**Counting Sort**

# Counting Sort

Counting sort is an algorithm for sorting a collection of objects according to keys that are small integers; that is, it is an integer sorting algorithm.

It operates by counting the number of objects that have each distinct key value, and using arithmetic 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 and minimum key values, so it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items. However, it is often used as a subroutine in another sorting algorithm, radix sort, that can handle larger keys more efficiently.

It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

# Algorithm

The algorithm loops over the items, computing a histogram of the number of times each key occurs within the input collection.

It then performs a prefix sum computation (a second loop, over the range of possible keys) to determine, for each key, the starting position in the output array of the items having that key.

Finally, it loops over the items again, moving each item into its sorted position in the output array.

for x in input:

count[key(x)] += 1

# calculate the starting index for each key:

total = 0

for i in range(k): # i = 0, 1, ... k-1

oldCount = count[i]

count[i] = total

total += oldCount

# copy to output array, preserving order of inputs with equal keys:

for x in input:

output[count[key(x)]] = x

count[key(x)] += 1

return output

After the first for loop, count[i] stores the number of items with key equal to i.

After the second for loop, it instead stores the number of items with key less than i.

Throughout the third loop, count[i] always stores the next position in the output array into which an item with key i should be stored.

The relative order of items with equal keys is preserved here; i.e., this is a stable sort.

Count sort animation:

<https://www.cs.usfca.edu/~galles/visualization/CountingSort.html>

# Implementation in C

#include <stdio.h>

#include <string.h>

#define ERROR -1

#define N 100

#define MAX\_RANGE\_VAL 1000

int countingSort(int \*arr\_arg, int n, int max\_val);

int main(void) {

int arr[N] = {0};

int n = 0;

int max = 0;

int i = 0;

int ret\_val = 0;

fprintf(stdout, "Enter no of elements : ");

fscanf(stdin, "%d", &n);

memset(arr, 0, N);

fprintf(stdout, "Enter all elements : ");

for(i = 0; i < n; i++) {

arr[i] = 0;

fscanf(stdin, "%d", &arr[i]);

if(0 == i) {

max = arr[0];

}

if(max < arr[i]) {

max = arr[i];

}

}

if(n > 1) {

ret\_val = countingSort(arr, n, max);

}

fprintf(stdout, "\nAfter sorting \n");

for(i = 0; i < n; i++) {

fprintf(stdout, "%d ", arr[i]);

}

return 0;

}

int countingSort(int \*arr\_arg, int n, int max\_val)

{

int countArr[MAX\_RANGE\_VAL + 1] = {0};

int sortArr[N] = {0};

int i = 0;

memset(countArr, 0, MAX\_RANGE\_VAL);

memset(sortArr, 0, N);

// store frequency of each array element

for(i = 0; i < n; i++) {

countArr[arr\_arg[i]] = countArr[arr\_arg[i]] + 1;

}

// update countArr to store actual position of elements in array.

for(i = 1; i <= max\_val; i++) {

countArr[i] = countArr[i] + countArr[i - 1];

}

for(i = (n - 1); i >= 0; i--) {

countArr[arr\_arg[i]] = countArr[arr\_arg[i]] - 1;

sortArr[countArr[arr\_arg[i]]] = arr\_arg[i];

}

for(i = 0; i < n; i++) {

arr\_arg[i] = sortArr[i];

//fprintf(stdout, "%d ", sortArr[i]);

}

return 0;

}

Output:

Enter no of elements : 10

Enter all elements : 2 3 2 4 5 6 5 4 9 8

After sorting

2 2 3 4 4 5 5 6 8 9

# Complexity Analysis

Time Complexity: O(n+k) where n is the number of elements in input array and k is the range of input.

Auxiliary Space: O(n+k)

Points to be noted:

1. Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted. Consider the situation where the input sequence is between range 1 to 10K and the data is 10, 5, 10K, 5K.
2. It is not a comparison based sorting. It running time complexity is O(n) with space proportional to the range of data.
3. It is often used as a sub-routine to another sorting algorithm like radix sort.
4. Counting sort uses a partial hashing to count the occurrence of the data object in O(1).
5. Counting sort can be extended to work for negative inputs also.

# References

<https://www.geeksforgeeks.org/sorting-algorithms/>

<https://www.geeksforgeeks.org/counting-sort/>