Array

1. **Find the Missing Number in an Array**

**public** **class** TestDemo {

**public** **static** **void** main(String[] args) {

**int**[] arr = { 1, 2, 4, 5, 6 };

**int** n = arr.length + 1;

**int** expectedSum = n \* (n + 1) / 2;

**int** actualSum = 0;

**for** (**int** val : arr) {

actualSum += val;

}

**int** missingElement = expectedSum - actualSum;

System.***out***.println(missingElement);

}

}

**Complexity Analysis:**

* **Time Complexity:** O(n) (Single loop to calculate actualSum)
* **Space Complexity:** O(1) (Only a few integer variables are used)

==============================================================

**2:- Find Duplicates in an Array**



**3:- merged two sorted array**

**package** test;

**import** java.util.HashSet;

**public** **class** TestDemo {

**public** **static** **void** main(String[] args) {

**int**[] arr1 = { 1, 3, 5, 7 };

**int**[] arr2 = { 2, 4, 6, 8 };

**int**[] mergedArray = **new** **int**[arr1.length + arr2.length];

**int** i = 0, j = 0, k = 0;

**while** (i < arr1.length && j < arr2.length) {

**if** (arr1[i] < arr2[j]) {

mergedArray[k++] = arr1[i++];

} **else** {

mergedArray[k++] = arr2[j++];

}

}

**while** (i < arr1.length) {

mergedArray[k++] = arr1[i++];

}

**while** (j < arr2.length) {

mergedArray[k++] = arr2[j++];

}

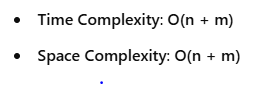
**for**(**int** val: mergedArray) {

System.***out***.print(val +" ");

}

}

}



**4:- Find the First and Second Smallest Numbers in an Array**

**package** test;

**public** **class** TestDemo {

**public** **static** **void** main(String[] args) {

**int**[] arr = { 3, 1, 5, 2, 4 };

**int** firstSmallest = Integer.***MAX\_VALUE***;

**int** secondSmallest = Integer.***MAX\_VALUE***;

**for** (**int** val : arr) {

**if** (val < firstSmallest) {

secondSmallest = firstSmallest;

firstSmallest = val;

} **else** **if** (val < secondSmallest && secondSmallest != firstSmallest) {

secondSmallest = val;

}

}

System.***out***.println(firstSmallest + " " + secondSmallest);

}

}

**Time and Space Complexity:**

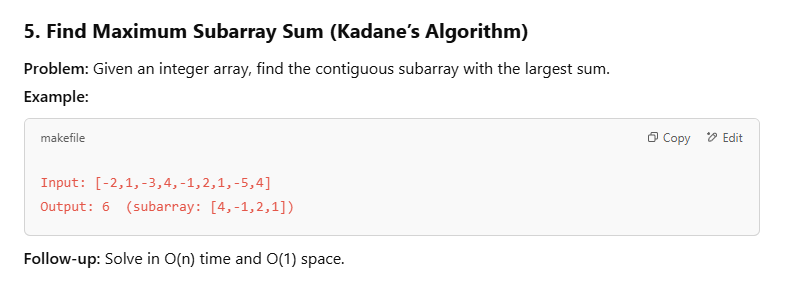
1. **Time Complexity**:
   * The program only loops through the array **once** using a single for loop, making it **O(n)**, where n is the length of the array.
2. **Space Complexity**:
   * The space complexity is **O(1)** because you only use a constant amount of space (for the firstSmallest and secondSmallest variables) and don't use any extra space that grows with the input size.

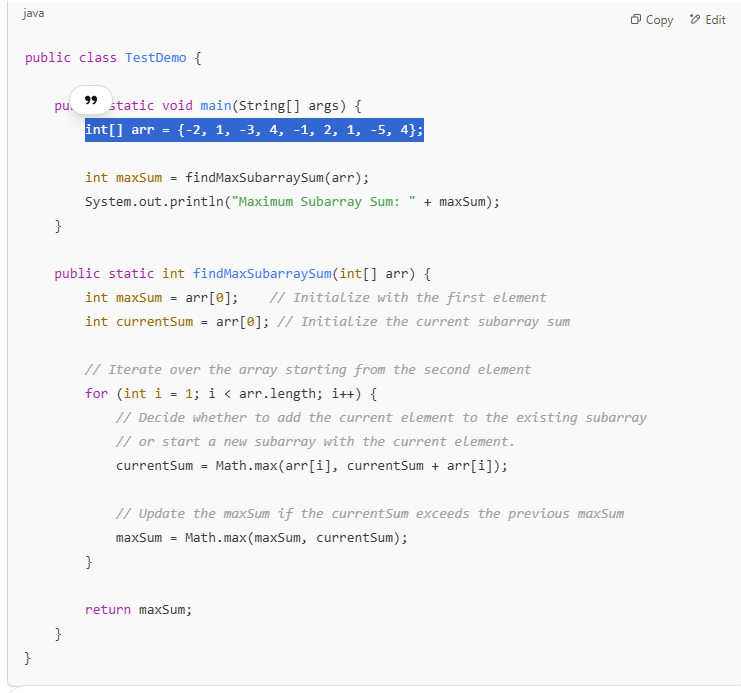
**Conclusion:**

Your code is efficient and works correctly, and the complexities are:

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

Kadane’s algorithm





**Time Complexity:**

* **O(n)**: The algorithm only iterates through the array once, so the time complexity is linear in terms of the number of elements in the array.

**Space Complexity:**

* **O(1)**: The algorithm uses only a constant amount of extra space for the variables maxSum and currentSum.

6:- Longest Consecutive Sequence

**package** test;

**import** java.util.HashMap;

**public** **class** TestDemo {

**public** **static** **void** main(String[] args) {

**int**[] arr = { 100, 4, 200, 1, 3, 2 };

HashMap<Integer, Boolean> hm = **new** HashMap<>();

**for** (**int** val : arr) {

hm.put(val, **true**);

}

**for** (**int** val : arr) {

**if** (hm.containsKey(val - 1)) {

hm.put(val, **false**);

}

}

**int** ml = 0;

**int** msp = 0;

**for** (**int** val : arr) {

**if** (hm.get(val) == **true**) {

**int** tl = 1;

**int** tsp = val;

**while** (hm.containsKey(tsp + tl)) {

tl++;

}

**if** (tl > ml) {

ml = tl;

msp = tsp;

}

}

}

**for** (**int** i = 0; i < ml; i++) {

System.***out***.print(msp + i +" ");

}

}

}

**Total Time Complexity:**

* **O(n)** for the first loop
* **O(n)** for the second loop
* **O(n^2)** for the third loop (due to the nested while loop)

Thus, the **overall time complexity** is **O(n^2)**.

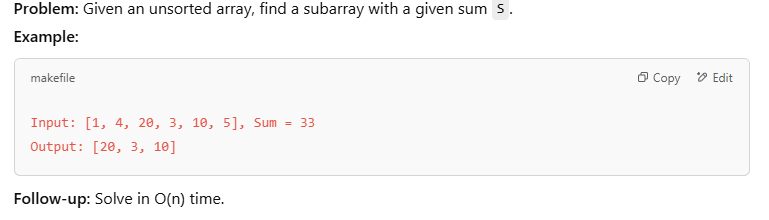
**Space Complexity Analysis:**

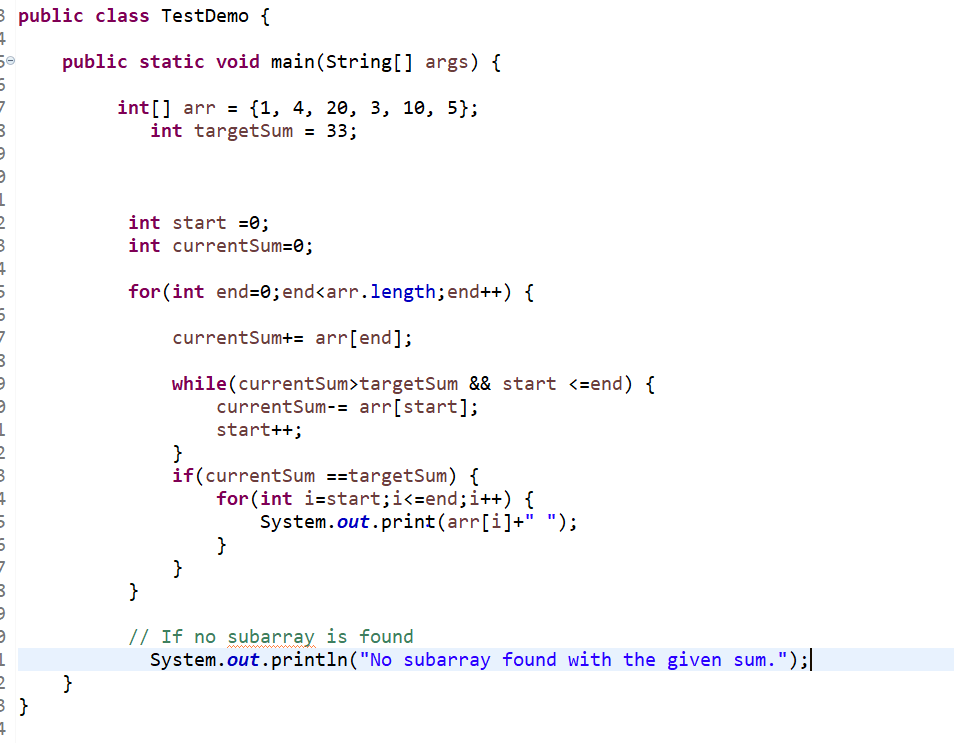
1. **HashMap hm**:
   * You store each element of the array arr in the HashMap with the value true or false, so the space required for the HashMap is **O(n)**.
2. **Other variables (ml, msp, tl, tsp)**:
   * These variables require constant space, i.e., **O(1)**.

**Total Space Complexity:**

* The space complexity is primarily determined by the HashMap, which requires **O(n)** space.

7:---- Find a Subarray with a Given Sum (Sliding Window)

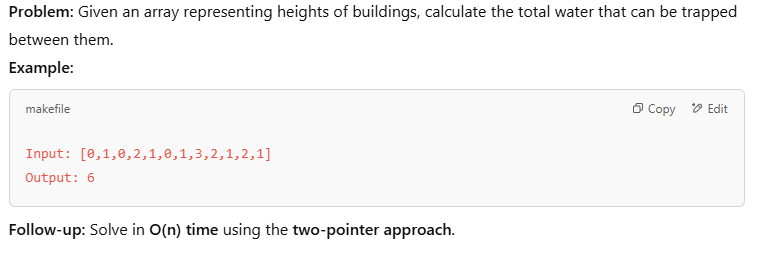




// output 20 3 10

Time complexity – o(n) and space complexity :- o(1)

8:- Rain Trapping water



**package** test;

**public** **class** TestDemo {

**public** **static** **void** main(String[] args) {

**int**[] heights = { 0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1 };

**int** trap = *trap*(heights);

System.***out***.println(trap);

}

**private** **static** **int** trap(**int**[] heights) {

**int** start = 0;

**int** end = heights.length - 1;

**int** leftMax = 0;

**int** rightMax = 0;

**int** waterTrapped = 0;

**while** (start <= end) {

**if** (heights[start] <= heights[end]) {

**if** (heights[start] > leftMax) {

leftMax = heights[start];

} **else** {

waterTrapped += leftMax - heights[start];

}

start++;

} **else** {

**if** (heights[end] > rightMax) {

rightMax = heights[end];

} **else** {

waterTrapped += rightMax - heights[end];

}

end--;

}

}

**return** waterTrapped;

}

}

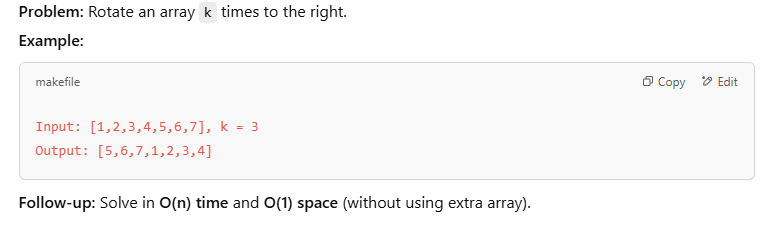
**Time Complexity:**

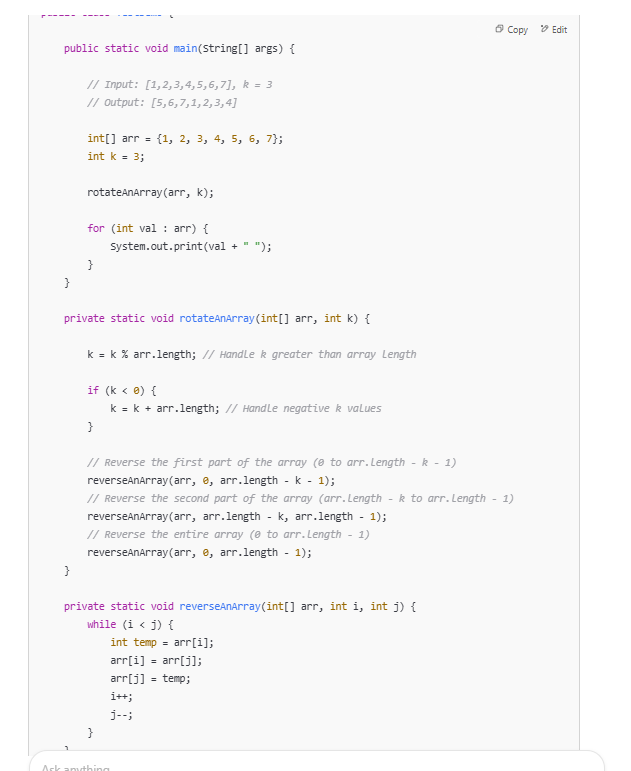
* **O(n)**: We traverse the array once, where n is the length of the array. Each element is visited at most twice (once by the left pointer and once by the right pointer).

**Space Complexity:**

* **O(1)**: We are only using a constant amount of extra space to store leftMax, rightMax, and waterTrapped, so the space complexity is constant.

9:-- Rotate an Array by K Steps (In-Place)





**Time Complexity:**

* **O(n)**: We are doing constant work for each element of the array in all three reverse steps. The reverse operation takes linear time (O(n)), and it is done three times, but the total time complexity remains **O(n)**.

**Space Complexity:**

* **O(1)**: We are performing the rotations in place, so we do not use any extra space for storing the result. The space complexity is constant.

**Output:**

For the input array [1, 2, 3, 4, 5, 6, 7] and k = 3, the output will be:

CopyEdit

5 6 7 1 2 3 4

This successfully rotates the array 3 times to the right.

==========================================================================

10 : intersection between two array

**package** test;

**import** java.util.HashSet;

**public** **class** TestDemo {

**public** **static** **void** main(String[] args) {

// Example input arrays

**int**[] arr1 = { 1, 2, 2, 1 };

**int**[] arr2 = { 2, 2 };

// Find the intersection and print the result

*findIntersection*(arr1, arr2);

}

**public** **static** **void** findIntersection(**int**[] arr1, **int**[] arr2) {

// Create a hash set to store elements from arr1

HashSet<Integer> set = **new** HashSet<>();

// Add all elements from arr1 to the set

**for** (**int** num : arr1) {

set.add(num);

}

// Iterate through arr2 and print the intersection elements

**for** (**int** num : arr2) {

**if** (set.contains(num)) {

System.***out***.print(num + " ");

set.remove(num); // Remove to avoid duplicates in the intersection

}

}

}

}