Daniel Spector

Computer Science Fundamentals: Searching & Sorting

To review some of the material that I learned over the last few weeks, I wanted to create a series dedicated to computer science fundamentals. My hope is that I will be able to help others while learning at the same time.

Searching and sorting are huge issues in computer science. By examining different approaches to solving these problems, we can understand why they're so important to understand.

Let's give an example. Say you have a phone book and you need to find a specific person's name. If you want to find the person's name you have a few ways you can do it. The most obvious way is to start at the beginning and start going through names one by one. This is called a "linear search" or more colloquially, the brute-force method. Assuming the name is in the phone book you are looking at, you are guaranteed to find the right answer. The problem is that if the person you're looking for happens to be the last person in the book, you're going to have to go through every listing there.

What you we described above is an algorithm. When I started programming, just hearing the word "algorithm" would make me tense up. Over time, I learned that algorithms are nothing more than a set of repeatable instructions for solving a particular problem. The linear search method's algorithm, in pseudo-code, would look something like this:

```
I have a value called "Matz". I need to find "Matz" in this list which I am re
ferring to as a phone book.
Start at the beginning of the list. Compare the first item. If the item that I
found is equal to “Matz”, return the phone number. Otherwise, move on to the n
ext item in the list
```

Obviously, the algorithm above is not very efficient. Intuitively, we can understand that if "Matz" is the last person listed, we're in for a long night. Computer scientists have adopted a common language for evaluating the efficiency of algorithms using "Big O Notation". The linear search algorithm has a Big O of N. N refers to the number of items in the list. Big O refers to the worst-case scenario when measuring an algorithm's performance. If we have a phone book with a million names, we could have to look through each of those million names before we found Matz. On the other hand, Omega refers to the best case scenario. The Omega of the linear search algorithm is 1 because if “Matz” is the first item in the phone book, it will be an easy night. When the Omega of algorithm and the Big O of an algorithm match, we can shorten it and refer to its efficiency as Theta.

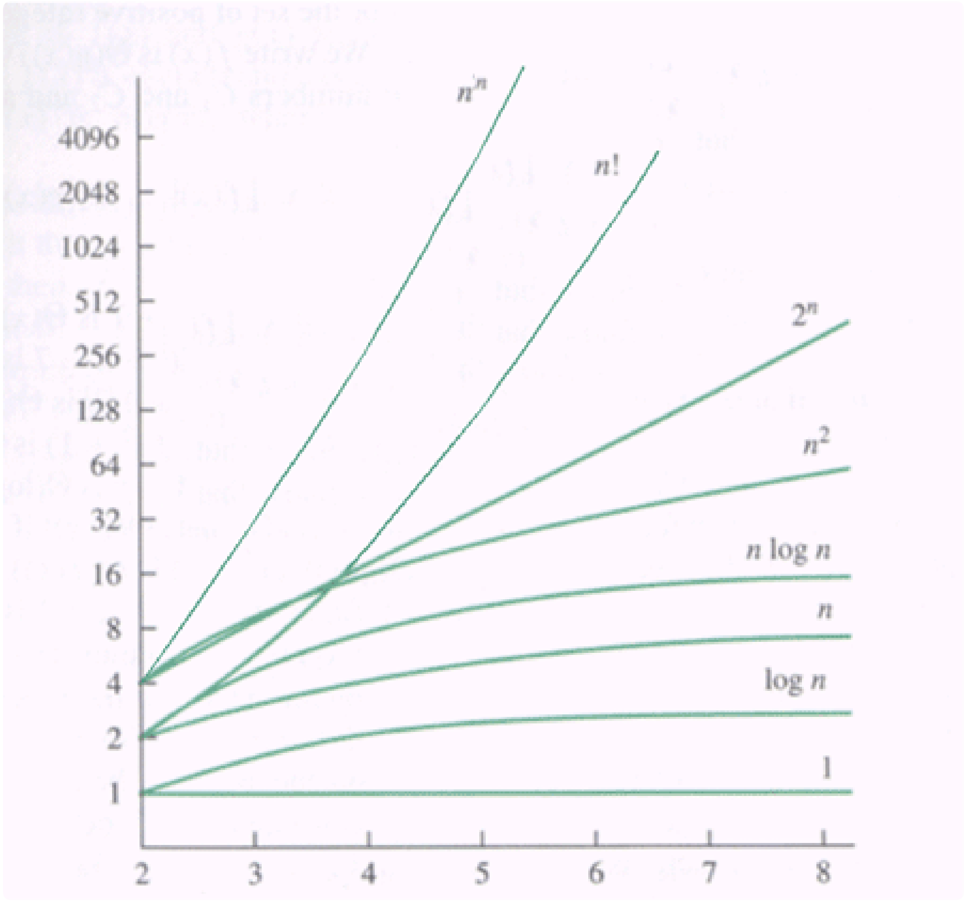
Don’t be intimated when you see Big O. Seriously. It’s just a measurement of time. I used to get really intimidated when I saw math notation so I’ve purposely left it off of this discussion. Let’s talk a little more about how we can improve the search algorithm above.

Most people don’t find a number in a phone book by starting at the beginning. Since we know that the phone book is in alphabetical order, another algorithm we might want to implement is called “binary search”. The pseudo-code for a binary search could look something like this:

```
Open up a phone book in the middle. Compare a name you find. If the name you s
ee if alphabetically after Matz, use the left half. Otherwise, use the right h
alf. Repeat.

We split the phone book in half and compare where we stand. If we’re looking a
t “Ruby”, we can throw away everything that comes after it and repeat the sequ
ence. Open up to the halfway-point and compare. Throw out what we don’t need.
```

The binary search algorithm is actually very efficient. The reason is that even if the number of phone listings were to jump from one million to two million, we would only need one extra step. The Big O efficiency of the binary search algorithm is what is called log n. Log n indicates that even as n grows larger and larger, the time it takes to complete the exercise barely rises. As you can see in the image below, increases in n barely move the needle when you have an algorithm with an efficiency of log n. A Big O of n will increase the time needed to solve the equation linearly. In the worse case, you can have an efficiency of N-Squared which means that as the number of items increases, the time to compute increases exponentially. Ouch.



Binary search is great but it has one critical flaw. It depends on a sorted list. If the list you are searching through is not sorted, a binary search won't help you.

How are you going to sort a list? There are almost countless ways a sort can be implemented, I'm only going to look at a few.

Selection Sort

The selection sort is one of the most inefficient methods of sorting. The selection sorting algorithm works by dividing the list into two parts, one part which is already sorted and the other which needs sorting. Initially, the sorted section is empty. To sort the list, the selection sort will walk through the entire list and determine the lowest number. It then adds that number to its sorted list and continues. The selection sort is so inefficient because even in the best case scenario it has an O of n-squared. Even if your list is perfectly sorted to begin with, since the algorithm has to walk through the entire list each time in order to make sure that it knows which is the smallest number, it doesn't gain any time by having the list initially sorted. On the other hand, if the list is completely unsorted, the efficiency will quickly reach an O of n-squared which makes it a poor choice for most use-cases.

Bubble Sort

The bubble sort got it's name because the largest items will "bubble" to the top. In order to implement a bubble sort, start at the beginning of your array. Compare the first item of the array against the second item in the array. If the first item is greater than the second item, switch the two positions. Continue by comparing the second item to the third item etc… When you reach the end of the array, start the process over again.

Intuitively, a bubble sort doesn't seem that efficient. In the worst case scenario it has a Big O of the dreaded n-squared. As the number of items in a list increases, the time to sort them can increase exponentially. The reason is that you must go back to the beginning of the array and repeat until the items are sorted. You can only move them one at a time, which really slows you down. For larger data sets, bubble sort can take a painfully long time. On the other hand, the best case scenario for a bubble sort is an O of n. This is because you would only need to step through the list once in order to determine that it was already sorted and the time it takes will simply be determined by the number of items that you are sorting.

Insertion Sort

The insertion sort works a little bit differently than the two described above. Imagine you are playing the card game called President. If you've never played the game, you don't have to know anything besides that each player is given a random set of cards to start. Due to the rules of the game, most players choose to order their cards from lowest to highest. Usually people will order their cards by moving along and "inserting" the cards into the proper order. You select a card, put it in its proper sorted place and move on to the next card, shifting the cards as needed. If you have a 2, 5, 10 and K in your hand, and you came across a 7, you would insert the 7 in between the 5 and the 10. In the best case, the insertion sort has an efficiency of N because if you were dealt a perfectly sorted hand, you would only need to walk through the cards once in order to determine they were correctly sorted. On the other hand, the worst case scenario would mean you were dealt a hand perfectly out of order and the efficiency of the sorting would reach an O of n-squared.

Merge Sort

Finally we move on to merge sort. A merge sort is a little more difficult to conceptualize but has large benefits when implemented correctly. A merge sort follows a very simple, yet powerful algorithm.

From David Malan at Harvard:

```
On input of n elements:
 If n < 2
   Return.
 Else:
   Sort left half of elements.
   Sort right half of elements.
   Merge sorted halves.
```

What you'll notice is that there is an element of recursion to the merge sort. Recursion is the process by which a method calls itself and performs that action. Here, the merge method says to "sort" the left and right half of the elements. What happens is that the list will continue to separate out into continuously smaller right and left halves until it reaches 1 (when n is less than 2). Once that happens, the method will "return" and merge the sorted halves, which means that the two parts will merge together. The halves continue to merge together until the list is broken up into two sorted halves and at that point they combine together into one large sorted list.

The merge sort actually "cheats" compared to the other sort algorithms we talked about because it uses a second, separate list in order to keep track of the sorted elements. Under the hood, it bounces back and forth between the two arrays in order to keep track of its sorted parts. This is a common tradeoff in computer science. In order to make our sorting algorithm more efficient, we are forced to use additional memory.

The merge sort is the most efficient of the ones we've looked at. In the worse-case scenario, the merge sort is an O of N log n. The N in front of the log n is there because you can't avoid walking through each item in your list in order to determine whether they are properly sorted. However, since you are constantly splitting and merging, much like the binary search, the efficiency of the actual sorting is log n because doubling the number of items in the list is only one additional operation.

Wrapping Up

There are many other sorting algorithms that we haven't discussed. Ruby uses a version of the quick-sort algorithm under-the-hood when you call .sort on Enumerable object. Understand and conceptualizing the different sorting algorithms is difficult. In order to help, I would recommend two resources with a more visual representation. One is a popular video which uses sounds and motion to give you a sense of how 15 different sorting algorithms operate. The video can be found here. The second resource is sorting-algorithms.com which allows you to easily compare different sort algorithms and get a sense of the relative efficiency or inefficiency of their operations.

I hope you found this information helpful in understanding the different searching and sorting algorithms. A special thanks goes to the CS50 class at Harvard and especially the instructor, Prof. David Malan, who's extremely lucid explanations really helped clarify my understanding of the above topics. I hope I did it justice.

If you have any comments or questions, please don't hesitate to reach out. You can find me on Twitter @danielspecs.

« Return to the home page 27 Apr 2014