INTRODUCTION TO ALGORITHMS

GLENN SUN
OLGA RADKO MATH CIRCLE ADVANCED 2

NOVEMBER 15, 2020

Over the next two weeks, we'll explore the background and meaning of the P vs. NP problem, one of the most important open problems in mathematics. If you can resolve the P vs. NP problem, not only will you be famous for the rest of your life, but you will also immediately get a million dollars!

At a high level, the P vs. NP problem is about whether or not fast algorithms exist to solve a certain class of problems. We'll start today by talking about algorithms in general and what it means to be fast, and next week we'll talk about the classes of problems called P and NP.

An *algorithm* is just a set of unambiguous instructions to solve a problem. For example, consider the problem of finding the largest number from a list of numbers. That is, if you are given the list $(9, 9, 222, -10, 3)$, you are supposed to output 222. To have a correct algorithm, your instructions have to work *in general*, no matter what the list is.

For example, if we call the input to the above problem $(x_1, \ldots, x_n)$, the following could work as an algorithm:

```
1 m ← x_1        // Remember a number called m and set its value to x_1
2 for each x_i in (x_1, ..., x_n) do
3 |  if x_i > m then
4 |  |  m ← x_i    // Update the value of m to whatever x_i currently is
5 return m
```

Exercise 1. Explain why the above algorithm works.

Here are some of the things that a computer can do. Try to use these in your algorithms:

- Read, memorize, and update numbers
- Add, subtract, multiply, and divide numbers
- Compare numbers (less than/greater than/equal to)
- Do something if something else is true
- Do something for every element in a list (this is called a loop)

Exercise 2. Brainstorm different algorithms to solve the following problems. Don't worry about how fast or slow your method is, just that it is unambiguous and works. Can you also explain why your algorithms always produce the correct output?

1. Given a list of numbers $(x_1, \ldots, x_n)$, find the range of the list, that is, the maximum value minus the minimum value.
2. Given two lists of numbers $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$, determine if the two lists have any numbers in common.
3. (Challenge) Given a list of numbers $(x_1, \ldots, x_n)$, output a list with the same numbers but sorted from smallest to biggest.

Now let's talk about the amount of time computers take to run your algorithms. You can assume that almost everything takes one unit of time to do, except loops. The amount of time taken by a loop is the the amount of time spent performing one iteration, times the number of iterations of the loop.

When we analyze the running time of an algorithm, we typically analyze the *worst case running time*. This means that if the algorithms uses conditional statements like "if", then some steps might be skipped for certain inputs, but when we analyze the running time, we consider the longest possible time on any input of length n.

For example, in the example algorithm to compute the maximum number in a list, lines 1 and 5 are both executed once, lines 2 and 3 are both executed exactly n times, and line 4 is inside an "if" clause, so it is executed up to n times, depending on the input. In this case, we say that the running time of the algorithm is $2 + 3n$.

Exercise 3. What is the worst case running time for each of your algorithms from Exercise 2, in terms of n, where n is the length of the input list(s)?

You probably noticed that analyzing the running time of algorithms in so much detail is a laborious task, and it is sometimes ambiguous too, since whether an operation takes 1 unit of time or 2 units of time or 10 units of time often just depends on the computer it's running on. What is important is that those basic operations take an *constant* amount of time. So we introduce a coarser way to describe how long algorithms run that hides things like constants, in order for us to talk about running time in a more natural way.

Definition 1. A function $f(n)$ is said to be $O(g(n))$ (pronounced "big-oh of $g(n)$" or "order $g(n)$") if there exists N and c such that

$$\text{if } n > N, \text{ then } |f(n)| \leq c|g(n)|.$$

In other words, when n is big, $f(n)$ grows like a multiple of $g(n)$ (or grows slower).

Exercise 4 (Challenge, requires calculus)**.** Show that if there exists a constant a such that

$$\lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| \leq a,$$

then $f(n)$ is $O(g(n))$. But also, show that even if the limit does not exist, it is still possible that $f(n)$ is $O(g(n))$. (If you take advanced mathematics classes in college, you might learn that there is another kind of limit called lim sup, pronounced lim-soup, that always exists and makes this into an if and only if statement.)

Exercise 5. Fill in this following table with "yes" or "no" for whether or not the function on the left is any of the orders on top. (Hint: Graph each function. Also, $\lceil x \rceil$ means rounding x up to the nearest integer.)

	$O(1)$	$O(\log n)$	$O(\sqrt{n})$	$O(n)$	$O(n^2)$	$O(2^n)$	$O(n!)$
35							
$5n^2 + 2$							
3^n							
$\lceil \log(n) \rceil$							
$n + 2\sqrt{n}$							

Exercise 6. For each of the algorithms you came up with in Exercise 2, write the running time as big-O of a simple function, such as the functions in the top row in Exercise 5. Try to use the *tightest* simple function possible. For example, if your algorithm runs in $O(n)$ time, don't write $O(n^2)$.

Exercise 7. Show that with big-O notation, the base of a logarithm doesn't matter. That is, if $f(n)$ is $O(\log_a(n))$ for some $a > 1$, then $f(n)$ is also $O(\log_b(n))$ for all $b > 1$.

Exercise 8. The higher/lower game works as follows: Your friend picks a number between 1 and n and doesn't tell you what the number is (but you know n). Your job is to figure out what the number is. You can only ask your friend questions of the form "Is your number higher than, lower than, or equal to k?" for different numbers k. Show that there is an algorithm to find the number in $O(\log n)$ steps.