

**Time Complexity:** The time complexity while traversing the arrays in while loop is O(n+m) in worst case and sorting is O(nlog(n) + mlog(m)). So overall time complexity of the code becomes O((n+m)log(n+m)).

**Space Complexity:** As the function doesn’t use any extra array for any operations, the space complexity is O(1).

**8-Merge Overlapping Intervals**

* For example, let the given set of intervals be {{1,3}, {2,4}, {5,7}, {6,8}}.
* The intervals {1,3} and {2,4} overlap with each other, so they should be merged and become {1, 4}.
* Similarly, {5, 7} and {6, 8} should be merged and become {5, 8}

**Algorithm:**

1. Sort the intervals based on increasing order of

starting time.

2. Push the first interval on to a stack.

3. For each interval do the following

a. If the current interval does not overlap with the stack top,

push it.

b. If the current interval overlaps with stack top and ending time of current interval is more than that of stack top,

update stack top with the ending time of current interval.

4. At the end stack contains the merged intervals.

public class MergeOverlappingIntervals {  
 public static void mergeIntervals(Interval[] intervalArray) {  
 // Base Case  
 if (intervalArray.length <= 0)  
 return;  
  
 // Create an empty stack of intervals  
 Stack<Interval> stack = new Stack<>();  
  
 // sort the intervals in increasing order of start time  
 Arrays.*sort*(intervalArray, new Comparator<Interval>() {  
 public int compare(Interval i1, Interval i2) {  
 return i1.start - i2.start;  
 }  
 });  
  
 System.*out*.println("Interval Array after sorting...");  
 *printArray*(intervalArray);  
  
 // push the first interval to stack  
 stack.push(intervalArray[0]);  
  
 // Start from the next interval and merge if necessary  
 for (int i = 1; i < intervalArray.length; i++) {  
 //get interval from stack top.  
 //peek() Looks at the object at the top of this stack without removing it.  
 Interval top = stack.peek();  
  
 // if current interval is not overlapping , push it.  
 if (top.end < intervalArray[i].start)  
 stack.push(intervalArray[i]);  
  
 // Otherwise update the ending time of top if ending of current interval is more  
 else if (top.end < intervalArray[i].end) {  
 top.end = intervalArray[i].end;  
 stack.pop();  
 stack.push(top);  
 }  
 }  
  
 System.*out*.print("The Merged Intervals are: ");  
 while (!stack.isEmpty()) {  
 Interval t = stack.pop();  
 System.*out*.print("[" + t.start + "," + t.end + "] ");  
 }  
 }  
  
 public static void main(String args[]) {  
 Interval arr[] = new Interval[4];  
 arr[0] = new Interval(1, 3);  
 arr[1] = new Interval(6, 8);  
 arr[2] = new Interval(2, 4);  
 arr[3] = new Interval(5, 7);  
   
 System.*out*.println("Interval Array initially...");  
 *printArray*(arr);  
 *mergeIntervals*(arr);  
 }  
  
 private static void printArray(Interval arr[]) {  
 for (int i = 0; i < arr.length; i++) {  
 System.*out*.println(arr[i] + " ");  
 }  
 }  
}  
  
class Interval {  
 int start, end;  
  
 Interval(int start, int end) {  
 this.start = start;  
 this.end = end;  
 }  
  
 @Override  
 public String toString() {  
 return "Interval{" +  
 "start=" + start +  
 ", end=" + end +  
 '}';  
 }  
}

Time complexity of the method is **O(nLogn)** which is for sorting.

Once the array of intervals is sorted, merging takes linear time.

Interval Array initially...

Interval{start=1, end=3}

Interval{start=6, end=8}

Interval{start=2, end=4}

Interval{start=5, end=7}

Interval Array after sorting...

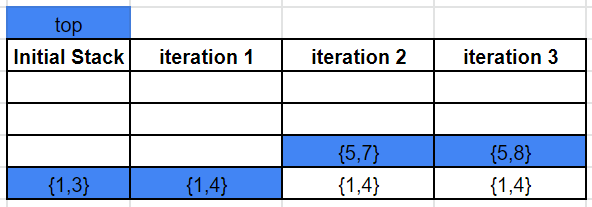
Interval{start=1, end=3}

Interval{start=2, end=4}

Interval{start=5, end=7}

Interval{start=6, end=8}

The Merged Intervals are: [5,8] [1,4]



**9 - Next Permutation**

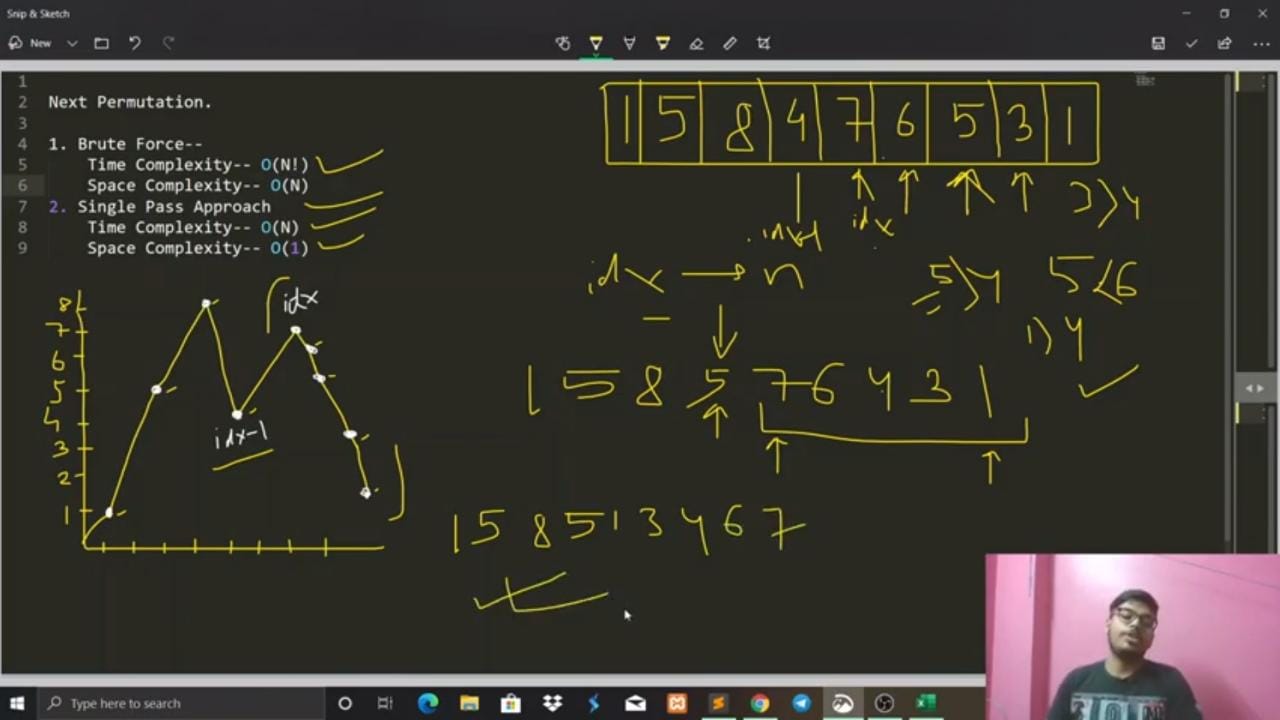
Implement the next permutation, which rearranges the list of numbers into Lexicographically next greater permutation of list of numbers. If such arrangement is not possible, it must be rearranged to the lowest possible order i.e. sorted in an ascending order

Suppose we have an integer n. We have to return 1 to n in lexicographic order. So for example when 13 is given, then the output will be [1, 10, 11, 12, 13, 2, 3, 4, 5, 6, 7, 8, 9].

**Algo :**

1. Iterate from right side.
2. Identify such element, where the number to left is less than the number to right (both should be adjacent)
   1. Store it as idx
   2. Previous element will become idx – 1
3. Identify the element bigger and nearest to idx – 1 in array from idx to end of input array
4. Swap the numbers identified in #3 and idx – 1
5. Reverse elements from idx to end.

That’s it.



public class NextPermutation {  
 public static boolean findNextPermutation(int data[]) {  
 //Base Case  
 if (data.length <= 1)  
 return false;  
  
 int last = data.length - 2;  
  
 while (last >= 0) {  
 if (data[last] < data[last + 1]) {  
 break;  
 }  
 last--;  
 }  
  
 // If there is no increasing pair there is no higher order permutation  
 if (last < 0)  
 return false;  
  
 int nextGreater = data.length - 1;  
  
 // Find the rightmost successor to the pivot  
 for (int i = data.length - 1; i > last; i--) {  
 if (data[i] > data[last]) {  
 nextGreater = i;  
 break;  
 }  
 }  
  
 // Swap the successor and the pivot  
 data = *swap*(data, nextGreater, last);  
  
 // Reverse the suffix  
 data = *reverse*(data, last + 1, data.length - 1);  
  
 return true;  
 }  
  
 public static int[] swap(int data[], int left, int right) {  
 int temp = data[left];  
 data[left] = data[right];  
 data[right] = temp;  
 return data;  
 }  
  
 public static int[] reverse(int data[], int left, int right) {  
 while (left < right) {  
 int temp = data[left];  
 data[left++] = data[right];  
 data[right--] = temp;  
 }  
 return data;  
 }  
  
 public static void main(String args[]) {  
 int data[] = {1, 5, 8, 4, 7, 6, 5, 3, 1};  
 if (!*findNextPermutation*(data))  
 System.*out*.println("There is no higher order permutation for input");  
 else {  
 System.*out*.println(Arrays.*toString*(data));  
 }  
 }  
}

<https://youtu.be/TOvWyFT0xCw> -> turorial.

**10 - Inversion Count for an array indicates –**

how far (or close) the array is from being sorted. If the array is already sorted, then the inversion count is 0, but if the array is sorted in the reverse order, the inversion count is the maximum.

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

Input: arr[] = {8, 4, 2, 1}

Output: 6

Explanation: Given array has six inversions:

(8, 4), (4, 2), (8, 2), (8, 1), (4, 1), (2, 1).

Input: arr[] = {3, 1, 2}

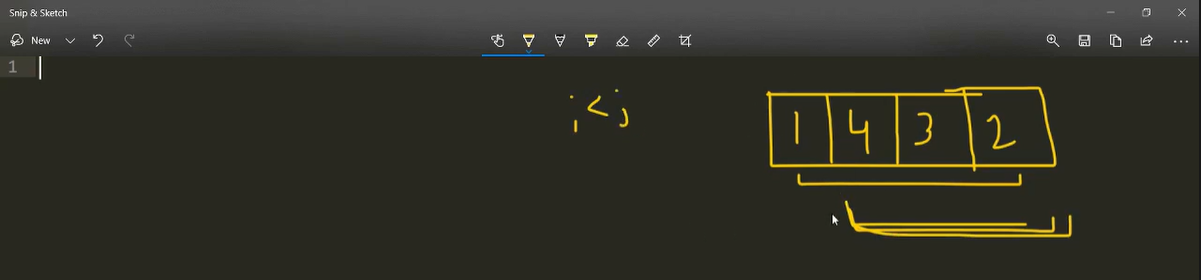
Output: 2

Explanation: Given array has two inversions:

(3, 1), (3, 2)

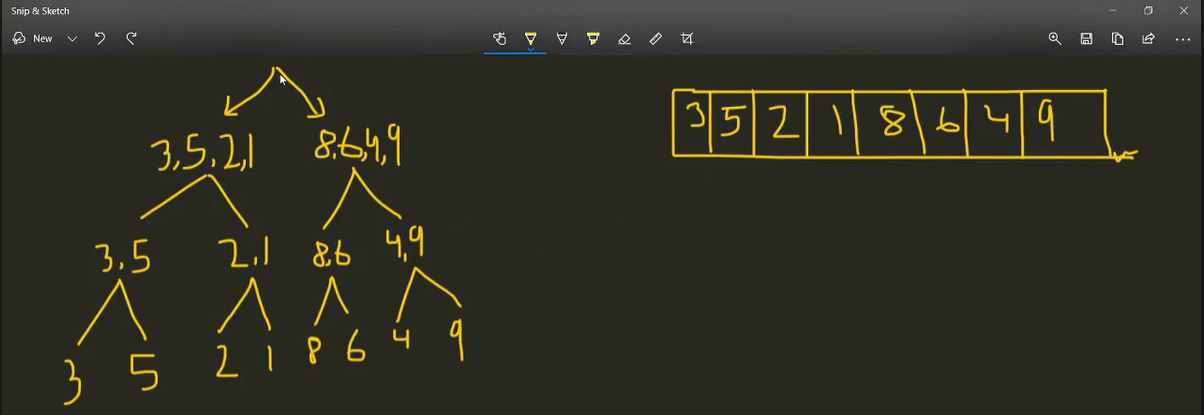
**Algorithm: -**

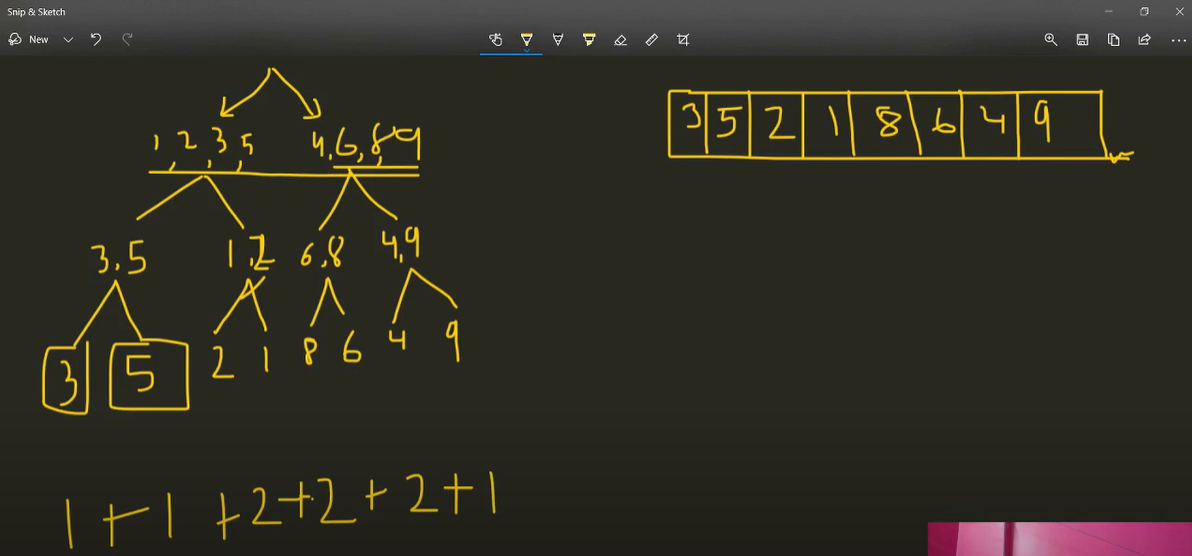
* Traverse through the array from start to end
* For every element, find the count of elements smaller than the current number up to that index using another loop.
* Sum up the count of inversion for every index.
* Print the count of inversions.



**Merge Sort Solution:**

1. The idea is similar to merge sort, divide the array into two equal or almost equal halves in each step until the base case is reached.
2. Create a function merge that counts the number of inversions when two halves of the array are merged,
   1. create two indices i and j, i is the index for the first half, and j is an index of the second half.
   2. if a[i] is greater than a[j], then there are (mid – i) inversions. because left and right subarrays are sorted,
   3. so all the remaining elements in left-subarray (a[i+1], a[i+2] … a[mid]) will be greater than a[j].
3. Create a recursive function to divide the array into halves and find the answer by summing the number of inversions is the first half, the number of inversions in the second half and the number of inversions by merging the two.
4. The base case of recursion is when there is only one element in the given half.
5. Print the answer





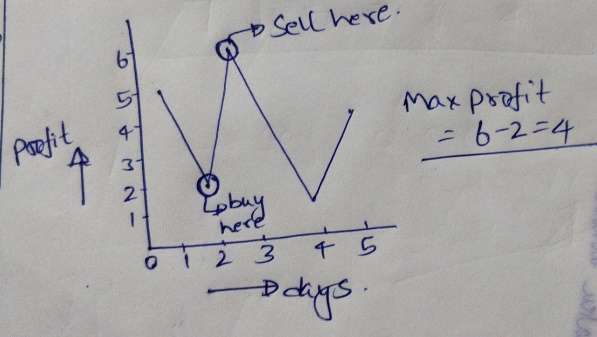
public class InversionCountMergeSort {  
 private static int mergeSortAndCount(int[] arr, int l, int r) {  
 int invCount = 0;  
  
 if (l < r) {  
 int m = (l + r) / 2;  
  
 // Total inversion count = left subArray count + right subArray count + merge count  
 invCount += *mergeSortAndCount*(arr, l, m);  
 invCount += *mergeSortAndCount*(arr, m + 1, r);  
 invCount += *mergeAndCount*(arr, l, m, r);  
 }  
 return invCount;  
 }  
  
 private static int mergeAndCount(int[] arr, int left, int mid, int right) {  
 int[] leftArray = Arrays.*copyOfRange*(arr, left, mid + 1);  
 int[] rightArray = Arrays.*copyOfRange*(arr, mid + 1, right + 1);  
  
 int i = 0, j = 0, k = left, invCount = 0;  
  
 while (i < leftArray.length && j < rightArray.length) {  
 if (leftArray[i] <= rightArray[j])  
 arr[k++] = leftArray[i++];  
 else {  
 arr[k++] = rightArray[j++];  
 invCount += (mid + 1) - (left + i);  
 }  
 }  
 while (i < leftArray.length)  
 arr[k++] = leftArray[i++];  
 while (j < rightArray.length)  
 arr[k++] = rightArray[j++];  
 return invCount;  
 }  
  
 public static void main(String[] args) {  
 int[] arr = {3, 5, 2, 1, 8, 6, 4, 9};  
 System.*out*.println(*mergeSortAndCount*(arr, 0, arr.length - 1));  
 }  
}

**Time Complexity: O(n log n)**, The algorithm used is divide and conquer, So in each level, one full array traversal is needed, and there are log n levels, so the time complexity is O(n log n).

**Space Complexity: O(n)**, Temporary array.

**11- Stocks Buy and Sell to maximize profit (buy or sell possible only once)**

Example:



Brute force is to

1. iterate over array
2. for every next day to the current iteration, check if bought on that day and sold, what is the profit.
3. Store if more than maxProfit
4. Display maxProfit

Time Complexity – O(n^2)

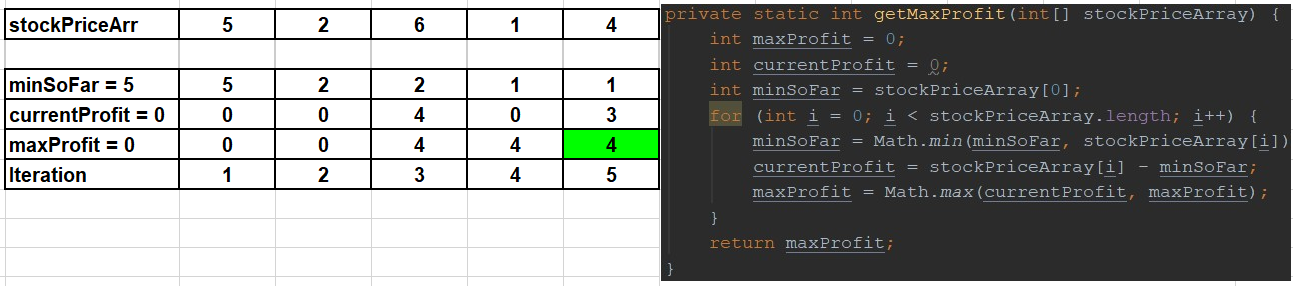
Space Complexity – O(1)

**Better Solution:**

Maintain the variables mention in the below code snippet and complete calculation in one iteration only 😊

Time Complexity – O(n)

Space Complexity – O(1)



public class StockBuySell {  
 private static int getMaxProfit(int[] stockPriceArray) {  
 int maxProfit = 0;  
 int currentProfit = 0;  
 int minSoFar = stockPriceArray[0];  
 for (int i = 0; i < stockPriceArray.length; i++) {  
 minSoFar = Math.*min*(minSoFar, stockPriceArray[i]);  
 currentProfit = stockPriceArray[i] - minSoFar;  
 maxProfit = Math.*max*(currentProfit, maxProfit);  
 }  
 return maxProfit;  
 }  
  
 public static void main(String[] args) {  
 int[] stockPriceArray = {5, 2, 6, 1, 4};  
 System.*out*.println(*getMaxProfit*(stockPriceArray));  
 }  
}