Chapter 1

Preliminaries

> It is commonly believed that more than 70% (!) of the effort and cost of developing a complex software system is devoted, in one way or another, to error correcting.
>
> Algorith., pg. 107 [Harel (1987)]

1.1 What is correctness?

To show that an algorithm is correct, we must show somehow that it does what it is supposed to do. The difficulty is that the algorithm unfolds in time, and it is tricky to work with a variable number of steps, i.e., while-loops. We are going to introduce a framework for proving algorithm (and program) correctness that is called *Hoare's logic*. This framework uses induction and invariance (see Section 9.1), and logic (see Section 9.4) but we are going to use it informally. For a formal example see Section 9.4.4.

We make two assertions, called the *pre-condition* and the *post-condition*; by correctness we mean that whenever the pre-condition holds *before* the algorithm executes, the post-condition will hold *after* it executes. By *termination* we mean that whenever the pre-condition holds, the algorithm will stop running after finitely many steps. Correctness without termination is called *partial correctness*, and *correctness* per se is partial correctness *with* termination. All this terminology is there to connect a given problem with some algorithm that purports to solve it. Hence we pick the pre and post condition in a way that reflects this relationship and proves it true.

These concepts can be made more precise, but we need to introduce some standard notation: Boolean connectives: $\wedge$ is "and," $\vee$ is "or" and $\neg$ is "not." We also use $\rightarrow$ as Boolean implication, i.e., $x \rightarrow y$ is logically equivalent to $\neg x \vee y$, and $\leftrightarrow$ is Boolean equivalence, and $\alpha \leftrightarrow \beta$ expresses $((\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha))$. $\forall$ is the "for-all" universal quantifier, and $\exists$ is the "there exists" existential quantifier. We use "$\Rightarrow$" to abbreviate the word "implies," i.e., $2|x \Rightarrow x$ is even, while "$\not\Rightarrow$" abbreviates "does not imply."

Let A be an algorithm, and let $\mathcal{I}_A$ be the set of all *well-formed* inputs for A; the idea is that if $I \in \mathcal{I}_A$ then it "makes sense" to give I as an input to A. The concept of a "well-formed" input can also be made precise, but it is enough to rely on our intuitive understanding—for example, an algorithm that takes a pair of integers as input will not be "fed" a matrix. Let $O = A(I)$ be the output of A on I, if it exists. Let α_A be a pre-condition and β_A a post-condition of A; if I satisfies the pre-condition we write $\alpha_A(I)$ and if O satisfies the post-condition we write $\beta_A(O)$. Then, partial correctness of A with respect to pre-condition α_A and post-condition β_A can be stated as:

$$(\forall I \in \mathcal{I}_A)[(\alpha_A(I) \wedge \exists O(O = A(I))) \rightarrow \beta_A(A(I))]. \tag{1.1}$$

In words: for any well formed input I, if I satisfies the pre-condition and $A(I)$ produces an output (i.e., terminates), then this output satisfies the post-condition.

Full correctness is (1.1) together with the assertion that for all $I \in \mathcal{I}_A$, $A(I)$ terminates (and hence there exists an O such that $O = A(I)$).

Problem 1.1. *Modify (1.1) to express full correctness.*

A fundamental notion in the analysis of algorithms is that of a *loop invariant*; it is an assertion that stays true after each execution of a "while" (or "for") loop. Coming up with the right assertion, and proving it, is a creative endeavor. If the algorithm terminates, the loop invariant is an assertion that helps to prove the implication $\alpha_A(I) \rightarrow \beta_A(A(I))$.

Once the loop invariant has been shown to hold, it is used for proving partial correctness of the algorithm. So the criterion for selecting a loop invariant is that it helps in proving the post-condition. In general many different loop invariants (and for that matter pre and post-conditions) may yield a desirable proof of correctness; the art of the analysis of algorithms consists in selecting them judiciously. We usually need induction to prove that a chosen loop invariant holds after each iteration of a loop, and usually we also need the pre-condition as an assumption in this proof.

1.1.1 *Complexity*

Given an algorithm $\mathcal{A}$, and an input x, the running time of $\mathcal{A}$ on x is the number of steps it takes $\mathcal{A}$ to terminate on input x. The delicate issue here is to define a "step," but we are going to be informal about it: we assume a Random Access Machine (a machine that can access memory cells in a single step), and we assume that an assignment of the type $x \leftarrow y$ is a single step, and so are arithmetical operations, and the testing of Boolean expressions (such as $x \geq y \wedge y \geq 0$). Of course this simplification does not reflect the true state of affairs if for example we manipulate numbers of 4,000 bits (as in the case of cryptographic algorithms). But then we redefine steps appropriately to the context.

We are interested in *worst-case complexity*. That is, given an algorithm $\mathcal{A}$, we let $T^{\mathcal{A}}(n)$ to be the maximal running time of $\mathcal{A}$ on any input x of size n. Here "size" means the number of bits in a reasonable fixed encoding of x. We tend to write $T(n)$ instead of $T^{\mathcal{A}}(n)$ as the algorithm under discussion is given by the context. It turns out that even for simple algorithms $T(n)$ maybe very complicated, and so we settle for asymptotic bounds on $T(n)$.

In order to provide asymptotic approximations to $T(n)$ we introduce the *Big O* notation, pronounced as "big-oh." Consider functions f and g from $\mathbb{N}$ to $\mathbb{R}$, that is, functions whose domain is the natural numbers but can range over the reals. We say that $g(n) \in O(f(n))$ if there exist constants $c, n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $g(n) \leq cf(n)$, and the *little o* notation, $g(n) \in o(f(n))$, which denotes that $\lim_{n\to\infty} g(n)/f(n) = 0$. We also say that $g(n) \in \Omega(f(n))$ if there exist constants c, n_0 such that for all $n \geq n_0$, $g(n) \geq cf(n)$. Finally, we say that $g(n) \in \Theta(f(n))$ if it is the case that $g(n) \in O(f(n)) \cap \Omega(f(n))$. If $g(n) \in \Theta(f(n))$, then $f(n)$ is called an *asymptotically tight bound* for $g(n)$, and it means that $f(n)$ is a very good approximation to $g(n)$. Note that in practice we will often write $g(n) = O(f(n))$ instead of the formal $g(n) \in O(f(n))$; a slight but convenient abuse of notation.

For example, $an^2 + bn + c = \Theta(n^2)$, where $a > 0$. To see this, note that $an^2 + bn + c \leq (a + |b| + |c|)n^2$, for all $n \in \mathbb{N}$, and so $an^2 + bn + c = O(n^2)$, where we took the absolute value of b, c because they may be negative. On the other hand, $an^2 + bn + c = a((n + c_1)^2 - c_2)$ where $c_1 = b/2a$ and $c_2 = (b^2 - 4ac)/4a^2$, so we can find a c_3 and an n_0 so that for all $n \geq n_0$, $c_3 n^2 \leq a((n + c_1)^2 - c_2)$, and so $an^2 + bn + c = \Omega(n^2)$.

Problem 1.2. *Find c_3 and n_0 in terms of a, b, c. Then prove that for $k \geq 0$, $\sum_{i=0}^{k} a_i n^i = \Theta(n^k)$; this shows the simplifying advantage of the Big O.*

1.1.2 *Division*

What could be simpler than integer division? We are given two integers, x, y, and we want to find the quotient and remainder of dividing x by y. For example, if $x = 25$ and $y = 3$, then $q = 3$ and $r = 1$. Note that the q and r returned by the division algorithm are usually denoted as $\text{div}(x, y)$ (the *quotient*) and $\text{rem}(x, y)$ (the *remainder*), respectively.

Algorithm 1.1 Division

Pre-condition: $x \geq 0 \wedge y > 0 \wedge x, y \in \mathbb{N}$

1: $q \leftarrow 0$
2: $r \leftarrow x$
3: **while** $y \leq r$ **do**
4: $\quad r \leftarrow r - y$
5: $\quad q \leftarrow q + 1$
6: **end while**
7: **return** q, r

Post-condition: $x = (q \cdot y) + r \wedge 0 \leq r < y$

We propose the following assertion as the loop invariant:

$$x = (q \cdot y) + r \wedge r \geq 0, \tag{1.2}$$

and we show that (1.2) holds after each iteration of the loop. Basis case (i.e., zero iterations of the loop—we are just before line 3 of the algorithm): $q = 0, r = x$, so $x = (q \cdot y) + r$ and since $x \geq 0$ and $r = x$, $r \geq 0$.

Induction step: suppose $x = (q \cdot y) + r \wedge r \geq 0$ and we go once more through the loop, and let q', r' be the new values of q, r, respectively (computed in lines 4 and 5 of the algorithm). Since we executed the loop one more time it follows that $y \leq r$ (this is the condition checked for in line 3 of the algorithm), and since $r' = r - y$, we have that $r' \geq 0$. Thus,

$$x = (q \cdot y) + r = ((q + 1) \cdot y) + (r - y) = (q' \cdot y) + r',$$

and so q', r' still satisfy the loop invariant (1.2).

Now we use the loop invariant to show that (if the algorithm terminates) the post-condition of the division algorithm holds, *if* the pre-condition holds. This is very easy in this case since the loop ends when it is no longer true that $y \leq r$, i.e., when it is true that $r < y$. On the other hand, (1.2) holds after each iteration, and in particular the last iteration. Putting together (1.2) and $r < y$ we get our post-condition, and hence partial correctness.

To show termination we use the least number principle (LNP). We need to relate some non-negative monotone decreasing sequence to the algorithm; just consider $r_0, r_1, r_2, \ldots$, where $r_0 = x$, and r_i is the value of r after the i-th iteration. Note that $r_{i+1} = r_i - y$. First, $r_i \geq 0$, because the algorithm enters the while loop only if $y \leq r$, and second, $r_{i+1} < r_i$, since $y > 0$. By LNP such a sequence "cannot go on for ever," (in the sense that the set $\{r_i | i = 0, 1, 2, \ldots\}$ is a subset of the natural numbers, and so it has a least element), and so the algorithm must terminate.

Thus we have shown full correctness of the division algorithm.

Problem 1.3. *What is the running time of algorithm 1.1? That is, how many steps does it take to terminate? Assume that assignments (lines 1 and 2), and arithmetical operations (lines 4 and 5) as well as testing "$\leq$" (line 3) all take one step.*

Problem 1.4. *Suppose that the precondition in algorithm 1.1 is changed to say: "$x \geq 0 \wedge y > 0 \wedge x, y \in \mathbb{Z}$," where $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. Is the algorithm still correct in this case? What if it is changed to to the following: "$y > 0 \wedge x, y \in \mathbb{Z}$"? How would you modify the algorithm to work with negative values?*

Problem 1.5. *Write a program that takes as input x and y, and outputs the intermediate values of q and r, and finally the quotient and remainder of the division of x by y.*

1.1.3 *Euclid*

Given two positive integers a, b, their *greatest common divisor*, denoted as $\gcd(a, b)$, is the greatest integer that divides both. Euclid's algorithm, presented as algorithm 1.2, is a procedure for finding the greatest common divisor of two numbers. It is one of the oldest know algorithms; it appeared in Euclid's *Elements* (Book 7, Propositions 1 and 2) around 300 BC.

Note that to compute $\text{rem}(n, m)$ in lines 1 and 3 of Euclid's algorithm we need to use algorithm 1.1 (the division algorithm) as a subroutine; this is a typical "composition" of algorithms. Also note that lines 1 and 3 are executed from left to right, so in particular in line 3 we first do $m \leftarrow n$, then $n \leftarrow r$, and finally $r \leftarrow \text{rem}(m, n)$. This is important for the algorithm to work correctly, because when we are executing $r \leftarrow \text{rem}(m, n)$, we are using the newly updated values of m, n.

Algorithm 1.2 Euclid

Pre-condition: $a > 0 \wedge b > 0 \wedge a, b \in \mathbb{Z}$

1: $m \leftarrow a$; $n \leftarrow b$; $r \leftarrow \text{rem}(m, n)$
2: **while** $(r > 0)$ **do**
3: $\quad m \leftarrow n$; $n \leftarrow r$; $r \leftarrow \text{rem}(m, n)$
4: **end while**
5: **return** n

Post-condition: $n = \gcd(a, b)$

To prove the correctness of Euclid's algorithm we are going to show that after each iteration of the while loop the following assertion holds:

$$m > 0, n > 0 \text{ and } \gcd(m, n) = \gcd(a, b), \tag{1.3}$$

that is, (1.3) is our loop invariant. We prove this by induction on the number of iterations. Basis case: after zero iterations (i.e., just before the while loop starts—so after executing line 1 and before executing line 2) we have that $m = a > 0$ and $n = b > 0$, so (1.3) holds trivially. Note that $a > 0$ and $b > 0$ by the pre-condition.

For the induction step, suppose $m, n > 0$ and $\gcd(a, b) = \gcd(m, n)$, and we go through the loop one more time, yielding m', n'. We want to show that $\gcd(m, n) = \gcd(m', n')$. Note that from line 3 of the algorithm we see that $m' = n, n' = r = \text{rem}(m, n)$, so in particular $m' = n > 0$ and $n' = r = \text{rem}(m, n) > 0$ since if $r = \text{rem}(m, n)$ were zero, the loop would have terminated (and we are assuming that we are going through the loop one more time). So it is enough to prove the assertion in Problem 1.6.

Problem 1.6. *Show that for all* $m, n > 0$, $\gcd(m, n) = \gcd(n, \text{rem}(m, n))$.

Now the correctness of Euclid's algorithm follows from (1.3), since the algorithm stops when $r = \text{rem}(m, n) = 0$, so $m = q{\cdot}n$, and so $\gcd(m, n) = n$.

Problem 1.7. *Show that Euclid's algorithm terminates, and establish its Big O complexity.*

Problem 1.8. *How would you make the algorithm more efficient? This question is asking for simple improvements that lower the running time by a constant factor.*

Problem 1.9. *Modify Euclid's algorithm so that given integers* m, n *as input, it outputs integers* a, b *such that* $am + bn = g = \gcd(m, n)$. *This is called the* extended Euclid's algorithm. *Follow this outline:*

(a) Use the LNP to show that if $g = \gcd(m, n)$, then there exist a, b such that $am + bn = g$.
(b) Design Euclid's extended algorithm, and prove its correctness.
(c) The usual Euclid's extended algorithm has a running time polynomial in $\min\{m, n\}$; show that this is the running time of your algorithm, or modify your algorithm so that it runs in this time.

Problem 1.10. *Write a program that implements Euclid's extended algorithm. Then perform the following experiment: run it on a random selection of inputs of a given size, for sizes bounded by some parameter N; compute the average number of steps of the algorithm for each input size $n \leq N$, and use* `gnuplot`[1] *to plot the result. What does $f(n)$—which is the "average number of steps" of Euclid's extended algorithm on input size n—look like? Note that size is not the same as value; inputs of size n are inputs with a binary representation of n bits.*

1.1.4 *Palindromes*

Algorithm 1.3 tests if a string is a *palindrome*, which is a word that read the same backwards as forwards, e.g., `madamimadam` or `racecar`.

In order to present this algorithm we need to introduce a little bit of notation. The *floor* and *ceil* functions are defined, respectively, as follows: $\lfloor x \rfloor = \max\{n \in \mathbb{Z} | n \leq x\}$ and $\lceil x \rceil = \min\{n \in \mathbb{Z} | n \geq x\}$, and $\lfloor x \rceil$ refers to the "rounding" of x, and it is defined as $\lfloor x \rceil = \lfloor x + \frac{1}{2} \rfloor$.

Algorithm 1.3 Palindromes

Pre-condition: $n \geq 1 \wedge A[0 \ldots n-1]$ is a character array
1: $i \leftarrow 0$
2: **while** $(i < \lfloor \frac{n}{2} \rfloor)$ **do**
3: **if** $(A[i] \neq A[n-i-1])$ **then**
4: **return** F
5: **end if**
6: $i \leftarrow i + 1$
7: **end while**
8: **return** T
Post-condition: return T iff A is a palindrome

[1] Gnuplot is a command-line driven graphing utility (`http://www.gnuplot.info`). Also, Python has a plotting library `matplotlib` (`https://matplotlib.org`).

Let the loop invariant be: after the k-th iteration, $i = k + 1$ and for all j such that $1 \leq j \leq k$, $A[j] = A[n - j + 1]$. We prove that the loop invariant holds by induction on k. Basis case: before any iterations take place, i.e., after zero iterations, there are no j's such that $1 \leq j \leq 0$, so the second part of the loop invariant is (vacuously) true. The first part of the loop invariant holds since i is initially set to 1.

Induction step: we know that after k iterations, $A[j] = A[n-j+1]$ for all $1 \leq j \leq k$; after one more iteration we know that $A[k+1] = A[n-(k+1)+1]$, so the statement follows for all $1 \leq j \leq k+1$. This proves the loop invariant.

Problem 1.11. *Using the loop invariant argue the partial correctness of the palindromes algorithm. Show that the algorithm terminates.*

It is easy to manipulate strings in Python; a segment of a string is called a *slice*. Consider the word `palindrome`; if we set the variables `s` to this word,

```
s = 'palindrome'
```

then we can access different slices as follows:

```
print s[0:5]      palin
print s[5:10]     drome
print s[5:]       drome
print s[2:8:2]    lnr
```

where the notation `[i:j]` means the segment of the string starting from the i-th character (and we always start counting at zero!), to the j-th character, including the first but excluding the last. The notation `[i:]` means from the i-th character, all the way to the end, and `[i:j:k]` means starting from the i-th character to the j-th (again, not including the j-th itself), taking every k-th character.

One way to understand the string delimiters is to write the indices "in between" the numbers, as well as at the beginning and at the end. For example

$$_0\mathtt{p}_1\mathtt{a}_2\mathtt{l}_3\mathtt{i}_4\mathtt{n}_5\mathtt{d}_6\mathtt{r}_7\mathtt{o}_8\mathtt{m}_9\mathtt{e}_{10}$$

and to notice that a slice `[i:j]` contains all the symbols between index i and index j.

Problem 1.12. *Using Python's inbuilt facilities for manipulating slices of strings, write a succinct program that checks whether a given string is a palindrome.*

1.1.5 *Further examples*

In this section we provide more examples of algorithms that take as integers as input, and manipulate them with a while-loop. We also present an example of an algorithm that is very easy to describe, but for which no proof of termination is known (algorithm 1.6). This supports further the notion that proofs of correctness are not just pedantic exercises in mathematical formalism but a real certificate of validity of a given algorithmic solution.

Problem 1.13. *Give an algorithm which takes as input a positive integer n, and outputs "yes" if $n = 2^k$ (i.e., n is a power of 2), and "no" otherwise. Prove that your algorithm is correct.*

Problem 1.14. *What does algorithm 1.4 compute? Prove your claim.*

Algorithm 1.4 See Problem 1.14

```
1: x ← m ; y ← n ; z ← 0
2: while (x ≠ 0) do
3:         if (rem(x, 2) = 1) then
4:                 z ← z + y
5:         end if
6:         x ← div(x, 2)
7:         y ← y · 2
8: end while
9: return z
```

Problem 1.15. *What does algorithm 1.5 compute? Assume that a, b are positive integers (i.e., assume that the pre-condition is that $a, b > 0$). For which starting a, b does this algorithm terminate? In how many steps does it terminate, if it does terminate?*

Algorithm 1.5 See Problem 1.15

```
1: while (a > 0) do
2:         if (a < b) then
3:                 (a, b) ← (2a, b − a)
4:         else
5:                 (a, b) ← (a − b, 2b)
6:         end if
7: end while
```

Consider algorithm 1.6 given below.

Algorithm 1.6 Ulam's algorithm

Pre-condition: $a > 0$

```
x ⟵ a
while last three values of x not 4, 2, 1 do
        if x is even then
                x ⟵ x/2
        else
                x ⟵ 3x + 1
        end if
end while
```

This algorithm is different from all the algorithms that we have seen thus far in that there is no known proof of termination, and therefore no known proof of correctness. Observe how simple it is: for any positive integer a, set $x = a$, and repeat the following: if x is even, divide it by 2, and if it is odd, multiply it by 3 and add 1. Repeat this until the last three values obtained were $4, 2, 1$. For example, if $a = 22$, then one can check that x takes on the following values: $22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, \mathbf{4}, \mathbf{2}, \mathbf{1}$, and algorithm 1.6 terminates. It is conjectured that regardless of the initial value of a, as long as a is a positive integer, algorithm 1.6 terminates. This conjecture is known as "Ulam's problem,"[2] and despite decades of work no one has been able to solve this problem.

In fact, recent work shows that variants of Ulam's problem have been shown undecidable. We will look at undecidability in Chapter 9, but [Lehtonen (2008)] showed that for a very simple variant of the problem where we let x be $3x + t$ for x in a particular set A_t (for details see the paper), there simply is no algorithm whatsoever that will decide for which initial a's the new algorithm terminates and for which it does not.

Problem 1.16. *Write a program that takes a as input and displays all the values of Ulam's problem until it sees $4, 2, 1$ at which point it stops. You have just written an almost trivial program for which there is no proof of termination. Now do an experiment: compute how many steps it takes to reach $4, 2, 1$ for all $a < N$, for some N. Any conjectures?*

[2] It is also called "Collatz Conjecture," "Syracuse Problem," "Kakutani's Problem," or "Hasse's Algorithm." While it is true that a rose by any other name would smell just as sweet, the preponderance of names shows that the conjecture is a very alluring mathematical problem.

1.2 Ranking algorithms

The algorithms we have seen so far in the book are classical but to some extent they are "toy examples." In this section we want to demonstrate the power and usefulness of some very well known "grown up" algorithms. We will focus on three different ranking algorithms. Ranking items is a primordial human activity, and we will take a brief look at ranking procedures that range from the ancient, such as Ramon Llull's, a 13-th century mystic and philosopher, to old, such as Marquis de Condorcet's work discussed in Section 1.2.3, to the state of the art in Google's simple and elegant PageRank discussed in the next section.

1.2.1 *PageRank*

In 1945, Vannevar Bush wrote an article in the Atlantic Monthly entitled *As we may think* [Bush (1945)], where he demonstrated an eerie prescience of the ideas that became the World Wide Web. In that gem of an article Bush pointed out that information retrieval systems are organized in a linear fashion (whether books, databases, computer memory, etc.), but that human conscious experience exhibits what he called "an associative memory." That is, the human mind has a semantic network, where we think of one thing, and that reminds us of another, etc. Bush proposed a blueprint for a human-like machine, the "Memex," which had ur-web characteristics: digitized human knowledge interconnected by associative links.

When in the early 1990s Tim Berners-Lee finally implemented the ideas of Bush in the form of HTML, and ushered in the World Wide Web, the web pages were static and the links had a navigational function. Today links often trigger complex programs such as Perl, PHP, MySQL, and while some are still navigational, many are transactional, implementing actions such as "add to shopping cart," or "update my calendar."

As there are now billions of active web pages, how does one search them to find relevant high-quality information? We accomplish this by ranking those pages that meet the search criteria; pages of a good rank will appear at the top — this way the search results will make sense to a human reader who only has to scan the first few results to (hopefully) find what he wants. These top pages are called *authoritative* pages.

In order to rank authoritative pages at the top, we make use of the fact that the web consists not only of pages, but also of *hyperlinks* that connect these pages. This hyperlink structure (which can be naturally modeled by a

directed graph) contains a lot of latent human annotation that can be used to automatically infer authority. This is a profound observation: after all, items ranked highly by a user are ranked so in a subjective manner; exploiting the hyperlink structure allows us to connect the subjective experience of the users with the output of an algorithm!

More specifically, by creating a hyperlink, the author gives an implicit endorsement to a page. By mining the collective judgment expressed by these endorsements we get a picture of the quality (or subjective perception of the quality) of a given web page. This is very similar to our perception of the quality of scholarly citations, where an important publication is cited by other important publications. The question now is how do we convert these ideas into an algorithm. A seminal answer was given by the now famous PageRank algorithm, authored by S. Brin and L. Page, the founders of Google — see [Brin and Page (1998)]. PageRank mines the hyperlink structure of the web in order to infer the relative importance of the pages.

Consider Figure 1.1 which depicts a web page X, and all the pages $T_1, T_2, T_3, \ldots, T_n$ that point to it. Given a page X, let $\mathrm{C}(X)$ be the number of distinct link that leave X, i.e., these are links anchored in X that point to a page outside of X. Let $\mathrm{PR}(X)$ be the page rank of X. We also employ a parameter d, which we call the *damping factor*, and which we will explain later.

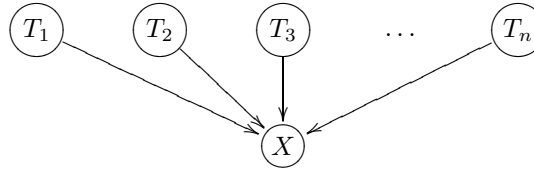


Fig. 1.1 Computing the rank of page A.

Then, the page rank of X can be computed as follows:

$$\mathrm{PR}(X) = (1-d) + d\left[\frac{\mathrm{PR}(T_1)}{\mathrm{C}(T_1)} + \frac{\mathrm{PR}(T_2)}{\mathrm{C}(T_2)} + \cdots + \frac{\mathrm{PR}(T_n)}{\mathrm{C}(T_n)}\right]. \qquad (1.4)$$

We now explain (1.4): the damping factor d is a constant $0 \leq d \leq 1$, and usually set to .85. The formula posits the behavior of a "random surfer" who starts clicking on links on a random page, following a link out of that page, and clicking on links (never hitting the "back button") until the random surfer gets bored, and starts the process from the beginning by going to a random page. Thus, in (1.4) the $(1-d)$ is the probability of

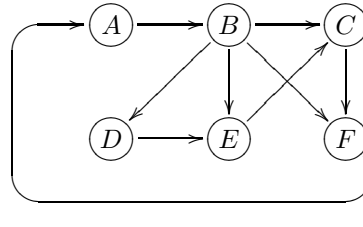
choosing X at random, while $\frac{\mathrm{PR}(T_i)}{\mathrm{C}(T_i)}$ is the probability of reaching X by coming from T_i, normalized by the number of outlinks from T_i. We make a slight adjustment to (1.4): we normalize it by the size of the web, N, that is, we divide $(1-d)$ by N. This way, the chance of stumbling on X is adjusted to the overall size of the web.

The problem with (1.4) is that it appears to be circular. How do we compute $\mathrm{PR}(T_i)$ in the first place? The algorithm works in stages, refining the page rank of each page at each stage. Initially, we take the egalitarian approach and assign each page a rank of $1/N$, where N is the total number of pages on the web. Then recompute all page ranks using (1.4) and the initial page ranks, and continue. After each stage $\mathrm{PR}(X)$ gets closer to the actual value, and in fact converges fairly quickly. There are many technical issues here, such as knowing when to stop, and handling a computation involving N which may be over a trillion, but this is the PageRank algorithm in a nut shell.

Of course the web is a vast collection of heterogeneous documents, and (1.4) is too simple a formula to capture everything, and so Google search is a lot more complicated. For example, not all outlinks are treated equally: a link in larger font, or emphasized with a "`<STRONG>`" tag, will have more weight. Documents differ internally in terms of language, format such as PDF, image, text, sound, video; and externally in terms of reputation of the source, update frequency, quality, popularity, and other variables that are now taken into account by a modern search engine. The reader is directed to [Franceschet (2011)] for more information about PageRank.

Furthermore, the presence of search engines also affects the web. As the search engines direct traffic, they themselves shape the ranking of the web. A similar effect in Physics is known as the *observer effect*, where instruments alter the state of what they observe. As a simple example, consider measuring the pressure in your tires: you have to let some air out, and therefore change the pressure slightly, in order to measure it. All these fascinating issues are the subject matter of Big Data Analytics.

Problem 1.17. *Consider the following small network:*



Compute the PageRank of the different pages in this network using (1.4) with damping factor $d = 1$, that is, assuming all navigation is done by following links (no random jumps to other pages).

Problem 1.18. *Write a program which computes the ranks of all the pages in a given network of size N. Let the network be given as a 0-1 matrix, where a 1 in position (i, j) means that there is a link from page i to page j. Otherwise, there is a 0 in that position. Use (1.4) to compute the page rank, starting with a value of $1/N$. You should stop when all values have converged — does this algorithm always terminate? Also, keep track of all the values as fractions a/b, where $\gcd(a, b) = 1$; Python has a convenient fractions library:* `import fractions`.

1.2.2 *A stable marriage*

Suppose that we want to match interns with hospitals, or students with colleges; both are instances of the *admission process problem*, and both have a solution that optimizes, to a certain degree, the overall satisfaction of all the parties concerned. The solution to this problem is an elegant algorithm to solve the so called "stable marriage problem," which has been used since the 1960s for the college admission process and for matching interns with hospitals.

An instance of the *stable marriage problem* of size n consists of two disjoint finite sets of equal size; a set of *boys* $B = \{b_1, b_2, \ldots, b_n\}$, and a set of *girls* $G = \{g_1, g_2, \ldots, g_n\}$. Let "$<_i$" denote the ranking of boy b_i; that is, $g <_i g'$ means that boy b_i prefers g over g'. Similarly, "$<^j$" denotes the ranking of girl g_j. Each boy b_i has such a ranking (linear ordering) $<_i$ of G which reflects his preference for the girls that he wants to marry. Similarly each girl g_j has a ranking (linear ordering) $<^j$ of B which reflects her preference for the boys she would like to marry.

A *matching* (or *marriage*) M is a 1-1 correspondence between B and G. We say that b and g are *partners* in M if they are matched in M and write $p_M(b) = g$ and also $p_M(g) = b$. A matching M is *unstable* if there is a pair (b, g) from $B \times G$ such that b and g are not partners in M but b prefers g to $p_M(b)$ and g prefers b to $p_M(g)$. Such a pair (b, g) is said to *block* the matching M and is called a *blocking pair* for M (see figure 1.2). A matching M is stable if it contains no blocking pairs.

It turns out that there always exists a stable marriage solution to the matching problem. This solution can be computed with the celebrated

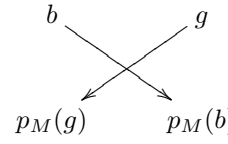


Fig. 1.2 A blocking pair: b and g prefer each other to their partners $p_M(b)$ and $p_M(g)$.

algorithm due to Gale and Shapley ([Gale and Shapley (1962)]) that outputs a stable marriage for any input B, G, regardless of the ranking[3].

The matching M is produced in stages M_s so that b_t always has a partner at the end of stage s, where $t \leq s$. However, the partners of b_t do not get better, i.e., $p_{M_t}(b_t) \leq_t p_{M_{t+1}}(b_t) \leq_t \cdots$. On the other hand, for each $g \in G$, if g has a partner at stage t, then g will have a partner at each stage $s \geq t$ and the partners do not get worse, i.e., $p_{M_t}(g) \geq^t p_{M_{t+1}}(g) \geq^t \ldots$. Thus, as s increases, the partners of b_t become less preferable and the partners of g become more preferable.

At the end of stage s, assume that we have produced a matching

$$M_s = \{(b_1, g_{1,s}), \ldots, (b_s, g_{s,s})\},$$

where the notation $g_{i,s}$ means that $g_{i,s}$ is the partner of boy b_i after the end of stage s.

We will say that partners in M_s are *engaged*. The idea is that at stage $s+1$, b_{s+1} will try to get a partner by *proposing* to the girls in G in his order of preference. When b_{s+1} proposes to a girl g_j, g_j accepts his proposal if either g_j is not currently engaged or is currently engaged to a less preferable boy b, i.e., $b_{s+1} <^j b$. In the case where g_j prefers b_{s+1} over her current partner b, then g_j breaks off the engagement with b and b then has to search for a new partner.

Problem 1.19. *Show that each b need propose at most once to each g.*

From problem 1.19 we see that we can make each boy keep a bookmark on his list of preference, and this bookmark is only moving forward. When a boy's turn to choose comes, he starts proposing from the point where his bookmark is, and by the time he is done, his bookmark moved only forward. Note that at stage $s+1$ each boy's bookmark cannot have moved beyond the girl number s on the list without choosing someone (after stage

[3]In 2012, the Nobel Prize in Economics was awarded to Lloyd S. Shapley and Alvin E. Roth "for the theory of stable allocations and the practice of market design," i.e., for the stable marriage algorithm.

Algorithm 1.7 Gale-Shapley

```
1: Stage 1: b_1 chooses his top g and M_1 ⟵ {(b_1, g)}
2: for s = 1, ..., s = |B| − 1, Stage s + 1: do
3:         M ⟵ M_s
4:         b* ⟵ b_{s+1}
5:         for b* proposes to all g's in order of preference: do
6:                 if g was not engaged: then
7:                         M_{s+1} ⟵ M ∪ {(b*, g)}
8:                         end current stage
9:                 else if g was engaged to b but g prefers b*: then
10:                        M ⟵ (M − {(b, g)}) ∪ {(b*, g)}
11:                        b* ⟵ b
12:                        repeat from line 5
13:                end if
14:        end for
15: end for
```

s only s girls are engaged). As the boys take turns, each boy's bookmark is advancing, so some boy's bookmark (among the boys in $\{b_1, \ldots, b_{s+1}\}$) will advance eventually to a point where he must choose a girl.

The discussion in the above paragraph shows that stage $s+1$ in algorithm 1.7 must end. The concern here was that case (ii) of stage $s+1$ might end up being circular. But the fact that the bookmarks are advancing shows that this is not possible.

Furthermore, this gives an upper bound of $(s+1)^2$ steps at stage $(s+1)$ in the procedure. This means that there are n stages, and each stage takes $O(n^2)$ steps, and hence algorithm 1.7 takes $O(n^3)$ steps altogether. The question, of course, is what do we mean by a step? Computers operate on binary strings, yet here the implicit assumption is that we compare numbers and access the lists of preferences in a single step. But the cost of these operations is negligible when compared to our idealized running time, and so we allow ourselves this poetic license to bound the overall running time.

Problem 1.20. *Show that there is exactly one girl that was not engaged at stage s but is engaged at stage $(s+1)$ and that, for each girl g_j that is engaged in M_s, g_j will be engaged in M_{s+1} and that $p_{M_{s+1}}(g_j) <^j p_{M_s}(g_j)$. (Thus, once g_j becomes engaged, she will remain engaged and her partners will only gain in preference as the stages proceed.)*

Problem 1.21. *Suppose that* $|B| = |G| = n$. *Show that at the end of stage* n, M_n *will be a stable marriage.*

We say that a pair (b, g) is *feasible* if there exists a stable matching in which b, g are partners. We say that a matching is boy-*optimal* if every boy is paired with his highest ranked feasible partner. We say that a matching is boy-*pessimal* if every boy is paired with his lowest ranking feasible partner. Similarly, we define girl-*optimal/pessimal*.

Problem 1.22. *Show that our version of the algorithm produces a boy-optimal and girl-pessimal stable matching. Does this mean that they ordering of the boys is irrelevant?*

Problem 1.23. *Implement algorithm 1.7.*

1.2.3 *Pairwise Comparisons*

A fundamental application of algorithmic procedures is to choose the best option from among many. The selection requires a ranking procedure that guides it, but given the complexity of the world in the Information Age, the ranking procedure and selection are often done based on an extraordinary number of criteria. It may also require the chooser to provide a justification for the selection and to convince someone else that the best option has indeed been chosen. For example, imagine the scenario where a team of doctors must decide whether or not to operate on a patient [Kakiashvili *et al.* (2012)], and how important it is to both select the optimal course of action and provide a strong justification for the final selection. Indeed, a justification in this case may be as important as selecting the best option.

Considerable effort has been devoted to research in search engine ranking [Easley and Kleinberg (2010)], in the case of massive amount of highly heterogeneous items. On the other hand, relatively little work has been done in ranking smaller sets of highly similar (homogeneous) items, differentiated by a large number of criteria. Today's state of the art consists of an assortment of domain-specific *ad hoc* procedures, which are highly domain dependent: one approach in the medical profession [Kakiashvili *et al.* (2012)]; another in the world of management [Koczkodaj *et al.* (2014)], etc.

Pairwise Comparisons (PC) has a surprisingly old history for a method that to a certain degree is not widely known. The ancient beginnings are often attributed to a thirteenth century mystic and philosopher Ramon

Lull. In 2001 a manuscript of Llull's was discovered, titled *Ars notandi, Ars eleccionis, and Alia ars eleccionis* (see [Hägele and Pukelsheim (2001); Faliszewski *et al.* (2010)]) where he discussed voting systems and prefigures the PC method. The modern beginnings are attributed to the Marquis de Condorcet (see [Condorcet (1785)], written four years before the French Revolution, and nine years before losing his head to the same). Just as Llull, Condorcet applied the PC method to analyzing voting outcomes. Almost a century and a half later, Thurstone [Thurstone (1927)] refined the method and employed a psychological continuum with the scale values as the medians of the distributions of judgments.

Modern PC can be said to have started with the work of Saaty in 1977 [Saaty (1977)], who proposed a finite nine-point scale of measurements. Furthermore, Saaty introduced the *Analytic Hierarchy Process* (AHP), which is a formal method to derive ranking orders from numerical pairwise comparisons. AHP is widely used around the world for decision making, in education, industry, government, etc. [Koczkodaj (1993)] proposed a smaller five-point scale, which is less fine-grained than Saaty's nine-point, but easier to use. Note that while AHP is a respectable tool for practical applications, it is nevertheless considered by many [Dyer (1990); Janicki (2011)] as a flawed procedure that produces arbitrary rankings.

Let $X = \{x_1, x_2, \ldots, x_n\}$ be a finite set of objects to be ranked. Let a_{ij} express the numerical preference between x_i and x_j. The idea is that a_{ij} estimates "how much better" x_i is compared to x_j. Clearly, for all i, j, $a_{ij} > 0$ and $a_{ij} = 1/a_{ji}$. The intuition is that if $a_{ij} > 1$, then x_i is preferred over x_j by that factor. So, for example, Apple's Retina display has four times the resolution of the Thunderbolt display, and so if x_1 is Retina, and x_2 is Thunderbolt, we could say that the image quality of x_1 is four times better than the image quality of x_2, and so $a_{12} = 4$, and $a_{21} = 1/4$. The assignment of values to the a_{ij}'s are often done subjectively by human judges. Let $A = [a_{ij}]$ be a *pairwise comparison matrix*, also known as a *preference matrix*. We say that a pairwise comparison matrix is *consistent* if for all i, j, k we have that $a_{ij}a_{jk} = a_{ik}$. Otherwise, it is *inconsistent*.

Theorem 1.24 (Saaty). *A pairwise comparison matrix A is consistent if and only if there exist $w_1, w_2, \ldots, w_n$ such that $a_{ij} = w_i/w_j$.*

Problem 1.25. *Note that the w_i's that appear in Theorem 1.24 create a ranking, in that x_j is preferable to x_i if and only if $w_i < w_j$. Suppose that A is a consistent PC matrix. How can you extract the w_i's from A?*

In practice, the subjective evaluations a_{ij} are seldom consistent, which poses a set of problems ([Janicki and Zhai (2011)]), namely, how do we: (i) measure inconsistency and what level is acceptable? (ii) remove inconsistencies, or lower them to an acceptable level? (iii) derive the values w_i starting with an inconsistent ranking A? (iv) justify a certain method for removing inconsistencies? An inconsistent matrix has value in that the degree of inconsistency measures, to some extent, the degree of subjectiveness of the referees. But we need to be able to answer the questions in the above paragraph before we can take advantage in a meaningful way of an inconsistent matrix.

Problem 1.26. *[Bozóki and Rapcsák (2008)] propose several methods for measuring inconsistencies in a matrix (see especially Table 1 on page 161 of their article). Consider implementing some of these measures. Can you propose a method for resolving inconsistencies in a PC matrix?*

1.3 Answers to selected problems

Problem 1.1. $(\forall I \in \mathcal{I}_A)[\exists O(O = A(I)) \wedge (\alpha_A(I) \rightarrow \beta_A(A(I)))]$. This says that for any well formed input I, there is an output, i.e., the algorithm A terminates. This is expressed with $\exists O(O = A(I))$. Also, it says that if the well formed input I satisfies the pre-condition, stated as the antecendent $\alpha_A(I)$, then the output satisfies the post-condition, stated as the consequent $\beta_A(A(I))$.

Problem 1.2. Clearly,

$$an^2 + bn + c \geq an^2 - |b|n - |c| = n^2(a - |b|/n - |c|/n^2) \qquad (1.5)$$

$|b|$ is finite, so $\exists n_b \in \mathbb{N}$ such that $|b|/n_b \leq a/4$. Similarly, $\exists n_c \in \mathbb{N}$ such that $|c|/n_c^2 \leq a/4$. Let $n_0 = max\{n_b, n_c\}$. For $n \geq n_0$, $a - |b|/n_0 - |c|/n_0^2 \geq a - a/4 - a/4 = a/2$. This, combined with (1.5), grants:

$$\frac{a}{2}n^2 \leq an^2 + bn + c$$

for all $n \geq n_0$. We need only to assign c_3 the value $a/2$ to complete the proof that $an^2 + bn + c \in \Omega(n^2)$.

Next we deal with the general polynomial with a positive leading coefficient. Let

$$p(n) = \sum_{i=1}^{k} a_i n^i = n^k \sum_{i=1}^{k} a_i / n^{k-i},$$

where $a_k > 0$. Clearly $p(n) \leq n^k \sum_{i=1}^{k} |a_i|$ for all $n \in \mathbb{N}$, so $p(n) = O(n^k)$. Moreover, every a_i is finite, so for each $i \in \mathbb{N}$ such that $0 \leq i \leq k-1$, $\exists n_i$ such that $a_i/n^{k-i} \leq a_k/2k$ for all $n \geq n_i$. Let n_0 be the maximum of these n_i's. $p(n)$ can be rewritten as $n^k(a_k + \sum_{i=0}^{k-1} a_i/n^{k-i})$, so

$$p(n) \geq n^k(a_k - \sum_{i=0}^{k-1} a_i/n^{k-i}).$$

We have shown that for $n \geq n_0$, $\sum_{i=0}^{k-1} a_i/n^{k-i} \leq a_k - k(a_k/2k) = a_k/2$, so let $c = a_k/2$. For all $n \geq n_0$, $p(n) \geq (a_k - a_k/2)n^k = cn^k$. Thus, $p(n) = \Omega(n^k)$.

We have shown that $p(n) \in O(n^k)$ and $p(n) \in \Omega(n^k)$, so $p(n) = \Theta(n^k)$.

Problem 1.3. The while loop starts with $r = x$, and then y is subtracted each time; this is bounded by x (the slowest case, when $y = 1$). Each time the while loop executes, it tests $y \leq r$, and recomputes r, q, and so it costs 3 steps. Adding the original two assignments ($q \leftarrow 0$,$r \leftarrow x$), we get a total of $3x + 2$ steps. Note that we assume that x, y are presented in binary (the usual encoding), and that it takes $\log_2 x$ bits to encode x, and so the running time is $3 \cdot 2^{\log_2 x} + 2$, i.e., the running time is *exponential* in the length of the input! This is not a desirable running time; if x were big, say 1,000 bits, and y small, this algorithm would take longer than the lifetime of the sun (10 billion years) to end. There are much faster algorithms for division such as the Newton-Raphson method.

Problem 1.4. The original precondition (under which the algorithm is correct) is:

$$x \geq 0 \wedge y > 0 \wedge x, y \in \mathbb{N}$$

where $\mathbb{N} = \{0, 1, 2, \dots\}$. So in the first case our work has already been done for us; any member of $\mathbb{Z}$ which is ≥ 0 is also in $\mathbb{N}$ (and any member of $\mathbb{N}$ is in $\mathbb{Z}$), so these preconditions are equivalent. Given that the algorithm was correct under the original precondition, it is also correct under the new one. In the second case it is not correct: consider $x = -5$ and $y = 2$, so initially $r = -5$, and the loop would not execute, and $r \geq 0$ in the post-condition would not be true.

Problem 1.6. First observe that if u divides x and y, then for any $a, b \in \mathbb{Z}$ u also divides $ax + by$. Thus, if $i|m$ and $i|n$, then

$$i|(m - qn) = r = \text{rem}(m, n).$$

So i divides both n and $\text{rem}(m, n)$, and so i has to be bounded by their greatest common divisor, i.e., $i \leq \gcd(n, \text{rem}(m, n))$. As this is true

for every i, it is in particular true for $i = \gcd(m, n)$; thus $\gcd(m, n) \leq \gcd(n, \text{rem}(m, n))$. Conversely, suppose that $i|n$ and $i|\text{rem}(m, n)$. Then $i|m = qn + r$, so $i \leq \gcd(m, n)$, and again, $\gcd(n, \text{rem}(m, n))$ meets the condition of being such an i, so we have $\gcd(n, \text{rem}(m, n)) \leq \gcd(m, n)$. Both inequalities taken together give us $\gcd(m, n) = \gcd(n, \text{rem}(m, n))$.

Problem 1.7. Let r_i be r after the i-th iteration of the loop. Note that $r_0 = \text{rem}(m, n) = \text{rem}(a, b) \geq 0$, and in fact every $r_i \geq 0$ by definition of remainder. Furthermore:

$$\begin{aligned} r_{i+1} &= \text{rem}(m_{i+1}, n_{i+1}) \\ &= \text{rem}(n_i, r_i) \\ &= \text{rem}(n_i, \text{rem}(m_i, n_i)) \\ &= \text{rem}(n_i, r_i) < r_i. \end{aligned}$$

and so we have a decreasing, and yet non-negative, sequence of numbers; by the LNP this must terminate. To establish the complexity, we count the number of iterations of the while-loop, ignoring the swaps (so to get the actual number of iterations we should multiply the result by two).

Suppose that $m = qn + r$. If $q \geq 2$, then $m \geq 2n$, and since $m \leftarrow n$, m decreases by at least a half. If $q = 1$, then $m = n + r$ where $0 < r < n$, and we examine two cases: $r \leq n/2$, so n decreases by at least a half as $n \leftarrow r$, or $r > n/2$, in which case $m = n + r > n + n/2 = 3/2n$, so since $m \leftarrow n$, m decreases by $1/3$. Thus, it can be said that in all cases at least one element in the pair decreases by at least $1/3$, and so it can be said that the running time is bounded by k such that $3^k = m \cdot n$, and so by $O(\log(m \cdot n)) = O(\log m + \log n)$. As inputs are assumed to be given in binary, we can conclude from this that the running time is linear in the size of the input.

A tighter analysis, known as Lamé's theorem, can be found in [Cormen *et al.* (2009)] (theorem 31.11), which states that for any integer $k \geq 1$, if $a > b \geq 1$ and $b < F_{k+1}$, where F_i is the i-th Fibonacci number (see Problem 9.5), then it takes fewer than k iterations of the while-loop (not counting swaps) to run Eucild's algorithm.

Problem 1.8. When $m < n$ then $\text{rem}(m, n) = m$, and so $m' = n$ and $n' = m$. Thus, when $m < n$ we execute one iteration of the loop only to swap m and n. In order to be more efficient, we could add line 2.5 in algorithm 1.2 saying **if** $(m < n)$ **then** swap(m, n).

Problem 1.9. (a) We show that if $d = \gcd(a, b)$, then there exist u, v such that $au + bv = d$. Let $S = \{ax + by | ax + by > 0\}$; clearly $S \neq \emptyset$. By LNP

there exists a least $g \in S$. We show that $g = d$. Let $a = q \cdot g + r$, $0 \leq r < g$. Suppose that $r > 0$; then

$$r = a - q \cdot g = a - q(ax_0 + by_0) = a(1 - qx_0) + b(-qy_0).$$

Thus, $r \in S$, but $r < g$—contradiction. So $r = 0$, and so $g|a$, and a similar argument shows that $g|b$. It remains to show that g is greater than any other common divisor of a, b. Suppose $c|a$ and $c|b$, so $c|(ax_0 + by_0)$, and so $c|g$, which means that $c \leq g$. Thus $g = \gcd(a, b) = d$.

(b) Euclid's extended algorithm is algorithm 1.8. Note that in the algorithm, the assignments in line 1 and line 8 are evaluated left to right.

Algorithm 1.8 Extended Euclid's algorithm

Pre-condition: $m > 0, n > 0$

1: $a \leftarrow 0$; $x \leftarrow 1$; $b \leftarrow 1$; $y \leftarrow 0$; $c \leftarrow m$; $d \leftarrow n$
2: **loop**
3: $q \leftarrow \text{div}(c, d)$
4: $r \leftarrow \text{rem}(c, d)$
5: **if** $r = 0$ **then**
6: stop
7: **end if**
8: $c \leftarrow d$; $d \leftarrow r$; $t \leftarrow x$; $x \leftarrow a$; $a \leftarrow t - qa$; $t \leftarrow y$; $y \leftarrow b$; $b \leftarrow t - qb$
9: **end loop**

Post-condition: $am + bn = d = \gcd(m, n)$

We can prove the correctness of algorithm 1.8 by using the following loop invariant which consists of four assertions:

$$am + bn = d, \quad xm + yn = c, \quad d > 0, \quad \gcd(c, d) = \gcd(m, n). \qquad \text{(LI)}$$

The basis case:

$$am + bn = 0 \cdot m + 1 \cdot n = n = d$$
$$xm + yn = 1 \cdot m + 0 \cdot n = m = c$$

both by line 1. Then $d = n > 0$ by pre-condition, and $\gcd(c, d) = \gcd(m, n)$ by line 1. For the induction step assume that the "primed" variables are the

result of one more full iteration of the loop on the "un-primed" variables:

$$\begin{aligned} a'm + b'n &= (x - qa)m + (y - qb)n && \text{by line 8} \\ &= (xm - yn) - q(am + bn) \\ &= c - qd && \text{by induction hypothesis} \\ &= r && \text{by lines 3 and 4} \\ &= d' && \text{by line 8} \end{aligned}$$

Then $x'm = y'n = am+bn = d = c'$ where the first equality is by line 8, the second by the induction hypothesis, and the third by line 8. Also, $d' = r$ by line 8, and the algorithm would stop in line 5 if $r = 0$; on the other hand, from line 4, $r = \text{rem}(c, d) \geq 0$, so $r > 0$ and so $d' > 0$. Finally,

$$\begin{aligned} \gcd(c', d') &= \gcd(d, r) && \text{by line 8} \\ &= \gcd(d, \text{rem}(c, d)) && \text{by line 4} \\ &= \gcd(c, d) && \text{see problem 1.6} \\ &= \gcd(m, n). && \text{by induction hypothesis} \end{aligned}$$

For partial correctness it is enough to show that if the algorithm terminates, the post-condition holds. If the algorithm terminates, then $r = 0$, so $\text{rem}(c, d) = 0$ and $\gcd(c, d) = \gcd(d, 0) = d$. On the other hand, by (LI), we have that $am + bn = d$, so $am + bn = d = \gcd(c, d)$ and $\gcd(c, d) = \gcd(m, n)$.

(c) On pp. 292–293 in [Delfs and Knebl (2007)] there is a nice analysis of their version of the algorithm. They bound the running time in terms of Fibonacci numbers, and obtain the desired bound on the running time.

Problem 1.11. For partial correctness of algorithm 1.3, we show that if the pre-condition holds, and *if* the algorithm terminates, then the post-condition will hold. So assume the pre-condition, and suppose first that A is *not* a palindrome. Then there exists a smallest i_0 (there exists one, and so by the LNP there exists a smallest one) such that $A[i_0] \neq A[n - i_0 + 1]$, and so, after the first $i_0 - 1$ iteration of the while-loop, we know from the loop invariant that $i = (i_0 - 1) + 1 = i_0$, and so line 4 is executed and the algorithm returns F. Therefore, "A not a palindrome" $\Rightarrow$ "return F."

Suppose now that A *is* a palindrome. Then line 4 is never executed (as no such i_0 exists), and so after the $k = \lfloor \frac{n}{2} \rfloor$-th iteration of the while-loop, we know from the loop invariant that $i = \lfloor \frac{n}{2} \rfloor + 1$ and so the while-loop is not executed any more, and the algorithm moves on to line 8, and returns T. Therefore, "A is a palindrome" $\Rightarrow$ "return T."

Therefore, the post-condition, "return T iff A is a palindrome," holds. Note that we have only used part of the loop invariant, that is we used the fact that after the k-th iteration, $i = k+1$; it still holds that after the k-th iteration, for $1 \leq j \leq k$, $A[j] = A[n-j+1]$, but we do not need this fact in the above proof.

To show that the algorithm terminates, let $d_i = \lfloor \frac{n}{2} \rfloor - i$. By the pre-condition, we know that $n \geq 1$. The sequence $d_1, d_2, d_3, \ldots$ is a decreasing sequence of positive integers (because $i \leq \lfloor \frac{n}{2} \rfloor$), so by the LNP it is finite, and so the loop terminates.

Problem 1.12. It is very easy once you realize that in Python the slice `[::-1]` generates the reverse string. So, to check whether string `s` is a palindrome, all we do is write `s == s[::-1]`.

Problem 1.13. The solution is given by algorithm 1.9

Algorithm 1.9 Powers of 2

Pre-condition: $n \geq 1$
  $x \leftarrow n$
  while $(x > 1)$ **do**
    if $(2|x)$ **then**
      $x \leftarrow x/2$
    else
      stop and return "no"
    end if
  end while
  return "yes"
Post-condition: "yes" $\iff$ n is a power of 2

Let the loop invariant be: "x is a power of 2 iff n is a power of 2."

We show the loop invariant by induction on the number of iterations of the main loop. Basis case: zero iterations, and since $x \leftarrow n$, $x = n$, so obviously x is a power of 2 iff n is a power of 2. For the induction step, note that if we ever get to update x, we have $x' = x/2$, and clearly x' is a power of 2 iff x is. Note that the algorithm always terminates (let $x_0 = n$, and $x_{i+1} = x_i/2$, and apply the LNP as usual).

We can now prove correctness: if the algorithms returns "yes", then after the last iteration of the loop $x = 1 = 2^0$, and by the loop invariant n is a power of 2. If, on the other hand, n is a power of 2, then so is every x, so eventually $x = 1$, and so the algorithm returns "yes".

Problem 1.14. Algorithm 1.4 computes the product of m and n, that is, the returned $z = m \cdot n$. A good loop invariant is $x \cdot y + z = m \cdot n$.

Problem 1.17. We start by initializing all nodes to have rank 1/6, and then repeatedly apply the following formulas, based on (1.4):

$$\begin{aligned}
\mathrm{PR}(A) &= \mathrm{PR}(F) \\
\mathrm{PR}(B) &= \mathrm{PR}(A) \\
\mathrm{PR}(C) &= \mathrm{PR}(B)/4 + \mathrm{PR}(E) \\
\mathrm{PR}(D) &= \mathrm{PR}(B)/4 \\
\mathrm{PR}(E) &= \mathrm{PR}(B)/4 + \mathrm{PR}(D) \\
\mathrm{PR}(F) &= \mathrm{PR}(B)/4 + \mathrm{PR}(C)
\end{aligned}$$

The result is given in figure 1.3.

	0	1	2	3	4	5	6	...	17
A	0.17	0.17	0.21	0.25	0.29	0.18	0.20		0.22
B	0.17	0.17	0.17	0.21	0.25	0.29	0.18		0.22
C	0.17	0.21	0.25	0.13	0.14	0.16	0.19	...	0.17
D	0.17	0.04	0.04	0.04	0.05	0.06	0.07		0.06
E	0.17	0.21	0.08	0.08	0.09	0.11	0.14		0.11
F	0.17	0.21	0.25	0.29	0.18	0.20	0.23		0.22
Total	1.00	1.00	1.00	1.00	1.00	1.00	1.00	...	1.00

Fig. 1.3 Pagerank convergence in Problem 1.17. Note that the table is obtained with a spreadsheet: all values are rounded to two decimal places, but column 1 is obtained by placing 1/6 in each row, column 2 is obtained from column 1 with the formulas, and all the remaining columns are obtained by "dragging" column 2 all the way to the end. The values converged (more or less) in column 17.

Problem 1.19. After b proposed to g for the first time, whether this proposal was successful or not, the partners of g could have only gotten better. Thus, there is no need for b to try again.

Problem 1.20. b_{s+1} proposes to the girls according to his list of preference; a g ends up accepting, and if the g who accepted b_{s+1} was free, she is the new one with a partner. Otherwise, some $b^* \in \{b_1, \ldots, b_s\}$ became disengaged, and we repeat the same argument. The g's disengage only if a better b proposes, so it is true that $p_{M_{s+1}}(g_j) <^j p_{M_s}(g_j)$.

Problem 1.21. Suppose that we have a blocking pair $\{b, g\}$ (meaning that $\{(b, g'), (b', g)\} \subseteq M_n$, but b prefers g to g', and g prefers b to b'). Either b

came after b' or before. If b came before b', then g would have been with b or someone better when b' came around, so g would not have become engaged to b'. On the other hand, since (b', g) is a pair, no better offer has been made to g after the offer of b', so b could not have come after b'. In either case we get an impossibility, and so there is no blocking pair $\{b, g\}$.

Problem 1.22. To show that the matching is boy-optimal, we argue by contradiction. Let "*g is an optimal partner for b*" mean that among all the stable matchings g is the best partner that b can get.

We run the Gale-Shapley algorithm, and let b be the first boy who is rejected by his optimal partner g. This means that g has already been paired with some b', and g prefers b' to b. Furthermore, g is at least as desirable to b' as his own optimal partner (since the proposal of b is the first time during the run of the algorithm that a boy is rejected by his optimal partner). Since g is optimal for b, we know (by definition) that there exists some stable matching S where (b, g) is a pair. On the other hand, the optimal partner of b' is ranked (by b' of course) at most as high as g, and since g is taken by b, whoever b' is paired with in S, say g', b' prefers g to g'. This gives us an unstable pairing, because $\{b', g\}$ prefer each other to the partners they have in S.

Yes, this means that the ordering of the boys is immaterial, because there is a unique boy-optimal matching, and it is independent of the ordering of the boys.

To show that the Gale-Shapley algorithm is girl-pessimal, we use the fact that it is boy-optimal (which we just showed). Again, we argue by contradiction. Suppose there is a stable matching S where g is paired with b, and g prefers b' to b, where (b', g) is the result of the Gale-Shapley algorithm. By boy-optimality, we know that in S we have (b', g'), where g' is not higher on the preference list of b' than g, and since g is already paired with b, we know that g' is actually lower. This says that S is unstable since $\{b', g\}$ would rather be together than with their partners.

1.4 Notes

This book is about proving things about algorithms; their correctness, their termination, their running time, etc. The art of mathematical proofs is a difficult art to master; a very good place to start is [Velleman (2006)].

On page vii we mentioned the North-East blackout of 2003. At the time the author was living in Toronto, Canada, on the 14th floor of an apartment

building (which really was the 13th floor, but as number 13 was outlawed in Toronto elevators, after the 12th floor, the next button on the elevator was 14). After the first 24 hours, the emergency generators gave out, and we all had to climb the stairs to our floors; we would leave the building, and scavenge the neighborhood for food and water, but as refrigeration was out in most places, it was not easy to find fresh items. In short, we really felt the consequences of that algorithmic error intimately.

In the footnote to Problem 1.10 we mention the Python library `matplotlib`. Below we provide a simple example, plotting the functions $f(x) = x^3$ and $h(x) = -x^3$ over the interval $[0, 10]$ using this library:

```
import matplotlib.pyplot as plt
import numpy as np

def f(x):
    return x**3
def h(x):
  return -x**3

Input = np.arange(0,10.1,.5)
Outputf = [f(x) for x in Input]
Outputh = [h(x) for x in Input]

plt.plot(Input,Outputf,'r.',label='f - label')
plt.plot(Input,Outputh,'b--',label='h - label')
plt.xlabel('This is the X axis label')
plt.ylabel('This is the Y axis label')
plt.suptitle('This is the title')
plt.legend()
plt.show()
```

Of course, `matplotlib` has lots of features; see the documentation for more complex examples.

The palindrome `madamimadam` comes from Joyce's *Ulysses*. We discussed the string manipulating facilities of Python in the section on palindromes, Section 1.1.4, but perhaps the most powerful language for string manipulations is Perl. For example, suppose that we have a text that contains hashtags which are words of characters that start with '`#`', and we wish to collect all those hashtags into an array. One trembles at the prospect of having to implement this in, say, the C programming language, but in Perl

this can be accomplished in one line:

```
@TAGS = ($TEXT =~ m/\#([a-zA-Z0-9]+)/g);
```

where `$TEXT` contains the text with zero or more hashtags, and the array `@TAGS` will be a list of all the hashtags that occur in `$TEXT` without the '`#`' prefix. For the great pleasure of Perl see [Schwartz *et al.* (2011)].

Search engines are complex and vast software systems, and ranking pages is not the only technical issue that has to be solved. For example, parsing keywords to select relevant pages (pages that contain the keywords), before any ranking is done on these pages, is also a challenging task: the search system has to solve many problems, such as *synonymy* (multiple ways to say the same thing) and *polysemy* (multiple meanings), and many others. See [Miller (1995)].

Section 1.2.2 is based on §2 in [Cenzer and Remmel (2001)]. For another presentation of the Stable Marriage problem see chapter 1 in [Kleinberg and Tardos (2006)]. The reference to the Marquis de Condorcet in the first sentence of section 1.2.2 comes from the PhD thesis of Yun Zhai ([Zhai (2010)]), written under the supervision of Ryszard Janicki. In that thesis, Yun Zhai references [Arrow (1951)] as the source of the remark regarding the Marquis de Condorcet's early attempts at pairwise ranking. There is a wonderfully biting description of Condorcet and his ideas in Roger Kimball's *The Fortunes of Permanence* [Kimball (2012)], pp. 237–244. Condercet may have given us the method of Pairwise Comparisons, but he was a tragic figure of the Enlightenment: he promised "*perfectionnement même de l'espèce humaine*" ("the absolute perfection of the human race"), but his utopian ideas were the precursor of countless hacks who insisted on perfecting man whether he wanted it or not, ushering in the inevitable tyrannical excesses that are the culmination of utopian dreams.

Professor Thomas L. Saaty (Theorem 1.24) died on August 14, 2017. He was a distinguished professor at the University of Pittsburghs Katz School of Business. The government of Poland gave Prof. Saaty a national award after its use of his theory AHP for making decisions resulted in the country initially not joining the European Union.