Fundamental Algorithms

Course 1, 2020

Cluj-Napoca

Agenda

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

Administrative stuff

- English track + Romanian track A
  - Rodica Potolea
    - Professor, Computer Science Department
    - Room C09
    - Rodica.Potolea@cs.utcluj.ro
- Romanian B track
  - Camelia Lemnaru – part 1
    - Camelia.Lemnaru@cs.utcluj.ro
  - Ciprian Oprisa – part 2
    - Ciprian.Oprisa@cs.utcluj.ro

Structure of the course

- **Lectures (MS Teams + moodle)**
  - https://moodle.cs.utcluj.ro/course/view.php?id=292
  - Slides + discussions + pseudocode
  - Open course with Q&As sessions.
  - Stop us and ask questions whenever you have. If you have a question, most probably other students have the same question!
- **Tutorials (MS Teams + moodle)**
  - Problem solving – analysis and design, evaluation, comparisons
  - Pseudocode
- **Labs same content, every group a different faculty member or (former) PhD student, graduate/master**
  - Each group – separate channel + moodle
  - Problem solving (algorithms implementation, testing and evaluation)
  - C/C++

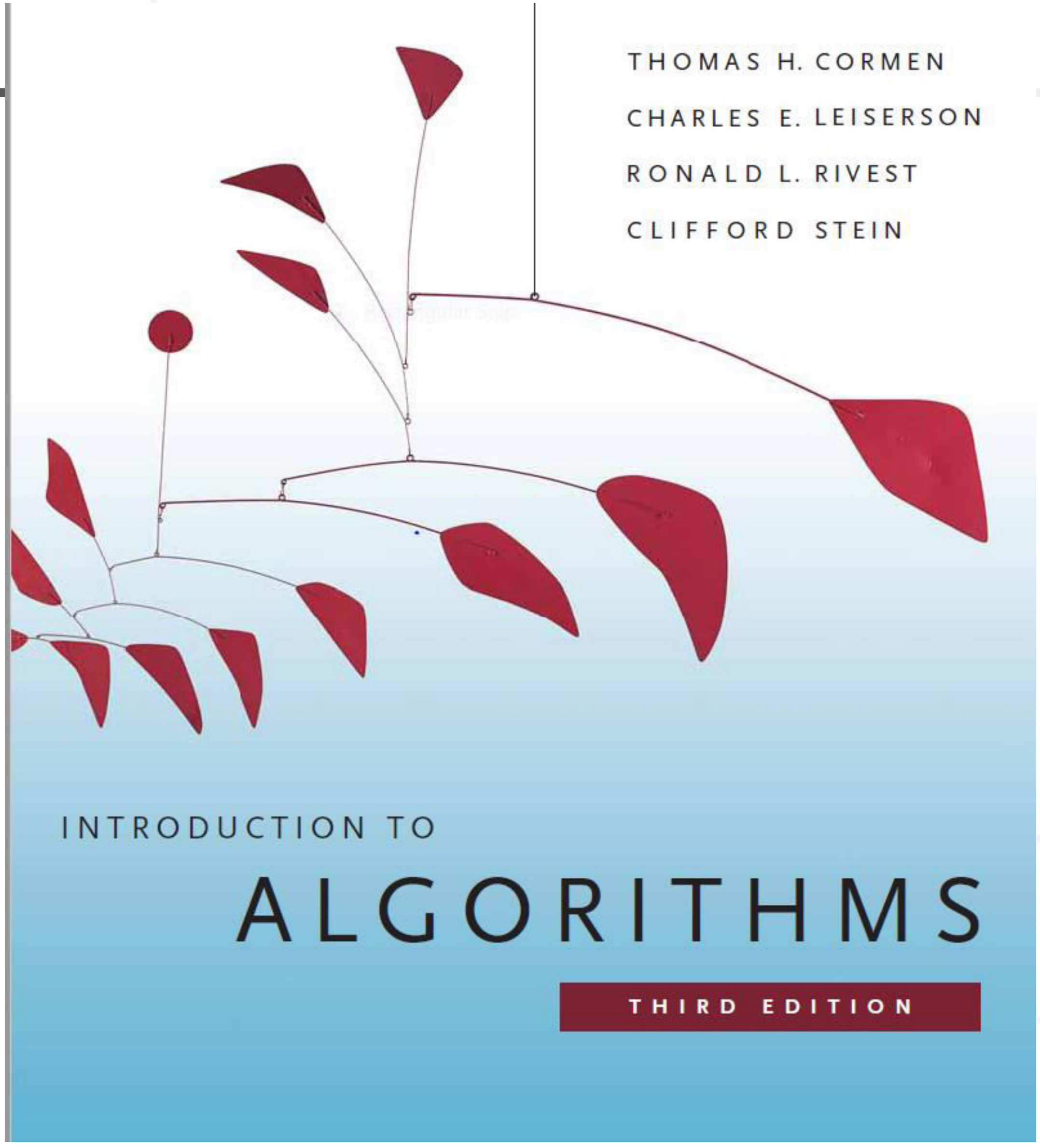
Lab sessions info

Group	Enrollment key (moodle)	TA	URL Lab session
30221	Group@30221_2020	***Richard Ardelean***	
30222	Group@30222_2020	***Paul Helmer***	
30223	Group@30223_2020	***Robert Vacareanu***	
30224_1	Group@30224_1_2020	***Olariu Eliza***	
30224_2	Group@30224_2_2020	***Chira Codrin***	
30225	Group@30225_2020	***Voichita Iancu***	
30226	Group@30226_2020	***Vasile Suciu***	See Teams
30227	Group@30227_2020	***Ramona Tolas***	See Teams
30228	Group@30228_2020	***Cristian Militaru***	See Teams
30229	Group@30229_2020	***Raluca Portase***	See Teams
302210	Group@302210_2020	***Dan Toderici***	See Teams
30421	Group@30421_2020	***Csongor Varady***	
30422	Group@30422_2020	***Ciprian Oprisa***	
30423	Group@30423_2020	***Anda Stoica***	
30424	Group@30424_2020	***Tibor Kadar***	
CSC	Group@CSC_2020	***Camelia Lemnaru***	See Teams

Textbook

- Bible:
- Cormen, Leiserson, Rivest, (Stern)
- Introduction to Algorithms, first edition 1990 (second/third edition 2001) MIT Press
- Have it on moodle (url) – e-copy
- CS Department Library, Baritiu 26-28, Room M04 – hard copy

go immediately to check the library!!!



Evaluation

- **Course quizzes**
  - Between 3 and 7 quizzes, during the course, un-announced
    - Target info in the current course and ALL the things discussed up to that point
    - Delivered on Moodle
  - 20% of the Final Grade; CANNOT retake the quizzes
- **Hands on evaluation (laboratory assignments)**
  - Stay in your group
  - 10 assignments
  - Every (other) lab deadline on an assignment (various thresholds; we encourage evolution & knowledge/skills increase)
  - Late assignments policy:
    - <u>Some</u> assignments can be submitted 1 week late: 80% of max 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 be allowed to take the final exam)
- **FE**
  - 2-3h examination (moodle): algorithm traces, questions, algorithm design for specific problems, complexity analysis; open books

What is this course about?

- **NOT a programming course**
- **NOT a Data Structures 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

What is an algorithm?

- An **algorithm** is

What is an algorithm?

- An **algorithm** is
  - “Word used by programmers when they do not want to explain what they did”

What is an algorithm?

- An **algorithm** is
  - "Word used by programmers when they do not want to explain what they did"
  - "Something that made something do something in some amount of time"

What is an algorithm?

- An **algorithm** is
  - “Word used by programmers when they do not want to explain what they did”
  - “Something that made something do something in some amount of time”
  - “When a piece of code from stackoverflow works but you don't know why and how!”

What is an algorithm?

- An **algorithm** is
  - "Word used by programmers when they do not want to explain what they did"
  - "Something that made something do something in some amount of time"
  - "When a piece of code from stackoverflow works but you don't know why and how!"
  - **A sequence of computational steps that transform the input into the output.**
    - **specific computational procedure for achieving the desired input/output relationship.**

What is an algorithm?

- An **algorithm** is
  - "Word used by programmers when they do not want to explain what they did"
  - "Something that made something <u>do something</u> in some <u>amount of time</u>"
  - "When a piece of code from stackoverflow <u>works</u> but you don't know why and <u>how</u>!"
  - **A sequence of computational steps that transform the input into the output.**
    - **specific computational procedure for achieving the desired input/output relationship.**

An algorithm ...

- …has to be

An algorithm ...

- …has to be
  - correct
    - *"Program testing can be used to show the presence of bugs, but never to show their absence"* (Dijkstra, 1970, "Notes On Structured Programming")

An algorithm ...

- …has to be
  - correct
  - efficient
    - main goal of this course
    - more on this soon…

An algorithm …

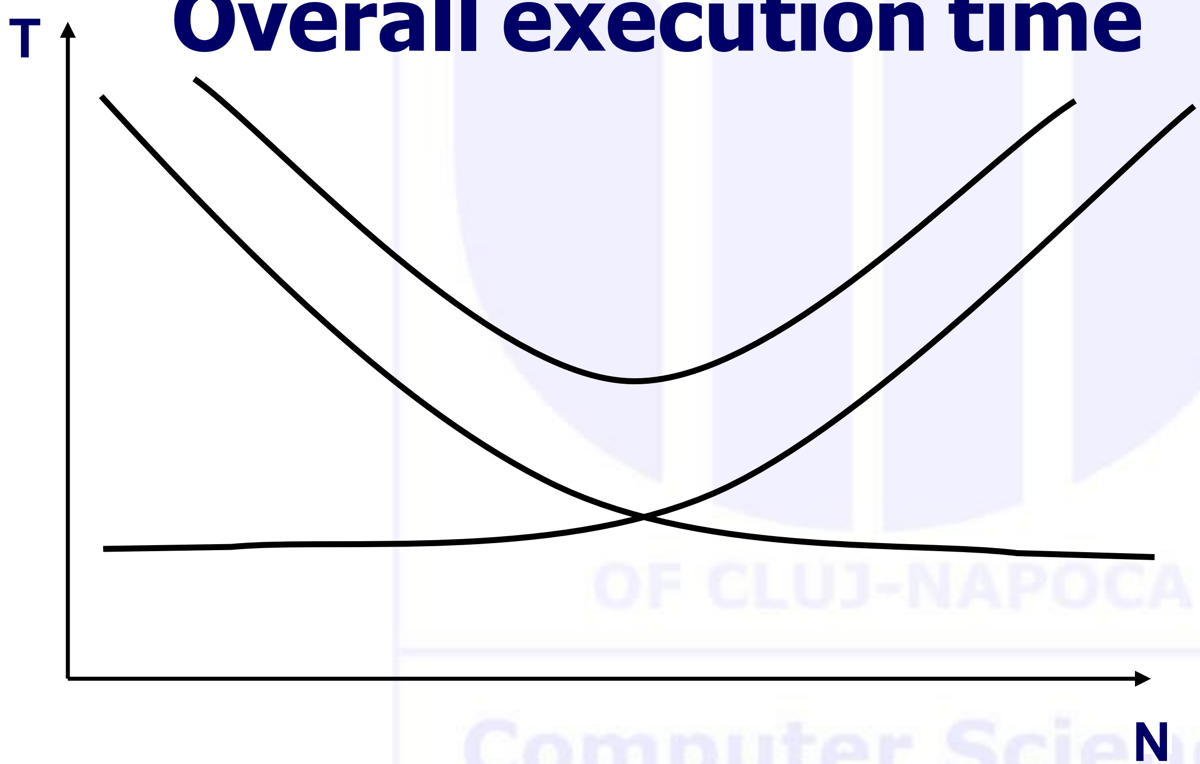
- …has to be
  - correct
  - efficient
  - *easy to implement*
    - *see https://en.wikipedia.org/wiki/Galactic_algorithm*

Complexity

- **Algorithm Complexity vs Problem Complexity!**
  - **Highly related (**details soon)
- Algorithm complexity **question:** What is the amount of resources required to run THE algorithm?
- **Parameters to be evaluated**
  - **Time**
  - **Memory**
  - **Other** (secondary memory accesses, network traffic, etc.)
- **Time** (components – in parallel execution)
  - *Computation time*
    - As the number of processors increases, computation time decreases
  - *Communication time* (data transfer, partial results transfer, information communication)
    - The opposite

Algorithm Complexity– cont.

Overall execution time



Algorithm Complexity – cont.

- Denote the **efficiency** of an algorithm by the <**time**> required to solve the problem.
  - <time> can be replaced with any other resource, but it is the most important
- **How to actually evaluate efficiency?**
  - Measure ACTUAL 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)
- Handled by the **Analysis of Algorithms** field

Problem Complexity

- Handled by **Computational Complexity Theory** field
- The question: What is the **least amount of resources** necessary by **any of the possible** (known/unknown) **algorithms that could solve a given problem?**
- Mathematical models of computation
- Establish the *practical* limits on what computers (and algorithms) can/cannot do
- In practice, 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) represents 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 $t_2(n) \cong 2n^3$

    … more on this will follow
- **Relative complexity evaluation**
  - between various algorithms
  - efficiency has **degrees of comparison**
  - Alg1 is more / less efficient than Alg2

Complexity – cont. (Optimality)

- **Absolute comparison?**
  - compare with some **absolute measure**?
  - **reference value = problem complexity**
  - Provides info about the **optimality** of an algorithm
  - Optimality does **NOT** have **degrees of comparison**
  - An algorithm is either optimal or NOT optimal
- How to operationalize this
  - What do you compare on the algorithm side?
  - What does problem complexity even mean, from a practical standpoint?

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 of the problem!)
  - 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. (Optimality)

- **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 $\Omega()$ 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 $t_1(n)= 3n^2+3n+5 => O(n^2)$
  - For $t_2(n)= 2n^3 + 100n^2 +25n+1000 => O(n^3)$
- 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 (sometimes, 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 algorithms:
  - 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 size 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 size 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: **$O(n^k)$:** $n_2 = v^{1/k} \cdot n$

Alg2: **$O(2^n)$:** $n_2 = n + \lg V$

CL: For exp algs, no matter how many times we increase the speed of the system, the size 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

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

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

…

How many levels?

Complexity – cont.

Level	**Calling tree**	# *Ops*
0	n^c	$=> n^c$
1	$(n/b)^c \quad (n/b)^c \quad \ldots \quad (n/b)^c$	$=> a\,(n/b)^c$
2	$\ldots\ (n/b^2)^c \quad (n/b^2)^c \quad \ldots. \quad (n/b^2)^c\ \ldots$	$=> a^2(n/b^2)^c$
…	…	…
$\log_b n$		$=> a^{\log_b n}\,(n/b^{\log_b n})^c$

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

Geometric progression:

first_term =1

ratio (q) $= a/b^c$

number of terms = $\log_b n+1$

Complexity – cont.

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

Cases:

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

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

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

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

Complexity – cont.

Case 3: $q>1$, $a>b^c$

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

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

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

Q? Algorithm examples?

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. $c=0 \quad \Rightarrow f(n)=ct$

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

Q? Algorithm examples?

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

Required Bibliography

- From the Bible – Chapters 2, 3 and 4.6 -> 4.6 (inclusive)