**Algorithm Implementation – Counting Sort:**

**Part 1 – High Level Algorithm:**

Suppose we had an unsorted array such that all elements in the array range between 0-K. For example, suppose we used K = 4, and we had the array {4, 3, 1, 0, and 2}. The following steps could be used to sort the array.

* Create a counts array. This array is of size (K + 1) and it has the indexes 0 – K. Each index value represents the count (the number of occurences) of that index in the unsorted array. The above array would yield the counts array [ 1, 1,1,1,1]. This indicates that there is one occurrence from every integer ( 0 - K =4). If there were two 4’s, then the counts array would be [ 1, 1,1,1,2]. To summarize, if index 4 has a two, that means there are two 4’s in the unsorted array.
* Intuitively, if we wanted to find the correct sorted position of the 3 for example (in the unsorted array which is at position 1), we would sum up all the elements in the counts array from index zero to (3-1). Since the counts array shows that there is only one 1 and one 2, it would makes sense that the three would be located at location (1 + 1) because it must be right after all the elements smaller than it. So, this means that;

*sorted position of unsorted element =*

* So now we can get the sorted position of any unsorted element by creating a counts array and using the formula above. To sort the array we could create a new array called “sorted array”. We can then place every item in the unsorted array in its correctly sorted position (since we can compute this). We have to create a new array because we don’t want to overwrite data that we are going to need later. If we were to not create a new array, we would be putting items in their correctly sorted position and then later on assuming they aren’t sorted and trying to sort them.
* Finally once we have a completely sorted array, and we are done using the unsorted array during the sort process, we can copy all the contents in the sorted array to the unsorted array to sort the unsorted array.

**Part 2 – Implementation withFlawed Get Sorted Position:**

|  |
| --- |
| **public** **class** CountingSort {  **public** **static** **final** **int** *K* = 4;  **public** **static** **int**[] getCountsArray(**int**[] array){  **int** counts[] = **new** **int**[*K* + 1];  **for** (**int** i = 0; i < array.length; i++){  counts[array[i]]++;  }  **return** counts;  }  **public** **static** **void** arrayCopy(**int**[] dest, **int**[] src){  **for**(**int** i = 0; i < dest.length; i++){  dest[i] = src[i];  }  }  **public** **static** **int**[] getSortedArray(**int**[] array, **int**[] counts){  **int**[] sortedArray = **new** **int**[array.length];  **for**(**int** i = 0; i < array.length; i++){  **int** sortedPosition = *getSortedPosition*(counts, array[i]);  sortedArray[sortedPosition] = array[i];  }  **return** sortedArray;  }  /\*\*  \* This method is buggy  \*/  **public** **static** **int** getSortedPosition(**int**[] counts, **int** unsortedElement){  **int** sortedPosition = 0;  **for**(**int** i = 0; i < unsortedElement; i++){  sortedPosition += counts[i];  }  **return** sortedPosition ;  }  **public** **static** **void** countsort(**int** array[]){  **int**[] counts = *getCountsArray*(array);  **int**[] sortedArray = *getSortedArray*(array, counts);  *arrayCopy*(array, sortedArray);  }  } |

**Part 2 – Problem and Solution for ‘Get Sorted Position’:**

**Problem:** The problem with our previous implementation is that if there are two duplicate integers, they will map to the exact same location. We want it to be such that if you spot the first occurrence, it gets mapped to an index, and then the second occurrence would get mapped to the next index and not just overwrite the previous index. For example, suppose we had the input array **int** array[] = {4,3,4,0,1,2}; This would give you the following output with the buggy sort function above: [0, 1, 2, 3, 4, 0]. The reason this is the output is because it would look at the first four and figure out the correctly sorted position is index 4 (which is correct). But then it would look at the second four and then find that the correctly sorted position is 4 (which it is usually but since this the second four we have seen, we need to move it one over to the right).

**Solution:** You could introduce an array called “remaining counts”. The counts would hold all the counts of elements and the remaining counts would hold all the remaining counts or the counts you have not placed in their correct sorted position in the sorted array yet. Each time you place an element in its correct sorted position in the sorted array, you update remaining counts. Now when you see a duplicate item, you know how many more of this item are coming so you know how much space to add.

|  |
| --- |
| **public** **class** CountingSort {  **public** **static** **final** **int** *K* = 5;  **public** **static** **int**[] getCountsArray(**int**[] array){  **int** counts[] = **new** **int**[*K* + 1];  **for** (**int** i = 0; i < array.length; i++){  counts[array[i]]++;  }  **return** counts;  }  **public** **static** **void** arrayCopy(**int**[] dest, **int**[] src){  **for**(**int** i = 0; i < dest.length; i++){  dest[i] = src[i];  }  }  **public** **static** **int**[] getSortedArray(**int**[] array, **int**[] counts, **int**[] remainingCounts){  **int**[] sortedArray = **new** **int**[array.length];  **for**(**int** i = 0; i < array.length; i++){  **int** sortedPosition = *getSortedPosition*(counts, array[i], remainingCounts);  sortedArray[sortedPosition] = array[i];  remainingCounts[array[i]]--;  }  **return** sortedArray;  }  **public** **static** **int** getSortedPosition(**int**[] counts, **int** unsortedElement, **int**[] remainingCounts){  **int** sortedPosition = 0;  **for**(**int** i = 0; i < unsortedElement; i++){  sortedPosition += counts[i];  }  **if**(*notLastOccurrence*(unsortedElement, remainingCounts)){  sortedPosition += *numOfRemainingOccurences*(unsortedElement, remainingCounts);  }  **return** sortedPosition ;  }    **public** **static** **boolean** notLastOccurrence(**int** unsortedElement, **int**[] counts){  **return** counts[unsortedElement] > 1;  }    **public** **static** **int** numOfRemainingOccurences(**int** unsortedElement, **int**[] counts){  **return** counts[unsortedElement] - 1;  }  **public** **static** **void** countsort(**int** array[]){  **int**[] counts = *getCountsArray*(array);  **int**[] remainingCounts = *getCountsArray*(array);  **int**[] sortedArray = *getSortedArray*(array, counts, remainingCounts);  *arrayCopy*(array, sortedArray);  }  } |

**Important Note:** Suppose you have the original array **int** array[] = {4,3,0,0,1,2};. When we calculate the correct sorted position of the first zero, it would be 2 but then when we get into the if statement we would add the number of remaining zeroes (which would be 1 more zero) to get 3. Then the other zero would get a sorted position of 2. If you notice, the orignal array with contents {4,3,01, 02, 1, 2} would be sorted to {4,3,02, 01, 1, 2}. Keep in mind that duplicate items will be placed in there reverse order they appeared in the unsorted array.