Writing and Comparing Algorithms

2

Imagine you are a student at the University of Bristol; lectures for the day are finished, and you fancy a drink to celebrate. How do you get from the Computer Science Department to the Student Union? Easy! Just ask Google Maps:

http://maps.google.com/?saddr='bristol+bs8+1ub'&daddr='bristol+bs8+1ln'

What comes back is a fancy map plus a list of directions which resemble the following:

1. Start at Bristol, BS8 1UB, UK.
2. Head west on B4051 toward A4018; continue to follow B4051.
3. Continue on A4018.
4. Continue on B3129; go through two roundabouts; destination will be on the left.
5. Finish at Bristol, BS8 1LN, UK.

The directions have some features worth discussing in more detail. Each **line** in the directions represents a static description of some active **step** we should perform while following them. The lines have to be followed, or **processed**, in order: we start at the first step and once that is complete, move on to the next one. If one of the steps is missed out for example, or we start at the third step instead of the first, the directions do not work. The directions are (fairly) unambiguous. It may help to know that the road B4051 is more usually called Park Street, but line #4 is not "go through *some* roundabouts"; we should go through *exactly* two roundabouts. This means it is always clear how to process each line (i.e., exactly what steps to perform): we never become confused because there is not enough information and we do not know what to do.

2.1 Algorithms

As a result of the features described above, our directions at least loosely satisfy the definition of an **algorithm** [1]: they are simply an abstract description of how to solve a problem. Of course in this case the problem is helping someone get from one place to another, but we might just as easily write an algorithm to solve other problems as well.

D. Page, N. Smart, *What Is Computer Science?*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-04042-4_2,

2.1.1 What Is an Algorithm?

It is unusual to enforce any strict rules about *how* we should write an algorithm down: as long as it makes sense to whoever is reading it, we can use whatever format we want. This often means using **pseudo-code**, literally a "sort of" program [11]. An algorithm can have inputs, which we name, and can provide some outputs; we say the algorithm takes (or is given) some arguments as input, and returns (or produces) a result as output. Based on this, imagine we write the following algorithm:

```
1 algorithm FERMAT-TEST(n) begin
2 |   Choose a random number a between 2 and n − 1 inclusive
3 |   Compute d = gcd(a, n), the greatest common divisor of a and n
4 |   If d ≠ 1 then return false as the result
5 |   Compute t = a^(n−1) (mod n)
6 |   If t = 1 then return true as the result, otherwise return false
7 end
```

This looks more formal, but there is no magic going on: the algorithm is simply a list of five lines we can process. Note that we have numbered each of the lines so we can refer to them. In line #1 we name the algorithm FERMAT-TEST and show that it accepts a single argument called n as input; in lines #2 to #6 we list the directions themselves. The lines #4 and #6 are a bit special since they can produce output from the algorithm.

We **invoke** (or use) FERMAT-TEST by giving a concrete value to each of the inputs. For example, if we write

FERMAT-TEST(221)

basically what we mean is "follow the algorithm FERMAT-TEST, but each time you see n substitute 221 instead". Like the travel directions, invoking FERMAT-TEST means we perform some active step (or steps) for each line in the algorithm:

Step #1 Choose a random number between 2 and 220 inclusive, e.g., $a = 11$.
Step #2 Compute $d = \gcd(11, 221) = 1$.
Step #3 Since $d = 1$, carry on rather than returning a result.
Step #4 Compute $t = 11^{221-1} \pmod{221} = 81$.
Step #5 Since $t \neq 1$, return **false** as the result.

After step #5 the algorithm **terminates**: from the input 221, we have computed the result **false**. What this result means of course depends on the purpose of the algorithm; the purpose of FERMAT-TEST is certainly less clear than the travel directions we started off with!

FERMAT-TEST is actually quite a famous algorithm [7] due to the French mathematician Pierre de Fermat. The purpose is to tell if n, the input, is a **prime** [12] number or not. Lines #2 to #4 ensure we select an a that is **co-prime** [5] to n; this means a and n have no common factors. If we were to pick an a such that a divided n, then clearly n cannot be prime, so this possibility is ruled out before we carry

on. Having computed t, if the result we get after lines #5 to #6 is **false** then n definitely is *not* a prime number; it is composite. On the other hand, if the result is **true** then n *might* be a prime number. The more times we invoke FERMAT-TEST with a given n and get **true** as a result, the more confident we are that n is a prime number. FERMAT-TEST works quite well, except for the so-called **Carmichael numbers** [3] which trick the algorithm: they are composite, but FERMAT-TEST always returns **true**. The smallest Carmichael number is $561 = 3 \cdot 11 \cdot 17$: whatever a it chooses, FERMAT-TEST will always compute $t = 1$ and return **true**.

Try this out for yourself: pick some example values of n, and work through the steps used by FERMAT-TEST to compute a result; you could even use a friend to generate random values of a to make sure you cannot cheat! Using the algorithm, see if you can identify some

1. other Carmichael numbers, or
2. Mersenne primes, which have the special form $n = 2^k - 1$ for an integer value k,

or, get a friend to give you an n (which they *know* is either prime or composite) and try to decide whether or not it is prime.

Different Styles of Structure

There are of course *lots* of ways we can write down the same algorithm. For example, where the inputs and outputs need more explanation (e.g., to explain their type or meaning), it is common to write it in a slightly more verbose but otherwise similar way:

```
1 algorithm FERMAT-TEST begin
    Input: An integer n
    Output: false if n is composite, otherwise true if n is probably prime
2   Choose a random number a between 2 and n − 1 inclusive
3   Compute d = gcd(a, n), the greatest common divisor of a and n
4   If d ≠ 1 then return false as the result
5   Compute t = a^(n−1) (mod n)
6   If t = 1 then return true as the result, otherwise return false
7 end
```

As more of a contrast, Fig. 2.1 shows a *totally* different way to describe the same algorithm. You might find this more natural, or easier to read: this form is called a **flow chart** [9] and can be traced back to the early 1920*s*. Since then, flow charts have become a common way of describing "real life" algorithms such as troubleshooting instructions that tell you what to do when your television breaks down, or for capturing decision making procedures in organisations. The point is, both our written

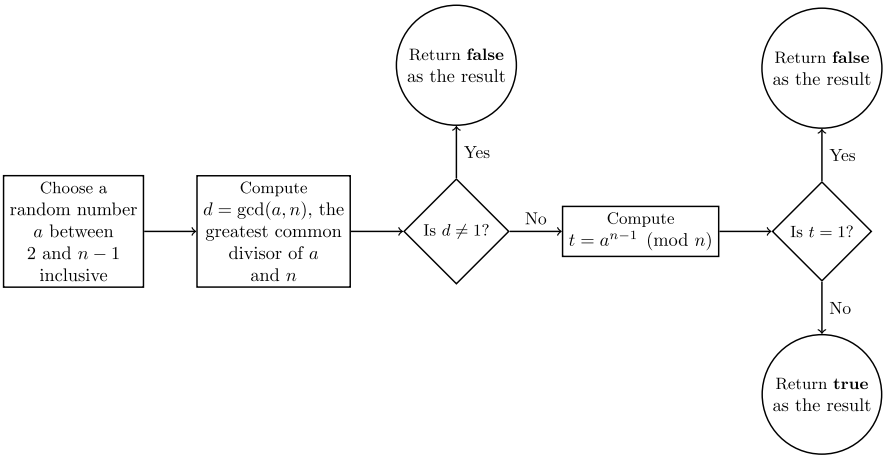

Fig. 2.1 A flow chart description of FERMAT-TEST

and flow chart versions of FERMAT-TEST describe *exactly* the same thing: they are *both* algorithms.

Different Styles of Notation

There is one final, minor issue remaining that we need to be careful about whatever style of algorithm we opt for. As an example, look at the original FERMAT-TEST algorithm again, specifically at lines #3 and #4: both make use of the $=$ (or "equals") symbol. In line #3 we mean "assign the value produced by computing $\gcd(a, n)$ to d" whereas in line #4 we mean "evaluate to **true** if and only if the value d does not equal one". It is reasonable to argue that we might confuse the two meanings. For example, we might mistakenly interpret "compute $t = a^{n-1} \pmod{n}$" as "compute the value **false**" since the value t does not equal $a^{n-1} \pmod{n}$; we have not assigned *any* value to t yet! To prevent this possible ambiguity, Computer Scientists often use

1. the $=$ symbol to mean **equality**, and
2. the $\leftarrow$ symbol to mean **assignment**.

As such, we should really rewrite the algorithm as follows:

```
1 algorithm FERMAT-TEST(n) begin
2 |   Choose a random number a between 2 and n − 1 inclusive
3 |   Compute d ← gcd(a, n), the greatest common divisor of a and n
4 |   If d ≠ 1 then return false as the result
5 |   Compute t ← a^(n−1) (mod n)
6 |   If t = 1 then return true as the result, otherwise return false
7 end
```

2.1.2 What Is not an Algorithm?

Algorithms ≠ Functions

It is tempting to treat algorithms like Mathematical functions. After all, invoking an algorithm *looks* like the use of a Mathematical function; writing SIN(90) uses the function SIN to compute a result using the input 90 for example. Strictly speaking, the difference (or at least one difference) comes down to the idea of **state**.

For Mathematical functions, writing something like $\text{SIN}(x)$ might give different results depending on x, but once we have chosen an x we get the same result *every* time. For example SIN(90) *always* equals 1 so we are entitled to write $\text{SIN}(90) = 1$, or use 1 whenever we see SIN(90). But algorithms are different: they may depend on the context they are invoked in; there may be some state which effects how an algorithm behaves and hence the result it returns. Another way to say the same thing is that algorithms might have **side effects** which alter the state; in contrast, functions are **pure** in the sense they are totally self contained.

FERMAT-TEST demonstrates this fact neatly: line #2 reads "choose a random number a". Since this can be *any* number (as long as it is co-prime to n), it is perfectly possible we might choose a different one each time we invoke FERMAT-TEST. So the first time we invoke

FERMAT-TEST(221)

we might choose $a = 11$ in line #2 as above; we know the result in this case is **false**. However, imagine we invoke the algorithm a second time

FERMAT-TEST(221)

and choose $a = 47$ instead. This time, the result is **true**. So we have got a different result with the same input: the context (represented in this case by whatever we are choosing random numbers with) influences what result we get, not just the input.

Algorithms ≠ Programs

It is tempting to treat algorithms like programs. If you have written any programs before, there is a good chance FERMAT-TEST has a similar look and feel. Usually when we talk about a program, we mean something which is intended to be **executed** by a real computer: a word processor, a web-browser, or something like that. This means a program should be written in a "machine readable" form: each step or action required by a program needs to be something a computer can actually do.

Within FERMAT-TEST for example, we are required to compute $a^{n-1} \pmod{n}$ for some a and n. Would most computers know how to do this? Probably not. The computer might know how to do multiplication, and we might be able to explain to it how to do exponentiation using another algorithm, but it is not reasonable that the computer will know how to do the computation itself.

So in contrast to a "machine readable" program, we have written the FERMAT-TEST algorithm in a far less restrictive "human readable" form. Whereas a program relates to a real computer, you can think of an algorithm as relating to some abstract

or make believe uber-computer which does not have any similar restrictions. The algorithm needs to be **implemented**, by rewriting it in a programming language, before it is ready for execution by a real computer.

2.2 Algorithms for Multiplication

Imagine you are asked to multiply one number x by another number y. You might perform multiplication in your head without thinking about it, but more formally what does multiplication actually mean? If you met someone from a different planet who does not know what multiplication is, how could you describe to them the steps required to compute $x \cdot y$? Clearly the answer is to write an algorithm that they can follow. As a starting point, notice that multiplication is just repeated addition. So for example

$$x \cdot y = \underbrace{x + x + \cdots + x + x}_{y \text{ copies}},$$

which means if we select $y = 6$ then we obviously have

$$x \cdot 6 = x + x + x + x + x + x.$$

Based on this approach, we can write an algorithm that describes how to multiply x by y. We just need a list of careful directions that someone could follow:

```
1 algorithm MULTIPLY-REPEAT(x, y) begin
2 |   t ← 0
3 |   for i from 1 upto y do
4 |   |   t ← t + x
5 |   end
6 |   return t
7 end
```

The basic idea of this algorithm is simple: add together y copies of x, keeping track of the value we have accumulated so far in t. Of course you could argue the algorithm is *much* harder to read and understand because we have replaced this English description with something that looks much more formal. On the other hand, by doing this we have made things more precise. That is, there is no longer any room for someone to misinterpret the description if they know how each **construct** behaves:

- The construct in line #2 is another example of the **assignment** we saw earlier: it assigns a value (on the right) to a name, or variable (on the left). When we write $X \leftarrow Y$, the idea is to set the variable X to a value given by evaluating the expression Y. In this case, the construct $t \leftarrow 0$ sets the variable t to the value 0: every time we evaluate t in some expression after this, we can substitute 0 until t is assigned a new value.

- The construct that starts in line #3 is an example of a **loop**. When we write **for** X **do** Y, the idea is to repeatedly process the block Y for values dictated by X; we say that the construct **iterates** over Y. Our loop is **bounded** because we know how many times we will process Y before we start.

  In this case the block is represented by line #4, and hence $t \leftarrow t + x$ is iterated over for values of i in the range $1 \ldots y$. So basically the loop does the same thing as copying out line #4 a total of y times, i.e.,

$$\left.\begin{array}{l} t \leftarrow t + x \;\}\; i = 1 \\ t \leftarrow t + x \;\}\; i = 2 \\ \qquad\qquad \vdots \\ t \leftarrow t + x \;\}\; i = y \end{array}\right\} \; y \text{ copies}$$

- The construct in line #6 is an example of a **return**. When we write **return** X, the idea is that we evaluate X and return this as the result of the algorithm. In this case, we return the value accumulated in t as the result.

Now that we know the meaning of each line, we can invoke the algorithm and perform the steps required to compute a result. Imagine we select $x = 3$ and $y = 6$ for example; the steps we perform would be something like the following:

Step #1 Assign $t \leftarrow 0$.
Step #2 Assign $t \leftarrow t + x$, i.e., $t \leftarrow 0 + 3 = 3$.
Step #3 Assign $t \leftarrow t + x$, i.e., $t \leftarrow 3 + 3 = 6$.
Step #4 Assign $t \leftarrow t + x$, i.e., $t \leftarrow 6 + 3 = 9$.
Step #5 Assign $t \leftarrow t + x$, i.e., $t \leftarrow 9 + 3 = 12$.
Step #6 Assign $t \leftarrow t + x$, i.e., $t \leftarrow 12 + 3 = 15$.
Step #7 Assign $t \leftarrow t + x$, i.e., $t \leftarrow 15 + 3 = 18$.
Step #8 Return $t = 18$.

After eight steps we reassuringly find the result is $x \cdot y = 18$. The key thing to realise is that fundamentally we are still just following directions: lines in our algorithm might be more formal than "go through two roundabouts", but as long as we know what they mean we can carry out the corresponding steps just as easily.

Another way of looking at what multiplication means is to see that it simply adds another "weight" to the digits that describe y. It might look odd, but imagine we wrote y out as an n-bit binary number, i.e., we write

$$y = \sum_{i=0}^{n-1} y_i \cdot 2^i$$

where clearly each $y_i \in 0, 1$. Then, we could write

$$x \cdot y = x \cdot \sum_{i=0}^{n-1} y_i \cdot 2^i = \sum_{i=0}^{n-1} y_i \cdot x \cdot 2^i .$$

What we are doing is taking each weighted digit of y, and adding a further weight x to the base which is used to express y in. Again, as an example, selecting $y = 6_{(10)} = 110_{(2)}$ we find that we still get the result we would expect to

$$\begin{aligned} y \cdot x &= y_0 \cdot x \cdot 2^0 + y_1 \cdot x \cdot 2^1 + y_2 \cdot x \cdot 2^2 \\ &= 0 \cdot x \cdot 2^0 + 1 \cdot x \cdot 2^1 + 1 \cdot x \cdot 2^2 \\ &= 0 \cdot x + 2 \cdot x + 4 \cdot x \\ &= 6 \cdot x \end{aligned}$$

So far so good. Except that this still looks unpleasant to actually compute. For example we keep having to compute those powers of two to weight the terms. Fortunately, a British mathematician called William Horner worked out a scheme to do this more neatly [10]. Sometimes this is termed Horner's rule (or scheme): bracket the thing we started with in such a way that instead of having to compute the powers of two independently we sort of accumulate them as we go. This is best shown by example:

$$\begin{aligned} y \cdot x &= y_0 \cdot x + 2 \cdot \big(y_1 \cdot x + 2 \cdot \big(y_2 \cdot x + 2 \cdot (0) \big) \big) \\ &= 0 \cdot x + 2 \cdot \big(1 \cdot x + 2 \cdot \big(1 \cdot x + 2 \cdot (0) \big) \big) \\ &= 0 \cdot x + 2 \cdot \big(1 \cdot x + 2 \cdot (1 \cdot x + 0) \big) \\ &= 0 \cdot x + 2 \cdot \big(1 \cdot x + 2 \cdot (1 \cdot x) \big) \\ &= 0 \cdot x + 2 \cdot (1 \cdot x + 2 \cdot x) \\ &= 0 \cdot x + 2 \cdot (3 \cdot x) \\ &= 0 \cdot x + 6 \cdot x \\ &= 6 \cdot x \end{aligned}$$

In much the same way as above, we can write an algorithm that describes how to multiply x by y using this approach:

```
1  algorithm MULTIPLY-HORNER(x, y) begin
2  |  t ← 0
3  |  for i from |y| − 1 downto 0 do
4  |  |  t ← 2 · t
5  |  |  if y_i = 1 then
6  |  |  |  t ← t + x
7  |  |  end
8  |  end
9  |  return t
10 end
```

This algorithm again accepts two inputs called x and y, the two numbers we would like to multiply together, and again produces one output that gives us the result $x \cdot y$. This time, the basic idea is to write y in binary, and process it from left-to-right one digit at a time. In terms of the bracketing, we work from inside outward, applying Horner's rule and keeping track of an accumulated value called t:

- The construct in line #2 is another **assignment**. We have already seen what this means: it sets the variable t to the value 0.

- The construct that starts in line #3 is another **loop**; this is a little different from the previous one we saw. The first difference is that the values of i go downward rather than upward: the block, now represented by lines #4 to #7, is iterated over for i in the range $|y|-1\ldots 0$. That is fine though, we still just copy out lines #4 to #7 a total of $|y|$ times making sure the right values of i are alongside each copy, i.e.,

$$\left.\begin{array}{l}\left.\begin{array}{l}t \leftarrow 2\cdot t\\ \textbf{if } y_i = 1 \textbf{ then } t \leftarrow t + x\end{array}\right\} i = |y|-1\\ \qquad\vdots\\ \left.\begin{array}{l}t \leftarrow 2\cdot t\\ \textbf{if } y_i = 1 \textbf{ then } t \leftarrow t + x\end{array}\right\} i = 1\\ \left.\begin{array}{l}t \leftarrow 2\cdot t\\ \textbf{if } y_i = 1 \textbf{ then } t \leftarrow t + x\end{array}\right\} i = 0\end{array}\right\} |y| \text{ copies}$$

  The second difference is that the block actually uses i within it. That is fine as well: wherever we see an i, we can substitute the right value for that iteration. For example, the last copy from above becomes

$$\begin{array}{l}t \leftarrow 2\cdot t\\ \textbf{if } y_0 = 1 \textbf{ then } t \leftarrow t + x\end{array}$$

  after we substitute in the value $i = 0$.
- The construct that start starts on line #5 is an example of a **condition**; it forms part of the block iterated over by the loop. When we write **if** X **then** Y, the idea is that we perform a test: if X evaluates to **true** then we process the block Y, otherwise we skip it. In this case, we test the i-th bit of y: if $y_i = 1$ then we add x to t in line #6, otherwise t remains unchanged.
- The construct in line #9 is another **return**. We have already seen what this means: it returns t as the result.

This algorithm is slightly more complicated than the last one. However, in the same way as before we know what each line means so we can invoke the algorithm and carry out steps in order to compute a result. Imagine we again select $x = 3$ and $y = 6_{(10)} = 110_{(2)}$; the steps we perform would be something like the following:

Step #1 Assign $t \leftarrow 0$.
Step #2 Assign $t \leftarrow 2\cdot t$, i.e., $t \leftarrow 2\cdot 0 = 0$.
Step #3 Since $y_2 = 1$, assign $t \leftarrow t + x$, i.e., $t \leftarrow 0 + 3 = 3$.
Step #4 Assign $t \leftarrow 2\cdot t$, i.e., $t \leftarrow 2\cdot 3 = 6$.
Step #5 Since $y_1 = 1$, assign $t \leftarrow t + x$, i.e., $t \leftarrow 6 + 3 = 9$.
Step #6 Assign $t \leftarrow 2\cdot t$, i.e., $t \leftarrow 2\cdot 9 = 18$.
Step #7 Since $y_0 = 0$, skip the assignment $t \leftarrow t + x$.
Step #8 Return $t = 18$.

The algorithm has clearly computed the result in a different way (i.e., the steps themselves are different), but we still find that $x\cdot y = 18$ as expected.

Although reading and following invocations of *existing* algorithms is a good start, writing your own algorithms is really the only way to get to grips with this topic.

In Chap. 1 we met the HAMMING-WEIGHT function: it counts the number of elements in a binary sequence that are equal to one. Write an algorithm that can compute HAMMING-WEIGHT(x) for a suitable input sequence x; demonstrate how it does so by listing the steps (similar to above) for an example x.

2.3 Algorithms for Exponentiation

So much for multiplication, what about the exponentiation we needed in FERMAT-TEST? It turns out that we can pull the same trick again. In the same was as we wrote multiplication as repeated addition, we can write exponentiation as repeated multiplication:

$$x^y = \underbrace{x \cdot x \cdot \cdots \cdot x \cdot x}_{y \text{ copies}}.$$

If we again select $y = 6$ then we obviously have

$$x^6 = x \cdot x \cdot x \cdot x \cdot x \cdot x.$$

The thing to notice is that there is a duality here: where there was an addition in our description of multiplication, that has become a multiplication in our description of exponentiation; where there was a multiplication, this has become an exponentiation. As such, we can adapt the MULTIPLY-REPEAT algorithm as follows:

```
1 algorithm EXPONENTIATE-REPEAT(x, y) begin
2 |   t ← 1
3 |   for i from 1 upto y do
4 |   |   t ← MULTIPLY-HORNER(t, x)
5 |   end
6 |   return t
7 end
```

One purpose of this is to highlight a subtle but fairly obvious fact: we are allowed to invoke one algorithm from within another one. In this case, we needed to multiply t by x in line #4; MULTIPLY-REPEAT and MULTIPLY-HORNER both compute the same result, so we could have used either of them here to do what we wanted. Either way, the idea is that half way through following the steps within

the EXPONENTIATE-REPEAT algorithm, we stop for a while and follow steps from MULTIPLY-HORNER instead so as to compute the value we need. Again selecting $x = 3$ and $y = 6$, the steps we perform would be something like the following:

Step #1 Assign $t \leftarrow 1$.

Step #2 Invoke MULTIPLY-HORNER(t, x), i.e., invoke MULTIPLY − HORNER$(1, 3)$,

Step #2.1 Assign $t \leftarrow 0$.

Step #2.2 Assign $t \leftarrow 2 \cdot t$, i.e., $t \leftarrow 2 \cdot 0 = 0$.

Step #2.3 Since $y_1 = 1$, assign $t \leftarrow t + x$, i.e., $t \leftarrow 0 + 1 = 1$.

Step #2.4 Assign $t \leftarrow 2 \cdot t$, i.e., $t \leftarrow 2 \cdot 1 = 2$.

Step #2.5 Since $y_0 = 1$, assign $t \leftarrow t + x$, i.e., $t \leftarrow 2 + 1 = 3$.

Step #2.6 Return $t = 3$.

then assign $t \leftarrow 3$.

Step #3 Invoke MULTIPLY-HORNER(t, x), i.e., invoke MULTIPLY − HORNER$(3, 3)$,

Step #3.1 Assign $t \leftarrow 0$.

Step #3.2 Assign $t \leftarrow 2 \cdot t$, i.e., $t \leftarrow 2 \cdot 0 = 0$.

Step #3.3 Since $y_1 = 1$, assign $t \leftarrow t + x$, i.e., $t \leftarrow 0 + 3 = 3$.

Step #3.4 Assign $t \leftarrow 2 \cdot t$, i.e., $t \leftarrow 2 \cdot 1 = 6$.

Step #3.5 Since $y_0 = 1$, assign $t \leftarrow t + x$, i.e., $t \leftarrow 6 + 3 = 9$.

Step #3.6 Return $t = 9$.

then assign $t \leftarrow 9$.

...

Step #8 Return $t = 729$.

Nothing has changed: we are still just following directions. We need to keep track of which algorithm we are following and ensure the names we give to variables do not get mixed up, but other than that things are not fundamentally more complicated.

Of course, we can also use Horner's rule by replacing all the additions with multiplications, and all multiplications with exponentiations. Using the same example as previously, we would end up with

$$\begin{aligned}
x^y &= x^{y_0} \cdot \left(x^{y_1} \cdot \left(x^{y_2} \cdot (1)^2 \right)^2 \right)^2 \\
&= x^0 \cdot \left(x^1 \cdot \left(x^1 \cdot (1)^2 \right)^2 \right)^2 \\
&= x^0 \cdot \left(x^1 \cdot \left(x^1 \cdot 1 \right)^2 \right)^2 \\
&= x^0 \cdot \left(x^1 \cdot \left(x^1 \right)^2 \right)^2 \\
&= x^0 \cdot \left(x^1 \cdot x^2 \right)^2 \\
&= x^0 \cdot \left(x^3 \right)^2 \\
&= x^0 \cdot x^6 \\
&= x^6
\end{aligned}$$

Unsurprisingly, our method for multiplying one number by another has a dual which is able to exponentiate one number by another:

```
1  algorithm EXPONENTIATE-HORNER(x, y) begin
2  |  t ← 1
3  |  for i from |y| − 1 downto 0 do
4  |  |  t ← MULTIPLY-HORNER(t, t)
5  |  |  if y_i = 1 then
6  |  |  |  t ← MULTIPLY-HORNER(t, x)
7  |  |  end
8  |  end
9  |  return t
10 end
```

You should compare EXPONENTIATE-HORNER as given above, line-by-line, with MULTIPLY-HORNER given earlier. Notice that they use the same idea; their structure is the same, we simply changed all the additions to multiplications. EXPONENTIATE-HORNER is often called the **square-and-multiply** algorithm [6] since it performs of a sequence of squaring (line #4) and multiplication (line #6): one squaring is performed for every bit of y whether it is equal to zero or one, and one multiplication for those bits of y which are equal to one.

The EXPONENTIATE-HORNER algorithm (so MULTIPLY-HORNER as well, since they are similar) is sometimes described as processing y in a left-to-right order: if you write y in binary, it starts at the left-hand end and finishes at the right-hand end. For instance, we had

$$y = 6_{(10)} = 110_{(2)}$$

and processed $y_2 = 1$ first, then $y_1 = 1$ then finally $y_0 = 0$. There is an alternative version of the algorithm that processes y the other way around, i.e., right-to-left. Do some research into this alternative: write down the algorithm, and convince yourself it will computes the same result using some examples. Can you think why the left-to-right version might be preferred to the right-to-left alternative, or vice versa?

2.4 Computational Complexity

Imagine we have two algorithms that solve the *same* problem, but do so in *different* ways. This should not be hard, because we already have some suitable candidates: given an x and y, MULTIPLY-REPEAT and MULTIPLY-HORNER compute the same result $x \cdot y$ differently. So armed with these, or another example of your choice, here

is a question: which one of the algorithms is the "best"? Actually, maybe we should go back a step: what does "best" even *mean*? Fastest? Shortest? Most attractive? All are reasonable measures, but imagine we select the first one and focus on selecting the algorithm which gives us a result in the least time.

Suppose we implemented the algorithms we would like to compare. This would give one way of comparing one against the other, we could simply time how long the corresponding programs take to execute on a computer. There are, however, many factors which might influence how long the execution of each program takes. For example:

- the skill of the programmer who implements the algorithm,
- the programming language used,
- the speed at which the computer can execute programs,
- the input, and
- the algorithm implemented by the program.

We know little or nothing about the first three factors, so have to focus on the latter two. Our goal is to introduce the subject of **computational complexity** [4]. This might sound scary, but is essentially about selectively ignoring detail: via a series of sane simplifications, each focusing on the most important, big picture issues, we can determine the quality of one algorithm compared to another.

2.4.1 Step Counting and Dominant Steps

As a guess at how long an algorithm would take to give a result if it *were* implemented and executed, we could count how many steps it takes. If you think about it, this makes perfect sense: the more steps the algorithm takes, the longer it will take to give result. But this would be quite a boring task if the algorithm had many steps. So the first simplification we make is to focus just on a small set of dominant steps, i.e., those steps we think are the most important. For algorithms that sort things, maybe the number of comparisons is the most important thing to count; for algorithms that process sequences of things, maybe the number of accesses to the sequence is the most important thing to count.

Since we are comparing algorithms that perform multiplication, it makes sense that we are interested mainly in arithmetic operations: the most important thing to count is the number of additions each algorithm uses to compute a result. Table 2.2 shows how many additions each algorithm performs for a range of inputs (ignore the columns marked $f(n)$ and $g(n)$ for now). It only includes a limited sample of inputs, but already we can identify two problems: first, the number of addition depends on y, and, second, it is not clear cut which algorithm uses the least number of additions. That is, sometimes MULTIPLY-REPEAT uses less, sometimes MULTIPLY-HORNER uses less. So in terms of answering our question, we are not really much better off than when we started.

Table 2.1 A table showing values of y and the number of bits in their binary representation

y	$\log_2(y+1)$	$\lceil \log_2(y+1) \rceil$
$1_{(10)} = 1_{(2)}$	1.000	1
$2_{(10)} = 10_{(2)}$	1.584	2
$3_{(10)} = 11_{(2)}$	2.000	2
$4_{(10)} = 100_{(2)}$	2.322	3
$5_{(10)} = 101_{(2)}$	2.585	3
$6_{(10)} = 110_{(2)}$	2.807	3
$7_{(10)} = 111_{(2)}$	3.000	3
$8_{(10)} = 1000_{(2)}$	3.170	4
⋮	⋮	⋮

2.4.2 Problem Size and Step Counting Functions

Usually we would like to make a general judgement about an algorithm which is independent of the input. To do this, we need to make another simplification: instead of thinking about the number of steps for a particular input y, we will shift things to consider a **problem size** n. The basic idea is that we try to write down a function for the number of steps an algorithm takes in terms of n, the problem size; this is a **step counting** function. Once we have such functions, we can forget about the algorithms themselves and simply compare the associated step counting functions with each other.

There is not really a general definition of what the problem size should mean; basically it is just a measure of how hard the problem is. For example, if we have an algorithm that sorts a sequence of numbers, the length of the sequence makes a good problem size: if n is larger, the problem is harder in the sense that the algorithm has to do more work. In our case, suppose we are looking at n-bit numbers: each input x and y has n binary digits. If n is larger, the problem is harder in the sense that the algorithm has to do more work: multiplying together large numbers is harder than multiplying small numbers together. We can describe n in terms of y as

$$n = \lceil \log_2(y+1) \rceil.$$

That is, we add one to y then it takes the logarithm to base two, and then round the result up to the nearest integer (this is called the **ceiling** [8] function); Table 2.1 details some results.

Consider MULTIPLY-REPEAT to start with: if we have n-bit inputs, how many additions will the algorithm perform? Looking at the algorithm again, we can see that the best case is when the inputs are really small: that way, we are clearly going to perform the least additions. The smallest number we can write in n bits is always going to be 0; this means we perform 0 additions. What about the worst case? This would be represented by the largest input we can write using n bits; this is $2^n - 1$. If

Table 2.2 A table showing the number of additions performed by MULTIPLY-REPEAT and MULTIPLY-HORNER, and the values of the associated step counting functions

y	$\lceil \log_2(y+1) \rceil$	MULTIPLY-REPEAT	$f(n) = 2^n - 1$	MULTIPLY-HORNER	$g(n) = 2 \cdot n$
1	1	1	1	2	2
2	2	2	3	3	4
3	2	3	3	4	4
4	3	4	7	4	6
5	3	5	7	5	6
6	3	6	7	5	6
7	3	7	7	6	6
8	4	8	15	5	8
⋮	⋮	⋮	⋮	⋮	⋮

we set $n = 2$ for example, the largest number we can write is three: the next largest number, i.e., four, needs $n = 3$. In the worst case then, we can write the step counting function

$$f(n) = 2^n - 1$$

to describe how many additions we would do. If $n = 2$, we are talking about 2-bit numbers and $f(n) = 3$ indicates that we would do at most three additions.

Now consider MULTIPLY-HORNER. If we have n-bit inputs, the loop in the algorithm iterates n times: we perform one iteration for each bit of the input y. This means we do n additions as a result of line #4. In the best case, all the bits of the input are zero (i.e., each $y_i = 0$) so we *never* process line #6 and do no more additions; in the worst case all the bits are one (i.e., each $y_i = 1$) and we *always* process line #6 and do n more additions. So in the worst case our step counting function will be

$$g(n) = 2 \cdot n.$$

Choosing $n = 2$ again, $g(n) = 4$ indicates that we would do four additions.

Table 2.2 shows the results of the step counting functions along side the actual number of steps performed by the algorithms. Clearly the two are not the same; the key thing to realise is that the actual number of steps is always less than or equal to the step counting function for the corresponding algorithm. This is because we developed each step counting function in relation to the worst case behaviour for a given input size, not the actual behaviour for that specific input.

2.4.3 Ignoring Small Problems and Minor Terms

The next simplification we make is to assume n is always large. In a way, this is obviously sensible: if n were small then nobody cares what algorithm we choose,

they will all be fast enough. The main result of making such an assumption is that we can make our step counting functions simpler. For example, we currently have the function

$$f(n) = 2^n - 1.$$

If n is going to be large, who cares about the 1 term? This will be incidental in comparison to the 2^n term. Imagine we select $n = 8$ for example: $2^8 = 256$ and $2^8 - 1 = 255$ are close enough that we may as well just treat them as the same and rewrite the function as

$$f(n) = 2^n.$$

What about the other step counting function? Consider the following *sequence* of step counting functions:

$$\langle 1 \cdot n, 2 \cdot n, 3 \cdot n, 4 \cdot n, 5 \cdot n, 6 \cdot n, \ldots \rangle.$$

Assuming n is positive, if we read the sequence from left-to-right each function is greater than the last one. For example $3 \cdot n$ will *always* be greater than $2 \cdot n$, no matter what n is. This means the number of steps would grow as we read from left-to-right: the associated algorithms would be slower and slower.

The natural end to this sequence is the point where one of the functions is $n \cdot n = n^2$. This is basically saying that n^2 is greater than *all* the functions in our sequence: if we select *any* constant $c < n$, then $c \cdot n$ is going to be less than n^2. As a result, another simplification we can make is to treat any step counting function that looks like $c \cdot n$ as n instead. This means we can rewrite

$$g(n) = 2 \cdot n$$

as

$$g(n) = n.$$

On one hand, this seems mad: an algorithm whose step counting function is $200 \cdot n$ will take 100 times more steps than one whose step counting function is $2 \cdot n$. So how can we rationally treat them as the same? Well, consider running an algorithm on two different computers: one is about ten years old, and one is very modern. The same algorithm will probably run 100 times slower on the old computer than it does on the new one: a ten year old computer is likely to be 100 times slower than a new one. This comparison is nothing to do with the algorithm in the sense that the computers are what makes the difference. So whether we take the step counting function as being $2 \cdot n$ or $100 \cdot n$ does not matter, the constant will eventually be one if we wait for a computer which is fast enough. So in comparing algorithms we always ignore constants terms, thus treat both $2 \cdot n$ and $100 \cdot n$ as the function n.

Actually writing

$$100 \cdot n = n$$

looks a bit odd, so instead we use the **big-O notation** [2] and we write the above non-equation as the equation

$$100 \cdot n = O(n)$$

which basically says that if n is big enough, then the function on the left behaves *at worst* like the function inside the big-O (having ignored any constants). The phrase "at worst" is crucial; we can write $n = O(n^2)$ since n is at worst n^2, but this is slightly lazy since we also know that $n = O(n)$.

2.4.4 Studying the Growth of Functions

Have a look at Fig. 2.2. It plots the growth of a few simple step counting functions as the problem size n grows; actually, it only uses very small values of n because some of the functions fly off the top quite quickly. What we are trying to do is visualise how the functions behave, relating the shape of the plot for a given step counting function to the number of steps taken by the associated algorithm.

1. The plot for $h(n) = 1$ is flat: it does not matter what the problem size is, there is no growth in the function. This is great! No matter how large the problem is, the step counting function says the algorithm will take a constant number of steps.
2. The next plot is for $h(n) = \log_2 n$. Unlike $h(n) = 1$, this function grows as n grows; the larger we make the problem size, the more steps the algorithm takes. On the other hand, $h(n) = \log_2 n$ grows quite slowly. We can see, from the trajectory that the plot is taking, that even for large values of n, $h(n) = \log_2 n$ will not be too large (in comparison to some of the other functions).
3. The plot for $h(n) = n$ is not so great. This time, the growth is constant: if the problems size is n, the algorithm takes n steps. If we compare the trajectory of the function $h(n) = n$ with $h(n) = \log_2 n$, $h(n) = n$ is clearly going to be much larger than $h(n) = \log_2 n$ in the long run.
4. Then things take a turn for the worse. The functions $h(n) = n^2$ and $h(n) = 2^n$ grow *really* quickly compared to the others. The function $h(n) = 2^n$ shows what we call exponential growth: looking at the almost vertical trajectory of $h(n) = 2^n$, it will be *enormous* even for fairly small values of n. So for large values of n, the algorithm will take so many steps to produce a result that it is probably not worth even invoking it!

Just so this hits home, notice that the function $h(n) = n$ is equal to 100 if $n = 100$: if the problem size is 100, the algorithm takes 100 steps. On the other hand, the function $h(n) = 2^n$ is equal to

$$1267650600228229401496703205376$$

if $n = 100$. That is a *lot* of steps!

Fig. 2.2 A graph illustrating the growth of several functions as the problem size is increased

Now, if we put all this evidence together we can try to answer our original question: given a problem size n, the number of steps taken by MULTIPLY-REPEAT can approximated by the function $f(n) = 2^n$; for MULTIPLY-HORNER the number of steps is approximated by the function $g(n) = n$. We have seen that f grows so quickly that even with quite small n, it is fair to say it will *always* be larger than g. This implies MULTIPLY-HORNER gives us a result in the least time, and is therefore the "best" algorithm. Putting this into context helps show the value of our analysis: in real computers, 32-bit numbers are common. If we wanted to multiply such numbers together MULTIPLY-REPEAT would take approximately 4294967296 steps in the worst case; if we selected MULTIPLY-HORNER instead, we only need approximately 32 steps. Hopefully it is clear that even though we have made simplifications along the way, they would have to be wrong on a monumental scale to make our

selection the incorrect one! So to cut a long story short, computational complexity has allowed us to reason about and compare algorithms in a meaningful, theoretical way: this is a *very* powerful tool *if* used for the right job.

The big-O notation is one of the most important concepts, and also items of Mathematical notation, in Computer Science. It is worth getting used to it now, so consider the following nine examples:

$$\begin{aligned}
2n+1 &= O(n)\\
2^n+3n &= O\left(2^n\right)\\
6n^2+3n+1 &= O\left(6n^2\right)\\
\frac{n}{2}+\log n &= O(n)\\
2^{n\log n} &= O\left(2^n\right)\\
6n^2+3n+1 &= O\left(n^2+n\right)\\
(\log n)^3 &= O(n)\\
(\log n)^3 &= O\left((\log n)^4\right)\\
(\log n)^3 &= O\left((\log n)^3\right)
\end{aligned}$$

One of them is wrong, can you work out which one (and the reason why)?

In turns out big-O is a short-hand for big-Omicron, if you want to be exact about the Greek symbol used. Within this context, using other Greek symbols allows us make other statements about a function: there is a whole family, including o (or little-O), ω (or little-Omega), Ω (or big-Omega), and Θ (or big-Theta) for instance. Do some research into what these mean, and why we might use them *instead* of big-O in specific situations.

References

1. Wikipedia: Algorithm. http://en.wikipedia.org/wiki/Algorithm
2. Wikipedia: Big-O notation. http://en.wikipedia.org/wiki/Big_O_notation
3. Wikipedia: Carmichael number. http://en.wikipedia.org/wiki/Carmichael_number
4. Wikipedia: Computational complexity theory. http://en.wikipedia.org/wiki/Computational_complexity_theory
5. Wikipedia: Coprime. http://en.wikipedia.org/wiki/Coprime
6. Wikipedia: Exponentiation by squaring. http://en.wikipedia.org/wiki/Exponentiation_by_squaring
7. Wikipedia: Fermat primality test. http://en.wikipedia.org/wiki/Fermat_primality_test

8. Wikipedia: Floor and ceiling functions. http://en.wikipedia.org/wiki/Floor_and_ceiling_functions
9. Wikipedia: Flow chart. http://en.wikipedia.org/wiki/Flowchart
10. Wikipedia: Horner's scheme. http://en.wikipedia.org/wiki/Horner_scheme
11. Wikipedia: Pesudo-code. http://en.wikipedia.org/wiki/Pseudo_code
12. Wikipedia: Prime number. http://en.wikipedia.org/wiki/Prime_number

http://www.springer.com/978-3-319-04041-7

What Is Computer Science?
An Information Security Perspective
Page, D.; Smart, N.
2014, XVIII, 232 p. 84 illus., Softcover
ISBN: 978-3-319-04041-7