Algorithm efficiency analysis

Mădălina Răschip, Cristian Gațu

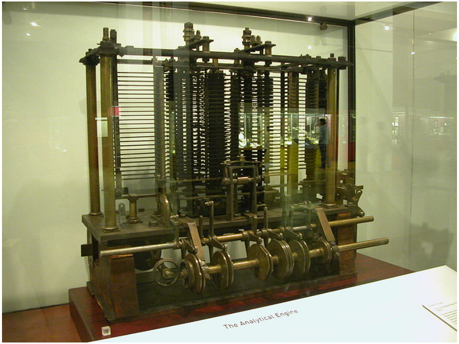
Faculty of Computer Science
"Alexandru Ioan Cuza" University of Iași, Romania

DS 2018/2019

Content

Execution time

"As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time?"
Charles Babbage (1864)



Analytic machine

Challenge

For high dimension input data, the algorithm will still solve the problem?

Knuth (1970): Scientific methods should be used in order to understand the algorithm performance.

Algorithm efficiency analysis

- Complexity analysis.
- Estimation of the **computing resources** volume required to execute the algorithm:
  - **memory space**: data storage required space;
  - **execution time**: algorithm execution time.
- **Efficient algorithm**: requires a reasonable amount of computing resources:
  - efficiency is measured with respect to the memory space or execution time;
- Utility:
  - to establish the algorithm performance and supply guarantees on this performance;
  - to compare algorithms.

Time efficiency analysis

"It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings."

— *Alan Turing, Bounding-off errors in matrix processes, 1947*

Time efficiency analysis

We need to set

I. A computing model.

von Neumann model – RAM (random access machine):

- the processing are executed sequentially;
- memory is an infinite set of cells;
- the time to access any data is the same (does not depend on the memory location);
- the memory cells store "small" values (their dimension is polynomial bounded);
- the execution time of one processing step does not depend on the operand values.

Computing model

- Involves an abstraction, a brute simplification.
- External memory:
  - a real machine has a complex memory hierarchy ;
  - it exists special algorithms designed for big datasets that a stored in the external memory;
  - fast memory – limited dimension / external memory – unlimited;
  - there are special input / output operations that transfer information between these two types.
- Parallel processing:
  - (*SIMD (Single Instruction, Multiple Data)* - parallel execution of one instruction on multiple data;
  - *multithreading* simultaneous - running multiple execution threads on the same processor;
  - multiple processors, *multicore* processors, etc.;
  - distributed systems.

II. A unit for the execution time

- Pseudo-cod (lecture 1):
  - variables and elementary data types; instructions; procedures and functions.
- Execution time of one elementary processing:
  - elementary operations: assignment, arithmetical operations, comparisons, logical operations;
  - each elementary operation requires a single time unit for its execution.
- The total execution time equals the number of executed elementary operations.

Problem size

- Assumption: the computing resources volume depends on the input data volume.
- **Problem size**: memory volume required to store the input data.
  - Can be expressed as:
    - number of input data components or
    - number of bits required to store the input data.
  - Number of bits required to store the value n is $[log_2 n] + 1$.

Problem size: examples

- Test whether a number n is prime: n (or $log_2 n$).
- Finding the minimum of an array: $x[0..n-1]$: n.
- Addition of two matrices $(m \times n)$: $m \times n$.

Content

Example 1. Sum of the first n natural numbers

Input: $n >= 1$
Output: sum $s = 1 + 2 + \cdots + n$
Problem size: n

Example 1. Sum of the first n natural numbers

Input: $n >= 1$
Output: sum $s = 1 + 2 + \cdots + n$
Problem size: n

Function *sum(n)*
begin

```
1    s ← 0
2    i ← 1
3    while i <= n do
4        s ← s + i
5        i ← i + 1
6    return s
end
```

Operation	Cost	Repetition nr.
1	c1	1
2	c2	1
3	c3	n+1
4	c4	n
5	c5	n

$$T(n) = (c3 + c4 + c5)n + (c1 + c2 + c3)$$
$$= a * n + b$$

Example 1. Sum of the first n natural numbers

It is assumed that all operations have the same **unitary cost**.

- $T(n) = 3(n+1)$;
- The constants are not important.
- Execution time depends **linearly** on the problem size.
- Equivalent algorithm:

  $s \leftarrow 0$
  for $i \leftarrow 1$ **to** n **do**
  $\quad s \leftarrow s + i$

  - management of the counter: $2(n+1)$ operations;
  - sum computation: $(n+1)$ operations (s initializing and updating).

Example 2. Multiplication of two matrices

Input: $A(m \times n), B(n \times p)$
Output: $C = A * B, \quad C_{i,j} = \sum_{k=1}^{n} A_{ik} B_{kj}, \quad i = 1, \ldots, m, j = 1, \ldots, p$
Problem size: $m \times n \times p$

Example 2. Multiplication of two matrices

Input: $A(m \times n), B(n \times p)$

Output: $C = A * B, \quad C_{i,j} = \sum_{k=1}^{n} A_{ik} B_{kj}, \quad i = 1, \ldots, m, j = 1, \ldots, p$

Problem size: $m \times n \times p$

Function *product*($a[0..m-1, 0..n-1], b[0..n-1, 0..p-1]$)
begin
    for $i \leftarrow 0$ **to** $m-1$ **do**
        for $j \leftarrow 0$ **to** $p-1$ **do**
            $c[i,j] \leftarrow 0$
            for $k \leftarrow 0$ **to** $n-1$ **do**
                $c[i,j] \leftarrow c[i,j] + a[i,k] * b[k,j]$
    return $c[0..m-1, 0..p-1]$
end

Example 2. Multiplication of two matrices

Operation	Cost	Repetition nr.
1	2(m+1)	1
2	2(p+1)	m
3	1	mp
4	2(n+1)	mp
5	2	mpn

$T(m, n, p) = 4mnp + 5mp + 4m + 2$

Example 2. Multiplication of two matrices

Operation	Cost	Repetition nr.
1	2(m+1)	1
2	2(p+1)	m
3	1	mp
4	2(n+1)	mp
5	2	mpn

$T(m, n, p) = 4mnp + 5mp + 4m + 2$

Remark: it is not necessary to fill the entire table; it is sufficient to account only the **dominant operation**.

- The most frequent (costly) operation: $a[i, k] * b[k, j]$.
- Execution time estimation: $T(m, n, p) = mnp$.

Example 3. Minimum of an array

Input: $x[0..n-1]$, $n \geq 1$
Output: $m = \min(x[0..n-1])$
Problem size: n

Example 3. Minimum of an array

Input: $x[0..n-1]$, $n \geq 1$
Output: $m = \min(x[0..n-1])$
Problem size: n

```
Function
 minimum(x[0..n − 1])
begin
1    m ← x[0]
2    i ← 1
3    while i < n do
4        if x[i] < m then
5            m ← x[i]
6        i ← i + 1
7    return m
end
```

Operation	Cost	Repetition nr.
1	1	1
2	1	1
3	1	n
4	1	n-1
5	1	t(n)
6	1	n-1

$$T(n) = 3n + t(n)$$

Example 3. Minimum of an array

Execution time depends on:

- problem size;
- input data properties.

Extreme cases have to be considered:

- most favorable case
  - $x[0] <= x[i], i = 0, \ldots, n-1 \Rightarrow t(n) = 0 \Rightarrow T(n) = 3n$
- worst-case
  - $x[0] > x[1] > \ldots > x[n-1] \Rightarrow t(n) = n-1 \Rightarrow T(n) = 4n-1$
- $3n <= T(n) <= 4n-1$
  Both the lower and the upper bound depends linearly on the problem size.
- If only the basic operation (comparison $x[i] < m$) is counted then: $T(n) = n-1$

Example 4. Sequential search

Input: $x[0..n-1]$, $n >= 1$ and v a value (search key)
Output: truth value of the statement "v belongs to $x[0..n-1]$"
Problem size: n

Example 4. Sequential search

Input: $x[0..n-1]$, $n >= 1$ and v a value (search key)
Output: truth value of the statement "v belongs to $x[0..n-1]$"
Problem size: n

Function *search*($x[0..n-1], v$)
begin
1 $i \leftarrow 0$
2 **while** $x[i] != v$ *and* $i < n-1$ **do**
3 $i \leftarrow i+1$
4 **if** $x[i] == v$ **then**
5 *found* $\leftarrow$ *true*
else
6 *found* $\leftarrow$ *false*
return *found*
end

Operation	Cost	Repetition nr.
1	1	1
2	2	t(n)+1
3	1	t(n)
4	1	1
5	1	1
6	1	1

$T(n) = 1 + 3t(n) + 4$

Example 4. Sequential search

Running time depends on:

- problem size;
- input data properties.

- Most favorable case
  - $x[0] = v \Rightarrow t(n) = 0 \Rightarrow T(n) = 5$

- worst-case
  - $x[n-1] = v$ or
    $(v! = x[0], ..., v! = x[n-1]) \Rightarrow t(n) = n - 1 \Rightarrow T(n) = 3n + 2$

- If the comparison $x[i]! = v$ is considered to be dominant:
  - most favorable case : $T(n) = 2$;
  - worst case: $T(n) = n + 2$.

Example 5. Insertion sort

Input: a sequence of numbers $(a_1, ..., a_n)$
Output: a permutation $(a_{\sigma_1}, ..., a_{\sigma_n})$ such that $a_{\sigma_1} \leq a_{\sigma_2} \leq, \ldots, \leq a_{\sigma_n}$
Problem size: n

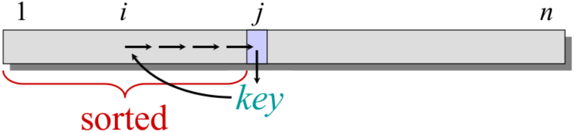
Example 5. Insertion sort

Input: a sequence of numbers $(a_1, \ldots, a_n)$

Output: a permutation $(a_{\sigma_1}, \ldots, a_{\sigma_n})$ such that $a_{\sigma_1} \leq a_{\sigma_2} \leq, \ldots, \leq a_{\sigma_n}$

Problem size: n

Procedure *insertion-sort*$(a[0..n-1], n)$
begin
    for $j \leftarrow 1$ **to** $n-1$ **do**
        $key \leftarrow a[j]$
        $i \leftarrow j-1$
        while $i >= 0$ *and* $a[i] > key$ **do**
            $a[i+1] \leftarrow a[i]$
            $i \leftarrow i-1$
        $a[i+1] \leftarrow key$
end



Insertion sort: example

8 1 4 9 2 6

Insertion sort: example

8	1	4	9	2	6
8	1	4	9	2	6

Insertion sort: example

8	1	4	9	2	6

8	1	4	9	2	6
1	8	4	9	2	6

Insertion sort: example

8	1	4	9	2	6

8	1	4	9	2	6
1	8	4	9	2	6
1	4	8	9	2	6

Insertion sort: example

8	1	4	9	2	6

8	1	4	9	2	6
1	8	4	9	2	6
1	4	8	9	2	6
1	4	8	9	2	6

Insertion sort: example

8	1	4	9	2	6
8	1	4	9	2	6
1	8	4	9	2	6
1	4	8	9	2	6
1	4	8	9	2	6
1	2	4	8	9	6

Insertion sort: example

8	1	4	9	2	6
8	1	4	9	2	6
1	8	4	9	2	6
1	4	8	9	2	6
1	4	8	9	2	6
1	2	4	8	9	6
1	2	4	6	8	9
1	2	4	6	8	9

Example 5. Insertion sort

Operation	Cost	Repetition nr.
1	c_1	n
2	c_2	$n-1$
3	c_3	$n-1$
4	c_4	$\sum_{j=2}^{n} t_j$
5	c_5	$\sum_{j=2}^{n}(t_j-1)$
6	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	c_7	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j-1) + c_6 \sum_{j=2}^{n}(t_j-1) + c_7(n-1)$$

Example 5. Insertion sort

- Running time depends on:
  - problem size;
  - input data properties.

Example 5. Insertion sort

- Running time depends on:
  - problem size;
  - input data properties.

- Most favorable case: that array is already sorted.
  $t_j = 1, \quad j = 2, \ldots, n$
  $T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$

Example 5. Insertion sort

- Running time depends on:
  - problem size;
  - input data properties.

- Most favorable case: that array is already sorted.

  $t_j = 1, \quad j = 2, \ldots, n$

  $T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$

- Worst-case: the array is sorted in the reversed order.

  $t_j = j, \quad j = 2, \ldots, n$

  $T(n) = c_1 n + (n-1)(c_2 + c_3 + c_7) + c_4(\frac{n(n+1)}{2} - 1) + c_5 \frac{n(n-1)}{2} + c_6 \frac{n(n-1)}{2}$

  $= (\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2})n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7)n - (c_2 + c_3 + c_4 + c_7)$

Example 5. Insertion sort

- Running time depends on:
  - problem size;
  - input data properties.

- Most favorable case: that array is already sorted.
  $t_j = 1, \quad j = 2, \ldots, n$
  $T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$

- Worst-case: the array is sorted in the reversed order.
  $t_j = j, \quad j = 2, \ldots, n$
  $T(n) = c_1 n + (n-1)(c_2 + c_3 + c_7) + c_4(\frac{n(n+1)}{2} - 1) + c_5 \frac{n(n-1)}{2} + c_6 \frac{n(n-1)}{2}$
  $= (\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2})n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7)n - (c_2 + c_3 + c_4 + c_7)$

- Average case: all permutations have the same outcome probability.

- Is the insertion-sort a fast algorithm?

Algorithm analysis

- **The most favorable case**
  - A lower bound for the running time;
  - Identify inefficient algorithms:
    - if an algorithm has a high cost in the most favorable case then it might be not an acceptable solution.
- **Worst-case**
  - The highest running time with respect to all possible input date;
  - An upper-bound for the running time;
  - The upper-bound is more important than the lower bound.

Average-case analysis

- There are situations where both the most favorable and the worst-case are exceptions:
  - these cases analysis does not provide enough information.
- **Average-case analysis** aims at providing information on the algorithm behavior on arbitrary input data.
  - It is based on the knowledge of the input data **probability distribution**.
  - Knowledge (estimation) of the outcome probability of each possible input instance.
  - **Average running time** is the average of running times corresponding to all input instances.

Average-case analysis

- Assumptions on the input data distribution probability:
  - input instances can be grouped in classes (the running time is the same for instances of same class);
  - there are m classes of input instances;
  - the outcome probability of an instance from k class is P_k;
  - the running time for a instance from k class is $T_k(n)$.
- The average running time:

$$T_a(n) = P_1 T_1(n) + P_2 T_2(n) + ... + P_m T_m(n)$$

- If all classes have the same outcome probability:

$$T_a(n) = (T_1(n) + T_2(n) + ... + T_m(n))/m$$

Example 4. Sequential search (re-visited)

- Assumptions:
  - probability that v belongs to the array is: p
    - v can be found with same probability on any position of the array;
    - probability that v is found on the k-th position: p/n;
  - probability that v is not found in the array: $1 - p$.

- $T_a(n) = \frac{p(1+2+..+n)}{n} + (1-p)n = \frac{p(n+1)}{2} + (1-p)n = (1 - \frac{p}{2})n + \frac{p}{2}$
  - if $p = 0.5$, then $T_a(n) = \frac{3}{4}n + \frac{1}{4}$;
  - the average time depends linearly on the input data size.

- Remark: the average time is not necessarily the arithmetic mean of the extreme case running times.

Algorithm analysis steps

1. Problem size identification.
2. Dominant operation identification.
3. Running time estimation (dominant operation running times).
4. if the running time depends on the input data properties then the following cased are analyzed:
   - most favorable case;
   - worst-case;
   - average case.
5. The complexity order (class) is established.

Algorithm efficiency analysis

- Aim: find the impact on the running time of the problem size increase.
- The detailed expression of the running time is not required.
- Is is enough to identify:
  - Running time **growth order**;
  - **Efficiency class (complexity)** of the algorithm.

Content

Growth order

- **Dominant term**: the term that becomes significantly higher when the problem size increases.
  - Dictates the algorithm behavior when the problem size increases.

Running time	Dominant term
$T1(n) = an + b$	an
$T2(n) = alogn + b$	$alogn$
$T3(n) = an^2 + bn + c$	an^2
$T4(n) = a^n + bn + c$	a^n

$(a > 1)$

Growth order

- **Growth order**: specifies the growth of the dominant term with respect to the problem size.

Growth order

- **Growth order**: specifies the growth of the dominant term with respect to the problem size.

- What is happening with the dominant term when the problem size grows k times?

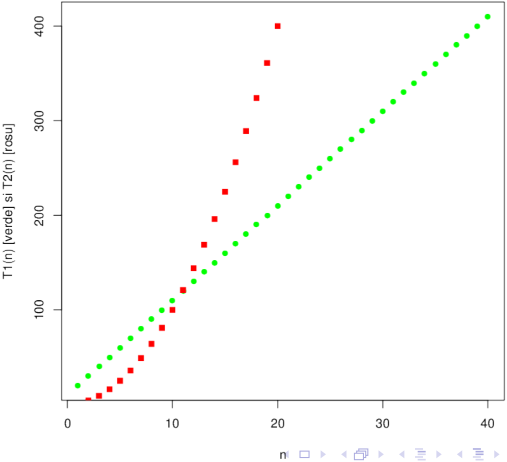
$T_1(n) = an$	$T'_1(kn) = akn$	$= kT_1(n)$	linear
$T_2(n) = alogn$	$T'_2(kn) = alog(kn)$	$= T_2(n) + alogk$	logarithmic
$T_3(n) = an^2$	$T'_3(kn) = a(kn)^2$	$= k^2 T_3(n)$	square
$T_4(n) = a^n$	$T'_4(kn) = a^{kn} = (a^n)^k$	$= T_4(n)^k$	exponential

Growth order

Allows comparison of two algorithms:

- the algorithm with lower growth order is more efficient;
- comparison is made for big values of the problem size (asymptotic case).

Example:
$T1(n) = 10n + 10$
$T2(n) = n^2$



A comparison of growth orders

Dependence of different algorithms running times with respect to problem size (assume one processor that executes 10^6 operations per second; if the running time exceeds 10^{25} years then "na" is displayed.

n	$log_2 n$	$nlog_2 n$	n^2	n^3	2^n	$n!$
10	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	4 sec
30	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$< 1 sec$	18 min	na
50	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$< 1 sec$	36 years	na
10^2	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	1 sec	10^{17} years	na
10^3	$<$ 1 sec	$<$ 1 sec	1 sec	18 min	na	na
10^4	$<$ 1 sec	$<$ 1 sec	2 min	12 days	na	na
10^5	$<$ 1 sec	2 sec	3 hours	32 years	na	na
10^6	$<$ 1 sec	20 sec	12 days	31710 years	na	na

Growth orders

In order to compare the growth orders of two running times $T1(n)$ and $T2(n)$, the $lim_{n\to\infty}\frac{T1(n)}{T2(n)}$ is computed.

- if $lim = 0$: $T1(n)$ has a smaller growth order than $T2(n)$;
- if $lim = c, c > 0$ constant: $T1(n)$ and $T2(n)$ have the same growth order;
- if $lim = \infty$: $T1(n)$ has a bigger growth order than $T2(n)$.

Content

Algorithm efficiency analysis

Computation examples

Growth order

Asymptotic notation

Asymptotic analysis

- The analysis of running times for **small** values of problem size does not allow to identify the efficient and inefficient algorithms.
- Growth order differences becomes more significant when the problem size increases.
- **Asymptotic analysis**: studies the running time properties when the problem size grows to infinity (big size problems).
  - algorithms can be classified using notations: O, Ω, Θ

Asymptotic growth order. O notation

Let $f, g : \mathbb{N} \to \mathbb{R}_+$ two functions of problem size.

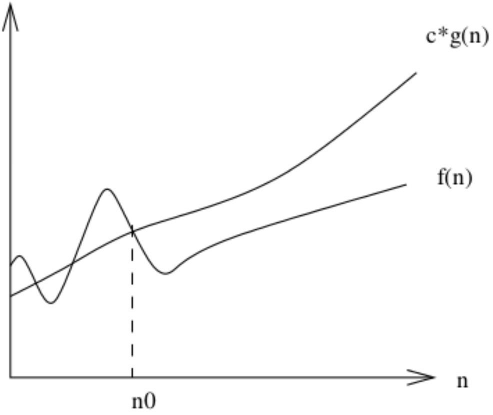
Definition

$O(g(n)) = \{f(n) : \exists c > 0, \exists n_0 \in \mathbb{N} \; a.î. \; \forall n >= n_0, 0 <= f(n) <= cg(n)\}$.

Notation: $\quad f(n) = O(g(n))$

($f(n)$ has a growth order at most equal to the one of $g(n)$.)

O notation



For big enough values of n, $f(n)$ is upper bounded by $g(n)$ multiplied by a positive constant.

O notation

Examples:

1. $T(n) = 3n + 3 \Rightarrow T(n) \in O(n)$
   $4n >= 3n + 3, c = 4, n_0 = 3, g(n) = n$

O notation

Examples:

1. $T(n) = 3n + 3 \Rightarrow T(n) \in O(n)$
   $4n >= 3n + 3, c = 4, n_0 = 3, g(n) = n$

2. $3n^2 - 100n + 6 = O(n^2)$
   $3n^2 > 3n^2 - 100n + 6$

O notation

Examples:

1. $T(n) = 3n + 3 \Rightarrow T(n) \in O(n)$
   $4n >= 3n + 3, c = 4, n_0 = 3, g(n) = n$

2. $3n^2 - 100n + 6 = O(n^2)$
   $3n^2 > 3n^2 - 100n + 6$

3. $3n^2 - 100n + 6 = O(n^3)$
   $0.01n^3 > 3n^2 - 100n + 6$

O notations - proprieties

1. $f(n) \in O(f(n))$ (reflexive).
2. $f(n) \in O(g(n)), g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$ (transitive).
3. If $T(n) = a_d n^d + a_{d-1} n^{d-1} + ... + a_1 n + a_0$ then $T(n) \in O(n^k)$ for any $k >= d$.
   - example: $n \in O(n^2)$
4. If for the worst case $T(n) <= g(n)$, then $T(n) \in O(g(n))$.
   - Sequential search: $5 \leq T(n) \leq 3n + 2 \Rightarrow$ the algorithm belongs to $O(n)$ class.

Ω notation

Definition

$\Omega(g(n)) = \{f(n)$: $\exists c > 0, n_0 \in \mathbb{N}$ $s.t.$ $\forall n >= n_0 : f(n) \geq cg(n)\}$

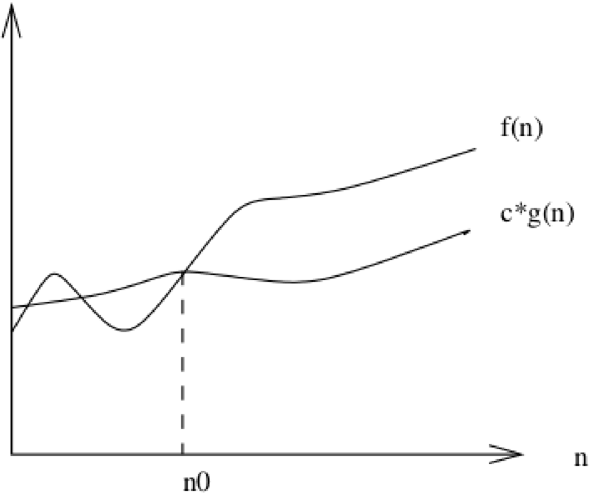
Notation: $f(n) = \Omega(g(n))$

($f(n)$ has a growth order at least as big as the one of $g(n)$.)

Examples:

1. $T(n) = 3n + 3 \Rightarrow T(n) \in \Omega(n)$
   $3n <= 3n + 3, c = 3, n_0 = 1, g(n) = n$
2. $5 <= T(n) <= 3n + 2 \Rightarrow T(n) \in \Omega(1)$
   $c = 5, n_0 = 1, g(n) = 1$

Ω notation



For big values of n, the $f(n)$ function is lower bounded by $g(n)$ multiplied by a positive constant.

Ω notation – properties

1. $f(n) \in \Omega(f(n))$ (reflexive).
2. $f(n) \in \Omega(g(n)), g(n) \in \Omega(h(n)) \Rightarrow f(n) \in \Omega(h(n))$ (transitive).
3. If $T(n) = a_d n^d + a_{d-1} n^{d-1} + ... + a_1 n + a_0$ then $T(n) \in \Omega(n^k)$ for any $k <= d$.
   - example: $n^2 \in \Omega(n)$

Θ notation

Definition

$\Theta(g(n)) =$
$\{f(n) : \exists c_1, c_2 > 0, n_0 \in \mathbb{N} \textit{ a.î. } \forall n \geq n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n)\}.$

Notation: $f(n) = \Theta(g(n))$

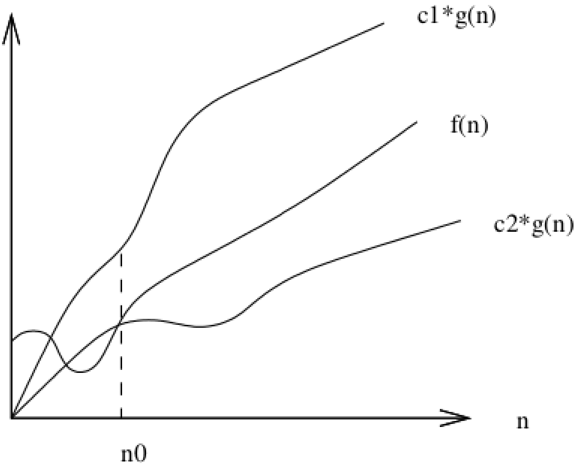
($f(n)$ has the same growth order as $g(n)$.)

Examples:

1. $T(n) = 3n + 3 \Rightarrow T(n) \in \Theta(n)$
   $c_1 = 2, c_2 = 4, n_0 = 3, g(n) = n$

2. Finding minimum of an array:
   $3n <= T(n) <= 4n - 1 \Rightarrow T(n) \in \Theta(n)$
   $c_1 = 3, c_2 = 4, n_0 = 1$

Θ notation



For big enough values of n, $f(n)$ is both lower and upper bounded by $g(n)$ multiplied by some positive constants.

Θ notations – properties

1. $f(n) \in \Theta(f(n))$ (reflexive).
2. $f(n) \in \Theta(g(n)), g(n) \in \Theta(h(n)) \Rightarrow f(n) \in \Theta(h(n))$ (transitive).
3. $f(n) \in \Theta(g(n)) \Rightarrow g(n) \in \Theta(f(n))$ (symmetric).
4. If $T(n) = a_d n^d + a_{d-1} n^{d-1} + ... + a_1 n + a_0$ then $T(n) \in \Theta(n^d)$.
5. $\Theta(cg(n)) = \Theta(g(n))$ for any constant c.
   Special cases:
   - $\Theta(c) = \Theta(1)$
   - $\Theta(log_a h(n)) = \Theta(log_b h(n))$ for any $a, b > 1$
6. $\Theta(f(n) + g(n)) = \Theta(max\{f(n), g(n)\})$

Θ notations – properties

7. $\Theta(g(n)) \subset O(g(n))$.

   Example: $f(n) = 10n \lg n + 5, \quad g(n) = n^2$
   $f(n) \leq g(n)$ for any $n \geq 36 \Rightarrow f(n) \in O(g(n))$
   But there are not such constants c and n_0 such that
   $cn^2 \leq 10nlgn + 5$ for any $n \geq n_0$.

8. $\Theta(g(n)) \subset \Omega(g(n))$.

   Example: $f(n) = 10nlgn + 5, \quad g(n) = n$
   $f(n) \geq 10g(n)$ for any $n \geq 1 \Rightarrow f(n) \in \Omega(g(n))$
   But there are not such constants c and n_0 such that $10nlgn + 5 \leq cn$ for any $n \geq n_0$.

9. $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.

Θ notation – examples

1. Multiplication of two matrices: $T(m,n,p) = 4mnp + 5mp + 4m + 2$.

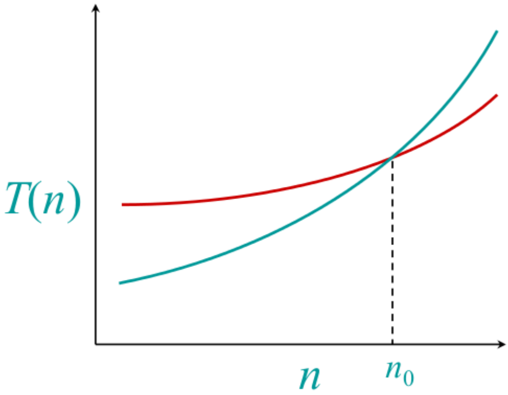
   Definition extension to the case where the problem size depends on more values:

   $f(m,n,p) \in \Theta(g(m,n,p))$ if it exists $c_1, c_2 > 0$ and $m_0, n_0, p_0 \in \mathbb{N}$ such that $c_1 g(m,n,p) \leq f(m,n,p) \leq c_2 g(m,n,p)$ for any $m \geq m_0, n \geq n_0, p \geq p_0$.

   Thus $T(m,n,p) \in \Theta(mnp)$.

2. Sequential search: $5 \leq T(n) \leq 3n + 2$.
   If $T(n) = 5$ then it does not exists c_1 such that $5 \geq c_1 n$ for big enough values of $n \Rightarrow T(n)$ does not belong to $\Theta(n)$.

Θ notation – examples



When n is large enough, an algorithm of $\Theta(n^2)$ complexity is more efficient than one of $\Theta(n^3)$ complexity.

Algorithm classification using the O notation

$$O(1) \subset O(\log n) \subset O(\log^k n) \subset O(n) \subset O(n^2) \subset \cdots \subset O(n^{k+1}) \subset O(2^n)$$

$$\begin{array}{ll}
\{A \quad | T_A(n) = O(1)\} & = \text{constant algorithm class;} \\
\{A \quad | T_A(n) = O(\log n)\} & = \text{logarithmic algorithm class;} \\
\{A \quad | T_A(n) = O(log^k n)\} & = \text{poly-logarithmic algorithm class;} \\
\{A \quad | T_A(n) = O(n)\} & = \text{linear algorithm class;;} \\
\{A \quad | T_A(n) = O(n^2)\} & = \text{quadratic algorithm class;} \\
\{A \quad | T_A(n) = O(n^{k+1})\} & = \text{polynomial algorithm class;} \\
\{A \quad | T_A(n) = O(2^n)\} & = \text{exponential algorithm class.}
\end{array}$$

$(k \geq 2)$