Episode: Analyzing Algorithms rev. 2008/01-22 23:45

Reading: GT 1.1.0–1.1.3, pp. 5–11, GT 1.2.0–1.2.3, 13–17, 19–20, GT 1.3.0–1.3.2, pp. 21–24 [up to 90 minutes].

The most important concept of this lecture is *big-Oh*.

We will be returning to these ideas many times, so get a basic understanding and mark things that are hard to understand. After the lecture check whether something still remains unclear and search help from your teachers.

In 1.3.0–1.3.2 mark things that are completely new to you.

Ingredients: Counting operations, worst/best/average case, growth of functions, big-Oh, big-Theta, big-Omega

---

Why analyze performance?

- To allow *comparing* different algorithms or data structures
- To *predict* performance (running time, memory usage).
- To set values of parameters (*configure* an algorithm or a data structure).

Limitations of Experimental Analysis

- Can only be done for a subset of instances
- Results of experimental analysis are invalidated quickly with changes of hardware, operating systems and programming languages
- It requites that an implementation is made

Requirements for an analytical Framework

- Take into account all inputs (all instances)
- Be able to compare efficiency in a HW/OS-independent manner
- Can be used to analyze designs (as opposed to implementations)

Counting Operations

ARRAY-MAX (A :integer array)	worst case no. of ops
1 $currentMax \leftarrow A[0]$	1 assignment + 1 array access
2 **for** $i \leftarrow 1$ **to** $A.size - 1$	1 initialization + n comparisons
	$(n-1)$ incrementations
3 **do if** $currentMax < A[i]$	$(n-1)$ comparisons and array accesses
4 **then** $currentMax \leftarrow A[i]$	$(n-1)$ assignments and array accesses
5 **return** $currentMax$	1 function return

Figure 1: Computing the value of a maximum element in an array

We assume that each of the following operations takes some unspecified constant time: assignments, function calls, arithmetic operations, comparisons, array accesses, object references, function returns.

Let us count the maximum number of basic operations executed (see Figure above):

$$2 + 1 + n + 5(n-1) + 1 = 7n - 1 \tag{1}$$

Counting operations frees us from calculating tedious running times (which are also hardware/etc dependent).

Conclusion: for an input of size n ARRAY-MAX executes at most a proportional number of basic operations.

Growth Rate of Functions

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.

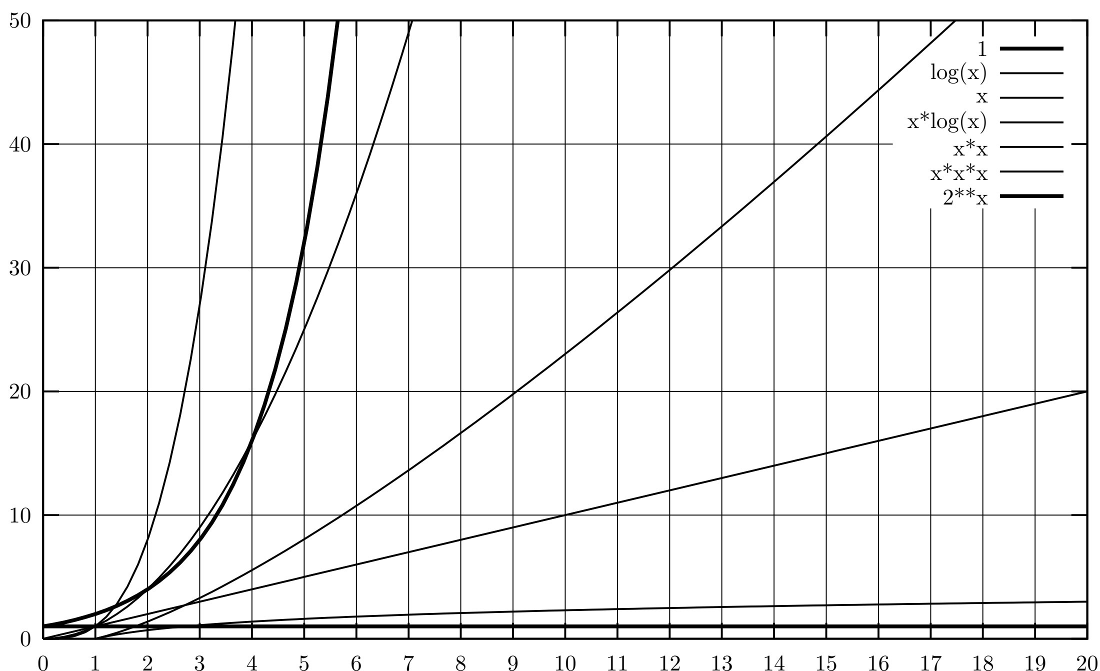

Figure 2: The same functions on a graph. Note that 2^x appears slower than x^3. This is due to the fact that the former overtakes the later outside of the range of values visible on the graph.

Big-Oh

Definition 1 (big-Oh). *A function $f(n)$ is said to be $O(g(n))$ iff there exist constants $c > 0$ and $n_0 > 1$ such that $f(n) \leq cg(n)$ for all $n > n_0$.*

Say "$O(f(n))$ time" instead of "$f(n)$ time" to abstract from initial irregularity

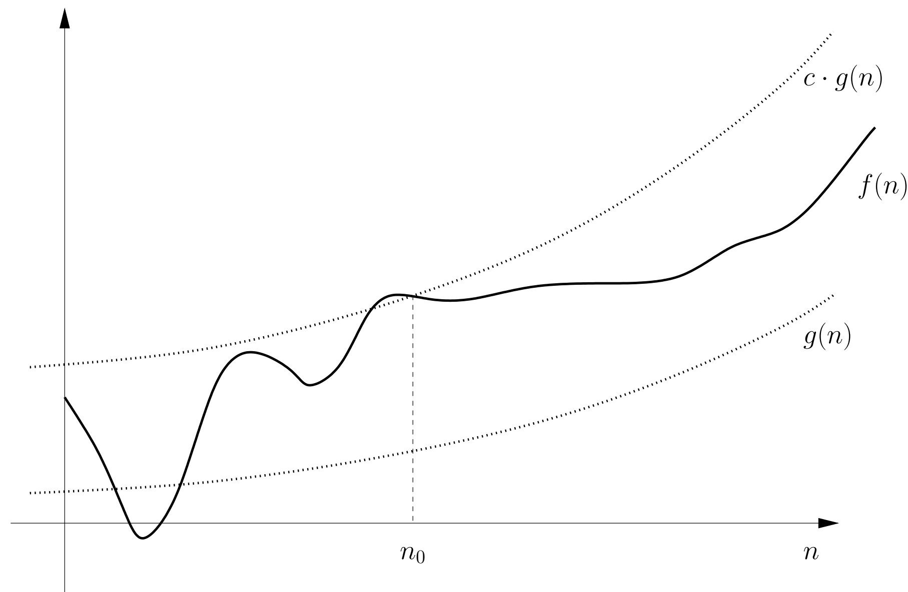

Figure 3: Illustrating Def. 1

of f and to ignore "noise".

In analysis we can ignore parts of the program that only contribute a small amount to the total running time.

The worst-case running time of ARRAY-MAX is $O(n)$.

Definition 2 (big-Omega). *A function $f(n)$ is said to be $\Omega(g(n))$ iff there exist constants $c > 0$ and $n_0 > 1$ such that $f(n) \geq cg(n)$ for all $n > n_0$ (or simply $g(n)$ is $O(f(n))$).*

Definition 3 (big-Theta). *A function $f(n)$ is said to be $\Theta(g(n))$ iff $f(n)$ is both $O(g(n)$ and $\Omega(g(n))$.*

	corresponds to
$f(n)$ is $O(g(n))$	"$f(n) \leq g(n)$"
$f(n)$ is $\Omega(g(n))$	"$f(n) \geq g(n)$"
$f(n)$ is $\Theta(g(n))$	"$f(n) = g(n)$"

Table 2: Intuitions for asymptotic notations