The Running Time of a Program

We measure the running time of a program as a function of the size of its input. Thus, if a program runs in linear time, its running time grows as a constant times the size of the input.

The size of the input?

How do we measure the size of the input to a program? In general, it is the number of bits (or characters in a finite alphabet) to represent the input using some reasonable convention for representation. If it is not exactly that, then it is generally closely related. For example, we often measure the size of a sorting problem by the number of numbers or records to be sorted, ignoring how many bits are needed to represent the individual numbers or records themselves. This is a reasonable measure of a sorting problem if the numbers or records to be sorted each fit in one memory word, or even a few memory words, so that comparing two numbers or keys takes a constant amount of time.

This definition of size as the length of the string of bits to represent the input means that there are in general many inputs of a given size. When we talk about "the" running time of a program of a given size, which of the inputs of that size are we referring to? This ambiguity gives rise to several notions of running time: worst case (the maximum of the running times for all inputs of the given size), best case (the minimum of the running times for all inputs of the given size) or average over all inputs of a given size (which requires that we define some probability distribution over the inputs of a given size, frequently the uniform distribution.) In most of what follows, we use the "worst case" definition of running time, since, if we can give a good upper bound on the worst case, it automatically is also an upper bound on the best case and the average case running times.

Ignoring linear scale factors

Computers run faster and have more memory than they did 35 years ago (by factors of 1000 or more) and they will continue to improve. Actual running times of programs from 35 years ago are quaint relics, but more abstract analyses of algorithms from that era are still relevant. The general effect of improvements in sequential computers has been a linear scaling of running times. The potential effect of large scale hardware parallelism is much more dramatic, but is still largely unrealized.

One approach to achieve more generality is to count the number of instructions executed by a program in the course of a computation, making the assumption that different instructions will take approximately the same basic cycle time. In this approach, we could take a higher-level language program fragment, for example

```
sum := 0;
i := 1;
while i <=  n
begin
  sum := sum + i;
   i := i + 1
end
```

This program fragment leaves the sum of the first `n` positive integers in the variable `sum`. If we consider how a compiler might translate this to TC-201 assembly language, we might get the following program fragment (ignoring the allocation of memory locations for the variables.)

```
      load zero
      store sum
      load one
      store i
loop: load i
      sub  n
      skippos
      jump next
      jump done
next: load sum
      add i
      store sum
      load i
      add one
      store i
      jump loop
done:
```

For an input n, this program will execute $11n + 8$ instructions before it reaches the label `done`. According to the instruction count measure, this is the running time of this program. This has the advantage of being concrete and well defined, and is similar to measures used in theoretical analyses of algorithms.

However, since hardware has gotten very cheap and there is a constant drive for faster computer speeds, an actual hardware implementation executing a program like the one above might actually execute some of the instructions in a loop in parallel with each other. Thus, even this concrete and well defined measure for assembly language programs may sometimes be an inaccurate model in practice.

This analysis also makes certain assumptions about what the compiler might do in translating the higher-level instructions to assembly language. Depending on the compiler, there might be more or fewer instructions in the basic loop above. One could even imagine a compiler performing a "little Gauss optimization" (if the underlying machine had multiply and divide operations) using the fact that the sum of the first n positive integers is $n(n + 1)/2$ and producing the following code.

```
     load n
     add one
     store i
     multiply n
     divide two
     store sum
```

(Note that this hypothetical compiler has taken pains to make the final value of `i` correct for this piece of code.) This implementation executes 6 instructions for every n. We'll arbitrarily rule out this kind of behavior in the compiler, and content ourselves with analyzing the higher-level language fragment as follows.

The while loop is executed n times, and the body of the while contains operations that each take constant time, so the running time of the whole fragment can be expressed as a constant times n plus a constant. The particular values of the constants we will leave unspecified, for the reasons indicated above. This has the benefit that we need not specify the size of the finite alphabet used to represent the inputs (whether binary or bigger), since the lengths of the strings of symbols representing the inputs will differ by constant multiples. (For example, a binary string will be about 3.32 times as long as a decimal string.)

Big Oh, Big Omega, and Big Theta

There is a conventional notation for the running time of programs that ignores linear scale factors and lower order terms. We give (slightly) simplified definitions of these concepts.

If $f(n)$ and $g(n)$ are functions from the natural numbers to the natural numbers, we say that

$$f(n) = O(g(n))$$

pronounced "f of n is big Oh of g of n" if there exists a constant c and a natural number n_0 such that

$$f(n) \leq cg(n)$$

for all $n \geq n_0$. This means that some constant multiple of g dominates f for all but a finite number of values of n. In the typical usage, $f(n)$ expresses the running time of the program as a function of n, the input size, and $g(n)$ is simplified as much as possible.

For example, if $f(n) = 13n + 10$ and $g(n) = n$, we can say that

$$13n + 10 = O(n).$$

To justify this claim, we must exhibit numbers c and n_0 satisfying the conditions of the definition. It happens that $c = 14$ and $n_0 = 10$ suffice. That is, for all $n \geq 10$, it is the case that

$$13n + 10 \leq 14n.$$

These values work, but other values would work too. For example, it is also true that for all $n \geq 100$,

$$13n + 10 \leq 100n,$$

so choosing $c = n_0 = 100$ would also work.

Note that the meaning of $=$ in this definition is inconsistent with the properties we expect of equality. For example, we normally expect that if $A = B$ and $C = B$, then $A = C$. However, while it is true that

$$4n + 1 = O(n)$$

this does NOT allow us to conclude that

$$13n + 10 = 4n + 1?????$$

The custom is to use the "simplest" expression possible inside the O notation, omitting constant factors and lower order terms. Thus, while it is true that

$$13n + 10 = O(4n + 1)$$

we prefer the simpler form of the expression on the right, namely,

$$13n + 10 = O(n).$$

Thus, we can finally state that the running time of the while loop above is $O(n)$.

Note that an expression like $f(n) = O(g(n))$ gives an UPPER BOUND on the values of f. For example, it is also true that

$$13n + 10 = O(n^2).$$

That is, the Big-Oh notation does not require that the expression be the "best" upper bound on the function.

There is an analogous notation for LOWER BOUNDS. We say

$$f(n) = \Omega(g(n)),$$

if $g(n) = O(f(n))$, that is, if there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$. The constant c may be any positive real number.

If both $f(n) = O(g(n))$ and and $f(n) = \Omega(g(n))$, then we say that

$$f(n) = \Theta(g(n)).$$

This means that the order of growth of $f(n)$ and $g(n)$ is the same, up to constant multiples, as n becomes large. An alternative characterization is that there exist positive constants c_0, c_1 and n_0 such that

$$c_0 g(n) \leq f(n) \leq c_1 g(n),$$

for all $n \geq n_0$. For example, we have

$$13n + 10 = \Theta(n)$$

because for all $n \geq 10$ we have

$$13n \leq 13n + 10 \leq 14n.$$

Another example

Consider the function $w(n) = 4n^2 - 6n + 17$. To show that $w(n) = \Theta(n^2)$, we can proceed by showing that $w(n) = O(n^2)$ and $w(n) = \Omega(n^2)$. For the first, we'd like to find constants c and n_0 such that

$$4n^2 - 6n + 17 \leq cn^2$$

for all $n \geq n_0$. It suffices to take $c = 5$ and $n_0 = 3$, because for all $n \geq 3$,

$$17 \leq 5n^2 - 4n^2 + 6n = n^2 + 6n.$$

Thus, $w(n) = O(n^2)$. For the second, we'd like to find constants $c > 0$ and n_0 such that

$$4n^2 - 6n + 17 \geq cn^2$$

for all $n \geq n_0$. It suffices to choose $c = 3$ and $n_0 = 0$, because for all $n \geq 0$,

$$4n^2 - 3n^2 = n^2 \geq 6n - 17.$$

Thus, $w(n) = \Omega(n^2)$ and we can conclude $w(n) = \Theta(n^2)$.

Because the function $f(n) = n$ doubles when its argument n is doubled, while $g(n) = n^2$ is multiplied by 4 when its argument is doubled, and $h(n) = n^3$ is multiplied by 8 when its argument is doubled, asymptotic bounds on a function give us an idea of how it changes when its input is changed. If the function is a running time, this gives us an idea of how the running time will be affected as we increase the problem size.