**Algorithm Implementation – Counting Sort**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Part 1 - High Level Algorithm Design:**  **public** **class** CountingSort{  **public** **static** **final** **int** *K* = 4;  **public** **static** **final** **int** *N* = 10;  **public** **static** **void** countsort(**int**[] array){}    **public** **static** **int**[] getCounts(**int**[] array){}    **public** **static** **void** insert(**int**[] array, **int** index, **int** insertItem, **int** amount){}    **public** **static** **void** useCountsToSortArray(**int**[] array, **int**[] counts){}  }   |  |  | | --- | --- | | **Variable** | **Description** | | K | This variable indicates the upper limit of the range the integer values can be in the array to be sorted. For example, if K = 9, than count sort will assume that the integers in the array to be sorted range between 0 – 9. | | N | This variable indicates the number of elements in the array that is to be sorted. |  |  |  | | --- | --- | | **Method** | **Description** | | getCounts | Given an array of integers that range between 0 – k, this method returns an array of the counts of each integer. The index represents the integer and the value represents the count. Since the integers are between 0-K, the size of the counts array is (k+1) with indices 0-k. For example, the array [0,1,2,3,4] with n = 6 and k = 5 would return a counts array with values [1,1,1,1,1,0] since each possible value has one count except for 5 which has zero. The sum of the counts array should be equal to N. | | append | Given an array, an index, an insert item and an amount, this method will insert the insert item ‘amount’ times in the array at index ‘index’. For example, if you have the array [1,2,0,0] and you pass it with insert item 9, index 2 and amount 2, it will put two 9’s starting at index 2. This will yield the array [1,2,9,9]. | | useCountsToSortArray | Given a counts array, and the original unsorted array, this method will sort the original unsorted array. | |

|  |
| --- |
| **Part 2 – Use Counts To Sort Array Explanation:**  Suppose you are given an integer array of size 7 (n = 7) and the integers range between 0 and 3 (k = 3). Suppose we were given the counts array of this array and it contains values [0,2,1,4]. What this counts array tells us is that the originally array has no 0’s, two 1’s, a single 2 and 4 3’s. Remember, the sum of the counts array should be equal to n. Since we know the counts, we can use this information to deduce that the sorted array is [1,1,2,3,3,3,3]. The way we figured this out is we iterated over the entire counts array. For every time item in the array, we insert this to the unsorted array ‘amount’ times where amount is the exact count of that integer. |

|  |
| --- |
| **Part 3 – Algorithm Implementation:**  **public** **class** CountingSort{  **public** **static** **final** **int** *K* = 5;  **public** **static** **final** **int** *N* = 10;  **public** **static** **void** countsort(**int**[] array){  **int**[] counts = *getCounts*(array);  *useCountsToSortArray*(array, counts);  }    **public** **static** **int**[] getCounts(**int**[] array){  **int** counts[] = **new** **int**[*K*];  **for** (**int** item : array){  counts[item]++;  }    **return** counts;  }    **public** **static** **void** insert(**int**[] array, **int** index, **int** insertItem, **int** amount){  **for**(**int** i = 0; i < amount; i++){  array[index] = insertItem;  index++;  }  }    **public** **static** **void** useCountsToSortArray(**int**[] array, **int**[] counts){  **int** sortedIterator = 0;  **for**(**int** i = 0; i < counts.length; i++){  *insert*(array, sortedIterator, i, counts[i]);  sortedIterator += counts[i];  }  }  } |

**Part 4 – Complexity Analysis:**

**Time Complexity Analysis:**

* The “count sort” function calls “get counts” and “use counts to sort array”.
* “Get counts” iterates over an array of size k+1. It’s time complexity is O(k + 1) = O(k).
* “use counts to sort array” iterates over the “counts” array which is of size (k + 1). Then, in the insert function, it does the insert steps ‘amount’ times and amount is the count of that item. So the complexity is . Since we can factor out this number of constant steps (C1)then the function simplifies to . Remember, the summation of the counts array is equal to n. This means that the complexity of this function is O(C1×n)
* This means that “count sort” has a time complexity of O((k+1) + (C1×n)) = O(n + k).

**Space Complexity Analysis:**

* You only introduce a counts array. This array has a size of K + 1. This means that the space complexity is O(k+1) = O(k).

|  |  |  |  |
| --- | --- | --- | --- |
| **Best Time Complexity** | **Average Time Complexity** | **Worst Time Complexity** | **Space Complexity** |
| *O(n + k)* | *O(n + k)* | *O(n + k)* | *O(k)* |