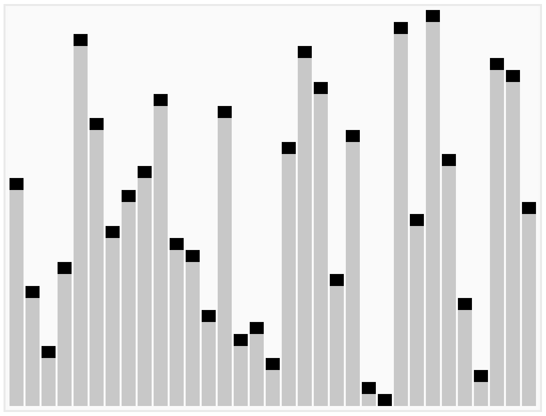

Source: https://en.wikipedia.org/wiki/Quicksort

Sorting algorithms: Quick Sort implementation in Go

Rafał Gałus
Follow

Sep 15, 2019 · 4 min read

One of the most common operation on data that developers do is sorting. Most of us use built-in functions in the programming language we use. And that’s totally fine. But if you want to learn how this actually works you will find out different algorithms and one of them is quick sort, which is one of the most popular and performant around them.

In this post I will show you the basic quick sort implementation in Go and describe step by step how it works. First things first:

What is Quick Sort?

As wikipedia states:

Quicksort is an efficient sorting algorithm, serving as a systematic method for placing the elements of a random access file or an array in order. Developed by British computer scientist Tony Hoare in 1959[…]. When implemented well, it can be about two or three times faster than its main competitors, merge sort and heapsort.

Okay.. But how it actually works?

Imagine that you have an array of numbers { 6, 2, 7, 4, 1 }. Quick Sort iterates over an array with a pivot value (will cover that later, but it's one of the array elements) to create two sub-arrays containing values less-than and greater-than pivot value (e.q. for pivot value 4 in the array above it would create {2, 1} and {6, 7} and pivot value would be included in one of them). Then the same process applies to these sub arrays and so on till all elements are sorted. There is great animated graph on wikipedia illustrating how it works (you can see it at the top of this page).

Go implementation

Let's try to implement it in Go then. First, let's create our *sort* package with QuickSort function that accepts slice of integers and return it.

In the code above we have our QuickSort function that takes slice of integers as a parameter, clones it (lines 4–8) so we don't work on actual passed array (keep immutability). Then we will call our actual sorting logic and return sorted array.

Let's add our sorting function that we will use recursively:

Let’s explain it line by line. Our function accepts 3 parameters: first one is our array to sort, second and third param is index value to indicate start and end of a sub-array within our main array (e.q. array contains {1, 2, 3, 4, 5} so sort(arr, 1, 4) would sort sub-array of {2, 3, 4} because passed parameters points to this part of our array. You can see here that we don’t actually create new arrays, we just point out function to different parts of our main array to sort).

Next we define our pivot value (line 2) which can be a random element of our sub-array but usually it’s its last value. Below that we define splitIndex (line 3) which will contain index value of position in the array that splits it into two new subarrays: less-than pivot and greater-than pivot. Then we iterate over our subarray defined by function parameters (lines 5-14) and we check if value at this index is lower than pivot value, if so we move it to the beginning of an array, switching positions with element at splitIndex (lines 9–10). Then we increase our splitIndex value by 1 (line 12) to indicate that before this index there are values less-than pivot and after this index there are only values greater-than pivot. Then we move our pivot value next to last value that was lower than it which is at splitIndex (lines 16–17).

That’s it. When function finishes its execution, subarray contains values sorted like *[…lower-than-pivot, {splitIndex}, …greater-than-pivot]*. So for example if we would call sort([]int{5, 3, 8, 2, 4}, 0, 4), result of a function would be {3, 2, 4, 5, 8} and splitIndex would be 2. We have subarray of lower than pivot which is {3,2,4} and greater than pivot which is {5,8}.

Let’s add remaining logic:

Since we are continuously creating sub-arrays finally we will end up with empty sub-arrays or arrays containing single value, so we need to add simple condition (lines 2–4) to stop executing sort on this sub-array since it’s already sorted.

Finally, we want to recursively call our sort function for created subarrays (lines 23–24). So it applies the same process, finding all lower-greater values and defining subarrays till every element in our main array is sorted.

So our complete implementation looks like this:

You can find source code here on GitHub https://github.com/rgalus/go-quicksort