ETH zürich

Felix Friedrich

Data Structures and Algorithms

Course at D-MATH (CSE) of ETH Zurich

Spring 2021

1. Introduction

Overview, Algorithms and Data Structures, Correctness, First Example

Goals of the course

- Understand the design and analysis of fundamental algorithms and data structures.
- An advanced insight into a modern programming model (with C++).
- Knowledge about chances, problems and limits of the parallel and concurrent computing.

Contents

data structures / algorithms

The notion invariant, cost model, Landau notation
algorithms design, induction
searching, selection and sorting
amortized analysis
dynamic programming
dictionaries: hashing and search trees
Fundamental algorithms on graphs,
shortest paths, Minimum Spanning Tree, Max-Flow
Minimum Spanning Trees, Fibonacci Heaps

prorgamming with C++

RAII, Move Konstruktion, Smart Pointers,
Templates and generic programming
Exceptions
functors and lambdas
threads, mutex and monitors
promises and futures

parallel programming

parallelism vs. concurrency, speedup (Amdahl/Gustavson), races, memory reordering, atomic registers, RMW (CAS,TAS), deadlock/starvation

1.2 Algorithms

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

Algorithm

Algorithm
Well-defined procedure to compute output data from input data

Example Problem: Sorting

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

Possible input

$(1, 7, 3)$, $(15, 13, 12, -0.5)$, $(999, 998, 997, 996, \dots, 2, 1)$, (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.

Therefore we consider algorithms sometimes ”in the average“ and most often in the ”worst case“.

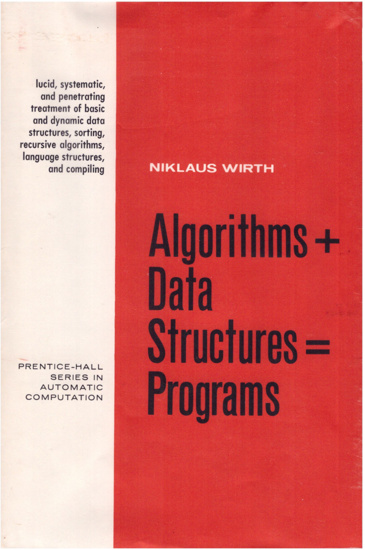
Possible solution

How many times are the lines executed each?

```
void sort(std::vector<int> a){
  std::size n = a.size()
  for (std::size i = 0; i<n ; ++i)
    for (std::size j = i+1; j<n; ++j)
      if (a[j] < a[i])
        std::swap(a[i],a[j])
}
```

Data Structures

- A data structure is a particular way of organizing data in a computer so that they can be used efficiently (in the algorithms operating on them).
- Programs = algorithms + data structures.



Examples for algorithmic problems

- Tables and statistis: sorting, selection and searching
- routing: shortest path algorithm, heap data structure
- DNA matching: Dynamic Programming
- evaluation order: Topological Sorting
- autocompletion and spell-checking: Dictionaries / Trees
- fast lookup : Hash-Tables
- the Travelling Salesman: Dynamic Programming, Minimum Spanning Tree, Simulated Annealing

Characteristics

- Extremely large number of potential solutions
- Practical applicability

Typical Design Steps Example

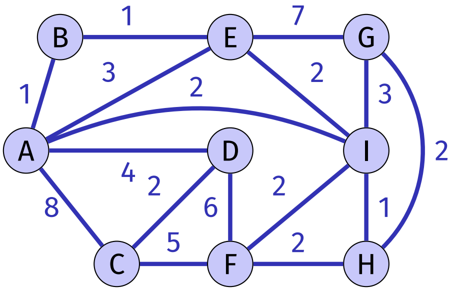
Route planning

Typical Design Steps

1. Specification of the problem: find best (shortest time) path from A to B
2. Abstraction: graph with nodes, edges and egde-weights
3. Idea (heureka!): Dijkstra
4. Data-structures and algorithms: e.g. adjacency matrix / adjacency list, min-heap, hash-table ...
5. Runtime analysis: $\mathcal{O}((n+m)\cdot\log n)$
6. Implementation: Representation choice (e.g. adjacency matrix/ adjacency list/ objects)

Difficult Problem: Travelling Salesman

Given: graph (map) with nodes (cities) and weighted edges (roads with length)
Wanted: Loop road through all cities such that each city is visited once (Hamilton-cycle) with minimal overall length.



The best known algorithm has a running time that increase exponentially with the number of nodes (cities).

Already finding a Hamilton cycle is a difficult problem in general. In contrast, the problem to find an Eulerian cycle, a cycle that uses each *edge* once, is a problem with polynomial running time.

Hard problems.

- NP-complete problems: no known efficient solution (the existence of such a solution is very improbable – but it has not yet been proven that there is none!)
- Example: travelling salesman problem

This course is *mostly* about problems that can be solved efficiently (in polynomial time).

Efficiency

- 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).

Reality: resources are bounded and not free:

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

Actually, this course is nearly only about efficiency.

2. Efficiency of algorithms

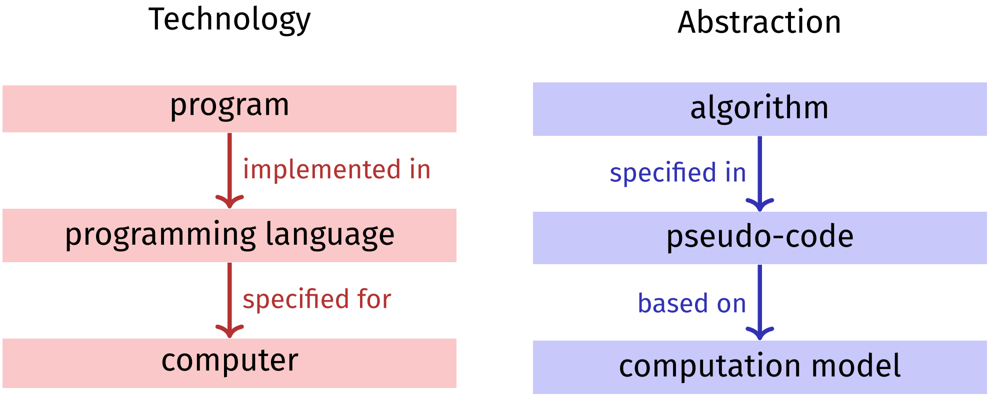
Efficiency of Algorithms, Random Access Machine Model, Function Growth, Asymptotics [Cormen et al, Kap. 2.2,3,4.2-4.4 | Ottman/Widmayer, Kap. 1.1]

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.

Programs and Algorithms



Technology Model

Random Access Machine (RAM) Model

- Execution model: instructions are executed one after the other (on one processor core).
- Memory model: constant access time (big array)
- Fundamental operations: computations ($+$,$-$,$\cdot$,...) comparisons, assignment / copy on machine words (registers), 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.

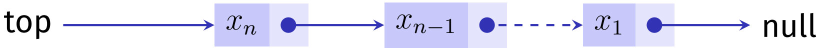
Size of the Input Data

- Typical: number of input objects (of fundamental type).
- Sometimes: number bits for a *reasonable* / *cost-effective* representation of the data.
- fundamental types fit into word of size : $w \geq \log(\text{sizeof(mem)})$ bits.

For Dynamic Data Strcutures

Pointer Machine Model

- Objects bounded in size can be dynamically allocated in constant time
- Fields (with word-size) of the objects can be accessed in constant time 1.



Asymptotic behavior

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

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

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.

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.

Example 1

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 the perspective of the *asymptotic behavior* of the program, the bounds are unimportant.

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 $g : \mathbb{N} \rightarrow \mathbb{R}$.
Definition:[1]

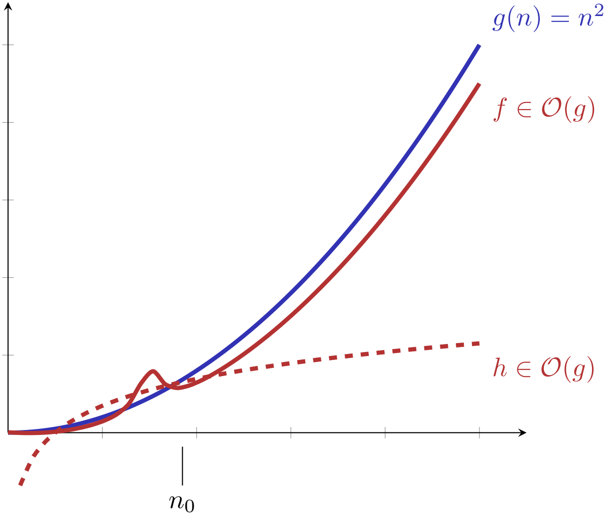
$$\begin{aligned}\mathcal{O}(g) = \{f : \mathbb{N} \rightarrow \mathbb{R} |\\ &\exists\, c > 0, \exists n_0 \in \mathbb{N} :\\ &\forall\, n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}\end{aligned}$$

Notation:

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

[1] Ausgesprochen: Set of all functions $f : \mathbb{N} \rightarrow \mathbb{R}$ that satisfy: there is some (real valued) $c > 0$ and some $n_0 \in \mathbb{N}$ such that $0 \leq f(n) \leq n \cdot g(n)$ for all $n \geq n_0$.

Graphic

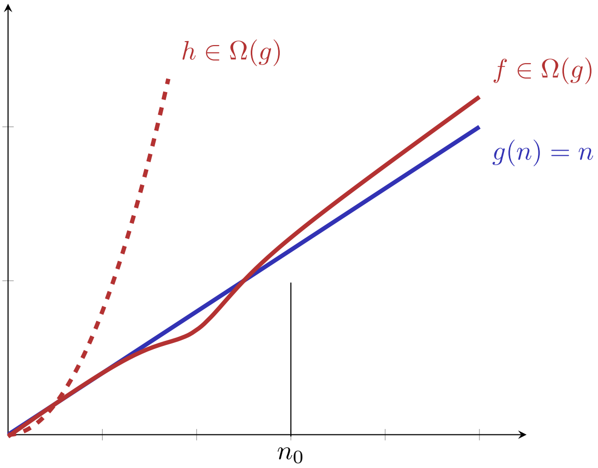


Converse: asymptotic lower bound

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

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

Example



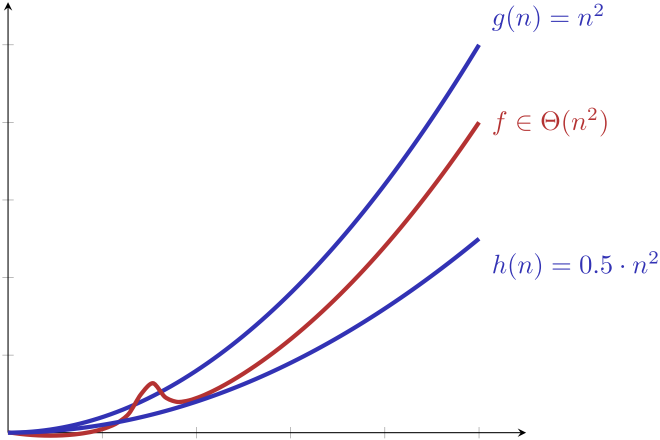
Asymptotic tight bound

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

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

Simple, closed form: exercise.

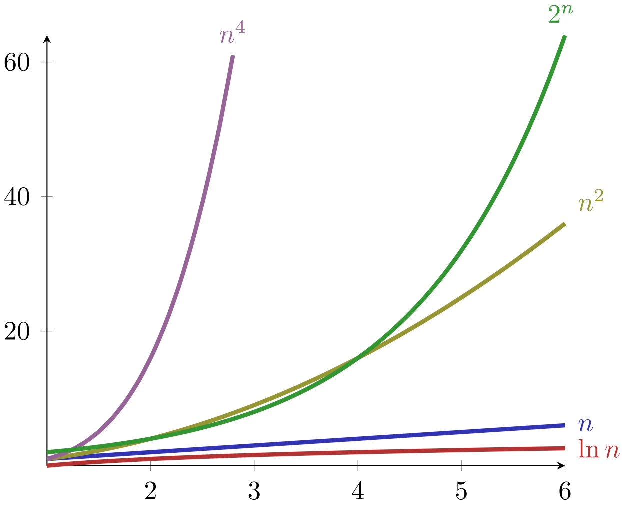
Example



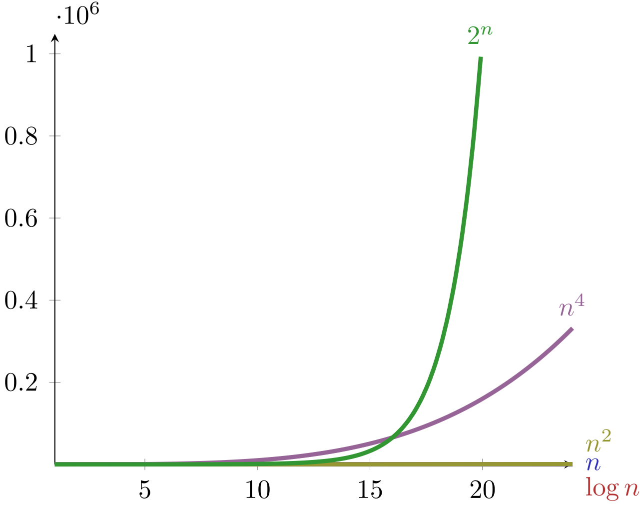
Notions of Growth

$\mathcal{O}(1)$	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
$\mathcal{O}(c^n)$	exponential	Travelling Salesman Dynamic Programming
$\mathcal{O}(n!)$	factorial	Travelling Salesman naively

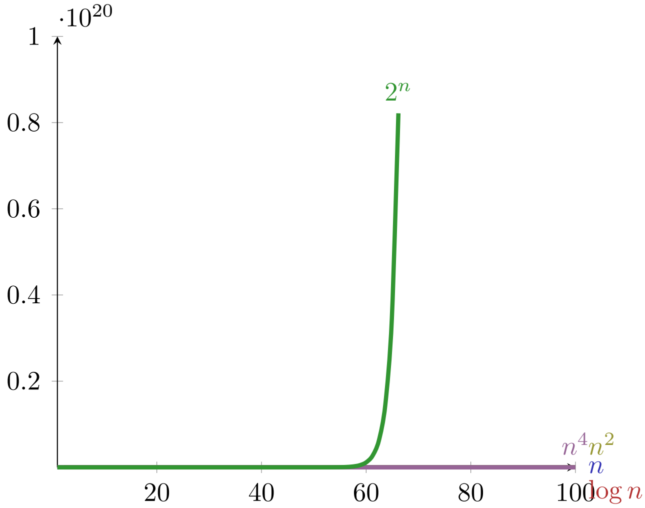
Small n



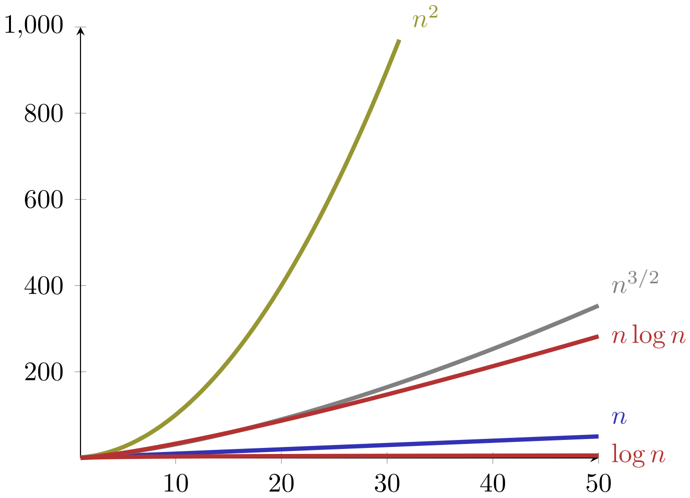
Larger n



“Large” n



Logarithms



Time Consumption

Assumption 1 Operation = $1\mu s$.

problem size	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$	$7\mu s$	$13\mu s$	$20\mu s$	$30\mu s$
n	$1\mu s$	$100\mu s$	$1/100s$	$1s$	17 minutes
$n \log_2 n$	$1\mu s$	$700\mu s$	$13/100\mu s$	$20s$	8.5 hours
n^2	$1\mu s$	$1/100s$	1.7 minutes	11.5 days	317 centuries
2^n	$1\mu s$	10^{14} centuries	$\approx \infty$	$\approx \infty$	$\approx \infty$

Useful Tool

Theorem 2

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. $\frac{f(n)}{g(n)} \underset{n\to\infty}{\longrightarrow} \infty \Rightarrow g \in \mathcal{O}(f)$, $\mathcal{O}(g) \subsetneq \mathcal{O}(f)$.

About the Notation

Common casual notation

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

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

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

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

We avoid this notation where it could lead to ambiguities.

Reminder: Efficiency: Arrays vs. Linked Lists

- Memory: our `avec` requires roughly n ints (vector size n), our `llvec` roughly $3n$ ints (a pointer typically requires 8 byte)
- Runtime (with `avec` = `std::vector`, `llvec` = `std::list`):

```
prepending (insert at front) [100,000x]:
    ► avec:     675 ms
    ► llvec:     10 ms
appending (insert at back) [100,000x]:
    ► avec:       2 ms
    ► llvec:      9 ms
removing first [100,000x]:
    ► avec:     675 ms
    ► llvec:      4 ms
removing last [100,000x]:
    ► avec:       0 ms
    ► llvec:      4 ms
```

```
removing randomly [10,000x]:
    ► avec:       3 ms
    ► llvec:    113 ms
inserting randomly [10,000x]:
    ► avec:      16 ms
    ► llvec:    117 ms
fully iterate sequentially (5000 elements) [5,000x]:
    ► avec:     354 ms
    ► llvec:    525 ms
```

Asymptotic Runtimes

With our new language ($\Omega, \mathcal{O}, \Theta$), we can now state the behavior of the data structures and their algorithms more precisely

Typical asymptotic running times (Anticipation!)

Data structure	Random Access	Insert	Next	Insert After Element	Search
`std::vector`	$\Theta(1)$	$\Theta(1)$ A	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
`std::list`	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
`std::set`	–	$\Theta(\log n)$	$\Theta(\log n)$	–	$\Theta(\log n)$
`std::unordered_set`	–	$\Theta(1)$ P	–	–	$\Theta(1)$ P

A = amortized, P=expected, otherwise worst case

Complexity

Complexity of a problem P

Minimal (asymptotic) costs over all algorithms A that solve P.

Complexity of the single-digit multiplication of two numbers with n digits is $\Omega(n)$ and $\mathcal{O}(n^{\log_3 2})$ (Karatsuba Ofman).

Complexity

Problem	Complexity	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\Omega(n\log n)$
		$\Uparrow$	$\Uparrow$	$\Uparrow$	$\Downarrow$
Algorithm	Costs[2]	$3n-4$	$\mathcal{O}(n)$	$\Theta(n^2)$	$\Omega(n\log n)$
		$\Downarrow$	$\Updownarrow$	$\Updownarrow$	$\Downarrow$
Program	Execution time	$\Theta(n)$	$\mathcal{O}(n)$	$\Theta(n^2)$	$\Omega(n\log n)$

[2] Number fundamental operations