CHAPTER 5. ESTIMATING THE GROWTH RATE OF A FUNCTION

5.1 The definition of a good and a bad estimate for a function

As I mentioned at the end of Chapter 3, you need to study functions because the time taken to execute a program depends upon the size of the input you enter into the program. Of course, it also depends upon the computer you are using – the latest supercomputer will execute the same program on the same input faster than some old vacuum-tube antique – so to make your analysis independent of the computer you use, instead of measuring the time taken to execute the program you count the number of operations the computer will perform while executing the program. And to make your analysis independent of the computer language you use, instead of programs you consider algorithms, which are like programs except that they are written in human language instead of a computer language.

But you don't count the exact number of operations because the formula for the exact number of operations is usually too complicated to be useful for comparing the speed of different algorithms for solving the same problem. Suppose, for example, that you want to sort an array of size n and you have a choice of two algorithms: algorithm S that will do $2n^2 - n + 3$ operations and algorithm M that will do $10n \log_2 n + 8n + 2$ operations (I'm making the coefficients up). Which one should you make into a program? You won't find either of these functions in a table because they don't make tables with an entry for each list of coefficients; so, being a practical sort of person, you program both algorithms and, to avoid using up valuable computer time, you run both programs on an array of size 8. Algorithm S does 123 operations and algorithm M does 248 operations; so you conclude that algorithm S is faster and you run it on the array you really want to sort, which has 1000 elements. The computer does 1,999,003 operations, and if it's as slow as an Apple 2e, which does about 1000 operations a second, it takes about half an hour to sort your array. How long would algorithm M have taken? Well, the lunatic who thought that $2^{10} = 1000$ wasn't that far off: $2^{10} = 1024$ (which is, after all, why they call 2^{10} bytes a kilobyte); so $\log_2 1000$ is close to but less than 10 and algorithm M would have done less than 108,002 operations and taken less than 2 minutes. You could have saved a half an hour of computing time, not to mention the time you took to program algorithm S, if only you had known how to estimate how fast a function of n grows as n gets big.

Given a function $f(n)$, which represents the number of operations an algorithm will do on an input of size n, you need to find a function $g(n)$ which is a good estimate for $f(n)$ for large n and which is simple enough that you can easily compare the growth rate of functions by comparing their estimates. What does it mean that $g(n)$ is an estimate for $f(n)$?

You could insist that $f(n) \le g(n)$ for all n, but that's asking for too much. In that case, to estimate $2n^2 - n + 3$ you'd need to use a function like $2n^2 + 3$, and that function is still too complicated. And it isn't necessary for $f(n)$ to be at most $g(n)$ **for all *n*** because you only care about what happens **when *n* gets big**; any algorithm will run quickly if n is small.

Suppose you insist instead that $f(n) \le g(n)$ for all big enough n. To express that idea mathematically, you say that there must exist some positive real number k, called a *threshold*, such that if $n \ge k$, then $f(n) \le g(n)$. In that case, to estimate $2n^2 - n + 3$ you could use a function like $5n^2$. If $n \ge 1$ (the threshold), then $n^2 \ge n \ge 1$, so that $2n^2 - n + 3 \ge 2n^2 - 0 + 3n^2 = 5n^2$.

But even $5n^2$ is unnecessarily complicated. The function $10n \log_2 n + 8$ has bigger coefficients than 5, but it's smaller than $5n^2$ for big enough n. In general, multiplying a function by a constant is less important when n gets big than changing the function to one that grows at a different rate; so for the sake of simplicity we ignore constant multipliers.

To this end, instead of insisting that $f(n) \le g(n)$ for all n big enough, we will be satisfied if there is **some** positive real number c, the *factor*, such that if n is big enough, then $f(n) \le cg(n)$. Returning to our function $2n^2 - n + 3$, we have shown that if $n \ge 1$, then $2n^2 - n + 3 \le 5n^2$; so for the factor $c = 5$ and the threshold $k = 1$, for all $n \ge k$, $2n^2 - n + 3 \le cn^2$. The function n^2 is simple enough and it gives enough information to be able to estimate the growth rate of $2n^2 - n + 3$.

Formalizing the definition of estimate implied by that example, we say that $g(n)$ is an *estimate* of $f(n)$ if there exist two positive real numbers k (the *threshold*) and c (the *factor*) such that for all $n \ge k$, $f(n) \le cg(n)$. We denote the proposition that $g(n)$ is an estimate of $f(n)$ by writing $f(n) \in O(g(n))$. In mathematical terminology, then,

$$f(n) \in O(g(n)) \Leftrightarrow (\exists k > 0)(\exists c > 0)(\forall n \ge k)(f(n) \le cg(n)).$$

Some of you may have seen a definition of O that uses absolute values. Why did I use a definition that doesn't use absolute values? Recall that $f(n)$ represents the number of operations an algorithm will do on an input of size n, and algorithms don't usually do zero operations or fewer. In order for a positive $f(n)$ to be at most $cg(n)$ for large n and positive c, $g(n)$ has to be positive for large n. With these restrictions, we can dispense with absolute values in the definition of O.

Is n^3 an estimate for $2n^2 - n + 3$? Well, we have already shown that for all $n \ge 1$, $2n^2 - n + 3 \le 5n^2$. If $n \ge 1$, then $n^3 \ge n^2$; so $2n^2 - n + 3 \le 5n^3$. This means that n^3 is also an estimate for $2n^2 - n + 3$. But is it a good estimate? Recall that when $n = 1000$, $2n^2 - n + 3 =$ 1,999,003, and an Apple 2e will sort a size-1000 array in about half an hour using algorithm S. When $n = 1000$, $n^3 = 1{,}000{,}000{,}000$, and if algorithm S did 1,000,000,000 operations to sort a size-1000 array, then your Apple would take about ten days to do it! If you discovered algorithm S and submitted an article saying that its running time is in $O(n^3)$, then the referee of your article would conclude that S stands for "slow" and reject the article after first showing it to all his colleagues so that they could laugh at you. Evidently, n^3 is a bad estimate for $2n^2 - n + 3$.

On the other hand, for large n, $2n^2 - n + 3$ is about twice as big as n^2 (for $n = 1000$, $2n^2 - n + 3 = 1{,}999{,}003$ and $n^2 = 1{,}000{,}000$), so that algorithm S takes only about twice as long to run as it would if its real running time were n^2 operations. If you said that the running time was in $O(n^2)$, then the referee would be prepared to excuse you for underestimating the running time by a constant factor. Intuitively, then, n^2 is a good estimate for $2n^2 - n + 3$.

Given that $g(n)$ is an estimate for $f(n)$, what sort of mathematical definition could we give for $g(n)$ to be a good estimate or a bad estimate for $f(n)$? One possibility would be to define $g(n)$

to be a *good estimate* for $f(n)$ if $f(n)$ is also an estimate for $g(n)$ and a *bad estimate* otherwise. Does this definition make n^3 a bad estimate for $2n^2 - n + 3$ and n^2 a good estimate?

For n^3 to be a bad estimate for $2n^2 - n + 3$ under this definition, for any threshold $k > 0$ and any factor $c > 0$, there would have to exist an $n \geq k$ such that $n^3 > c(2n^2 - n + 3)$. If $n \geq 3$, $2n^2 - n + 3 \leq 2n^2$. If in addition $n > 2c$, then $n^3 > 2cn^2 \geq c(2n^2 - n + 3)$. Given any $k > 0$ and $c > 0$, let $n = \max(3,k,2c+1)$. Then $n \geq k$, and since $n \geq 3$ and $n > 2c$, $n^3 > c(2n^2 - n + 3)$; so under this definition, n^3 is indeed a bad estimate for $2n^2 - n + 3$.

For n^2 to be a good estimate for $2n^2 - n + 3$ under this definition, there must be a threshold $k > 0$ and a factor $c > 0$ such that for any $n \geq k$, $n^2 \leq c(2n^2 - n + 3)$. If $n \geq 1$, $n^2 \geq n$ so that $2n^2 - n + 3 \geq 2n^2 - n^2 + 3 = n^2 + 3 \geq n^2$, so that with threshold $k = 1$ and factor $c = 1$, the definition is satisfied and n^2 is indeed a good estimate for $2n^2 - n + 3$.

Having found a good estimate (n^2) for $2n^2 - n + 3$, the number of operations needed by algorithm S to sort a size-n array, let's try to find one for $10n\log_2 n + 8n + 2$, the number of operations needed by algorithm M. If $n \geq 2$, $\log_2 n \geq 1$, so that $n\log_2 n \geq n \geq 1$. Therefore, $10n\log_2 n + 8n + 2 \leq 10n\log_2 n + 8n\log_2 n + 2n\log_2 n = 20n\log_2 n$, so that with threshold $k = 2$ and factor $c = 20$, $n\log_2 n$ is an estimate for $10n\log_2 n + 8n + 2$. Is it a good estimate? Well, if $n \geq 1$, $\log_2 n \geq 0$, so that $n\log_2 n \leq 10n\log_2 n \leq 10n\log_2 n + 8n + 2$, and the definition of estimate is satisfied with threshold $k = 1$ and factor $c = 1$.

Now that we have found good estimates for the running times our our two sorting algorithms (n^2 for algorithm S and $n\log_2 n$ for algorithm M), how can we use this information to help us decide which algorithm will run faster for big n? We recall that in Chapter 4 we proved by induction that $2^n \geq n$ for every integer $n \geq 0$. Taking logarithms to base 2, we find that $n \geq \log_2 n$ for every integer $n \geq 1$, so that $\log_2 n \in O(n)$, meaning that algorithm M is at least as rapid as algorithm S for big enough n to within some constant factor. But by our unfortunate experiment we have found that for big n, algorithm M is considerably **more** efficient than algorithm S. How could we have predicted this by using our estimates for the running times of those two algorithms? By substituting some big value of n, of course, but since any n we choose may not be big enough, we would have to prove that $n \notin O(\log_2 n)$. Most elementary textbooks do not prove this proposition because it is difficult to do directly from the definition of O.

In the next chapter I will present two algorithms for solving the same problem, one whose running time has a good estimate of $\sqrt{n}$ and the other whose running time has a good estimate of $\log_2 n$. To choose between these two algorithms, we would have to know that $\log_2 n \in O(\sqrt{n})$ and that $\sqrt{n} \notin O(\log_2 n)$. In the next section I will prove a general theorem that says that for any real $c > 0$ and $b > 1$, n^c is a bad estimate for $\log_b n$ and b^n is a bad estimate for n^c so that, in particular, the algorithm that runs in $\log_2 n$ time is the one to choose. I did prove that theorem directly from the definitions of good and bad estimate, but the proof occupies several pages and I've never met another professor, let alone a student, who would have the patience to do so. Using the idea of limits from calculus, that theorem can be proved in a few lines and the sort of results we obtained in this section can be derived without the guesswork we had to use here.

5.2 Finding and comparing good estimates using limits

You may have noticed that although the growth rate of a function appears in the title and in a couple of places in Section 5.1, I haven't yet defined it. To do so, I have to use the concept of **limit** from calculus. We say that $f(n)$ *tends to* the number c as n tends to infinity if for every positive error-tolerance ε, there is a positive threshold k such that for all $n \geq k$, $f(n)$ differs from c by at most ε. And we say that $f(n)$ tends to infinity as n tends to infinity if for every positive lower bound b there is a threshold k such that for all $n \geq k$, $f(n) \geq b$. In mathematical notation:

$$\lim_{n\to\infty} f(n) = c \Leftrightarrow (\forall e > 0)(\exists k > 0)(\forall n \geq k)(|\, f(n) - c \,| \leq \varepsilon);$$

$$\lim_{n\to\infty} f(n) = \infty \Leftrightarrow (\forall b > 0)(\exists k > 0)(\forall n \geq k)(f(n) \geq b).$$

I implicitly used this idea to show that for every real number $b > 1$ the function $f(x) = b^x$ is surjective on the set of positive real numbers. As x tends to infinity, b^x tends to infinity, and as x tends to minus infinity, b^x tends to zero. That is why for every real number $y > 0$ there is an x small enough that $b^x < y$ and another x large enough that $b^x > y$, so that there must be a third x between the other two such that $b^x = y$.

We now have the tools to compare the growth rates of functions. We say that $f(n)$ *grows faster than* $g(n)$ if $f(n)/g(n)$ tends to infinity as n tends to infinity. We say that $f(n)$ *grows slower than* $g(n)$ if $f(n)/g(n)$ tends to zero as n tends to infinity. Recalling that $f(n)$ and $g(n)$ are both positive for big enough n, we say that $f(n)$ and $g(n)$ grow *at the same rate* if $f(n)/g(n)$ tends to some positive constant as n tends to infinity. For example, since $\lim_{n\to\infty}(n^3/n^2) = \lim_{n\to\infty}(n) = \infty$, n^3 grows faster than n^2. And since $\lim_{n\to\infty}(n^2/n^3) = \lim_{n\to\infty}(1/n) = 0$, n^2 grows slower than n^3. And since

$$\lim_{n\to\infty}\left((2n^2 - n + 3)/n^2\right) = \lim_{n\to\infty}(2n^2/n^2) - \lim_{n\to\infty}(n/n^2) + \lim_{n\to\infty}(3/n^2) = 2 - 0 + 0 = 2,$$

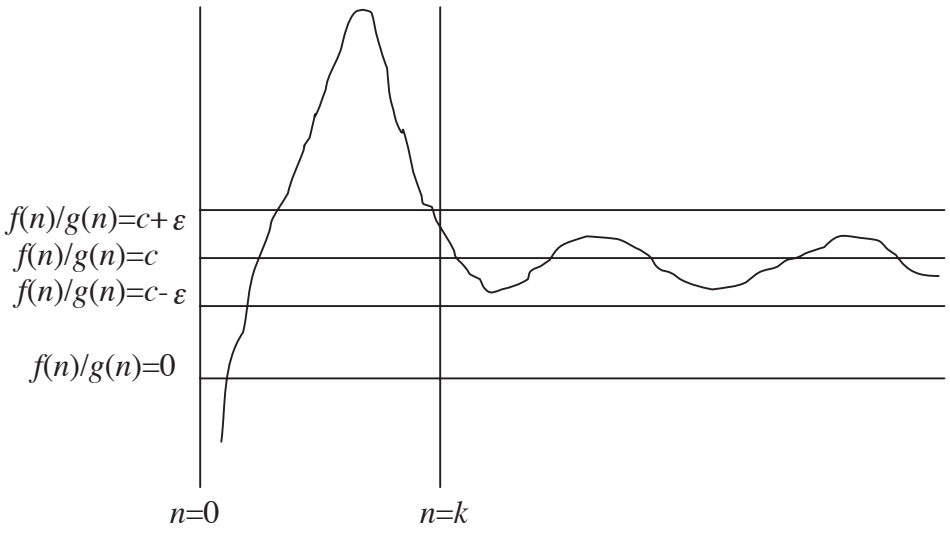
$2n^2 - n + 3$ grows at the same rate as n^2.

You may have noticed that we have already used these functions of n in the previous section. You may also have noticed that n^2, which grows at the same rate as $2n^2 - n + 3$, is a good estimate for $2n^2 - n + 3$, and that n^3, which grows faster than n^2 and therefore faster than $2n^2 - n + 3$, is a bad estimate for $2n^2 - n + 3$. The following theorem generalizes these observations and shows how limits can be used to deduce facts about estimates.

Theorem. Let $f(n)$ and $g(n)$ be two functions from the set of natural numbers (or from the set of non-negative real numbers) to the set of real numbers that are positive for large n.

a) If $f(n)$ and $g(n)$ grow at the same rate, then $g(n)$ is a good estimate for $f(n)$.
b) If $f(n)$ grows slower than $g(n)$, then $g(n)$ is a bad estimate for $f(n)$.
c) If $f(n)$ grows faster than $g(n)$, then $g(n)$ is not an estimate for $f(n)$.

Proof. Since $g(n)$ is positive for large n, we can assume that the threshold k used in the definition of O is big enough that $g(n) > 0$ for all $n \geq k$ and we can divide by $g(n)$ without fear of dividing by zero. Then $g(n)$ is an estimate for $f(n)$ means that there is a threshold k such that $f(n)/g(n)$ is bounded above by some positive constant c for all $n \geq k$ (see the diagram below).

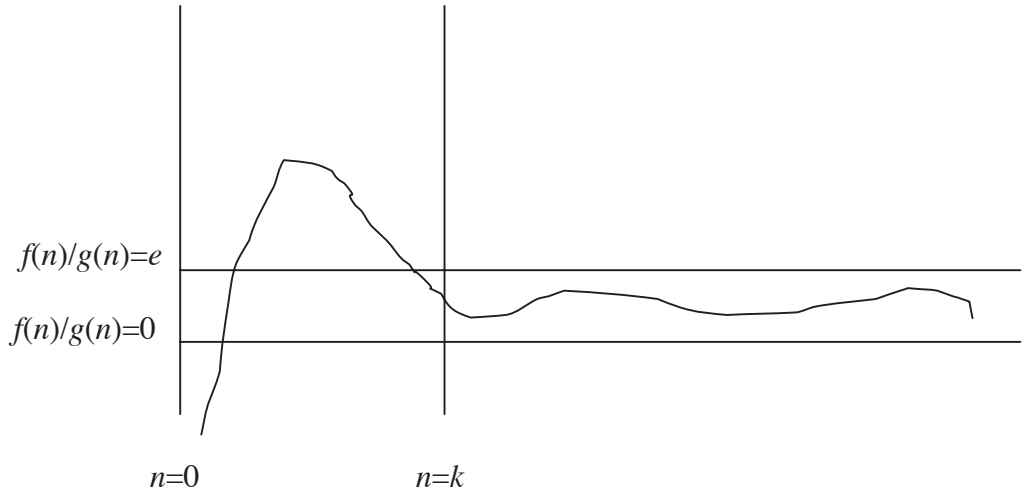


If $f(n)$ and $g(n)$ grow at the same rate, then $g(n)$ is an estimate for $f(n)$.

Suppose that $f(n)$ and $g(n)$ grow at the same rate. Then for any error-tolerance $\varepsilon > 0$ there is a threshold k such that $f(n)/g(n)$ is bounded between $c + \varepsilon$ and $c - \varepsilon$ for some positive constant c for all $n \geq k$ (see the diagram above). Pick any $\varepsilon > 0$; then there is a threshold k such that $f(n)/g(n)$ is bounded above by $c + \varepsilon$ for all $n \geq k$, so that $g(n)$ is an estimate for $f(n)$.

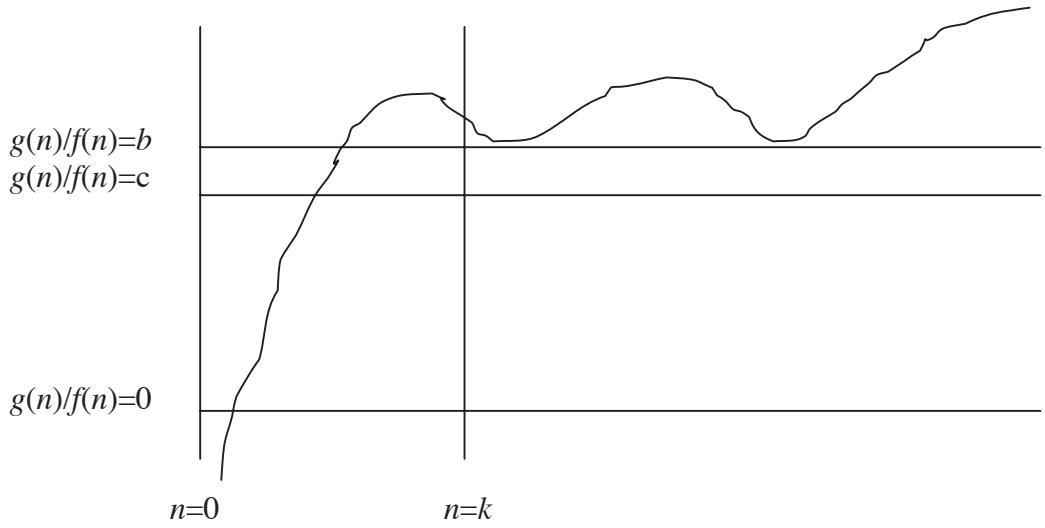
How do we know that $g(n)$ is a **good** estimate for $f(n)$? Well, since $f(n)/g(n)$ tends to c, a positive constant, $g(n)/f(n)$ tends to $1/c$, another positive constant; so $f(n)$ is an estimate for $g(n)$, which means that $g(n)$ is a good estimate for $f(n)$.

Now suppose that $f(n)$ grows slower than $g(n)$. Then for any error-tolerance $\varepsilon > 0$ there is a threshold k such that $f(n)/g(n)$ is bounded between $+\varepsilon$ and $-\varepsilon$ for all $n \geq k$. But since we are assuming that $f(n)$ and $g(n)$ are positive for big n, $f(n)/g(n)$ is bounded between ε and 0 (see the diagram below). Pick any $\varepsilon > 0$; then there is a threshold k such that $f(n)/g(n)$ is bounded above by e for all $n \geq k$, so that $g(n)$ is an estimate for $f(n)$.



If $f(n)$ grows slower than $g(n)$, then $g(n)$ is an estimate for $f(n)$.

How do we know that $g(n)$ is a **bad** estimate for $f(n)$? Well, since $f(n)/g(n)$ tends to zero, $g(n)/f(n)$ tends to infinity. This means that for any lower bound b, there is a threshold k such that $g(n)/f(n) \geq b$ for all $n \geq k$ (see diagram below). For $f(n)$ to be an estimate for $g(n)$, $g(n)/f(n)$ would have to be bounded **above** by some positive number c for big n, but instead $g(n)/f(n)$ is bounded **below** by **any** lower bound b you want to choose; so all you have to do is choose some $b > c$ and $g(n)/f(n)$ will be **bigger** than c instead of less than or equal to c for big n. This means that $f(n)$ is not an estimate for $g(n)$, so that $g(n)$ is a bad estimate for $f(n)$.



If $f(n)$ grows slower than $g(n)$, then $f(n)$ is not an estimate for $f(n)$.

Finally, if $f(n)$ grows faster than $g(n)$, then $g(n)$ grows slower than $f(n)$, so that $g(n)$ is not an estimate for $f(n)$ (but $f(n)$ is an estimate for $g(n)$ – a bad one). QED.

We are now in a position to prove the theorem promised in the previous section.

Theorem. For any real $c > 0$ and $b > 1$, n^c is a bad estimate for $\log_b n$ and b^n is a bad estimate for n^c.

Proof. According to the previous theorem, all we have to prove is that n^c grows faster than $\log_b n$ but slower than b^n. To do so, we use *l'Hospital's rule*, not the one that says that if you enter a hospital you will probably get sick, but the one that says that $f(n)/g(n)$ tends to the same limit as $f'(n)/g'(n)$ (the apostrophe means you take the derivative). The derivative of n^c is cn^{c-1} and the derivative of $\log_b n = \ln n / \ln b$ is $1/(n \ln b)$ ($\ln(n)$ means $\log_e(n)$); so $n^c / \log_b n$ tends to the same limit as $cn^{c-1}/(1/(n \ln b)) = (c \ln b)n^c$, which is infinity because $c > 0$. This proves that n^c grows faster than $\log_b n$; now we will prove that n^c grows slower than b^n. The derivative of $b^n = e^{n \ln b}$ is $(\ln b)e^{n \ln b} = (\ln b)b^n$; so n^c / b^n tends to the same limit as $cn^{c-1}/(\ln b)b^n$. The first application of l'Hospital's rule diminishes the exponent of n by 1 and multiplies the fraction by a constant but leaves all n factors b. Applying this rule at least c times will make the exponent of n zero or less and multiply the fraction by a constant and there still will be all n factors b. Since $b > 1$, the fraction will tend to 0. This proves that n^c grows slower than b^n. **End of proof**!

If you really want to crack your teeth, try proving this theorem directly from the definition of O without using any results that need limits to derive.

Now suppose we have some complicated function $f(n)$ and we want to find a simpler function $g(n)$ that is a **good** estimate for $f(n)$. We use the above two theorems together with the following two theorems.

Theorem. If $f_1(n)$ grows at least as fast as $f_2(n)$, then $f_1(n)$ and $f_1(n) + f_2(n)$ grow at the same rate.

Proof. Always assuming that all the functions are positive for big n so that we can divide by them, $(f_1(n) + f_2(n))/f_1(n) = 1 + f_2(n)/f_1(n)$. Since $f_2(n)/f_1(n)$ tends to a **non-negative** constant c, (a positive constant if $f_1(n)$ grows at the same rate as $f_2(n)$ or 0 if $f_1(n)$ grows faster than $f_2(n)$), $(f_1(n) + f_2(n))/f_1(n)$ tends to the **positive** constant $c+1$, and the result follows from the first theorem of this section, QED.

This theorem can be easily extended to the sum of any number of functions, as long as this number is independent of n. This implies that to get a good estimate for the sum of several functions, just take the function (or functions) that grow the fastest and find a simple function $g(n)$ that grows at the same rate; then $g(n)$ will serve as a good estimate for the whole sum. For example, take any polynomial with positive coeffients: $a_m n^m + ... + a_0$. The term that grows the fastest is $a_m n^m$ and it grows as the same rate as n^m; so n^m is a good estimate for the polynomial. We can relax the restriction that all the coefficients be positive because the terms with zero or negative coefficients will be zero or negative for all $x > 0$. It suffices that a_m be positive. A special case is the polynomial $2n^2 - n + 3$, for which we found the good estimate n^2 in section 5.1 by guessing a threshold and a factor.

In general, though, you have to be careful about subtraction. You have to make sure that the fastest growing terms don't cancel each other out or leave a function that is negative for big n. For example, take the function $f(n) = (n+1)^2 - n^2$. Since n grows faster than 1, it would be tempting to simplify $n + 1$ to n, which would then simplify $f(n)$ to 0, but 0 is not an estimate for $f(n)$. Since there is a subtraction, you have to expand $(n+1)^2$ to see what happens to the fastest growing terms. In this way we find that $f(n) = (n^2 + 2n + 1) - n^2 = 2n + 1$. The fastest growing terms do cancel each other out, and of the remaining terms, $2n$ grows faster than 1; so n is a good estimate for $f(n)$.

Did you ever hear the story about the little tailor who killed seven flies and boasted that he was the strongest creature on Earth? His claim was contested by two giants. He certainly wasn't stronger than they were, but he was smarter. He hid from them until they fell asleep, and then he pocketed some stones and climbed a tree directly above them. Then he threw a stone at each of the giants, waking them both up. Each giant thought that the other one had thrown a stone at him; so they fought until they were both dead. The tailor claimed credit for having killed the giants, making him indeed the strongest creature on Earth.

I read once that children will accept any joke during a lesson, however irrelevant the joke is to the material being taught in the lesson, whereas adult students insist that jokes be relevant. So what is the relevance of that story? Well, the two copies of n^2, the fastest growing terms, are the two giants that annihilate each other, while, of the remaining terms, $2n$, which grows faster than 1, is the little tailor and 1 is the 7 flies he killed, making him the champion worthy of imposing his good estimate n on the entire function!

Some elementary textbooks present the following analogue of the above theorem: if $g_1(n)$ is an estimate for $f_1(n)$ and $g_2(n)$ is an estimate for $f_2(n)$, then $\max(g_1(n), g_2(n))$ is an estimate for $f_1(n) + f_2(n)$. This is true, but it says nothing about good estimates. Besides, it has another disadvantage, as the following example will illustrate. Suppose you want to find an estimate for $n^2 + 2^n$. To use this version of the theorem, you have to find $\max(n^2, 2^n)$. When $n = 5$, $\max(n^2, 2^n) = 2^n$ because $2^5 = 32$ whereas $5^2 = 25$, but when $n = 3$, $\max(n^2, 2^n) = n^2$ because $3^2 = 9$ whereas $2^3 = 8$. Neither n^2 nor 2^n is equal to $\max(n^2, 2^n)$ for all n; so you'd need to use $\max(n^2, 2^n)$ itself as an estimate, but that is hardly a simple function! Of course, for estimates all you care about is which one is bigger for big enough n. If you redefine $\max(g_1(n), g_2(n))$ to be the one that is bigger for all big enough n, assuming that one of them is, then you can choose 2^n as the estimate – as soon as you have proved that $2^n \geq n^2$ for all $n \geq 4$. How to do so? By induction - try it! Of course, if you know that 2^n grows faster than n^2 (a special case of a result we already proved), you can conclude that $2^n \geq n^2$ for all big enough n, so that 2^n is a estimate for $n^2 + 2^n$. But once you know that 2^n grows faster than n^2, you can already conclude that 2^n is a **good** estimate for $n^2 + 2^n$ using the results of this section!

Having treated sums and differences of functions, let's try our hand at products and quotients. Some elementary textbooks prove that if $g_1(n)$ is an estimate for $f_1(n)$ and $g_2(n)$ is an estimate for $f_2(n)$, then $g_1(n)\, g_2(n)$ is an estimate for $f_1(n)\, f_2(n)$. This too is true, but it says nothing about either good estimates or quotients. The following theorem lets you get good estimates for the product and the quotient of two functions.

Theorem. Suppose that $f_1(n)$ grows at the same rate as $g_1(n)$ and $f_2(n)$ grows at the same rate as $g_2(n)$. Then $f_1(n)\, f_2(n)$ grows at the same rate as $g_1(n)\, g_2(n)$ and $f_1(n)/f_2(n)$ grows at the same rate as $g_1(n)/g_2(n)$.

Proof. $(f_1(n)\, f_2(n))/(g_1(n)\, g_2(n)) = (f_1(n)/g_1(n))(f_2(n)/g_2(n))$. Since $f_1(n)/g_1(n)$ tends to a positive constant c_1 and $f_2(n)/g_2(n)$ tends to a positive constant c_2, $(f_1(n)/g_1(n))(f_2(n)/g_2(n))$ will tend to the positive constant $c_1\, c_2$, so that $f_1(n)\, f_2(n)$ grows at the same rate as $g_1(n)\, g_2(n)$.

Similarly $(f_1(n)/f_2(n))/(g_1(n)/g_2(n)) = (f_1(n)/g_1(n))/(f_2(n)/g_2(n))$, which tends to the positive constant c_1/c_2, so that $f_1(n)/f_2(n)$ grows at the same rate as $g_1(n)/g_2(n)$. QED.

Now let's use all these theorems to find a good estimate for a sadistically complicated function. Let $f(n) = \dfrac{\left(n^n + n^2 \times 2^n \times \log_2 n - n(n^{n-1} - n^4)\right)}{\left(2n^5 + 3n^4(\log_2 n)^{17}\right)}$. Seeing that there is a subtraction in the numerator, we multiply $n^{n-1} - n^4$ by n so that the numerator is equal to $n^n + n^2 \times 2^n \times \log_2 n - n^n + n^5$. The fastest growing terms n^n cancel each other out, leaving $n^2 \times 2^n \times \log_2 n + n^5$. Of these two remaining terms, the first one contains an exponential and the second one only a power; so the first one grows faster than the second one and the numerator grows at the same rate as the first term $n^2 \times 2^n \times \log_2 n$. Now, of the two terms in the denominator, which one grows faster? Well, $2n^5/3n^4(\log_2 n)^{17} = 2n/3(\log_2 n)^{17} = \dfrac{2}{3}\left(\dfrac{n^{1/17}}{\log_2 n}\right)^{17}$. Since any positive power of n grows faster than $\log_b n$ to any base $b > 1$, that fraction tends to infinity; so $2n^5$ grows faster than $3n^4(\log_2 n)^{17}$ and the denominator grows at the same rate as n^5. This means that $f(n)$ grows at the same rate as $(n^2 \times 2^n \times \log_2 n)/n^5 = 2^n \times \log_2 n / n^3$, which is a good estimate for $f(n)$. I defy you to derive and prove the same result directly from the definition of O!

Of course, there are functions $f(n)$ to which these theorems do not apply. For example, let $f(n) = n$ if n is odd and 1 if n is even. The function $g(n) = n$ is a bad estimate for $f(n)$, but you can't use any of the above theorems to prove it because $f(n)/g(n) = 1$ if n is odd and $1/n$ if n is even, which has no limit. You need to prove it directly from the definition of O. The function $g(n)$ is an estimate for $f(n)$ because $g(n) \le f(n)$ for all $n \ge 1$. And $f(n)$ is not an estimate for $g(n)$ because $g(n)/f(n) = n$ for all even n and therefore is not bounded by any constant for big n. Furthermore, even though n is a bad estimate for $f(n)$, you can't find a simple function that is a better estimate because $f(n) = n$ for all odd n; so any function that grows slower than n will not be an estimate at all! Some professors like to give you more complicated functions for which no good estimate can be found just to force you to use the definition of O instead of theorems that depend on the concept of limit, but even I am not that sadistic. After all, $f(n)$ is supposed to represent the number of operations done by an algorithm on an input of size n, and most reasonable algorithms do not have such pathological time complexities.

CHAPTER 6. ALGORITHMS – COMPLEXITY ANALYSIS AND CORRECTNESS PROOF USING A LOOP INVARIANT

An algorithm is a finite series of instructions to solve a problem. As a non-mathematical example, suppose that the problem you want to solve is to profit from a money-losing factory.

```
Procedure fraud;
   insure the factory for megabucks;
   blow up the factory;
   hire a shady lawyer;
end fraud.
```

Before programming an algorithm on a computer, you first have to refine it so that each instruction can be translated into a line of code. Here is a refinement of the instruction "blow up the factory" into three instructions:

```
   hire an illiterate worker;
   give him a job with explosives to be done late in the evening;
   leave the factory early.
```

As a more mathematical example, suppose you are allowed to eat one candy out of a pile of candies. Since they all taste equally good, you will of course choose the biggest one. You could, of course, spread them out on a table and look at them all at once to find the biggest one, but a computer can't do that; it can only compare two numbers. To model what a computer would have to do, you take the topmost candy from the pile. Then you compare the candy in your hand with the topmost one left in the pile. If the one in your hand is bigger, you throw away the topmost one in the pile; otherwise you throw away the one in your hand and take the topmost candy from the pile. Either way, the one in your hand is the bigger of the first two candies you examined. You continue this process until you have examined all the candies. At each stage, the candy in your hand is the biggest one you've examined so far, so that once you have examined all the candies, the one in your hand is the biggest one of all, and you transfer it to your mouth.

Here is the algorithm written as a *pseudocode*, which is a description of an algorithm that is not written in any particular computer language but that is sufficiently refined that it can be translated line for line into a program.

```
real function max(n: natural; A: array[1..n] of real);
                                   {max(n,A) is the largest of the n≥1 real numbers in the array A.}
   local variables i: natural; CM: real;
                        {CM stands for current maximum – it's the size of the candy in your hand.}
   CM←A[1];              {The value in A[1] gets copied into CM when you take the first candy.}
   for i←2 to n do                                  {For i = 2,3,…,n, do whatever is in the loop.}
       {At this point CM contains the largest among the values A[1],…,A[i-1] examined so far.}
      if A[i]>CM then CM←A[i] end if;
                   {Now CM contains the largest among the values A[1],…,A[i] examined so far.}
   end for;                        {Now CM contains the largest among all the values A[1],…,A[n]}
   return CM;                {Transfer the value of CM to the program calling max – your mouth.}
end max.
```

To show how this algorithm works, we do a *trace* of the program as it acts on the size-10 array (1,4,2,8,5,8,9,3,6,7), showing the value of i, $A[i]$ and CM before the loop is entered and at the end of each iteration of the loop. Note that for each i, CM is equal to the largest among the values $A[1],\ldots,A[i]$ examined so far, so that after the last iteration, CM, the value returned, is equal to the largest among all the values $A[1],\ldots,A[n]$ and you really do eat the biggest candy.

i	1	2	3	4	5	6	7	8	9	10=n	
A[i]	1	4	2	8	5	8	9	3	6	7	
CM	1	4	4	8	8	8	9	9	9	9	returned value.

The trace indicates that the algorithm works, but it doesn't constitute a **proof**. Note first that the algorithm will **not** work if $n = 0$. The array A does not have a first element; so the instruction CM$\leftarrow$A[1] will copy some garbage into the variable CM. A user-friendly algorithm would have a test for whether $n = 0$ and an instruction to exit and print an error message in that case – presumably a message more printable than what you would say if someone offered to let you eat the candy of your choice from an empty pile. The absence of error testing is a common source of bugs even in software that gets sold in stores. I once played a computer game in which you could climb either over or under a rope. I tried to walk through it instead, wondering whether the rope would stop me from walking through it or else magically disappear. Much to my chagrin, it did neither of those things; instead, the game crashed. Apparently the apparent thickness of the rope on the screen was calculated by dividing the real thickness by the distance from the player to the rope. By trying to walk through the rope I made the distance zero, and since there was no test for division by zero, the game crashed. Another well-known computer error, a disagreement over whether distance should be measured in metric or Imperial units, once made a spaceship crash on Mars!

The algorithm max does not contain a test for $n = 0$ but it does contain in the first comment the condition under which the algorithm will work: $n \geq 1$. This is called a *precondition*. The statement "max(n,A) is the largest of the n ≥ 1 real numbers in the array A" says what the algorithm will output if the precondition is satisfied; this statement is called a *postcondition*. If the postcondition is satisfied whenever the precondition is satisfied, then the algorithm is said to be *correct*.

How can we prove that the algorithm max is correct? The algorithm contains a loop, and the comment at the beginning of the loop gives a condition that should be satisfied at the beginning of each iteration of the loop - CM contains the largest among the values $A[1],\ldots,A[i\text{-}1]$. Such a statement is called a *loop invariant*. We prove that this condition is satisfied at the beginning of each iteration of the loop by mathematical induction on the number of iterations of the loop.

Basic step: the first iteration. At the beginning of the first iteration, $i = 2$ and CM contains the value $A[1]$ because the instruction CM$\leftarrow$A[1] put it there (and because the precondition $n \geq 1$ is satisfied). Since $i - 1 = 1$, the set $\{A[1],\ldots,A[i\text{-}1]\}$ consists of the sole value $A[1]$, which is trivially the largest value in the set; so the loop invariant is satisfied.

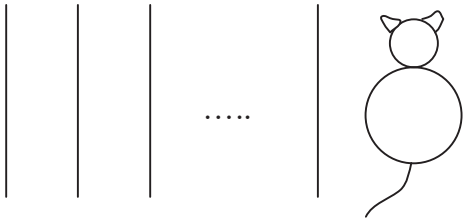
Induction step. Suppose that at the beginning of a given iteration, CM contains the largest among the values $A[1],\ldots,A[i\text{-}1]$ and that the condition for executing the body of the loop ($i \leq n$) is satisfied as well. If $A[i] >$ CM, then $A[i]$ is the largest among the values $A[1],\ldots,A[i]$, and this value $A[i]$ is copied into CM. If $A[i] \leq$ CM, then CM is already the largest among the

values $A[1],\ldots,A[i]$, and CM retains this value. In either case, at the end of this iteration of the loop, CM will contain the largest among the values $A[1],\ldots,A[i]$. Then i gets increased to $i + 1$ and the algorithm goes to the beginning of the next iteration of the loop. For this new, larger value of i, CM now contains the largest among the values $A[1],\ldots,A[i\text{ -}1]$. To see this, trace the execution of the iteration of the loop for which $i = 4$. At the beginning of this iteration, CM contains 4, the largest of the $i - 1 = 3$ values $A[1] = 1$, $A[2] = 4$ and $A[3] = 2$. At the end of this iteration, CM contains 8, the largest of the $i = 4$ values $A[1] = 1$, $A[2] = 4$, $A[3] = 2$ and $A[4] = 8$. At the beginning of the next iteration, $i = 5$ and CM contains 8, the largest of the $i - 1 = 4$ values $A[1] = 1$, $A[2] = 4$, $A[3] = 2$ and $A[4] = 8$. By induction, CM contains the largest among the values $A[1],\ldots,A[i\text{ -}1]$ **for every value of *i* including the one that violates the condition for executing the body of the loop**.

But we aren't finished yet. We have one more step to do.

Terminal step. We have to prove that the algorithm is guaranteed to terminate for every n that satisfies the precondition and that when it does terminate, the postcondition will be satisfied. The termination proof is easy: after n - 1 iterations of the loop, i will increase to $n + 1$ and the condition for executing the body of the loop will no longer be satisfied. With $i = n + 1$ the loop invariant says that CM contains the largest among the values $A[1],\ldots,A[n]$. At this point the algorithm skips to the statement following the loop and returns the value CM, so that the value returned (max(n,A)) will be the largest number in the array – the postcondition. We have proved that if the precondition is satisfied, so is the postcondition. The algorithm is correct, QED.

The process of proving the correctness of an algorithm containing a loop can be illustrated by using dominos, each one representing an iteration of the loop. In this case there is a last domino, because for the algorithm to terminate there must be a last iteration. As before, the basic step is pushing over the first domino and the induction step is each domino except the last one knocking over its neighbour to the right. The terminal step is that there is a last domino and that it falls on the cat's tail, making the poor cat say meow.



The termination proof is an essential component of a correctness proof. It can be argued that an algorithm doesn't have to terminate because, for example, a spaceship's computer never stops working. But it keeps on giving answers, whereas an algorithm that goes on forever without ever giving any answers isn't worth very much. As an example, consider the following algorithm for counting down to the launch of a spaceship: **while** $n \neq 0$ **do** $n \leftarrow n\text{-}1$; **end while**. If you enter a negative value for n, the algorithm will continue to count forever and the spaceship will never get off the ground!

How many operations does the algorithm max do as a function of n? There are n - 1 iterations of the loop. In each iteration, there is a comparison (**if** A[i] > CM) and perhaps an assignment (CM$\leftarrow$A[i]). In addition there are 2 operations on i, the index of the loop: at the beginning of the loop it is compared with n and at the end of the loop it is increased to $i + 1$. This makes $4(n - 1)$ operations so far. As well, there is the initial assignment $i\leftarrow 2$ and the final comparison of i with n when i has increased to $n + 1$, the assignment CM$\leftarrow$A[1] and the transfer of CM to the program that calls max, for a total of $4n$ operations. The total number of operations done by this algorithm is thus in $O(n)$, but we could have figured that out without calculating the exact number of operations. There are $n - 1 \in O(n)$ iterations of the loop, each iteration does $O(1)$ operations, and outside the loop an additional $O(1)$ operations are done; so the total number of operations is in $O(n)$. Doing the estimation this early is simpler than calculating the exact number of operations. The exact number is not particularly informative because we don't know whether a comparison will take the same amount of time as an assignment; so early estimation gave us all the useful information we could get. In this case the estimate we obtained was good. As we shall see later, this is not always the case.

6.1 Searching algorithms

Suppose you have an array of items of the same type and another item of the same type, called the *key*. You want to know whether the array contains any items that are identical to the key and, if so, you want to find the location of the leftmost occurrence of the key in the array. For example, suppose you have the following size-10 array of integers.

i	1	2	3	4	5	6	7	8	9	10
A[i]	2	2	2	3	3	3	5	5	5	7

If the key is 3, you want your searching algorithm to return 4. If the key is 4, you want your algorithm to return 0 to indicate that there are no occurrences of the key in the array.

The simplest searching algorithm, *sequential search*, is based on two well-known ideas. The first one comes from Lewis Carroll's book Alice in Wonderland: start at the beginning, keep going until you get to the end and then stop. The second is a riddle. Question: Why is it that you always find what you're looking for in the last place you look for it? Answer: Because once you find it, you stop looking! In mathematical terms, you start at the beginning of the array and examine successively the first item, the second item, the third item and so on until either you find the key in the array or else you get to the end of the array without finding the key. Here is a pseudocode of this algorithm.

```
natural function SequentialSearch(n: natural; A: array[1..n] of some type; key: same type)
   local variable i: natural;
   i←1;
   while i ≤ n {None of the items A[1],…,A[i-1] is equal to the key.} and A[i] ≠ key do
      i←i + 1;
   end while;
   if i ≤ n then {A[i] is the leftmost occurrence of the key} return i;
   else {there is no occurrence of the key in the array} return 0; end if;
end Sequential Search.
```

If the key is 3, then when $i = 4$, the condition for executing the body of the loop will be false; so the algorithm with jump over the loop. Since $i = 4$ and $n = 10$, $i \leq n$; so the algorithm will return 4, the index of the leftmost occurrence of 3 in the array. Similarly, if the key is 7, the condition for executing the body the loop will be violated when $i = 10$; so the algorithm will return 10. If the key is 4, then when $i = 10$, the condition for executing the body of the loop will still be satisfied and i will increase to 11. Now the first condition ($i \leq n$) will be false, and since the conjunction $p \wedge q$ is false if p is false, a smart algorithm will jump out of the loop without ever testing the second condition ($A[i] \neq$ key), which couldn't be tested anyway because $A[11]$ isn't part of the array! Since $i > n$, the algorithm will return 0.

How can we be sure that the algorithm works? Note that there is no precondition: if $n = 0$, the condition for executing the body of the loop will be false for $i = 1$; so the algorithm will return 0, which is the right answer since the key cannot be found in an empty array. The loop invariant says that for each iteration of the loop, after testing whether $i \leq n$ but before testing whether $A[i] \neq$ key, none of the items $A[1],\ldots,A[i-1]$ is equal to the key.

Basic step: the first iteration. Here $i = 1$ and the loop invariant says none of the items $A[1],\ldots,A[0]$ is equal to the key, which must be true because that part of the array is empty.

Induction step. Suppose that the condition for executing the body of the loop is satisfied and that none of the items $A[1],\ldots,A[i-1]$ is equal to the key. Since $i \leq n$, we can examine $A[i]$, and since $A[i] \neq$ key, none of the items $A[1],\ldots,A[i]$ is equal to the key. The body of the loop changes i to $i + 1$ so that for this new value of i, none of the items $A[1],\ldots,A[i-1]$ is equal to the key.

Terminal step. After at most n iterations of the loop, either $A[i] =$ key or $i = n + 1$. If $A[i] =$ key, since none of the items $A[1],\ldots,A[i-1]$ is equal to the key, the leftmost occurrence of the key in the array must be $A[i]$, and the algorithm returns its index i, as required. If $i = n + 1$, the loop invariant says that none of the items $A[1],\ldots,A[n]$ is equal to the key; so there is no occurrence of the key in the array and the algorithm returns 0 as required. The algorithm works, QED.

How many operations does it do as a function of n? Well, if you're lucky, you'll find the key in the first place you look, and then the algorithm will do $O(1)$ operations. If you're unlucky, you'll find it in the last place you could have looked – $A[n]$ – or, if you're really unlucky, you won't find it at all. In the latter case the algorithm will do all n iterations of the loop and $O(n)$ is a good estimate for the number of operations the algorithm will have to do. This is its *worst-case time complexity* – the greatest number of operations an algorithm will do as a function of n.

How many operations will the algorithm do in the average case? To answer that question, we have to make certain assumptions about the probability of each event occurring. **If** there is at least one occurrence of the key in the array and **if** the first occurrence is equally likely to be in any position in the array, **then** the average number of iterations of the loop will be $n/2$, so that $O(n)$ is still a good estimate. In general, average-case estimates are much harder to make than worst-case estimates and are usually omitted from elementary courses. Besides, an average-case estimate gives you no guarantee about how long the algorithm will take for a given instance of the problem. For example, suppose you are searching among the n keys on your key ring for the one that opens your front door. If they all look alike, you will use sequential search, trying all

your keys one after another until one of them opens your door, and since your key is sure to be on your key ring, you will find it after at most *n* tries. On the average, you will find it after about *n*/2 tries. This average-case estimate is satisfactory under ordinary conditions. But suppose that you're being chased by a young man with a knife. It isn't enough to know that in the average case you will have enough time to open your door, enter, close the door and lock it before he reaches it. You need to have a guarantee that you can escape even in the worst case – that the key that opens your door is the last one you try – otherwise you will keep on running. With computer programs the price of underestimating the worst-case time taken by a program to execute an algorithm isn't quite that dramatic, but there are circumstances under which it could be frustrating. In the prehistoric days when they still used punch cards, some computer rooms contained a special compiler for debugging programs. You stood in line, and when it was your turn, you put your stack of cards in a reader. After 2 seconds of CPU time, the computer would stop executing your program and start executing the program of the person behind you in the line. If you wanted to test your program with biggest data that the computer could execute within 2 seconds, then you would need to know the maximum time, not the average time, that the computer would take as a function of the data size.

The time taken to solve a problem depends upon the size of the data, but also on the algorithm you use to solve the problem. For example, you wouldn't look up a word in a dictionary or a name in a telephone directory using sequential search because it would take you an unreasonably long time. Fortunately you don't have to use sequential search because the words in a dictonary and the names in a telephone directory are sorted in alphabetical order and there is a faster algorithm for searching for a key in a sorted list. In a dictionary or a directory you make an assumption about the approximate location of a word or name based on the first letter. In a general sorted array, no such assumption can be made. For example, if you compared a number with the average of all the numbers in a sorted array, you would assume that 9 is near the beginning of the array (1,2,3,4,5,6,7,8,9,100) because the average number is 14.5 which is bigger than 9. With no knowledge of the array except that it is sorted, your best bet is to play it safe: you choose the strategy that minimizes the maximum number of comparisons that bad luck could force you to make. You compare the key (9) with the number in the middle of the array (5). Since 9 > 5, you know that 9 can't be to the left of 5 in the array; so, if it is in the array, it must be to the right of 5 – that is, in the sub-array (6,7,8,9,100). Again, you compare 9 with the number in the middle of the array (8), and since 9 > 8, if it is in the array it must be in the sub-array to the right of 8; that is, (9,100). Now this array has no middle, or rather, it has two near-middles: 9 and 100. If you compare the key with 9, you've lucked out and found the key in the third comparison. If you compare the key with 100, since 9 < 100 you know that 9 is to the left of 100; so if it's in the array it must be in the subarray to the left of 100, which is (9). A fourth comparison verifies that the remaining array element is indeed the key.

Suppose you are searching for the **leftmost** occurrence of a key in a sorted array. If the array is (2,2,2,3,3,3,5,5,5,7) and the key is 3, comparing the key with the middle element of the array, which is a 3 but not the leftmost 3, for equality won't help you. The following version of the algorithm suggested by the above example finds the leftmost occurrence of a key in a sorted array or determines that there are no occurrences of the key in the array. It has the added advantage that you compare the key with an element of the array only once and not twice, once for equality and once to decide if the key is bigger or smaller than the array element.

```
natural function BinarySearch(n: natural; A: array[1..n] of an ordered type; key: same type);
        {BinarySearch(n,A,key) is the index of the leftmost occurrence of the key in the array A
  of size n ≥ 1, sorted in increasing order, or 0 if there are no occurrences of the key in the array.}
   local variables: s,m,b: natural; {s,m,b are the indices of the leftmost, middle and rightmost
                                     element of the sub-array in which you are searching for the key}
   s←1; b←n;                        {In the beginning you are searching the whole array.}
   while b > s do                   {while this sub-array has more than 1 element}
      {Loop invariant: if the key is in the array, its leftmost occurrence is among A[s],…,A[b].}
     m←(s+b) div 2;                {the average of s and b rounded down if s + b is odd}
     if key > A[m] then            {A[s] ≤ … ≤ A[m] < key; so the key can't be in that part of A}
       s←m + 1;                    {Search in the subarray (A[m+1],…,A[b]).}
     else                          {key ≤ A[m] ≤ … ≤ A[b];
                                    so the leftmost occurrence of the key can't be to the right of A[m]}
       b←m;                        {Search in the subarray (A[s],…,A[m]).}
     end if;
   end while;                      {Now b = s; so if there is an occurrence of the key in A,
                                    its leftmost occurrence must be A[s].}
   if key = A[s] then
     return s;
   else
     return 0;
   end if;
end BinarySearch.
```

We trace this algorithm twice with the size-10 array that we used for sequential search:

i	1	2	3	4	5	6	7	8	9	10
A[i]	2	2	2	3	3	3	5	5	5	7

Suppose that key = 3.

s	b	m	A[m]	
1	10	5	3	Since 3 = key ≤ *A*[5] = 3, the leftmost occurrence isn't to the right of *A*[5].
1	5	3	2	Since 3 = key > *A*[3] = 2, the leftmost occurrence isn't *A*[3] or to its left.
4	5	4	3	Since 3 = key ≤ *A*[4] = 3, the leftmost occurrence isn't to the right of *A*[4].
4	4			The only possibility is *A*[4], and since key = *A*[4] = 3, return 4.

Suppose that key = 4.

s	b	m	A[m]	
1	10	5	3	Since 4 = key > *A*[5] = 3, the leftmost occurrence isn't *A*[5] or to its left.
6	10	8	5	Since 4 = key ≤ *A*[8] = 5, the leftmost occurrence isn't to the right of *A*[8].
6	8	7	5	Since 4 = key ≤ *A*[7] = 5, the leftmost occurrence isn't to the right of *A*[7].
6	7	6	3	Since 4 = key > *A*[6] = 3, the leftmost occurrence isn't *A*[6] or to its left.
7	7			The only possiblity is *A*[7], and since 4 = key ≠ *A*[7] = 5, return 0.

The precondition for this algorithm to work is that A is an array of size $n \geq 1$ sorted in increasing order. The postcondition is that the value returned is the index of the leftmost occurrence of the key in A if there are any occurrences or 0 otherwise. The loop invariant is that if there are any occurrences of the key in A, then the leftmost occurrence must be in the sub-array $A[s..b] = (A[s],\ldots,A[b])$.

Basic step. At the beginning of the first iteration of the loop, $s = 1$ and $b = n$; so the loop invariant says that if there are any occurrences of the key in A, then the leftmost occurrence must be in the entire array, which is trivially true.

Induction step. Suppose that at the beginning of some iteration of the loop, if there are any occurrences of the key in A, then the leftmost occurrence must be between $A[s]$ and $A[b]$ inclusive, and that $b > s$ so that the body of the loop will be executed. There are two cases to consider. Suppose that key $> A[m]$. Then all the elements to the left of $A[m]$ will be less than the key; so the leftmost occurrence can't $A[m]$ or to its left. If there are any occurrences of the key in A, then its leftmost occurrence must be between $A[m+1]$ and $A[b]$. When s is set to $m+1$, for the new values of s and b, the loop invariant is now true at the beginning of the next iteration. Suppose that key $\leq A[m]$. Then all the elements to the right of $A[m]$ will be at least as big as the key. If key $< A[m]$, then neither $A[m]$ nor any of the elements to its right can be equal to the key. If key $= A[m]$, then the leftmost occurrence of the key could be $A[m]$ but can't be to its right. In either case, the leftmost occurrence of the key can't be to the right of $A[m]$; so if there are any occurrences, the leftmost one must be between $A[s]$ and $A[m]$. When b is set to m, for the new values of s and b, the loop invariant is now true at the beginning of the next iteration.

Terminal step. We first prove that at every iteration, either s goes up or b goes down. Now m is at most $(s+b)/2$, which is strictly less than b if $s < b$; so if b is set to m, it goes down. And m is at least s; so if s is set to $m + 1$, it goes up. In either case, the integer $b - s$ will fall by at least 1 without ever becoming negative, so that after a finite number of iterations it falls to 0. Now $s = b$, so that the algorithm exits the loop, and the loop invariant says that if there are any occurrences of the key in A, then the leftmost one must be in the size-1 subarray that consists of $A[s]$. If key $= A[s]$, this must be the leftmost occurrence and the algorithm returns s. Otherwise, there are no occurrences of the key in A and the algorithm returns 0. The algorithm works, QED.

From the above example, we see that the number of iterations of the loop can be different for the same value of n: when they key was 3 there were 3 iterations and when the key was 4 there were 4 iterations. What is the **greatest** number of times that this algorithm will execute the loop as a function of n?

From the above example, we saw that when the sub-array to be searched was $A[1..10]$, of even size 10, we either searched the size-5 subarray $A[1..5]$ or the size-5 subarray $A[6..10]$. The sub-array to be searched was divided into two sub-arrays of exactly the same size. But when the sub-array to be searched was $A[1..5]$ or $A[6..10]$ of odd size 5, it got divided into two sub-arrays of sizes 2 and 3. When the key was 3 we got lucky and got to search the size-2 sub-array $A[4..5]$, but when the key was 4 we got unlucky and had to search the size-3 sub-array $A[6..8]$ and then our bad luck continued and we had to search the size-2 sub-array $A[6..7]$.

It appears that a size-n sub-array always gets divided into sub-arrays of sizes $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, so that in the worst case the size of the sub-array to be searched will systematically go from n to $\lceil n/2 \rceil$. The size of the sub-array $A[s..b]$ is equal to $b - s + 1$. Initially, when $s = 1$ and $b = n$, the size is $n - n + 1 = n$, not surprisingly. The algorithm quits the loop when $b = s$ so that the size of the sub-array to be searched is equal to $s - s + 1 = 1$. If $b > s$, then either b gets set to m or else s gets set to $m + 1$, so that the size changes from $b - s + 1$ to either $m - s + 1$ or $b - m$. Now m is $\lfloor (b+s)/2 \rfloor$. The table below gives the various possibilities.

Parity of b+s	even	odd
Value of m	(b+s)/2	(b+s-1)/2
Left size m-s+1	(b-s)/2 + 1 = $\lceil (b - s+1)/2 \rceil$	(b-s+1)/2
Right size b-m	(b-s)/2 = $\lfloor (b - s+1)/2 \rfloor$	(b-s+1)/2

Whether n is even or odd, a size-n array will be divided into 2 sub-arrays of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, and if our luck is systematically bad, as it must be assumed to be in a worst-case analysis, it will always be the sub-array of size $\lceil n/2 \rceil$ that will be searched next.

This is an example of a *divide-and-conquer* algorithm. You divide the current sub-array array into two sub-arrays, and then the luck of the draw chooses which one you have to search next. In a worst-case analysis you pretend that the luck of the draw is an intelligent opponent who will always leave you with the biggest possible sub-array to search in order to maximize the number of comparisons you will have to make. To minimize the number of comparisons your nasty opponent can force on you, you divide the current sub-array into two sub-arrays of sizes as nearly equal as possible, which is what the algorithm BinarySearch does.

An algorithm of this type can be illustrated by the following model. Instead of a size-n array, suppose that your mother gives you a plate of n beans that you find totally disgusting, insists that you eat them all, and leaves the room. You have a dog that will eat one bean before discovering how disgusting it is. And you have a little brother, who offers you a deal: you divide the pile of beans into two piles without cutting any bean in two, and for one dollar, he will choose one of the piles and eat all the beans in the pile. Your problem is to reduce the pile of beans to a single bean, to be given to your dog, while paying your brother as little as possible. Assuming that he is intelligent enough to leave you with the bigger pile each time, your best strategy is to divide the pile into two sub-piles of sizes as nearly equal as possible. In that case, one dollar will reduce the size of a pile from n to $\lceil n/2 \rceil$. How much money will you have to pay him to reduce a pile of n beans to a single bean?

In mathematical terms, how many times do you have to divide n by 2, rounding up each time, to reduce it to 1? If $n = 8$, it takes 3 divisions – 8,4,2,1. If $n = 16$, it takes 4 divisions – 16, 8, 4, 2, 1. In general, if $n = 2^k$, then it will take k divisions, so that if n is a power of 2, then it takes $\log_2 n$ divisions. Now suppose that n is not necessarily a power of 2. For example, suppose that $n = 9$. It will now take 4 divisions – 9, 5, 3, 2, 1. Since $\log_2 9$ is between 3 and 4 and the number of divisions is 4, which is equal to $\lceil \log_2 9 \rceil$, it is reasonable to assume that in general the number of divisions necessary for a given n is $\lceil \log_2 n \rceil$.

To prove that assumption, we use generalized induction on n.

Basic step: $n = 1$ and $n = 2$. Now $\log_2 1 = 0$ and it takes 0 divisions to reduce 1 to 1. And $\log_2 2 = 1$ and it takes 1 division to reduce 2 to 1.

Induction step. To simplify the notation, we note that $k = \lceil \log_2 n \rceil$ if and only if $2^{k-1} < n \le 2^k$. We assume that $n > 2$ and, as the induction hypothesis, that it takes $\lceil \log_2 m \rceil$ divisions to reduce m to 1 for any positive integer $m < n$. Since $\lceil n/2 \rceil$ is a positive integer less than n for any integer $n > 1$, the number d of divisions necessary to reduce $\lceil n/2 \rceil$ to 1 satisfies the inequalities $2^{d-1} < \lceil n/2 \rceil \le 2^d$. To reduce n to 1, we do 1 division to reduce it to $\lceil n/2 \rceil$ and then d more divisions to reduce it all the way down to 1, so that the total number of divisions is $d + 1$. Multiplying the previous pair of inequalities by 2, we find that $2^d < 2 \times \lceil n/2 \rceil \le 2^{d+1}$. If n is even, then $\lceil n/2 \rceil = n/2$ so that $2^d < n \le 2^{d+1}$. If n is odd, then $\lceil n/2 \rceil = (n+1)/2$, so that $2^d < n+1 \le 2^{d+1}$. Then certainly $n \le 2^{d+1}$. To show that $2^d < n$, we recall that since n is odd, $n + 1$ is even; so, since 2^d is even if $d > 0$, $n + 1$ must exceed 2^d by at least 2, so that $2^d < n$. If $d = 0$, then $\lceil n/2 \rceil = 1$ so that $n \le 2$, but we eliminated this case by assuming that $n > 2$. In any case, $2^d < n \le 2^{d+1}$, so that it takes $d + 1 = \lceil \log_2 n \rceil$ divisions to reduce n to 1, QED.

The complexity of binary search is in $O(\log_2 n)$ which, as we know, grows more slowly than n, the complexity of sequential search. For example, it could take as many as 1000000 comparisons to search a size-1000000 dictionary using sequential search but no more than 20 comparisons using binary search.

6.2 Sorting algorithms

Since a sorted array can be searched much faster than an unsorted one, if an array has to be searched often it's a good idea to sort it first. This section discusses several sorting algorithms.

The simplest sorting algorithm is called *Selection Sort*. You search for the smallest element in the array and put it in position 1 by exchanging it with the first element. Then you search for the second-smallest element, which is the smallest element in the part of the array to the right of the first element, and put it in position 2 by exchanging it with the second element. You continue this process until all the elements except the largest one are in their right places; then the largest element too will be in its right place – the rightmost position – and the array will be sorted.

A purist may object that terms like the smallest element, the second-smallest element, etc. have a precise meaning only if all the elements are distinct. If not, we can still state arbitrarily that x is "the" ith smallest element element of an array if i - 1 of the elements are less than or equal to x and all the other elements are greater than or equal to x.

This algorithm has two loops, one inside the other. The inner loop finds the position of the smallest element in the sub-array $A[i..n]$, which is the ith smallest element in the array, and the outer loop puts it in position i for $i = 1, 2, \ldots, n$ - 1, sorting the array into increasing order.

Here is the algorithm.

```
procedure SelectionSort(n: natural; A: array[1..n] of an ordered type);
   local variables i,j,pos: natural);
   for i←1 to n-1 do              {The kth smallest element is in position k for all k, 1 ≤ k ≤ i -1.}
      pos←i;
      for j←i+1 to n do
         if A[j] < A[pos] then pos←j; end if;
      end for;    {pos is now the position of the smallest element in A[i..n], the ith smallest in A}
      exchange A[i] with A[pos];              {Now the ith smallest element is in position i too.}
   end for;
      {Now the kth smallest element is in position k for all k, 1 ≤ k ≤ n - 1, and thus for k = n too.}
end SelectionSort.
```

The inner loop is like the procedure max presented in the beginning of this chapter, with three differences. It searches the sub-array $A[i..n]$ instead of the whole array. It finds the smallest rather than the largest element of an array. And it finds the position rather than the value of that element, which is necessary if we want to exchange it with the element in position i. I will assume that you are capable of adapting the trace and the proof of max to trace and prove the inner loop; so I will do so for the outer loop only.

First the trace, with the original array printed and then with i, pos and the current array printed at the end of each iteration of the outer loop.

```
      k     1     2     3     4     5     6     7     8
   A[k]     5     3     2     7     2     3     5     2
i=1, pos=3  2     3     5     7     2     3     5     2
i=2, pos=5  2     2     5     7     3     3     5     2
i=3, pos=8  2     2     2     7     3     3     5     5
i=4, pos=5  2     2     2     3     7     3     5     5
i=5, pos=6  2     2     2     3     3     7     5     5
i=6, pos=7  2     2     2     3     3     5     7     5
i=7, pos=8  2     2     2     3     3     5     5     7
```

Now the proof, assuming that the inner loop puts the index of the smallest element of the sub-array $A[i..n]$ into the variable pos. We show that for each i, the smallest i - 1 elements are all in their right places at the beginning of the ith iteration of the outer loop.

Basic step: $i = 1$. Here there are 0 elements to consider; so the statement is trivially true.

Induction step. Suppose that $1 \le i \le n - 1$ and that at the beginning of iteration i, the smallest i - 1 elements are all in their right places. The inner loop copies into the variable pos the index of the smallest element of the sub-array $A[i..n]$. That means that all the other elements of this sub-array are greater than or equal to A[pos]. The first i - 1 elements of the array are all less than or equal to A[pos] because they are the smallest i - 1 elements and A[pos] isn't among them. This makes A[pos] the ith smallest element of the array according to our loose definition of the term. The exchange puts this element in the ith position, so that now the smallest i elements are all in their right places. At the end of the loop, i increases to $i + 1$, so that for this new value of i, the smallest i - 1 elements are all in their right places.

Terminal step. The outer loop is iterated at most n times and, for each of these, the inner loop is iterated at most n times, or at most n^2 times altogether. For any n this is a finite number; so after a finite number of operations i will increase to n and the algorithm terminates. The loop invariant now states that the smallest n - 1 elements are in their right places, so that the nth smallest element (the biggest one) must be in the only place left (position n), which is its right place. The array is now sorted in increasing order and the algorithm works, QED.

Note that the terminal step contains an early estimate of the number of iterations of the inner loop: at most n^2, so that the time complexity of the algorithm is in $O(n^2)$. Is this early estimate a good one? When $i = 1$, the inner loop is iterated n-1 times; when $i = 2$ it is iterated n --2 times and so on, so that the total number of iterations is $(n\text{-}1) + (n\text{-}2) + \ldots + 2 + 1 = n(n\text{-}1)/2$, and we know that n^2 is a good estimate for that expression. That identity was chosen with more in mind than just an excuse to tell the joke about hand-shaking!

Another simple sorting algorithm is called *Insertion Sort*. In this algorithm, we assume that the first i - 1 elements are in increasing order (but not necessarily the i - 1 smallest elements) and we insert the ith element into its right place among them. Here is the algorithm.

```
procedure InsertionSort(n: natural; A: array[1..n] of an ordered type);
  local variables i,j: natural; x: the same type as an array element;
  for i←2 to n do                          {The first i - 1 elements are in increasing order.}
    x←A[i]; j←i - 1;                              {Store A[i] for insertion into A[1..i-1].}
    while j > 0 and A[j] > x do          {A[i] should be somewhere to the left of A[j].}
      A[j+1]←A[j];
      j←j - 1;
     end while;                          {Now everything in A[1..i-1] that's bigger than A[i]
                                                        has been pushed one space to the right.}
    A[j+1]←x ;                                {Insert x, the old A[i], into the hole between
                                        the elements bigger than x and the ones less than or equal to x,
                                                   so that the first i elements are in increasing order.}
  end for;                                    {Now the whole array is in increasing order.}
end InsertionSort.
```

This algorithm too makes $n(n\text{-}1)/2$ comparisons in the worst case – when the array is originally sorted in decreasing order and all the elements are distinct. Its advantage over Selection Sort: if the array is originally nearly sorted in increasing order, and this happens fairly often in real applications, Insertion Sort makes far fewer than $n(n\text{-}1)/2$ comparisons. Its disadvantage: in the worst case, it takes $n(n\text{-}1)/2$ operations to push to the right the elements in $A[1..i\text{ -}1]$ that are bigger than $A[i]$ for all i, whereas Selection Sort makes only n-1 exchanges, each of which moves 3 array elements.

A variation of Insertion Sort, called *Binary Insertion Sort*, uses binary search to find the position in $A[1..i\text{-}1]$ into which to insert $A[i]$. There are i possible places for $A[i]$ to go; so the binary search does at most $\lceil \log_2 i \rceil$ comparisons to find the right slot. Placing all n elements of the array takes at most $\lceil \log_2 2 \rceil + \lceil \log_2 3 \rceil + \ldots + \lceil \log_2 n \rceil \le n \lceil \log_2 n \rceil$ comparisons. The number of comparisons done by this algorithm is in $O(n \log_2 n)$ but not the total number of operations,

because it does $n(n\text{-}1)/2$ pushes in the worst case. I leave you to write a pseudocode for Binary Insertion Sort and to trace and prove the correctness of both types of Insertion Sort. In what follows I will describe some algorithms that do $O(n \log_2 n)$ operations in total.

The simplest of these algorithms is Merge Sort. It is based on the fact that two sorted lists of total length n can be merged into a single sorted list in $O(n)$ time. To illustrate the merging process, imagine a little red schoolhouse that is so out of date that the classes are segregated into girls' classes and boys' classes and so small that there is only one graduating class of each gender. Each year the school holds a graduation ceremony in which all the graduating students walk onto the stage in increasing order of height to receive their diplomas. The principal tells the home room teacher of each graduating class to sort his or her students in increasing order of height and then merges the two lines of students into a single line. He measures the shortest girl against the shortest boy and sends the shorter of those two students to the first place in the coed line, with a tie decided in the girl's favour out of chivalry. Then he measures the shortest remaining boy against the shortest remaining girl and so on until either all the girls or all the boys are in the coed line. Finally he sends the remaining students, either all girls or all boys, into the coed line with no comparisons. In the following example, the numbers represent the height of the students in centimeters minus 160 and the girls' numbers are underlined to distinguish them from the boys of the same height.

```
Girls: 2  3  3  5  6  8  9 | Boys: 3  4  4  5  9 10 12 | Coed:
Girls: 3  3  5  6  8  9 | Boys: 3  4  4  5  9 10 12 | Coed: 2
Girls: 3  5  6  8  9 | Boys: 3  4  4  5  9 10 12 | Coed: 2  3
Girls: 5  6  8  9 | Boys: 3  4  4  5  9 10 12 | Coed: 2  3  3
Girls: 5  6  8  9 | Boys: 4  4  5  9 10 12 | Coed: 2  3  3  3
Girls: 5  6  8  9 | Boys: 4  5  9 10 12 | Coed: 2  3  3  3  4
Girls: 5  6  8  9 | Boys: 5  9 10 12 | Coed: 2  3  3  3  4  4
Girls: 6  8  9 | Boys: 5  9 10 12 | Coed: 2  3  3  3  4  4  5
Girls: 6  8  9 | Boys: 9 10 12 | Coed: 2  3  3  3  4  4  5  5
Girls: 8  9 | Boys: 9 10 12 | Coed: 2  3  3  3  4  4  5  5  6
Girls: 9 | Boys: 9 10 12 | Coed: 2  3  3  3  4  4  5  5  6  8
Girls:| Boys: 9 10 12 | Coed: 2  3  3  3  4  4  5  5  6  8  9
Girls:| Boys:10 12 | Coed: 2  3  3  3  4  4  5  5  6  8  9  9
Girls:| Boys:12 | Coed: 2  3  3  3  4  4  5  5  6  8  9  9 10
Girls:| Boys:| Coed: 2  3  3  3  4  4  5  5  6  8  9  9 10 12
```

This process does at most n - 1 comparisons because at least the tallest student, whether a girl or a boy, will be sent into the coed line with no comparison. It moves all n students and, for each one, changes at most 3 indices, one for each of the lines, n times; so the total complexity is in $O(n)$. Having done the trace and the analysis, I leave it to you to write the pseudocode and prove its correctness.

Now suppose you have a size-n array to sort. Initially it can be considered as n consecutive arrays, each of size 1, and a size-1 array is by definition sorted. As a first step, you merge the first (i.e. leftmost) array with the second, the third with the fourth and so on, into sorted size-2 arrays, leaving a size-1 array at the extreme right if n is odd. Then you merge these arrays into bigger ones and so on until you get a single sorted array of size n (see the illustration below in which $n = 9$).

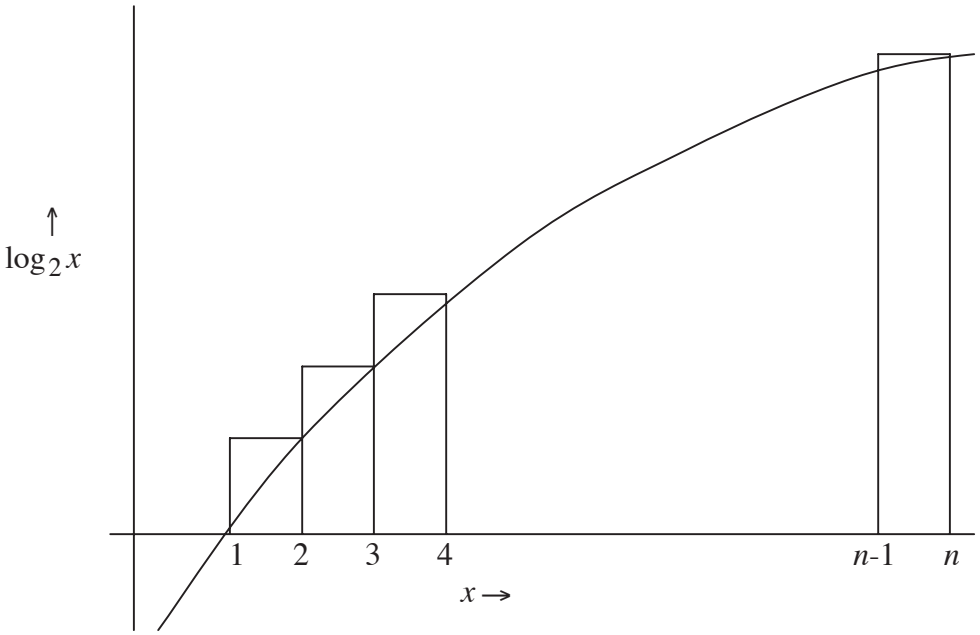
```
|5|2|1|9|7|3|6|4|8|
|2 5|1 9|3 7|4 6|8|
|1 2 5 9|3 4 6 7|8|
|1 2 3 4 5 6 7 9|8|
|1 2 3 4 5 6 7 8 9|
```

In each step of this algorithm, the total number of operations needed to make each sub-array is proportional to its length; so the total number of operations done by this step is proportional to the sum of the lengths of all the sub-arrays, which is n. How many steps are there? Initially there are n sub-arrays. Suppose that after a certain number of steps there are $i > 1$ sub-arrays. If i is even, then each sub-array gets merged with another one, leaving $i/2$ sub-arrays. If i is odd, then the rightmost array doesn't get merged but each of the other i - 1 sub-arrays do, leaving $(i-1)/2 + 1 = (i+1)/2$ sub-arrays. In either case, the number of sub-arrays is now $\lceil i/2 \rceil$. The number of steps is the number of times you have to divide n by 2, rounding up each time if necessary, to reduce it to 1, and we have already calculated this number: $\lceil \log_2 n \rceil$. This means that the total number of operations needed for this algorithm to sort an array of size n is in $O(n \log_2 n)$, as advertised.

To bound the number of comparisons from above, we make an early estimate: it takes at most n - 1 comparisons to merge two sorted arrays of total length n into a single sorted array, but we simplify n - 1 to n. Now the total number of comparisons at each step is at most n; so the total number of comparisons done by the whole algorithm is at most $n\lceil \log_2 n \rceil$.

Is this a good estimate? We show that not only can't Merge Sort do any better, but neither can any other sorting algorithm that is restricted to comparing array elements with each other. We assume that all the elements are distinct so that there are $n!$ possible orders into which they can be sorted. Each comparison divides the pile of possible orders into two sub-piles and leaves you with one of them, depending upon the result of the comparison. For example, if there are 3 elements x, y and z, then initially all 6 orders xyz, xzy, yxz, yzx, zxy, zyx are possible. Now compare x with y. If $x > y$, then only the 3 orders yxz, yzx, zyx are possible, whereas if $y > x$ only the other 3 orders are possible. To minimize the maximum number of comparisons that a nasty opponent can force on you, you will choose a comparison that divides the pile of remaining orders into two piles of sizes as nearly equal as possible, and you can't do any better than dividing a pile of size i into a pile of size $\lfloor i/2 \rfloor$ and a pile of size $\lceil i/2 \rceil$, allowing your opponent to leave you with the pile of size $\lceil i/2 \rceil$. Whatever sorting algorithm you use, it will terminate when the number of possible orders has been reduced to 1. It follows that no sorting algorithm can guarantee to sort a size-n array using fewer comparisons than the number of times you have to divide $n!$ by 2, rounding up each time, to reduce it to 1, and this number is $\lceil \log_2(n!) \rceil$.

Now $\log_2(n!) = \log_2 1 + \log_2 2 + \ldots + \log_2 n$. As you can see from the illustration below, this number is greater than the area under the curve $y = \log_2 x$ from $x = 1$ to $x = n$, which is $\int_1^n \log_2 x\,dx = (1/\ln 2)\int_1^n \ln x\,dx =$ (using elementary calculus) $n \log_2 n - (n-1)/\ln 2$. On the other hand, $n\lceil \log_2 n \rceil < n \log_2 n + n$. The difference between the upper bound and the lower bound is in $O(n)$; so not only can't you sort a size-n array in a number of comparisons that grows slower than the number required by Merge Sort, you can't even make the coefficient of the fastest-growing term any lower than 1. All you can hope to do is shave off $O(n)$ comparisons.

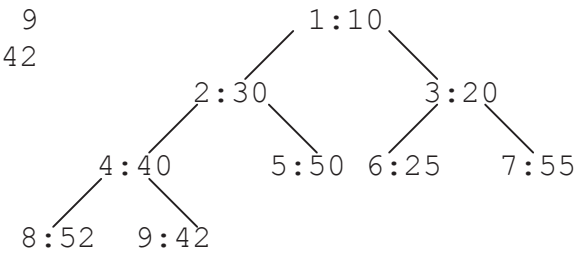


When you program Merge Sort (and I hope you will do so), you will see that you need an auxiliary array of size n to do the merging. You could merge each pair of sub-arrays into the corresponding positions in the auxiliary array and then copy the auxiliary array back into the original one, or you could save most of this copying by alternately merging from the original array to the auxiliary one and back and then, if the sorted array ends up in the auxiliary one, you copy it back into the original one. But one way or another, you still need an auxiliary array.

There is an algorithm that sorts a size-n array in $O(n \log_2 n)$ operations without using an auxiliary array. It is called Heap Sort because it uses a data structure called a *binary heap*. A binary heap is an array A with the property that for any $i > 1$, $A[i] \geq A[\lfloor i/2 \rfloor]$. It is easy to see that the smallest element in a binary heap is $A[1]$. A binary heap becomes easier to visualize if it is represented by a binary tree of the type illustrated below. The binary heap operations we will need are inserting an element into the heap and removing the smallest element all the while preserving the heap property.

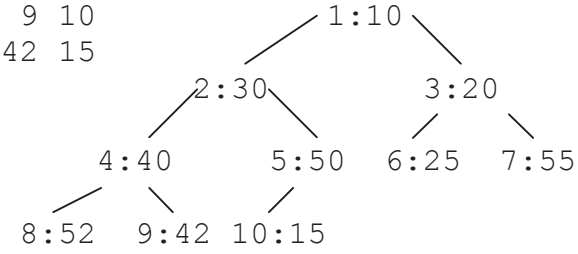
Imagine a prison in which the array index i represents the number of the cell into which the guard has put a prisoner and $A[i]$ represents his fighting strength: the better he fights, the lower the number. The best fighter, the barn boss, is housed in cell 1, closest to the exit. Whenever an order comes to release a prisoner, the guard releases the barn boss; so all the prisoners aspire to that status. Accordingly, they arrange themselves into a hierarchy in which the prisoner in cell $i > 1$ is a slave of the one in cell $\lfloor i/2 \rfloor$, his boss, and by definition a slave can't beat up his boss. In the example below, the hierarchy is illustrated by a binary tree, where each prisoner (identified by his cell number and fighting strength) is linked to his boss, if he has one, and his two slaves, if he has two, by a line.

```
Cell no.  1  2  3  4  5  6  7  8  9
Prisoner 10 30 20 40 50 25 55 52 42
```



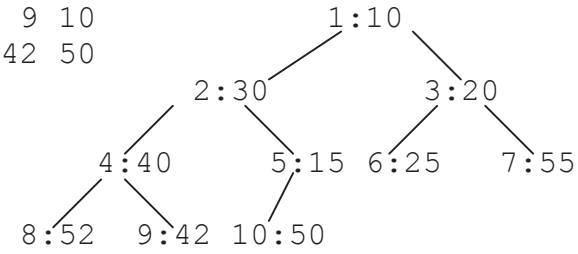
Now suppose a new man, with fighting strength 15, is thrown into jail. The guard, being lazy, opens up cell #10 and puts him in. This makes him a slave of prisoner #5 with fighting strength 50 (see the diagram below).

```
Cell no.  1  2  3  4  5  6  7  8  9 10
Prisoner 10 30 20 40 50 25 55 52 42 15
```



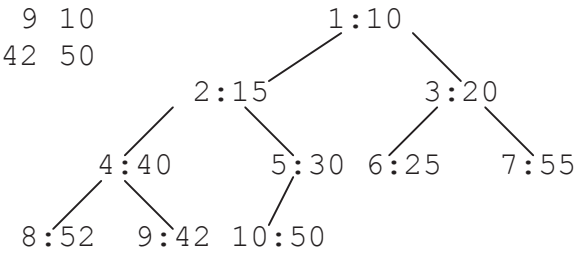
The new prisoner challenges his immediate boss to a fight, beats him up and trades places with him in the hierarchy, and the guard responds by putting each one in the other's cell. Now the new prisoner is a slave of prisoner #2 with fighting strength 30 (see the diagram below).

```
Cell no.  1  2  3  4  5  6  7  8  9 10
Prisoner 10 30 20 40 15 25 55 52 42 50
```



Now he challenges his new immediate boss to a fight, beats him up, trades places with him in the hierarchy and swaps cells with him. He is now a slave of prisoner #1 with fighting strength 10 (see the diagram below).

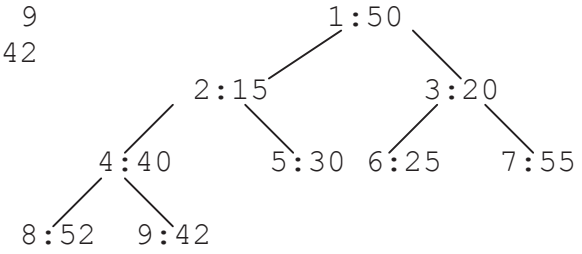
```
Cell no.  1  2  3  4  5  6  7  8  9 10
Prisoner 10 15 20 40 30 25 55 52 42 50
```



He challenges his new immediate boss to a fight and gets beat up. He has now found his appropriate slot in the hierachy. This process is called *promotion*. The promotion that takes the greatest number of comparisons is the one that promotes prisoner #*n* to the status of barn boss. His cell number gets divided by 2, rounding down each time, until it reaches 1. The number of divisions necessary to do this is $\lfloor \log_2 n \rfloor$ (you can prove this assertion using the same method used in the case of rounding up); so this is the greatest number of comparisons a promotion can make.

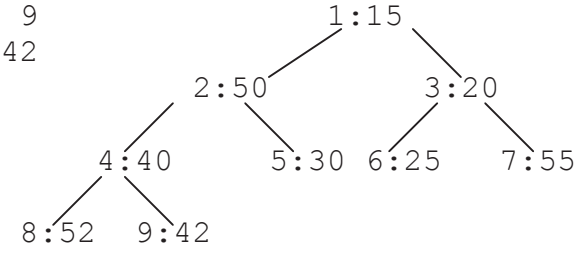
Now an order comes to release a prisoner. According to an agreement reached between the guard and the prisoners, the guard releases the barn boss. Being lazy, he wants to minimize the distance he has to walk to survey all the prisoners; so he closes the last cell and puts the prisoner that used to occupy that cell into cell #1 vacated by the barn boss (see the diagram below).

```
Cell no.  1  2  3  4  5  6  7  8  9
Prisoner 50 15 20 40 30 25 55 52 42
```



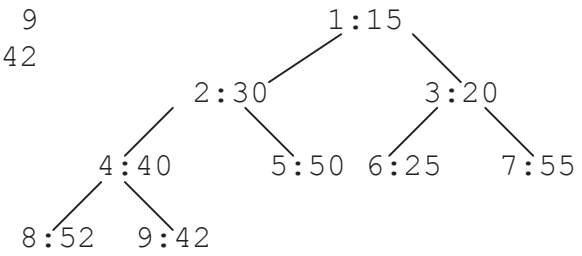
This makes him the new barn boss, but he doesn't deserve that status. His two slaves have a fight for the right to challenge their undeserving boss. Prisoner #2 wins that fight, challenges his boss, beats him up, trades places with him in the hierarchy and swaps cells with him (see the diagram below).

```
Cell no.  1  2  3  4  5  6  7  8  9
Prisoner 15 50 20 40 30 25 55 52 42
```



The poor guy has been demoted from position 1 to position 2, but he isn't even worthy of that position. His new slaves fight for the right to challenge him. Prisoner #5 wins that fight, challenges his boss, beats him up, trades places with him in the hierarchy and swaps cells with him (see the diagram below).

```
Cell no.  1  2  3  4  5  6  7  8  9
Prisoner 15 30 20 40 50 25 55 52 42
```



The poor guy has no more slaves; so he has reached is appropriate place in the hierarchy. This process is called a *demotion*. The demotion that takes the greatest number of comparisons is the one that demotes the imposter barn boss into position #*n*. The number of steps is $\lfloor \log_2 n \rfloor$, but each step requires two comparisons, one between the two slaves for the right to challenge the boss and one for the challenge itself.

Once you have a heap, it is easy to convert it into an array sorted in decreasing order. The smallest element is in position 1. You remove it as shown above and then put it in position *n*. Now the sub-array *A*[1..*n*-1] is a heap and its smallest element, the second-smallest one in the whole array, is in position 1. You remove it and put it in position *n* - 1, and so on until the biggest element is in position 1. We trace this algorithm on the heap shown in the nearest diagram above. In the table below, a vertical line separates what's left of the heap with the sorted part of the array.

```
Index    1  2  3  4  5  6  7  8  9
Element 15 30 20 40 50 25 55 52 42
        20 30 25 40 50 42 55 52|15
        25 30 42 40 50 52 55|20 15
        30 40 42 55 50 52|25 20 15
        40 50 42 55 52|30 25 20 15
        42 50 52 55|40 30 25 20 15
        50 55 52|42 40 30 25 20 15
        52 55|50 42 40 30 25 20 15
        55|52 50 42 40 30 25 20 15
```

This process takes $\lfloor \log_2(n-1) \rfloor + \lfloor \log_2(n-2) \rfloor + \ldots + \lfloor \log_2(1) \rfloor \le n \log_2 n$ demotion steps, each of which takes 2 comparisons; so the number of comparisons necessary to convert a heap into a sorted array is less than (and near to) $2n \log_2 n$. But how many comparisons are necessary to convert an arbitrary array into a heap? It would be tempting to insert the elements one at a time into a heap, but this would take about $n \log_2 n$ comparisons and there is a more efficient way. An element *A*[*i*] is said to *violate the heap condition* if it is bigger than either *A*[2*i*] or *A*[2*i*+1], assuming these elements are actually in the array (i.e. if the prisoner in cell *i* has at least one slave who can beat him up). Now, the last element that actually has any slaves is $A[\lfloor n/2 \rfloor]$. We demote that element, so that the new $A[\lfloor n/2 \rfloor]$ doesn't violate the heap condition. We then demote $A[\lfloor n/2 \rfloor - 1]$, and so on until after we have demoted *A*[1] we have a heap. We illustrate that process using a trace.

```
Index       1  2  3  4  5  6  7  8  9
Element    40 20 55 15 50 42 52 25 30
Demote 15: 40 20 55 15 50 42 52 25 30
Demote 55: 40 20 42 15 50 55 52 25 30
Demote 20: 40 15 42 20 50 55 52 25 30
Demote 40: 15 20 42 25 50 55 52 40 30
```

This algorithm makes only $O(n)$ comparisons. Intuitively speaking, most of the prisoners undergo only a few demotions steps because they are close to the bottom of the binary tree. The prisoners on the lowest level undergo no demotion steps because they have no slaves. The prisoners one level up undergo at most one demotion step and there are at most n of them (actually, at most $n/2$ but I'm too lazy to prove that assertion and multiplicative constants are irrelevant in O analyses). There are half as many prisoners one level up – at most $n/2$ of them – and they undergo at most two demotion steps. There are at most $n/4$ on the next level, and they undergo at most three demotion steps, and so on. The total number of demotion steps is at most $n(1/1 + 2/2 + 3/4 + 4/8 + 5/16 + \ldots)$. Of course there are only a finite number of levels, but the sum in parentheses can only go up if you let it continue to infinity. Now there is an algebraic trick for finding the exact value of sum of this infinite series (see Section 10.1), but all we need to know is that the series converges. Since the ratio of each term to the one before it tends to $1/2 < 1$, the ratio test says that the series converges; so the total number of demotion steps is bounded by a constant multiplied by n. Each demotion step involves 2 comparisons; so the number of comparisons too is in $O(n)$ and so is the total complexity of turning a size-n array into a heap.

This means that the total number of comparisons needed by Heap Sort to sort a size-n array is $2\,n \log_2 n + O(n)$ – twice as many as Merge Sort. So now we have two algorithms that sort a size-n array in worst-case time $O(n \log_2 n)$. There is Heap Sort, which does twice as many comparisons as Merge Sort but doesn't use an auxiliary array and has the added advantage that if you only want the i smallest elements you can stop after finding them. And there is Merge Sort, which is twice as fast as Heap Sort but uses an auxiliary array and makes you sort the entire array no matter how few of the smallest elements you want to find. So which of these two algorithms is used in the real world to sort arrays? The winner is (drum roll, please) … **neither of them!** Instead, it is another sorting algorithm called Quick Sort, which has a worst-case time complexity of $O(n^2)$. And a complicated analysis, which I will spare you, shows that even in the average case it makes about $1.4\,n \log_2 n$ comparisons, whereas Merge Sort makes only $n \log_2 n$ of them even in the worst case. But experiments have shown that with enough optimizations, Quick Sort can be made to run faster than Merge Sort on randomly generated arrays. These optimizations are outside the scope of a mathematical monograph like this one; so I will not discuss Quick Sort any further. Besides, I haven't thought of a funny way to illustrate it.

Do you recall that in the beginning of Chapter 5 I supposed the existence of two sorting algorithms: Algorithm S that sorts a size-n array in a number of operations for which n^2 is a good estimate and Algorithm M that sorts a size-n array in a number of operations for which $n \log n$ is a good estimate? Now we can identify these two algorithms: S stands for Selection Sort and M stands for Merge Sort.

6.3 Number theory algorithms

As a first example, we present an algorithm for computing the quotient and the remainder of the division of a non-negative integer a by positive integer d. Suppose you have $a \geq 0$ candies and you want to distribute them to $d > 0$ children so that each child gets the same number q of candies (otherwise some of the children will cry) and there are $r < d$ candies left over for you (otherwise all the children will call you a pig). As long as there are at least as many candies as children, you give one candy to each child, keeping a record of the number q of candies you have given to each child and the number r of candies you have left. When there are fewer candies left than children, q will be the quotient and r will be the remainder. Here is the algorithm.

```
procedure divide(a,d,q,r: natural)
                          {If d > 0, this algorithm finds q and r such that a = dq + r and 0 ≤ r < d.}
   r←a; q←0;
   while r ≥ d do                                  {loop invariant: a = dq + r and r ≥ 0}
      q←q + 1;
      r←r - d;
   end while;
end divide.
```

Trace for $a = 100$ and $d = 7$:

q	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
r	100	93	86	79	72	65	58	51	44	37	30	23	16	9	2

Since $100 = 7 \times 14 + 2$ and $0 \leq 2 < 7$, the algorithm works for this example.

Correctness proof.

Precondition: $d > 0$.
Postcondition: The algorithm terminates with $a = dq + r$ and $0 \leq r < d$.

Basic step. At the beginning of the first iteration of the loop, $r = a$ and $q = 0$; so $qd + r = a$ and $r \geq 0$ because a is a natural number, which is non-negative by definition.

Induction step. Suppose that at the beginning of a given iteration $a = dq + r$ and $r \geq 0$ and that $r \geq d$ so that the body of the loop will be executed. The first instruction in the loop adds 1 to q, which adds d to $dq + r$. Then the second instruction subtracts d from r, which subtracts d from $dq + r$, so that $dq + r$ has the same value at the beginning of the next iteration as it did at the beginning of this one, which is a. Also, since $r \geq d$ at the beginning of this iteration and d is subtracted from r, $r \geq 0$ at the beginning of the next iteration; so the loop invariant will be true at the beginning of the next iteration.

Terminal step. Since $d > 0$, each iteration of the loop subtracts at least 1 from r, which was initially a, so that after at most a steps, r will be less than d and the algorithm will terminate. The loop invariant says that $a = qd + r$ and that $r \geq 0$, and since now $d < r$, $0 \leq r < d$. The algorithm works, QED.

Note that during the terminal step we gave the time-complexity analysis of the algorithm: $O(a)$. The algorithm you learned in school treats one digit of a at each iteration, and since the

number of decimal digits of $a > 0$ is $\lfloor \log_{10} a \rfloor + 1$ (prove it!), the complexity of this algorithm is in $O(\log a)$. As an exercise, write a pseudocode for that algorithm, where the numbers are represented not in decimal but in binary.

The next example is an algorithm for identifying all the prime numbers up to n. It is called the sieve of Eratosthenes after the ancient Greek mathematician who discovered it. You write down all the positive integers less than or equal to n except 1. You cross out all the multiples of 2 except 2 itself. Then you cross out all the multiples of 3 except 3 itself. Since 4 is already crossed out, you ignore it. Since 5 is not crossed out, you cross out all its multiples except 5 itself, and so on until you get to $n/2$ (or a smaller number, as we shall see). The numbers that are not crossed out are the primes less than or equal to n. In the following trace, where $n = 114$, a number is underlined instead of being crossed out. Here is the initial list of numbers.

	7	13	19	25	31	37	43	49	55	61	67	73	79	85	91	97	103	109
2	8	14	20	26	32	38	44	50	56	62	68	74	80	86	92	98	104	110
3	9	15	21	27	33	39	45	51	57	63	69	75	81	87	93	99	105	111
4	10	16	22	28	34	40	46	52	58	64	70	76	82	88	94	100	106	112
5	11	17	23	29	35	41	47	53	59	65	71	77	83	89	95	101	107	113
6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	108	114

The multiples of 2 are all on the second, fourth and sixth line, making them easy to cross out. After you cross them out, except for 2 itself, you get this list.

	7	13	19	25	31	37	43	49	55	61	67	73	79	85	91	97	103	109
2	8	14	20	26	32	38	44	50	56	62	68	74	80	86	92	98	104	110
3	9	15	21	27	33	39	45	51	57	63	69	75	81	87	93	99	105	111
4	10	16	22	28	34	40	46	52	58	64	70	76	82	88	94	100	106	112
5	11	17	23	29	35	41	47	53	59	65	71	77	83	89	95	101	107	113
6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	108	114

The multiples of 3 are all on the third and sixth line, and the ones on the sixth line have already been crossed out.

	7	13	19	25	31	37	43	49	55	61	67	73	79	85	91	97	103	109
2	8	14	20	26	32	38	44	50	56	62	68	74	80	86	92	98	104	110
3	9	15	21	27	33	39	45	51	57	63	69	75	81	87	93	99	105	111
4	10	16	22	28	34	40	46	52	58	64	70	76	82	88	94	100	106	112
5	11	17	23	29	35	41	47	53	59	65	71	77	83	89	95	101	107	113
6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	108	114

Note that the smallest multiple of 3 that you crossed out for the first time is 9. The multiples of 4 have already been crossed out as multiples of 2. The multiples of 5 are all on ascending diagonals, making them easy to cross out too.

	7	13	19	25	31	37	43	49	55	61	67	73	79	85	91	97	103	109
2	8	14	20	26	32	38	44	50	56	62	68	74	80	86	92	98	104	110
3	9	15	21	27	33	39	45	51	57	63	69	75	81	87	93	99	105	111
4	10	16	22	28	34	40	46	52	58	64	70	76	82	88	94	100	106	112
5	11	17	23	29	35	41	47	53	59	65	71	77	83	89	95	101	107	113
6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	108	114

Note that the smallest multiple of 5 that you crossed out for the first time is 25. The multiples of 6 have already been crossed out as multiples of 2 or 3. The multiples of 7 are all on descending diagonals, making them easy to cross out too (now you know why I wrote 6 lines of numbers).

	7	13	19	25	31	37	43	49	55	61	67	73	79	85	91	97	103	109
2	8	14	20	26	32	38	44	50	56	62	68	74	80	86	92	98	104	110
3	9	15	21	27	33	39	45	51	57	63	69	75	81	87	93	99	105	111
4	10	16	22	28	34	40	46	52	58	64	70	76	82	88	94	100	106	112
5	11	17	23	29	35	41	47	53	59	65	71	77	83	89	95	101	107	113
6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	108	114

Note that the smallest multiple of 7 that you crossed out for the first time is 49. It seems as if the smallest multiple of d that gets crossed out for the first time will be d^2. The multiples of 8, 9 and 10 have already been crossed out as multiples of 2 or 3 or 5. And since $11^2 = 121 > 114$, all its multiples ≤ 114 have already been crossed out; so you can stop now.

On a computer you can't cross out numbers. Instead, you make an array P of size n, and itialize $P[i]$ to 1 for every i from 2 to n. To cross out i, you set $P[i]$ to 0. At the end, the primes $\leq n$ are the numbers i, $2 \leq i \leq n$, such that $P[i] = 1$. Here is the algorithm.

```
procedure primes(n: natural; P: array[2..n] of natural)
   local variables i,d: natural;
   for i←2 to n do P[i]←1 end for;
   d←2;
   while d*d ≤ n do                          {If i is a composite number < d×d, then P[i]=0}
      if P[d] = 1 then                                  {d is a prime, otherwise d is composite}
         for i←2*d to n in steps of d do P[i]←0; end for;
      end if;
      d←d+1;
   end while;                                {For all i, P[i]=1 if and only if i is a prime.}
end primes.
```

The trace having already been done, here is the correctness proof.

It is clear that if i is a prime, $P[i] = 1$, because the inner loop sets $P[i]$ to 0 only if i is a multiple of some integer $d > 1$ and $i \neq d$; so all we need to prove is that if i is a composite number $\leq n$, then $P[i]$ will get set to 0 by the time the algorithm has terminated.

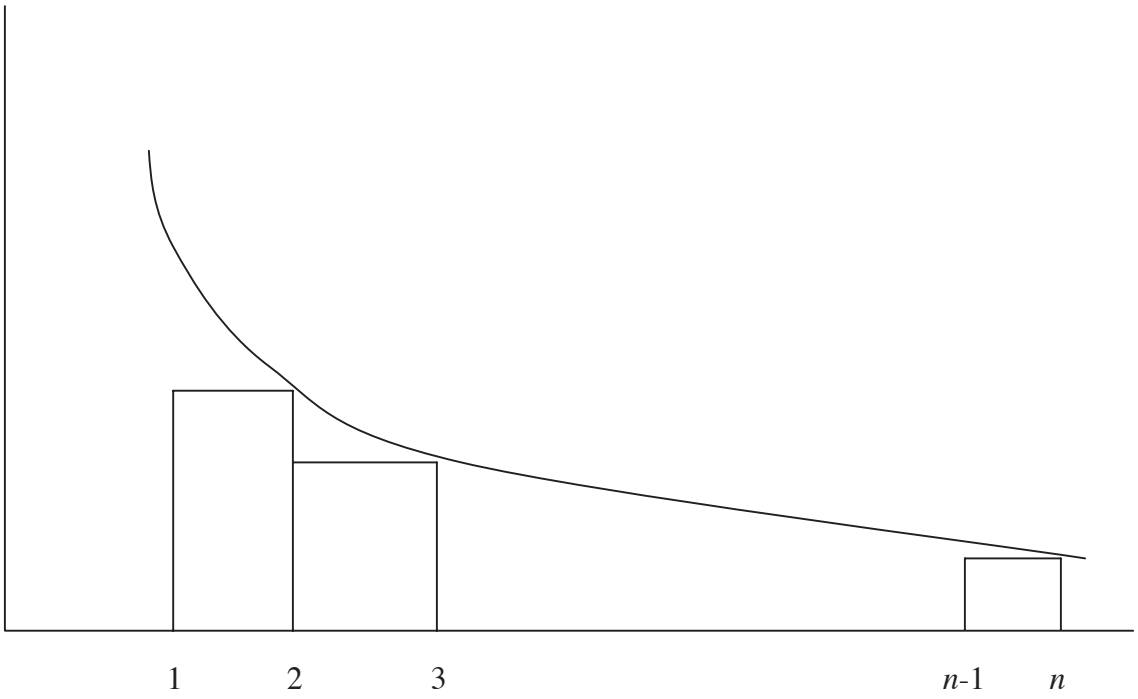
The loop invariant says that if i is a composite number $< d \times d$, then $P[i] = 0$. As is turns out, you don't need induction to prove this loop invariant. Since i is composite, it has some

factor greater than 1 but less than i. Let c be the smallest such factor (there must be a smallest one – that's the idea behind proving that the smallest exception isn't). We first show that $c < d$. The quotient i/c is also a factor of i which is greater that 1 but less than i. Since c is the smallest such factor of i, $i/c \geq c$, so that $i \geq c \times c$. Since $i < d \times d$, $c \times c < d \times d$; so, since c and d are both positive, $c < d$. We now prove that c must be a prime. If not, then c would have a factor f greater than 1 but less than c, in which case f would also be a factor of i, contradicting the minimality of c. Since c is a prime, $P[c]$ never gets set to 0; so when d was equal to c, the inner loop got executed and, since i is a multiple of c that is greater than c but less than or equal to n, $P[i]$ got set to 0.

Terminal step. After at most $\sqrt{n}$ iterations of the outer loop, $d \times d$ will be bigger than n and the algorithm terminates. Any $i \leq n$ is less than $d \times d$, so that, by the loop invariant, $P[i] = 0$. Eratosthenes got it right, QED.

How many operations does this algorithm take as a function of n? As an early estimate, we note that for every iteration of the outer loop the inner loop is executed at most n times. Each iteration of the outer loop increases d by 1, starting with $d = 2$ and ending when $d \times d > n$, so that the outer loop is executed about $\sqrt{n}$ times. This makes the total complexity of the algorithm $O(n\sqrt{n})$.

So far all the early estimates I have done have been good estimates. Is this early estimate too a good estimate? In estimating the number of iterations of the inner loop, I said that it gets executed at most n times for each d. In fact, it gets executed $\lfloor n/d \rfloor$-1 times. This number can only increase if we ignore the -1 and the floor function, so that the total number of iterations is at most $n\left(\frac{1}{2}+\frac{1}{3}+...+\frac{1}{\sqrt{n}}\right)$. The drawing below shows that $\frac{1}{2}+\frac{1}{3}+...+\frac{1}{n}$ is less than the area under the curve $y = 1/x$ from $x = 1$ to $x = n$, which is $\int_1^n \frac{1}{x}dx = \ln n$, so that $n\left(\frac{1}{2}+\frac{1}{3}+...+\frac{1}{\sqrt{n}}\right) < \frac{1}{2}\ln n$ and the complexity of the algorithm is in $O(n \log n)$.



Is **this** a good estimate? Ignoring -1 adds $\sqrt{n}$, ignoring the floor function adds at most $\sqrt{n}$ and replacing the area of the rectangles by the area under the curve adds only a constant; so these early estimates don't affect the growth rate of the time complexity. But I did make another assumption: that the inner loop would get executed for every value of d. In fact, the inner loop only gets executed when d is a prime. The real complexity is $n\left(\frac{1}{2}+\frac{1}{3}+\frac{1}{5}+\frac{1}{7}+...+\frac{1}{p}\right)$, where p is the biggest prime less than or equal to $\sqrt{n}$. To estimate that sum, we use a very advanced theorem of number theory – the **prime number theorem** – that says that for big n the number of primes less than or equal to n is about $n/\log_e(n)$, so that the odds that some big number n is a prime is about $1/\log_e(n)$. If instead of integrating $1/x$ we integrate $1/x\log_e(x)$ we get $\log_e\left(\log_e(n)\right)$, giving us a complexity estimate of $O(n\log\log n)$, which **is** a good estimate. And the moral of the story is? The more mathematics you know, the better you will be at estimating the complexity of algorithms!

The next algorithm on the list is one that finds all the prime factors of an integer $n \geq 2$ in increasing order (I proved earlier that n can be expressed uniquely as a product of primes in increasing order but I didn't yet say how to do it). For example, let $n = 8484$. We search for the smallest divisor of 8484 that is greater than 1. Starting with 2, we keep increasing the candidate divisor by 1 until we find an integer that divides 8484. As I pointed out in the proof of the previous algorithm, the smallest number greater than 1 that divides some number or other must be a prime. Since 2 divides 8484, it is our first prime factor. Now we search for the smallest divisor >1 of the quotient 8484/2 = 4242. Again 2 divides 4242; so 2 is our second prime factor and we search for the smallest divisor >1 of the quotient 4242/2 = 2121. Since 2 doesn't divide 2121 we try the next number 3, and since 3 does divide 2121, 3 must divide 8484 and we call 3 our third prime factor and search for the smallest divisor >1 of the quotient 2121/3 = 707. We don't have to start the search at 2 because if 2 divided 707 it would divide 2121 and we know that it doesn't; so we start with 3. We find that 707 isn't divisible by 3, 4, 5 or 6 but it is divisible by

7; so 7 is our next prime factor and we search for the smallest divisor of 707/7 = 101 starting with 7. We find that 101 isn't divisible by 7, 8, 9 or 10. Do we need to try 11? Since $11\times11 > 121$, if 11 divided 121, the quotient, which is also a factor of 101, would be smaller than 11, and we already know that no integer <11 divides 101; so 101 must be a prime – our last prime factor – and the prime factors of 8484 are, in increasing order, 2, 2, 3, 7, 101.

Here is the algorithm.

```
procedure prime_factors(n: natural; i: natural; P: array of natural);
                    {If n ≥ 2, then P[1], P[2],..., P[i] are the prime factors of n in increasing order.}
  local variables d,m,i: natural;
  m←n; i←0; d←2;
  while d*d ≤ m do                    {Loop invariant: P[1]×...×P[i]×m = n,  P[1] ≤ ... ≤ P[i],
                                       these numbers are all primes ≤ d and no integer >1 but < d divides m.}
    if d divides m then
      i←i + 1; P[i]←d; m←m/d;
    else
      d←d + 1;
    end if
  end while;
  i←i + 1; P[i]←m;
end prime_factors.
```

And here is the correctness proof. The precondition is that $n \geq 2$ – otherwise the algorithm will make $n = 0$ or 1 the sole prime factor and those numbers are not primes. The postcondition is that $P[1]\times\ldots\times P[i] = n$, that $P[1] \leq \ldots \leq P[i]$ and that these numbers are all primes.

Basic step. At the beginning of the first iteration of the loop, $i = 0$, so that there are no primes yet. The loop invariant says that $m = n$, which is true because m was set to n, and that no integer > 1 but < 2 divides m, which is true because there aren't any such integers.

Induction step. Suppose that at the beginning of some iteration of the loop, $P[1]\times\ldots\times P[i]\times m = n$, $P[1] \leq \ldots \leq P[i]$, these numbers are all primes $\leq d$ and no integer >1 but $< d$ divides m and that $d\times d \leq m$ so that the body of the loop will be executed. There are two cases to consider.

Suppose that d divides m. Then d is the smallest integer >1 that divides m; so d is a prime. The instructions $i\leftarrow i + 1$ and $P[i]\leftarrow d$ make $P[i]$, for the new value of i, a prime $\leq d$, so that $P[1] \leq \ldots \leq P[i]$ and these numbers are all primes $\leq d$, and also make $P[1]\times\ldots\times P[i]\times m$ equal to $n\times d$. Then the instuction $m\leftarrow m/d$ makes $P[1]\times\ldots\times P[i]\times m$ equal to n again. Any integer that divides the new m = the old m/d must divide the old m too, so that no integer >1 but $< d$ divides the new m either, and the whole loop invariant is satisfied.

Now suppose that d doesn't divide m. Then no integer >1 but $< d+1$ divides m, and the instruction $d\leftarrow d + 1$ arranges that for the new value of d, no integer >1 but $< d$ divides m. Also, all the primes $\leq d + 1$ = the new d. The rest of the loop invariant is unaffected by that instruction; so again the whole loop invariant is satisfied.

Terminal step. Each iteration of the loop either decreases m or increases d, so that after a finite number of iterations, $d \times d$ will grow greater than m. The loop invariant says that no integer >1 but $< d$ divides m. We show that m is a prime. If not, m would have a factor c greater than 1 but $< m$ so that $c \geq d$ and m/c, which is also a factor of m, satisfies $m/c \geq d$. Then $m \geq d \times d$, contradicting the termination condition. The instructions $i \leftarrow i + 1$ and $P[i] \leftarrow m$ arrange that the new $P[i]$ is a prime, so that $P[1],\ldots,P[i]$ are all primes. Since by the loop invariant $P[1] \times \ldots \times P[i-1] \times m = n$ and $P[i] = m$, $P[1] \times \ldots \times P[i] = n$. By the loop invariant, if $i > 1$, then $P[1] \leq \ldots \leq P[i-1]$ and these numbers are all primes $\leq d$. It remains to prove that if $i > 1$, then $P[i-1] \leq P[i]$. At the beginning of the last iteration of the loop, $d \times d \leq m$. If d divides m, then $d \leq m/d$, so that $d \leq$ the new m. Since at the end of the algorithm $P[i-1] = d$ and $P[i] = m$, $P[i-1] \leq P[i]$. If d doesn't divide m, then at the end of the algorithm $P[i-1] \leq d \leq d \times d \leq m = P[i]$. The algorithm works, QED.

How many operations does this algorithm do as a function of n? Each iteration of the loop either divides m by at least 2 or else adds 1 to d. If n is a prime, then each iteration will add 1 to d and there will be about $\sqrt{n}$ iterations. This is the worst case when n is large, because adding 1 to d multiplies $d \times d$ by less than 2 unless $d = 2$; so it contributes less than dividing m by d towards approaching the termination condition. The complexity of this algorithm is thus in $O(\sqrt{n})$. If you want to find a precise upper bound, try to prove by generalized induction that the number of iterations is never greater than $\lfloor \sqrt{n} \rfloor - 1$ unless $n = 8$.

It would be tempting to make d run only over primes, but to do so, you'd have to first pre-compute a list of all the primes less than or equal to $\sqrt{n}$, and that takes more time than the factorizing algorithm. One optimization that is often used is to make d skip all the multiples of 2 except 2 and all the multiples of 3 except 3. Then d would run over the following list of numbers: 2, 3, 5, 7, 11, 13, 17, 19, 23, 25, … . How would you generate that list of numbers? Starting with 5, the numbers increase alternately by 2 and 4; so you keep a variable c that alternates between 2 and 4 and add c instead of 1 to d as soon as d increases to 5. Try programming that version of the factorization algorithm – it's the one commonly used.

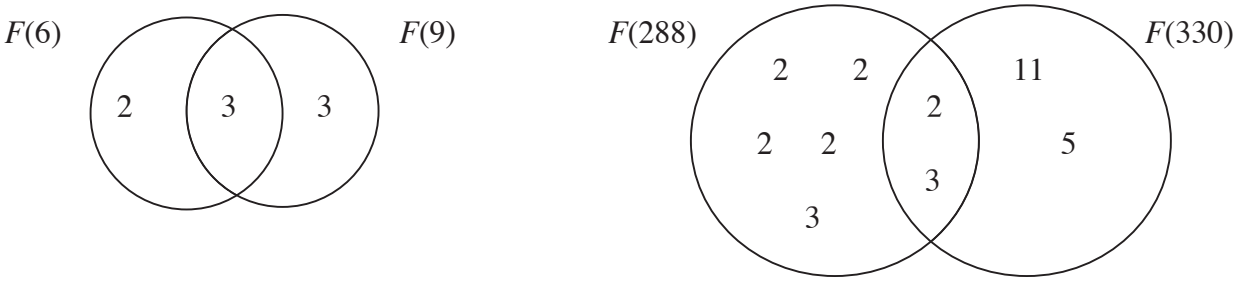
There are factorization algorithms that are asymptotically faster than $O(\sqrt{n})$, but none that runs in a time proportional to a power of the number of digits in the decimal representation of n, although it hasn't yet been proved that no such algorithm exists. If you discover such an algorithm, all the world's spy organizations will be after you, because a commonly used method of coding messages depends on the fact that big integers can't (yet) be factored quickly and your discovery will make their messages insecure. That is the premise behind the film Sneakers.

Number theory has been used to code messages for thousands of years. One such code was invented by Julius Ceasar. To code a message, you take every letter in the message and advance it by 3 letters in the alphabet, so that A becomes D, B becomes E and so on until W becomes Z. Now X, Y and Z have no code and A, B and C aren't the code for anything; so X is coded as A, Y as B and Z as C. Number theory comes in because if you set A = 0, B = 1, … , Z = 25, then the coded letter = (the original letter + 3) mod 26 and the original letter = (the coded letter – 3) mod 26.

Suppose that Julius Caesar wants to engage his enemy Pompeii in battle. He sends his general a message instructing him to do so. He would, of course, write his message in Latin before coding it, but we'll write it in French instead. The message is ALLEZY (contracted from ALLEZ-Y to avoid giving it away) and is coded as DOOHCB so that any spy intercepting the message will think that the courier carrying the message is some kind of nut and let him pass. The general receives the message, decodes it and understands what Caesar wants him to do but, being in the field, he has information that Caesar doesn't have: the enemy is not Pompeii, as Caesar thinks, but someone whose army is too ferocious for Caesar's army to defeat. To explain his refusal to engage this army in battle, he sends a coded message to Caesar identifying the enemy: AHQD. You decode the message. The answer appears at the end of this chapter.

As a final example, we consider two algorithms for calculating GCD(a,b), the *greatest common divisor* of the integers a and b, defined as the greatest integer d such that d divides a and d divides b. Note that GDC(0,0) is not defined. Every integer d divides 0; so the set of common divisors of 0 and 0 is the set of all the integers, which has no greatest member. On the other hand, if $a \neq 0$, GCD(a,0) = GCD(0,a) = $|a|$, the absolute value of a, because every integer divides 0, so that GCD(a,0) = GDC(0,a) = the greatest divisor of a, which is $|a|$. In general, GCD(a,b) = GCD($|a|$,$|b|$), and it is a positive integer, because if d divides a, then both $+d$ and $-d$ divide both $+a$ and $-a$.

The greatest common divisor of two positive integers a and b can be expressed in terms of the *multiset* (set whose elements are not necessarily distinct) of prime factors of each of them – call these multisets $F(a)$ and $F(b)$. Then GCD(a,b) is the product of all the prime factors in $F(a) \cap F(b)$. For example, $F(6)$={2,3} and $F(9)$={3,3}, the intersection of these two multisets is {3} (see the left Venn diagram below) and GCD(6,9)=3. Similarly (see the right Venn diagram below) GCD(288,330)=6.



The *least common multiple* of a and b, denoted by LCM(a,b), is defined as the smallest positive integer m such that a divides m and b divides m. Clearly (which means "I'm too lazy to prove it; so you do it") LCM(a,b) is the product of all the prime factors in $F(a) \cup F(b)$. For example, LCM(6,9) = 2×3×3 = 18. How would you calculate LCM(288,330)? You could multiply all the prime factors in $F(288) \cup F(330)$, but there is an easier way. The product ab is the product of all the prime factors in $F(a)$ multiplied by the product of the prime factors in $F(b)$, which includes all the prime factors in $F(a) \oplus F(b)$ once and all the prime factors in $F(a) \cap F(b)$ twice; so to go from $F(ab)$ to $F(a) \cup F(b)$ you have to divide ab by the product of all the prime factors in $F(a) \cap F(b)$ so that they will be included only once. This means that LCM(a,b) = (ab)/GCD(a,b), but if you want to avoid working with unnecessarily big numbers you wouldn't use that formula as is. For example, LCM(6,9) = (6×9)/3 = 54/3 = 18, but you can

avoid working with such a big number as 54 by dividing first: either calculate $(6/3)\times 9 = 2\times 9 = 18$ or calculate $(6/2)\times 6 = 3\times 6 = 18$. Similarly LCM(288,330) = $288\times 330/6 = 288\times 55 = 1440\times 11 = 15840$. In general, once you know GCD(a,b) you can calculate LCM(a,b) in $O(1)$ operations, so that any algorithm for computing GCD also computes LCM – you get two for the price of one (plus $O(1)$ more operations).

One algorithm for computing GCD follows from the above expression of GCD(a,b) in terms of the prime factors of a and b: you factorize a and b, compute $F(a)\cap F(b)$ and take the product of all its members. Using the above factorization algorithm, this takes $O(\sqrt{|a|}+\sqrt{|b|})$ operations in the worst case, when $|a|$ and $|b|$ are both primes. A somewhat more efficient version of the same procedure is illustrated by the example below. Suppose you want to calculate GCD(288,330). You initialize g to 1, d to 2, n to 288 and m to 330. Since 2 divides both 288 and 330, you multiply g by 2 (it is now 2), divide n by 2 (it is now 144) and divide m by 2 (it is now 165). Since 2 doesn't divide **both** 144 and 165, you increase d from 2 to 3. Since 3 divides both 144 and 165, you multiply g by 3 (it is now 6), divide n by 3 (it is now 48) and divide m by 3 (it is now 55). Since 3 doesn't divide both 48 and 55 you increase it to 5 (remember, we're skipping multiples of 2 and 3 except 2 and 3 themselves), which also doesn't divide both 48 and 55, and then to 7, which you don't need to test because $7\times 7 = 49 > 48$. The final value of g, which is 6, is GCD(288,330). This algorithm takes $O(\min(\sqrt{|a|},\sqrt{|b|}))$ operations in the worst case – when GCD(a,b) = 1. I could ask you to write a pseudocode for this algorithm and prove its correctness, but I won't, because there is a more efficient way to compute GCD(a,b).

Here is the pseudocode of the algorithm, the so-called Euclidean algorithm. It is an optimized version of Euclid's algorithm for computing GCD(a,b).

```
natural function Euclid(a,b: integer);
                    {If a and b are not both 0, Euclid(a,b) is the greatest common divisor of a and b.}
  local variables m,n,r: natural;
  m←max(abs(a),abs(b)); n←min(abs(a),abs(b));          {abs(x) is the absolute value of x}
  while n > 0 do                  {Loop invariant: m > 0, n ≥ 0 and GCD(m,n) = GCD(a,b).}
    r←m mod n;                                  {r is the remainder of the division of m by n}
    m←n;
    n←r;
  end while;
  return m;                                                  {m = GCD(m,0) = GCD(a,b)}
end Euclid.
```

Let's trace this algorithm by computing GCD(-288,330).

```
r       42  36   6   0
m 330 288  42  36   6  (value returned)
n 288  42  36   6   0
```

The algorithm returns the correct value of GCD(-288,330). It's black magic! Or, at least it seems like black magic until you observe that GCD(330,288) = GCD(288,42) = GCD(42,36) = GCD(36,6) = GCD(6,0) = 6, and if you don't believe me, calculate all these greatest common

divisors the old way. Is it a coincidence that GCD(m,n) stays the same from one iteration of the loop to the next?

Let (m,n) be a pair of positive integers. Then there are integers q and r such that $m = qn + r$ and $0 \le r < n$. If a non-zero integer d divides n and r, then d must divide m: m/d is the integer $q(n/d) + (r/d)$. Every common divisor of n and r is a common divisor of m and n. Similarly, if d divides m and n, then d must divide r: r/d is the integer $(m/d) - q(n/d)$. Every common divisor of m and n is a common divisor of n and r. The set of common divisors of n and r is equal to the set of common divisors of m and n. Since $m > 0$ and $n \ge 0$, this set has a greatest member, so that the greatest common divisor of n and r is equal to the greatest common divisor of m and n – that is, GCD(n,r) = GCD(m,n). This is most of the induction step of the following correctness proof.

Precondition: a and b are not both 0; otherwise Euclid(a,b) = 0 and GCD(0,0) is not defined.

Postcondition: Euclid(a,b) = GCD(a,b).

Basic step. At the beginning of the first iteration of the loop, m is the larger of the absolute value of a and the absolute value of b and n is the smaller one. These numbers are both non-negative and since they are not both 0, the larger one m is positive. Also, GCD(m,n) = GCD(a,b) because GCD(a,b) is not changed by changing the sign of a and/or b or by exchanging them. The loop invariant is satisfied.

Induction step: Suppose that at the beginning of some iteration of the loop, the loop invariant is satified and that $n > 0$ (the condition for the body of the loop to be executed), so that m and n are both positive and GCD(m,n) = GCD(a,b). As we have shown above, if r is the remainder of the division of m by n, then GCD(n,r) = GCD(m,n), so that GCD(n,r) = GCD(a,b). When n is copied into m and r is copied into n, for the new values of m and n, GCD(m,n) = GCD(a,b). Also, since n was positive, m will be positive, and since r is always non-negative by the definition of remainder, n will be non-negative; so the loop invariant is satisfied at the beginning of the next iteration.

Terminal step. Since, by the definition of remainder, $r < n$, when r is copied into n, the new n will be less than the old one but still non-negative. After a finite number of iterations, n must drop to 0 and the algorithm jumps out of the loop. By the loop invariant, GCD(m,n) = GCD(a,b). But now $n = 0$ and, since $m > 0$ by the loop invariant, GCD(m,n) = m. The algorithm returns m =GCD(a,b), so that Euclid(a,b) = GCD(a,b). Euclid got it right, QED.

How many iterations will this algorithm do as a function of $n = \min(|a|, |b|)$ in the worst case? To calculate the greatest number d of iterations that this algorithm will do as a function of n, we ask the inverse question: what is the smallest value of n that will force this algorithm to do d iterations? Call this number $f(d)$. Then, if $n < f(d)$, the algorithm will do fewer than d iterations – that is, at most d - 1 of them.

If $n = 0$ the algorithm will do 0 iterations; so $f(0) = 0$. If $n = 1$ the algorithm will do 1 iteration because dividing by 1 leaves a remainder of 0; so $f(1) = 1$. If $n = 2$, the algorithm will do either 1 iteration (if the bigger number is even) or two of them (if the bigger number is odd);

so $f(2) = 2$. For $d > 2$, consider the sequence of successive values $n = x_d, x_{d-1}, \ldots, x_1, x_0 = 0$ taken by the variable n. When $n = x_i$, m will be x_{i+1} and r will be x_{i-1}. This observation holds for $i = 1, 2, 3, \ldots, d - 1$ and can be made to hold for $i = d$ by defining x_{d+1} to be $\max(|a|,|b|)$. We recall that $m = qn + r$ for some integer q, so that $x_{i+1} = qx_i + x_{i-1}$. In general, $x_{i+1} > x_i > x_{i-1}$, the sole exception being that x_{d+1} could be equal to x_d if $|a| = |b|$. In that case only one iteration will be done; so this case can be ignored since $d > 2$. The general case of the inequality implies that $m > n > r$ so that $q \geq 1$. Given n and r, m is minimized by setting q equal to 1 so that $x_{i+1} = x_i + x_{i-1}$ for each $i \geq 2$. To minimize n for a given value of d, we set x_0 to 0, x_1 to 1, x_2 to 2 and for each $i \geq 2$ we set $x_{i+1} = x_i + x_{i-1}$. This means that $n = f(d)$ (and $m = f(d+1)$) where $f(0) = 0, f(1) = 1, f(2) = 2$ and $f(i) = f(i - 1) + f(i - 2)$ for every $i \geq 3$. This recurrence is the same one satisfied by the Fibonacci numbers; the only difference is the second Fibonacci number is 1 and the third one is 2; so that unless $d = 0$, $f(d)$ is the $(d+1)$st Fibonacci number. If you want to make the algorithm do d iterations for the smallest possible positive a and b, you make a and b the $(d+1)$st and $(d+2)$nd Fibonacci number, respectively. For example, the fifth Fibonacci number is 5 and the sixth one is 8. The sequence of values taken by n will be 5,3,2,1,0 – it will take 4 iterations for the algorithm to terminate.

Inversely, if $n = \min(|a|,|b|)$ is less than the $(d+1)$st Fibonacci number, the algorithm will do no more than $d - 1$ iterations. We proved in Section 4.2 that the $(d+1)$st Fibonacci number is greater than or equal to 1.5^{d-1} and we showed how to modify that proof to obtain the sharper bound $\left((1+\sqrt{5})/2\right)^{d-1}$, which is never attained for $d \geq 2$ because a Fibonacci number is an integer and $\left((1+\sqrt{5})/2\right)^{d-1}$ isn't an integer. If $n \leq \left((1+\sqrt{5})/2\right)^{d-1}$, which is about 1.61^{d-1}, then the algorithm will do at most $d - 1$ iterations; so the number of iterations can never exceed the inverse function, which is the logarithm of n to the base of about 1.61. By changing the base to 10, Lamé proved that the number of iterations cannot exceed five times the number of decimal digits of n. For our purposes it suffices to observe that the number of operations done by this algorithm is in $O(\log n)$.

So now we have two algorithms for computing GCD(a,b): factorization, which runs in $O(\sqrt{n})$ operations using the naïve algorithm presented here and for which no known algorithm runs in any power of $\log n$, and the Euclidean algorithm, which runs in $O(\log n)$. To choose between those two algorithms, you have to be able to analyze them both and to know that $\log n$ grows slower than any positive power of n, so that for big n the Euclidean algorithm runs faster than any algorithm that includes factorization. The more mathematics you know, the better you will be at choosing the fastest algorithm to solve your problems!

Earlier in this section, I promised that at the end of the chapter I would tell you who Caesar's mysterious, ferocious enemy is. It's Xena, the warrior princess, who commands an army of Amazons. In the episode that inspired that anecdote, she doesn't command an army of Amazons. Instead, she tricks Caesar and Pompeii into fighting each other so that their armies would destroy each other, like the two giants who get tricked by the little tailor, and would no longer be strong enough to menace the community she wants to save. Fortunately for her, nobody in either Caesar's or Pompeii's army is wise to her trick; so the generals in the field obey their respective leaders' instructions to commence hostilities, unlike the general in my anecdote, and it's the crafty Xena who emerges victorious.