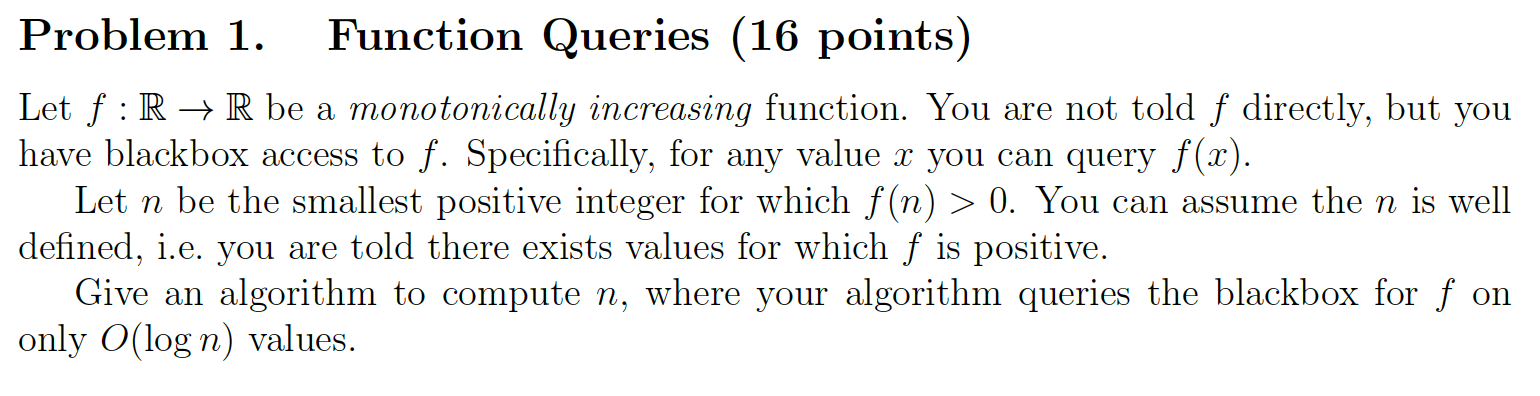
**Homework 2**

**Indukuri Vijay Anand Varma (VXI210000)**

**Karthik Raghunath Ananda Kumar (KXA200005)**



**Intuition:**

We have to find an element which is greater than zero in monotonically increasing function. This is nothing but finding first non-zero element in an increasing array. Using binary search, we can search for an element in sorted array in O (log n) time. Similarly, we can use binary search to find the first non-zero element in a monotonically increasing function.

Following is the pseudo code for binary search in a sorted array. If f (x) is negative, we will increment our lower bound to mid+1. Similarly, if f(x) is positive, we decrement the higher bound to mid. And then we call the findFirstPositive function recursively after updating low and high values. Base case is when low and high are equal. After reaching base case we return low because this will be the index of first positive number.

**Pseudo Code:**

def findFirstPositive (f, low, high):

if (low = high) return low

mid = (low + high) / 2

if (a[mid] <= 0):

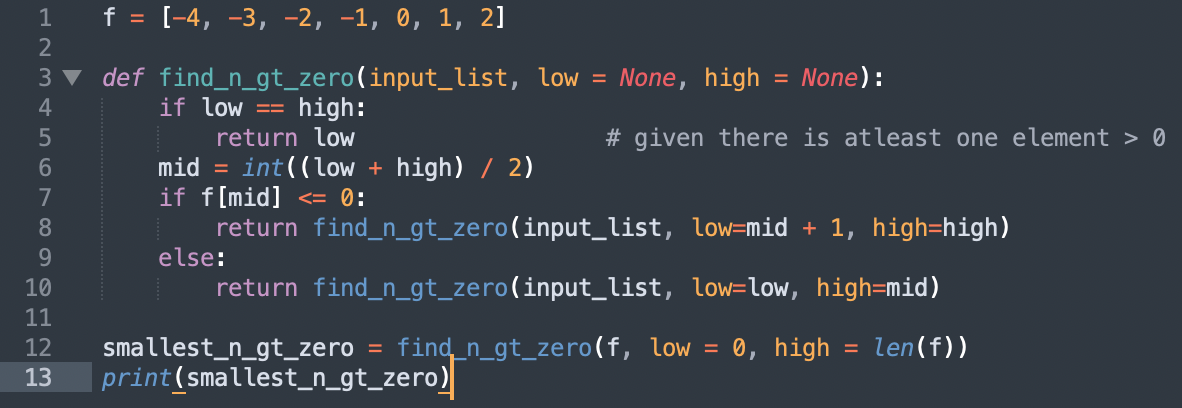
return findFirstPositive (f, mid+1, high)

else:

return findFirstPositive (f, low, mid)

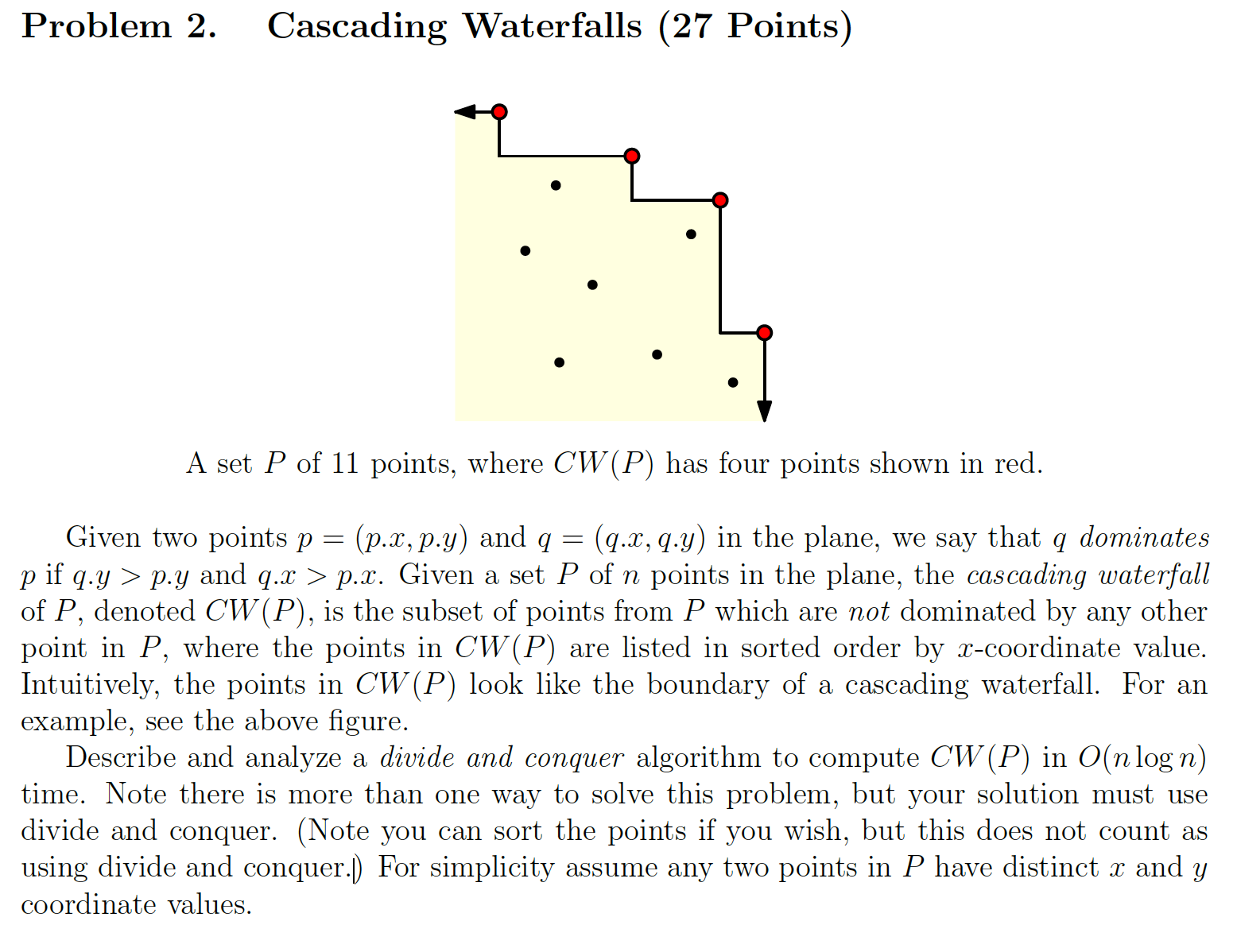
**Executable Code:**

We wrote the following code assuming that function is in the form of a sorted array.



**Time Complexity:**

Above algorithm is nothing but a binary search in sorted array. We know that binary search takes O (log(n)) time complexity. Thus, above algorithm will run in O (log(n)) time complexity.

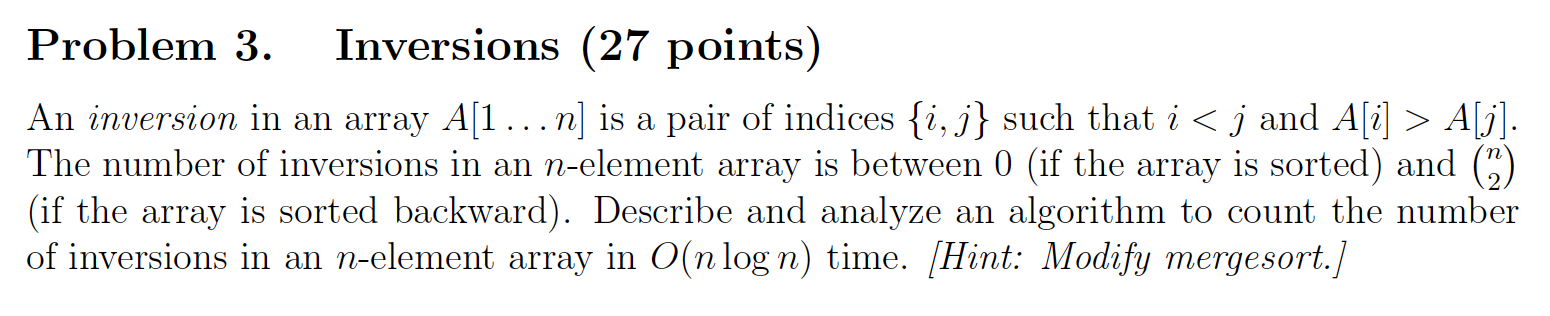


**Intuition:**

**Pseudo Code:**

**Executable Code:**

**Time Complexity:**



**Intuition:**

We can intuitively say that if array is sorted in ascending order, then number of inversions will be 0. We can just modify the merge sort algorithm to get number of inversions. This can be done by initializing a counter and this counter is incremented when we encounter smaller value in right subpart.

We can get the inversion count by summing up inversion counts in left half, right half and inversion counts across both halves. We can get inversion counts from left and right halves using divide and conquer. Now we only need to get the inversion count across. This is done when a smaller element is found in the right half. When smaller element is found in right half then we increment the counter by number of larger elements found in the left half. Then we get the final count by adding all the inversions in left, inversions in right and inversions across.

**Pseudo Code:**

def divideAndConquer(points, low = 0, high = points.length - 1):

if (low == high):

return 0

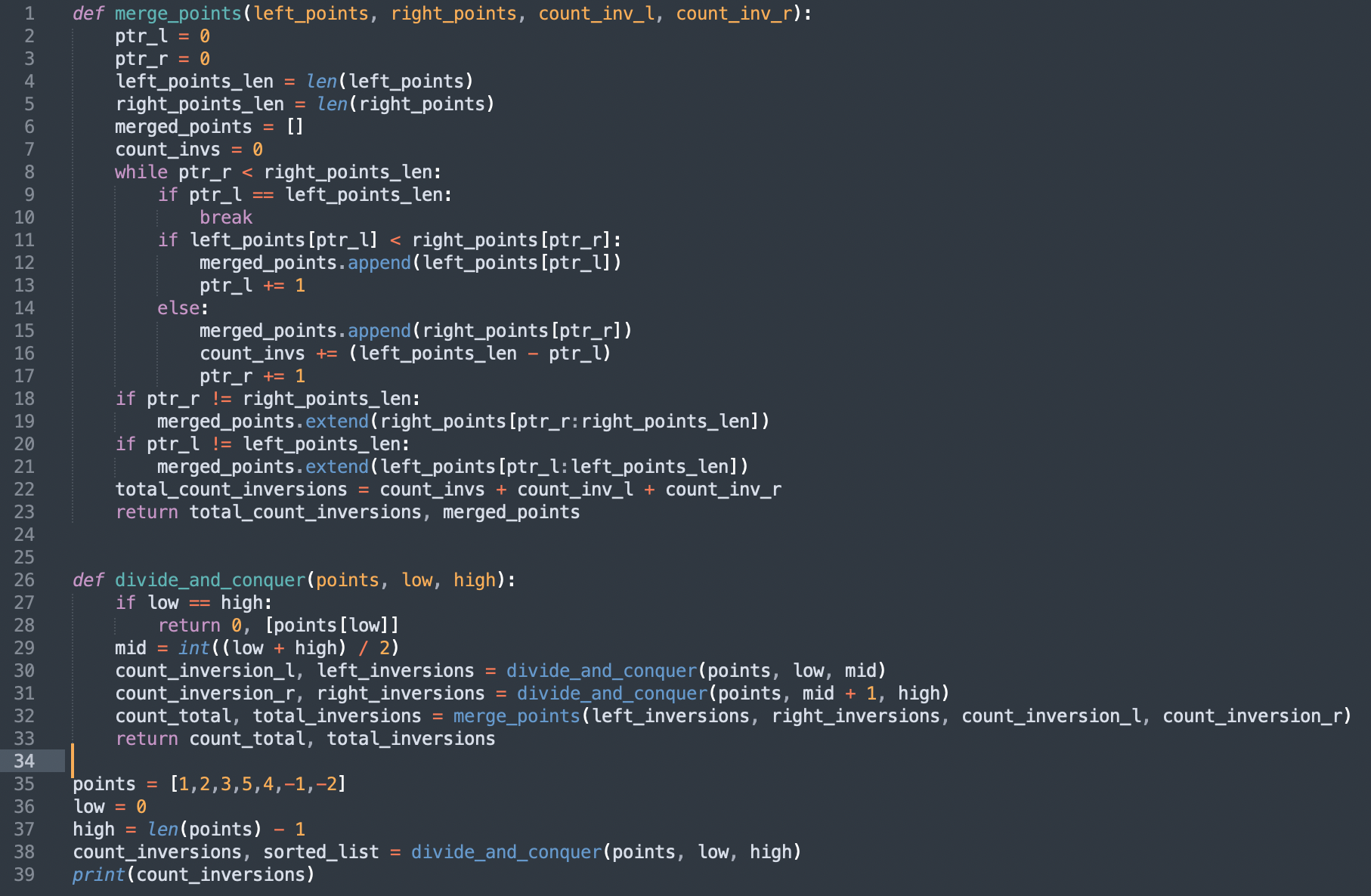
mid = (low + high) / 2

count\_left = divideAndConquer(points, low, mid)

count\_right = divideAndConquer(points, mid+1, high)

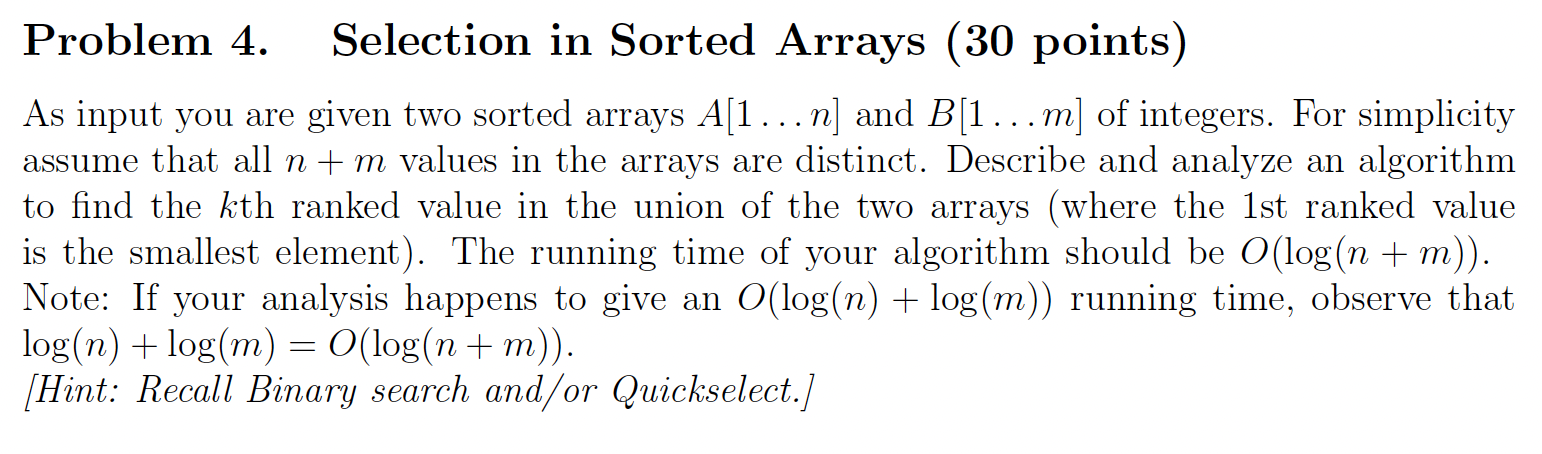
**Executable Code:**

We return inversion count and also sorted points from merge\_points function to keep track of the merged points as well.



**Time Complexity:**

Above algorithm runs in the same way as merge sort and so running time complexity will be same as that of merge sort. Thus, time complexity will be O (n log(n))



**Intuition:**

We have to find kth indexed element in the union of two given arrays. We can do this by merging both the arrays initially and then finding the kth element. But merging both the arrays will take O (n + m) time complexity. We have to solve this question in O (log (n + m)) time complexity.

Firstly, we take mid points of both the arrays (n/2 and m/2) and then if summation of these two mid indexes is less than k, then we can remove first half of the array which has lower mid value and update k to k – mid. Now we run the recursive main function after updating k and updating the array that has lower mid value.

If summation of both mid points is greater than k, then we remove higher half elements from Array that has higher mid value and run the recursive main function with same k and updating the array that has higher mid value.

Finally, whenever one of the arrays become null, we directly take the updated kth element from the array that is not null.

**Pseudo Code:**

def kthLargest(arr1, arr2, k):

if (arr1.length = 0): return arr2[k]

else if (arr2.length = 0): return arr1[k]

mid1 = arr1.length / 2

mid2 = arr2.length / 2

if (mid1 + mid2 < k):

if (arr[mid1] > arr[mid2]):

return kthLargest(arr1[], arr2[mid2+1:], k-mid2-1)

else: return kthLargest(arr1[mid1+1:], arr2[], k-mid1-1)

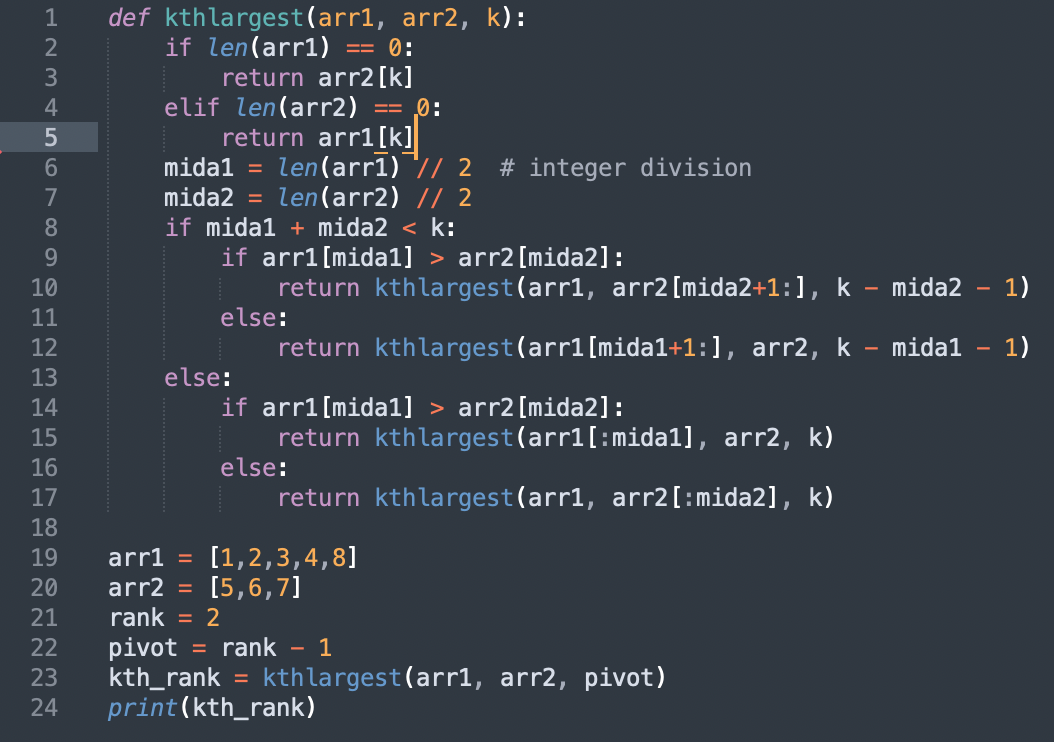
else:

if (arr[mid1] > arr[mid2]):

return kthLargest(arr1[:mid1], arr2[], k)

else return kthLargest(arr1[], arr2[:mid2], k)

**Executable Code:**



**Time Complexity:**

Time complexity will be T(n/2) + T(m/2), since worst case scenario we will iterate through each half of both arrays. So time complexity will be O (log(n) + log(m)) which is O (log(n + m))