Algorithms

or: The Unofficial Notes on the Georgia Institute of Technology's **CS6515**: *Graduate Algorithms*

George Kudrayvtsev
george.k@gatech.edu

Last Updated: January 7, 2020

PART I

Notes

Preface

> *I read that Teddy Roosevelt once said, "Do what you can with what you have where you are." Of course, I doubt he was in the tub when he said that.*
>
> — Bill Watterson, *Calvin and Hobbes*

Before we begin to dive into all things algorithmic, I'll enumerate a few things I do in this notebook to elaborate on concepts:

- An item that is **highlighted like this** is a "term;" this is some vocabulary that will be used and repeated regularly in subsequent sections. I try to cross-reference these any time they come up again to link back to its first defined usage; most mentions are available in the Index.
- An item in a maroon box, like...

Boxes: A Rigorous Approach

... this often represents fun and interesting asides or examples that pertain to the material being discussed. They are largely optional, but should be interesting to read and have value, even if it's not immediately rewarding.

- An item in a blue box, like...

Quick Maffs: Proving That the Box Exists

... this is a mathematical aside; I only write these if I need to dive deeper into a concept that's mentioned in lecture. This could be proofs, examples, or just a more thorough explanation of something that might've been "assumed knowledge" in the text.

I also sometimes include margin notes like the one here (which just links back here) that reference content sources so you can easily explore the concepts further. Linky

Dynamic Programming

The **dynamic programming** (commonly abbreviated as DP to make undergraduates giggle during lecture) problem-solving technique is a powerful approach to creating efficient algorithms in scenarios that have a lot of repetitive data.

A classic toy example that we'll start with to demonstrate the power of dynamic programming is a series of increasingly-efficient algorithms to compute the **Fibonacci sequence**:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

In general, $F_n = F_{n-1} + F_{n-2}$, with the exceptional base-case that $F_n = n$ for $n \leq 1$. The simplest, most naïve algorithm for calculating the n^{th} Fibonacci number just recurses on each F_m as needed:

Algorithm 1.1: Fib1 (n), a naïve, recursive Fibonacci algorithm.

Input: An integer $n \geq 0$.
Result: F_n

if $n \leq 1$ **then**
  return n
end
return Fib1 $(n-1)$ + Fib1 $(n-2)$

Notice that each branch of the recursion tree operates independently despite them doing almost identical work: we know that to calculate F_{n-1} we need F_{n-2}, but we are also calculating F_{n-2} separately for its own sake… That's a lot of extra work that increases *exponentially* with n!

Wouldn't it be better if we kept track of the Fibonacci numbers that we generated as we went along?

PART II

Homework Walkthroughs

This part of the notes is dedicated to walking through the homework assignments for this course. Everyone taking the course should have the PDFs readily available to them, so this will only feature the solutions (with allusions to the instruction).

Homework #0

This diagnostic assignment should be relatively easy to follow by anyone who has taken an algorithms course before. If it's not fairly straightforward, I highly recommend putting in extra prerequisite effort now rather than later to get caught up on the "basics."

PART III

Additional Assignments

This part of the notes is dedicated to walking through the homework assignments for U.C. Berkeley's CS170 (*Efficient Algorithms and Intractable Problems*) which uses the same textbook. I specifically reference the assignments from spring of 2016 because that's the last time Dr. Umesh Vazirani (an author of the textbook, *Algorithms*) appears to have taught the class. Though the course follows a different path through the curriculum, I believe the rigor of the assignments will help greatly in overall understanding of the topic.

PDFs of the homework assignments are available on my website, where I also may sometimes walk through certain problems in greater depth. The problems themselves are heavily inspired by the problems from *Algorithms* itself.

Homework #1

CS 170: Homework #1

Don't forget that in computer science, our logarithms are always in base-2! That is, $\log n = \log_2 n$. Now recall the definition of big-O:

> *Given two functions, f and g, we can say that $f = O(g)$ if:*
>
> $\exists c \in \mathbb{N}$ such that $f \leq c \cdot g$
>
> *That is, there exists some constant c that makes g always bigger than f.*

Note that if $g = O(f)$, we can also say $f = \Omega(g)$. And if both are true (that is, if $f = O(g)$ *and* $f = \Omega(g)$) then $f = \Theta(g)$. With that in mind...

3.1 Compare Growth Rates

I'll only do a handful of these since they should generally be pretty straightforward and mostly rely on recalling esoteric algebraic manipulation rules.

(a) Given

$$f(n) = n^{1/2}$$
$$g(n) = n^{2/3}$$

we can determine that $\boxed{f = O(g)}$ because:

$$f \overset{?}{\leq} c \cdot g$$
$$\frac{f}{g} \overset{?}{\leq} c$$
$$\frac{n^{1/2}}{n^{2/3}} = n^{-1/6} \leq 1 \qquad \forall n \in \mathbb{N}$$

That is, there are no natural numbers that would make this fraction $\frac{1}{n^{1/6}}$ larger than $c = 1$. Note that we don't actually need this level of rigor; we can also just say: $f = O(g)$ because it has a smaller exponent.

(b) Given

$$f(n) = n \log n$$
$$g(n) = (\log n)^{\log n}$$

Considering the fact that $g(n)$ is an exponential while $f(n)$ is a polynomial, we should intuitively expect $\boxed{f = O(g)}$. By expanding the logarithm with a quick undo-redo (that is, by adding in the base-2 no-op), we can make this even more evident:

$$\begin{aligned}
g(n) &= (\log n)^{\log n} \\
&= 2^{\log\left((\log n)^{\log n}\right)} && \text{recall that applying the base to a logarithm is a no-op, so: } b^{\log_b x} = x \\
&= 2^{\log n \cdot \log\log n} && \text{we can now apply the power rule: } \log x^k = k \log x \\
&= 2^{\log n^{\log\log n}} && \text{then the power rule of exponents: } x^{ab} = (x^a)^b \\
&= n^{\log\log n} && \text{and finally undo our first step}
\end{aligned}$$

Very clearly, n^m will grow *much* faster than some nm.

3.2 Geometric Growth

We need to prove the big-Os of geometric series under all conditions:

$$\sum_{i=0}^{k} c^i = \begin{cases} \Theta(c^k) & \text{if } c > 1 \\ \Theta(k) & \text{if } c = 1 \\ \Theta(1) & \text{if } c < 1 \end{cases}$$

We'll break this down piece-by-piece. For shorthand, let's say $f(k) = \sum_{i=0}^{k} c_i$.

Case 1: $c < 1$**.** Let's start with the easy case. Our claim $f = \Theta(1)$ means:

$$\exists m \text{ such that } f(k) \leq m \cdot 1 \qquad \textbf{and}$$
$$\exists n \text{ such that } f(k) \geq n \cdot 1$$

Remember from calculus that an infinite geometric series for these conditions converges readily:

$$\sum_{i=0}^{\infty} c^i = \frac{1}{1-c}$$

This is a hard-cap on our series (since $k \leq \infty$), so we can confidently say that $f = O(1)$ for $m = \frac{1}{1-c}$. We also have to show the inverse, too, which is trivial: just let $n = 1$. Since the sum can only get bigger with k, the smallest is $f(k = 0) = 1$.

Case 2: $c = 1$**.** This case is actually even easier... Notice that the series' sum is always just k:

$$f(k) = \sum_{i=0}^{k} c^i = \underbrace{1 + 1 + \ldots + 1}_{k \text{ times}} = k$$

meaning our claim of $f = \Theta(k)$ is self-evident.

Case 3: $c > 1$**.** The final case results in an increasing series; the formula for the sum for k terms is given by:

$$f(k) = \sum_{i=0}^{k} c^i = \frac{1 - c^{k+1}}{1 - c} \tag{3.1}$$

Our claim is that this is capped by $f(k) = \Theta(c^k)$. We know that obviously the following holds true:[1]

$$c^{k+1} > c^{k+1} - 1 > c^k$$

Notice that this implies $c^{k+1} - 1 = O(c^{k+1})$, and $c^{k+1} - 1 = \Omega(c^k)$. That's really close to what we want to show... What if we divided the whole thing by $c - 1$?

$$\frac{c^{k+1}}{c-1} > \frac{c^{k+1} - 1}{c-1} > \frac{c^k}{c-1}$$

Now notice that the middle term is $f(k)$! Thus,

$$\frac{c}{c-1}c^k > f(k) > \frac{1}{c-1}c^k$$

Meaning it *must* be the case that $f = \Theta(c^k)$ because we just showed that it had c^k as both its upper and lower bounds (with different constants $m = \frac{c}{c-1}, n = \frac{1}{c-1}$).

Quick Maffs: Deriving the geometric summation.

We want to prove (3.1):

$$f(k) = \sum_{i=0}^{k} c^i = \frac{1 - c^{k+1}}{1 - c}$$

The proof is a simple adaptation of this page for our specific case of $i = 0$

[1] This is true for $k > 0$, but if $k = 0$, then we can show $f(0) = \Theta c^0$ just like in *Case 2*.

and $a_0 = 1$. First, note the complete expansion of our summation:

$$\sum_{i=0}^{k} c^i = 1 + c + c^2 + \ldots + c^k$$
$$= c^k + c^{k-1} + \ldots + c^2 + c + 1$$

or, more conventionally for polynomials...

A basic property of polynomials is that dividing $x^n - 1$ by $x - 1$ gives:

$$\frac{x^n - 1}{x - 1} = x^{n-1} + x^{n-2} + \ldots + x^2 + x + 1$$

By reverse-engineering this, we can see that our summation can be the result of a similar division:

$$\frac{c^{k+1} - 1}{c - 1} = c^k + c^{k-1} + \ldots + c^2 + c + 1$$ ■

3.3 Recurrence Relations

We'll be solving the given recurrence relations using the Master theorem. Recall the Master theorem:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where n is the size of the input problem, a is the number of subproblems in the recursion, and b is the factor by which the subproblem size is reduced in each recursive call.

(a) $T(n) = 3T(n/4) + 4n$

Index of Terms