Computer Science II

Course at D-BAUG, ETH Zurich

Felix Friedrich & Hermann Lehner

FS 2018

1. Introduction

Algorithms and Data Structures, a First Example

Goals of the course

- Understand the design and analysis of fundamental algorithms and data structures.
- Basics about design and implementation of databases.

Contents

data structures / algorithms

The notion invariant, cost model, Landau notation

algorithms design, induction

searching, selection and sorting

dictionaries: hashing and search trees

dynamic programming

graphs, shortest paths, backtracking, maximum flow

Software Engineering

Files and Exceptions

Java Streams API

Databases

ER model, relational model, SQL

1.1 Algorithms

[Cormen et al, Kap. 1;Ottman/Widmayer, Kap. 1.1]

Algorithm

Algorithm: well defined computing procedure to compute *output* data from *input* data

example problem

Input : A sequence of n numbers $(a_1, a_2, \ldots, a_n)$
Output : Permutation $(a'_1, a'_2, \ldots, a'_n)$ of the sequence $(a_i)_{1 \leq i \leq n}$, such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Possible input

$(1, 7, 3)$, $(15, 13, 12, -0.5)$, (1) ...

Every example represents a *problem instance*

The performance (speed) of an algorithm usually depends on the problem instance. Often there are “good” and “bad” instances.

Examples for algorithmic problems

- Tables and statistis: sorting, selection and searching
- routing: shortest path algorithm, heap data structure
- DNA matching: Dynamic Programming
- fabrication pipeline: Topological Sorting
- autocompletion and spell-checking: Dictionaries / Trees
- Symboltables (compiler) : Hash-Tables
- The travelling Salesman: Dynamic Programming, Minimum Spanning Tree, Simulated Annealing
- Drawing at the computer: Digitizing lines and circles, filling polygons
- Page-Rank: (Markov-Chain) Monte Carlo ...

Characteristics

- Extremely large number of potential solutions
- Practical applicability

Data Structures

- Organisation of the data tailored towards the algorithms that operate on the data.
- Programs = algorithms + data structures.

A dream

- If computers were infinitely fast and had an infinite amount of memory ...
- ... then we would still need the theory of algorithms (only) for statements about correctness (and termination).

The reality

Resources are bounded and not free:

- Computing time $\rightarrow$ Efficiency
- Storage space $\rightarrow$ Efficiency

1.2 Ancient Egyptian Multiplication

Ancient Egyptian Multiplication

Ancient Egyptian Multiplication[2]

Compute $11 \cdot 9$

11	9
~~22~~	~~4~~
~~44~~	~~2~~
88	1
99	$-$

9	11
18	5
~~36~~	~~2~~
72	1
99	

1. Double left, integer division by 2 on the right
2. Even number on the right $\Rightarrow$ eliminate row.
3. Add remaining rows on the left.

[2] Also known as russian multiplication

Advantages

- Short description, easy to grasp
- Efficient to implement on a computer: double = left shift, divide by 2 = right shift

Beispiel

left shift $9 = 01001_2 \rightarrow 10010_2 = 18$
right shift $9 = 01001_2 \rightarrow 00100_2 = 4$

Questions

- Does this always work (negative numbers?)?
- If not, when does it work?
- How do you prove correctness?
- Is it better than the school method?
- What does “good” mean at all?
- How to write this down precisely?

Observation

If $b > 1$, $a \in \mathbb{Z}$, then:

$$a \cdot b = \begin{cases} 2a \cdot \frac{b}{2} & \text{falls } b \text{ gerade,} \\ a + 2a \cdot \frac{b-1}{2} & \text{falls } b \text{ ungerade.} \end{cases}$$

Termination

$$a \cdot b = \begin{cases} a & \text{falls } b = 1, \\ 2a \cdot \frac{b}{2} & \text{falls } b \text{ gerade,} \\ a + 2a \cdot \frac{b-1}{2} & \text{falls } b \text{ ungerade.} \end{cases}$$

Recursively, Functional

$$f(a,b) = \begin{cases} a & \text{falls } b = 1, \\ f(2a, \frac{b}{2}) & \text{falls } b \text{ gerade,} \\ a + f(2a, \frac{b-1}{2}) & \text{falls } b \text{ ungerade.} \end{cases}$$

Implemented

```
// pre: b>0
// post: return a*b
int f(int a, int b){
    if(b==1)
        return a;
    else if (b%2 == 0)
        return f(2*a, b/2);
    else
        return a + f(2*a, (b-1)/2);
}
```

Correctnes

$$f(a,b) = \begin{cases} a & \text{if } b = 1, \\ f(2a, \frac{b}{2}) & \text{if } b \text{ even}, \\ a + f(2a \cdot \frac{b-1}{2}) & \text{if } b \text{ odd}. \end{cases}$$

Remaining to show: $f(a,b) = a \cdot b$ for $a \in \mathbb{Z}$, $b \in \mathbb{N}^+$.

Proof by induction

Base clause: $b = 1 \Rightarrow f(a,b) = a = a \cdot 1$.
Hypothesis: $f(a,b') = a \cdot b'$ für $0 < b' \leq b$
Step: $f(a, b+1) \overset{!}{=} a \cdot (b+1)$

$$f(a, b+1) = \begin{cases} f(2a, \overbrace{\frac{b+1}{2}}^{\leq b}) = a \cdot (b+1) & \text{if } b \text{ odd}, \\ a + f(2a, \underbrace{\frac{b}{2}}_{\leq b}) = a + a \cdot b & \text{if } b \text{ even}. \end{cases}$$

■

Recursion vs. Iteration

```
// pre: b>0
// post: return a*b
int f(int a, int b){
  if(b==1)
    return a;
  else if (b%2 == 0)
    return f(2*a, b/2);
  else
    return a + f(2*a, (b-1)/2);
}
```

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0){
      res += a;
      --b;
    }
    a *= 2;
    b /= 2;
  }
  return res;
}
```


Invariants!

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0){
      res += a;
      --b;
    }
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Sei $x := a \cdot b$.

here: $x = \boxed{a \cdot b + res}$

if here $x = a \cdot b + res$ ...

... then also here $x = a \cdot b + res$
b even

here: $x = a \cdot b + res$
here: $x = a \cdot b + res$ und $b = 0$
Also $res = x$.

Conclusion

The expression $a \cdot b + res$ is an *invariant*

- Values of a, b, res change but the invariant remains basically unchanged
- The invariant is only temporarily discarded by some statement but then re-established
- If such short statement sequences are considered atomiv, the value remains indeed invariant
- In particular the loop contains an invariant, called *loop invariant* and operates there like the induction step in induction proofs.
- Invariants are obviously powerful tools for proofs!

Analysis

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0){
      res += a;
      --b;
    }
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Ancient Egyptian Multiplication corresponds to the school method with radix 2.

$$
\begin{array}{ccccccccl}
1 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & & 1 & 0 & 0 & 1 \quad (9) \\
 & & & & 1 & 0 & 0 & 1 & \quad (18) \\
\hline
 & & & & 1 & 1 & 0 & 1 & 1 \\
 & & 1 & 0 & 0 & 1 & & & \quad (72) \\
\hline
 & & 1 & 1 & 0 & 0 & 0 & 1 & 1 \quad (99)
\end{array}
$$

Efficiency

Question: how long does a multiplication of a and b take?

- Measure for efficiency
  - Total number of fundamental operations: double, divide by 2, shift, test for “even”, addition
  - In the recursive and recursive code: maximally 6 operations per call or iteration, respectively
- Essential criterion:
  - Number of recursion calls or
  - Number iterations (in the iterative case)
- $\frac{b}{2^n} \leq 1$ holds for $n \geq \log_2 b$. Consequently not more than $6\lceil \log_2 b \rceil$ fundamental operations.

2. Efficiency of algorithms

Efficiency of Algorithms, Random Access Machine Model, Function Growth, Asymptotics [Cormen et al, Kap. 2.2,3,4.2-4.4 | Ottman/Widmayer, Kap. 1.1]

Efficiency of Algorithms

Goals

- Quantify the runtime behavior of an algorithm independent of the machine.
- Compare efficiency of algorithms.
- Understand dependece on the input size.

Technology Model

Random Access Machine (RAM)

- Execution model: instructions are executed one after the other (on one processor core).
- Memory model: constant access time.
- Fundamental operations: computations ($+$,$-$,$\cdot$,...) comparisons, assignment / copy, flow control (jumps)
- Unit cost model: fundamental operations provide a cost of 1.
- Data types: fundamental types like size-limited integer or floating point number.

Asymptotic behavior

An exact running time can normally not be predicted even for small input data.

- We consider the asymptotic behavior of the algorithm.
- And ignore all constant factors.

Example

An operation with cost 20 is no worse than one with cost 1
Linear growth with gradient 5 is as good as linear growth with gradient 1.

2.2 Function growth

$\mathcal{O}$, Θ, Ω [Cormen et al, Kap. 3; Ottman/Widmayer, Kap. 1.1]

Superficially

Use the asymptotic notation to specify the execution time of algorithms.

We write $\Theta(n^2)$ and mean that the algorithm behaves for large n like n^2: when the problem size is doubled, the execution time multiplies by four.

More precise: asymptotic upper bound

provided: a function $g : \mathbb{N} \to \mathbb{R}$.

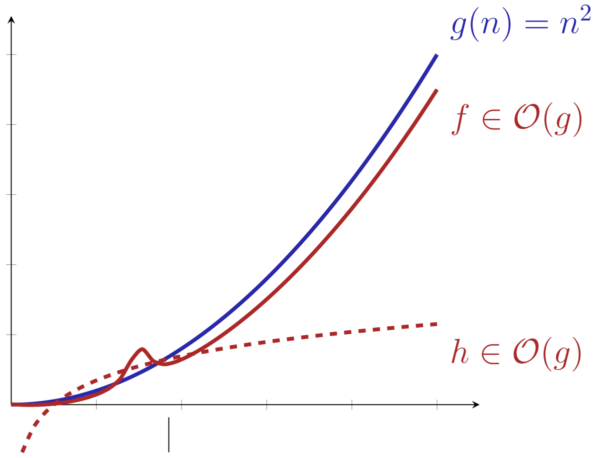
Definition:

$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} | \\ \exists c > 0, n_0 \in \mathbb{N} : 0 \leq f(n) \leq c \cdot g(n) \ \forall n \geq n_0\}$$

Notation:

$$\mathcal{O}(g(n)) := \mathcal{O}(g(\cdot)) = \mathcal{O}(g).$$

Graphic



Examples

$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} | \ \exists c > 0, n_0 \in \mathbb{N} : 0 \leq f(n) \leq c \cdot g(n) \ \forall n \geq n_0\}$$

$f(n)$	$f \in \mathcal{O}(?)$	Example
$3n + 4$	$\mathcal{O}(n)$	$c = 4, n_0 = 4$
$2n$	$\mathcal{O}(n)$	$c = 2, n_0 = 0$
$n^2 + 100n$	$\mathcal{O}(n^2)$	$c = 2, n_0 = 100$
$n + \sqrt{n}$	$\mathcal{O}(n)$	$c = 2, n_0 = 1$

Property

$$f_1 \in \mathcal{O}(g), f_2 \in \mathcal{O}(g) \Rightarrow f_1 + f_2 \in \mathcal{O}(g)$$

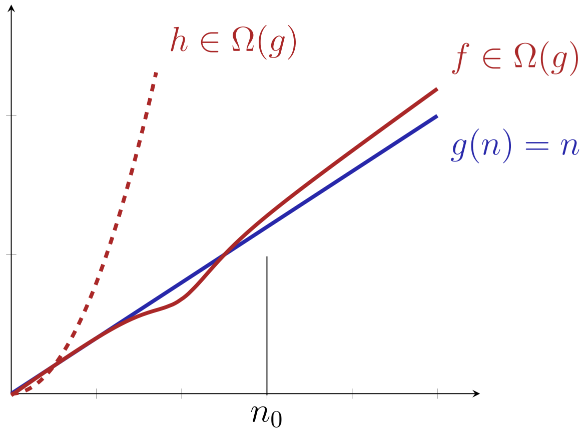
Converse: asymptotic lower bound

Given: a function $g : \mathbb{N} \to \mathbb{R}$.

Definition:

$$\Omega(g) = \{f : \mathbb{N} \to \mathbb{R} \mid \exists c > 0, n_0 \in \mathbb{N} : 0 \leq c \cdot g(n) \leq f(n)\ \forall n \geq n_0\}$$

Example



Asymptotic tight bound

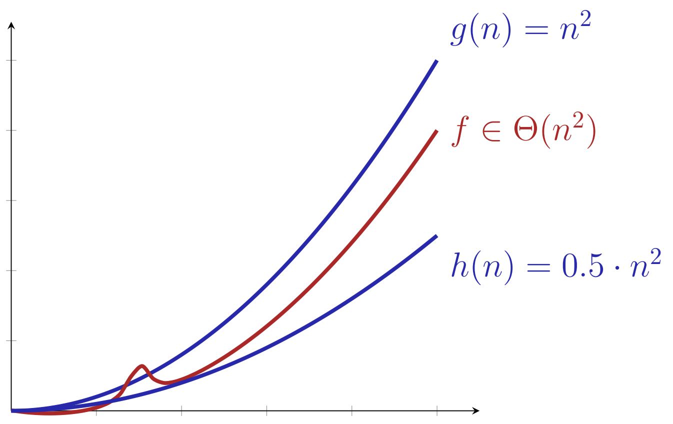
Given: function $g : \mathbb{N} \to \mathbb{R}$.

Definition:

$$\Theta(g) := \Omega(g) \cap \mathcal{O}(g).$$

Simple, closed form: exercise.

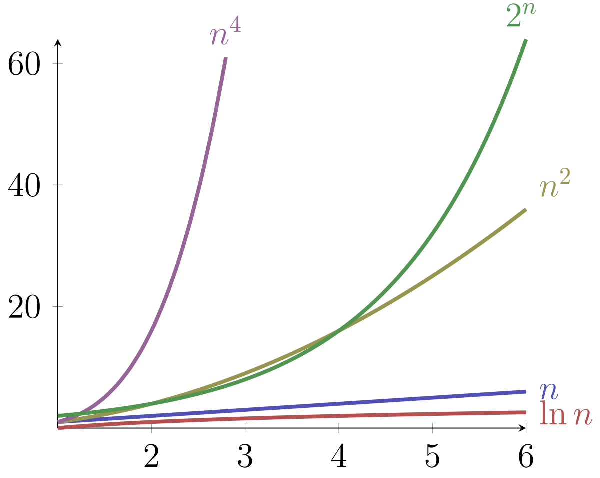
Example



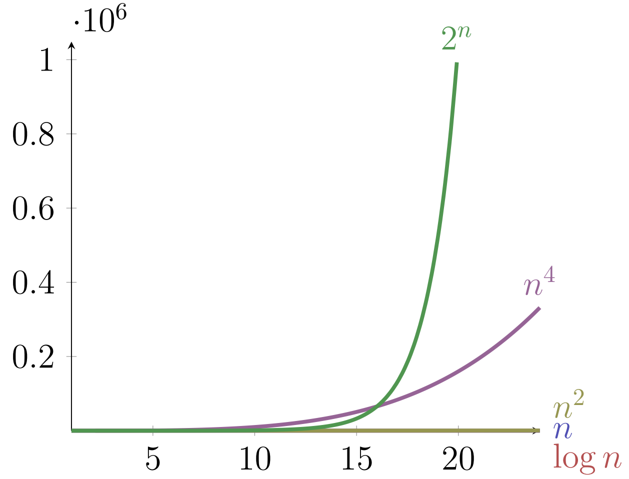
Notions of Growth

$\mathcal{O}(1)$	bounded	array access
$\mathcal{O}(\log\log n)$	double logarithmic	interpolated binary sorted sort
$\mathcal{O}(\log n)$	logarithmic	binary sorted search
$\mathcal{O}(\sqrt{n})$	like the square root	naive prime number test
$\mathcal{O}(n)$	linear	unsorted naive search
$\mathcal{O}(n \log n)$	superlinear / loglinear	good sorting algorithms
$\mathcal{O}(n^2)$	quadratic	simple sort algorithms
$\mathcal{O}(n^c)$	polynomial	matrix multiply
$\mathcal{O}(2^n)$	exponential	Travelling Salesman Dynamic Programming
$\mathcal{O}(n!)$	factorial	Travelling Salesman naively

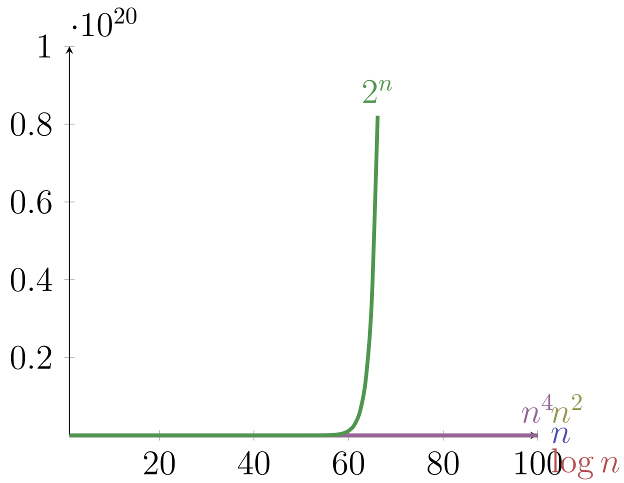
Small n



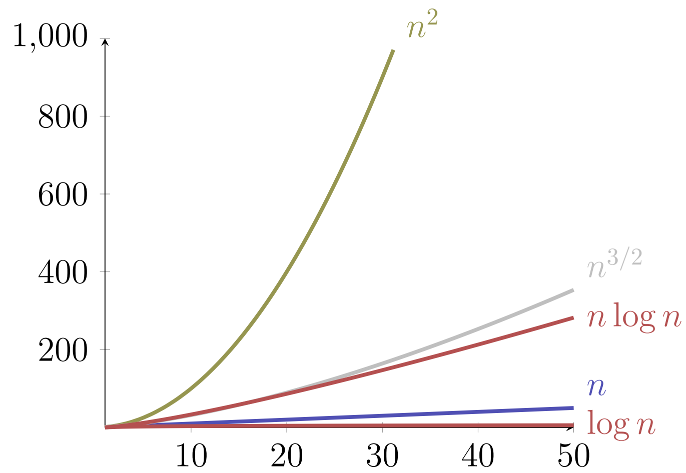
Larger n



“Large” n



Logarithms



Examples

- $n \in \mathcal{O}(n^2)$ correct, but too imprecise:
  $n \in \mathcal{O}(n)$ and even $n \in \Theta(n)$.
- $3n^2 \in \mathcal{O}(2n^2)$ correct but uncommon:
  Omit constants: $3n^2 \in \mathcal{O}(n^2)$.
- $2n^2 \in \mathcal{O}(n)$ is wrong: $\frac{2n^2}{cn} = \frac{2}{c}n \underset{n\to\infty}{\to} \infty$!
- $\mathcal{O}(n) \subseteq \mathcal{O}(n^2)$ is correct
- $\Theta(n) \subseteq \Theta(n^2)$ is wrong $n \notin \Omega(n^2) \supset \Theta(n^2)$

Useful Tool

Theorem

Let $f, g : \mathbb{N} \to \mathbb{R}^+$ be two functions, then it holds that

1. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in \mathcal{O}(g)$, $\mathcal{O}(f) \subsetneq \mathcal{O}(g)$.
2. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C > 0$ (C *constant*) $\Rightarrow f \in \Theta(g)$.
3. $\frac{f(n)}{g(n)} \underset{n\to\infty}{\to} \infty \Rightarrow g \in \mathcal{O}(f)$, $\mathcal{O}(g) \subsetneq \mathcal{O}(f)$.

About the Notation

Common notation

$$f = \mathcal{O}(g)$$

should be read as $f \in \mathcal{O}(g)$.

Clearly it holds that

$$f_1 = \mathcal{O}(g), f_2 = \mathcal{O}(g) \not\Rightarrow f_1 = f_2!$$

Beispiel

$n = \mathcal{O}(n^2), n^2 = \mathcal{O}(n^2)$ but naturally $n \neq n^2$.

Algorithms, Programs and Execution Time

Program: concrete implementation of an algorithm.

Execution time of the program: measurable value on a concrete machine. Can be bounded from above and below.

Beispiel

3GHz computer. Maximal number of operations per cycle (e.g. 8). $\Rightarrow$ lower bound.
A single operations does never take longer than a day $\Rightarrow$ upper bound.

From an *asymptotic* point of view the bounds coincide.