Fundamental Algorithms

WS 2006/2007

Jens Ernst

Lehrstuhl für Effiziente Algorithmen

Institut für Informatik

General Information:

- Audience: Students of the program “Computational Science and Engineering” (CSE)
- Lecture: 2 hours/wk
- Practice Session (not mandatory): 2 hours/wk

General Information (contd):

- Lecturer: Dr. Jens Ernst, Zimmer 03.13.061

  Email: ernstj@in.tum.de

  Tel. 289 – 19426

  Office hours: None (just call)

- Lecture: Tue. 11:15 - 13:00, Room 03.11.018

- Practice Session: What day/time suits you?

- Homework assignments: Not mandatory, but recommended; Not required for admission to the exams
- Tests: Midterm and Final exams
- Dates will be announced plenty ahead of time

- Lecture Material:
  1. Introduction, Basics and Notation
  2. Developing Algorithms by Induction
  3. Searching and Sorting
  4. Data Structures and Advanced Searching
  5. Graph Algorithms
  6. Text Algorithms
  7. Algebraic and Numerical Algorithms
  8. Data Compression

•Recommended Literature:

- Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein, “*Introduction to Algorithms*” MIT Press, Cambridge MA, 2. Edn, 2001
- Robert Sedgewick, “*Algorithms*” Pearson Education, München 2002
- S. Dasgupta, C.H. Papadimitriou, U.V. Vazirani Available online at http://www.cse.ucsd.edu/~dasgupta/mcgrawhill/all.pdf

Note: None of these is “the textbook” for this course. Please take notes in class.

1. Introduction:

Definition (*Algorithm*): An algorithm is a uniquely defined procedure to obtain the desired output, given some set of input data. Here we consider algorithms satisfying the following properties:

- ***sequential***: At each point in time exactly one operation is carried out

  Remark: Parallel and distributed algorithms are non-sequential

- ***deterministic***: At each point in time, the next operation to be carried out is uniquely defined

- **Remark 1**: *Complexity theory*, for instance is concerned with non-deterministic algorithms in which each step can have two or more subsequent steps.
- **Remark 2**: *Randomized* Algorithms can decide between alternative operations as a result of a random event, e.g. by flipping a coin.
- ***statically finite***: The description of the algorithm (e.g. in the form of pseudo source code requires only a finite amount of space.

- ***dynamically finite*: At each point in time during the execution of the algorithm, only a finite amount amount of storage is used.**
- ***termination***: For any input, the execution ends after a finite number of steps.

  Remark: This may not be the case for *online algorithms*, that do not know their entire input at the beginning of their execution.

Standard Examples of Algorithmic Problems:

- Data organization and efficient data access in a web search engine
- Data storage and efficient data manipulation in a database
- Assembly of the entire human genome sequence
- Computing a VLSI layout
- Routing of TCP/IP packets in the internet
- Compression of an audio or video file
- Efficient encryption and decryption of a set of secret data to be transmitted over a non-trustworthy medium

etc…

Algorithms and Efficiency:

Typically, the efficiency of algorithms is assessed in terms of *running time and storage usage*. Both are specified as a function of input size (given in bits). **(Why?)**

- *Running time* is mostly measured as the *number of operations* carried out during the execution (e.g. number of arithmetic operations or number of comparisons).

Example: Suppose some machine can carry out one operation per microsecond. Let us consider several algorithms of varying efficiency for the same problem: ...

(contd.)

For various input sizes *n*, we die give the running time *T(n)* (wall clock time in seconds) for different algorithms requiring $t(n)=1000n$, $1000n \cdot \log(n)$, $100n^2$ *or* 2^n operations.

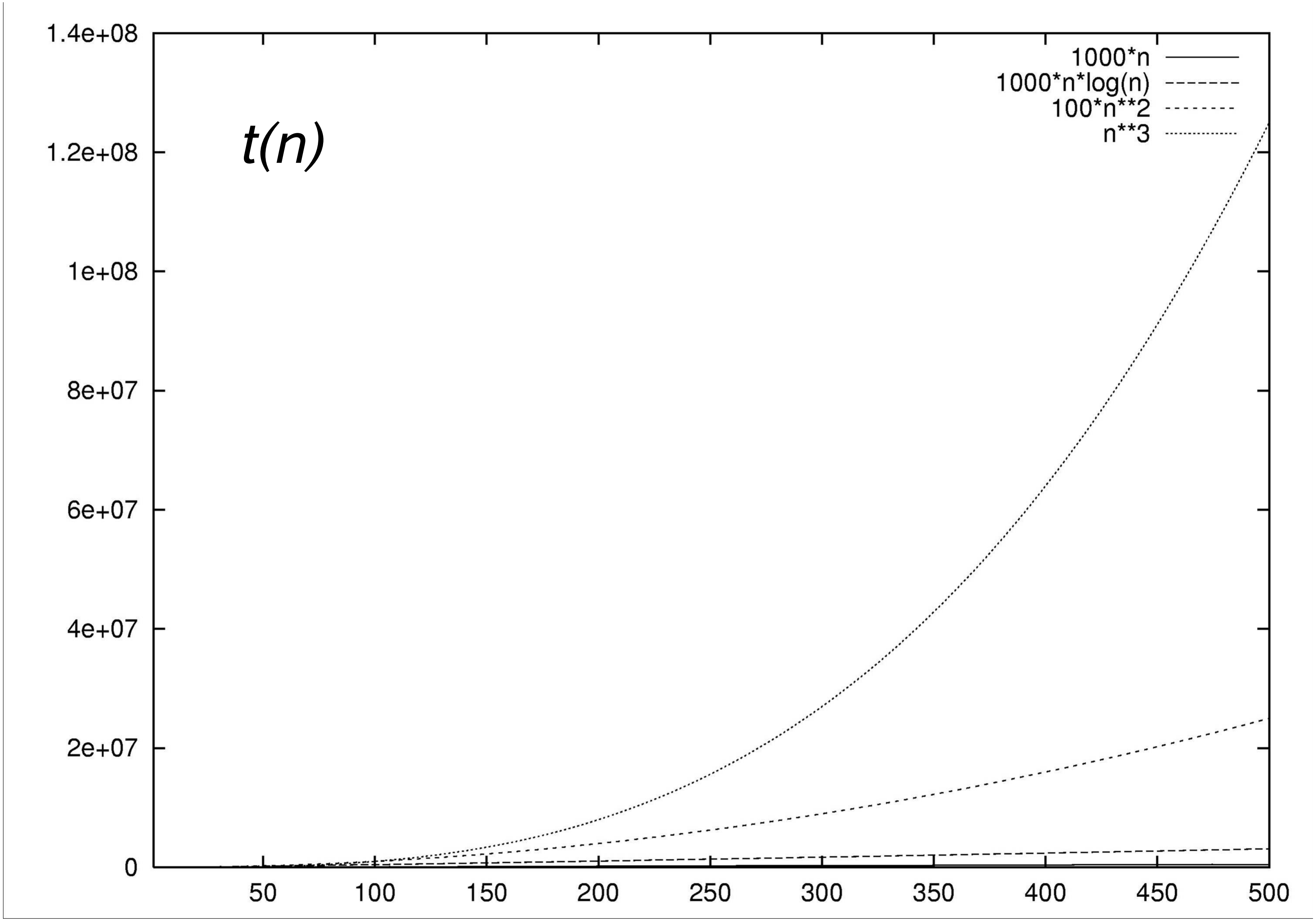
	20	**50**	**100**	**200**	**500**	**1000**	**10000**
1000 n	0.02s	0.05s	0.1s	0.2s	0.5s	1s	10s
1000 n log n	0.09s	0.3s	0.6s	1.5s	4.5s	10s	2 min
100 n^2	0.04s	0.25s	1s	4s	25s	2 min	2.8 h
10 n^3	0.02s	1s	10s	1 min	21 min	2.7 h	116 d
2^n	1s	35 y	3×10^4 cent				

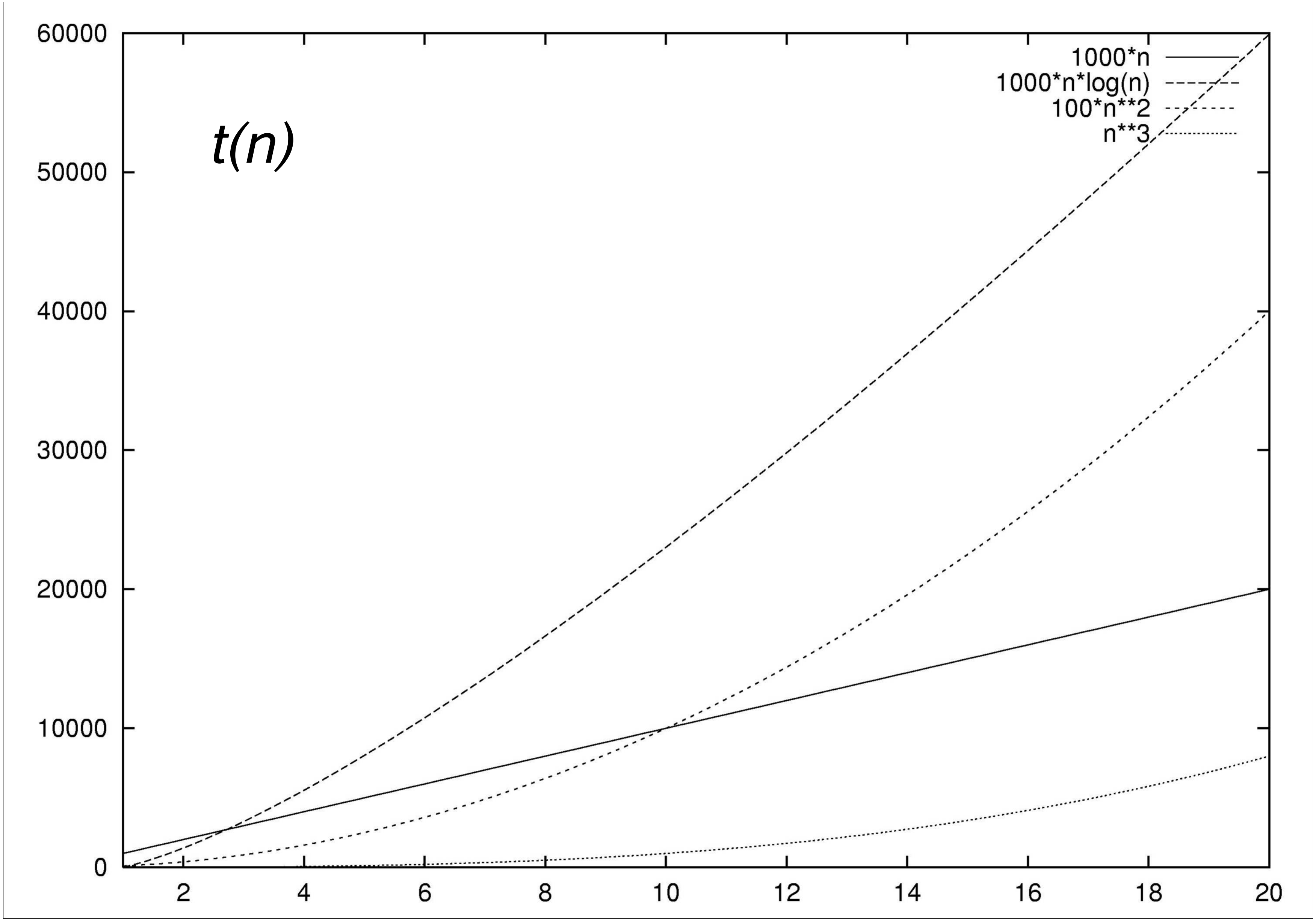
As you see, if input size *n* grows, the practical usability of your algorithm depends entirely on its *complexity*.

Unfortunately, this fact is often ignored in the software industry.

“Let's just go and buy a faster machine ...”

Note: The only thing that changes as a result of a faster machine (e.g. executing *two* operations per microsecond) is a *constant factor* in the running time. But as *n* grows, it's the *asymptotic complexity* that matters.


t(n)
1000*n
1000*n*log(n)
100*n**2
n**3
1.4e+08
1.2e+08
1e+08
8e+07
6e+07
4e+07
2e+07
0
50
100
150
200
250
300
350
400
450
500


t(n)
1000*n
1000*n*log(n)
100*n**2
n**3
0
10000
20000
30000
40000
50000
60000
2
4
6
8
10
12
14
16
18
20

What is the maximum tolerable *n* ?

Suppose, our machine can execute f operations per second (in the example: $f = 10^6$). Let the algorithm require $t(n)$ *operations to solve a problem of size* n.
Then the wall clock execution time $T(n)$ is

$$T(n)=t(n)/f \quad \text{[sec]}.$$

If the computation needs to have finished after s operations, the input size is limited to

$$n \leq t^{-1}(s \cdot f)$$

(where we assume that $t(n)$ is a strictly growing function).

Remark: This shows us the effect of increasing the processor frequency ***f***, as you do by buying a faster machine:

Example: Let the time complexity be $t(n)=n^2$ and let s be the maximum tolerable time for the computation. Using a machine twice (1000 times) as fast, the allowable input size n increases by **only a factor of 1.414 (31.62)**.

In the case of $t(n)=2^n$, n can be allowed to grow by only a *constant* of ***log*(2)=1 (**by $\lfloor$**log(1000)**$\rfloor$**=9) !**

Hence, if your algorithm is too complex, the benefit of a faster machine diminishes.

Goals of this course:

- Introduction to formalisms and terminology for algorithm design
- Formalization of algorithmic problems
- Fundamental techniques in algorithm design
- Algorithms for standard problems
- Techniques for analyzing time and space complexity
- Primitive and higher data structures
- Homework assignments and practice sessions
- Hints on implementation and other practical issues