**Algorithm 1:**

**Pseudo code:**

**find\_kth\_elem(arr1, s1, arr2, s2, k)**

// arr1 is an array with size s1

// arr2 is an array with size s2

// k is the position of the needed element (starting from 1)

{

mergedArr = array of size (s1 + s2)

merge(arr1, s1, arr2, s2, mergedArr)

return mergedArr[k - 1]

}

**merge(arr1, s1, arr2, s2, mergedArr)**

{

ind1 = 0, ind2 = 0, i = 0

while(ind1 < s1 && ind2 < s2){

if(arr1[ind1] <= arr2[ind2]) mergedArr[i++] = arr1[ind1++];

else mergedArr[i++] = arr2[ind2++];

}

while(ind1 < s1){

mergedArr[i++] = arr1[ind1++];

}

while(ind2 < s2){

mergedArr[i] = arr2[ind2];

}

}

**Time complexity:**

The time complexity of **the merge algorithm is O(s1 + s2)** which is **O(n)**.

So the overall time complexity of **the find\_kth\_elem algorithm** is n+c

, c is a constant, which is **O(n)** for all cases **best, average, and worst case.**

**Space complexity:**

**O(s1 + s2)** for defining new sorted array that combines the other two arrays.

**Algorithm 2:**

**Pseudo code:**

**find\_kth\_elem\_alg2(arr1, s1, arr2, s2, k){**

**if(s1 > s2) {**

**return find\_kth\_elem\_alg2(arr2, s2, arr1, s1, k);**

**}**

**if(!s1 && s2) return arr2[k - 1];**

**low = max(0,k-s2-1), high = min(k,s1);**

**return find\_kth\_elem\_alg2\_src(arr1, s1, arr2, s2, k, low, high);**

**}**

**find\_kth\_elem\_alg2\_src(arr1, s1, arr2, s2, k, l, h){**

**if(l <= h){**

**cut1 = (l + h) / 2;**

**cut2 = k - cut1 - 2;**

**l1 = arr1[cut1];**

**l2 = cut2 >= 0 ? arr2[cut2] : INT\_MIN;**

**r1 = cut1 + 1 < s1 ? arr1[cut1 + 1] : INT\_MAX;**

**r2 = cut2 + 1 < s2 ? arr2[cut2 + 1] : INT\_MAX;**

**if(l1 <= r2 && l2 <= r1) {**

**return max(l1, l2);**

**}**

**else if (l1 > r2)**

**return find\_kth\_elem\_alg2\_src(arr1, s1, arr2, s2, k, l, cut1 - 1);**

**else**

**return find\_kth\_elem\_alg2\_src(arr1, s1, arr2, s2, k, cut1 + 1, h);**

**}**

**return arr2[k - 1];**

**}**

**Time complexity:** The time complexity the algorithm is **O (log (min (s1, s2)))**

in worst and average case because it does the binary search on the smaller array size and **O (1)** in best case when one of the array’s sizes is **0.**

**Space complexity: O (1)**

|  |  |  |
| --- | --- | --- |
|  | **Algorithm 1** | **Algorithm 2** |
| **Time complexity** | Best---------|  Average -- O**(n)**  Worst ------| | Best **O (1)**  Average --- **O (log (min (s1, s2)))**  Worst ---------| |
| **Space complexity** | **O (s1 + s2)** | **O (1)** |