

Growth of Functions & Asymptotic Complexity

Andrzej Wąsowski
wasowski@itu.dk

August 29, 2007

Disclaimer: These notes do not aspire to be complete, correct or authoritative. Please refer to the textbook for official presentation of the material.

Ingredients

- basics
 - theoretical performance
 - experimental performance
- Big-O
- Main example
 - How to link theory & experiments
- “about” & “proportional to”

1 Problems and Solutions

Problem — A specification of relationship between input and output in general terms (eg. sequence of letters and a sorted sequence of letters).

Instance — A concrete input for a problem, as used in examples (eg. a sequence ABASDFSAD)

Algorithm — a problem-solving method suitable for implementation of a computer program (eg. a sorting algorithm)

Program — an executable implementation of an algorithm using a concrete programming language, on a concrete platform.

Data structure — method of organizing the data involved in a computation (eg. array, list, stack).

2 Why analyze performance?

To be able to *compare* different algorithms or data structures for the same task.

To *predict* performance (running time, memory usage).

To set values of parameters (*configure* an algorithm or a data structure).

3 An algorithmic view on performance

Performance of an algorithm is measured with respect to some primary parameter n (sometimes more than one).

Average case performance (*expected running time*) — an expected, or in other words “most probable”, running time of an algorithm on a randomly chosen input.

Worst-case performance — running time of an algorithm on a perverse input (the hardest instance).

Best-case performance — running time of an algorithm on the easiest input.

The unit in which we measure running time is called *growth of functions*.

4 An engineering view on performance

latency — measure of the amount of time that elapses in performing a given operation under a given operating load.

throughput — number of operations that can be performed in a given amount of time under a given operating load.

efficiency — amount of resources that must be consumed by the program to provide acceptable latency and throughput under a given operating load.

scalability — how many additional resources must be consumed by the software to maintain acceptable latency and throughput with increasing load.

The above values are measured in concrete units (seconds, operations, etc). They are typically established by benchmarking/profiling and confronted with theoretical analysis of scalability.

5 Growth of rate of several functions

Problem session 1 (10 min) order the following functions from the one that growth slowest to the one that growth fastest. Discuss in pairs, why do you think that this is the right order.

$$f_1(n) = n \cdot n, \quad f_2(n) = n^3, \quad f_3(n) = n, \quad f_4(n) = \log n, \\ f_5(n) = 2007^2, \quad f_6(n) = 2^n, \quad f_7(n) = 1, \quad f_8(n) = 3n .$$

The following table orders most common functions by rate of growth starting with the slowest. The right most column states the scalability of a program if given function characterizes its running time.

function	name	how does running time change if input size doubles?
1	constant time	does not change [excellent scalability]
$\log n$	logarithmic time	increases by a constant [very good scalability]
n	linear time	doubles [most "fair" scalability]
$n \log n$	—	slightly more than doubles [still efficient]
n^2	quadratic time	increases fourfold [inefficient]
n^3	cubic time	increases eightfold [inefficient]
2^n	exponential time	squares [intractable]

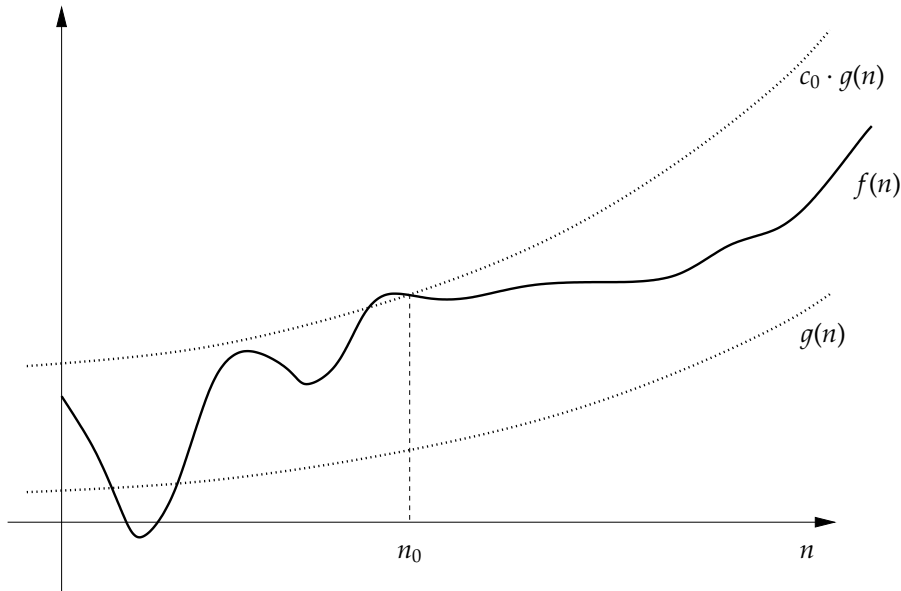
Table 1: Most important functions expressing asymptotic running time.

Figure 1: The same functions on a graph. Note that 2^x appears slower than x^3 . This is due to the fact that the former overtakes the later outside of the range of values visible on the graph.

Question. Have you ever written a program that has more than linear running time? Examples?

6 Formalizing the rate of growth

Definition 1. A function $f(n)$ is said to be $O(g(n))$ ¹ iff there exist constants c_0 and n_0 such that $f(n) < c_0g(n)$ for all $n > n_0$.



We say “ $O(f(n))$ time” instead of “ $f(n)$ time”, abstracting from initial irregularity of function f . We also ignore slowly growing terms (“noise”).

Consequently later in our analysis we can ignore parts of the program that only contribute a small amount to the total running time.

We can also classify all algorithms into relatively few classes based on their performance:

$$O(1) \subseteq O(\lg n) \subseteq O(n) \subseteq O(n \log n) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(2^n) . \quad (1)$$

¹read “big-Oh of $g(n)$ ”

The big-Oh notation behaves remarkably reasonably. Example:

$$(N + O(1)) \cdot (N + O(\log N) + O(1)) \quad (2)$$

is the same as

$$N^2 + O(N) + O(N \log N) + O(\log N) + O(N) + O(1) \quad (3)$$

which in turn is the same as

$$N^2 + O(N \log N) \quad (4)$$

and nearly the same as

$$O(N^2) \quad (5)$$

though the latter is less precise (as we lost the information that the constant in front of the fastest growing term is 1).

Remark: Most often we simplify the above complexity to $O(N^2)$. However in some special situations (like counting the nodes in a data structure, or counting number of iterations of a loop, as opposed to a general running time), we want to remember the constant in front of the fastest growing term. In these rare case the above simplification would be harmful.

Another reason shows up in the following example.

7 The Main Example

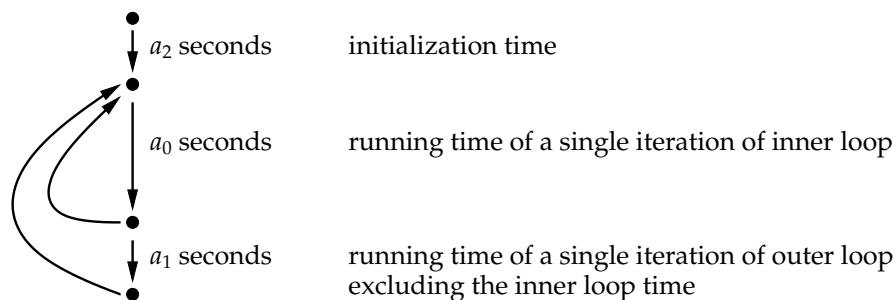
Consider a program with two nested loops:

```

1  ▷ some initialization
2  for  $i = 1$  to  $n$ 
3      do
4          while condition
5              do
6                  ...
7              ...

```

Which has the following control flow graph:



Assume that we have analyzed the program and we know that the inner loop iterates $2H_n$ times² per each iteration of outer loop, on average.

The outer loop iterates exactly n times.

Values a_0, a_1, a_2 are profiled running times for each of the three fragments of the code (running once), see figure.

The expected running time of the whole program can be estimated to be

$$2a_0nH_n + a_1n + a_2 \tag{6}$$

² H_n is the n th harmonic number; H_n is $\ln n + O(1)$

But we can also infer from that the running time of

$$2a_0nH_n + O(n) \tag{7}$$

as we know that the terms starting from linear and slower are insignificant asymptotically (remember that the first term is $O(n \ln n)$).

This analysis tells us that the values a_2 and a_1 do not affect scalability of this program and do not have to be profiled at all, if only scalability is needed.

Since $H_n = \ln n + O(1)$, the running time for big instances is well approximated by $2a_0n \ln n$, which is easy to compute.

Question: Where should the implementation be tuned for performance?

Question: What if the input size (n) growth by a factor of 2? (a scalability question)

$$\begin{aligned} \frac{2a_0(2n) \ln(2n) + O(2n)}{2a_0n \ln n + O(n)} &= \frac{2 \ln(2n) + O(1)}{\ln n + O(1)} = \dots \\ \dots &= 2 \frac{\ln(2n)}{\ln n + O(1)} + \frac{O(1)}{\ln n + O(1)} \leq 2 \frac{\ln(2n)}{\ln n} + \frac{O(1)}{\ln n} = \dots \\ \dots &= 2 \frac{\lg(2n)}{\lg n} + O\left(\frac{1}{\ln n}\right) = 2 \frac{1 + \lg n}{\lg n} + O\left(\frac{1}{\ln n}\right) \simeq 2 + O\left(\frac{1}{\ln n}\right) \tag{8} \end{aligned}$$

$\frac{1+\lg n}{\lg n}$ approaches 1 for large n , so for large n the running time of the algorithm roughly doubles when the input size doubles.

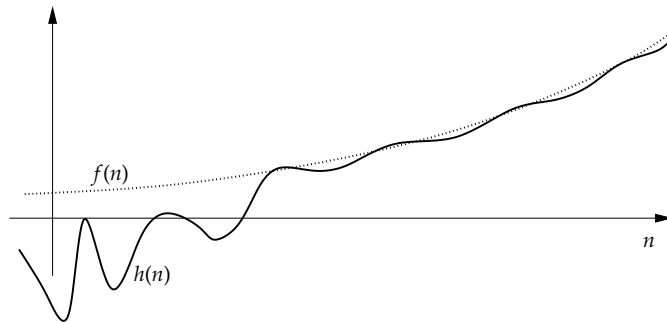
We have concluded that without even knowing a_0 ! So in order to conclude scalability no profiling is needed!

8 Proportional & About

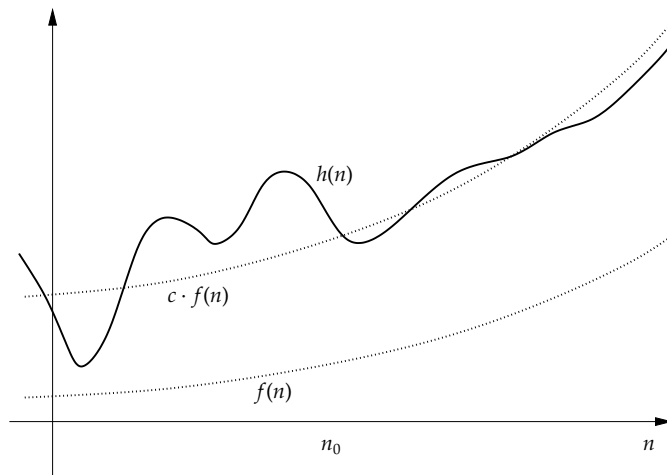
Our textbook introduces two additional notions for asymptotic complexity: “proportional to” and “about”. Both are somewhat nonstandard, but useful.

Definition 2. Assume that $f(n)$ is asymptotically larger than $g(n)$ (that is $\frac{g(n)}{f(n)} \rightarrow 0$ as $n \rightarrow \infty$). A function $h(n)$ is said to be about $f(n)$ iff $h(n)$ is $f(n) + O(g(n))$.

This is more precise than just saying that h is $(O(f(n)))$.



Definition 3. A function $h(n)$ is said to be proportional to $f(n)$ iff there exist a constant c and an asymptotically slower function $g(n)$ (in the same sense as above) such that h is $cf(n) + O(g(n))$.



The notion of *proportional-to* allows us to reason about scalability. Observe that in the analysis of scalability in our example we had to compute a ratio of two

running times, which ended up being approximately 2. If this running times were enclosed in big-Oh, then we would not know what constants their bear, and consequently we would not know what is their ratio.