**Counting Sort**

Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

It is a stable sorting technique, which is used to sort objects according to the keys that are small numbers. It counts the number of keys whose key values are same. This sorting technique is effective when the difference between different keys is not great, otherwise, it can increase the space complexity.

This algorithm is not comparison-based but is key-based done through some type of hashing.

**Conditions for Counting Sort**:

* Must be a nonnegative integer (original code)
* Must know the range of the values
* Linear complexity only

Originally, counting sort is an algorithm for sorting a collection of objects according to keys that are small positive integers. Although radix sorting itself dates back far longer, counting sort, and its application to radix sorting, were both invented by Harold H. Seward in 1954. It operates by counting the number of objects that possess distinct key values, and applying prefix sum on those counts to determine the positions of each key value in the output sequence. Its running time is linear in the number of items and the difference between the maximum key value and the minimum key value, so it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items. It is often used as a subroutine in radix sort, another sorting algorithm, which can handle larger keys more efficiently.

Bucket sort may be used in lieu of counting sort, and entails a similar time analysis. However, compared to counting sort, bucket sort requires linked lists, dynamic arrays, or a large amount of pre-allocated memory to hold the sets of items within each bucket, whereas counting sort stores a single number (the count of items) per bucket.

**Input and Output Assumptions**

In the most general case, the input to counting sort consists of a collection of n items, each of which has a non-negative integer key whose maximum value is at most k.

The output is an array of the elements ordered by their keys. Because of its application to radix sorting, counting sort must be a stable sort; that is, if two elements share the same key, their relative order in the output array and their relative order in the input array should match.

**Algorithm**

**function** CountingSort(input, *k*)

count ← array of *k* + 1 zeros

output ← array of same length as input

**for** *i* = 0 **to** length(input) - 1 **do**

*j* = key(input[*i*])

count[*j*] += 1

**for** *i* = 1 **to** *k* **do**

count[*i*] += count[*i* - 1]

**for** *i* = length(input) - 1 **downto** 0 **do**

*j* = key(input[*i*])

count[*j*] -= 1

output[count[*j*]] = input[*i*]

**return** output

countingSort(array, size)

max <- find largest element in array

initialize count array with all zeros

for j <- 0 to size

find the total count of each unique element and

store the count at jth index in count array

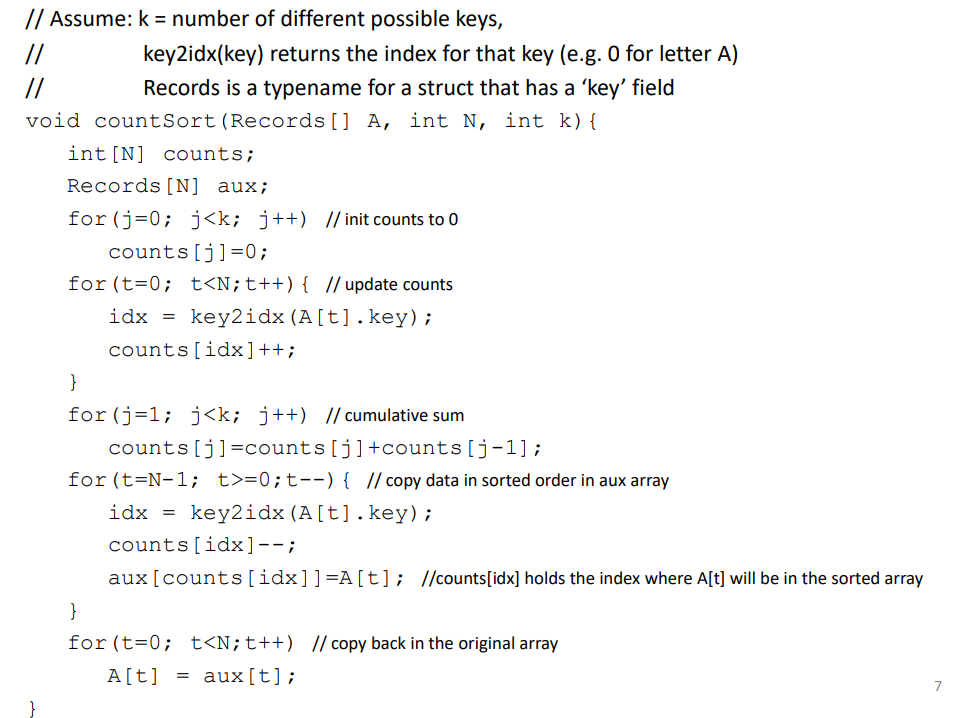
for i <- 1 to max

find the cumulative sum and store it in count array itself

for j <- size down to 1

restore the elements to array

decrease count of each element restored by 1



**Variation**

If each item to be sorted is itself an integer, and used as key as well, then the second and third loops of counting sort can be combined; in the second loop, instead of computing the position where items with key i should be placed in the output, simply append Count[i] copies of the number i to the output.

This algorithm may also be used to eliminate duplicate keys, by replacing the Count array with a bit vector that stores a one for a key that is present in the input and a zero for a key that is not present. If additionally, the items are the integer keys themselves, both second and third loops can be omitted entirely and the bit vector will itself serve as output, representing the values as offsets of the non-zero entries, added to the range's lowest value. Thus, the keys are sorted and the duplicates are eliminated in this variant just by being placed into the bit array.

