Orders of Growth in Python

The previous examples illustrate that processes can differ considerably in the rates at which they consume the computational resources of space and time. For some functions, we can exactly predict the number of steps in the computational process evolved by those functions. For example, consider the function `count_factors` below that counts the number of integers that evenly divide an input `n`, by attempting to divide it by every integer less than or equal to its square root. The implementation takes advantage of the fact that if k divides n and $k < \sqrt{n}$, then there is another factor $j = n / k$ such that $j > \sqrt{n}$.

```
1   from math import sqrt
2   def count_factors(n):
3       sqrt_n = sqrt(n)
4       k, factors = 1, 0
5       while k < sqrt_n:
6           if n % k == 0:
7               factors += 2
8           k += 1
9       if k * k == n:
10          factors += 1
11      return factors
12
13  result = count_factors(576)
```

Edit code

< Back Program terminated Forward >

The total number of times this process executes the body of the `while` statement is the greatest integer less than $\sqrt{n}$. Hence, we can say that the amount of time used by this function, typically denoted $R(n)$, scales with the square root of the input, which we write as $R(n)= \sqrt{n}$.

For most functions, we cannot exactly determine the number of steps or iterations they will require. One convenient way to describe this difference is to use the notion of *order of growth* to obtain a coarse measure of the resources required by a process as the inputs become larger.

Let n be a parameter that measures the size of the problem to be solved, and let $R(n)$ be the amount of resources the process requires for a problem of size n. In our previous examples we took n to be the number for which a given function is to be computed, but there are other possibilities. For instance, if our goal is to compute an approximation to the square root of a number, we might take n to be the number of digits of accuracy required. In general there are a number of properties of the problem with respect to which it will be desirable to analyze a given process. Similarly, $R(n)$ might measure the amount of memory used, the number of elementary machine operations performed, and so on. In computers that do only a fixed number of operations at a time, the time required to evaluate an expression will be proportional to the number of elementary machine operations performed in the process of evaluation.
We say that $R(n)$ has order of growth $\Theta(f(n))$, written $R(n)=\Theta(f(n))$ (pronounced "theta of $f(n)$"), if there are positive constants k_1 and k_2 independent of n such that
$k_1 f(n) \leq R(n) \leq k_2 f(n)$

for any sufficiently large value of n. In other words, for large n, the value $R(n)$ is sandwiched between two values that both scale with $f(n)$:

- A lower bound $k_1 f(n)$ and
- An upper bound $k_2 f(n)$

For instance, the number of steps to compute $n!$ grows proportionally to the input n. Thus, the steps required for this process grows as $\Theta(n)$. We also saw that the space required for the recursive implementation `fact` grows as $\Theta(n)$. By contrast, the iterative implementation `fact_iter` takes a similar number of steps, but the space it requires stays constant. In this case, we say that the space grows as $\Theta(1)$.

The number of steps in our tree-recursive Fibonacci computation `fib` grows exponentially in its input n. In particular, one can show that the nth Fibonacci number is the closest integer to

$$\sqrt{5}\,\phi_{n-2}$$

where ϕ is the golden ratio:

$$\phi = 21 + \sqrt{5} \approx 1.6180$$

We also stated that the number of steps scales with the resulting value, and so the tree-recursive process requires $\Theta(\phi_n)$ steps, a function that grows exponentially with n.

Orders of growth provide only a crude description of the behavior of a process. For example, a process requiring n_2 steps and a process requiring $1000\,n_2$ steps and a process requiring $3\,n_2 + 10\,n + 17$ steps all have $\Theta(n_2)$ order of growth. There are certainly cases in which an order of growth analysis is too coarse a method for deciding between two possible implementations of a function.

However, order of growth provides a useful indication of how we may expect the behavior of the process to change as we change the size of the problem. For a $\Theta(n)$ (linear) process, doubling the size will roughly double the amount of resources used. For an exponential process, each increment in problem size will multiply the resource utilization by a constant factor. The next example examines an algorithm whose order of growth is logarithmic, so that doubling the problem size increases the resource requirement by only a constant amount.

Source : http://inst.eecs.berkeley.edu/~cs61A/book/chapters/interpretation.html#orders-of-growth