Lecture 8_2.1

Computational Complexity Analysis

Paolo PRINETTO
Politecnico di Torino (Italy)
Univ. of Illinois at Chicago, IL (USA)
CINI Cybersecurity Nat. Lab. (Italy)
Paolo.Prinetto@polito.it

www.consorzio-cini.it
www.comitato-girotondo.org

cini Cybersecurity National Lab

License Information

Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Goal

- This lecture aims at defining the terms of the complexity of algorithms

Prerequisites

- None

Further readings

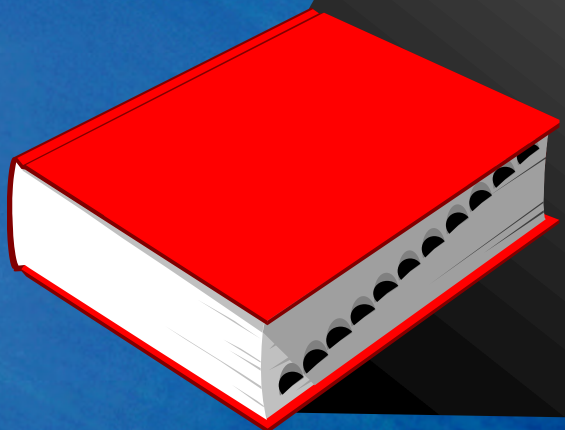
- Students interested in a deeper look at the covered topics can refer, for instance, to the books listed at the end of the lecture.
- A detailed presentation on Recurrences can be found in Lecture 8_2.2.

Outline

- Algorithm Complexity
- Computational analysis
- Asymptotic Behavior
- Notations O, Ω, Θ
- Examples of Computational analysis

Outline

- – Algorithm Complexity
- – Computational analysis
- – Asymptotic Behavior
- – Notations $\boldsymbol{O}$, $\boldsymbol{\Omega}$, $\boldsymbol{\Theta}$
- – Examples of Computational analysis



Algorithm

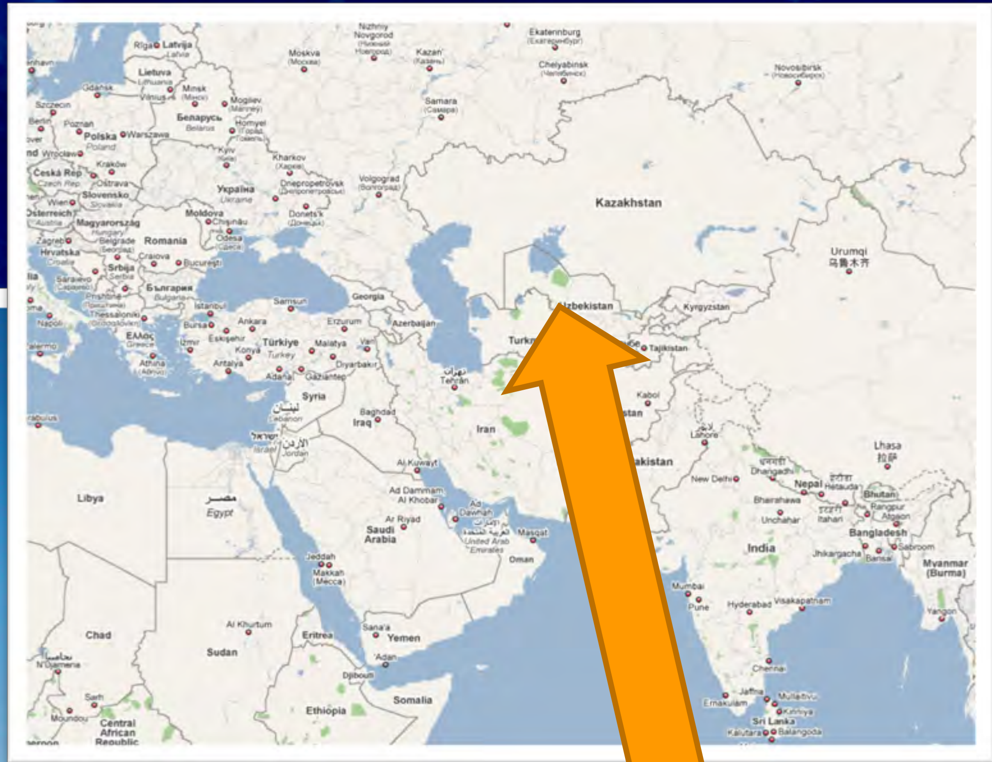
A sequence of computational steps that transform the input into the output. Each step must be “finite” in terms of required time & effort

The Arabic source,
al-K̲wārizmī
‘the man of K̲wārizm’
(now Khiva), was a name given to the mathematician Abū Jaʿfar Muhammad ibn Mūsa

Algorithm

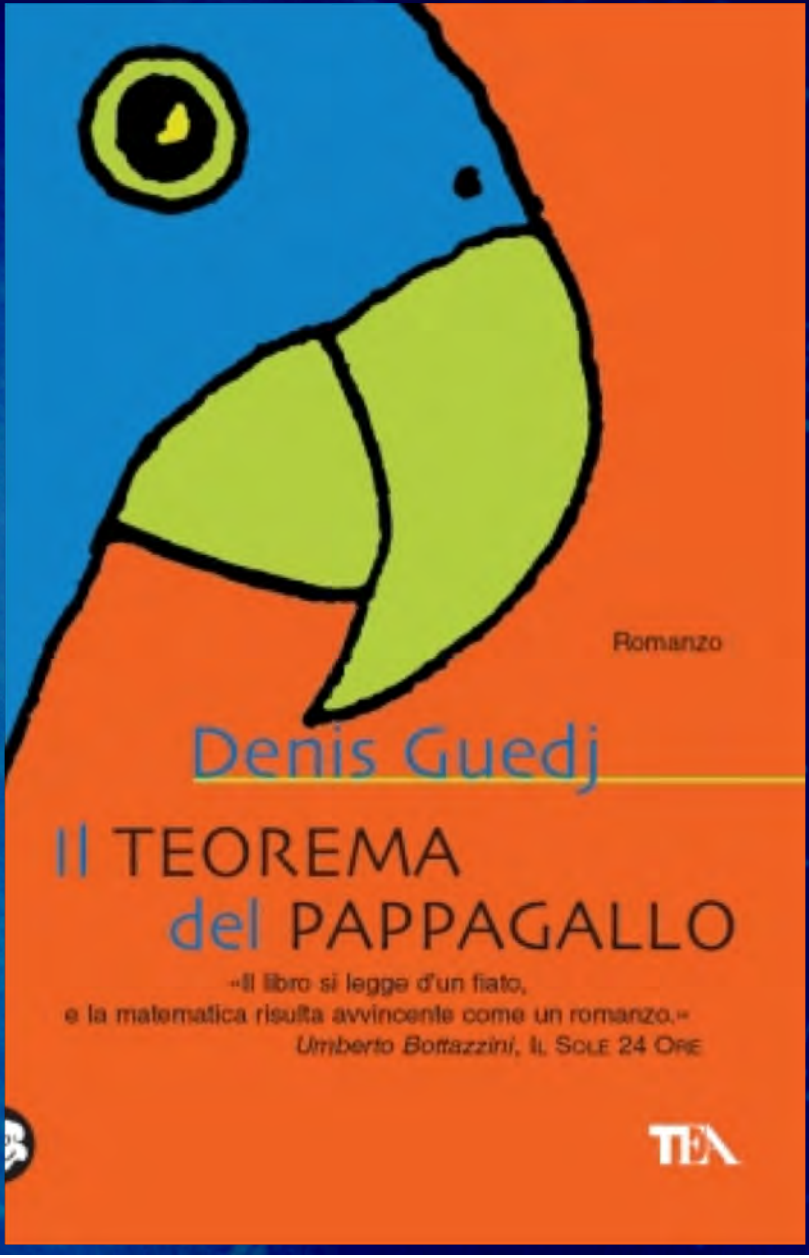
A sequence of computational steps that transform the input into the output. Each step must be “finite” in terms of required time & effort

Muhammad ibn Mūsā al-Khwārizmī



Baghdad,
780 – 850 AD

Hint


Romanzo
Denis Guedj
Il TEOREMA
del PAPPAGALLO
«Il libro si legge d'un fiato,
e la matematica risulta avvincente come un romanzo.»
Umberto Bottazzini, IL SOLE 24 ORE
TEA

Remark

Generally, a problem can be solved by using several algorithms or programs. Although, not all the solutions are equally good

Algorithm analysis

Is required in order to:

- Compare two or more different algorithms

Algorithm analysis

Is required in order to:

- Compare two or more different algorithms
- Foresee the behavior of an algorithm in extreme conditions

Algorithm analysis

Is required in order to:

- Compare two or more different algorithms
- Foresee the behavior of an algorithm in extreme conditions
- Adjust the algorithm parameters to get better results

How evaluating the "quality" of an algorithm?

- ***Subjective*** **criteria:**
  - **Simplicity**
  - **Clearness**
  - **Suitability w.r.t. the target problem**
- ***Objective*** **criteria:**
  - **Computational analysis.**

Efficiency

Ability of solving the proposed problem using a low consumption of computational resources

Resources consumption

- Two fundamental factors of *efficiency*:
  - *Spatial cost* or amount of memory required
  - *Temporal cost* or time required to solve the problem

- In order to solve a given problem, an algorithm or program A will be better than another B if A solves the problem in less time and/or uses less memory than B

Resources consumption

- Two fundamental factors of efficiency:
  - *Spatial cost* or amount of memory required
  - *Temporal cost* or time required to solve the problem

- In order to solve iven problem, an algorithm or program A will be er than another B if A solves the problem in less nd/or uses less memory than B

We will focus our attention mainly on temporal cost

Warning

Sometimes, just time or memory are not the only suitable parameters to appreciate the quality of a program

A simple example

- Let’s consider three simple programs

```c
int main(){ /*A1*/
  int m;
  m = 10 * 10;
  printf("%d\n", m);
}
```

```c
int main(){ /*A2*/
  int i,m; m=0;
  for (i=1; i<=10; i++)
    m = m + 10;
  printf("%d\n", m);
}
```

```c
int main(){ /*A3*/
  int i,j,m; m=0;
  for (i=1; i<=10; i++)
    for (j=1; j<=10; j++)
      m++;
  printf("%d\n", m);
}
```

What is each program actaully doing?

A simple example

- They all compute 10^2

```
int main(){ /*A1*/
   int m;
   m = 10 * 10; /*producto*/
   printf("%d\n", m);
}
```

```
int main(){ /*A2*/
   int i,m; m=0;
   for (i=1; i<=10; i++)
      m = m + 10; /*suma*/
   printf("%d\n", m);
}
```

```
int main(){ /*A3*/
   int i,j,m; m=0;
   for (i=1; i<=10; i++)
      for (j=1; j<=10; j++)
         m++;    /*sucesor*/
   printf("%d\n", m);
}
```

A simple example

- Let's analyze their computational time

```
int main(){ /*A1*/
   int m;
   m = 10 * 10;
   printf("%d\n", m);
}
```

```
int main(){ /*A2*/
   int i,m; m=0;
   for(i=1; i<=10; i++)
      m = m + 10;
   printf("%d\n", m);
}
```

```
int main(){ /*A3*/
   int i,j,m; m=0;
   for(i=1; i<=10; i++)
      for(j=1; j<=10; j++)
         m++;
   printf("%d\n", m);
}
```

Be t_*, t_+, t_s the times required to carried out a *product*, *sum*, and *successor*.

$T_{A1} =$ $T_{A2} =$ $T_{A3} =$

A simple example

- Let's analyze their computational time

```c
int main(){ /*A1*/
  int m;
  m = 10 * 10;
  printf("%d\n", m);
}
```

```c
int main(){ /*A2*/
  int i,m; m=0;
  for (i=1; i<=10; i++)
    m = m + 10;
  printf("%d\n", m);
}
```

```c
int main(){ /*A3*/
  int i,j,m; m=0;
  for (i=1; i<=10; i++)
    for (j=1; j<=10; j++)
      m++;
  printf("%d\n", m);
}
```

Be t_*, t_+, t_s the times required to carried out a *product*, *sum*, and *successor*.

$$T_{A1} = t_* \qquad T_{A2} = 10\,t_+ \qquad T_{A3} = 100\,t_s$$

A simple example

- Let's analyze their computational time

```
int main(){ /*A
  int m;
  m = 10 * 10;
  printf("%d\n",
}
```

```
/*A2*/
=0;
=10; i++)
0;
", m);
```

Which program is the best: A1, A2, or A3 ?

Be t_*, t_+, t_s the times required to carried out a ***product***, ***sum***, and ***successor***.

$$T_{A1} = t_* \qquad T_{A2} = 10\, t_+ \qquad T_{A3} = 100\, t_s$$

A simple example

- Let's assume that A1, A2, A3 are executed on four different computers with different characteristics (different times for *successor*, *sum* and *product* execution)

t_*	$100\ \mu s$	$50\ \mu s$	$100\ \mu s$	$200\ \mu s$
t_+	$10\ \mu s$	$10\ \mu s$	$5\ \mu s$	$10\ \mu s$
t_s	$1\ \mu s$	$2\ \mu s$	$1\ \mu s$	$0{,}5\ \mu s$
$A1$	$100\ \mu s$	$50\ \mu s$	$100\ \mu s$	$200\ \mu s$
$A2$	$100\ \mu s$	$100\ \mu s$	$50\ \mu s$	$100\ \mu s$
$A3$	$100\ \mu s$	$200\ \mu s$	$100\ \mu s$	$50\ \mu s$

A simple example

- Let's assume that A1, A2, A3 are execut... on four different co... ...eristics (different tim... ...oduct execution)

t_*	10	200 μs
t_+	1	10 μs
t_s		0,5 μs
$A1$	100	200 μs
$A2$	100	100 μs
$A3$	100	50 μs

For each computer there's a best one !!!

Remark

A good cost characterization should allow to establish the program *quality* independently of:

- the computer
- the particular sizes of the instances to process

Solution

A good computational characterization of a program:

Cost functional dependency with the size of input – *for large sizes* !

Outline

- Algorithm Complexity
- **Computational analysis**
- Asymptotic Behavior
- Notations O, Ω, Θ
- Examples of Computational analysis

Execution time

In most cases the execution time of an algorithm is influenced not from the actual *values* of input data, but from their overall number

Computational analysis

As a consequence, the computational complexity of a target problem is usually expressed as:

$$T = T(n)$$

where:

- n : *size of the problem:*
  # of the "instances" to be dealt with
- T : *Execution time* :
  # of elementary operations needed to solve the problem resorting to a given algorithm

Advantages

- Generally, programs are useful to solve problems of large sizes of input (if they are small, we could solve them manually)
- Considering large sizes of input, we can carry out simple approximations that considerably simplify cost analysis

Advantages

- Programs no longer depend on:
  - specific execution time values of the different elementary instructions used (if they don't depend on the size of input)
  - sizes of the input of specific instances of the program to solve

Outline

- Algorithm Complexity
- Computational analysis
- Asymptotic Behavior
- Notations O, Ω, Θ
- Examples of Computational analysis

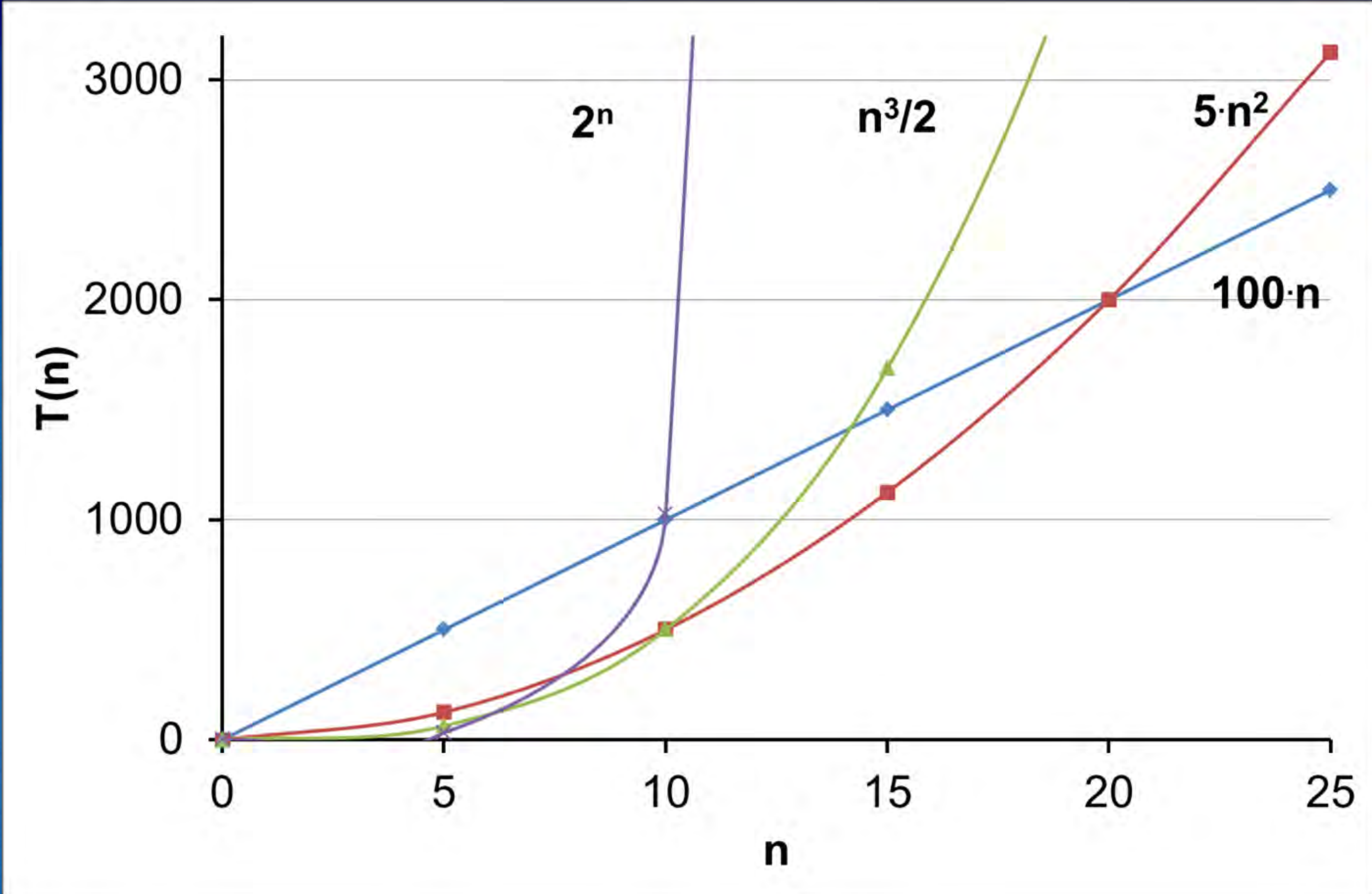
Asymptotic Behavior

- Execution time is often evaluated by studying its asymptotic behavior, i.e., through the performance of the function T(n), which describes the cost of running the algorithm when the size of the problem grows:

$$\lim_{n \to \infty} T(n)$$

- In this way we neglect:
  - constants that do not alter the order of ∞
  - the terms of a lower order

Asymptotic Behavior



Asymptotic Behavior - Advantages

- The analysis of the asymptotic behavior:
  - is independent from the characteristics of the compiler and the machine used to implement the algorithm in the form of program
  - allows to compare the algorithms underlying the programs, rather than the programs themselves
  - is the only tool that allows to determine the maximum approachable size of a given problem.

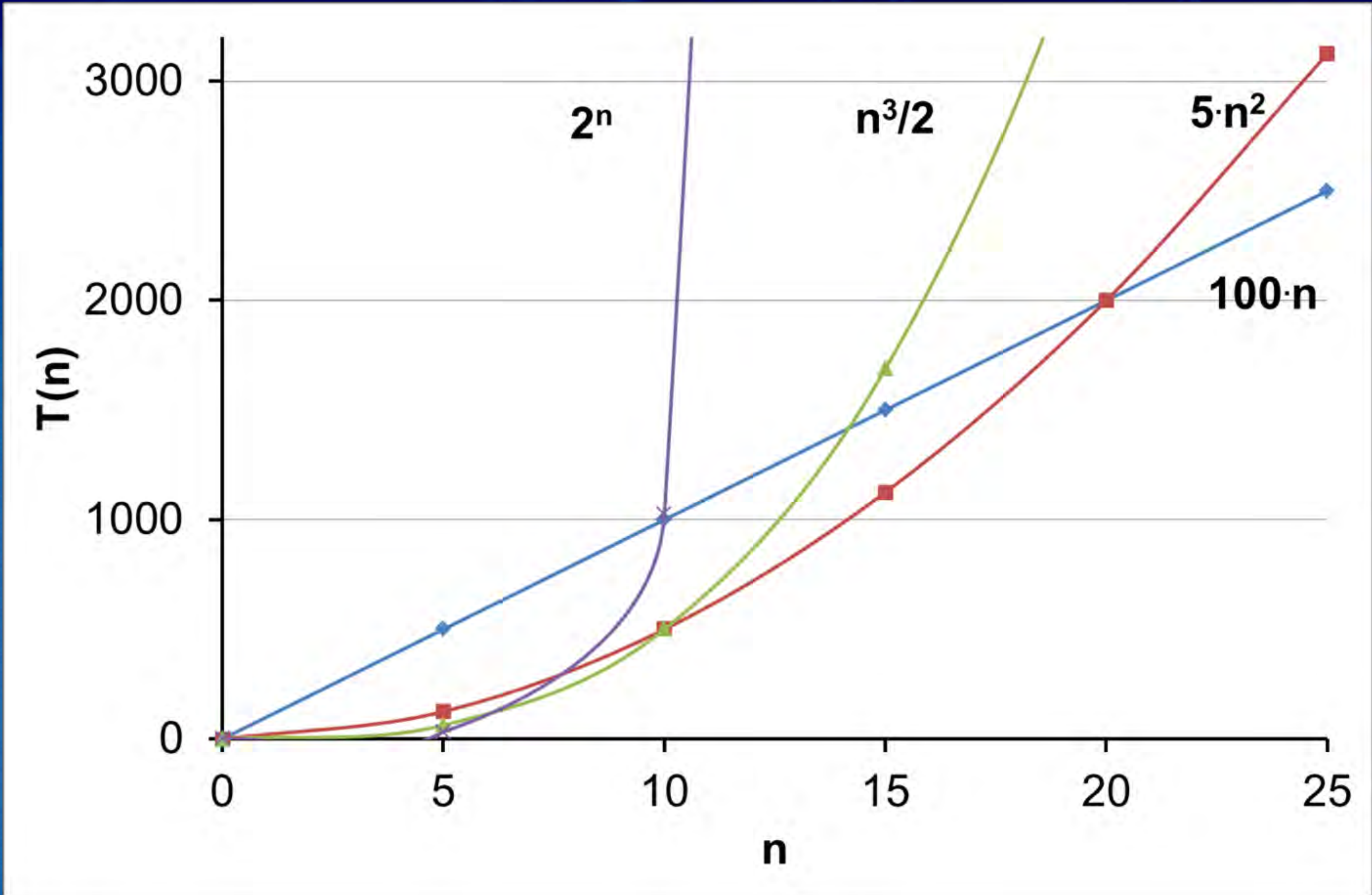
Asymptotic Behavior - Disadvantages

- **Studying only the asymptotic behavior of the function T(n) does not lead to the choice of the algorithm to use.**

Asymptotic Behavior - Disadvantages

- Other items to consider are in fact:
  - the memory amount actually occupied by the program
  - the number of calls to I/O functions
  - the amount of data that the program will have to process:
    - if n is predominantly small, you should consider the exact execution time, taking into account the constant of proportionality, rather than the asymptotic behavior

Asymptotic Behavior - Disadvantages

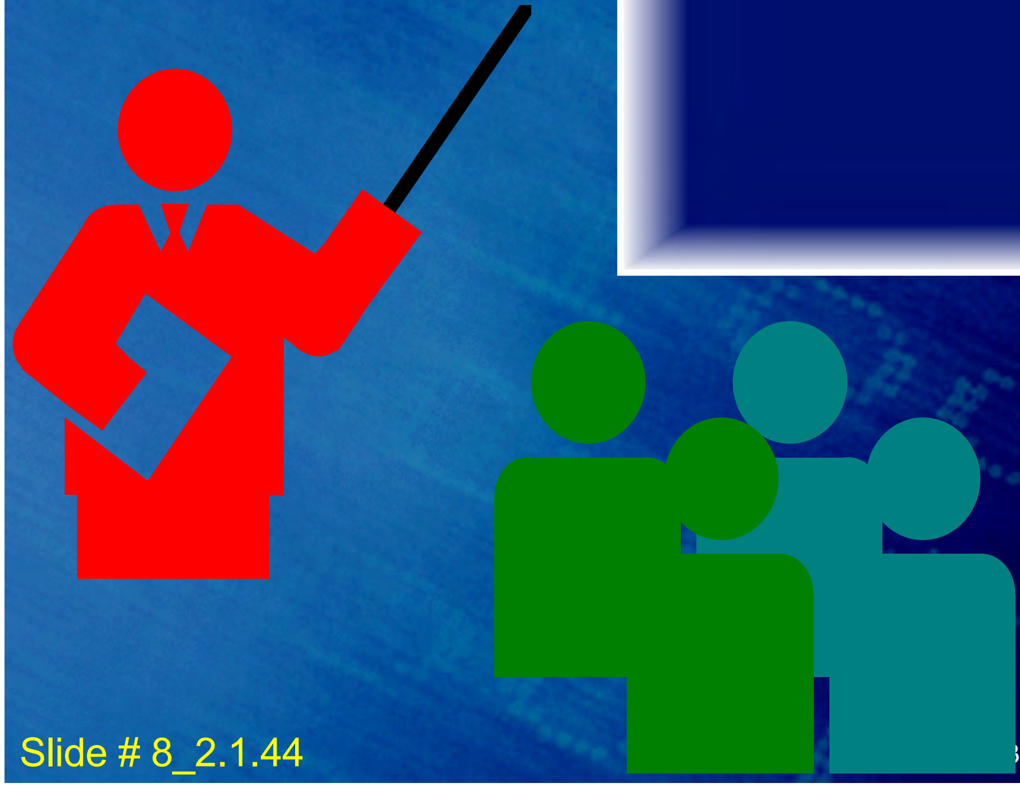


Step

**A STEP is the execution of a code segment which processing time either doesn't depend on the size of input of the considered program,
or it is bounded by a constant.**

Computational cost of a program

Number of STEPS as a function of the size of input of the program



Examples

```c
int main(){ /*A1*/
  int n, m;
  scanf("%d", &n);
  m = n * n; /*producto*/
  printf("%d\n", m);
}
```

```c
int main(){ /*A2*/
  int i, n, m;  m=0;
  scanf("%d", &n);
  for (i=1; i<=n; i++)
    m = m + n; /*suma*/
  printf("%d\n", m);
}
```

```c
int main(){ /*A3*/
  int i, j, n, m;  m=0;
  scanf("%d", &n);
  for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
      m++;  /*sucesor*/
  printf("%d\n", m);
}
```

$T_{A1}(n) =$ $T_{A2}(n) =$ $T_{A3}(n) =$

Examples

Examples

```c
int main(){ /*A1*/
  int n, m;
  scanf("%d", &n);
  m = n * n; /*producto*/
  printf("%d\n", m);
}
```

```c
int main(){ /*A2*/
  int i, n, m;  m=0;
  scanf("%d", &n);
  for (i=1; i<=n; i++)
    m = m + n; /*suma*/
  printf("%d\n", m);
}
```

```c
int main(){ /*A3*/
  int i, j, n, m;  m=0;
  scanf("%d", &n);
  for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
      m++;  /*sucesor*/
  printf("%d\n", m);
}
```

$$T_{A1}(n) = 1, \quad T_{A2}(n) = n, \quad T_{A3}(n) = n^2$$

Outline

- Algorithm Complexity
- Computational analysis
- Asymptotic Behavior
- Notations O, Ω, Θ
- Examples of Computational analysis

Notations O, Ω, Θ

- To describe the asymptotic behavior of an algorithm several notations were introduced.
- Among the most used are to be mentioned:
  - notation Ο: big O
  - notation Ω: big Omega
  - notation Θ: Teta

Notation O

- **An algorithm has a (upper delimitation for) complexity $O(f(n))$:**

$$T(n) = O(f(n))$$
$$T(n) \in O(f(n))$$

iff:

$$\exists c > 0 : \lim_{n \to \infty} \left| \frac{T(n)}{f(n)} \right| = c$$

or, according to the definition of limit, iff:

$$\exists c, n_0 : |T(n)| \leq c|f(n)| \quad \forall n \geq n_0$$

- **As n grows, $T(n)$ grows as maximum as $f(n)$, i.e., $f(n)$ is an *upper* limit to the growth of $T(n)$.**

Notation O – Practical Rules

- **Given that:**

$$T_1(n) = O(f_1(n))$$
$$T_2(n) = O(f_2(n))$$

it is true that:

$$T_1(n) + T_2(n) = O\big(max(f_1(n) + f_2(n))\big)$$
$$T_1(n) \cdot T_2(n) = O(f_1(n) \cdot f_2(n))$$

- **In addition, for any given constant c:**

$$O\big(c \cdot f(n)\big) = O(f(n))$$

Notation Ω

- An algorithm has a (lower delimitation for) complexity $\Omega(f(n))$:

$$T(n)=\Omega(\mathrm{f(n)})$$

iff:

$$\exists c, n_0 : \quad |T(n)| \geq c|f(n)| \quad \forall n \geq n_0$$

Notation Ω – Considerations

- As n grows, $T(n)$ grows at least as $f(n)$, i.e., $f(n)$ is a *lower* limit to the growth of $T(n)$.
- Notation Ω is useful to express the inherent complexity of a given problem.
- Notation Ω can be seen as O reverse, as it holds:

$$T(n)=\Omega(f(n)) \;\Rightarrow\; f(n)=O(T(n))$$

Notation Θ

- An algorithm has a complexity $\Theta(f(n))$:

$$T(n) = \Theta(\mathrm{f}(\mathrm{n}))$$

iff:

$$\exists c_1, c_2, n_0 : c_1|f(n)| \leq |T(n)| \leq c_2|f(n)| \quad \forall n \geq n_0$$

Notation Θ - Considerations

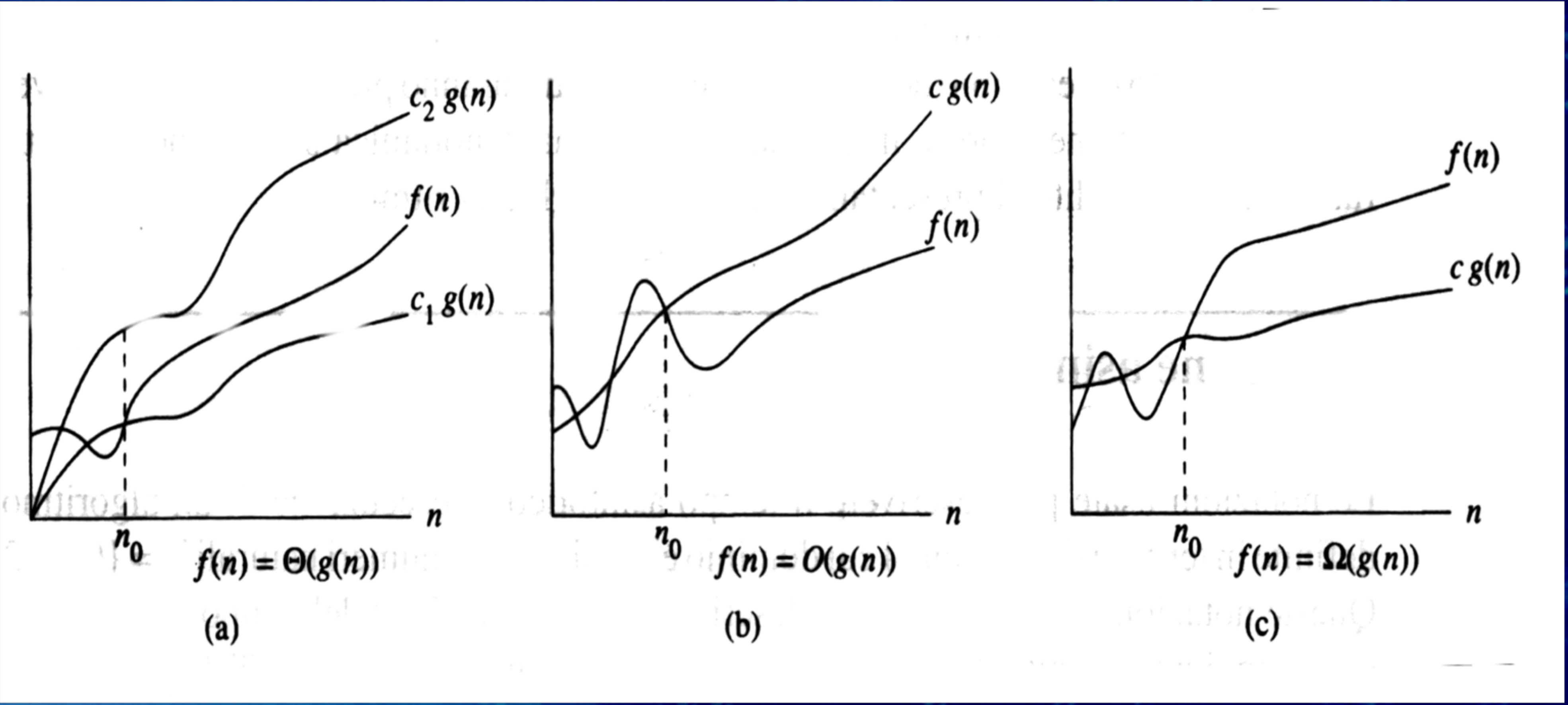
- If $T(n)=\Theta(f(n))$,
  - $T(n)$ e $f(n)$ grows similarly:

$$\begin{cases} T(n) = O(f(n)) \\ f(n) = O(f(n)) \end{cases}$$

  - $f(n)$ is at the same time upper and lower limitation for the growth of $T(n)$:

$$\begin{cases} T(n) = O(f(n)) \\ T(n) = \Omega(f(n)) \end{cases}$$

Notations Θ, O, Ω



Outline

- Algorithm Complexity
- Computational analysis
- Asymptotic Behavior
- Notations $\boldsymbol{O}$, $\boldsymbol{\Omega}$, $\boldsymbol{\Theta}$
- **Examples of Computational analysis:**
  - Iterative procedure
  - Recurrence
  - Recursive procedure

Example 1: Iterative procedure

```
        void bubble (int* A, int N){
        // sorts array into increasing order
        int i, j, temp;
(1)     for(i=0; i<N-1; i++)
(2)        for(j=N-1; j>=i+1; j--)
(3)           if(A[j-1]>A[j]){
(4)              temp = A[j-1];
(5)              A[j-1] = A[j];
(6)              A[j] = temp;
              }
        }
```

Example 1: Iterative procedure

```
       void bubble (int* A, int N){
       // sorts array into increasing order
       int i, j, temp;
(1)    for(i=0; i<N-1; i++)
(2)       for(j=N-1; j>=i+1; j--)
(3)          if(A[j-1]>A[j]){
(4)             temp = A[j-1];
(5)             A[j-1] = A[j];
(6)             A[j] = temp;
             }
       }
```

Example 1: Iterative procedure

```
       void bubble (int* A, int N){
       // sorts array into
       int i, j, temp;
(1)    for(i=0; i<N-1; i++)
(2)       for(j=N-1; j>=i+1; j   )
(3)          if(A[j-1]>A[j]){
(4)             temp = A[j-1];
(5)             A[j-1] = A[j];
(6)             A[j] = temp;
             }
       }
```

$$T_{4\div 6} = c_1$$

Example 1: Iterative procedure

```
       void bubble (int* A, int N){
       // sorts array into increasing order
       int i, j, temp;
(1)    for(i=0; i<N-1; i++)
(2)       for(j=N-1; j>=i+1; j--)
(3)          if(A[j-1]>A[j]){
(4)             temp = A[j-1];
(5)             A[j-1] = A[j];
(6)             A[j] = temp;
             }
       }
```

Example 1: Iterativ

$$T_{2 \div 6} = c_2 \cdot (n - i)$$

```
      void bubble (int* A, in      {
      // sorts array into incr     sing order
      int i, j, temp;
(1)   for(i=0; i<N-1; i++)
(2)      for(j=N-1; j>=i+1; j--)
(3)         if(A[j-1]>A[j]){
(4)            temp = A[j-1];
(5)            A[j-1] = A[j];
(6)            A[j] = temp;
            }
      }
```

Example 1: Iterative procedure

```
        void bubble (int* A, int N){
        // sorts array into increasing order
        int i, j, temp;
(1)     for(i=0; i<N-1; i++)
(2)        for(j=N-1; j>=i+1; j--)
(3)           if(A[j-1]>A[j]){
(4)              temp = A[j-1];
(5)              A[j-1] = A[j];
(6)              A[j] = temp;
              }
        }
```


Rel. 03.03.2019

Example 1: Iterative procedure

$$T_{1 \div 6} = \sum_{i=1}^{n-1} c \cdot (n-i)$$

```
                        * A, int N){
                         nto increasing order

(1)     for(i=0; i<N-1; i++)
(2)         for(j=N-1; j>=i+1; j--)
(3)             if(A[j-1]>A[j]){
(4)                 temp = A[j-1];
(5)                 A[j-1] = A[j];
(6)                 A[j] = temp;
                }
        }
```

Example 1: Iterative procedure

```
    void bubble (int* A, int N){
    // sorts array into increasing order
    int i, j, temp;
(1) for(i=0; i<N-1; i++)
(2)    for(j=N-1; j>=i+1; j--)
(3)       if(A[j-1]>A[j]){
(4)          temp = A[j-1];
(5)          A[j-1] = A[j];
(6)          A[j] = temp;
          }
    }
```

Example 1: Iterative procedure

```
    void bubble (int* A, int N){
    // sorts array into increasing order
    int i, j, temp;
(1) for(i=0; i<N-1; i++)
(2)    for(j=N-1; j>=i+1; j--)
(3)       if(A[j-1]>A[j]){
(4)          temp = A[j-1];
(5)          A[j-1] =
(6)          A[j] =
          }
    }
```

$$T(n) = T_{1\div 6} = \sum_{i=1}^{n-1} c \cdot (n-i)$$

Example 1: Iterative procedure

$$T(n) = T_{1\div 6} = \sum_{i=1}^{n-1} c \cdot (n-i) = c\sum_{i=1}^{n-1}(n-i) =$$

$$= c\left[\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i\right] = c\left[\sum_{i=1}^{n-1} n - \left(\sum_{i=1}^{n} i - n\right)\right] =$$

$$= c\left[n(n-1) - \frac{n(n+1)}{2} + n\right]$$

$$= c\left[\frac{2n^2 - 2n - n^2 - n + 2n}{2}\right] =$$

$$= c\left[\frac{n^2 - n}{2}\right] = O\left(n^2\right)$$

Outline

- Algorithm Complexity
- Computational analysis
- Asymptotic Behavior
- Notations O, Ω, Θ
- Examples of Computational analysis:
  - Iterative procedure
  - **Recurrence**
  - Recursive procedure

Recurrence

An equation that describes a function in terms of its value on smaller functions

Recurrence Examples

$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

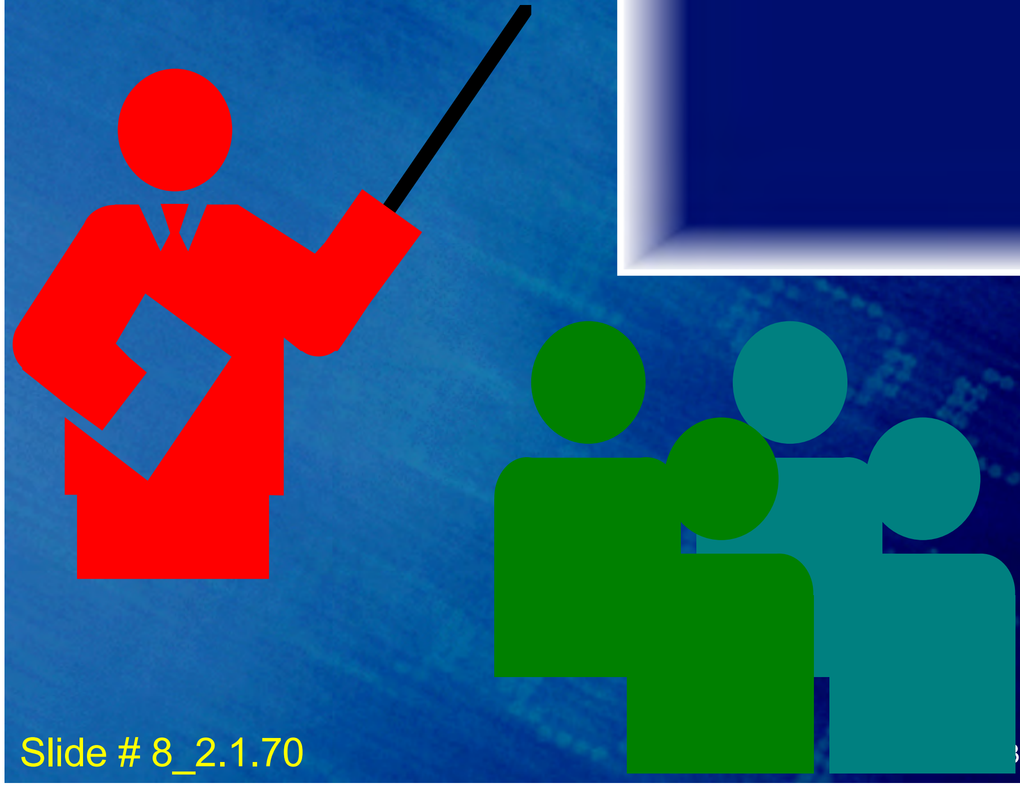
$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

Divide and conquer algorithm

An algorithm that divides the problem of size n into subproblems, each of size n/b



Recurrences' Complexity Analysis

Given the recurrence

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\dfrac{n}{b}\right) + cn & n > 1 \end{cases}$$

with a, b, c costants ≥ 0.

Recurrences' Complexity Analysis

Given the recurrence

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\dfrac{n}{b}\right) + cn & n > 1 \end{cases}$$

with a, b, c costants ≥ 0.

Its complexity is:

$$T(n) = \begin{cases} \Theta(n) & a < b \\ \Theta(n \log_b n) & a = b \\ \Theta\left(n^{\log_b a}\right) & a > b \end{cases}$$

Example 1

- Let's analyze:

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\dfrac{n}{2}\right) + cn & n > 1 \end{cases}$$

Example 1

- **Let's analyze:**

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\dfrac{n}{2}\right) + cn & n > 1 \end{cases}$$

- **Since in this case, a=b, we get:**

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

Example 2

- Let's analyze:

$$T(n) = \begin{cases} c & n = 1 \\ 9T(n/3) + n & n > 1 \end{cases}$$

Example 2

- Let's analyze:

$$T(n) = \begin{cases} c & n = 1 \\ 9T(n/3) + n & n > 1 \end{cases}$$

a = 9   b=3

$$T(n) = \begin{cases} \Theta(n) & a < b \\ \Theta(n \log_b n) & a = b \\ \Theta\left(n^{\log_b a}\right) & a > b \end{cases}$$

Example 2

- **Let's analyze:**

$$T(n) = \begin{cases} c & n = 1 \\ 9T(n/3) + n & n > 1 \end{cases}$$

a = 9   b=3

$$T(n) = \begin{cases} \Theta(n) & a < b \\ \Theta(n \log_b n) & a = b \\ \Theta\left(n^{\log_b a}\right) & a > b \end{cases}$$

$T(n) = \Theta(n^{\log_3 9}) = \Theta(n^2)$

Outline

- Algorithm Complexity
- Computational analysis
- Asymptotic Behavior
- Notations O, Ω, Θ
- Examples of Computational analysis:
  - Iterative procedure
  - Recurrence
  - **Recursive procedure**

Example 3: recursive procedure

```
        int fact (int n){
           // fact(n) computes n!
           int rv;
(1)        if(n<=1){
(2)           rv = 1;
(3)        } else{rv = rv * fact(n-1);}
           return rv;
        }
```

Example 3: recursive procedure

```
        int fact (int n){
            // fact(n) computes n!
            int rv;
(1)         if(n<=1){
(2)             rv = 1;
(3)         } else{rv = rv * fact(n-1);}
            return rv;
        }
```

$$T(n) = \begin{cases} d & \text{se } n \leq 1 \\ c + T(n-1) & \text{se } n > 1 \end{cases}$$

Example 3: recursive procedure

$$T(n)=c+T(n-1)=2c+T(n-2)=kc+T(n-k)$$

if $k=n-1$ it holds:

$$T(n)=(n-1)c+T(1)=(n-1)c+d$$

and so:

$$T(n)=O(n)$$

References

- A.V. Aho, J.E. Hopcroft, J.D. Ullman:
  "Design and Analysis of Computer Algorithms,"
  Addison Wesley, Reading MA (USA), 1974, pp. 364-427
- G. Ausiello, A. Marchetti-Spaccamela, M. Protasi:
  "Teoria e Progetto di Algoritmi Fondamentali,"
  Ed. Franco Angeli, Milano, 1985, pp. 1-75, 120-166
- S. Baase: "Computer Algorithms,"
  Addison Wesley, Reading MA (USA), 1988, pp. 319-360
- E. Horowitz, S. Sahni:
  "Fundamentals of Computer Algorithms,"
  Pittman, London (UK), 1978, pp. 501-613

References

- Z. Manna:
  "Teoria matematica della computazione,"
  Boringhieri, Torino, 1978, pp. 1-79
- C.H. Papadimitriou, K. Steiglitz:
  "Combinatorial Optimization: Algorithms and Complexity,"
  Prentice Hall, Englewood Cliffs NJ (USA), 1982, pp. 342-405
- E.M. Reingold, J. Nievergelt, N. Deo:
  "Combinatorial Algorithms: Theory and Practice,"
  Prentice Hall, Englewood Cliffs NJ (USA), 1977, pp. 401-422

References

- R. Sedgewick:
  "Algorithms in C,"
  Addison Wesley, Reading MA (USA), 1990, pp. 67-80, 633-643
- C.J. Van Wyk:
  "Data Structures and C Programs,"
  Addison Wesley, Reading MA (USA), 1988, pp. 3-48

Малые Автюхи, Калинковичский район, Республики Беларусь