Back to the fundamentals: Sorting algorithms in Swift (from scratch!)

Ennio Masi Follow
Oct 10, 2017 · 7 min read

You can read perfectly this post on my blog ennioma.com. 🚀

. . .

In the latest period I decided to read again the amazing Cormen and co. —Introduction to Algorithms book and it threw me back to an age ago when I attended the University.

So I wanted to write this post for personal knowledge but also because I've read a lot of complains around the web about questions like "*What's the fastest sorting algorithm?*" or "*Write down the bubble sort*" during the iOS interviews. So maybe this post could help you to prepare the next interview 🙂

The topics covered in this article are:

- Big O Notation
- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort
- Quick Sort
- Bonus :) Bucket Sort

Big O Notation

The first topic to cover before learning about sorting algorithms is the concept behind the **Big O Notation**, or before this, the concept of **asymptotic growth** of an algorithm:

> *[…] That is, we are concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound. Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.*

In this definition taken from the Cormen Book, there are two underlying concepts:

1. this asymptotic growth is something we use to compare algorithms, so we can answer properly to questions like "What's the fastest sorting algorithm?" considering their asymptotic growth;
2. it's a measure of the performances of the algorithm based on the input;

Using this definition we can completely skip any other formal definition, referring to the Big O notation considering these rules:

1. *This step is O(1)* –> this means that, regardless the input size, the considered step is constant;
2. *This cycle is O(N)*, where N is the input size –> this means that, the performance of this cycle is directly related to the input size; if a cycle is *O(N)* and another on the same input is *O(N^2)* then the first cycle is faster than the second on the same input;
3. If an algorithm is *O(2*N)* or *O(N+N)* we just say that it's *O(N)*! The constants don't have any value here!

Generally speaking we can refer to *asymptotic growth* in different scenarios but we usually refer to the worst case in time complexity: this is finally the Big O Notation we refer to! (**Hint**: The complexity could be expressed in terms of time or space)

Let's now move to talk about the most famous sorting algorithms 🤓

Bubble Sort

This algorithm nowadays has just an educational meaning because it's always slow (with *always* I mean it's slow in the best or worst case), it's **O(n^2)** in time complexity. This means that it needs always **n^2** steps to converge to the solution (i.e. input N = 50, number of steps to reach the solution = 1000).

It's called *Bubble* Sort because some items are moved quickly along the array and other slower and the effect is like a lot of bubbles in a cup of champagne 🤔

This is an in-place algorithm because it doesn't need any external storage to perform the sorting.

As you can see in this animation (for *ascending sorting*), the concept behind this algorithm is quite simple. If the order is ascending, the idea is to move the *higher* number to the end of the array for every iteration. This means that at the end of every iteration *i* the item at *array[i]* will be sorted and in the next iteration the algorithm will try to sort the array in *[0, i-1]*.

In order to make it a little bit more interesting, I've created an Array extension. This implies that you don't have to pass the array to sort to the method but you can do something like:

```swift
let myArray = [4, 2, 0, 1]

// by default the sort is ascending
let sorted = myArray.bubbleSort()

// ascending
let anotherSorted = myArray.bubbleSort(<)
```

But now let's focus on the algorithm:

```swift
import Foundation

extension Array where Element: Comparable {

    func bubbleSort(by areInIncreasingOrder: ((Element
        var data = self

```

So this code is quite simple:

1. iterate over the array until the second to last element;
2. iterate over the array until the *N-i* element: as I told you previously the point here is that at every iteration we are going to sort the array from the latest element back to the first one;
3. the algorithm swap two elements only if them don't respect the sort order; as you can notice we inject the desired order as Apple does!

The point 2 is the reason why this algorithm is always **O(n^2)**. Basically regardless from the current status of the array it cycles **two times** over the entire array.

Hint: As you can see I've also added a **swap** method but starting from swift 4 we can use the **Array.swapAt(:)** method and this is extra useful because if you try to apply my swap method calling **swap(a[i], a[i])** it will crash! You have to check that the two indexes passed to the method are different before calling my method!

Insertion Sort

To quickly understand how insertion sort works, think that it’s the usual way in which the human being sorts cards. Starting from the first one, we look for a place in which put the current card in order. Then we move to the second, looking for sorting that card and so on until the last one. You can see this in the next animation:

The main point here is that for every iteration the algorithm finds a *key index* and it looks back to the first one (in the opposite way if the sort is *descending*!) in order to find a sorted place for the current item.

In this way at the end of every iteration we have a sorted array up to the *current key index* and an unsorted array *from the current key index to the end*.

```
extension Array where Element: Comparable {

    func insertionSort(by areInIncreasingOrder: ((Elem
        var data = self

        for i in 1..<data.count { // 1
            let key = data[i] // 2
```

1. Initially the algorithm cycles over the second to last element;
2. it *blocks* the current *key* element;
3. the algorithm looks back until the first element in order to sort the array from 0 to key;
4. swap elements;

Also the insertion sort is **O(n^2)** but typically for a small chunk of data it works better then Bubble sort.

Selection Sort

This it the third common algorithm that is easy to be implemented but it's still quite slow. The selection sort is just a variation of the insertion sort but it works in a slightly different way.

```
extension Array where Element: Comparable {
    func selectionSort(by areInIncreasingOrder: ((Elem
        var data = self

        for i in 0..<(data.count-1) {
            var key = i // 1

```

As I told you previously, this algorithm is really similar to the insertionSort because it blocks a *key element* (1) and compare it with the next elements until the last one (2) (in the insertionSort the check was done on the opposite side!). At the end of the cycle, if there is an element that is smaller than the key element, then the algorithm swaps them. (the assumption in this description is that we are talking about an ascending order).

Merge Sort

The Merge Sort is, in this implementation, a usual divide-and-conquer algorithm. The idea behind this algorithm is that it proceeds with a top-down splitting of the array. Once this operation is completed, the array is sorted bottom-up, sorting initially every couple of adjacent elements and then proceeding up to the entire array.

A good quality of this algorithm is that its time complexity of is **O(nlogn)** (definitely better then the previous **O(n^2)**) in any case.

```
import Foundation

extension Array where Element: Comparable {
    private func merge(left: [Element], right: [Elemen
        var output: [Element] = []

        var mutableLeft = left
```

In my implementation, the mergeSort will modify the input array, so in order to include this into the extension, I've created a private *_emMergeSort* function which takes the array and the desired sort direction.

This is a recursive algorithm, this means that somewhere it call itself until a *base case* which manage the end of the recursion.

Our base case is if the current array (at the n-th level of recursion) contains only a single element. This is the end of the first phase, the so called *divide* part that you can see in my previous animation.

Once the base case is reached, the *merge function* (3) is called on every previously splitted chunk of data.

Quick Sort

The quick sort is another divide-and-conquer algorithm that manages the way in which it splits the array differently from the merge sort. A weird quality of this algorithm is that in the worst case it performs like bubble, selection and insertion sort, **$O(n^2)$**, and in the average case it's like the merge sort, **O(nlogn)**.

So the point is that it on average performs like the merge sort but it uses less space and it's proven that it works better then it on small data.

```
import Foundation

extension Array where Element: Comparable {

    private func _quickSort(_ array: [Element], by are
        if array.count < 2 { return array } // 0

```

I like this implementation ☺️ as the previous implementation, this is a recursive algorithm, so I needed to pass inside the *_quickSort* method the array to be sorted.

An important detail about this algorithm is that the *pivot* is used to create the current partition. In this algorithm I don't use any clever pick of the pivot element (1), I just take the first one for every partition. The problem is that it has been demonstrated that on large data, picking consistently the bad pivot could move the performance of the algorithm to **O(n^2)**.

As the merge sort, at each step the array is divided (2 and 3) splitting data for elements smaller than the pivot on the left and the other on the right (it's the opposite for a descending sort!).

Once the algorithm reaches the base case (0), it proceeds bottom-up merging the partitions that are already sorted, it has just to append each partition from the left to the right!

Special case: Bucket Sort

As we have seen every algorithm has its own qualities but sometimes there are algorithms that work really well for specific context.

I always focused on the following algorithm because a professor during an exam asked me if ***O(nlogn)*** is the best performance possible for sorting. And the answer is **no**! ☺️ (well, this could happen also during the interviews!)

Or at least, the answer is no if and only if we specify on which kind of data we apply this algorithm.

If we have a list of well distributed numbers from M to N, this algorithm can sort this list in a linear time **O(n)**.

The code is the following one:

```
import Foundation

extension Array where Element == Int {
    func bucketSort(reverse: Bool = false) -> [Element
        var data = self

        guard data.count > 0 else { return [] }
```

The main idea is quite easy:

1. create N buckets (the way in which we create the buckets could definitely improve the performances!);
2. run over the input array and increase the number of occurrences of buckets[current_index];
3. create the output array, iterating over the buckets in the desired order;

. . .

The repository for this article is on Github.

Feel free to contact me on Twitter for any comment. I'd love to hear any kind of feedback! ✌🏼 And I hope that this article can help you with your next interviews 😈

Thanks for reading! 👽