Improved Bucket Sort

By Nick Daniel

What’s Bucket Sort and why?

Bucket sort sorts by grouping elements into buckets depending on certain characteristics such as value (think grouping by number ranges). Those buckets are then each sorted; the idea is that since each bucket will be equal with a small number of elements, the sorting of each bucket would become constant resulting in overall linear complexity.

We’re able to achieve linear time complexity! However, this comes with its own downsides in that the algorithm requires linear space as well. This is **NOT** an in place algorithm. Alongside, it desires uniformly distributed data to ensure each bucket is equal in size.

So can this be improved?

Yes, it can be improved... mostly

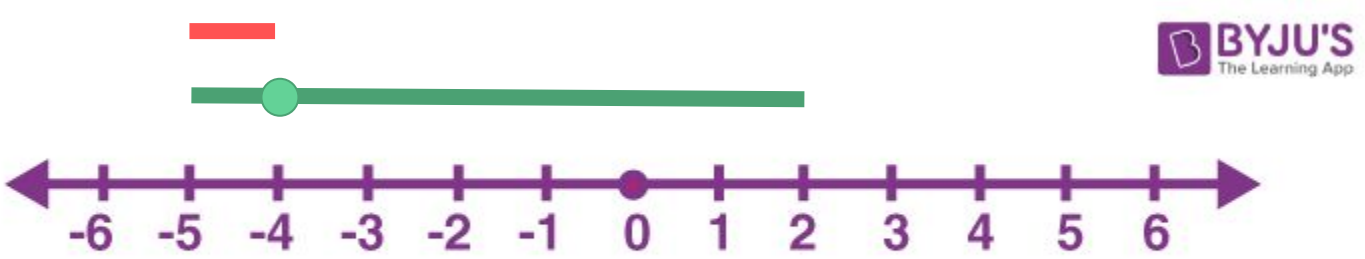
By convention, bucket sort uses insertion sort to sort each bucket. Knowing this, the bucket sort algorithm time complexity degrades to insertion sort time complexity when data is either clustered into a single bucket or when insertion sort worst case setup is encountered.

So why not change the sorting algorithm? You absolutely can and that's what I did. However, before just jumping in too quickly and replacing insertion sort, it’s important to recognize that insertion sort is used as it has low overhead compared to other sorting algorithms; overall, this results in faster performance for small arrays. The sort I chose to use is Introsort, a hybrid of insertion sort, heap sort, and quick sort. This helps bring the worst case complexity down to O(n * log(n)).

This is nice but what about the rest of the algorithm, involving the average case?

Better Bucket Grouping

A lot of implementations of bucket sort don’t really index into the buckets that well and don’t usually support negative values. In my implementation, I make sure to group each item into a bucket by its proportion to the overall range of numbers.



$$IDX = numBuckets * \frac{|currVal - min|}{|max - min|}$$

Improving Cache Friendliness

Profiling bucket sort in its naive implementation reveals a major bottleneck from not taking advantage of modern day cache. Usually, when the 2D array is implemented for the buckets, there exists two layers of indirection; this is terrible for cache performance.

Solution? A custom abstract data type. This data type will store a normal array with the expected amount of items in the bucket; however, if more is needed it will have a pointer to location in memory for extra storage. This works well as bucket sort already expects each bucket to be equal in size due to the uniformly distributed data.

Multithreading

Another method for improvement can be the use of multithreading. Modern day computers aren't usually limited to a single core; therefore, we can leverage the parallelization our computers usually provide. This integrates nicely with bucket sort since each bucket is independent of another. We can parallely sort each bucket and then combine the results at the end.

While profiling, I have realized that the actual grouping of items tends to be the more costly operation. This could also be parallelized; however, it would require the use of synchronization techniques to ensure safe access is made to buckets. This can be implemented rather quickly though since our buckets exists as an abstract data type. I just haven't had the time to implement and fully test it, so this is more of an hypothesis than a definitive answer.

Final Words

Unfortunately, I did not have enough time to properly generate statistics; however, from my rough testing, it appears that there exists a minimum of 2x performance increase compared to extremely efficient sorting algorithms such as pdqsort, that at times achieve linear complexity. The downside is the memory usage; an example that I encountered was the program using 3 GB of memory to sort a 1 GB array. Of course, this depends on the amount of buckets used and the input's size.

Overall, if memory is not a concern, this is definitely a powerful algorithm to utilize.