Design and Analysis of Algorithms CSCI 5454
Lecture 0 10 January 2011
Prof. Aaron Clauset

1 Why algorithms?

Q: Can I get a good programming job without knowing something about algorithms and data structures?

A: Yes... but do you really want to be programming GUIs your entire life?

Algorithms are the heart of computer science, and their essential nature is to automate some aspect of the collecting, organizing and processing of information. Today, information of all kinds is increasingly available in enormous quantities. However, our ability to make sense of all this information, to manage, organize and search it, and to use it for practical purposes, e.g., self-driving cars, adaptive computation, search algorithms for the Internet or for social networks, artificial intelligence, and many scientific applications, relies on the design of efficient algorithms, that is, algorithms that are fast, use little memory and provide guarantees on their performance.

1.1 The business pitch

1. Almost all big companies want programmers with knowledge of algorithms: Microsoft, Google, Facebook, Oracle, IBM, Yahoo, NIST, NOAA, etc.
2. In most programming job interviews, they will ask you several questions about algorithms and/or data structures. They may even ask you to write pseudo or real code on the spot.
3. Your knowledge of algorithms will set you apart from the large masses of interviewees who know only how to program.
4. If you want to start your own company, you should know that many startups are successful because theyve found better algorithms for solving a problem (e.g. Google, Akamai, etc.).

1.2 The intellectual pitch

1. You'll improve your research skills in almost any area.
2. You'll write better, faster, more elegant code.
3. You'll learn to think more abstractly, more mathematically, and more clearly.
4. It's one of the most challenging and interesting area of Computer Science.

1.3 An example

The following is a question commonly asked in job interviews:

- You are given an array with integers between 1 and 1,000,000.
- All integers between 1 and 1,000,000 are in the array at least once, and one of those integers is in the array twice

Q: Can you determine which integer is in the array twice?

Q: Can you do it while iterating through the array only once?

1.3.1 A naïve solution

Here's a simple and intuitive solution for solving this problem:

1. Create a new array of ints between 1 and 1,000,000; well use this array to count the occurrences of each number.
2. Initialize all entries to 0.
3. Go through the input array and each time a number is seen, update its count in the new array
4. Go through the count array and see which number occurs twice.
5. Return this number

How long does this algorithm take? This algorithm iterates through 1,000,000 numbers 3 times. It also uses twice as much space as the original input sequence. (Maybe 10^6 does seem very large, but imagine something 10^{10} or even bigger.)

If the size of the input array is n, we say mathematically that asymptotically, that is, as $n \to \infty$, this algorithm takes $\Theta(n)$ time. We can also specify the amount of additional space the algorithm takes, i.e., the memory beyond what's necessary to store the input array: $O(n)$. That is, both are *linear* in the size of the input sequence, and moreover, they are bounded above and below (a *tight* bound) by linear functions.

In fact, this solution is not efficient: it wastes both time and space. Can we do better?

1.3.2 A better solution

Recall from discrete mathematics that $\sum_{i=1}^{n} i = n(n+1)/2$. We can use this mathematical fact to make a much more efficient algorithm.

- Let S be the sum of the values in the input array.
- Let n be the largest integer in the array; in this case, $n = 1,000,000$.
- Let x be the value of the repeated number.
- Then, $S = n(n+1)/2 + x$.
- Thus, $x = S - n(n+1)/2$.

Thus, an efficient algorithm is the following:

1. Iterate through the input array, summing the numbers; let S be this sum; let n be the largest value observed in this iteration.
2. Let $x = S - n(n+1)/2$.
3. Return x.

How much time does this take? We iterate through the input array exactly once, so it's roughly three times faster than the naïve algorithm. Plus, we use only three constants to store our intermediate work, so this uses much less space. Asymptotically, it takes $O(n)$ time and $O(1)$ space.

1.4 So what?

Designing good algorithms matters. But, it's not always this easy to improve algorithms. Many of the algorithms we'll see in this course are already quite clever, but very few of them are the most efficient algorithm known for the problem they solve. The point here is that the naïve approach is almost always not very efficient, but with a little thought and the tools and tricks you learn in this class, you can almost always do better than the naïve approach.

2 Analyzing algorithms

2.1 RAM model of computation

In this class, we will use the RAM model of computation. This means that all instructions operate in serial (no concurrence, no parallel computation except when noted); all atomic operations like addition, multiplication, subtraction, read, compare, store, etc. take unit $O(1)$ time; and all atomic data (chars, ints, doubles, pointers, etc.) take up unit $O(1)$ space.

2.2 Worst-case analysis

We will generally be pessimistic when we evaluate resource (time, space) bounds. To analyze the running time, we'll focus on the worst possible input sequence. This is the "adversarial" model of algorithm analysis. Surprisingly, in many cases, we'll still be able to get fairly good bounds, and because it's a worst-case analysis, life can get no worse than we expect. In some cases, we may consider the "average" case, but often this case is about as bad as the worst case.

2.3 Asymptotic analysis

Unless otherwise specified, in this class we will only care about the asymptotic behavior of algorithms. This typically simplifies the analysis and communicates the core differences between different algorithms more concisely. I assume that you already are familiar with the basics of asymptotic analysis and are familiar with what $O(.)$, $\Theta(.)$, $\Omega(.)$, $o(.)$ and $\omega(.)$ mean.

The asymptotic behavior of an algorithm should be expressed as a function of the input size, which it conventionally denoted by n. (In some cases, there are multiple input parameters, e.g., in a graph $G = \{V, E\}$ where $|V| = n$ and $|E| = m$; in this case, the asymptotic behavior can be a function of multiple parameters.)

As a rule of thumb, in asymptotic analysis, we are not interested in constants, either multiplicative or additive, so $5n + 2$, n and $10000n$ are all $O(n)$ and 10^4, 2^{50} and 4 are all $O(1)$.

For big-O, we are interested mainly in the leading term (fastest growing) of the function, and thus we can neglect slower growing functions or trailing terms (but not multiplicative functions). Thus, $n^2 + n + \log n$, $10n^2 + n - \sqrt{n}$ and $n^2 + 10^6 n$ are all $O(n^2)$.

Chapter 3 in the textbook covers asymptotic analysis. Read it. Know it.

Here are some analogies and examples you can use in thinking about asymptotic notation.

O	"$\leq$"	This algorithm is $O(n^2)$, that is, worst case is $\Theta(n^2)$.
Θ	"$=$"	This algorithm is $\Theta(n)$, that is, best and worst case are $\Theta(n)$.
Ω	"$\geq$"	Any comparison-based algorithm for sorting is $\Omega(n \log n)$.
o	"$<$"	Can you write an algorithm for sorting that is $o(n^2)$?
ω	"$>$ "	This algorithm is not linear, it can take time $\omega(n)$.

Here are some more mathematical facts you should memorize:

1. L'Hopital's rule:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\frac{\partial}{\partial n} f(n)}{\frac{\partial}{\partial n}\, g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

2. $(x^y)^z = x^{yz}$
3. $x^y x^z = x^{y+z}$
4. $log_x y = z \implies x^z = y$
5. $x^{\log_x y} = y$ by definition
6. $\log(xy) = \log x + \log y$
7. $\log(x^c) = c \log x$
8. $\log_c(x) = \log x / \log c$

3 For next time

1. Read Chapters 1, 2 and 3 in CLRS.
2. Problem Set 1 available on class website; due February 1st.