Algorithms (I)

Yijia Chen
Shanghai Jiaotong University

Chapter 0. Prologue

Johann Gutenberg

Johann Gutenberg (c. 1398 – February 3, 1468)

In 1448 in the German city of Mainz a goldsmith named **Johann Gutenberg** discovered a way to print books by putting together movable metallic pieces.

Two ideas changed the world

Because of the typography, literacy spread, the Dark Ages ended, the human intellect was liberated, science and technology triumphed, the Industrial Revolution happened.

Many historians say we owe all this to typography. But others insist that the key development was not typography, but algorithms.

Decimal system

Gutenberg would write the number 1448 as MCDXLVIII.

How do you add two Roman numerals? What is

$$MCDXLVIII + DCCCXII?$$

The decimal system was invented in India around AD 600.

Using only 10 symbols, even very large numbers could be written down compactly, and arithmetic could be done efficiently on them by following elementary steps.

Al Khwarizmi

Al Khwarizmi (c.780 – c.850)

In the 12th century, Latin translations of his work on the Indian numerals, introduced the decimal system to the Western world. (Source: Wikipedia)

Algorithms

Al Khwarizmi laid out the basic methods for

- adding,
- multiplying,
- dividing numbers,
- extracting square roots,
- calculating digits of π.

These procedures were precise, unambiguous, mechanical, efficient, correct. They were algorithms, a term coined to honor the wise man after the decimal system was finally adopted in Europe, many centuries later.

Leonardo Fibonacci

Leonardo Fibonacci (c. 1170 – c. 1250)

Fibonacci helped the spread of the decimal system in Europe, primarily through the publication in the early 13th century of his Book of Calculation, the Liber Abaci. (Source: Wikipedia)

Fibonacci sequence

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

Formally

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0. \end{cases}$$

What is F_{100} or F_{200}?

An exponential algorithm

```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n - 1) + fib1(n - 2)
```

Three questions about an algorithm

1. Is it correct?
2. How much time does it take, as a function of n?
3. And can we do better?

The first question is moot here, as this algorithm is precisely Fibonacci's definition of F_n.

How much time

Let $T(n)$ be the number of computer steps needed to compute `fib1(n)`.

For $n \leq 1$

$$T(n) \leq 2$$

For $n > 1$

$$T(n) = T(n-1) + T(n-2) + 3.$$

By induction, for all $n \in \mathbb{N}$

$$T(n) \geq F_n$$

hence *exponential* in n.

Why exponential is bad?

$$T(200) \geq F_{200} \geq 2^{138} \approx 2.56 \times 10^{42}.$$

At this time, the fastest computer in the world is the Tianhe-1A system at the National Supercomputer Center in Tianjin. Its speed is

$$2.57 \times 10^{15}$$

steps per second.

Thus to compute F_{200} Tianhe-1A needs roughly

$$10^{27} \text{ seconds} \geq 10^{22} \text{ years.}$$

What about Moore's law?

Computer speeds have been doubling roughly every 18 months.

The running time of `fib1(n)` is proportional to

$$2^{0.694n} \approx (1.6)^n.$$

Thus, it takes 1.6 times longer to compute F_{n+1} than F_n.

So if we can reasonably compute F_{100} with this year's technology, then next year we will manage F_{101}. And the year after, F_{102}. And so on:

just one more Fibonacci number every year!

Such is the curse of exponential time.

Recall the three questions about an algorithm:

1. Is it correct?
2. How much time does it take, as a function of n?
3. And can we do better?

Now we know `fib1`(n) is correct and inefficient, so can we do better?

A polynomial algorithm

```
function fib2(n)
if n = 0 return 0
create an array f[0...n]
f[0] = 0, f[1] = 1
for i = 2...n:
    f[i] = f[i - 1] + f[i - 2]
return f[n]
```

The correctness of `fib2(n)` is trivial.

How long does it take? The inner loop consists of a single computer step and is executed $n-1$ times. Therefore the number of computer steps used by `fib2` is linear in n.

A more careful analysis

We have been counting the number of basic computer steps executed by each algorithm and thinking of these basic steps as *taking a constant amount of time*.

It is reasonable to treat addition as a single computer step if small numbers are being added, e.g., 32-bit numbers.

But the nth Fibonacci number is about $0.694n$ bits long, and this can far exceed 32 as n grows.

Arithmetic operations on arbitrarily large numbers cannot possibly be performed in a single, constant-time step.

A more careful analysis (cont'd)

We will see in Chapter 1 that the addition of two n-bit numbers takes time roughly *proportional to n*.

Thus `fib1`, which performs about F_n additions, uses a number of basic steps roughly *proportional to nF_n*.

Likewise, the number of steps taken by `fib2` is *proportional to n^2*, still polynomial in n and therefore exponentially superior to `fib1`.

But can we do even better than `fib2`?

Indeed we can: see Exercise 0.4.

Counting the number of steps

We've seen how sloppiness in the analysis of running times can lead to unacceptable inaccuracy.

But it is also possible to be too precise to be useful.

Expressing running time in terms of basic computer steps is already a simplification. The time taken by one such step depends crucially on the particular processor, etc. Accounting for these architecture-specific details is too complicated and yields a result that does not generalize from one computer to the next.

Counting the number of steps (cont'd)

It therefore makes more sense to seek an *machine-independent* characterization of an algorithm's efficiency.

To this end, we will always express running time by counting the number of basic computer steps, as *a function of the size of the input*.

Moreover, instead of reporting that an algorithm takes, say, $5n^3 + 4n + 3$ steps on an input of size n, it is much simpler to leave out lower-order terms such as $4n$ and 3 (which become insignificant as n grows).

Even the detail of the coeffcient 5 in the leading term (computers will be five times faster in a few years anyway), and just say that the algorithm takes time $O(n^3)$ (pronounced "big oh of n^3").

Big-O notation

$f(n)$ and $g(n)$ are the running times of two algorithms on inputs of size n.

> *Let $f(n)$ and $g(n)$ be functions from positive integers to positive reals. We say $f = O(g)$ (which means that "f grows no faster than g") if there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$.*

$f = O(g)$ is a very loose analog of "$f \leq g$." It differs from the usual notion of because of the constant c, so that for instance $10n = O(n)$.

Why disregard the constant?

We are choosing between two algorithms: One takes $f_1(n) = n^2$ steps, while the other takes $f_2(n) = 2n + 20$ steps.

Which is better? The answer depends on n:

- If $n \leq 5$, then $f_1(n) \leq f_2(n)$.
- If $n > 5$, then $f_2(n) < f_1(n)$.

f_2 scales much better as n grows, and therefore it is superior.

This superiority is captured by the big-O notation: $f_2 = O(f_1)$:

$$\frac{f_2(n)}{f_1(n)} = \frac{2n + 20}{n^2} \leq 22$$

for all $n \in \mathbb{N}$. On the other hand, $f_1 \neq O(f_2)$, since *the ratio*

$$\frac{f_1(n)}{f_2(n)} = \frac{n^2}{2n + 20}$$

can get arbitrarily large.

Why disregard the constant? (cont'd)

Recall $f_1(n) = n^2$ and $f_2(n) = 2n + 20$.
Suppose we have a third algorithm which uses

$$f_3(n) = n + 1$$

steps. Is this better than f_2? Certainly, but only by a constant factor.

The discrepancy between f_2 and f_3 is tiny compared to the huge gap between f_1 and f_2. In order to stay focused on the big picture, we treat functions as equivalent if they differ only by *multiplicative constants*.

Returning to the definition of big-O, we see that $f_2 = O(f_3)$:

$$\frac{f_2(n)}{f_3(n)} = \frac{2n + 20}{n + 1} \leq 20$$

and $f_3 = O(f_2)$ with $c = 1$.

Other similar notations

Just as $O(\cdot)$ is an analog of $\leq$, we also define analogs of $\geq$ and $=$ as follows:

$$f = \Omega(g) \text{ means } g = O(f)$$
$$f = \Theta(g) \text{ means } f = O(g) \text{ and } f = \Omega(g)$$

Recall $f_1(n) = n^2$, $f_2(n) = 2n + 20$ and $f_3(n) = n + 1$. Then

$$f_2 = \Theta(f_3) \quad \text{and} \quad f_1 = \Omega(f_3).$$

Some simple rules

Big-O notation lets us focus on the big picture. When faced with a complicated function like $3n^2 + 4n + 5$, we just replace it with $O(f(n))$, where $f(n)$ is as simple as possible.

In this particular example we'd use $O(n^2)$, because the quadratic portion of the sum dominates the rest.

Here are some commonsense rules that help simplify functions by omitting dominated terms:

1. Multiplicative constants can be omitted: $14n^2$ becomes n^2.
2. n^a dominates n^b if $a > b$: for instance, n^2 dominates n.
3. Any exponential dominates any polynomial: 3^n dominates n^5 (it even dominates 2^n).
4. Likewise, any polynomial dominates any logarithm: n dominates $(\log n)^3$. This also means, for example, that n^2 dominates $n \log n$.

Warning

Programmers and algorithm developers are very interested in constants and would gladly stay up nights in order to make an algorithm run faster by a factor of 2.

But understanding algorithms at the level of this book would be impossible without the simplicity afforded by big-O notation.