Counting Sort

Till now, all the sorting algorithms we have learned were comparison sort i.e., they compare the values of the elements to sort them but the counting sort is a non-comparison sort. It means that it sorts the input without comparing the values of the elements.

Comparison sorts have a lower bound of $\Omega(n \lg n)$ i.e., they can't perform better than this. And this is where non-comparison sorts get interesting because they can beat this lower bound and can perform faster like in linear time ($\Theta(n)$). This is because non-comparison sorts are generally implemented with few restrictions like counting sort has a restriction on its input which we are going to study further.

So, the restriction of the counting sort is that the input should only contain integers and they should lie in the range of 0 to k, for some integer k. For example, if the value of k is 10, then all the inputs must be between 0 to 10.

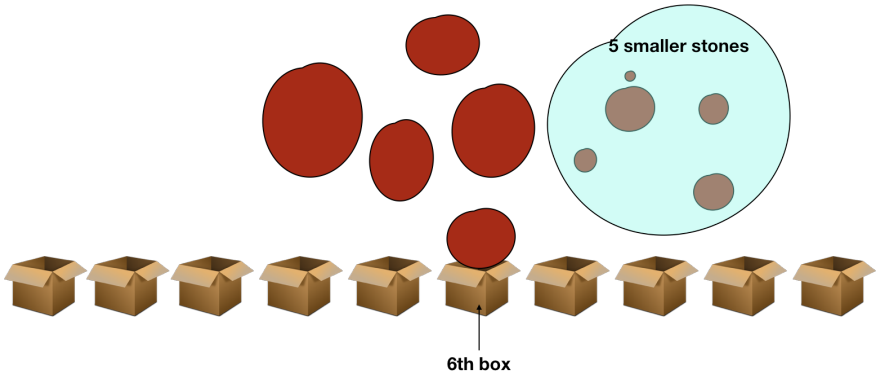
Also, we don't want the value of k to be too high because that will increase the running time of the counting sort. And we will see that the counting sort takes $\Theta(n)$ time when k is $O(n)$.

So, let's see the working of the counting sort.

Idea Behind Counting Sort

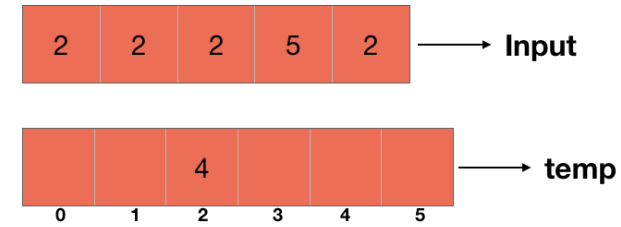
Suppose, there are 10 boxes and 10 stones. Now, you have to keep all these 10 stones in 10 boxes but in a sorted order i.e., the smallest stone will be in the first box and the largest stone will be in the 10^{th}. Of course, there are many different ways to do this but let's talk about one particular way which is related to the counting sort.

Suppose, you picked up a stone and you know that there are exactly 5 stones which are smaller than this stone, then without giving a single thought, you would place it in the 6^{th} box. This is the exact idea behind the counting sort.



Thus in the counting sort, we need an extra array to store the output like the boxes in the previous example. This array is going to store all the numbers which are in the input, so the size of this array should be n.

As the name suggests, we start by counting the number of times a number is in the input array. For example, if 2 is appearing 4 times in the input array then we count this number and store it in a different temporary array. And the way we are going to store this number in this temporary array is by changing the value of that element of the array whose index is equal to the number itself to a value number of times it is appearing. For example, if 2 is appearing 4 times, then the element with index 2 of the temporary array will have a value of 4. This is also illustrated in the picture given below.



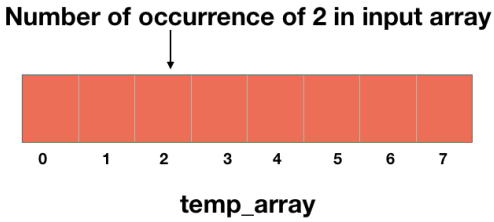
2 is appearing 4 times in input

So, if the range of the number in the input is large i.e., if the value of k is large, then the size of this temporary array will also be large. For example, if there are numbers from 0 to 10,000 in the input array, then we are going to need a temporary array at least of size 10,001 (range is starting from 0) to store how many times 10,000 is appearing in the input array. So, the size of this input array is $O(k)$.

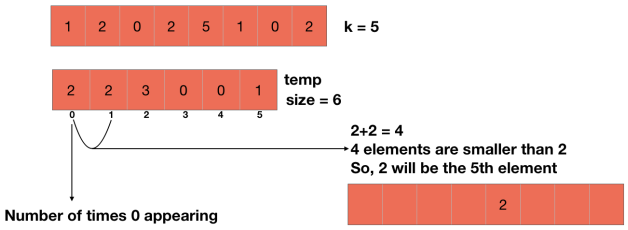
This is also one of the reasons why we want the value of k to be small because a large value of k would require a temporary array of large size.

With the help of this temporary array, we know how many elements are smaller than a particular element and then we can use the logic of the stones and the boxes to fill up the numbers in sorted order to the new output array.

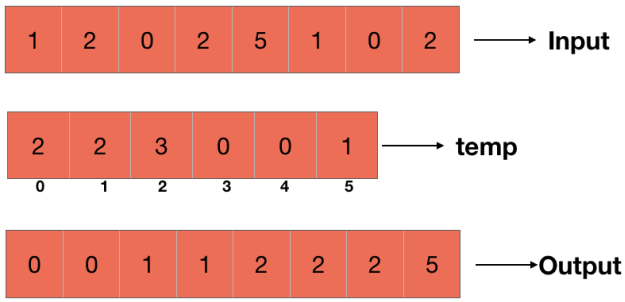
We start by making the temporary array.



Then, we fill the values of the temporary array with the help of the input array i.e., by counting the number of times an element is appearing in the input array.



Lastly, we fill our output array using the temporary array.

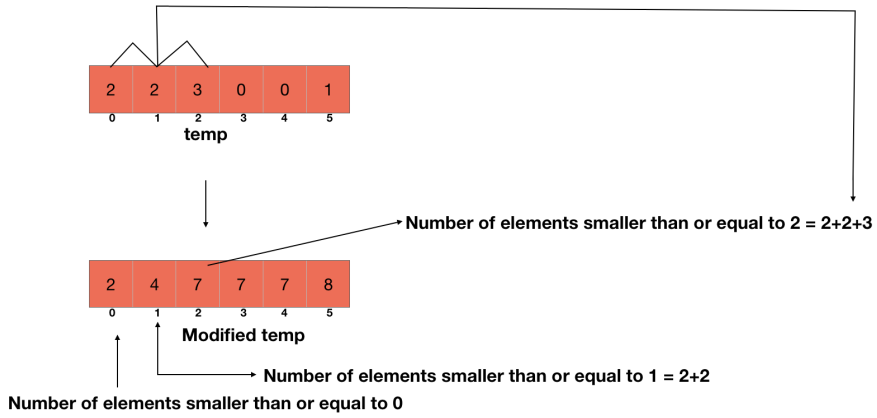


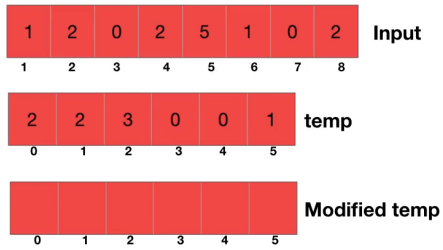
Now, let's see the proper working of the counting sort and how to properly implement it.

Working of Counting Sort

So, we start with a temporary of size k+1 and then we iterate over the input array and store the number of times an element is appearing in the input array to the temporary array.

As stated above, we need the count of the numbers smaller than a particular number to do the counting sort, so now we modify the temporary array to store the number of elements which are smaller than a particular element. This is shown in the picture given below.





Now, we know the number of elements smaller than a particular element, thus the last task is to fill up the output array.

Thus, we require arrays of sizes n (output array) and k (temporary array) for the process of counting sort. So, the counting sort has a space complexity of $O(n+k)$. From here, you can also see that reducing the value of k is beneficial to us.

After understanding the logic and the working of counting sort, let's write a code for it.

Code for Counting Sort

Our first task is to make a temporary array of size k+1 and make all its elements 0 i.e.,

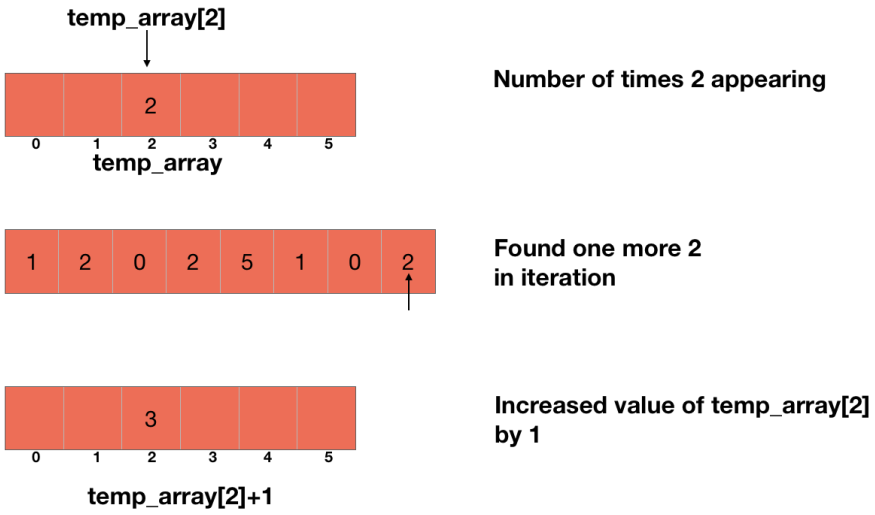
`temp_array[k+1]` → Initializing a temporary array

`for i in 0 to k` → Iterating over the array

`temp_array[i] = 0` → Making each element 0

Now, our next task is to store the number of occurrence of an element to the temporary array. So if an element is 2, then we have to store its count of occurrence to `temp_array[2]`

Thus, for any element `A[i]`, we are going to store its occurrence to `temp_array[A[i]]`. So every time we find an element `A[i]`, we will increase the count of `temp_array[A[i]]` by 1. For example, every time we find 2, we will increase `temp_array[2]` by 1.




```
for i in 1 to A.length
   temp[A[i]] = temp[A[i]] + 1
```

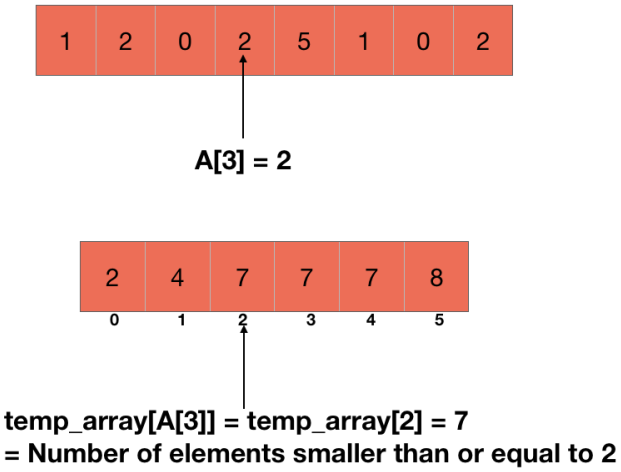
Now, the next task is to modify the elements of the temporary array and store the number of elements smaller than or equal to a particular element. This is simple, we just need to add all the elements of the temporary array before that particular element.

So, we can start iterating over the temporary array from the index 1 and store the sum as shown in the above picture.

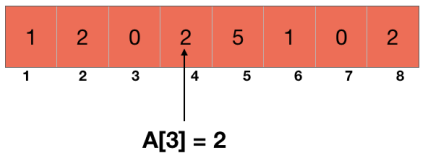
```
for i in 1 to k
   temp_array[i] = temp_array[i]+ temp_array[i-1]
```

Now, we are left with the final task of making the output array. For this, we need to iterate over the input array and for each element of the input array, we need to find the correct position where it should be in the output array.

So, let's say we are on an element `A[i]` of the input array, then `temp_array[A[i]]` stores the number of elements less than or equal to the number `A[i]`.



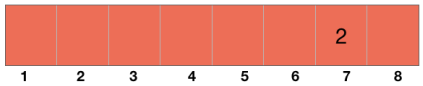
Now in the output array, `A[i]` will go to the index `temp_array[A[i]]` i.e., `output_array[temp[A[i]]] = A[i]`.



A[3] = 2

temp_array[A[3]] = temp_array[2] = 7
= Number of elements smaller than or equal to 2

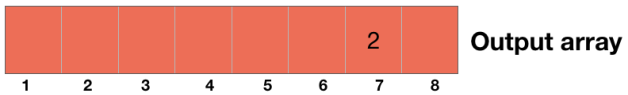
=> 2 will go at 7th index in output array



Output array

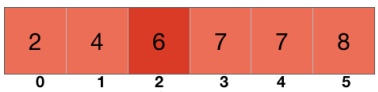
output_array[temp_array[A[3]]] = output_array[7] =2

Since `A[i]` is in the correct place in the output array, we need to decrease the count of it in the temporary array i.e., `temp_array[A[i]] = temp_array[A[i]] - 1`, so the places don't coincide in case of the duplicate numbers.



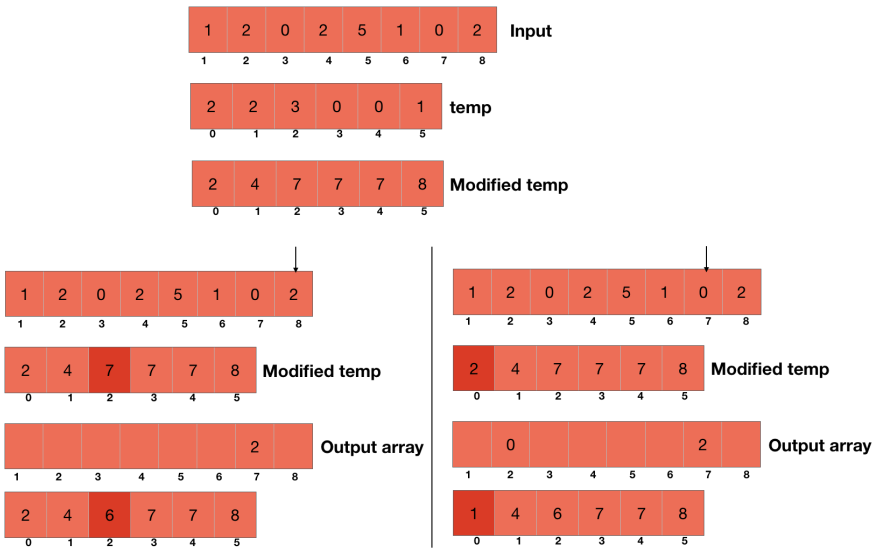
Output array

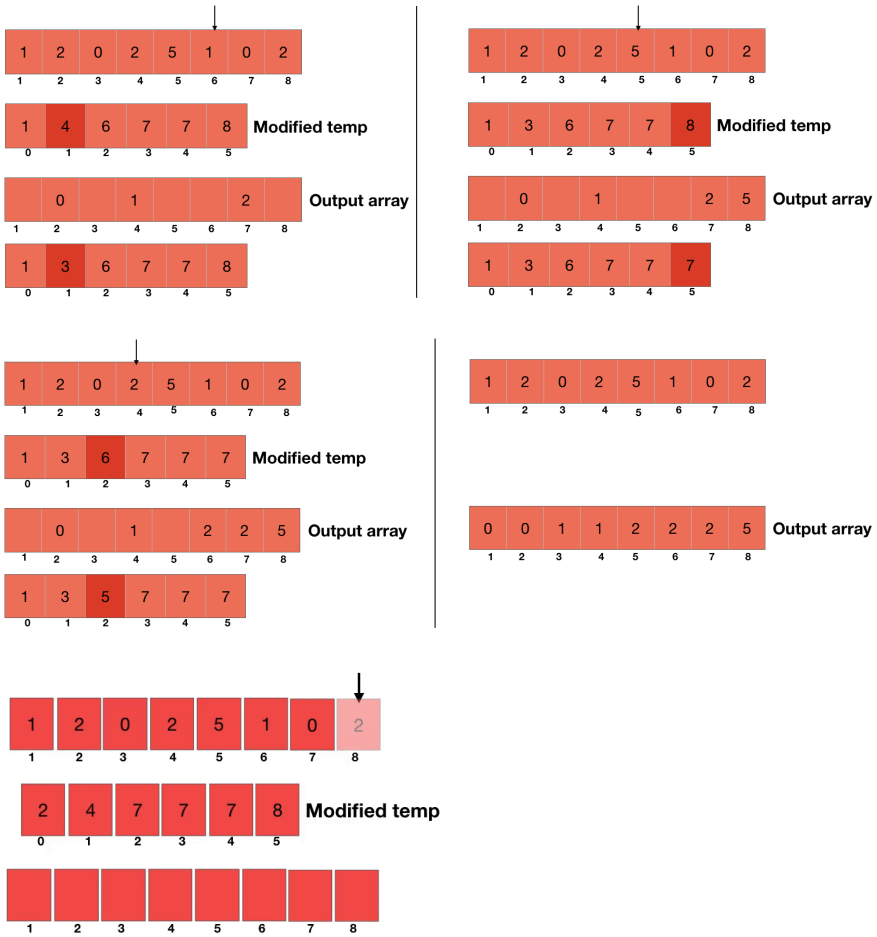
2 is at correct position



6 more elements less than or equal to 2

You can look at the pictures given below for the pictorial representaion of the counting sort.

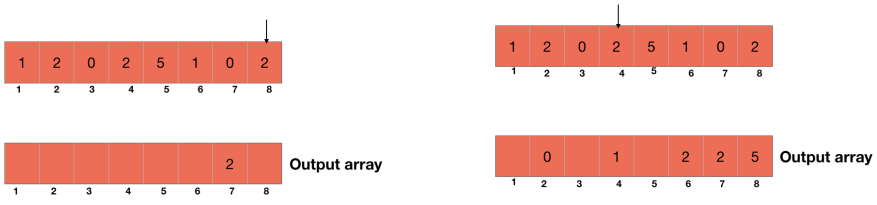




</>

Stability is also a property of an algorithm. Stability is when the numbers with same values appear in the same sequence in both input and the output. For example, if there are 4 twos in the input then the 2 which appeared first in the input will also appear first in the output and the 2 which appeared second in the input will also appear second in the output and so on.
Counting sort is a stable algorithm

As stated above, we are going to iterate to each element of the input array to find its correct position in the output array. Now, we have the option to make this iteration forward or backward. But take a case of iterating backward on the input array to fill up the output array. Here, among the common elements, the element which is at the last in the input array will be filled at last in the output array. This is shown in the picture given below. In this case, the counting sort will be a stable algorithm but this won't be possible if we iterate forward.



Last 2 filled first i.e.,
at last among common twos in output array

```
for i in A.length downto 1
   output_array[temp[A[i]]] = A[i]
   temp_array[A[i]] = temp_array[A[i]] - 1
```

So, let's sum up the above points and write the entire code for the counting sort.

```
COUNTING-SORT(A, output_array, k)

  temp_array[k+1]
  for i in 0 to k
    temp_array[i] = 0

  for i in 1 to A.length
    temp[A[i]] = temp[A[i]] + 1

  for i in 1 to k
    temp_array[i] = temp_array[i]+ temp_array[i-1]

  for i in A.length downto 1
    output_array[temp[A[i]]] = A[i]
    temp_array[A[i]] = temp_array[A[i]] - 1
```

C **Python** **Java**

```c
#include <stdio.h>

void counting_sort(int a[], int output_array[], int k, int size) {
  int temp_array[k+1];
  int i;

  for(i=1; i<=k; i++) {
    temp_array[i] = 0;
  }

  for(i=1; i<=size; i++) {
    temp_array[a[i]] = temp_array[a[i]] + 1;
  }

  for(i=1; i<=k; i++) {
    temp_array[i] = temp_array[i]+ temp_array[i-1];
  }

  for(i=size; i>=1; i--) {
    output_array[temp_array[a[i]]] = a[i];
    temp_array[a[i]] = temp_array[a[i]] - 1;
  }
}

int main() {
  //array is starting from 1. So, fake element -100 at index 0.
  int a[] = {-100, 4, 8, 1, 3, 10, 9, 2, 7, 5, 6};
  int output_array[11];

  counting_sort(a, output_array, 10, 10); //k is 10. size for function is 10, array from 1 to 10.

  //printing array
  int i;
  for(i=1; i<=10; i++) {
    printf("%d ",output_array[i]);
  }
  printf("\n");
  return 0;
}
```

Analysis of Counting Sort

The analysis of the counting sort is simple. For the first for loop i.e., to initialize the temporary array, we are iterating from 0 to k, so its running time is $\Theta(k)$. The next loop is running from 1 to A.length and thus has a running time of $\Theta(n)$. The next for loop is again iterating over the temporary array from 1 to k and thus has a running time of $\Theta(k)$. The last loop is again $\Theta(n)$.

So, the overall running time of the counting sort is $\Theta(k+n)$.

As stated earlier in this chapter, when the value of k will be around the value of n i.e., $k = O(n)$, then the running time of the counting sort will reduce to $O(n)$ and thus the counting sort will run in linear time. And that's why we were saying that we want the value of k to be of $O(n)$.

> “ Software is a gas; it expands to fill its container. ”
>
> - Nathan Myhrvold

PREV (/course/algorithms-bubble-sort/) (/course/algorithms-heapsort/) NEXT