Elementary Algorithms

Daniel Wisdom[*]

22 September 2018

1 Introduction

Today, we'll talk about two basic topics in algorithms, sorting and searching, that will provide a foundation for reasoning about algorithms. We'll also extend our discussion of computational complexity by discussing how quickly our algorithms run in practice. **Note that most of the content in this lecture is covered in AP Computer Science.**

2 Complexity

Computational complexity is a way of expressing how the running time of an algorithm depends on the size of the input. We use $O(f(n))$ to say that the the complexity, or big-O, of the algorithm is $f(n)$. $f(n)$ is often n, n^2, or $\log n$. This does not mean that with $O(n^2)$ and $n = 5$ the program will take 25 seconds. It means that if we graph runtime vs n, we will see a parabola.

The complexity we care about is almost always asymptotic complexity. This means the runtime with larger and larger values of n. This means we only care about the fastest growing term in complexity. Let's say I have a program that takes $n^2 + n$ seconds to run. I can rewrite this as $n * (n + 1)$. If $n = 1$ million, I have a runtime of one million times basically one million. Therefore this algorithm is $O(n^2)$.

Algorithmic complexity is at the heart of Senior Computer Team. Even a well written, finely tuned $O(n^3)$ algorithm will not beat a sloppy $O(n)$ solution. Instead of focusing on how to optimize code to run slightly faster[1], we will focus on choosing and inventing better algorithms that will run faster with larger inputs.

3 Common Complexities

These terms are in order from slowest to fastest growing.

- $O(1)$ - constant
- $O(\log n)$ - logarithmic
- $O(n)$ - linear
- $O(n \log n)$ - linearithmic (just say "n log n")
- $O(n^2)$ - quadratic
- $O(n^3)$ - cubic
- $O(2^n)$ - exponential
- $O(n!)$ - factorial

[*]Inspired by a lecture by Kevin Geng

[1]This is can be very useful in some circumstances. It tends to involve near-hardware level coding, which is hard.

4 Doubling test

We can experimentally measure the runtime of a program, in much the same way that one might conduct a scientific experiment. Normally, we use the doubling test: what happens to the runtime of our program each time we double the size of the input? This is because it makes the complexity easy to estimate: for an algorithm with a complexity of $O(logn)$, the runtime increases linearly, for $O(n)$, the runtime doubles, and for $O(n^2)$, the runtime quadruples.

Additionally, using the doubling test allows us to easily plot the runtime on a logarithmic scale. On such a scale, functions of the form $O(n^k)$ appear as straight lines with slope k. We can then estimate the value of k by performing a linear regression.

5 Sorting

The objective of a sorting algorithm is to rearrange the items in an array so that they are arranged in a well-defined order. Those items can be anything: numbers, strings, and even custom data types, so long as we have some way of determining how to order them. In Java, this can be done using the `Comparable` or `Comparator` interfaces, but we will use Python in this lecture, in order to avoid worrying too much about implementation details.

5.1 Selection sort

The first algorithm we will consider is quite simple. First, we find the smallest element in an array; then, we put it into the first position. Next, we find the smallest element among the remaining elements, and put it into the second position. We continue until we have sorted the entire array.

We need some way of quantifying the performance of this algorithm, so we will count the number of compares and exchanges performed. We make $n-1$ total passes through the array, each of which requires one exchange. However, we make $N-i-1$ compares for $i \in [0, n-1]$, so the number of compares is approximately $N^2/2$. Note that the number of operations remains constant, even if the input data is already sorted!

5.2 Insertion sort

Next, we consider another algorithm that is closer to how you might sort in real life. We loop through each of the items in an array, inserting each of them into the correct position among the items before it that have already been sorted. Coding insertion sort does require a bit of care.

If the array is already sorted, we only need to perform $n-1$ comparisons and 0 exchanges total! However, in the worst case, we may need to perform i comparisons and exchanges for $i \in [0, n-1]$. In total, we will perform approximately $N^2/2$ compares and exchanges.

5.3 Standard library sort

For practical purposes, it's best to use the standard library function for sorting instead of writing your own. All the library sorts are $O(n \log n)$.

- Java: `Arrays.sort(arr)` sorts an array, and `Collections.sort(arr)` sorts an ArrayList.
- Python: `arr.sort()` sorts an list in-place, and `sorted(arr)` returns a sorted copy of a list.
- C++: `std::sort(arr, arr + length)` sorts an array, and `std::sort(arr.begin(), arr.end())` sorts a vector container.

6 Searching

For this lecture, we use a narrow definition of searching: trying to find an item in an array. The simple way to do this, of course, is to loop through every item in the array, and check whether it is equal to the item we are looking for, with a complexity of $O(n)$.

However, we can use a technique called binary search to speed up this process. The basic idea is that if the items in an array are sorted, then we can quickly determine which half the item we are looking for is in by examining whether it is greater or less than the middle element. Because we can cut the search space in half at each step, the complexity of binary search is $O(logn)$. Note, however, that binary search is difficult to implement correctly.

Of course, binary search does require that the items in the array be sorted. So if sorting takes $O(nlogn)$ time and naive searching takes $O(n)$ time, then what use is binary search? The answer is that with binary search, sorting functions as sort of an investment. Once you've performed an $O(nlogn)$ sort, you can perform any number of searches in $O(logn)$ instead of $O(n)$.

7 Three-sum

Consider the following problem: we are given a list of n integers, and we would like to know how many subsets of three integers sum to 0.

It should not be difficult for us to come up with a solution that works: simply loop through all pairs of three integers using a for loop, then add them together, and check whether the result is zero.

We can try to get a sense of how efficient this program is by counting the number of array accesses. Using our mathematical knowledge, we can determine that if the inner loop never breaks, it will execute approximately $n^3/6$ times. Since the inner loop contains three array accesses, this code performs approximately $n^3/2$ array accesses for a complexity of $O(n^3)$.

How can we improve the efficiency of this algorithm? Note that if we want to find three elements that sum to zero, then from any two elements a and b we can determine the third, $c = -(a + b)$. If we loop through all possible combinations of two elements, then search for the third using binary search, then we can reduce the complexity of our algorithm to $O(n^2logn)$.

For comparison, when n=5000 the $O(n^3)$ algorithm takes $1.25 * 10^{11}$ operations, while the $O(n^2logn)$ algorithm takes $3.07 * 10^8$ operations. The improved algorithm is over 400 times faster.

How about four-sum? Can we do better than $O(n^3logn)$?

8 Newspaper Delivery

A long road has 100,000 houses on it, each with a unique integer address. All the addresses are given in a list in no particular order. 100,000 newspaper delivery trucks will each drop off a newspaper at every house on an interval [a,b], inclusive. All 100,000 of these intervals are given. If multiple trucks visit one house, that house gets multiple papers. Find the total number of newspapers delivered by all the trucks. Aim for an $O(n \log n)$ solution.