What is an Algorithm?

Solon P. Pissis

CWI & VU, Amsterdam

October 25, 2024

Algorithms

- What is an **algorithm**?
- Informally, an **algorithm** is a well-defined finite set of rules that specifies a sequential series of elementary operations to be applied to some data called the *input*, producing after a finite amount of time some data called the *output* [1].
- Word derives from the Persian mathematician Abū Abdallāh Muḥammad ibn Mūsā al-Khwārizmī (c. 780 - c. 850).
- The oldest non-trivial algorithm, that has survived to the present day, is the Euclidean algorithm, named after the Greek mathematician Euclid (fl. 300 BC), for computing the greatest common divisor of two natural numbers.
- To solve a *computational problem* we usually **design** an algorithm, and then **analyse** its performance.

Algorithms

- What is a **computational problem**?
- The 15th century Italian mathematician Leonardo Fibonacci is known for his famous sequence of numbers

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

  each the sum of its two immediate predecessors.
- More formally,

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

- But **what** is the precise value of F_{100} , or of F_{200} ? Fibonacci himself would surely have wanted to know such things.
- To **answer**, we need to **design** an algorithm for computing the nth Fibonacci number.

Algorithms

- An idea is to slavishly implement the recursive definition of F_n.

Algorithm 1 FIB1

```
procedure FIB1(n)
    if n = 0 then
        return 0
    end if
    if n = 1 then
        return 1
    end if
    return FIB1(n − 1) + FIB1(n − 2)
end procedure
```

- There are three questions we always ask about it:
  1. Is it correct?
  2. How much time does it take, as a *function* of n?
  3. And can we do better?

Algorithms

- The first question is moot here, as this algorithm is precisely Fibonacci's definition of F_n.
- Let $T(n)$ be the number of computer steps needed to compute FIB1(n); what can we say about this function? — **Analyse**
- For starters, if n is less than 2, the procedure halts almost immediately, after just a couple of steps.

$$T(n) \leq 2 \text{ for } n \leq 1.$$

- For larger values of n, there are two recursive invocations of FIB1: one taking time $T(n-1)$ and one taking time $T(n-2)$; plus three computer steps (checks on the value of n and a final addition).

$$T(n) = T(n-1) + T(n-2) + 3 \text{ for } n > 1.$$

Algorithms

- Compare this to the recurrence relation for F_n: we immediately see that $T(n) \geq F_n$.
- This is very bad news: the running time of the algorithm grows as fast as the Fibonacci numbers!
- $T(n)$ is *exponential* in n, which implies that the algorithm is impractically slow except for very small values of n.
- That is, to compute F_{200}, algorithm FIB1 executes $T(n) \geq F_{200} \geq 2^{138}$ elementary computer steps.
- Even on a very fast machine, FIB1(200) would take more than 2^{92} seconds.
- If we start the computation today, it would still be going long after the sun turns into a red giant star.
- The algorithm is *correct*, but can we do *better*?

Algorithms

- A more sensible scheme: store the intermediate results—the values $F_0, F_1, \ldots, F_{n-1}$—as soon as they become known.

Algorithm 2 FIB2

```
procedure FIB2(n)
    if n = 0 then
        return 0
    end if
    f[0...n] ← 0
    f[0] ← 0
    f[1] ← 1
    for i = 2, ..., n do
        f[i] ← f[i − 1] + f[i − 2]
    end for
    return f[n]
end procedure
```

- The correctness is self-evident – directly uses the definition.
- How long does it take?

Algorithms

- The for loop consists of a single computer step and is executed $n - 1$ times.
- Therefore the number of computer steps used by FIB2(n) is linear in n.
- From exponential we are down to *polynomial*, a huge breakthrough in running time.
- It is now perfectly reasonable to compute F_{200} or even $F_{200,000}$.

Algorithms

- 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 coefficient 5 in the leading term—computers will be five times faster in a few years anyway.
- We just say that the algorithm takes time $O(n^3)$ (pronounced "big oh of n^3").
- We define this notation precisely by thinking of $f(n)$ and $g(n)$ as the running times of two algorithms on inputs of size n.

Algorithms

Definition

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 $f(n) \leq c.g(n)$.

Asymptotic Complexity

- Saying $f = O(g)$ is a very loose analog of $f \leq g$.
- It differs from the usual notion of $\leq$ due to the constant c.
- This constant also allows us to disregard what happens for small values of n.
- Suppose $f_1(n) = n^2$ and $f_2(n) = 2n + 20$. Which is better?
- Well, this depends on the value of n. For $n \leq 5$, f_1 is smaller; thereafter, f_2 is the clear winner.
- In this case, f_2 scales much better as n grows, and therefore it is *superior*.

Asymptotic Complexity

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

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

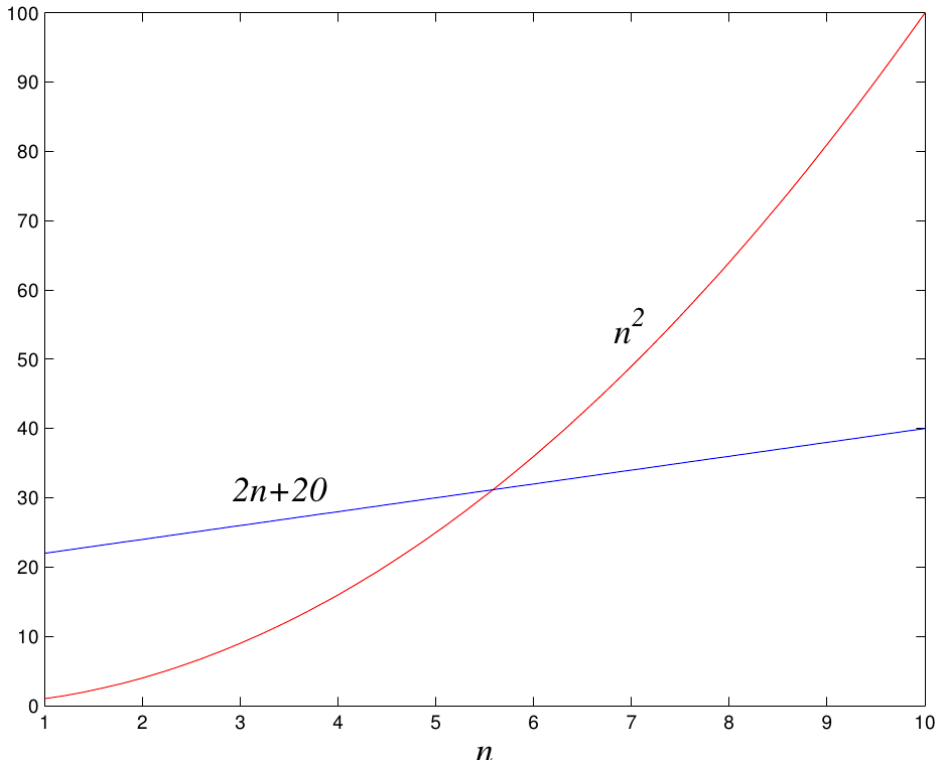
for all n.

- 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, and so no constant c will make the definition work.

Algorithms



Asymptotic Complexity

- Suppose $f_1(n) = n^2$, $f_2(n) = 2n + 20$, and $f_3(n) = n + 1$?
- We see that $f_2 = O(f_3)$, because

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

  but also $f_3 = O(f_2)$, this time with $c = 1$.
- Just as $O(.)$ is an analog of $\leq$ we can also define analogs of $\geq$ and $=$ as follows.

Asymptotic Complexity

Definition

Let $f(n)$ and $g(n)$ be functions from positive integers to positive reals. $f = \Omega(g)$ means $g = O(f)$.

Definition

Let $f(n)$ and $g(n)$ be functions from positive integers to positive reals. $f = \Theta(g)$ means $f = O(g)$ and $f = \Omega(g)$.

Example

$2n + 20 = \Theta(n + 1)$ and $n^2 = \Omega(n + 1)$.

Asymptotic Complexity

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

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

Asymptotic Complexity

Different types of **analysis** for computational resources (e.g. time, space, communication, etc.):

- **Worst case**: the biggest amount of some computational resource (e.g. longest running time) required by the algorithm for *any* input.
- Best case: the smallest amount of some computational resource required by the algorithm for *any* input.
- Average case: the amount of some computational resource required by the algorithm averaged over *all possible* inputs.

Bibliography (Slides based on) I


Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani.
Algorithms.
McGraw-Hill, 2008.