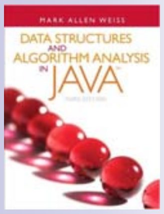
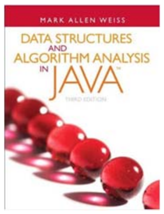


FLORIDA INTERNATIONAL UNIVERSITY

Algorithm Analysis (II)

Dr. Antonio L. Bajuelos

FIU School of Computing & Information Sciences



COP-3530 - Data Structures

Module #1: Algorithm Analysis (part II)

Outline:

- **Introduction to Asymptotic Analysis**
  - **Big-Oh (O())**
  - **Big-Omega (Ω())**
  - **Big-Theta (Θ())**
- **Examples of O(), Ω(), and Θ() calculations**
- **Typical growth rates**
- **Algorithm Complexity. Simplifying Rules**
- **Average vs Worst-case?**

Asymptotic Analysis. Some definitions

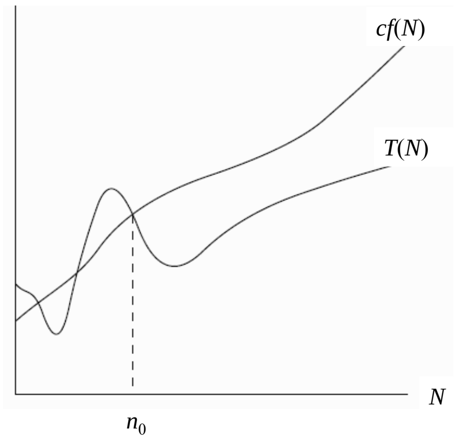
- **Definition #1 (Big-Oh)**
  - **$T(N) = O(f(N))$** if there are positive *constants* c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.
- **Definition #2 (Big-Ω)**
  - **$T(N) = \Omega(g(N))$** if there are positive *constants* c and n_0 such that $T(N) \geq cg(N)$ when $N \geq n_0$.
- **Definition #3 (Big-Θ)**
  - **$T(N) = \Theta(h(N))$** if and only if $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$.
- The idea of these definitions is to establish a relative order among functions.

3

Big-Oh (capital letter O, not a zero) notation

- **$T(N) = O(f(N))$**

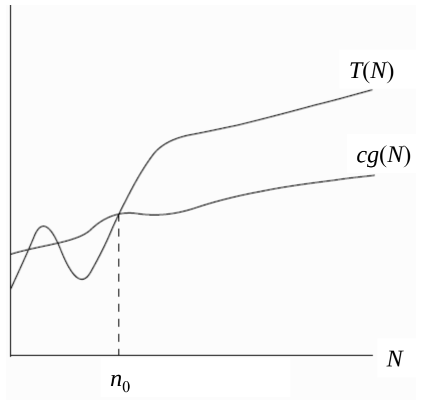
if there are positive *constants* c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.



4

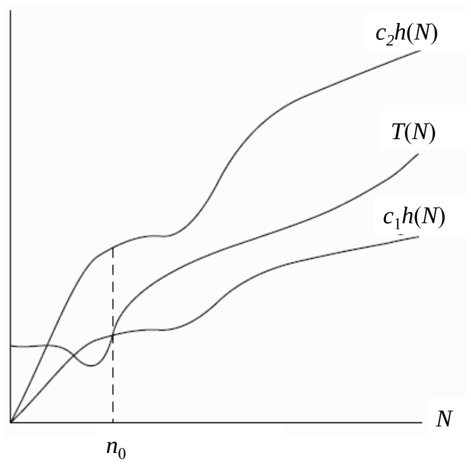
Ω (Big-Omega) notation

- ***T*(*N*) = Ω(*g*(*N*))**
  if there are positive *constants* c and n_0 such that $T(N) \geq cf(N)$ when $N \geq n_0$.



Θ (Big-Theta) notation

- ***T*(*N*) = Θ(*h*(*N*))**
  if and only if $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$.



Example of Big-Oh

- **T(*N*)=1000*N*² is O(*N*²)?**
  To do this we need to show that ∃ c, n_0 such that
  $$1000N^2 \leq cN^2 \text{ for all } N \geq n_0.$$
  It works, for example with c = 1001, n_0=1
  We also write:
  $$T(N)= 1000N^2 \text{ is } O(N^2)$$
- If $T(N) = 2N^2$, then $T(N) = O(N^4)$, $T(N) = O(N^3)$, and $T(N) = O(N^2)$ are all technically correct but…
- $T(N) = O(N^2)$ is the **best answer**. $T(N) = \Theta(N^2)$ says not only that $T(N) = O(N^2)$, but also that the result is as good (**tight**) as possible.

Exercise: Prove that T(*N*) is Ω(***N*²)**

7

Asymptotic Analysis. Important notes

- Constant n_0 is the **smallest value of n** for which the claim of an **upper bound** holds true. Usually n_0 is small, such as 1, but does not need to be.
- **Big-Oh** notation states a claim about the greatest amount of some resource (usually time) that is required by an algorithm for some input of size n.
- **Big-Ω** notation states a claim about the least amount of some resource (usually time) that is required by an algorithm for some input of size n.
- When the upper and lower bounds are the same within a constant factor, we indicate this by using Θ (**big-Theta**) notation.
- Only the **dominant terms** as n → ∞ need to be shown as the argument of "Big-Oh", "Big-Omega", and "Big-Theta".

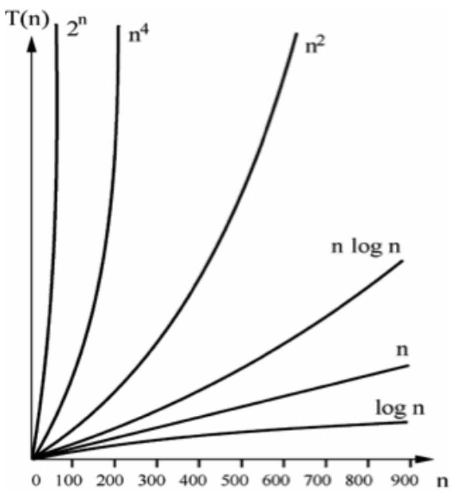
8

Typical growth rates

Function	Name
c	Constant
log*N*	Logarithmic
$\log^2 N$	Log-squared
N	Linear
N*log*N	"N log N"
N^2	Quadratic
N^3	Cubic
2^N	Exponential

9

Typical growth rates (graphic)



- Note that **$O(n^c)$** and **$O(c^n)$** are very different.
- Function **c^n** latter grows much, much faster, no matter how big the constant c is.
- A function that grows faster than any power of n is called **superpolynomial**.

10

Typical growth rates (running time)

Example: Assume we have a processor capable of one million operations per second

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

11

Algorithm Complexity. Simplifying Rules

- **Rule 1.**

  If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then

  (a) $T_1(N)+T_2(N) = O(\max(f(N), g(N)))$,

  (b) $T_1(N)*T_2(N) = O(f(N) * g(N))$

- **Rule 2.**

  If $T(N)$ is a polynomial of degree k, then $T(N) = O(N^k)$

- **Rule 3.**

  $\log^k N = O(N)$ for any constant k
  (Proof: applying k-times the L´Hospital rule)

Example:

If **$T(N) = 10\log N + 5(\log N)^3 + 7N + 3N^2 + 6N^3$** , then **$T(N)$ is $O(N^3)$.**

12

Algorithm Analysis: the Model

- To analyze algorithms, we need a **model of computation**.
- Our (**simpl**e) **model**:
  - normal computer - instructions are executed sequentially;
  - standard set of simple instructions, such as addition, multiplication, comparison, and assignment;
  - each instruction takes exactly one time unit to do anything (simple);
  - are no fancy operations, such as matrix inversion or sorting, that clearly cannot be done in one time unit;
  - we also assume infinite memory;

13

Algorithm Analysis: Average vs Worst-case?

- We are usually interested in the **worst case**:
  - What is the **maximum number of operations** that might be executed for a given problem size?
- **Example:** Insert an element into a sorted array.
  - **Algorithm:** Find the position of the new element (current element position) and move the current element and all of the elements that come after it one place to the right in the array.
  - In the **worst case**, inserting at the beginning of the array, all of the elements in the array must be moved.
  - **Worst-case**: is proportional to the number of elements in the array and so
    - Insert an element into a sorted array is **linear in the number of elements in the array**

14

Calculating the Running Time for a Program

- Simple Example: Java code fragment to calculate $\sum_{i=1}^{N} i^3$

```
public static int sum( int n )
{
  int partialSum;
  partialSum  = 0;
  for ( int i = 1; i <= n; i++ )
    partialSum  += i * i * i;
  return partialSum;
}
```

15

16