33 Running Time of Methods

1 Running time of methods

The addition of two numbers (of type `int`, say) on a machine takes a certain amount of time. The actual time taken depends on the machine, but a typical time for a good desktop machine might be, say, 10 nanoseconds. A nanosecond is one billionth of a second, and is abbreviated as *ns*. Other primitive operations such as multiplying or comparing two numbers, or and'ing two booleans, similarly take about 10 ns to complete. The following table gives some examples of the time to evaluate expressions:

Expression	*Time to evaluate (ns)*
`n*3`	10
`(n+3)*(n+3)`	30
`(100<n*n) && (n*n<=1000)`	50

Note that the time to carry out primitive operations on integers does not depend on how big or how small the number is: it takes as long to add 5 and 7 as is does to add 1265467 and 96785.

Invoking a method takes time, typically about 50 ns plus 10 ns per parameter (in addition to the time to evaluate arguments). A return statement takes about 50 ns, plus the time taken to evaluate any expression returned (the final action in a procedure is always a return, even if

there is no explicit return statement). For expensive machines, all these figures can be divided by a factor of 10 or more, and for more modest machines they can be multiplied by 10 or more. The total time for a method (or any piece of code) to run to completion is called its *running time*, (or *execution time*, or *time cost*, or simply *cost*). Consider the following method, for example:

```
static int foo(int n) {
    return (n*n+1)*(n*n+1);
}
```

Its running time is 60 ns to invoke the method (50 ns set up time, plus 10 ns for the single parameter), plus 50 ns for three multiplications and two additions, plus 50 ns for the return statement. That amounts to 160 ns in total. This will vary according to the machine on which the program runs. Note that it does not depend on the value of the argument: for example, the time to evaluate `foo(1)` is the same as the time to evaluate `foo(97)`.

Assignment statements take 10 ns plus the time to evaluate the expression on the right-hand side. For example, the time taken to execute `x=x*x+1` is 30 ns (10 ns each for `*`, `+`, and `=`). Array indexing (subscripting) takes 50 ns. For example, `w[i+1]=x` takes 70 ns (10 for `+`, 50 for `[]`, and 10 for `=`), as does `x=w[i+1]`. The time taken to execute a sequence of statements is the sum of the times to execute them individually. Consider the following alternative coding of `foo()`, for example:

```
static int foo2(int n) {
    int temp = n*n+1;
    return temp*temp;
}
```

The running time of `foo2(n)` is 150 ns – it is marginally faster than `foo(n)` because `n*n+1` is evaluated just once.

The time taken to execute an if-statement

```
if (expr) then stmtsT else stmtsF
```

is the time to evaluate `expr` plus the time to execute `stmtsT` if `expr` yields `true`, or `stmtsF` if `expr` yields `false`. For example, the time to execute

```
if (x>0) then x=1; else x = x*x;
```

is 20 ns if the boolean yields `true` (10 ns for `>` and 10 ns for the assignment) and 30 ns otherwise (10 ns for `>`, 10 ns for `*`, and 10 ns for the assignment). It frequently turns out in practice that each branch of an if-statement takes roughly the same amount of time to execute,

as in the preceding example. When that is the case, we take the cost of the if-statement as the cost of the slowest branch. For example, we would usually say that the cost of the preceding if-statement is 30 ns. Although this may lead to a slightly pessimistic view of the running time, the error is negligible.

Declarations of variables take no time, unless the variables are initialised in which case they are costed as assignments.

We have made a minor simplification in assuming that the primitive operations each take the same amount of time. In real machines there are some small differences: multiplication of reals, for example, would typically take a bit longer than multiplication of integers. It will become clear later that the simplification we have made is insignificant.

2 Running time of loops

Simple Loops

The running time of a while-loop is more difficult to measure because it depends on how many times the loop body is executed. Suppose that the body of `while(expr){stmts}` is executed *k* times. Then its running time is *k* times the time to execute `stmts`, plus *k*+1 times the time to evaluate `expr`. Consider, for example, the following code:

```
i = 0; s = 0;
while (i<100) {
    s = s+i*i; i= i+1;
}
```

Observe that the body of the loop is executed 100 times. The execution time of the code is 20 ns for the initial two assignments, plus 100×50 ns for the loop body, plus 101×10 ns for the boolean expression, giving a total running time of 6030 ns.

It is not usual that we can tell by inspection how many times the loop body is executed. Consider method `squares(n)` below which computes $1^2+2^2+3^2+...+n^2$ (for example, `squares(3)` yields 14):

```
static int squares(int n) { // return 1*1+2*2+3*3+...+n*n, 0<=n
    int i = 0; int s = 0;
    while (i<n) {
        i = i+1; s = s+i*i;
    }
    return s;
}
```

Observe that the time taken depends on the value supplied for parameter `n`. If `n` is 10, say, the loop body is executed 10 times, whereas if `n` is 1000 it is executed 1000 times. Clearly any measure of its running time will be an expression in which `n` occurs. In fact the execution time is 50 ns (invocation) plus 10 ns (parameter) plus 20 ns (initial assignments) plus (`n`+1)×10 ns (boolean expression) plus `n`×50 ns (loop body) plus 50 ns (return), giving a total running time of 140+60`n` ns. For example, the running time of `squares(100)` is 140+60×100 ns which is 6140 ns, whereas the running time of squares(1000) is 140+60×1000 ns which is 60140 ns. Evidently, if we increase the argument by a factor of *c*, the running time is also increased by a factor of *c*. We sometimes express this more informally as "double the argument, double the running time".

We can discover the running times for the machine on our desktop empirically. We have no hope of measuring times that are minute fractions of a second, but we overcome this by invoking the method a million times, say, and dividing the elapsed time by a million. A stop-watch suffices for reading the start and end-times, but it is easier to use the following method from class `System`:

```
static long nanoTime()
```

`System.nanoTime()` yields the elapsed time in nanoseconds since some base time (a nanosecond is one billionth of a second, and is abbreviated as *ns*). Note that the return type is `long`, the 64-bit integer type. The following is a program to measure running times of `squares()`:

```
class TimeMethod {

    static int squares(int n) {    .....
    }

   public static void main(String[] args) {
        int arg = 100; // method will be timed for argument arg
        int numCalls = 100000; // number of invocations of method
        long startTime = System.nanoTime();
        for(int i=0; i<=numCalls; i++)
            squares(arg);
        long endTime = System.nanoTime();
        long runningTime = (endTime-startTime) /numCalls;
        System.out.println(runningTime + " nanosecs");
   }
}
```

Sample timings on a basic desktop machine (call it Machine A) are roughly 1400 ns for `squares(100)` and 14000 ns for `squares(1000`. Although Machine A is evidently faster than the ideal machine for which we are calculating timings, it nevertheless behaves in accordance with the theoretical analysis, differing only by a constant factor (the speedup factor

being about 4.4 regardless of the argument). Expressed another way, the relationship between argument and running time is the same in both cases: doubling the argument of `squares()` doubles the running time.

A user's program can be temporarily suspended by the operating system when it must attend to other matters (such as monitoring communications on the network to which it is connected). In order to factor out time lost during any suspensions, you should confirm empirically determined running times by repeating the experiment a few times.

Nested loops

Consider method `allSquares()` below which returns an array of sums of squares. For example, an invocation of `allSquares(5)` returns an array of five elements as follows:

0	1	5	14	30

Component k of the array contains $0^2+1^2+...+k^2$. For example, component 3 contains $0^2+1^2++2^2+3^2$, i.e. 14.

```
static int[] allSquares(int n) {
// return sqrs[0..n-1] , n>0, where sqrs[k] equals 0*0+1*1+2*2+...+k*k
    int[] sqrs = new int[n];
    int k = 0;
    while (k<n) {
        sqrs[k] = squares(k); // running time not constant here!
        k = k+1;
    }
    return sqrs;
}
```

Observe that the loop body includes an invocation of `squares(k)` whose running time depends on the value of `k` (the higher the value of `k`, the greater the running time of `squares(k)`). As a consequence, the running time of the loop body, i.e.

```
sqrs[k] = squares(k);
k = k+1;
```

is not constant but increases with each successive iteration. For example, on the first iteration `k` has the value 0 and we calculate the running time to be 220 ns (50 for `[]`, 10 for =, 140+60×0 for `squares(0)`, 10 for =, and 10 for +), whereas on the 101st iteration `k` has the value 100 and the running time turns out to be 6220 ns. We say a loop is *simple* if the running time of its body is constant. The loop in `allSquares()` is not simple. A loop with a nested loop, or which invokes a method that employs a loop (as in the case of `allSquares()`), is rarely simple.

The running time of `allSquares(n)` turns out to be $240+200n+30n^2$ ns (the detailed calculations are given later). For `n`=1000, the cost is 30200240 ns or about 30 ms. Projected running times for other values of `n` are shown below, together with empirically measured running times for Machine A.

n	1000	2000	10000
Theoretical machine	30 ms	120 ms	3000 ms
Machine A	7 ms	27 ms	700 ms

Naturally the actual timings differ on the two machines, but the speedup factor is roughly the same at about 4.4 for all values of `n`. Observe also in both cases that if we multiply the argument by a factor of *c*, we multiply the running time by c^2. For example, the running time of `allSquares(2000)` is about 4 times the running time of `allSquares(1000)`, and the running time of `allSquares(10000)` is about 25 times the running time of `allSquares(2000)`.

The detailed calculation of the running time of `allSquares(n)` follows. It can be omitted without loss of continuity. The running time of the loop body is $220+60k$ ns, made up of $140+60k$ ns for `squares(k)` plus 80 ns for the other operations. Adding in 10 ns for the evaluation of the boolean expression, the cost of each iteration is $230+60k$ ns. The loop body is executed for `k` ranging from 0 to `n`-1, and so its total cost is:

$(230+60\times0) + (230+60\times1) + (230+60\times2) + \ ... \ + (230+60\times(n-1))$
$= \quad 230\times n + 60\times(1+2+ .. +(n-1))$
$= \quad 230n + 60\times(n(n-1)/2)$ (we use the fact that $1+2+..(n-1) = n(n-1)/2$)
$= \quad 230n + 30(n^2-n)$
$= \quad 200n + 30n^2$

For the total running time of `allSquares(n)` we add in the final evaluation of the boolean expression (10 ns), the invocation time (60 ns), the time to execute the return statement (50 ns), the assignments to variables `k` and `sqrs` (20 ns), and the cost of invoking `new`. A typical cost of invoking `new` is 100 ns. Adding up, the total execution time is $240+200n+30n^2$ ns.

3 Algorithm vs machine

The running time of a method which performs a certain function depends on both the speed of the machine and the quality of the coding. A method will run faster on a faster machine, and two different methods that perform the same function will have different running times on the same machine. More often than not, it turns out that the quality of the coding is more significant than the quality of the machine. For example, the following method has the same behaviour as `allSquares()` which was presented in the preceding section:

```
static int[] allSquares2(int n) {
```

```
// return sqrs[0..n-1] , n>0, where sqrs[k] equals 1*1+2*2+...+k*k
    int[] sqrs = new int[n];
    sqrs[0] = 0;
    int k = 1;
    while (k<n) {
        sqrs[k] = sqrs[k-1]+k*k;
        k = k+1;
    }
    return sqrs;
}
```

In this version, the running time of the loop body

```
sqrs[k] = sqrs[k-1]+k*k;
k = k+1;
```

is constant (in fact, 160 ns). Using the fact that the loop body is executed `n`-1 times, we easily calculate the running time as 120+170`n` ns. Some running times for various values of `n` are given in the table below in microseconds (a microsecond is one millionth of a second, abbreviated µs).

`n`	1000	2000	10000
Theoretical machine	170 µs	340 µs	1700 µs
Machine A	44 µs	86 µs	443 µs

First observe that experiment agrees with theory, allowing for a speedup factor of course. The speedup factor here is about 3.9. That this differs from previous speedup factors is to be expected as different operations will have different speedup rates. For example, addition of reals could be 10 times faster on one machine than on another, whereas multiplication might be only 5 times faster. Hence the speedup of a method that uses additions only will be greater than one which uses multiplications.

More importantly, observe that although `allSquares2()` does the same job as `allSquares()`, it is much faster. Indeed the speed gain is of a different quality than the speed gain achievable with a faster machine. A fast machine will speed up `allSquares(n)` by a constant factor, regardless of the value of `n`. The improved coding in `allSquares2(n)`, however, speeds up the computation by a factor that increases as `n` increases, and the rate of increase is dramatic. For n equal to 1000, 2000, and 10000 in turn, `allSquares2(n)` is faster than `allSquares(n)` by a factor of about 175, 350, and 1750, respectively. In general, the speedup factor is doubled when the argument is doubled. The moral is that a faster algorithm, if there is one, beats a faster machine by a distance.

Our primary interest is not in the absolute running time of a method, but in comparing the

relative merits of algorithms independently of the machine on which they run. For example, our analysis shows that the algorithm of `allSquares2(n)` is fundamentally faster than the algorithm of `allSquares(n)`, and the extent to which it is faster increases as `n` increases.

4 Time complexity

We calculated the running time of `squares(n)` above to be 140+60`n` ns. If the timings of the primitive operations were halved then the running time would be 70+30`n` ns, and if they were doubled it would be 280+120`n` ns. If the costs of + and * were increased five-fold, the cost of method invocation and return were halved, and the cost of all other operations remained unchanged, the running time would be 85+180`n` ns. It should be evident that the running time on any machine will have the shape $A+B$`n` where A and B stand for some constants. Only A and B vary from machine to machine. We say that the *time complexity function* or simply the *time complexity* of `squares(n)` is $A+B$`n` where A and B denote constants. When we state that the time complexity is $A+B$`n`, or whatever it may be, we do not state units such as nanoseconds or microseconds – that would be pointless as we don't know the values of A and B. If we wanted to know the values of A and B for a particular machine we could discover them empirically, in which case we would express A and B as nanoseconds or milliseconds. However, we are usually not interested in knowing the values of the constants.

As another example, recall that we calculated the running time of `allSquares(n)` as 240+200`n`+30`n`^{2} ns. This will change from machine to machine, but it will always be of the form $A+B$`n`$+C$`n`^{2} for A, B, and C some constants (A and B here are unrelated to A and B in the time complexity of `squares()`). Hence the time complexity of `allSquares(n)` is $A+B$`n`$+C$`n`^{2}. In contrast, we calculated the running time of `allSquares2(n)` as 120+170`n` ns, and so its time complexity is $E+F$`n` where E and F denote constants.

For a final example, recall that the running time of `foo(n)` is 160 ns. This figure will vary with the machine on which we run `foo()`, but it will always be a constant independent of `n`. The time complexity of `foo(n)` is therefore A where A denotes some constant.

We use time complexities in two ways. First, time complexity gives us a strong indication of how fast a method is, independently of the machine on which it runs. Although it doesn't tell us the running time for a particular machine, it tells us how the running time changes as the arguments change. To take the example of `squares()`, the time complexity allows us to say how longer it takes to run `squares(2*n)` or `squares(3*n)` in comparison with `squares(n)`. The time complexity of `squares(n)` is $A+B$`n`, and so the time to run `squares(`c`*n)` for c some constant is $A+Bc$`n`. Hence the time to run `squares(`c`*n)` is greater than the time to run `squares(n)` by a factor of $(A+Bc$`n`$)/(A+B$`n`$)$. As `n` increases, the relative influence of A in both $A+B$`n` and $A+Bc$`n` diminishes to insignificance, and so $(A+Bc$`n`$)/(A+B$`n`$)$ is approximately c for all but small values of `n`. Therefore the running time of `squares(`c`*n)` is about c times the running time of `squares(n)`. In short, if we double the size of the problem, we double the time it takes to solve it. We regard this as a moderately fast

solution. Most tasks in everyday life have a similar characteristic: it takes us twice as long to polish a floor that is twice the size. If we go through the same exercise for `allSquares()`, we find that the running time of `allSquares(`c`*n)` is c^2 times the running time of `allSquares(n)`. In this case, if we double the size of the problem, we multiply by 4 the time it takes to solve it. This is computationally expensive, because no matter how fast the machine, the running time increases rapidly as the size of the problem increases, and we soon reach a limit beyond which we cannot solve the problem in a reasonable time. We have already seen the figures.

The second role of time complexity is in comparing alternative solutions to a problem. The time complexity of a method is really a measure of the underlying algorithm rather than a particular encoding of it. Rearranging the code or modifying it in small ways to save a few assignments or multiplications does not change the time complexity – the value of the constants (called *A*, *B*, etc above) will change, but not the overall form.

5 Calculating time complexity OPTIONAL

We can calculate the time complexity of a method without knowing actual timings. First, view the method as being composed of blocks of code each of whose running times is constant. For each block, introduce a name for the time it takes, and note the number of times it is executed. For example, recall `squares()` from a preceding section:

```
static int squares(int n) { // return 1*1+2*2+3*3+...+n*n, 0<=n
    int i = 0; int s = 0;
    while (i<n) {
        i = i+1; s = s+i*i;
    }
    return s;
}
```

We identify the following constituent blocks of code in `squares(n)`:

code	*cost*	*no of executions*
invocation	C_0	1
`i=0; s=0;`	C_1	1
`i<n`	C_2	`n`+1
`i=i+1; s=s+i*i;`	C_3	`n`
`return s;`	C_4	1

Now add up the costs of the constituent component, which in the case of `squares(n)` is C_0+C_1+C_2(`n`+1)+C_3`n`+C_4, which is equivalent to (C_0+C_1+C_2+C_4) + (C_2+C_3)`n`, which we can write as A+B`n`, where A=C_0+C_1+C_2+C_4 and B=C_2+C_3.

We give further examples below, all of which are optional.

Example 1: sum of squares

A well-known formula in mathematics is that $1^2+2^2+3^2+...+n^2 = n(n+1)(2n+1)/6$. We can use this to write a faster version of `squares()`:

```
static int squares2(int n) { // return 1*1+2*2+3*3+...+n*n, 0<=n
    return (n*(n+1)*(2*n+1))/6;
}
```

`squares2(n)` consists of a single statement whose running time does not vary with `n`, and therefore its time complexity is some constant *A*. We can make a new version of `allSquares()` – call it `allSquares3()` – which is like `allSquares()` except that it invokes `squares2()` in place of `squares()`. The time complexity of `allSquares3(n)` turns out to be $A+Bn$ (it is an easy exercise to show this). Although this is the same time complexity as that of `allSquares2()`, the respective running times on a particular machine will be different. The difference is too small to register in the time complexity function, however.

Example 2: integer square root

The following method computes the integer square root of a natural number, i.e. the square root with any fraction dropped. For example, an invocation of `intSqrt(18)` returns 4, and an invocation of `intSqrt(9)` returns 3.

```
static int intSqrt(int n) { // square root of n rounded down, n>=0
    int i = 0;
    while ((i+1)*(i+1)) <= n) {
        i = i+1;
    }
    return i;
}
```

We show that `intSqrt(n)` has time complexity $A+B\lfloor\sqrt{n}\rfloor$, where *A* and *B* denote constants and $\lfloor\_\rfloor$ denotes rounding down (e.g. $\lfloor 3.79\rfloor=3$). `intSqrt()` has the same shape as `squares()` and `allSquares2()`, i.e. some initialisation statements, followed by a loop whose body is simple, and a return statement. Hence by exactly the same reasoning as we applied previously, we can deduce that the time complexity has the form $A+Bk$, where *k* denotes the number of times the loop body is executed. Now the loop is iterated for `i` = 0, 1, 2, ... until $(i+1)^2>n$, i.e. until $i>\sqrt{n}-1$, i.e. until $i=\lfloor\sqrt{n}\rfloor$. Therefore $k=\lfloor\sqrt{n}\rfloor$.

Example 3: selection sort

The following is an implementation of selection sort:

```
void sort(int[] w) {
```

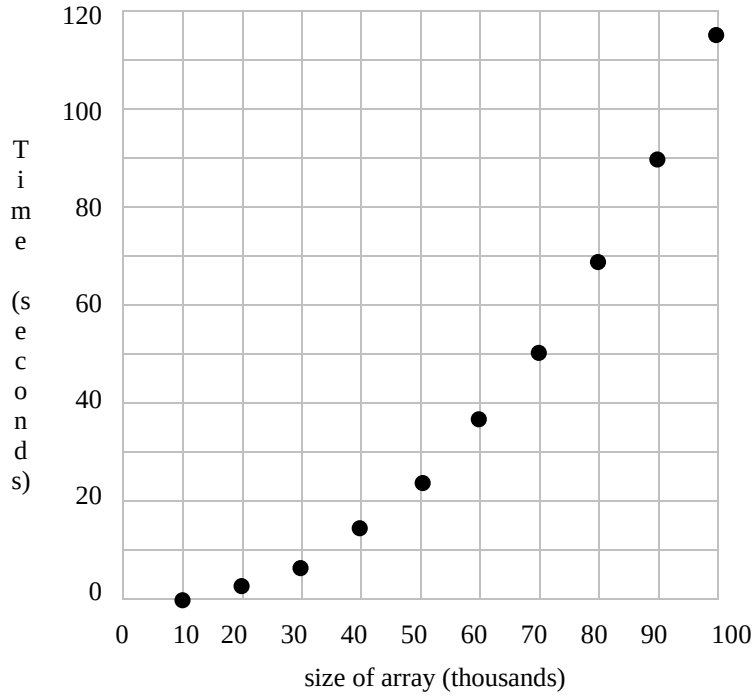
```
        int i = 0;
        while (i<w.length) {
           int m = i; int j = i+1;
           while (j<w.length) {
              if (w[j]<w[m])  m = j;
              j++;
           }
           int t = w[i]; w[i] = w[m]; w[m] = t;
           i++;
        }
    }
```

We show that the time complexity of `sort(w)` is $E+Fn+Gn^2$ for some constants E, F, and G, where `n` abbreviates `w.length`. Observe that the body of the inner loop is executed for `j` in the range `i+1` to `n` in steps of 1, and hence is executed `n–(i+1)` times, i.e. `n–i–1`. Therefore by the same reasoning as previously, the cost of a single execution of the body of the inner loop is $C+B$(`n–i–1`), for some constants B and C, which we may write as $A+B$(`n–i`) where $A=C–B$. Now the body of the main loop is executed for `i` = 0, 1, 2, ..., `n–1`, and hence its total running time is

$A+B$(`n–0`) + $A+B$(`n–1`) + $A+B$(`n–2`) + ... + $A+B$(`n–(n–1)`))
= $An+B$(1+2+...+`n`)
= $An+B$(`n`(`n`+1)/2) (using the formula 1+2+ .... +`n` = `n`(`n`+1)/2)

We now have to add in the cost of the initialisation `i=0`, the final evaluation of the boolean expression in the outer loop, and the invocation and return costs. These are independent of `n`, and so let them total constant D, say. Hence the final cost is $D+An+B$(`n`(`n`+1)/2). This is equivalent to $D+(A+B/2)n+Bn^2/2$ which we can re-write as $D+En+Fn^2$ for some constants E, and F.

The accompanying graph shows empirically determined running times for Machine A. The graph agrees with the theoretical analysis because the graph of any function of the form $D+En+Fn^2$ has a shape like the side of a bowl for $n \geq 0$. We have already noted that a time complexity of $D+En+Fn^2$ is costly because it grows so fast as `n` grows, and the graph bears this out. Two minutes to sort an array of 100,000 elements is unacceptably slow. In contrast, the time complexity of *Quicksort* is $A+Bn+Cn\log_2 n$ which is much smaller than $D+En+Fn^2$ (see the panel on logs). Indeed, repeating the experiment summarised in the graph but using *Quicksort* in place of selection sort, the running time is never more than a fraction of a second, even for an array of 100,000 elements.



log_2 n is pronounced "log (base 2) of *n*", where *n* denotes any positive number. It is the number of times 2 is multiplied by itself to arrive at *n*. For example, 2^{10}=1024 and so $log_2$1024 = 10. As another example, 2^{20} = 1048576 and so $log_2$1048576 = 20. Note that the log of a number is much smaller than the number itself. Furthermore, logs grow extremely slowly. For example, 1048576 is more than a thousand times 1024, and yet its log (base 2) is only twice that of 1024. Be careful not to confuse log_2 with √, e.g.√1048576 is 1024, whereas $log_2$1048576 = 20 – logs are smaller than square roots. Logs need not be whole numbers. For example, $log_2$1024 = 10 and $log_2$512 = 9, and hence the *log* (base 2) of any number between 512 and 1024 will be greater than 9 and less than 10. For example, log_2 800 = 9.64 correct to two decimal places. Logs can be taken with respect to any base. For example, log_{10} 1000 = 3 because 10^3=1000.

6 Big-Oh notation

Time complexity analyses the running time of a method as a function of what we call the *problem size*. In array sorting, for example, the length of the array is a measure of the problem size, and in the case of computing the integer square root of *n*, the magnitude of *n* is a measure of the problem size. It is conventional to use *n* to denote the problem size, although there is no technical reason why we should do so. If two methods with the same behaviour have different

time complexities, the difference in the running times will not be significant for small problem sizes. It is mostly when the problem size is big that major differences show up. Consequently, we often simplify time complexities by assuming n is large. This enables us to drop terms whose value is insignificant for large n. This simplified measure of complexity uses a notation called *big-Oh notation* or *O-notation*.

If the time complexity function of a method is say, $E+Fn+Gn^2$, then for large values of n the cost is dominated by the n^2 term – the other terms only make small contributions which diminish as n gets larger. For example, with E, F, and G all equal to 100, and n equal to 50000, the value of $E+Fn+Gn^2$ is $100+100\times50000+100\times50000\times50000$, which reduces to 100+5M+250000M (where M stands for one million). The relative contribution of the first two terms is negligible, and the final term dominates. Indeed, n^2 in the final term contributes a factor of 2500M whereas the constant G only contributes a factor of 100. Clearly n^2 is the primary source of the cost for large n, and we say that the time complexity of $E+Fn+Gn^2$ is $O(n^2)$ – pronounced "order n squared" or "big-Oh of n squared". As another example, if the time complexity of a method is $A+Bn$ for A and B constants, the complexity is $O(n)$ because for sufficiently large values of n, the contributions of A and B are swamped by n.

If a method has time complexity $O(n^2)$ we say that its running time is *quadratic*, or that it runs *in quadratic time*, and similarly for *constant time*, *linear time*, etc. as indicated in the accompanying table. In O-notation it is usual to write $O(\log n)$ without indicating the base of

Complexity function	*O-notation*	*In words*	*How good?*	*Sample time for n=1000*
A	$O(1)$	constant	almost instantaneous	.0001 secs
$A+B(\log_2 n)$	$O(\log n)$	log	stupendously fast	.001 secs
$A+B\sqrt{n}$	$O(\sqrt{n})$	square root	very fast	.03 secs
$A+Bn$	$O(n)$	linear	fast	.1 secs
$A+B(\log_2 n)+Cn$	$O(n)$	linear	fast	.1001 secs
$A+Bn+Cn(\log_2 n)$	$O(n \log n)$	n-log-n	pretty fast	1 sec
$A+Bn+Cn^2$	$O(n^2)$	quadratic	slow for large n	1 min 40 secs
$A+Cn^2+Dn^3$	$O(n^3)$	cubic	slow for moderate n	28 hours
$A+B2^n$	$O(2^n)$	exponential	impossibly slow	millenia

the logarithm. The final column in the table gives a sample running time for n=1000 when all constants have value 100 μs. This is intended to give a rough intuitive feel for relative running times.

O-notation tells us what in practice is the crucial question about running time: *how sensitive is the running time to the size of the problem*? A method with $O(\sqrt{n})$ complexity, where n denotes the size of the problem, is not very sensitive to increases in n – if we double the size of the problem, the running time is not doubled but increased by a factor of about 1.4 only. If the method has time complexity $O(n^2)$, however, it is quite sensitive to increases in the size of the

problem – doubling the problem size, for example, will increase the running time by a factor of 4. Methods whose running times are very sensitive to increases in the size of the problem are generally not usable for large inputs. There is a famous problem called the *travelling salesman problem* which takes as input a list of *n* cities which a salesman is about to visit, and produces a minimum-distance route for the salesman (a table of distances between all cities is available). All known solutions to the travelling salesman problem have $O(2^n)$ time complexity, and 2^n grows extremely rapidly indeed. For example, if it takes a mere 1 nanosecond to carry out a basic step of the algorithm, it will require at least 2^{50} ns to solve the problem for 50 cities, and that's more than 11 days. But the time taken for 100 cities is 2^{100} ns and that's millions of years! In short, the time complexity in *O*-notation gives us sufficient information to infer that the known solutions are impossibly slow when there is more than a small number of cities to be visited.

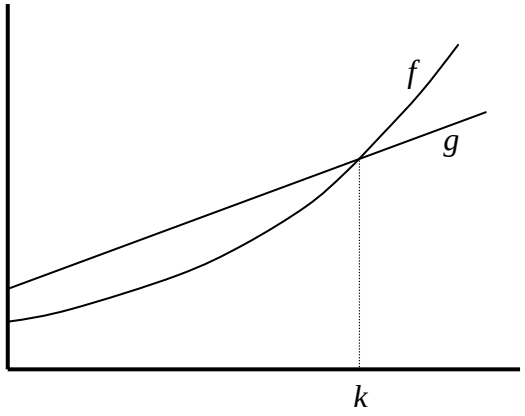
In comparing the performance of two algorithms for the same task, *O*-notation nearly always suffices. For example, suppose we are told that the time complexity of two rival algorithms *P0* and *P1* are $O(n^2)$ and $O(n^3)$, respectively, where *n* denotes the problem size. Without further information we would choose *P0* over *P1*. Would it have been any different if we had known the actual running times? Suppose the running times are $1000+100n^2$ and $5n^3$ milliseconds, respectively. Note that we have chosen the constants to favour *P1*. The times taken by *P0* and *P1* for various values of `n` are:

n	*P0*	*P1*
1	1.1 sec	.005 sec
2	1.4 sec	.04 sec
3	1.9 sec	.135 sec
20	41 sec	40 sec
100	16 min 41 sec	1 hr 23 min 20 sec
200	1 hr 6 min 41 sec	11 hr 6 min 40 sec

Observe that for small values of *n*, the differences are slight and indeed the running times are equal for *n* as low as about 20. The advantage of *P0* becomes overwhelming as *n* increases beyond 20, and it is clear that any reasonable customer would choose *P0*.

7 Mathematics of big-Oh

O-notation is a mathematical technique for summarising the behaviour of functions at large values of their arguments. It applies to all functions, not just complexity functions. To define it formally we need the notion of *eventually*. Let $f(n)$ and $g(n)$ be functions from the naturals to the reals. We say that *eventually* $f(n) \geq g(n)$ if $f(n) \geq g(n)$ for all *n* after a certain



point, as in the accompanying diagram where $f(n) \geq g(n)$ for all $n \geq k$. A function $f(n)$ is $O(n^2)$ if eventually $Cn^2 \geq f(n)$ for some constant C. Function $f(n)$ is $O(\log n)$ if eventually $C \log n \geq f(n)$ for some constant C. And so on. For example, to see that the function $2+3n+5n^2$ is $O(n^2)$ we need only observe that $7n^2 \geq 2+3n+5n^2$ for all $n \geq 3$. We choose the value 7 for constant C here, but many other choices would have been equally good, such as 43 or 165. On the other hand, the function 3^n is not $O(n^2)$ because it can be shown that for any constant C, $3^n \geq Cn^2$ for all but small values of n.

We mention two technical points. First, it follows from the formal definition that O-notation does not give a tight upper bound on the growth of a function. For example, the function $10+20n$ is $O(n)$ but it is also $O(n^2)$. In practice, however, we never say a function is $O(n^2)$ if we know it is also $O(n)$. Second, we explain why it is usual to omit the base of logs in O-notation. Actually, there is no added information in stating the base of logs in time complexity functions either. Consider a function such as $A+B\log_b n$. An elementary property of logs is $\log_b n = \log_b a \times \log_a n$, and so $A+B\log_b n$ is equivalent to $A+(B\log_b a)\log_a n$, which is equivalent to $A+C\log_b n$ for C a constant. Therefore, any function that is $O(\log_a n)$ is also $O(\log_b n)$.