ALGORITHMS AND DATA STRUCTURES I

Course 4

Motivation

S. Skiena – The Algorithm Design Manual

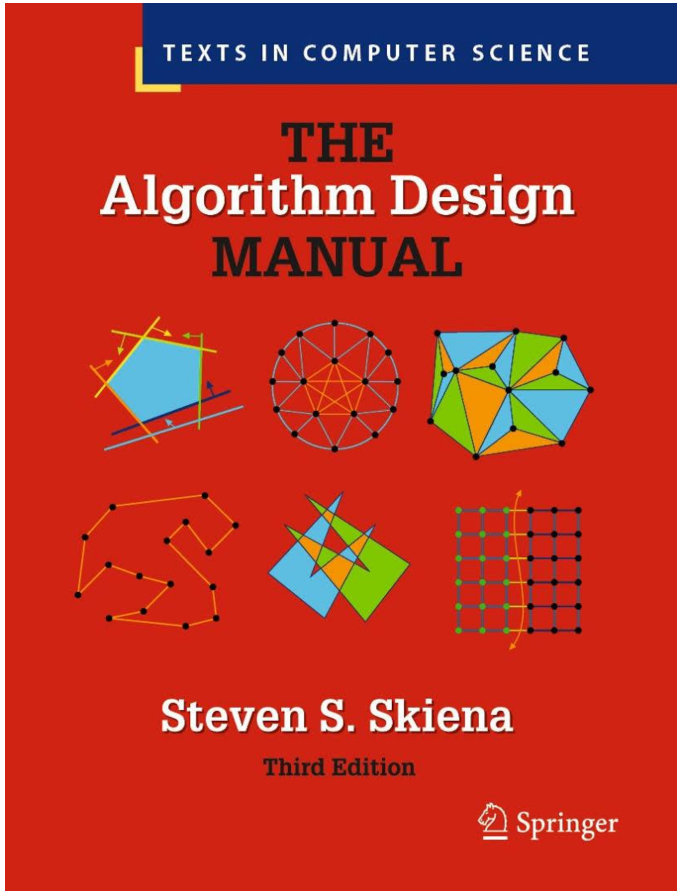
4-2. *[3]* For each of the following problems, give an algorithm that finds the desired numbers within the given amount of time. To keep your answers brief, feel free to use algorithms from the book as subroutines. For the example, $S = \{6, 13, 19, 3, 8\}$, $19 - 3$ maximizes the difference, while $8 - 6$ minimizes the difference.

(a) Let S be an *unsorted* array of n integers. Give an algorithm that finds the pair $x, y \in S$ that *maximizes* $|x - y|$. Your algorithm must run in $\boxed{O(n)}$ worst-case time.

(b) Let S be a *sorted* array of n integers. Give an algorithm that finds the pair $x, y \in S$ that *maximizes* $|x - y|$. Your algorithm must run in $\boxed{O(1)}$ worst-case time.

(c) Let S be an *unsorted* array of n integers. Give an algorithm that finds the pair $x, y \in S$ that *minimizes* $|x - y|$, for $x \neq y$. Your algorithm must run in $\boxed{O(n \log n)}$ worst-case time.

(d) Let S be a *sorted* array of n integers. Give an algorithm that finds the pair $x, y \in S$ that *minimizes* $|x - y|$, for $x \neq y$. Your algorithm must run in $\boxed{O(n)}$ worst-case time.



Previous Course

… We have seen what are the main stages of analyzing the efficiency of algorithms:

- Identifying the size of the problem
- Identification of dominant operation
- Execution time estimation (determining the number of executions of the dominant operation)

If the execution time depends on the properties of the input data, then analyze:

- Most favourable case = > lower edge of execution time
- Worst-case = > upper edge of execution time
- Average case=> average execution time

Today course

… The main purpose of analyzing the efficiency of algorithms is to determine how the execution time of the algorithm grows with increasing the problem size

… In order to obtain this information, it is not necessary to know the detailed expression of the execution time, but it is sufficient to identify:

- Order of growth in execution time (relative to the size of the problem)
- The class of efficiency (complexity) to which the algorithm belongs

Course structure

- **What is growth order?**
- What is asymptotic analysis?
- Some asymptotic notations
- Analysis of the efficiency of fundamental machining structures
- Efficiency classes
- Empirical analysis of algorithm efficiency

What is growth order?

- In the expression of execution time there is usually one term that becomes significantly larger than the other terms when the size of the problem increases.
- This term is called the dominant term and it dictates the behavior of the algorithm if the size of the problem becomes large

$T_1(n) = a \cdot n + b$ Dominant term: $a \cdot n$

$T_2(n) = a \cdot \log(n) + b$ Dominant term: $a \cdot \log(n)$

$T_3(n) = a \cdot n^2 + b \cdot n + c$ Dominant term: $a \cdot n^2$

$T_4(n) = a^n + b \cdot n + c, \ a > 1$ Dominant term: a^n

What is growth order?

- Let's consider what happens to the dominant term when the size of the problem increases k-fold:

$T_1(n) = a \cdot n$

$T_1'(k \cdot n) = a \cdot k \cdot n = k \cdot T_1(n)$ (increases as many times as the problem size increases – linear growth)

$T_2(n) = a \cdot \log(n)$

$T_2'(k \cdot n) = a \cdot \log(k \cdot n) = T_2(n) + a \cdot \log(k)$ (a constant term is added in relation to the size of the problem)

$T_3(n) = a \cdot n^2$

$T_3'(k \cdot n) = a \cdot (k \cdot n)^2 = k^2 \cdot T_3(n)$ (the growth factor is quadratic)

$T_4(n) = a^n$

$T_4'(k \cdot n) = a^{k \cdot n} = (a^n)^k = T_4(n)^k$ (the growth factor comes into play at the exponent)

What is growth order?

- Let's consider what happens to the dominant term when the size of the problem increases k-fold:

	Growth order
$T_1'(k \cdot n) = a \cdot k \cdot n = k \cdot T_1(n)$	Linear
$T_2'(k \cdot n) = a \cdot \log(k \cdot n) = T(n)_2 + a \cdot \log(k)$	Logarithmic
$T_3'(k \cdot n) = a \cdot (k \cdot n)^2 = k^2 \cdot T(n)_3$	Quadratic
$T_4'(k \cdot n) = a^{k \cdot n} = (a^n)^k = {T(n)_4}^k$	Exponential

How can the growth order be interpreted?

When comparing two algorithms, the one with the smaller growth order is considered to be more efficient

Remark: comparison is carried out for large dimensions of the size of the problem (asymptotic case)

Example. Lets consider the following two expressions of execution time

$T_1(n)=10n+10$ (linear order of growth)

$T_2(n)=n^2$ (quadratic order of growth)

If $n<=10$ then $T_1(n)>T_2(n)$

For this case, the growth order is only relevant for $n>10$

A comparison of growth orders

- Different types of dependence between execution time and problem size

n	$\log_2 n$	$n\log_2 n$	n^2	2^n	n!
10	3.3	33	100	1024	3628800
100	6.6	664	10000	10^{30}	10^{157}
1000	10	9965	1000000	10^{301}	10^{2567}
10000	13	132877	100000000	10^{3010}	10^{35659}

A comparison of growth orders

Hypothesis: each operation is executed in 10^{-9} sec

Remark: for execution times that depend exponentially or factorially on the problem size, processing becomes impossible to execute if *n > 10*

n	$\log_2 n$	$n\log_2 n$	n^2	2^n	$n!$
10 10^{-8} sec	3.3 $3*10^{-9}$ sec	33 $3*10^{-8}$ sec	100 10^{-7} sec	1024 10^{-6} sec	3628800 0.003 sec
100 10^{-7} sec	6.6 $6*10^{-9}$ sec	664 $6*10^{-7}$ sec	10000 10^{-5} sec	10^{30} 10^{13} years	10^{157} 10^{140} years
1000 10^{-6} sec	10 10^{-8} sec	9965 $9*10^{-6}$ sec	1000000 0.001 sec	10^{301} 10^{284}years	10^{2567} 10^{2550} years
10000 10^{-5} sec	13 $1.3*10^{-8}$ sec	132877 10^{-3} sec	100000000 0.1 sec	10^{3010} 10^{2993} years	10^{35659} 10^{35642} years

Comparison of growth orders

- The order of growth of two execution times $T_1(n)$ and $T_2(n)$ can be compared by calculating the limit of the ratio $T_1(n)/T_2(n)$ when n tends to infinity

- If the limit is 0 then $T_1(n)$ can be said to have an order of growth lower than $T_2(n)$

- If the limit is a strictly positive finite constant c ($c>0$), then $T_1(n)$ and $T_2(n)$ can be said to have the same order of growth

- If the limit is infinite then $T_1(n)$ can be said to have an order of growth greater than $T_2(n)$

Course structure

1. What is growth order?
2. **What is asymptotic analysis?**
3. Some asymptotic notations
4. Analysis of the efficiency of fundamental machining structures
5. Efficiency classes
6. Empirical analysis of algorithm efficiency

What is asymptotic analysis?

- Analysis of execution times for small values of the problem size does not allow differentiating between efficient and inefficient algorithms

- The differences in growth orders become increasingly significant as the problem size grows

- Asymptotic analysis aims to study the properties of execution time when the dimension of the problem tends to infinity (large-dimensional problem)

What is asymptotic analysis?

- Depending on the execution time properties when the size of the problem becomes large, the algorithm can be classified into different classes identified by standard notation

- Standard notations used to identify different efficiency classes are

  Θ (Theta)

  O (O)

  Ω (Omega)

Notation Θ

Let $f, g: \mathbb{N} \rightarrow \mathbb{R}_+$ two functions that depend on the problem size and take positive values

Definition

$f(n) \in \Theta(g(n))$ if there is $c_1, c_2 > 0$ and $n_0 \epsilon \mathbb{N}$ such that

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0$$

Remark Frequently, instead of the symbol of belonging, the symbol of equality is used:

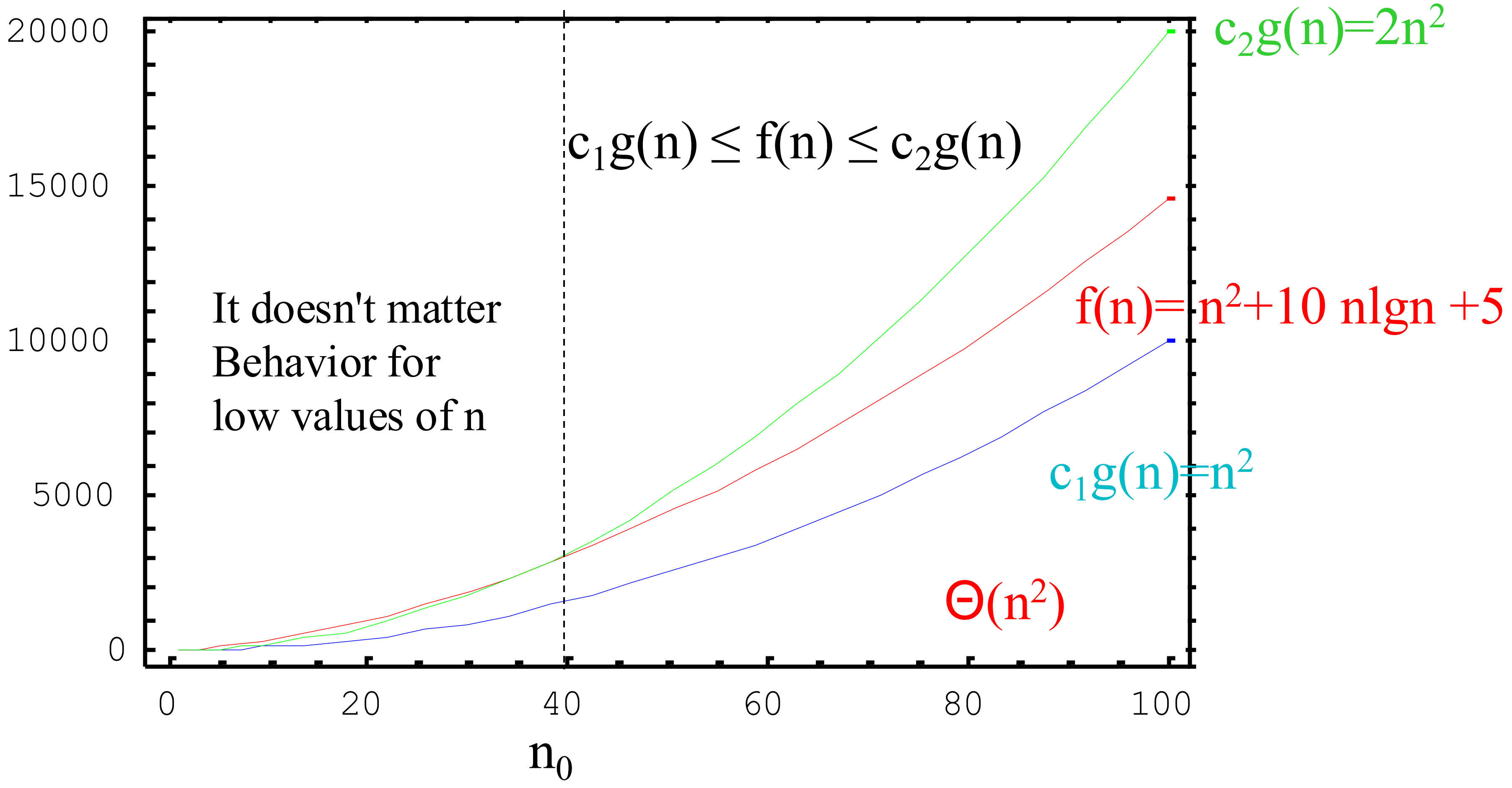
$f(n) = \Theta(g(n))$ (*f(n)* has the same order of growth as *g(n)*)

Examples

1. $T(n) = 3 \cdot n + 3 \Longrightarrow T(n) \in \Theta(n)$ (sau $T(n) = \Theta(n)$), $c_1 = 2, c_2 = 4, n_0 = 3, g(n) = n$
2. $T(n) = n^2 + 10 \cdot n \cdot \log(n) + 5 \Longrightarrow T(n) \in \Theta(n^2)$ (sau $T(n) = \Theta(n^2)$), $c_1 = 1, c_2 = 2, n_0 = 40, g(n) = n^2$

Notation Θ

Graphic illustration. For large values of *n*, *f(n)* is bounded, both upper and lower, by *g(n)* multiplied by some positive constants



Notation Θ. Properties

1. $T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \Longrightarrow \mathrm{T}(n) \in \Theta\left(n^k\right)$

2. $\Theta(c \cdot g(n)) = \Theta(g(n))$

3. $\mathrm{f}(n) \in \Theta\left(f(n)\right)$ (reflexivity)

4. $\mathrm{f}(n) \in \Theta(g(n)) \Longrightarrow \mathrm{g}(n) \in \Theta(f(n))$ (symmetry)

5. $\mathrm{f}(n) \in \Theta\left(g(n)\right), \mathrm{g}(n) \in \Theta(h(n)) \Longrightarrow \mathrm{f}(n) \in \Theta(h(n))$ (transitivity)

6. $\Theta(f(n) + g(n)) = \Theta(\max\{f(n), g(n)\})$

Notation O

Let $f, g: \mathbb{N} \to \mathbb{R}_+$ two functions that depend on the problem size and take positive values

Definition

$f(n) \in O\big(g(n)\big)$ if there is $c > 0$ and $n_0 \epsilon \mathbb{N}$ such that

$$f(n) \leq c \cdot g(n),\ \forall n \geq n_0$$

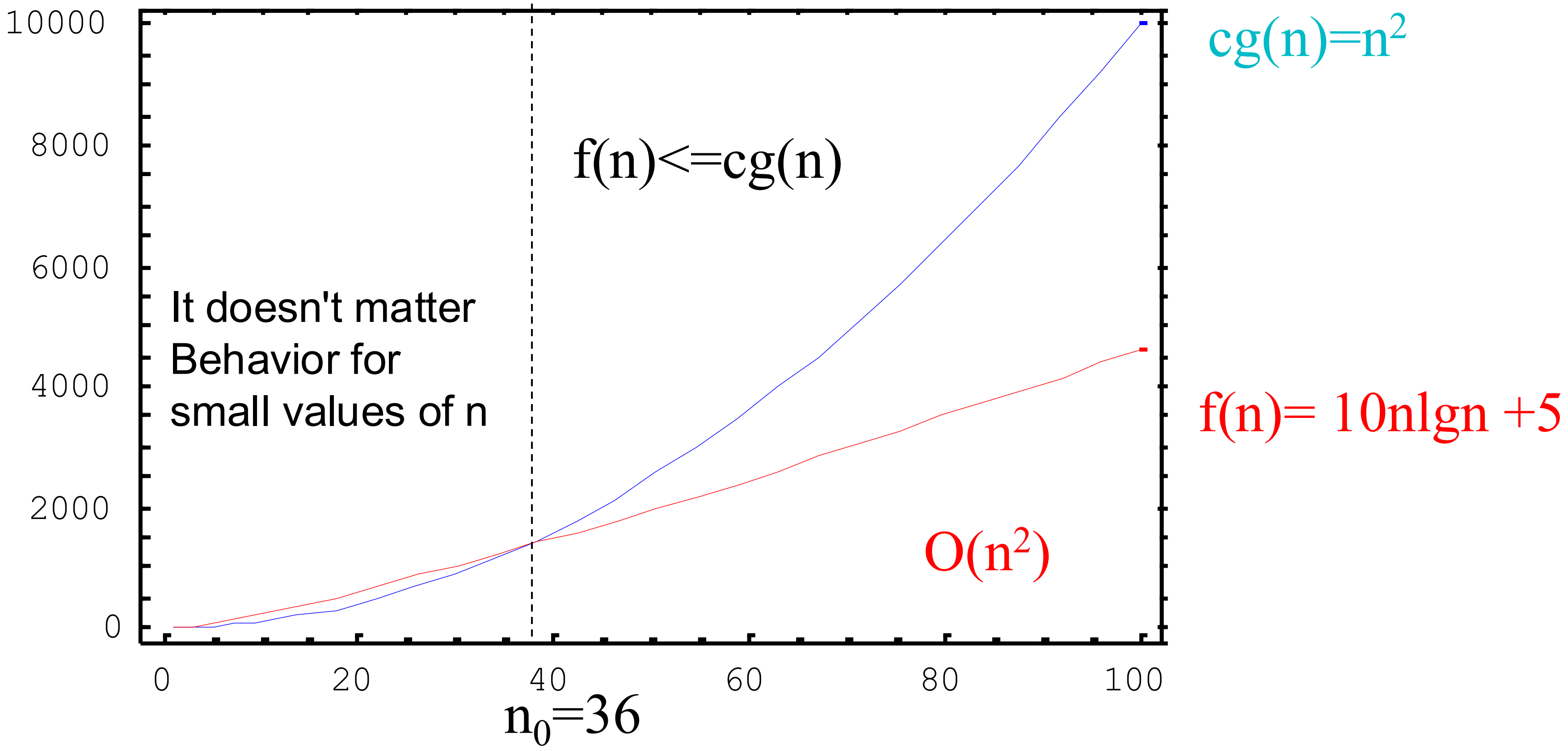
Remark f(n)=O(g(n)) (*f(n)* has an order of increment at most equal to that of *g(n)*)

Examples

1. $T(n) = 3 \cdot n + 3 \Longrightarrow T(n) \in O(n)$ (sau $T(n) = O(n)$), $c = 4, n_0 = 3, g(n) = n$
2. $6 \leq T(n) \leq 3(n+1) \Longrightarrow T(n) \in O(n)$(only the upper bound of the execution time is taken in consideration), $c = 4, n_0 = 3, g(n) = n$

Notation O

Graphic illustration. For large values of n, $f(n)$ is bounded by $g(n)$ multiplied by a positive constant



Notation O. Properties

1. $T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \Longrightarrow \mathrm{T}(n) \in O(n^d), \forall d \geq k$
2. $\mathrm{f}(n) \in O(f(n))$ (reflexivity)
3. $\mathrm{f}(n) \in O(g(n)), \mathrm{g}(n) \in O(h(n)) \Longrightarrow \mathrm{f}(n) \in O(h(n))$ (transitivity)
4. $\Theta(g(n)) \subset O(g(n))$

Remark

1. $\mathrm{n} \in O(n^2)$ the statement is mathematically correct, but in practice a tight upper bond is more appropriate
   $\mathrm{n} \in O(n \quad)$
2. $\mathrm{f(n)} = 10 \cdot n \cdot \lg(n) + 5,\ g(n) = n^2$
   $f(n) \leq g(n), \forall n \geq 36 \Longrightarrow f(n) \in O(g(n))$

   But there is no constants c and n_0 such that: $\mathrm{n} \geq n_0$ (so $\mathrm{f(n)} \notin \Theta(g(n))$)

Notation O

If by analysing the worst case is obtained:

T$(n) \leq g(n)$, then it can be assert that T$(n) \in O(g(n))$, T(n),

Example

Sequential search: $6 \leq T(n) \leq 3(n+1)$, (or $4 \leq T(n) \leq 2(n+1)$4<=T(n)<2(n+1) - depending on the algorithm variant used and the counted operations – see Course 3)

So the sequential search algorithm is of class O(n) – linear time complexity

Notation Ω

Let $f, g: \mathbb{N} \to \mathbb{R}_+$ two functions that depend on the problem size and take positive values

Definition

$f(n) \in \Omega(g(n))$ if there is $c > 0$ and $n_0 \epsilon \mathbb{N}$ such that

$$c \cdot g(n) \leq f(n),\ \forall n \geq n_0$$

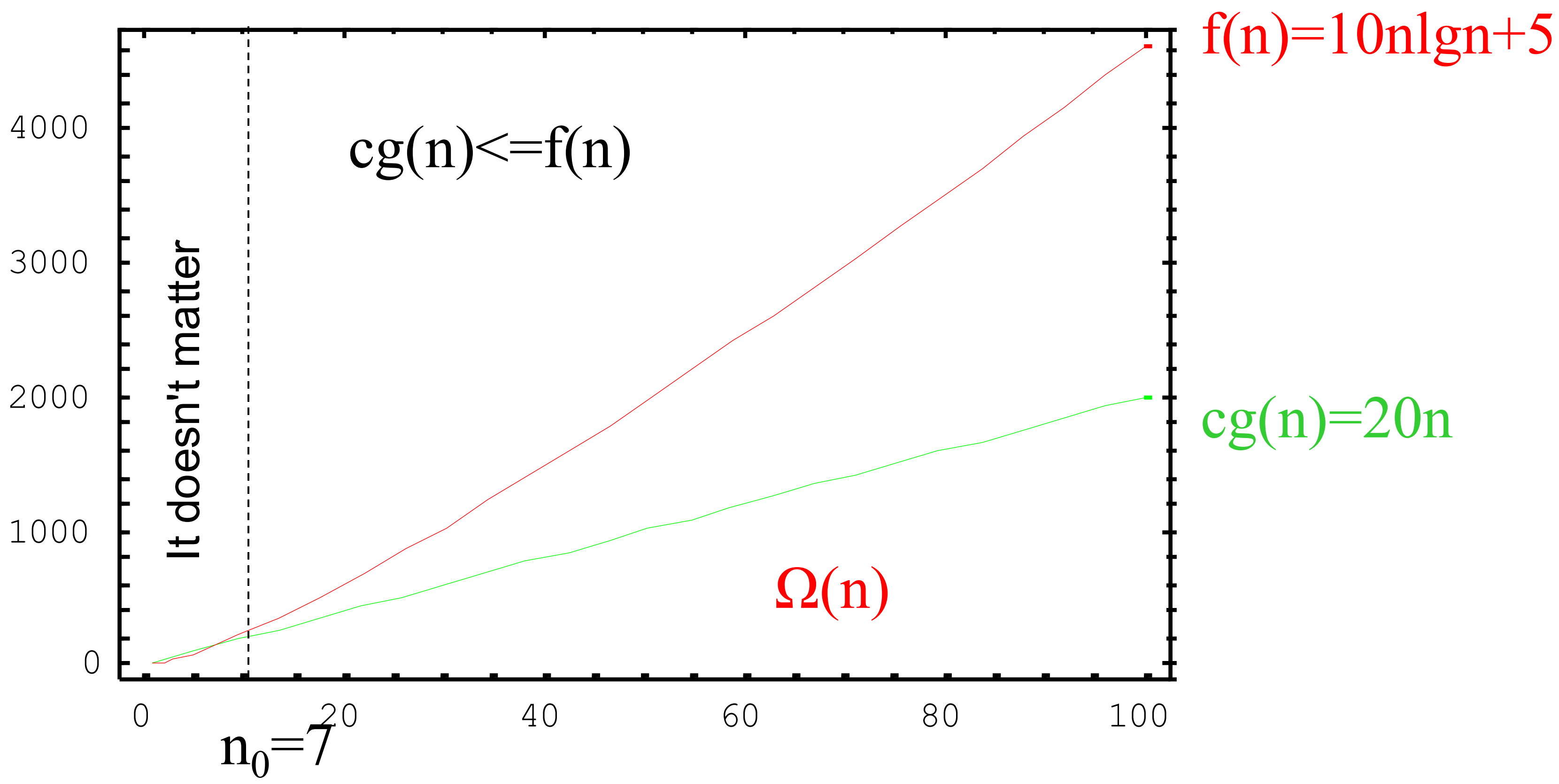
Remark f(n)= Ω(g(n)) (*f(n)* has an order of increment at least equal to that of *g(n)*)

Examples

1. $T(n) = 3 \cdot n + 3 \Longrightarrow T(n) \in \Omega(n)$ (sau $T(n) = \Omega(n)$), $c = 3, n_0 = 1, g(n) = n$
2. $6 \leq T(n) \leq 3(n+1) \Longrightarrow T(n) \in \Omega(n)$(only the lower bound of the execution time is taken in consideration), $c = 6, n_0 = 1, g(n) = 1$

Notation Ω

Graphic illustration. For large values of *n*, the function *f(n)* is bounded lower by *g(n)* possibly multiplied by a positive constant



Notation Ω. Properties

1. $T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \Longrightarrow \mathrm{T}(n) \in \Omega\left(n^d\right), \forall d \leq k$
2. $\Theta(g(n)) \in \Omega(g(n))$
3. $\Theta(g(n)) = O(g(n)) \in \Omega(g(n))$

What complexity class is appropiate?

- $4n^2 \in \Omega(1)$ true
- $4n^2 \in \Omega(n)$ true
- $4n^2 \in O(1)$ false
- $4n^2 \in O(n)$ false
- $4n^2 \in \Omega(n^2)$ true
- $4n^2 \in O(n^2)$ true
- $4n^2 \in \Omega(n^3)$ false
- $4n^2 \in \Omega(n^4)$ false
- $4n^2 \in O(n^3)$ true
- $4n^2 \in O(n^4)$ true

- $5n + 3$ is $O(n)$ true
- n is $O(5n + 3)$ true
- $5n + 3 = O(n)$ true
- $n^2 \in O(1)$ false
- $n^2 \in O(n)$ false
- $n^2 \in O(n^2)$ true
- $n^2 \in O(n^3)$ true
- $n^2 \in O(n^{100})$ true

$f(n) \in \Theta(g(n))$ if there is $c_1, c_2 > 0$ and $n_0 \epsilon \mathbb{N}$ such that

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n),\ \forall n \geq n_0$$

$f(n) \in O(g(n))$ if there is $c > 0$ and $n_0 \epsilon \mathbb{N}$ such that

$$f(n) \leq c \cdot g(n),\ \forall n \geq n_0$$

$f(n) \in \Omega(g(n))$ if there is $c > 0$ and $n_0 \epsilon \mathbb{N}$ such that

$$c \cdot g(n) \leq f(n),\ \forall n \geq n_0$$

Course structure

What is growth order? → What is asymptotic analysis? → Some asymptotic notations

→ **Analysis of the efficiency of fundamental machining structures** → Efficiency classes → Empirical analysis of algorithm efficiency

Analysis of the efficiency of fundamental structures

Sequential structure

A:

A_1	$\Theta(g_1(n))$	$O(g_1(n))$	$\Omega(g_1(n))$
A_2	$\Theta(g_2(n))$	$O(g_2(n))$	$\Omega(g_2(n))$
…	…	…	
A_k	$\Theta(g_k(n))$	$O(g_k(n))$	$\Omega(g_k(n))$

$\Theta(\max\{g_1(n), g_2(n), \ldots, g_k(n)\})$

$O(\max\{g_1(n), g_2(n), \ldots, g_k(n)\})$

$\Omega(\max\{g_1(n), g_2(n), \ldots, g_k(n)\})$

Analysis of the efficiency of fundamental structures

Conditional structure

P:

IF <conditie>

THEN P_1 $\Theta(g_1(n))$ $O(g_1(n))$ $\Omega(g_1(n))$

ELSE P_2 $\Theta(g_2(n))$ $O(g_2(n))$ $\Omega(g_2(n))$

$O(\max\{g_1(n),g_2(n)\})$

$\Omega(\min\{g_1(n),g_2(n)\})$

Analysis of the efficiency of fundamental structures

Repetitive processing

P:

FOR i←1, n DO

P1 $\Theta(1)$ ➔ $\Theta(n)$

FOR i←1,n DO

FOR j ← 1,n DO

P1 $\Theta(1)$ ➔ $\Theta(n^2)$

Remark: In the case of *k* overlapping cycles whose counter varies between 1 and n, the order of complexity is: n^k

Analysis of the efficiency of fundamental structures

Remark

If the counter limits are variable, then the number of operations performed must be explicitly calculated for each of the overlapping cycles;

Example:

m ← 1

FOR i ← 1,n DO

m ← 3*m {$m=3^i$}

FOR j ← 1,m DO

operation de cost Θ(1) {This is the dominant operation}

The order of complexity of processing is: $3+3^2+\ldots+3^n = (3^{n+1}-1)/2-1$

$\Theta(3^n)$

Course structure

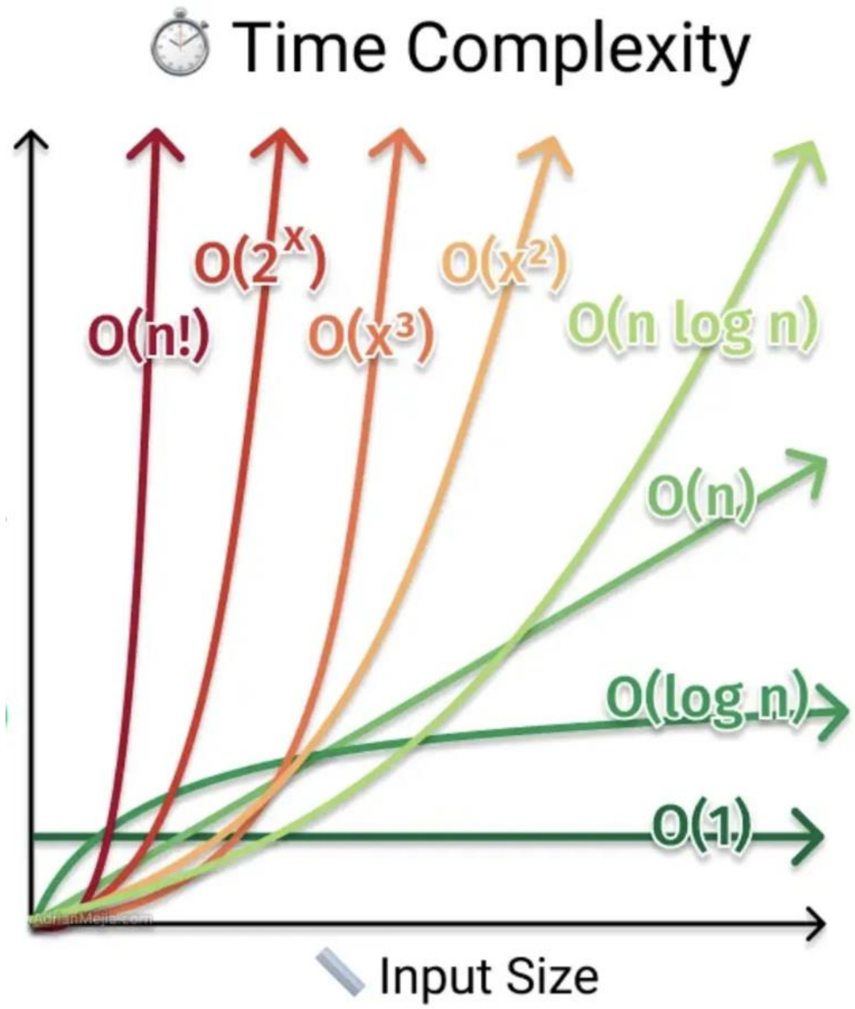
What is growth order? → What is asymptotic analysis? → Some asymptotic notations → Analysis of the efficiency of fundamental machining structures → **Efficiency classes** → Empirical analysis of algorithm efficiency

Efficiency classes

- Some of the most common classes of efficiency (complexity):

Class Name	Asymptotic notation	Algorithm
Logarithmic	$O(\lg n)$	Binary search
Linear	$O(n)$	Sequential search
Qadratic	$O(n^2)$	Insertion Sort
Cubic	$O(n^3)$	Multiplying two nxn matrices
Exponential	$O(2^n)$	Processing all subsets of a set with n elements
Factorial	$O(n!)$	Processing of all nth order permutations

Efficiency classes



Class Name	Asymptotic notation
Logarithmic	O(lgn)
Linear	O(n)
Qadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(2^n)$
Factorial	O(n!)

Example

Consider an array with n elements, x[1..n] having values of {1,...,n}. The array can have all elements distinct, or there can be a pair of elements with the same value (only one such pair). Check whether the elements of the array are all distinct or a pair of identical elements exists.

Example: n=5, x=[2, 1, 4, 1, 3] does not have all distinct elements

x=[2, 1, 4, 5, 3] has all the distinct elements

The problem is to identify an algorithm as efficient as possible in terms of execution time

Example

Variant 1

```
verify(x[1..n])
  i ←1
  d ← True
  while (d=True) and (i<n) do
    d ← NOT (search(x[i+1..n],x[i]))
    i ← i+1
  endwhile
  return d
```

Problem size: n

$1 <= T(n) <= T'(n-1)+T'(n-2)+\ldots+T'(1)$

$1 <= T(n) <= n(n-1)/2$

$T(n)$ is Ω (1), $T(n)$ is O (n^2)

```
serach(x[s..f],v)
  i ← s
  while i<f  AND x[i]!=v do
    i ← i+1
  endwhile
  if  x[i]=v then return True
            else return False
  endif
```

Subproblem size: k=f-s+1

$1 <= T'(k) <= k$

Favorable case: x[1]=x[2]

Unfavourable case: separate elements

Example

Variant 2

```
verify2(x[1..n])
  int f[1..n]   // frequences table
  f[1..n] ← 0
  for i ← 1 to n do f[x[i]] ← f[x[i]]+1  endfor
  i ← 1
  while f[i]<2 AND i<n do i ← i+1 endwhile
  if f[i]>=2  then return False
              else  return True
  endif
```

Problem size: n

n+1<= T(n)<=2n, $T(n) \in \Theta(n)$

Variant 3

```
verify3(x[1..n])
  int f[1..n]   // frequences table
  f[1..n] ← 0
  i ← 1
  while i<=n do
     f[x[i]] ← f[x[i]]+1
     if f[x[i]]>=2  then
          return False
     else i ← i+1
     endif
  endwhile
return True
```

Problem size: n

4<= T(n)<=2n

T(n) is O (n) , T(n) is Ω (1)

Example

Variant 4

Variant 2 and 3 are using an additional memory space of size n

Can this problem resolved using only an additionally space of dimension 1?

Hint: the elements in the table are distinct only if all of them are in the set {1,2,..,n} this means that their sum is n(n+1)/2

Variant 4 uses less memory that variant 3, but in medium case, variant 3 has a smaller execution time than variant 4

```
verify4(x[1..n])
  s ← 0
  for i ← 1,n do
    s ← s +x[i]
  endfor
  if s = n(n-1)/2 then return True
                  else return False
  endif
```

Problem size: n

T(n)=n

$T(n) = \Theta(n)$

More examples

Loop

```
for i ← 1,n do   //repeated n times
   operation() // cost Θ(1)
endfor
```

Iteration i: 1 2 3 4 … n

Counter: 1 1 1 1 … 1

$$T(n) = \sum_{i=1}^{n} 1 = n$$

Θ(n) O(n) Ω(n)

Nested Loop

```
for i ← 1,n do   //repeated n times
  for j ← 1,n do   //repeated n times
     operation() // cost Θ(1)
  endfor
endfor
```

Iteration i: 1 2 3 4 … n

Counter: n n n n … n

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n = n^2$$

$\Theta(n^2)$ $O(n^2)$ $\Omega(n^2)$

More examples

Neested Loop

for i ← 1,n do //repeated n times

  for j ← 1,i do //repeated i times

    operation() // cost Θ(1)

  endfor

endfor

Iteration i: 1 2 3 4 … n

Counter: 1 2 3 4 … n

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$\Theta(n^2)$ $O(n^2)$ $\Omega(n^2)$

Loop steps increse/decrese with a constant

for i ← 1,n,i+c do

  operation() // cost Θ(1)

endfor

Iteration i: 1 2 3 4 … n

Counter: 1 1+c 1+2c … 1+nc

$$T(n) = \sum_{i=1}^{\lfloor (n-1)/c \rfloor + 1} 1 = \left\lfloor \frac{n-1}{c} \right\rfloor + 1$$

Θ(n) O(n) Ω(n)

e.g. c=2 T(n)=1 + 3 + 5 +… +n=$(n+1)^2/4$

More examples

Loop

//e.g. c=10

```
for i ← 1,c*n do
   operation() // cost Θ(1)
endfor
```

Iteration: 1 2 3 4 … cn

Counter: 1 1 1 1 … 1

$$T(n) = \sum_{i=1}^{c \cdot n} 1 = c \cdot n$$

Θ(n) O(n) Ω(n)

Loop steps increse/decrese with a constant factor

```
for i ← 1,n,i*c do
   operation() // cost Θ(1)
endfor
```

```
for i ← 1,n,i/c do
 operation() //cost Θ(1)
endfor
```

Iteration: 1 2 3 4 … n

Counter: 1 c c^2 c^3…. c^k,

Counter: 1 c^{-1} c^{-2} c^{-3}…. c^{-k}

$c^k \leq n \Longrightarrow k = log_c n$

$\Theta(log_c n)$ $O(log_c n)$ $\Omega(log_c n)$

More examples

Execution time & complexity

```
function fun1(int n)
 m ← 1
 for i ← 1,n do
    m ← m + 1
 endfor
  if random() > 0.5 then return 0 endif //random function return a value in (0,1)
  for i ← 1,m*n do
    operation()  // cost Θ(1)
  endfor
endfunction
```

$n \leq T(n) \leq n \cdot m$, $\mathrm{m} = n \Longrightarrow n \leq T(n) \leq n^2$

O(n^2) Ω(n)

Execution time & complexity

```
Function fun2(int a[1..n])
  s ← 0
  for i ← 1,n do
    if a[i] < 0 then
      break
    else
      s ← s + a[i]
  endfor
  return s
```

$1 \leq T(n) \leq n,$

O(n) Ω(1)

Course structure

- What is growth order?
- What is asymptotic analysis?
- Some asymptotic notations
- Analysis of the efficiency of fundamental machining structures
- Efficiency classes
- **Empirical analysis of algorithm efficiency**

Empirical analysis of algorithm efficiency

Sometimes theoretical analysis of efficiency is difficult

In these cases, empirical analysis may be useful.

Empirical analysis can be used to:

- Formulating an initial assumption about the efficiency of the algorithm
- Comparing the efficiency of several algorithms designed to solve the same problem
- Analysis of the efficiency of an algorithm implementation (on a specific machine)
- Verifying the accuracy of an algorithm efficiency claim

General structure of empirical analysis

1. The purpose of the analysis shall be determined
2. A measure of efficiency is chosen (e.g. number of executions of operations or time required to execute processing steps)
3. The characteristics of the input data set to be used (size, value range...) are established
4. The algorithm is implemented or, if the algorithm is already implemented, the instructions necessary to perform the analysis are added (counters, functions for recording the time required for execution, etc.)
5. Input data is generated
6. Run the program for each input date and record the results
7. The results obtained shall be analysed

General structure of empirical analysis

Efficiency measure: chosen according to the purpose of the analysis

- If the goal is to identify the efficiency class, then the number of operations that are performed can be used;

- If the goal is to analyze/compare the implementation of an algorithm on a particular computing machine, then an appropriate measure would be physical time

General structure of empirical analysis

Input data set. Different categories of input data must be generated to capture the different operating cases of the algorithm

Some rules for generating input data:

- The input data must be of different sizes and with various values

- The test set should contain as arbitrary data as possible (not just exceptions)

General structure of empirical analysis

Implementation of the algorithm. As a rule, it is necessary to introduce monitoring processing

- Counter variables (if efficiency is estimated using the number of executions of operations)
- Call specific functions that return the current time (where the measure of efficiency is physical time)

Simple example in Python

(a better option would be to use a profiler – see Cprofile):

```
import time
startTime = time.time()
< ... Statements...>
stopTime = time.time()
print(" Running time (sec):" , (stopTime - startTime))
```

Next course

- Algorithm correctness
  - Preconditions
  - Postconditions
  - Algorithm invariant

Q&A