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 collecting, organizing and processing information. Today, information of all kinds is increasingly available in enormous quantities.
However, our ability to make sense of 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 *correct* and *efficient* algorithms, that is, algorithms that perform as intended on all inputs and 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, Apple, 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 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 they've found better algorithms for solving a problem (e.g. Google, Akamai, etc.).

1.2 The intellectual pitch

1. You will improve your research skills in almost any area.

2. You will write better, faster, more elegant code.

3. You will think more clearly, more abstractly and more mathematically.

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 containing integers between 1 and 1,000,000.
- Every integer from 1 and 1,000,000 is in the array once, but one 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 L of `int`s between 1 and 1,000,000; we'll use this array to count the occurrences of each number.
2. Initialize all entries to 0.
3. Iterate over the input array; each time a number i is seen, increment the count $L[i]$ in the new array
4. Iterate over L and see which number occurs twice, $L[i] > 1$.
5. Return that number, i.

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 doesn't seem too big, 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. How much better can we do?

The good news is that, while inefficient, this algorithm is *correct*, meaning that on every possible input, it returns the correct answer. To show that this is true, we need to prove that it will never fail on any input. We often use a proof-by-contradiction approach to do this. As a first step to analyzing any algorithm, try to identify a counter example for the claim that the algorithm is correct. If you can prove that no such counter example exists, you are done.

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, $S = n(n+1)/2 + x$,
- And $x = S - n(n+1)/2$.

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 The Punch Line

Designing good algorithms matters. As computer scientists, we aim to

1. design *correct* solutions to problems (does not fail on any input),
2. design *elegant* solutions to problems (simplicity is a virtue),
3. design *efficient* solutions (use as few resources as possible), and
4. provide performance guarantees where possible (you have thought carefully about the worst possible behavior).

Achieving these goals is not always easy. Many of the algorithms we'll see in this course are clever, but very few are the most efficient solution possible.

Generally, the first solution that comes to mind will not be either correct or the most efficient. But, if you think carefully about the mathematical structure of the problem, about the inputs that produce bad behavior and apply the tools you learn in this class, you can almost always do better. Your goal for this class should be to sharpen your ability to do these things.

2 Analyzing algorithms

Designing and analyzing algorithms is not a formulaic domain. It requires careful thinking and some degree of cleverness. The goal of the class is to equip you with a larger and more powerful set of tools that you can use to attack novel problems. As a graduate-level class, the pace will be faster, the scope broader and the mathematics deeper than an undergraduate algorithms class. However, much of the material is related.

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 types (chars, ints, doubles, pointers, etc.) take up unit $O(1)$ space.

2.2 Worst-, average- and best-case analysis

To provide rigorous *guarantees* of an algorithm's performance, we will typically consider the algorithm's *worst possible* behavior, in terms of resource usage (time & space). Identifying the algorithm input that produces the worst possible performance is a key part of this class. By the end of the class, if I describe an algorithm to you, you should be comfortable identifying these cases and using them to argument mathematically about the correctness and efficiency of the algorithm. This is the pessimistic or *adversarial* approach to algorithm analysis. Surprisingly, in many cases, we'll still be able to get fairly good bounds, and because it's a worst-case analysis, reality can not be any worse.

Alternatively, we may consider the *average case*, which is sometimes significantly better than the worst case, but not always. (Note: the term "average" case is meaningless without some reference to a probability distribution of inputs.) *Best case* analysis is inherently optimistic, and thus is not typically useful for providing algorithmic guarantees.

2.3 Asymptotic analysis

Unless otherwise specified, we will care only about the asymptotic behavior of algorithms. This approach often simplifies our work 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 is 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 On your own

1. Read Chapters 1, 2 and 3 in CLRS.
2. Problem Set 1 available now on class website; due Thursday, January 16.