Courses Practice Roadmap Pro

Algorithms and Data Structures for Beginners

13 / 35

13 - Bucket Sort

13:34

Mark Lesson Complete

View Code

Prev

Next

Suggested Problems

Status	Star	Problem	Difficulty	Video Solution	Code
☐	☆	Sort Colors	Medium		C++

Bucket Sort

By now, you are probably getting tired of sorting. Don't worry, this is the last sorting algorithm we will cover, and it is a pretty important one. It is not as popular or widely used as the previous algorithms we have covered. Bucket sort works well when the dataset to be sorted has values **within a specific range**.

Concept

Imagine we have an array of size 6 and it contains values of an inclusive range of $0-2$. The idea behind bucket sort is to create a "bucket" for each one of the numbers and map them to their respective buckets.

There will be a bucket for 0, 1 and 2. This bucket, which is just a position in a specified array will contain the frequencies of each one of the values within the range. For the sake of this example, we only have three values and accordingly we will have three buckets.

> *The term bucket will really just translate into a position in an array where we will map these frequencies.*

Once each one of the buckets is filled with the frequency of each one of the values, we will overwrite all the values in the original array such that they end up in the sorted order. This makes more sense looking at the pseudocode below:

```
fn bucketSort(arr):
    // Going by the example, assume arr only contains 0,1,2
    counts = [0, 0, 0]
    // Count the frequency of each value
    for each n in arr:
        counts[n] += 1
    // Overwrite the original array with values in the bucket
    i = 0
    for n = 0 until counts.length - 1:
        for j = 0 until counts[n]:
            arr[i] = n
            i += 1
    return arr
```

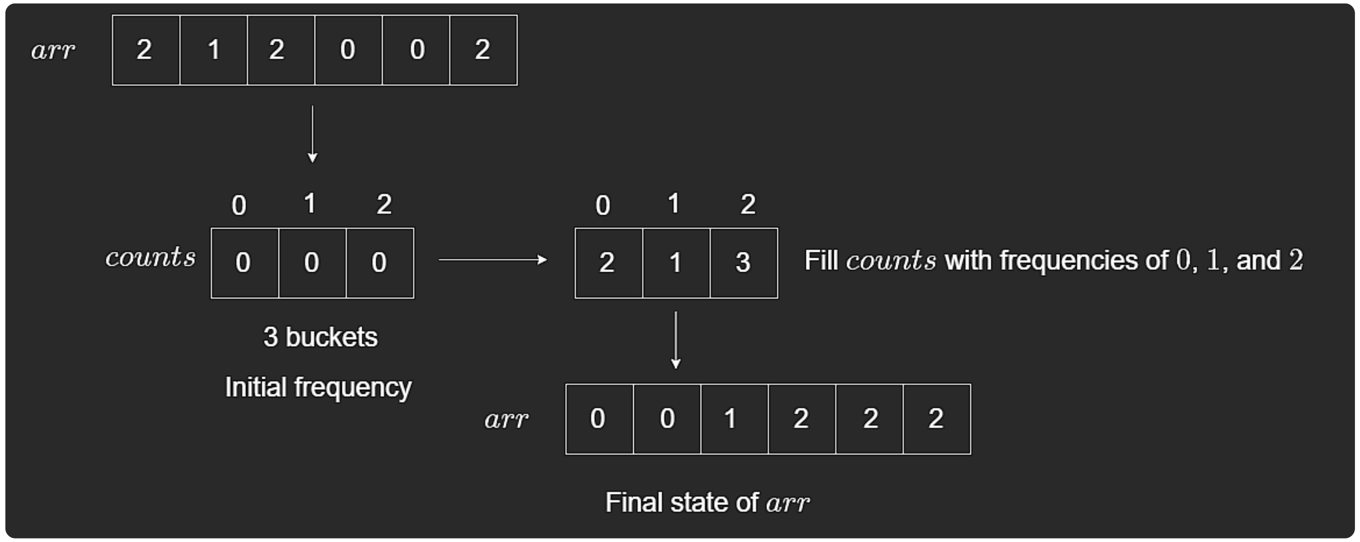
The first part of the pseudocode, right before we do `i = 0` , corresponds to filling up each one of the buckets. Let's explain the second part of the code a bit more.

```
i = 0
    for n = 0 until counts.length - 1:
        for j = 0 until counts[n]:
            arr[i] = n
            i += 1
    return arr
```

- The `i` pointer will keep track of the next insertion position for our original array, `arr` .
- The `n` pointer keeps track of the current position of the `counts` array.
- The `j` pointer keeps track of the number of times that `counts[n]` has appeared.

So, knowing that, we have our `counts` array which is `[2,1,3]` . And, our original input array is `[2,1,2,0,0,2]` .

At the first iteration, $n = 0$, which corresponds to 2 in `counts` . Our inner loop will run two times, overwriting `arr[0]` and `arr[1]` to be `0` . This makes perfect sense because we had two zeros and putting them in `arr` in an adjacent manner will result in them being sorted. This process will continue for each number and the ultimate state of `arr` will be `[0,0,1,2,2,2]` which is the ultimate goal.



Time Complexity

You may be looking at the nested for loop and immediately going, that is $O(n^2)$. That is not quite right. Let's do some analysis. We know that for the first for loop, we are performing n steps since we are going through all the elements and counting frequency.

The first for loop will run n times where n is the length of the `counts` array. However, the inner loop will only run until `counts[n]` , which is a different everytime. The first time it will be 2, then 1 and then 3. Therefore, our algorithm belongs belongs to $O(n)$.

> *It should be noted that nested for loops don't always mean a time complexity of $O(n^2)$. As we saw saw above, it is important to analyze how many times the inner for loop executes on each outer for loop iteration. More information on this can be found in the lessons section for Big-O Notation.*

Stability

Since we are overwriting the original array, there is no way to preserve the relative order of the values. There is no swapping that takes place either. Hence, it will stay unstable.

Closing Notes

So while the bucket sort algorithm runs in $O(n)$ time, we must remember that it will only work if the dataset is within a specified range. Generally, with algorithmic problems, the safest bet is making use of merge, or quicksort or insertion sort (for smaller and nearly sorted data sets) which is generally what is used by the in-built sort methods of programming languages, or sometimes a hybrid of all three.

Now that we can covered all the sorting algorithms, the run-times can be concluded below:

Algorithm	**Big-O Time Complexity**	**Note**
Insertion Sort	$O(n^2)$*	If fully, or nearly sorted, $O(n)$
Merge Sort	$O(n\ log\ n)$	
Quick Sort	$O(n\ log\ n)$*	Picking the largest or the smallest element as the pivot e.g. reverse sorted - $O(n^2)$
Bucket Sort	$O(n)$*	Assuming you have a value within a specified range