Fundamental Algorithms

Cluj-Napoca, 2.10.19

Agenda

- Administrative stuff
- What is/is NOT this course about
- Computational complexity
  - Basics
  - What and why
  - What NOT and why NOT

Administrative stuff

- Rodica Potolea
  - Professor, Computer Science Department
  - Room C09
  - Rodica.Potolea@cs.utcluj.ro

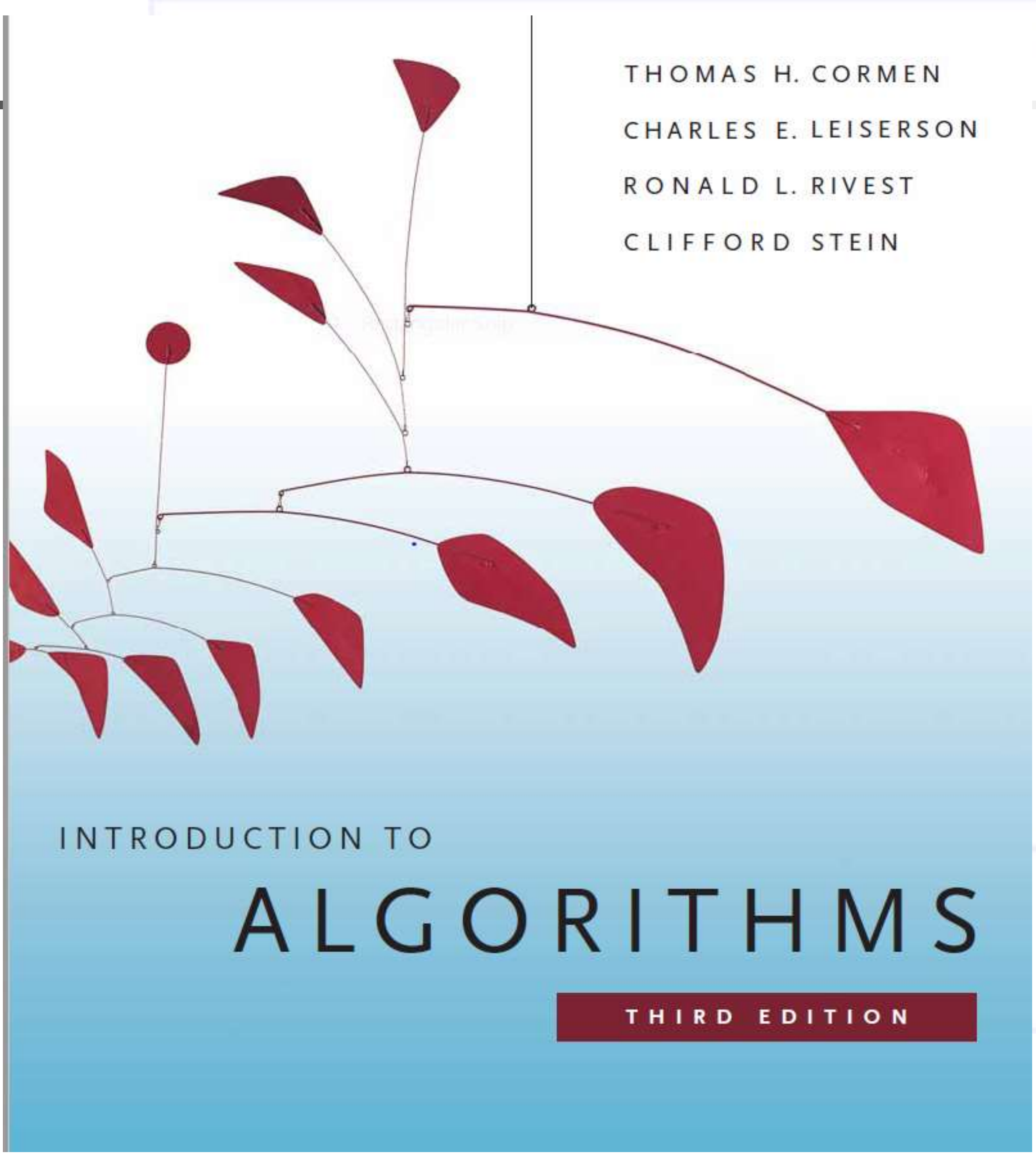
Structure of the course

- **Lectures (every Wednesday, 10-12, room D21)**
  - **Slides and/or blackboard**
  - **Pseudocode**
  - **Discussions**
  - **Open course with Q&As sessions.**
  - **Stop me and ask questions whenever you have. If you have a question, most probably other students have the same question!**
- **Tutorials (every other Tuesday 12 -14 room D12 or Wednesday 14-16 D12)**
  - **Problem solving – analysis and design, evaluation, comparisons**
  - **pseudocode**
- **Labs same content, every group a different faculty member or (former) PhD student, graduate/master**
  - **Problem solving (algorithms implementation, testing and evaluation)**
  - **C++**

Textbook

- **Bible:**
- **Cormen, Leiserson, Rivest, (Stern)**
- **Introduction to Algorithms, first edition 1990 (second/third edition 2001) MIT Press**
- **CS Department Library, Baritiu 26-28, Room M04**

go immediately to check the library!!!



Evaluation

- **MT**
  - after 5 lectures (week 7, hour 10-11)
  - 1h written examination, problems, open book and notes
  - 20% in the Final Grade; CANNOT retake the MT
- **Hands on evaluation (labs work)**
  - Stay in your group
  - 10 assignments
  - Every (other) lab deadline on an assignment (various thresholds; we encourage evolution&knowledge/skills increase)
  - Late assignments policy:
    - 1 week late: 80% from the work grade
    - More than 1 week, no grade (0) on the given assignment
    - Plagiarism policy – 0 tolerance!!! Don't even try!
  - 30% in the Final Grade (need grade of 5 or more to access the FE)
- **FE**
  - 3h written examination, problems, open books and notes

What is this course about?

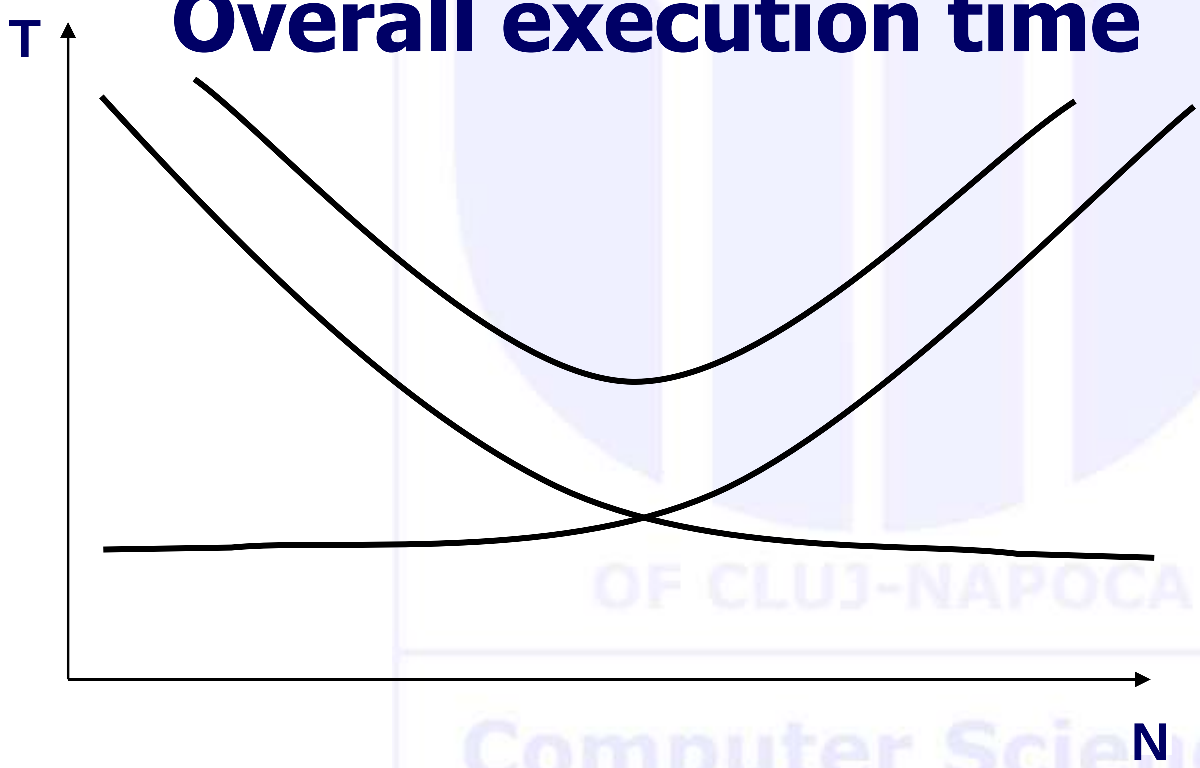
- **NOT a programming course**
- **NOT a DS course**
- **Course on Fundamental Algorithms**
  - **How to:**
    - evaluate algorithms performance
    - compare performance of different algorithms
    - design efficient and optimal algorithms
    - identify a solution to a problem
    - specific efficient algorithms on fundamental problems

Complexity

- **Parameters to be evaluated**
  - **Time**
  - **Memory**
  - **Other**
- **Time** (components)
  - Computation time
  - Communication time (data transfer, partial results transfer, information communication)
- **Computation time (in parallel execution)**
  - As the number of processors involved increases, the computation time decreases
- **Communication time**
  - Quite the opposite!!!

Complexity– cont.

Overall execution time



Complexity – cont.

- **Denote the efficiency of an algorithm by the time required to solve the problem.**
- **How to actually evaluate efficiency?**
  - Measure time
  - time = f(sec)? Why? Why not?
  - Estimate time t=f(n), n=input data size
- **Cases to be considered** (as executions do not always behave the same)
  - Best
  - Worst
  - Average
- **Cases relate to?**
  - the ***algorithm*** implementing the given problem (method/strategy – TBD) so every algorithm could have a different (distinct from other algorithms) best/worst case
  - the ***implementation*** of the algorithm (specific structures employed, the way they are manipulated)
- **Efficiency** – when discussing about the complexity (of a problem), we evaluate the efficiency of the solution (that is, a particular implementation of a given algorithm)
  - Relative
  - Absolute

Complexity – cont. (Efficiency)

- **Comparison between algorithms (relative comparison)**
  - $t(n)$ represent functions expressing execution time
  - Just asymptotic behavior matters (i.e. the term with the fastest growth is considered only)
  - Ex: given $t_1(n)= 3n^2+300n+50$
    $t_2(n)= 2n^3 + 10n^2 +2n+10$

  we count just as $t_1(n) \cong 3n^2$ and $t_2(n) \cong 2n^3$
- **Relative complexity evaluation**
  - between various algorithms
  - efficiency has **degrees of comparison**
  - Alg1 is more / less efficient than Alg2

Complexity – cont. (Optimality)

- **Absolute comparison?**
  - Is performed by comparing with some **absolute measure** (taken from? … we'll see soon. It's a reference value)
  - Defines a relation between the execution performance (we'll say time, yet, it is not quite time in sec/min/hours/…) and that specific measure
  - Provides info about the **optimality** of an algorithm
- **Absolute complexity evaluation**
  - between algorithms and the reference value
  - Optimality does **NOT** have **degrees of comparison**
  - An algorithm is either optimal or NOT optimal

Complexity – cont. (Optimality)

- **O – notation (big Oh function)**
  - Expresses the **upper bound** of a function

$O(g(n))=\{f(n) \mid \exists c, n_0>0,\ 0<=f(n)<=c \cdot g(n),\ \forall n>=n_0\}$

  - **$f(n)=O(g(n))$**
  - **O specifies the asymptotic upper bound**
  - It is related to the **algorithm** (expresses the execution time of the algorithm implementing a problem as a number of execution steps)

Complexity – cont. (Optimality)

- **Ω - notation**
  - Expresses the **lower bound** of a function

$\Omega(g(n))=\{f(n) \mid \exists c, n_0>0, 0<=c \cdot g(n) <=f(n), \forall n>=n_0\}$

  - **$f(n)= \Omega(g(n))$**
  - **Ω specifies the asymptotic lower bound**
  - It is related to the **problem** (expresses the theoretical number of steps required by the problem to be solved)

Complexity – cont. (Optimality)

- **Optimality is related to the lower bound absolute (Ω)**
- **Optimality is a superlative**
  - **Has NO DEGREE OF COMPARISON!!!**
  - **i.e. an algorithm is either**
    - **OPTIMAL,**
    - **or is NOT optimal;**
    - **there is no MORE/LESS optimal!**

Complexity – cont. (Optimality)

- **Absolute comparison defines a relation between O and Ω** (estimation of the performance of an algorithm solving a given problem in relation to lower bound)
  - So, compare O (big Oh function) with Ω
  - Which O?
  - Worst case. Why?
  - Asymptotic behavior (what happens when execution is the slowest?)
  - O() <= Ω () in the best or even average case
    - Ex: The sorting problem has its lower bound Ω(n lgn), and many sorting algorithms have O(1) best case and O(n) average case!!!

Complexity – cont.

- **An algorithm is optimal** if the running time of the algorithm to solve the problem in the worst case scenario equals the lower bound of the given problem and uses just constant additional memory:

$$O = \Omega$$

- **Generally,** we are interested in
  - EITHER developing algorithms with t(n) such that

$$\Omega <= t(n) <= O$$

where O = running time of the best known algorithm for the given problem

  - OR identifying the best known algorithms
- The good news
  - This is what we are doing in this course
- The bad news
  - many of the real-world problems do not have good algorithms
- Even worst
  - No such algorithms will exist (soon? EVER!). NPC problems (TBD …but … this is beyond the scope of this class. It's the master course!)

Complexity – cont.

Rules for estimating **O (Big Oh function)**

1. $O(c \cdot f(n)) = O(f(n))$
2. $O(f_1(n) \cdot f_2(n)) = O(f_1(n)) \cdot O(f_2(n))$
   in nested loops
3. $O(f_1(n)+f_2(n)) = O(f_1(n))+O(f_2(n))$
   in consecutive loops
4. When expressing O, only leading term is considered

Complexity – cont.

- **leading (lim)**

f1(n)	leads	f2(n)	
n^n		$n!$	
$n!$		a^n,	$a>1$
a^n		b^n,	$a>b$
a^n		n^b,	$a>1$
$\log_a n$		$\log_b n$,	$b>a>1$
$\log_a n$		1,	$a>1$

- **Vals of Ω() for some problems**
  - Searching $\Omega(\log n)$
  - Selection $\Omega(n)$
  - Sorting $\Omega(n \cdot \log n)$

The base of the log in CS is 2

Complexity – cont.

- Interpretation O(1): constant time (i.e. regardless the dimension of the input data, the algorithm has always the same running time)
- Asymptotic behavior:
  - For $t1(n)= 3n^2+3n+5 \Rightarrow O(n^2)$
  - For $t2(n)= 2n^3 + 100n^2 +25n+1000 \Rightarrow O(n^3)$
- For "real" values (i.e. small sizes of data, small n) it could be that the leading term is not leading:

  $100n^2 > 2n^3$!

  $100n^2 = 2n^3 : 2n^2$

  $100/2=n$

  So for n< 50, the second term in t2 grows faster!!!

Complexity – cont.

- **Ω** characterizes the **problem**, lower bound
- **O** characterizes the **algorithm** that solves that problem, upper bound
- if Ω = O in the worst case + no additional memory is used by the algorithm (sometime, logarithmic space allowed – to be discussed later – then optimal algorithm)
- If no optimal algorithm is known, what solutions are acceptable?
- Q: How fast the max dim (of the problem that can be solved on a computer) grows in case we increase the speed of the computer?
- How different **classes** of algorithms affect performance?

Complexity – cont.

- What classes are interesting (to be considered)?
- Experiment: let's consider 2 classes of algs:
  - Alg1: polynomial
  - Alg2: exponential
- Assume a new hardware system is built, and its speed increases **V** times (compared to our former system)
- **Q?** How does this increase the max dim of the problem to be solved on the new system?
- That is: estimate $n_2=f(V,n)$ given
  - V=increase of speed of the new machine
  - n=max dim on the former (let's call it old) machine

Complexity – cont.

Alg1: $\mathbf{O(n^k)}$

	Oper.	Time
M1(old):	n^k	T
M2(new):	n^k	T/V
	Vn^k	T

$$(n_2)^k = Vn^k = (v^{1/k}\, n)^k$$

$$\text{So, } n_2 = v^{1/k}\, n$$

Favorable consequence:

If the **speed** of the machine increases **V times**,

Then the max dimension of the problem increases $\mathbf{v^{1/k}}$ **times**.

Notes:

- $\mathbf{v^{1/k}}$ is small value
- But the degree of the polynomial (k) is small for most problems
- AND, it is a multiplicative increase

Complexity – cont.

Alg2: $\mathbf{O(2^n)}$

	Oper.	Time
M1(old):	2^n	T
M2(new):	2^n	T/V
	$V2^n$	T

$$(2)^{n_2} = V\,2^n = 2^{\lg V + n}$$

So $n_2 = n + \lg V$

Disadvantageous consequence!

If the speed increases V times,

Then the dimension increases with lgV.

The bad News:

- VERY small increase (**lg**)
- Even worst: it is **additive**!!! ☹

Complexity – cont.

Speed of the new computer in terms of the old one: $V_2 = V \cdot V_1$

Alg1: $\mathbf{O(n^k)}$: $n_2 = v^{1/k} \cdot n$

Alg2: $\mathbf{O(2^n)}$: $n_2 = n + \lg V$

CL: For exp algs, no matters how many times we increase the speed of the system, the dim increases with an additive constant!!!

Sol:

- avoid designing exponential solutions! NEVER EVER write exponential algorithms!!!
- are there any problems with unknown polynomial sols?
- P=NP ? 1 million USD problem (since 1971, Stephen Cook)

Complexity – cont.

- Evaluating the complexity for Divide et Impera algorithms

```
divide_et_impera(n, I, O)
    if n<=n0
        then    direct_solution(n, I, O)
        else    divide(n, I1,I2,…,Ia)
                divide_et_impera(n/b,I1,O1)      //a rec. calls
                divide_et_impera(n/b,I2,O2)
                …
                divide_et_impera(n/b,Ia,Oa)
                combine(O1,O2,…,Oa,O)
```

Complexity – cont.

- Assumption f(n) = time (complexity) of the alg – sequence except for the recursive calls (div&comb)
- $f(n)= n^c$
- $$t(n)= \begin{cases} t_0 & \text{if } n<n_0 \\ at(n/b)+f(n) & \text{if } n>=n_0 \end{cases}$$

This is something to remember:

$t(n)= at(n/b)+n^c$ a = number of recursive calls

b= division factor of the input

c = degree of the polynomial describing the run time of the sequence outside the recursive calls

Complexity – cont.

Calling tree

n^c => n^c

$(n/b)^c$ $(n/b)^c$ … $(n/b)^c$ => $a\,(n/b)^c$

… $(n/b^2)^c$ $(n/b^2)^c$ … $(n/b^2)^c$ … => $a^2(n/b^2)^c$

…

Nb of levels?

$\log_b n$

$t(n) = n^c + a\,(n/b)^c + a^2(n/b^2)^c + \ldots$

$= n^c[1 + a/b^c + (a/b^c)^2 + \ldots (a/b^c)^{\log_b n-1}]$

Geometric progression:

- first term=1
- ratio=a/b^c
- number of terms=$\log_b n$

Complexity – cont.

$t(n)=n^c[1+a/b^c+(a/b^c)^2+\ldots(a/b^c)^{\log_b n-1}]$

Cases:

1. $q<1;\ a<b^c$ => $O(n^c)$
2. $q=1;\ a=b^c$ => $O(n^c \cdot \log_b n)$
3. $q>1;\ a>b^c$ => $O(?)$

$t=first\_term \cdot (q^n-1)/q-1$

$t(n)= n^c[(a/b^c)^{\log_b n}-1]/[a/b^c-1]$

Take the asymptotic term: $n^c (a/b^c)^{\log_b n}$

Complexity– cont.

$t(n)= n^c[(a/b^c)^{\log_b n} -1]/[a/b^c-1]$ case 3 $(q>1)$ $a>b^c$

the asymptotic term $n^c (a/b^c)^{\log_b n}$

Q?: $O(n^c (a/b^c)^{\log_b n})=O(n^\alpha)$

if yes, $\alpha=?$

n^α	$=n^c (a/b^c)^{\log_b n}$	divide by n^c
$n^{\alpha-c}$	$=(a/b^c)^{\log_b n}$	apply $\log_b$
$(\alpha-c) \log_b n$	$= \log_b n \cdot \log_b (a/b^c)$	divide by $\log_b n$
$(\alpha-c)$	$= \log_b a-c$	add c
α	$= \log_b a$	

Complexity – cont.

Cl: if $f(n)= n^c$

1. **$a<b^c$ => $O(n^c)$**
2. **$a=b^c$ => $O(n^c \cdot \log_b n)$**
3. **$a>b^c$ => $O(n^{\log_b a})$** !! Independent of c

Obs: b should be scaler (b>1)

composition should comply the partition rule!

In most cases, either divide, or combine is some (almost) default operation (or it takes just O(1))

Ex: quick sort combine is default (sort insitu)

merge sort divide is almost default - computes the middle index O(1)

Complexity– cont.

- Particular cases:

1. c=1 => f(n)=n

$$t(n)=\begin{cases} O(n) & \text{if } a<b \\ O(n\cdot \log_b n) & \text{if } a=b \\ O(n^{\log_b a}) & \text{if } a>b \end{cases}$$

Ex: qsort a=b=2=>$O(n\cdot \log_2 n)=O(n\cdot \log n)$

IS qsort optimal? Justify!

Complexity – cont.

- Particular cases:

2. c=0 => f(n)=ct

Q? Exists such de algs?

$$t(n)=\begin{cases} N/A & \text{if } a<b^o \Leftrightarrow a<1! \\ O(\log_b n) & \text{if } a<b^o \Leftrightarrow a=1 \\ O(n^{\log_b a}) & \text{if } a<b^o \Leftrightarrow a>1 \end{cases}$$

Ex: a=1, b=2 search in BST => O(logn)

a=2, b=2 tree traversal=> O(n)

Sorting algorithms

- What is all about?
- Direct strategies – tutorial
- Advanced strategies - course