# **The Growth of Functions** (Rosen, 6th edition, Section 3.2) Carol Zander

When you choose an algorithm, there are many things to consider

- Amount of storage space used by its variables
- Is a network used?
- Is the data moved to and from disk?
- Running time (or complexity) of the algorithm

We will consider the last one, the running time, or complexity, of an algorithm. The efficiency of an algorithm is important. Reasons to analyze the efficiency of an algorithm include

- Analysis helps us to choose which solution to use to solve a problem
- Experimentation tells us about some test cases, but analysis tells us about all test cases in general
- Performance of the algorithm can be predicted before coding
- If you know ahead of time which parts of your application execute quickly and which parts execute slowly, then you know which parts to work on to improve the overall solution.

It is common for computer scientists to compare different algorithm techniques for solving the same problem. Benchmarks, collection of typical inputs, are used so comparisons of comparable algorithms can be used.

We are analyzing the running time without knowing the details of the input, although we often assume large data sets since running time on small data sets don't differ significantly. This is done by measuring the running time as a function of the size of the problem (for example, searching for an item in a collection of $n$ items, or sorting $n$ items). Note that the default variable used for size when discussing complexity is $n$ (although there is nothing sacred about calling it $n$, it's like the variable $i$ for an int).

Also note that different terminology means the same thing. Any of the following used mean the same thing:

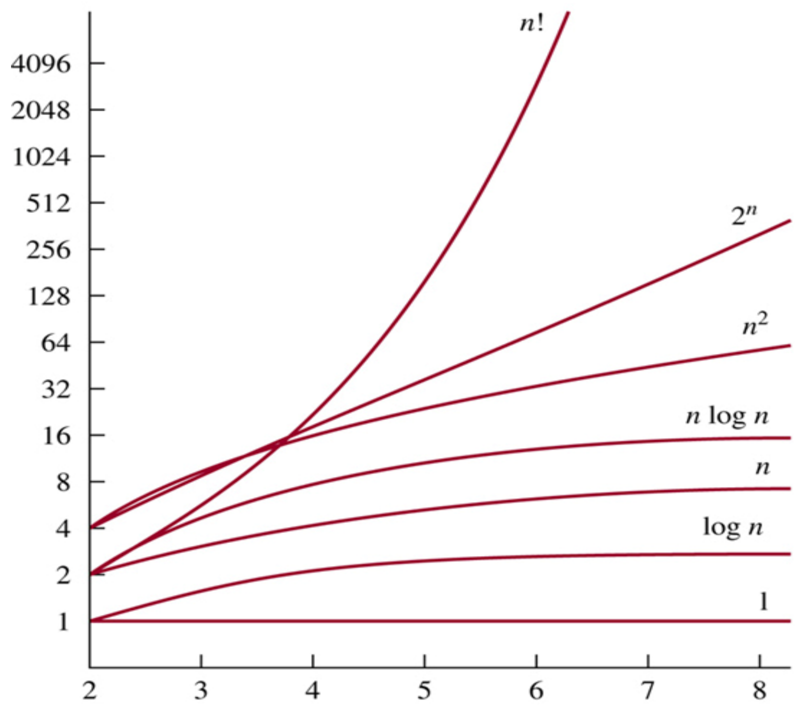
- analysis of an algorithm
- complexity (of an algorithm)
- running time (of an algorithm)
- order of an algorithm
- efficiency (of an algorithm)
- big O (of an algorithm) (note that the more correct $\Theta$ (omega) of an algorithm is rarely used)

## Definition of Running Time

Running time, $T(n)$, is a number of units of time taken by a program or algorithm on any input of size $n$. Usually $n$ is sufficiently large, for example, efficiency matters when sorting 1,000,000 items whereas when you are sorting 10 items, it is less relevant. $T(n) \approx c \cdot f(n)$ where $c$ is some positive constant . Some common running times are

| | |
|---|---|
| $T(n) = c \cdot 1$ | Constant time (does not depend on the input) |
| $T(n) = c \cdot \log n$ | Logarithmic time |
| $T(n) = c \cdot n$ | Linear time |
| $T(n) = n \cdot \log n$ | |
| $T(n) = c \cdot n^2$ | Quadratic time |
| $T(n) = c \cdot n^3$ | Cubed time |
| $T(n) = c \cdot n^b$ | Polynomial time |
| $T(n) = c \cdot b^n$ | Exponential ($b > 1$, usually $b = 2$) |
| $T(n) = c \cdot n!$ | Factorial time |

This figure from the text gives a good visualization of the different growth of functions:



Running time matters when an algorithm is chosen as can be seen in the following table for different values of n:

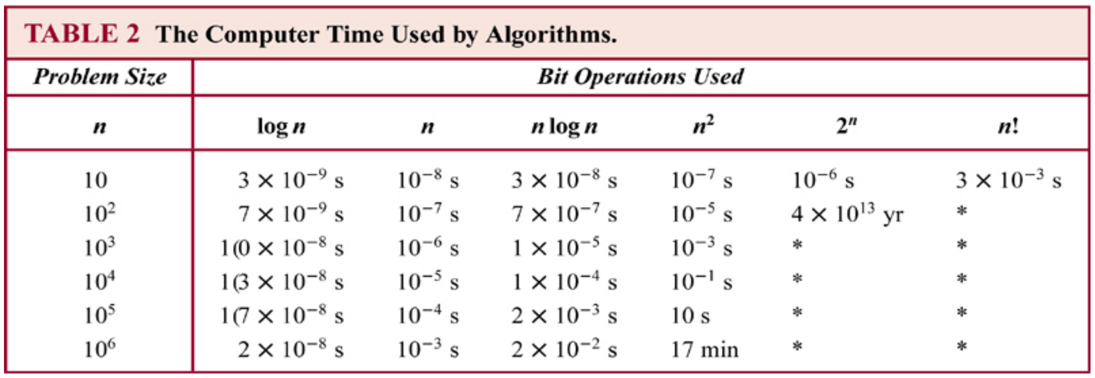


**TABLE 2 The Computer Time Used by Algorithms.**

| *Problem Size* | *Bit Operations Used* | | | | | |
|---|---|---|---|---|---|---|
| $n$ | $\log n$ | $n$ | $n \log n$ | $n^2$ | $2^n$ | $n!$ |
| 10 | $3 \times 10^{-9}$ s | $10^{-8}$ s | $3 \times 10^{-8}$ s | $10^{-7}$ s | $10^{-6}$ s | $3 \times 10^{-3}$ s |
| $10^2$ | $7 \times 10^{-9}$ s | $10^{-7}$ s | $7 \times 10^{-7}$ s | $10^{-5}$ s | $4 \times 10^{13}$ yr | * |
| $10^3$ | $1.0 \times 10^{-8}$ s | $10^{-6}$ s | $1 \times 10^{-5}$ s | $10^{-3}$ s | * | * |
| $10^4$ | $1.3 \times 10^{-8}$ s | $10^{-5}$ s | $1 \times 10^{-4}$ s | $10^{-1}$ s | * | * |
| $10^5$ | $1.7 \times 10^{-8}$ s | $10^{-4}$ s | $2 \times 10^{-3}$ s | 10 s | * | * |
| $10^6$ | $2 \times 10^{-8}$ s | $10^{-3}$ s | $2 \times 10^{-2}$ s | 17 min | * | * |

<u>Definition</u> $T(n) = O(f(n))$ if there exists two constants c, $k > 0$ such that $T(n) \le c\, f(n)$ whenever $n > k$.

Intuitively this definition means that the running time of the algorithm, T(n) is bounded by f(n) from above.

Suppose you have found that the running time of your algorithm, $T(n) = 3n^2 + 5n - 8$. <u>Prove $T(n) = O(n^2)$ .</u>

Proof: To prove $3n^2 + 5n - 8 = O(n^2)$, we must find constants c, $k > 0$ so that $3n^2 + 5n - 8 \le c \cdot n^2$ for all $n > k$ .

$$\begin{aligned} 3n^2 + 5n - 8 &\le 3n^2 + 5n && \text{(as long as } n \ge 1) \\ &\le 3n^2 + 5n^2 && \text{(as long as } n \ge 1) \\ &\le 8n^2 \end{aligned}$$

So let $c = 8$ and let $k = 1$ , then $3n^2 + 5n - 8 \le c \cdot n^2$ for all $n > k$ .

Note that the answer is not unique. There are many other ways that you could manipulate the inequality and many other values of c and k . For example, at the second step, you could instead use the fact that $5n \le n^2$ when $n > 5$. The reasoning underlying has me replace 5 with n: $5 \cdot n \le n \cdot n$ as long as n is at least 5:

$$\begin{aligned} 3n^2 + 5n - 8 &\le 3n^2 + 5n && \text{(as long as } n \ge 1) \\ &\le 3n^2 + n^2 && \text{(as long as } n \ge 5) \\ &\le 4n^2 \end{aligned}$$

In this case $c = 4$ and $k = 1$.

It is trivial to show that a function "on the order of" some function, say $n^2$, is big-O a function that is of a larger order, for example $n^3$ .

<u>Prove $n^2 = O(n^3)$.</u>

Proof: Since $n^2 \le n^3$ if $n \ge 1$, let $c = 1$ and $k = 1$ .

Generally though, we don't want this situation. We want a "tight" big-O.

<u>Prove $(n^2 + 1)/(n+1)$ is $O(n)$ .</u>

$$\begin{aligned} \text{Proof: } (n^2 + 1)/(n+1) &\le (n^2 + 2n + 1)/(n+1) && n \ge 1 \\ &\le (n+1)^2/(n+1) \\ &\le n+1 \\ &\le n+n = 2n \end{aligned}$$

So let $c = 2$ and $k = 1$ .

The definition can also be used to show that running times are not on the order of some function:

<u>Prove that $n^3 \ne O(n^2)$.</u>

Proof: Show that constants c and k do not exist so that $n^3 \le c \cdot n^2$ for $n > k$ .
If $n > 0$ (which by definition, it must be), then dividing the inequality by $n^2$ yields $n \le c$ .
Since n is a variable and c is a constant, there is no constant c that can be chosen so that n will always be less than c . Correspondingly, there is no k .

## Prove that $n^{10}$ is $O(3^n)$

Algebra was proving to be challenging. This was easier to see with experimentation. $3^n$ is an exponential function which eventually grows very fast while $n^{10}$ is a polynomial, a polynomial with a large power, but still a polynomial. The question is when does $3^n$ surpass $n^{10}$ ? I wrote a simple program comparing $3^n$ to $n^{10}$. Here is the output showing when $3^n$ surpasses $n^{10}$. You can see once $3^n$ starts growing, passing $n^{10}$, it grows big very fast.
(Output is from using an 8-byte long int.)

```
n = 30   n^10 =      590490000000000
         3^n  =      205891132094649

n = 31   n^10 =      819628286980801
         3^n  =      617673396283947

n = 32   n^10 =     1125899906842624
         3^n  =     1853020188851841

n = 33   n^10 =     1531578985264449
         3^n  =     5559060566555523

n = 34   n^10 =     2064377754059776
         3^n  =    16677181699666569

n = 35   n^10 =     2758547353515625
         3^n  =    50031545098999707
```

So let $k = 32$ and $c = 1$ . Then $n^{10} \le 1 \cdot 3^n$ f or all $n > 32$.

This is the way to attack it if you want c = 1. An alternative arithmetic approach is to work with k and figure out what c must be.

## Running Time in more detail

While most of the world uses big-O to mean running time, or skips words and just says, e.g., $n^2$, they don't always mean big-O, or worse case. In fact, almost always they mean average case, or they mean whatever is appropriate. And they often mean theta (or big theta) anyway, not big-O. To discuss this, first we need to define omega (or big omega).

Definition $T(n) = \Omega(f(n))$ if there exists two constants c, k > 0 such that $T(n) \ge c\, f(n)$ whenever $n > k$.

Just as big-O bounds f(n) from above, big theta bounds f(n) from below.

Definition $T(n) = \Theta(f(n))$ if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

This is usually what is meant by complexity, order of, etc. I will stick with the world's terminology, but to be somewhat more precise, I will always say "tight" big-O. This means if asked for tight big-O of some code, and big omega is $n^2$ , then a correct big-O answer of $n^3$ will be marked as incorrect.