Counting Sort Algorithm

Sorting algorithms are a must-know for any software engineer preparing for a technical interview. They will help you crack many questions during your coding round.

In this article, we'll give you a refresher on the counting algorithm:

- What Is Counting Sort?
- How Does Counting Sort Work?
- Counting Sort Algorithm
- Counting Sort Pseudocode
- Counting Sort Code
- Counting Sort Complexities
- Advantages of Counting Sort
- Disadvantages of Counting Sort
- FAQs on Counting Sort

What Is Counting Sort?

Counting sort is an algorithm used to sort the elements of an array by counting and storing the frequency of each distinct element in an auxiliary array. Sorting is done by mapping the value of each element as an index of the auxiliary array.

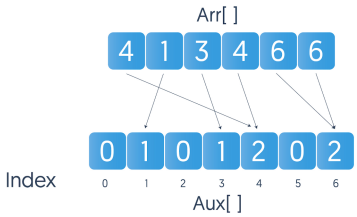
Counting sort is handy while sorting values whose range is not very large.

How Does Counting Sort Work?

Let's assume that array *Arr[]* = {4, 1, 3, 4, 6, 6} with the maximum element M (here it is 6) is to be sorted.

First, we take an auxiliary array *Aux[]* of size *M+1* (here it is 6+1 = 7) and initialize it with 0.

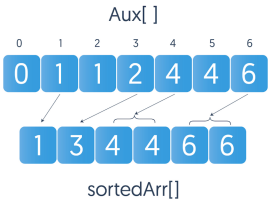
Next, we store the frequency of each unique element of array *Arr[]* in *Aux[]*.



Then, we calculate the prefix sum at every index of Aux[] array.

We then create an array *sortedArr[]* of size equal to the Arr[] array.

Next, traverse array Arr[] from right to left and update sortedArr[] as
sortedArr[Aux[Arr[i]] - 1] = Arr[i] and Aux[] as Aux[Arr[i]]--. The reason behind traversing Arr[] from right to left is to preserve the order of equal elements, which eventually makes this sorting algorithm stable.



Counting Sort Algorithm

Consider an array Arr[] of size N that we want to sort:

Step 1: Declare an auxiliary array *Aux[]* of size *max(Arr[])+1* and initialize it with 0.

Step 3: Calculate the prefix sum at every index of array Arr[].

Step 4: Create an array *sortedArr[]* of size *N*.

Step 5: Traverse array *Arr[]* from right to left and update sortedArr[] as
sortedArr[Aux[Arr[i]] - 1] - Arr[i]. Also, update Aux[] as Aux[Arr[i]]--.

As we are using the index of an array to map elements of array *Arr[],* if the difference between the maximum element and minimum element is huge, counting sort will not be feasible.

Counting sort is helpful when we have to sort many numbers that lie in a comparatively small range.

For example, take an array *Arr[]* = {1000000005,1001000000,1000000007}

Here, maximum element of *Arr[]* is 1001000000

Can we apply the counting sort algorithm to this type of array?

Yes, we can! Here's how:

- Create a variable *MIN* to store *min(Arr[]),* and subtract it from every element of *Arr[]*.
- Now, every element of the array will be in the range [0,1000000].
- Apply Step1,2,3, and 4 of the above algorithm.
- Traverse array *Arr[]* from right to left, and update sortedArr[] as sortedArr[Aux[Arr[i]] - 1] = Arr[i] + MIN. Also, update Aux[] as Aux[Arr[i]]--.

Counting Sort Pseudocode

Assumptions for input:

- Every value is non-negative
- The range of input should not be very large

Arr[] = array to be sorted

Aux[] = auxiliary array for mapping

sortedArr[] = sorted version of Arr[]

M = maximum element of Arr[]

N = size of Arr[]

```
// initialising array Aux[] with 0
for i = 0 to M + 1 do
    Aux[ i ] = 0

// Storing count of each element
// in array Aux[]
for i in Arr[] do
    Aux[ Arr[ i ] ] ++

for i from N-1 to 0 in Arr[] do
    sortedArr[ Aux[ Arr[i] ] - 1] = Arr[i]
    Aux[ Arr[i] ] --
```

```
return sortedArr[]
```

Counting Sort Code

We've used C++ to demonstrate how counting sort works. Use this as a reference to code in C, Java, Python, or any programming language of your choice.

```
#include <bits/stdc++.h>

using namespace std;

int main()

{

    int N = 8;

    int Arr[N] = {4, 3, 12, 1, 5, 5, 3, 9};



    // Finding the maximum element of array Arr[].

    int M = 0;

    for(int i=0;i<N;i++)

        M=max(M,Arr[i]);



    // Initializing Aux[] with 0

    int Aux[M+1];

    for(int i=0;i<=M;i++)

        Aux[i]=0;



    // Mapping each element of Arr[] as an index
```

```
    // of Aux[] array
    for(int i=0;i<N;i++)
        Aux[Arr[i]]++;
    // Calculating prefix sum at every index
    // of array Aux[]
    for(int i=1;i<=M;i++)
        Aux[i]+=Aux[i-1];

    // Creating sortedArr[] from Aux[] array
    int sortedArr[N];
    for(int i=N-1;i>=0;i--)
    {
        sortedArr[Aux[Arr[i]] - 1]=Arr[i];
        Aux[Arr[i]]--;
    }

    for(int i=0;i<N;i++)
        cout<<sortedArr[i]<<" ";

    return 0;
}
```

Output Example

Consider *Arr[]* = {4, 3, 12, 1, 5, 5, 3, 9}

Here, *M* = 12, so we will make an *Aux[]* array of size 13.

After mapping all elements of array *Arr[]* as the index of *Aux[]* array:

Aux[] = {0, 1, 0, 2, 1, 2, 0, 0, 0, 1, 0, 0, 1}

Calculate prefix sum of Aux[] array:

Aux[] = {0, 1, 1, 3, 4, 6, 6, 6, 6, 7, 7, 7, 8}

Now, traverse in array Arr[] from right to left and place each Arr[i] at its correct place in sortedArr[] with the help Aux[] array:

Finally, *sortedArr[]* = {1, 3, 3, 4, 5, 5, 9, 12}

Counting Sort Complexities

Time Complexity

In this algorithm:

- Finding *M* (maximum element of array *Arr[]*) and mapping elements of array *Arr[]* takes O(N) time
- Initializing *Aux[]* array takes *O(M)* time
- Making *sortedArr[]* takes *O(N+M)* time

Therefore, the overall time complexity is *O(N+M).*

- Worst-case time complexity: **O(N+M).**
- Average-case time complexity: **O(N+M).**
- Best-case time complexity: **O(N+M).**

Space Complexity

In this algorithm:

- Aux[] array takes O(M) space
- sortedArr takes O(N) space

Therefore, the space complexity is O(N+M).

Advantages of Counting Sort

- Counting sort generally performs faster than all comparison-based sorting algorithms, such as merge sort and quicksort, if the range of input is of the order of the number of input
- Counting sort is easy to code

Disadvantages of Counting Sort

- Counting sort doesn't work on decimal values
- Counting sort is inefficient if the range of values to be sorted is very large

FAQs on Counting Sort

Question 1: Give an example of a non-comparison-based sorting algorithm. Is it faster than comparison-based sorting algorithms?

Answer:

- Counting sort is an example of a non-comparison-based sorting algorithm — it sorts by mapping the value of each array element as an index of the auxiliary array.

- Yes, counting sort generally runs faster than all comparison-based sorting algorithms, such as *quicksort* and *merge sort,* provided:
range of input is equal to or less than the order of the number of inputs
- If the range of input is very large compared to the order of the number of inputs, then counting sort will be less efficient than quicksort or merge sort.

Question 2: Where should counting sort be used?

Answer: Counting sort works better than the other comparison-based sorting algorithms when we have to sort many numbers that lie in a comparatively smaller range on the number line.
Counting sort has a time complexity of O(N+M), where M is max(arr[])-min(arr[]) and N is equal to size(arr[]). Comparison-based sorting algorithms take O(N log N) time.

When (N+M) << N log N, we can definitely use counting sort, given that we can feasibly allocate memory of O(N+M). On the other hand, if N log N << (N + M), we should use a comparison-based sorting algorithm.

Question 3: Is counting sort stable?

Answer: Yes, counting sort is an example of a stable sorting algorithm, as it does not change the relative order of elements of the same value in the input.

Question 4. Is counting sort an in-place sorting algorithm?

Answer: No, counting sort is not an in-place sorting algorithm. In in-place sorting algorithms, only a small/constant auxiliary space is used; in counting sort, we use an auxiliary array of the size of the range of data.

Are You Ready to Nail Your Next Coding Interview?

Sorting algorithms interview questions feature in almost every coding interview for software developers. If you're looking for guidance and help to nail these questions and more, sign up for our free webinar.

As pioneers in the field of technical interview prep, we have trained thousands of software engineers to crack the toughest coding interviews and land jobs at their dream companies, such as Google, Facebook, Apple, Netflix, Amazon, and more!

Sign up now!

Article contributed by Abhinav Tiwari

Recommended Posts

Iterating ArrayLists in Java

How to Sort a Dataframe in R

JavaScript - Adding a Class Name to the Element

All Posts