From Aho, Hopcroft, + Ullman, Data Structures + Algorithms

has a value 4 in its *cursor* field; we regard this 4 as an index into the array *reclist*. The record has a true pointer in field *ptr* to another anonymous record. The record pointed to has an index in its *cursor* field indicating position 2 of *reclist*; it also has a **nil** pointer in its *ptr* field. □

1.4 The Running Time of a Program

When solving a problem we are faced frequently with a choice among algorithms. On what basis should we choose? There are two often contradictory goals.

1. We would like an algorithm that is easy to understand, code, and debug.
2. We would like an algorithm that makes efficient use of the computer's resources, especially, one that runs as fast as possible.

When we are writing a program to be used once or a few times, goal (1) is most important. The cost of the programmer's time will most likely exceed by far the cost of running the program, so the cost to optimize is the cost of writing the program. When presented with a problem whose solution is to be used many times, the cost of running the program may far exceed the cost of writing it, especially, if many of the program runs are given large amounts of input. Then it is financially sound to implement a fairly complicated algorithm, provided that the resulting program will run significantly faster than a more obvious program. Even in these situations it may be wise first to implement a simple algorithm, to determine the actual benefit to be had by writing a more complicated program. In building a complex system it is often desirable to implement a simple prototype on which measurements and simulations can be performed, before committing oneself to the final design. It follows that programmers must not only be aware of ways of making programs run fast, but must know when to apply these techniques and when not to bother.

Measuring the Running Time of a Program

The running time of a program depends on factors such as:

1. the input to the program,
2. the quality of code generated by the compiler used to create the object program,
3. the nature and speed of the instructions on the machine used to execute the program, and
4. the time complexity of the algorithm underlying the program.

The fact that running time depends on the input tells us that the running time of a program should be defined as a function of the input. Often, the running time depends not on the exact input but only on the "size" of the

made *header* point to this newly-created record. Internal to the machine, however, there is a memory address that can be used to locate the cell.

input. A good example is the process known as *sorting*, which we shall discuss in Chapter 8. In a sorting problem, we are given as input a list of items to be sorted, and we are to produce as output the same items, but smallest (or largest) first. For example, given 2, 1, 3, 1, 5, 8 as input we might wish to produce 1, 1, 2, 3, 5, 8 as output. The latter list is said to be *sorted smallest first*. The natural size measure for inputs to a sorting program is the number of items to be sorted, or in other words, the length of the input list. In general, the length of the input is an appropriate size measure, and we shall assume that measure of size unless we specifically state otherwise.

It is customary, then, to talk of $T(n)$, the running time of a program on inputs of size n. For example, some program may have a running time $T(n) = cn^2$, where c is a constant. The units of $T(n)$ will be left unspecified, but we can think of $T(n)$ as being the number of instructions executed on an idealized computer.

For many programs, the running time is really a function of the particular input, and not just of the input size. In that case we define $T(n)$ to be the *worst case* running time, that is, the maximum, over all inputs of size n, of the running time on that input. We also consider $T_{avg}(n)$, the average, over all inputs of size n, of the running time on that input. While $T_{avg}(n)$ appears a fairer measure, it is often fallacious to assume that all inputs are equally likely. In practice, the average running time is often much harder to determine than the worst-case running time, both because the analysis becomes mathematically intractable and because the notion of "average" input frequently has no obvious meaning. Thus, we shall use worst-case running time as the principal measure of time complexity, although we shall mention average-case complexity wherever we can do so meaningfully.

Now let us consider remarks (2) and (3) above: that the running time of a program depends on the compiler used to compile the program and the machine used to execute it. These facts imply that we cannot express the running time $T(n)$ in standard time units such as seconds. Rather, we can only make remarks like "the running time of such-and-such an algorithm is proportional to n^2." The constant of proportionality will remain unspecified since it depends so heavily on the compiler, the machine, and other factors.

Big-Oh and Big-Omega Notation

To talk about growth rates of functions we use what is known as *"big-oh"* notation. For example, when we say the running time $T(n)$ of some program is $O(n^2)$, read "big oh of n squared" or just "oh of n squared," we mean that there are positive constants c and n_0 such that for n equal to or greater than n_0, we have $T(n) \leq cn^2$.

Example 1.4. Suppose $T(0) = 1$, $T(1) = 4$, and in general $T(n) = (n+1)^2$. Then we see that $T(n)$ is $O(n^2)$, as we may let $n_0 = 1$ and $c = 4$. That is, for $n \geq 1$, we have $(n+1)^2 \leq 4n^2$, as the reader may prove easily. Note that we cannot let $n_0 = 0$, because $T(0) = 1$ is not less than $c0^2 = 0$ for any constant c. □

In what follows, we assume all running-time functions are defined on the nonnegative integers, and their values are always nonnegative, although not necessarily integers. We say that $T(n)$ is $O(f(n))$ if there are constants c and n_0 such that $T(n) \leq cf(n)$ whenever $n \geq n_0$. A program whose running time is $O(f(n))$ is said to have *growth rate* $f(n)$.

Example 1.5. The function $T(n) = 3n^3 + 2n^2$ is $O(n^3)$. To see this, let $n_0 = 0$ and $c = 5$. Then, the reader may show that for $n \geq 0$, $3n^3 + 2n^2 \leq 5n^3$. We could also say that this $T(n)$ is $O(n^4)$, but this would be a weaker statement than saying it is $O(n^3)$.

As another example, let us prove that the function 3^n is not $O(2^n)$. Suppose that there were constants n_0 and c such that for all $n \geq n_0$, we had $3^n \leq c2^n$. Then $c \geq (3/2)^n$ for any $n \geq n_0$. But $(3/2)^n$ gets arbitrarily large as n gets large, so no constant c can exceed $(3/2)^n$ for all n. □

When we say $T(n)$ is $O(f(n))$, we know that $f(n)$ is an upper bound on the growth rate of $T(n)$. To specify a lower bound on the growth rate of $T(n)$ we can use the notation $T(n)$ is $\Omega(g(n))$, read "big omega of $g(n)$" or just "omega of $g(n)$," to mean that there exists a positive constant c such that $T(n) \geq cg(n)$ infinitely often (for an infinite number of values of n).†

Example 1.6. To verify that the function $T(n) = n^3 + 2n^2$ is $\Omega(n^3)$, let $c = 1$. Then $T(n) \geq cn^3$ for $n = 0, 1, \ldots$.

For another example, let $T(n) = n$ for odd $n \geq 1$ and $T(n) = n^2/100$ for even $n \geq 0$. To verify that $T(n)$ is $\Omega(n^2)$, let $c = 1/100$ and consider the infinite set of n's: $n = 0, 2, 4, 6, \ldots$. □

The Tyranny of Growth Rate

We shall assume that programs can be evaluated by comparing their running-time functions, with constants of proportionality neglected. Under this assumption a program with running time $O(n^2)$ is better than one with running time $O(n^3)$, for example. Besides constant factors due to the compiler and machine, however, there is a constant factor due to the nature of the program itself. It is possible, for example, that with a particular compiler-machine combination, the first program takes $100n^2$ milliseconds, while the second takes $5n^3$ milliseconds. Might not the $5n^3$ program be better than the $100n^2$ program?

The answer to this question depends on the sizes of inputs the programs are expected to process. For inputs of size $n < 20$, the program with running time $5n^3$ will be faster than the one with running time $100n^2$. Therefore, if the program is to be run mainly on inputs of small size, we would indeed

† Note the asymmetry between big-oh and big-omega notation. The reason such asymmetry is often useful is that there are many times when an algorithm is fast on many but not all inputs. For example, there are algorithms to test whether their input is of prime length that run very fast whenever that length is even, so we could not get a good lower bound on running time that held for all $n \geq n_0$.

prefer the program whose running time was $O(n^3)$. However, as n gets large, the ratio of the running times, which is $5n^3/100n^2 = n/20$, gets arbitrarily large. Thus, as the size of the input increases, the $O(n^3)$ program will take significantly more time than the $O(n^2)$ program. If there are even a few large inputs in the mix of problems these two programs are designed to solve, we can be much better off with the program whose running time has the lower growth rate.

Another reason for at least considering programs whose growth rates are as low as possible is that the growth rate ultimately determines how big a problem we can solve on a computer. Put another way, as computers get faster, our desire to solve larger problems on them continues to increase. However, unless a program has a low growth rate such as $O(n)$ or $O(n\log n)$, a modest increase in computer speed makes very little difference in the size of the largest problem we can solve in a fixed amount of time.

Example 1.7. In Fig. 1.11 we see the running times of four programs with different time complexities, measured in seconds, for a particular compiler-machine combination. Suppose we can afford 1000 seconds, or about 17 minutes, to solve a given problem. How large a problem can we solve? In 10^3 seconds, each of the four algorithms can solve roughly the same size problem, as shown in the second column of Fig. 1.12.

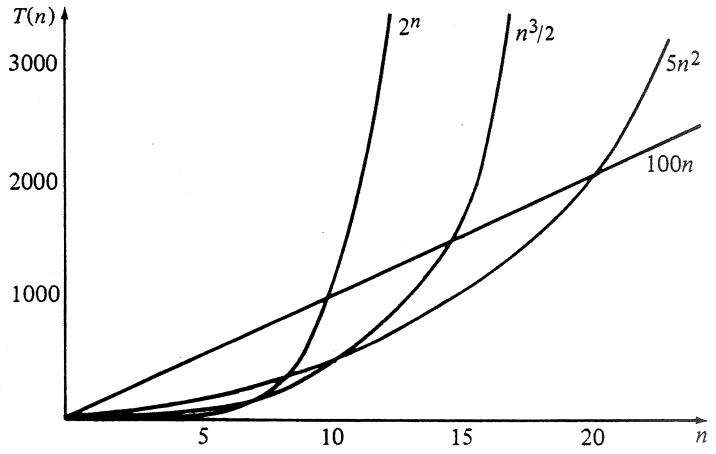

Fig. 1.11. Running times of four programs.

Suppose that we now buy a machine that runs ten times faster at no additional cost. Then for the same cost we can spend 10^4 seconds on a problem where we spent 10^3 seconds before. The maximum size problem we can now solve using each of the four programs is shown in the third column of Fig. 1.12, and the ratio of the third and second columns is shown in the fourth

column. We observe that a 1000% improvement in computer speed yields only a 30% increase in the size of problem we can solve if we use the $O(2^n)$ program. Additional factors of ten speedup in the computer yield an even smaller percentage increase in problem size. In effect, the $O(2^n)$ program can solve only small problems no matter how fast the underlying computer.

Running Time $T(n)$	Max. Problem Size for 10^3 sec.	Max. Problem Size for 10^4 sec.	Increase in Max. Problem Size
$100n$	10	100	10.0
$5n^2$	14	45	3.2
$n^3/2$	12	27	2.3
2^n	10	13	1.3

Fig. 1.12. Effect of a ten-fold speedup in computation time.

In the third column of Fig. 1.12 we see the clear superiority of the $O(n)$ program; it returns a 1000% increase in problem size for a 1000% increase in computer speed. We see that the $O(n^3)$ and $O(n^2)$ programs return, respectively, 230% and 320% increases in problem size for 1000% increases in speed. These ratios will be maintained for additional increases in speed. □

As long as the need for solving progressively larger problems exists, we are led to an almost paradoxical conclusion. As computation becomes cheaper and machines become faster, as will most surely continue to happen, our desire to solve larger and more complex problems will continue to increase. Thus, the discovery and use of efficient algorithms, those whose growth rates are low, becomes more rather than less important.

A Few Grains of Salt

We wish to re-emphasize that the growth rate of the worst case running time is not the sole, or necessarily even the most important, criterion for evaluating an algorithm or program. Let us review some conditions under which the running time of a program can be overlooked in favor of other issues.

1. If a program is to be used only a few times, then the cost of writing and debugging dominate the overall cost, so the actual running time rarely affects the total cost. In this case, choose the algorithm that is easiest to implement correctly.
2. If a program is to be run only on "small" inputs, the growth rate of the running time may be less important than the constant factor in the formula for running time. What is a "small" input depends on the exact running times of the competing algorithms. There are some algorithms, such as the integer multiplication algorithm due to Schonhage and Strassen [1971], that are asymptotically the most efficient known for their problem, but have never been used in practice even on the largest problems, because the constant of proportionality is so large in comparison to other simpler, less "efficient" algorithms.
3. A complicated but efficient algorithm may not be desirable because a person other than the writer may have to maintain the program later. It is hoped that by making the principal techniques of efficient algorithm design widely known, more complex algorithms may be used freely, but we must consider the possibility of an entire program becoming useless because no one can understand its subtle but efficient algorithms.
4. There are a few examples where efficient algorithms use too much space to be implemented without using slow secondary storage, which may more than negate the efficiency.
5. In numerical algorithms, accuracy and stability are just as important as efficiency.

1.5 Calculating the Running Time of a Program

Determining, even to within a constant factor, the running time of an arbitrary program can be a complex mathematical problem. In practice, however, determining the running time of a program to within a constant factor is usually not that difficult; a few basic principles suffice. Before presenting these principles, it is important that we learn how to add and multiply in "big oh" notation.

Suppose that $T_1(n)$ and $T_2(n)$ are the running times of two program fragments P_1 and P_2, and that $T_1(n)$ is $O(f(n))$ and $T_2(n)$ is $O(g(n))$. Then $T_1(n)+T_2(n)$, the running time of P_1 followed by P_2, is $O(\max(f(n),g(n)))$. To see why, observe that for some constants c_1, c_2, n_1, and n_2, if $n \geq n_1$ then $T_1(n) \leq c_1 f(n)$, and if $n \geq n_2$ then $T_2(n) \leq c_2 g(n)$. Let $n_0 = \max(n_1, n_2)$. If $n \geq n_0$, then $T_1(n) + T_2(n) \leq c_1 f(n) + c_2 g(n)$. From this we conclude that if $n \geq n_0$, then $T_1(n) + T_2(n) \leq (c_1 + c_2)\max(f(n), g(n))$. Therefore, the combined running time $T_1(n) + T_2(n)$ is $O(\max(f(n), g(n)))$.

Example 1.8. The *rule for sums* given above can be used to calculate the running time of a sequence of program steps, where each step may be an arbitrary program fragment with loops and branches. Suppose that we have three steps whose running times are, respectively, $O(n^2)$, $O(n^3)$ and $O(n \log n)$. Then the running time of the first two steps executed sequentially is $O(\max(n^2, n^3))$ which is $O(n^3)$. The running time of all three together is $O(\max(n^3, n \log n))$ which is $O(n^3)$. □

In general, the running time of a fixed sequence of steps is, to within a constant factor, the running time of the step with the largest running time. In rare circumstances there will be two or more steps whose running times are *incommensurate* (neither is larger than the other, nor are they equal). For example, we could have steps of running times $O(f(n))$ and $O(g(n))$, where

$$f(n) = \begin{cases} n^4 \text{ if } n \text{ is even} \\ n^2 \text{ if } n \text{ is odd} \end{cases} \qquad g(n) = \begin{cases} n^2 \text{ if } n \text{ is even} \\ n^3 \text{ if } n \text{ is odd} \end{cases}$$