# Data Structures and Algorithms

Course at D-MATH (CSE) of ETH Zurich

Felix Friedrich

FS 2019

## Welcome!

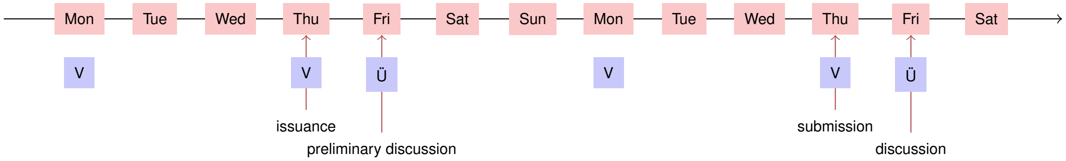
### Course homepage

`http://lec.inf.ethz.ch/DA/2019`

### The team:

| | |
|---|---|
| Assistants | Philippe Schlattner |
| | Jan Stratmann |
| | Robin Worreby |
| | Robin Vogtland |
| Back-Office | Aritra Dhar |
| | Pesho Ivanov |
| Lecturer | Felix Friedrich |

## Exercises



- Exercises availabe at lectures.
- Preliminary discussion in the following recitation session
- Solution of the exercise until the day before the next recitation session.
- Dicussion of the exercise in the next recitation session.

## Exercises

- The solution of the weekly exercises is thus voluntary but *stronly* recommended.

## It is so simple!

For the exercises we use an online development environment that requires only a browser, internet connection and your ETH login.

If you do not have access to a computer: there are a a lot of computers publicly accessible at ETH.

## literature

**Algorithmen und Datenstrukturen**, *T. Ottmann, P. Widmayer*, Spektrum-Verlag, 5. Auflage, 2011

**Algorithmen - Eine Einführung**, *T. Cormen, C. Leiserson, R. Rivest, C. Stein*, Oldenbourg, 2010

**Introduction to Algorithms**, *T. Cormen, C. Leiserson, R. Rivest, C. Stein* , 3rd ed., MIT Press, 2009

**The C++ Programming Language**, *B. Stroustrup*, 4th ed., Addison-Wesley, 2013.

**The Art of Multiprocessor Programming**, *M. Herlihy, N. Shavit*, Elsevier, 2012.

## Relevant for the exam

Material for the exam comprises

- Course content (lectures, handout)
- Exercises content (exercise sheets, recitation hours)

Written exam (120 min). Examination aids: four A4 pages (or two sheets of 2 A4 pages double sided) either hand written or with font size minimally 11 pt.

## Offer

- Doing the weekly exercise series $\rightarrow$ bonus of maximally 0.25 of a grade points for the exam.
- The bonus is proportional to the achieved points of **specially marked bonus-task**. The full number of points corresponds to a bonus of 0.25 of a grade point.
- The **admission** to the specially marked bonus tasks can depend on the successul completion of other exercise tasks. The achieved grade bonus expires as soon as the course has been given again.

## Offer (Concretely)

- 4 bonus exercises in total; 3/4 of the points suffice for the exam bonus of 0.25 marks
- You can, e.g. fully solve 3 bonus exercises, or solve 4 bonus exercises to 75% each, or ...
- Bonus exercises must be unlocked ($\rightarrow$ experience points) by successfully completing the weekly exercises
- It is again not necessary to solve all weekly exercises completely in order to unlock a bonus exercise
- Details: exercise sessions, online exercise system (Code Expert)

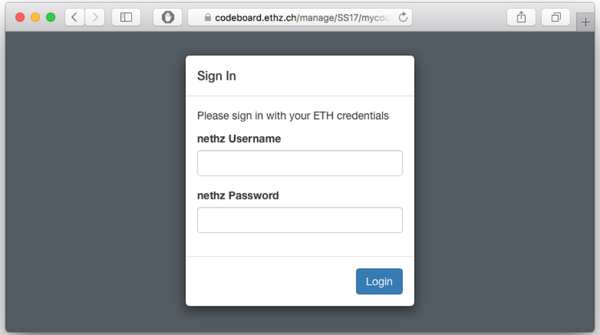
## Academic integrity

**Rule:** You submit solutions that you have written yourself and that you have understood.

We check this (partially automatically) and reserve our rights to adopt disciplinary measures.

## Exercise group registration I

- Visit `http://expert.ethz.ch/enroll/SS19/da`
- Log in with your nethz account.



Sign In

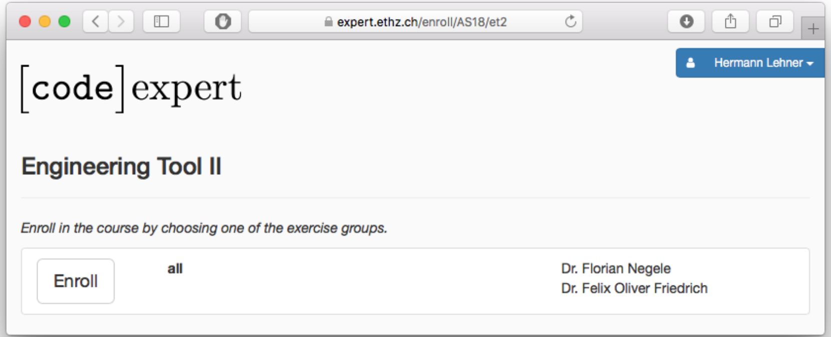
Please sign in with your ETH credentials

nethz Username

nethz Password

## Exercise group registration II

Register with the subsequent dialog for an exercise group.

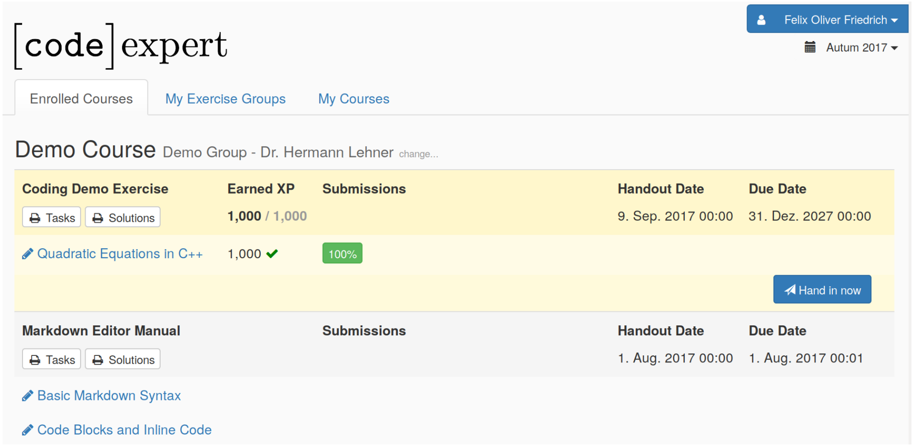


Engineering Tool II

*Enroll in the course by choosing one of the exercise groups.*

| | | |
|---|---|---|
| Enroll | **all** | Dr. Florian Negele<br>Dr. Felix Oliver Friedrich |

## Overview



[code] expert

Felix Oliver Friedrich

Autum 2017

Enrolled Courses | My Exercise Groups | My Courses

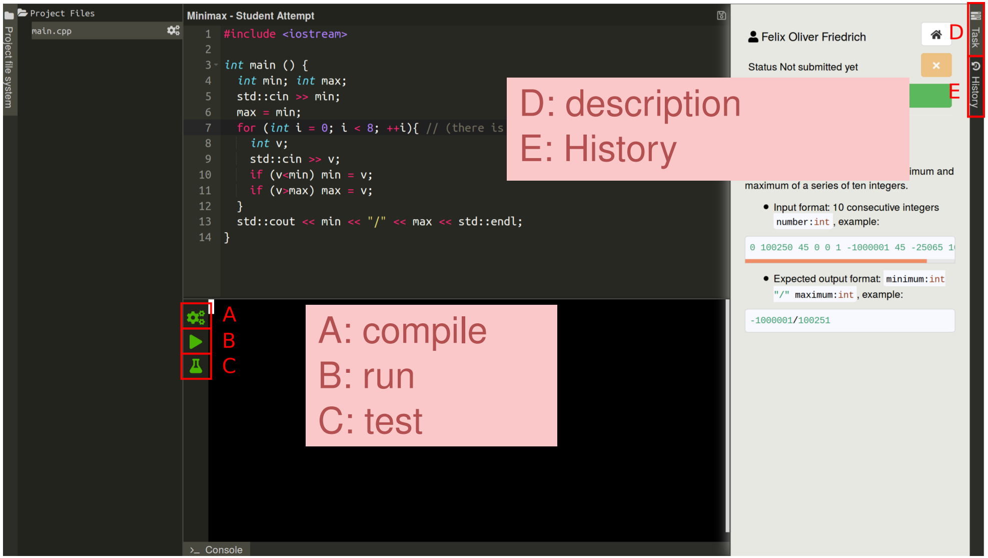
Demo Course Demo Group - Dr. Hermann Lehner change...

| Coding Demo Exercise | Earned XP | Submissions | Handout Date | Due Date |
|---|---|---|---|---|
| Tasks Solutions | **1,000** / 1,000 | | 9. Sep. 2017 00:00 | 31. Dez. 2027 00:00 |
| Quadratic Equations in C++ | 1,000 ✓ | 100% | | |

Hand in now

| Markdown Editor Manual | Submissions | Handout Date | Due Date |
|---|---|---|---|
| Tasks Solutions | | 1. Aug. 2017 00:00 | 1. Aug. 2017 00:01 |
| Basic Markdown Syntax | | | |
| Code Blocks and Inline Code | | | |

# Programming Exercise



```
#include <iostream>

int main () {
  int min; int max;
  std::cin >> min;
  max = min;
  for (int i = 0; i < 8; ++i){ // (there is
    int v;
    std::cin >> v;
    if (v<min) min = v;
    if (v>max) max = v;
  }
  std::cout << min << "/" << max << std::endl;
}
```

Felix Oliver Friedrich

Status Not submitted yet

imum and maximum of a series of ten integers.

- Input format: 10 consecutive integers `number:int`, example:

```
0 100250 45 0 0 1 -1000001 45 -25065 1
```

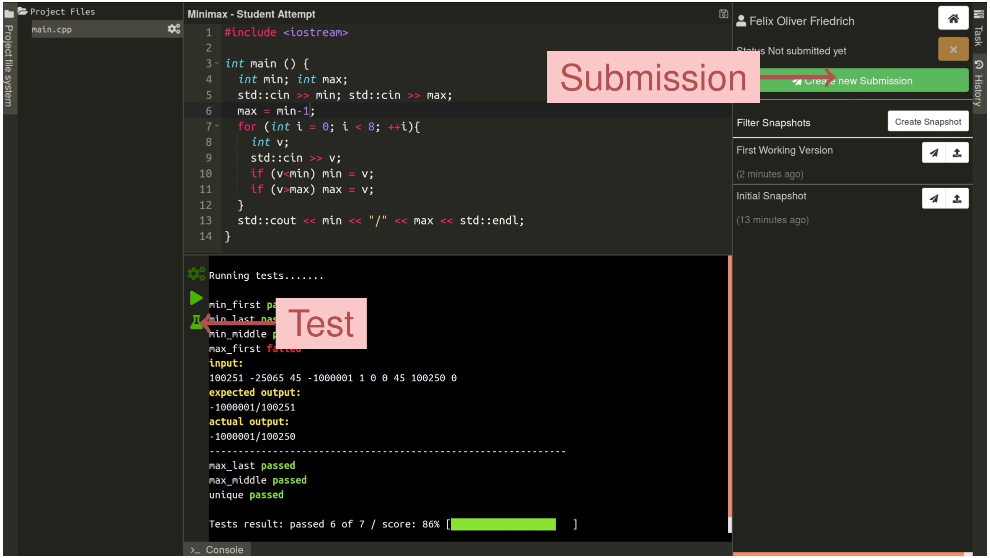
- Expected output format: `minimum:int` `"/"` `maximum:int`, example:

```
-1000001/100251
```

D: description
E: History

A: compile
B: run
C: test

# Test and Submit



```
#include <iostream>

int main () {
  int min; int max;
  std::cin >> min; std::cin >> max;
  max = min-1;
  for (int i = 0; i < 8; ++i){
    int v;
    std::cin >> v;
    if (v<min) min = v;
    if (v>max) max = v;
  }
  std::cout << min << "/" << max << std::endl;
}
```

```
Running tests.......

min_first pa
min_last pas
min_middle p
max_first failed
input:
100251 -25065 45 -1000001 1 0 0 45 100250 0
expected output:
-1000001/100251
actual output:
-1000001/100250
--------------------------------------------------------------
max_last passed
max_middle passed
unique passed

Tests result: passed 6 of 7 / score: 86% [                ]
```

Status Not submitted yet

Create new Submission

Filter Snapshots | Create Snapshot

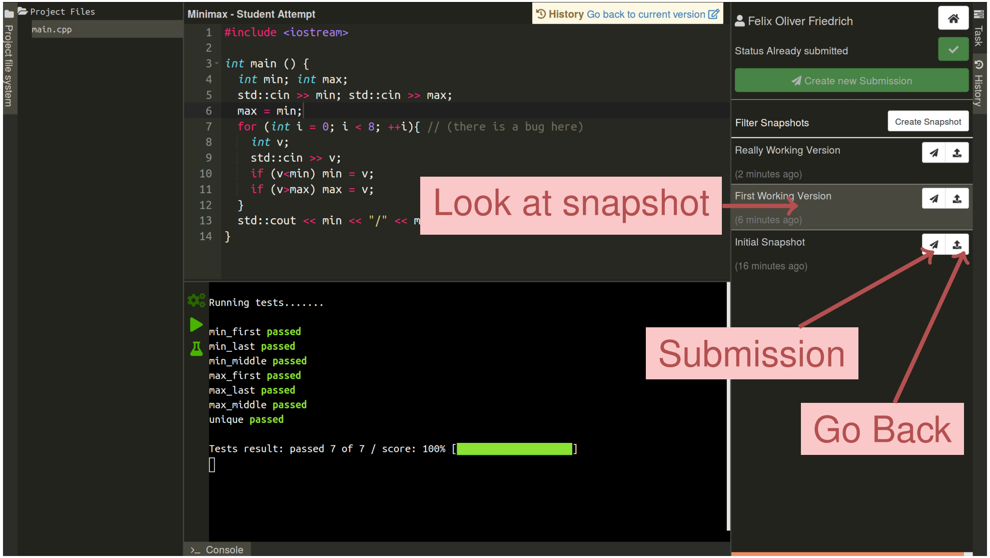
First Working Version
(2 minutes ago)

Initial Snapshot
(13 minutes ago)

## Where is the Save Button?

- The file system is transaction based and is saved permanently (“autosave”). When opening a project it is found in the most recent observed state.
- The current state can be saved as (named) *snaphot*. It is always possible to return to saved snapshot.
- The current state can be submitted (as snapshot). Additionally, each saved named snapshot can be submitted.

# Snapshots



```
#include <iostream>

int main () {
  int min; int max;
  std::cin >> min; std::cin >> max;
  max = min;
  for (int i = 0; i < 8; ++i){ // (there is a bug here)
    int v;
    std::cin >> v;
    if (v<min) min = v;
    if (v>max) max = v;
  }
  std::cout << min << "/" << m
}
```

```
Running tests.......

min_first passed
min_last passed
min_middle passed
max_first passed
max_last passed
max_middle passed
unique passed

Tests result: passed 7 of 7 / score: 100% [                    ]
```

## Should there be any Problems ...

- with the course content
    - definitely attend all recitation sessions
    - ask questions there
    - and/or contact the assistant
- further problems
    - Email to lecturer (Felix Friedrich)
- We are willing to help.

# 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

van-Emde Boas Trees, Splay-Trees

Fundamental algorithms on graphs,

shortest paths, 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, atomir registers, RMW (CAS,TAS), deadlock/starvation

## 1.2 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 $(a_1, a_2, \dots, a_n)$

**Output**: Permutation $(a'_1, a'_2, \dots, a'_n)$ of the sequence $(a_i)_{1 \le i \le n}$, such that $a'_1 \le a'_2 \le \dots \le a'_n$

**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

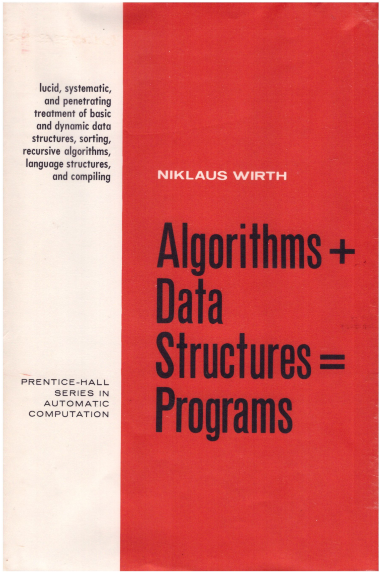
- Tables and statistis: sorting, selection and searching
- routing: shortest path algorithm, heap data structure
- DNA matching: Dynamic Programming
- evaluation order: Topological Sorting
- autocomletion and spell-checking: Dictionaries / Trees
- Fast Lookup : Hash-Tables
- The travelling Salesman: Dynamic Programming, Minimum Spanning Tree, Simulated Annealing

## Characteristics

- Extremely large number of potential solutions
- Practical applicability

## 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.


lucid, systematic, and penetrating treatment of basic and dynamic data structures, sorting, recursive algorithms, language structures, and compiling

NIKLAUS WIRTH

Algorithms + Data Structures = Programs

PRENTICE-HALL SERIES IN AUTOMATIC COMPUTATION

## Efficiency

Illusion:

- 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.**

## 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).**

# 2. Efficiency of algorithms

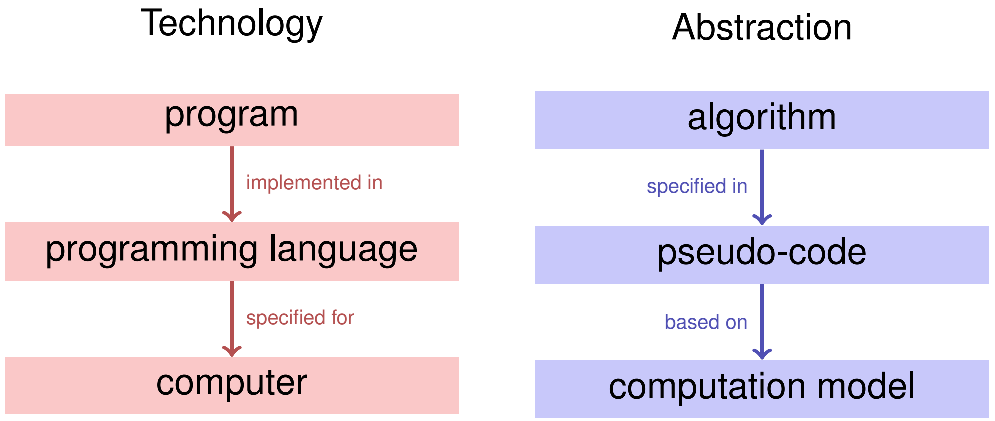
Efficiency of Algorithms, Random Access Machine Model, Function Growth, Asymptotics [Cormen et al, Kap. 2.2,3,4.2-4.4 | Ottman/Widmayer, Kap. 1.1]

## 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)*

- 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).

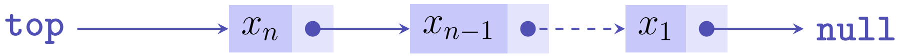
Sometimes: number bits for a *reasonable / cost-effective* representation of the data.

fundamental types fit into word of size : $w \geq \log(\text{sizeof(mem)})$ bits.

## Pointer Machine Model

We assume

- 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.

**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 $1$.

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

## 2.2 Function growth

$\mathcal{O}$, $\Theta$, $\Omega$ [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:[1]

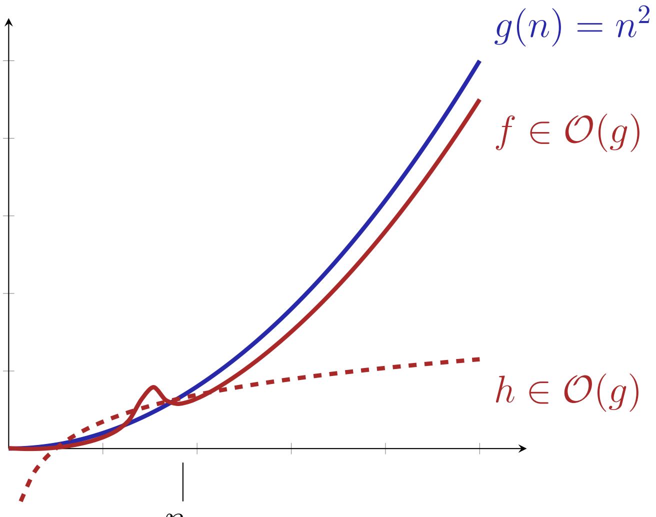
$$\begin{aligned}\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} | \\ &\exists\, c > 0, \exists n_0 \in \mathbb{N} : \\ &\forall\, n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}\end{aligned}$$

Notation:

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

[1] Ausgesprochen: Set of all functions $f : \mathbb{N} \to \mathbb{R}$ 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



## Examples

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

| $f(n)$ | $f \in \mathcal{O}(?)$ | 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

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

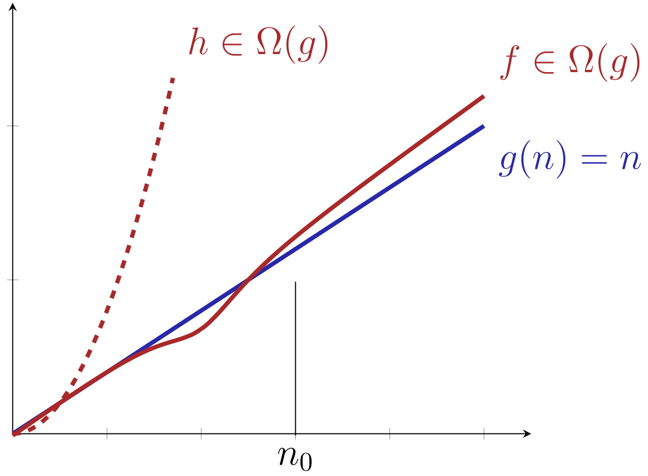
## Converse: asymptotic lower bound

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

Definition:

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

## Example



## Asymptotic tight bound

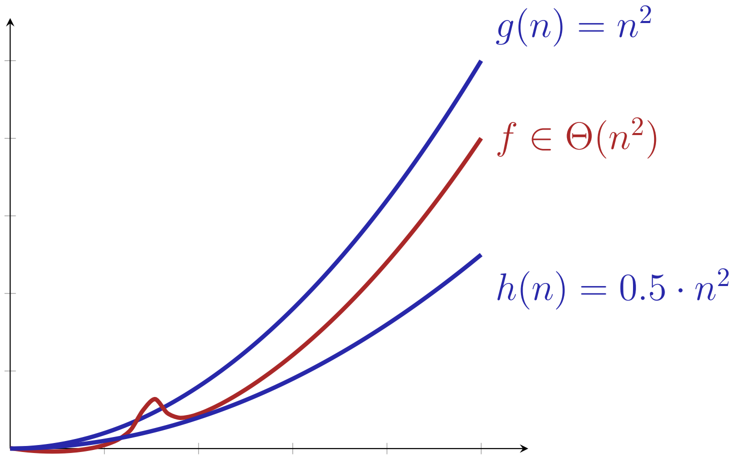
Given: function $g : \mathbb{N} \to \mathbb{R}$.

Definition:

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

Simple, closed form: exercise.

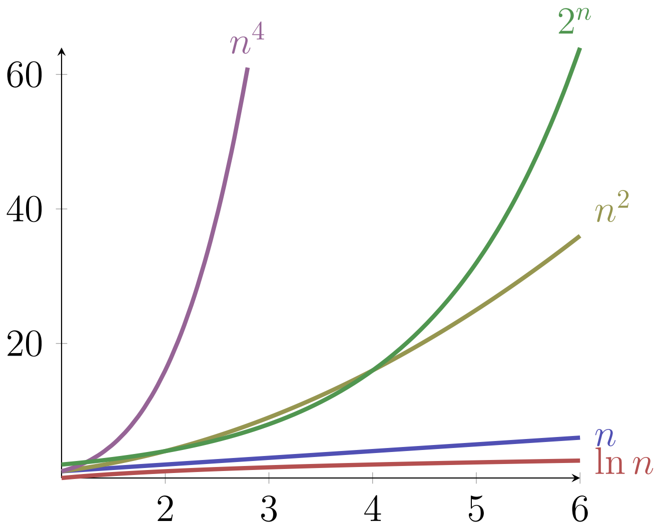
## Example



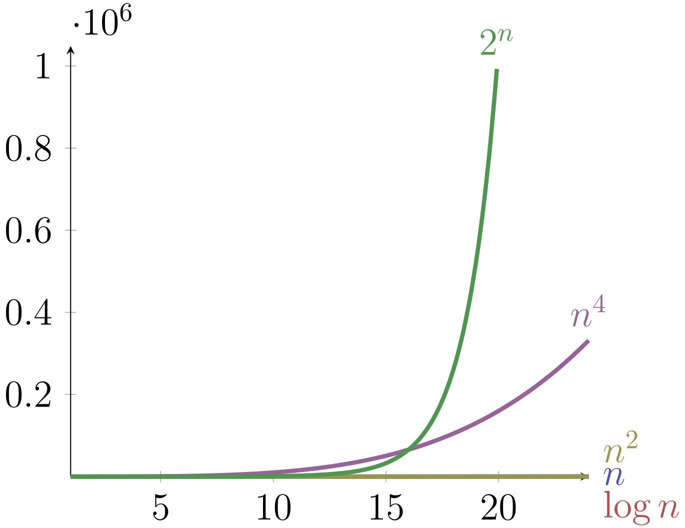
## Notions of Growth

| | | |
|---|---|---|
| $\mathcal{O}(1)$ | 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 |
| $\mathcal{O}(2^n)$ | exponential | Travelling Salesman Dynamic Programming |
| $\mathcal{O}(n!)$ | factorial | Travelling Salesman naively |

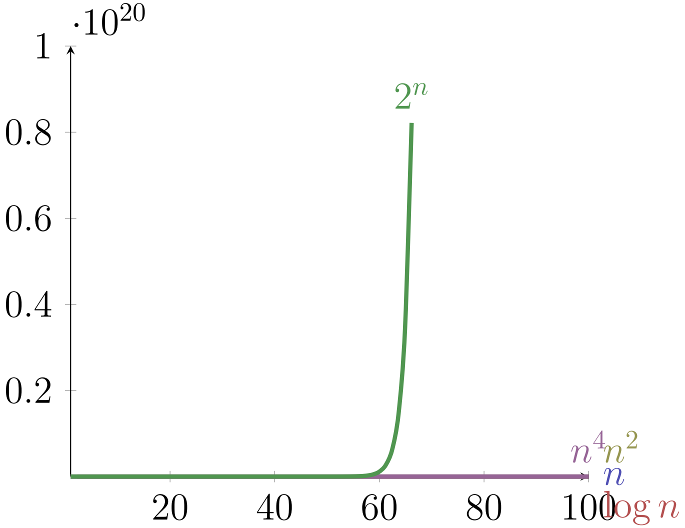
## Small $n$



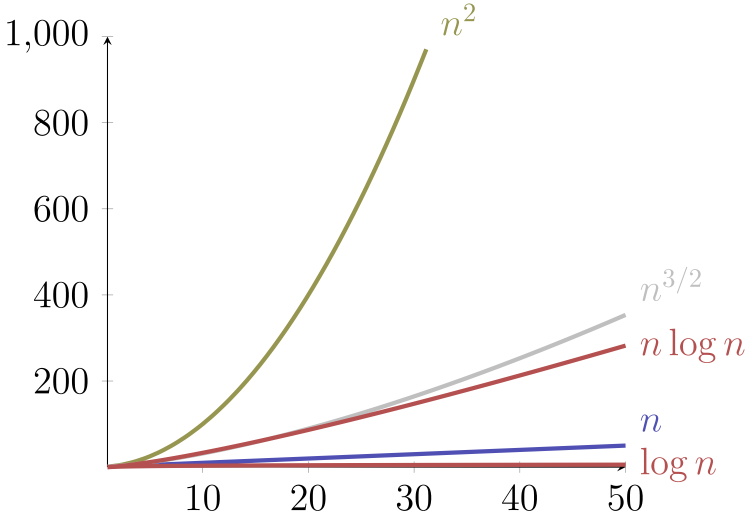
## Larger $n$



## “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**

*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

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

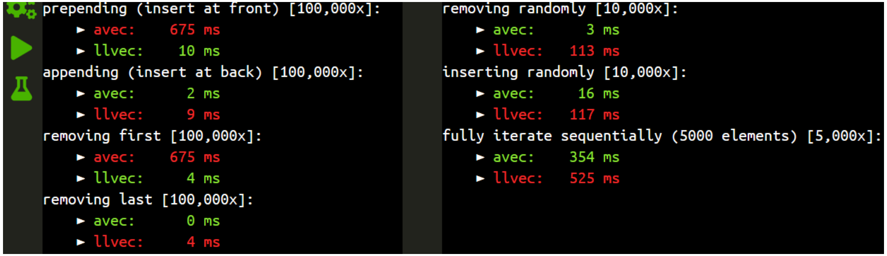
**Beispiel**

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

**Example:**

$$\begin{tikzcd}
\text{Problem} & \text{Complexity} & \mathcal{O}(n) & \mathcal{O}(n) & \mathcal{O}(n^2) \\
\text{Algorithm} & \text{Costs}^2 & 3n-4 \arrow[u] \arrow[d] & \mathcal{O}(n) \arrow[u] \arrow[d, leftrightarrow] & \Theta(n^2) \arrow[u] \arrow[d, leftrightarrow] \\
\text{Program} & \text{Execution time} & \Theta(n) & \mathcal{O}(n) & \Theta(n^2)
\end{tikzcd}$$

## 3. Examples

Show Correctness, Recursion and Recurrences
[References to literatur at the examples]

## 3.1 Ancient Egyptian Multiplication

Ancient Egyptian Multiplication– Example on how to show correctness of algorithms.

## Ancient Egyptian Multiplication[3]

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.

[3] 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* | $9 = 01001_2 \to 10010_2 = 18$ |
| *right shift* | $9 = 01001_2 \to 00100_2 = 4$ |

## Questions

- For which kind of inputs does the algorithm deliver a correct result (in finite time)?
- How do you prove its correctness?
- What is a good measure for Efficiency?

## The Essentials

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 as a function

```
// 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: Mathematical Proof

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

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

## Correctnes: Mathematical Proof by Induction

Let $a \in \mathbb{Z}$, to show $f(a,b) = a \cdot b \quad \forall\, b \in \mathbb{N}^+$.

*Base clause:* $f(a,1) = a = a \cdot 1$

*Hypothesis:* $f(a,b') = a \cdot b' \quad \forall\, 0 < b' \leq b$

*Step:* $f(a,b') = a \cdot b' \quad \forall\, 0 < b' \leq b \overset{!}{\Rightarrow} f(a,b+1) = a \cdot (b+1)$

$$f(a,b+1) = \begin{cases} f(2a, \overbrace{\frac{b+1}{2}}^{0<\cdot\leq b}) \overset{i.H.}{=} a \cdot (b+1) & \text{if } b > 0 \text{ odd,} \\ a + f(2a, \underbrace{\frac{b}{2}}_{0<\cdot<b}) \overset{i.H.}{=} a + a \cdot b & \text{if } b > 0 \text{ even.} \end{cases}$$

$\blacksquare$

## [Code Transformations: End Recursion]

The recursion can be writen as *end recursion*

```
// 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);
}
```

$\longrightarrow$

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

## [Code-Transformation: End-Recursion $\Rightarrow$ Iteration]

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

$\longrightarrow$

```
int f(int a, int b) {
  int res = 0;
  while (b != 1) {
    int z = 0;
    if (b % 2 != 0){
      --b;
      z = a;
    }
    res += z;
    a *= 2; // neues a
    b /= 2; // neues b
  }
  res += a; // Basisfall b=1
  return res;
}
```

## [Code-Transformation: Simplify]

```
int f(int a, int b) {
  int res = 0;
  while (b != 1) {
    int z = 0;
    if (b % 2 != 0){
      --b;   ⟶ Teil der Division
      z = a; ⟶ Direkt in res
    }
    res += z;
    a *= 2;
    b /= 2;
  }
  res += a;  ⟶ in den Loop
  return res;
}
```

$\longrightarrow$

```
// 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;
    a *= 2;
    b /= 2;
  }
  return res;
}
```

## Correctness: Reasoning using 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 it operates there like the induction step in induction proofs.
- Invariants are obviously powerful tools for proofs!

## [Further simplification]

```
// 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;
}
```

$\longrightarrow$

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

# [Analysis]

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

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

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

## 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.

## 3.2 Fast Integer Multiplication

[Ottman/Widmayer, Kap. 1.2.3]

## Example 2: Multiplication of large Numbers

Primary school:

$$
\begin{array}{cccccc|c}
 & a & b & & c & d & \\
 & 6 & 2 & \cdot & 3 & 7 & \\
\hline
 & & & & 1 & 4 & d \cdot b \\
 & & & 4 & 2 & & d \cdot a \\
 & & & & 6 & & c \cdot b \\
 & & 1 & 8 & & & c \cdot a \\
\hline
= & & 2 & 2 & 9 & 4 &
\end{array}
$$

$2 \cdot 2 = 4$ single-digit multiplications. $\Rightarrow$ Multiplication of two $n$-digit numbers: $n^2$ single-digit multiplications

## Observation

$$\begin{aligned} ab \cdot cd &= (10 \cdot a + b) \cdot (10 \cdot c + d) \\ &= 100 \cdot a \cdot c + 10 \cdot a \cdot c \\ &\quad + 10 \cdot b \cdot d + b \cdot d \\ &\quad + 10 \cdot (a - b) \cdot (d - c) \end{aligned}$$

## Improvement?

$$
\begin{array}{cccccc|l}
 & a & b & & c & d & \\
 & 6 & 2 & \cdot & 3 & 7 & \\
\hline
 & & & & 1 & 4 & d \cdot b \\
 & & & 1 & 4 & & d \cdot b \\
 & & & 1 & 6 & & (a-b)\cdot(d-c) \\
 & & & 1 & 8 & & c \cdot a \\
 & & 1 & 8 & & & c \cdot a \\
\hline
= & & 2 & 2 & 9 & 4 &
\end{array}
$$

$\rightarrow 3$ single-digit multiplications.

## Large Numbers

$$6237 \cdot 5898 = \underbrace{62}_{a'}\underbrace{37}_{b'} \cdot \underbrace{58}_{c'}\underbrace{98}_{d'}$$

Recursive / inductive application: compute $a' \cdot c'$, $a' \cdot d'$, $b' \cdot c'$ and $c' \cdot d'$ as shown above.

$\rightarrow 3 \cdot 3 = 9$ instead of $16$ single-digit multiplications.

## Generalization

Assumption: two numbers with $n$ digits each, $n = 2^k$ for some $k$.

$$\begin{aligned}(10^{n/2}a + b) \cdot (10^{n/2}c + d) &= 10^n \cdot a \cdot c + 10^{n/2} \cdot a \cdot c \\ &+ 10^{n/2} \cdot b \cdot d + b \cdot d \\ &+ 10^{n/2} \cdot (a - b) \cdot (d - c)\end{aligned}$$

Recursive application of this formula: algorithm by Karatsuba and Ofman (1962).

## Analysis

$M(n)$: Number of single-digit multiplications.

Recursive application of the algorithm from above $\Rightarrow$ recursion equality:

$$M(2^k) = \begin{cases} 1 & \text{if } k = 0, \\ 3 \cdot M(2^{k-1}) & \text{if } k > 0. \end{cases}$$

## Iterative Substition

Iterative substition of the recursion formula in order to guess a solution of the recursion formula:

$$\begin{aligned} M(2^k) &= 3 \cdot M(2^{k-1}) = 3 \cdot 3 \cdot M(2^{k-2}) = 3^2 \cdot M(2^{k-2}) \\ &= \ldots \\ &\overset{!}{=} 3^k \cdot M(2^0) = 3^k. \end{aligned}$$

## Proof: induction

*Hypothesis H*:

$$M(2^k) = 3^k.$$

*Base clause ($k = 0$)*:

$$M(2^0) = 3^0 = 1. \quad \checkmark$$

*Induction step ($k \to k+1$)*:

$$M(2^{k+1}) \overset{\text{def}}{=} 3 \cdot M(2^k) \overset{\text{H}}{=} 3 \cdot 3^k = 3^{k+1}.$$

$\blacksquare$

## Comparison

Traditionally $n^2$ single-digit multiplications.

Karatsuba/Ofman:

$$M(n) = 3^{\log_2 n} = (2^{\log_2 3})^{\log_2 n} = 2^{\log_2 3 \log_2 n} = n^{\log_2 3} \approx n^{1.58}.$$

Example: number with $1000$ digits: $1000^2/1000^{1.58} \approx 18$.

## Best possible algorithm?

We only know the upper bound $n^{\log_2 3}$.

There are (for large $n$) practically relevant algorithms that are faster. Example: Schönhage-Strassen algorithm (1971) based on fast Fouriertransformation with running time $\mathcal{O}(n \log n \cdot \log \log n)$. The best upper bound is not known.

Lower bound: $n$. Each digit has to be considered at least once.

## Appendix: Asymptotics with Addition and Shifts

For each multiplication of two $n$-digit numbers we also should take into account a constant number of additions, subtractions and shifts

Additions, subtractions and shifts of $n$-digit numbers cost $\mathcal{O}(n)$

Therefore the asymptotic running time is determined (with some $c > 1$) by the following recurrence

$$T(n) = \begin{cases} 3 \cdot T\left(\frac{1}{2}n\right) + c \cdot n & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

## Appendix: Asymptotics with Addition and Shifts

Assumption: $n = 2^k$, $k > 0$

$$\begin{aligned}
T(2^k) &= 3 \cdot T\left(2^{k-1}\right) + c \cdot 2^k \\
&= 3 \cdot (3 \cdot T(2^{k-2}) + c \cdot 2^{k-1}) + c \cdot 2^k \\
&= 3 \cdot (3 \cdot (3 \cdot T(2^{k-3}) + c \cdot 2^{k-2}) + c \cdot 2^{k-1}) + c \cdot 2^k \\
&= 3 \cdot (3 \cdot (...(3 \cdot T(2^{k-k}) + c \cdot 2^1)...) + c \cdot 2^{k-1}) + c \cdot 2^k \\
&= 3^k \cdot T(1) + c \cdot 3^{k-1}2^1 + c \cdot 3^{k-2}2^2 + ... + c \cdot 3^0 2^k \\
&\leq c \cdot 3^k \cdot (1 + 2/3 + (2/3)^2 + ... + (2/3)^k)
\end{aligned}$$

Die geometrische Reihe $\sum_{i=0}^{k} \varrho^i$ mit $\varrho = 2/3$ konvergiert für $k \to \infty$ gegen $\frac{1}{1-\varrho} = 3$.

Somit $T(2^k) \leq c \cdot 3^k \cdot 3 \in \Theta(3^k) = \Theta(3^{\log_2 n}) = \Theta(n^{\log_2 3})$.

## 3.3 Maximum Subarray Problem

Algorithm Design – Maximum Subarray Problem [Ottman/Widmayer, Kap. 1.3]
Divide and Conquer [Ottman/Widmayer, Kap. 1.2.2. S.9; Cormen et al, Kap. 4-4.1]

## Algorithm Design

Inductive development of an algorithm: partition into subproblems, use solutions for the subproblems to find the overal solution.

*Goal:* development of the asymptotically most efficient (correct) algorithm.

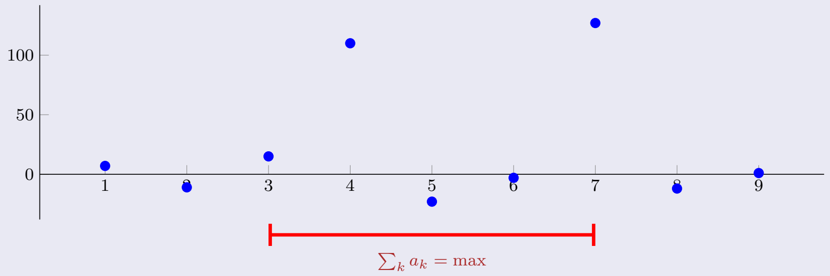
*Efficiency* towards run time costs (# fundamental operations) or /and memory consumption.

## Maximum Subarray Problem

*Given:* an array of $n$ real numbers $(a_1, \ldots, a_n)$.

*Wanted:* interval $[i, j]$, $1 \le i \le j \le n$ with maximal positive sum $\sum_{k=i}^{j} a_k$.

**Example:** $a = (7, -11, 15, 110, -23, -3, 127, -12, 1)$



## Naive Maximum Subarray Algorithm

**Input**: A sequence of $n$ numbers $(a_1, a_2, \dots, a_n)$
**Output**: $I$, $J$ such that $\sum_{k=I}^{J} a_k$ maximal.

```
$M \leftarrow 0$; $I \leftarrow 1$; $J \leftarrow 0$
for $i \in \{1, \dots, n\}$ do
    for $j \in \{i, \dots, n\}$ do
        $m = \sum_{k=i}^{j} a_k$
        if $m > M$ then
            $M \leftarrow m$; $I \leftarrow i$; $J \leftarrow j$
return $I, J$
```

## Analysis

**Theorem**

*The naive algorithm for the Maximum Subarray problem executes $\Theta(n^3)$ additions.*

Beweis:

$$
\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i+1) &= \sum_{i=1}^{n}\sum_{j=0}^{n-i}(j+1) = \sum_{i=1}^{n}\sum_{j=1}^{n-i+1} j = \sum_{i=1}^{n}\frac{(n-i+1)(n-i+2)}{2} \\
&= \sum_{i=0}^{n}\frac{i\cdot(i+1)}{2} = \frac{1}{2}\left(\sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} i\right) \\
&= \frac{1}{2}\left(\frac{n(2n+1)(n+1)}{6} + \frac{n(n+1)}{2}\right) = \frac{n^3+3n^2+2n}{6} = \Theta(n^3).
\end{aligned}
$$

■

## Observation

$$\sum_{k=i}^{j} a_k = \underbrace{\left(\sum_{k=1}^{j} a_k\right)}_{S_j} - \underbrace{\left(\sum_{k=1}^{i-1} a_k\right)}_{S_{i-1}}$$

*Prefix sums*

$$S_i := \sum_{k=1}^{i} a_k.$$

## Maximum Subarray Algorithm with Prefix Sums

**Input**: A sequence of $n$ numbers $(a_1, a_2, \ldots, a_n)$
**Output**: $I$, $J$ such that $\sum_{k=J}^{J} a_k$ maximal.

```
$\mathcal{S}_0 \leftarrow 0$
for $i \in \{1, \ldots, n\}$ do // prefix sum
    $\mathcal{S}_i \leftarrow \mathcal{S}_{i-1} + a_i$
$M \leftarrow 0$; $I \leftarrow 1$; $J \leftarrow 0$
for $i \in \{1, \ldots, n\}$ do
    for $j \in \{i, \ldots, n\}$ do
        $m = \mathcal{S}_j - \mathcal{S}_{i-1}$
        if $m > M$ then
            $M \leftarrow m$; $I \leftarrow i$; $J \leftarrow j$
```

## Analysis

**Theorem**

*The prefix sum algorithm for the Maximum Subarray problem conducts $\Theta(n^2)$ additions and subtractions.*

Beweis:

$$\sum_{i=1}^{n} 1 + \sum_{i=1}^{n} \sum_{j=i}^{n} 1 = n + \sum_{i=1}^{n} (n - i + 1) = n + \sum_{i=1}^{n} i = \Theta(n^2)$$

■

## divide et impera

**Divide and Conquer**

Divide the problem into subproblems that contribute to the simplified computation of the overal problem.

$$\begin{tikzcd}
 & & P_{22} \arrow[r] & S_{22} \arrow[dr, dash] & & \\
 & P_2 \arrow[ur, dash] \arrow[dr, dash] & & & S_2 \arrow[ddr, dash] & \\
 & & P_{21} \arrow[r] & S_{21} \arrow[ur, dash] & & \\
\text{Problem } P \arrow[uur, dash] \arrow[ddr, dash] & & & & & \text{Solution} \\
 & & P_{12} \arrow[r] & S_{12} \arrow[dr, dash] & & \\
 & P_1 \arrow[ur, dash] \arrow[dr, dash] & & & S_1 \arrow[uur, dash] & \\
 & & P_{11} \arrow[r] & S_{11} \arrow[ur, dash] & &
\end{tikzcd}$$

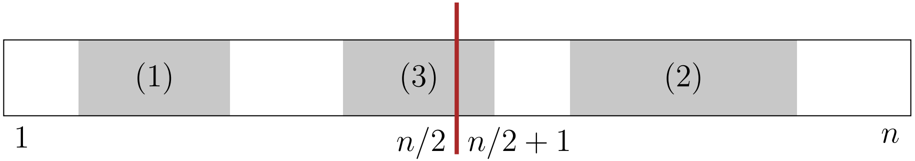
## Maximum Subarray – Divide

- Divide: Divide the problem into two (roughly) equally sized halves: $(a_1, \ldots, a_n) = (a_1, \ldots, a_{\lfloor n/2 \rfloor}, \quad a_{\lfloor n/2 \rfloor + 1}, \ldots, a_1)$
- Simplifying assumption: $n = 2^k$ for some $k \in \mathbb{N}$.

## Maximum Subarray – Conquer

If $i$ and $j$ are indices of a solution $\Rightarrow$ case by case analysis:

1. Solution in left half $1 \leq i \leq j \leq n/2 \Rightarrow$ Recursion (left half)
2. Solution in right half $n/2 < i \leq j \leq n \Rightarrow$ Recursion (right half)
3. Solution in the middle $1 \leq i \leq n/2 < j \leq n \Rightarrow$ Subsequent observation



## Maximum Subarray – Observation

Assumption: solution in the middle $1 \leq i \leq n/2 < j \leq n$

$$\begin{aligned} S_{\max} &= \max_{\substack{1 \leq i \leq n/2 \\ n/2 < j \leq n}} \sum_{k=i}^{j} a_k = \max_{\substack{1 \leq i \leq n/2 \\ n/2 < j \leq n}} \left( \sum_{k=i}^{n/2} a_k + \sum_{k=n/2+1}^{j} a_k \right) \\ &= \max_{1 \leq i \leq n/2} \sum_{k=i}^{n/2} a_k + \max_{n/2 < j \leq n} \sum_{k=n/2+1}^{j} a_k \\ &= \max_{1 \leq i \leq n/2} \underbrace{S_{n/2} - S_{i-1}}_{\text{suffix sum}} + \max_{n/2 < j \leq n} \underbrace{S_j - S_{n/2}}_{\text{prefix sum}} \end{aligned}$$

## Maximum Subarray Divide and Conquer Algorithm

```
Input:    A sequence of $n$ numbers $(a_1, a_2, \dots, a_n)$
Output:   Maximal $\sum_{k=i'}^{j'} a_k$.
if $n = 1$ then
    return $\max\{a_1, 0\}$
else
    Divide $a = (a_1, \dots, a_n)$ in $A_1 = (a_1, \dots, a_{n/2})$ und $A_2 = (a_{n/2+1}, \dots, a_n)$
    Recursively compute best solution $W_1$ in $A_1$
    Recursively compute best solution $W_2$ in $A_2$
    Compute greatest suffix sum $S$ in $A_1$
    Compute greatest prefix sum $P$ in $A_2$
    Let $W_3 \leftarrow S + P$
    return $\max\{W_1, W_2, W_3\}$
```

## Analysis

**Theorem**

*The divide and conquer algorithm for the maximum subarray sum problem conducts a number of* $\Theta(n \log n)$ *additions and comparisons.*

## Analysis

```
Input:   A sequence of $n$ numbers $(a_1, a_2, \ldots, a_n)$
Output:  Maximal $\sum_{k=i'}^{j'} a_k$.
if $n = 1$ then
$\Theta(1)$ return $\max\{a_1, 0\}$
else
$\Theta(1)$ Divide $a = (a_1, \ldots, a_n)$ in $A_1 = (a_1, \ldots, a_{n/2})$ und $A_2 = (a_{n/2+1}, \ldots, a_n)$
$T(n/2)$ Recursively compute best solution $W_1$ in $A_1$
$T(n/2)$ Recursively compute best solution $W_2$ in $A_2$
$\Theta(n)$ Compute greatest suffix sum $S$ in $A_1$
$\Theta(n)$ Compute greatest prefix sum $P$ in $A_2$
$\Theta(1)$ Let $W_3 \leftarrow S + P$
$\Theta(1)$ return $\max\{W_1, W_2, W_3\}$
```

## Analysis

Recursion equation

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(\frac{n}{2}) + a \cdot n & \text{if } n > 1 \end{cases}$$

## Analysis

Mit $n = 2^k$:

$$\overline{T}(k) = \begin{cases} c & \text{if } k = 0 \\ 2\overline{T}(k-1) + a \cdot 2^k & \text{if } k > 0 \end{cases}$$

Solution:

$$\overline{T}(k) = 2^k \cdot c + \sum_{i=0}^{k-1} 2^i \cdot a \cdot 2^{k-i} = c \cdot 2^k + a \cdot k \cdot 2^k = \Theta(k \cdot 2^k)$$

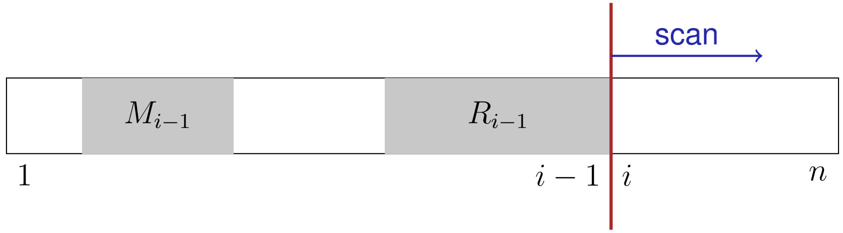
also

$$T(n) = \Theta(n \log n)$$

■

## Maximum Subarray Sum Problem – Inductively

Assumption: maximal value $M_{i-1}$ of the subarray sum is known for $(a_1, \ldots, a_{i-1})$ $(1 < i \leq n)$.



$a_i$: generates at most a better interval at the right bound (prefix sum).

$$R_{i-1} \Rightarrow R_i = \max\{R_{i-1} + a_i, 0\}$$

## Inductive Maximum Subarray Algorithm

**Input**: A sequence of $n$ numbers $(a_1, a_2, \dots, a_n)$.
**Output**: $\max\{0, \max_{i,j} \sum_{k=i}^{j} a_k\}$.

```
$M \leftarrow 0$
$R \leftarrow 0$
for $i = 1 \dots n$ do
    $R \leftarrow R + a_i$
    if $R < 0$ then
        $R \leftarrow 0$
    if $R > M$ then
        $M \leftarrow R$
return $M$;
```

## Analysis

**Theorem**

*The inductive algorithm for the Maximum Subarray problem conducts a number of $\Theta(n)$ additions and comparisons.*

## Complexity of the problem?

Can we improve over $\Theta(n)$?

Every correct algorithm for the Maximum Subarray Sum problem must consider each element in the algorithm.

Assumption: the algorithm does not consider $a_i$.

1. The algorithm provides a solution including $a_i$. Repeat the algorithm with $a_i$ so small that the solution must not have contained the point in the first place.
2. The algorithm provides a solution not including $a_i$. Repeat the algorithm with $a_i$ so large that the solution must have contained the point in the first place.

## Complexity of the maximum Subarray Sum Problem

**Theorem**

*The Maximum Subarray Sum Problem has Complexity $\Theta(n)$.*

Beweis: Inductive algorithm with asymptotic execution time $\mathcal{O}(n)$.
Every algorithm has execution time $\Omega(n)$.
Thus the complexity of the problem is $\Omega(n) \cap \mathcal{O}(n) = \Theta(n)$. ■

## 3.4 Appendix

Derivation of some mathemmatical formulas

## Sums

$$\sum_{i=0}^{n} i^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$$

Trick:

$$\sum_{i=1}^{n} i^3 - (i-1)^3 = \sum_{i=0}^{n} i^3 - \sum_{i=0}^{n-1} i^3 = n^3$$

$$\sum_{i=1}^{n} i^3 - (i-1)^3 = \sum_{i=1}^{n} i^3 - i^3 + 3i^2 - 3i + 1 = n - \frac{3}{2} n \cdot (n+1) + 3 \sum_{i=0}^{n} i^2$$

$$\Rightarrow \sum_{i=0}^{n} i^2 = \frac{1}{6}(2n^3 + 3n^2 + n) \in \Theta(n^3)$$

Can easily be generalized: $\sum_{i=1}^{n} i^k \in \Theta(n^{k+1})$.

## Geometric Series

$$\sum_{i=0}^{n} \rho^i \overset{!}{=} \frac{1-\rho^{n+1}}{1-\rho}$$

$$\begin{aligned}\sum_{i=0}^{n} \rho^i \cdot (1-\varrho) &= \sum_{i=0}^{n} \rho^i - \sum_{i=0}^{n} \rho^{i+1} = \sum_{i=0}^{n} \rho^i - \sum_{i=1}^{n+1} \rho^i \\ &= \rho^0 - \rho^{n+1} = 1-\rho^{n+1}.\end{aligned}$$

For $0 \leq \rho < 1$:

$$\sum_{i=0}^{\infty} \rho^i = \frac{1}{1-\rho}$$

# 4. Searching

Linear Search, Binary Search, (Interpolation Search,) Lower Bounds
[Ottman/Widmayer, Kap. 3.2, Cormen et al, Kap. 2: Problems
2.1-3,2.2-3,2.3-5]

## The Search Problem

Provided

- A set of data sets

**examples**

telephone book, dictionary, symbol table

- Each dataset has a key $k$.
- Keys are comparable: unique answer to the question $k_1 \leq k_2$ for keys $k_1$, $k_2$.

Task: find data set by key $k$.

## Search in Array

Provided

- Array $A$ with $n$ elements $(A[1], \dots, A[n])$.
- Key $b$

Wanted: index $k$, $1 \leq k \leq n$ with $A[k] = b$ or ”not found”.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 22 | 20 | 32 | 10 | 35 | 24 | 42 | 38 | 28 | 41 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

## Linear Search

Traverse the array from $A[1]$ to $A[n]$.

- *Best case:* $1$ comparison.
- *Worst case:* $n$ comparisons.
- Assumption: each permutation of the $n$ keys with same probability. *Expected* number of comparisons for the successful search:

$$\frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}.$$

## Search in a Sorted Array

Provided

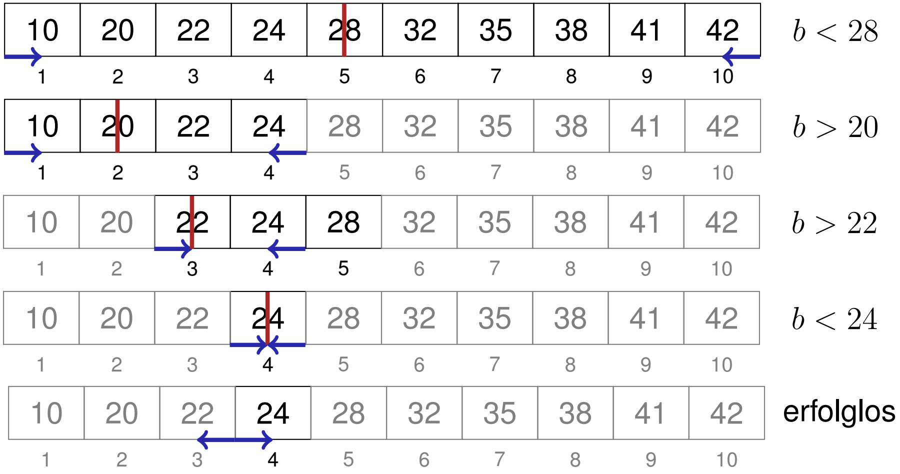
- Sorted array $A$ with $n$ elements $(A[1], \dots, A[n])$ with $A[1] \leq A[2] \leq \cdots \leq A[n]$.
- Key $b$

Wanted: index $k$, $1 \leq k \leq n$ with $A[k] = b$ or ”not found”.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 20 | 22 | 24 | 28 | 32 | 35 | 38 | 41 | 42 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

## Divide and Conquer!

Search $b = 23$.



## Binary Search Algorithm BSearch$(A[l..r], b)$

**Input:** Sorted array $A$ of $n$ keys. Key $b$. Bounds $1 \leq l \leq r \leq n$ or $l > r$ beliebig.
**Output:** Index of the found element. $0$, if not found.

```
$m \leftarrow \lfloor (l+r)/2 \rfloor$
if $l > r$ then // Unsuccessful search
    return NotFound
else if $b = A[m]$ then// found
    return $m$
else if $b < A[m]$ then// element to the left
    return BSearch($A[l..m-1], b$)
else // $b > A[m]$: element to the right
    return BSearch($A[m+1..r], b$)
```

## Analysis (worst case)

Recurrence ($n = 2^k$)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Compute:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c = ... \\ &= T\left(\frac{n}{2^i}\right) + i \cdot c \\ &= T\left(\frac{n}{n}\right) + \log_2 n \cdot c = d + c \cdot \log_2 n \in \Theta(\log n) \end{aligned}$$

## Analysis (worst case)

$$T(n) = \begin{cases} d & \text{if } n = 1, \\ T(n/2) + c & \text{if } n > 1. \end{cases}$$

**Guess** : $T(n) = d + c \cdot \log_2 n$

**Proof by induction:**

- Base clause: $T(1) = d$.
- Hypothesis: $T(n/2) = d + c \cdot \log_2 n/2$
- Step: $(n/2 \to n)$

$$T(n) = T(n/2) + c = d + c \cdot (\log_2 n - 1) + c = d + c \log_2 n.$$

## Result

**Theorem**

*The binary sorted search algorithm requires* $\Theta(\log n)$ *fundamental operations.*

## Iterative Binary Search Algorithm

**Input:** Sorted array $A$ of $n$ keys. Key $b$.
**Output:** Index of the found element. $0$, if unsuccessful.

```
$l \leftarrow 1$; $r \leftarrow n$
while $l \leq r$ do
    $m \leftarrow \lfloor (l+r)/2 \rfloor$
    if $A[m] = b$ then
        return $m$
    else if $A[m] < b$ then
        $l \leftarrow m+1$
    else
        $r \leftarrow m-1$
return *NotFound*;
```

## Correctness

Algorithm terminates only if $A$ is empty or $b$ is found.

**Invariant:** If $b$ is in $A$ then $b$ is in domain $A[l..r]$

**Proof by induction**

- Base clause $b \in A[1..n]$ (oder nicht)
- Hypothesis: invariant holds after $i$ steps.
- Step:
  $b < A[m] \Rightarrow b \in A[l..m-1]$
  $b > A[m] \Rightarrow b \in A[m+1..r]$

## [Can this be improved?]

Assumption: *values* of the array are uniformly distributed.

**Example**

Search for ”Becker” at the very beginning of a telephone book while search for ”Wawrinka" rather close to the end.
Binary search always starts in the middle.

Binary search always takes $m = \left\lfloor l + \frac{r-l}{2} \right\rfloor$.

## [Interpolation search]

Expected relative position of $b$ in the search interval $[l, r]$

$$\rho = \frac{b - A[l]}{A[r] - A[l]} \in [0, 1].$$

New 'middle': $l + \rho \cdot (r - l)$

Expected number of comparisons $\mathcal{O}(\log \log n)$ (without proof).

? Would you always prefer interpolation search?

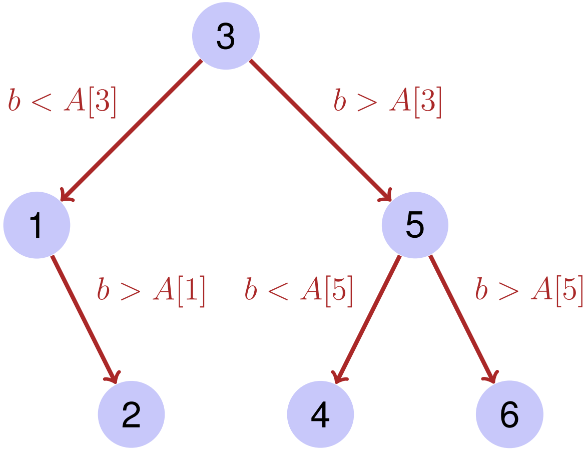
! No: worst case number of comparisons $\Omega(n)$.

## Lower Bounds

Binary Search (worst case): $\Theta(\log n)$ comparisons.

Does for *any* search algorithm in a sorted array (worst case) hold that number comparisons = $\Omega(\log n)$?

## Decision tree



- For any input $b = A[i]$ the algorithm must succeed $\Rightarrow$ decision tree comprises at least $n$ nodes.
- Number comparisons in worst case = height of the tree = maximum number nodes from root to leaf.

## Decision Tree

Binary tree with height $h$ has at most
$2^0 + 2^1 + \cdots + 2^{h-1} = 2^h - 1 < 2^h$ nodes.

$$2^h > n \Rightarrow h > \log_2 n$$

Decision tree with $n$ node has at least height $\log_2 n$.

Number decisions = $\Omega(\log n)$.

**Theorem**

*Any comparison-based search algorithm on sorted data with length $n$ requires in the worst case $\Omega(\log n)$ comparisons.*

## Lower bound for Search in Unsorted Array

**Theorem**

*Any comparison-based search algorithm with unsorted data of length $n$ requires in the worst case $\Omega(n)$ comparisons.*

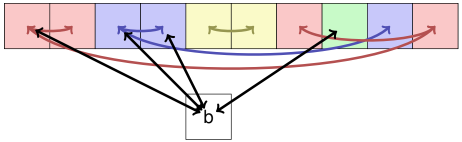
## Attempt

**Correct?**

”Proof”: to find $b$ in $A$, $b$ must be compared with each of the $n$ elements $A[i]$ $(1 \leq i \leq n)$.

Wrong argument! It is still possible to compare elements within $A$.

## Better Argument



- Different comparisons: Number comparisons with $b$: $e$ Number comparisons without $b$: $i$
- Comparisons induce $g$ groups. Initially $g = n$.
- To connect two groups at least one comparison is needed: $n - g \leq i$.
- At least one element per group must be compared with $b$.
- Number comparisons $i + e \geq n - g + g = n$. ■

# 5. Selection

The Selection Problem, Randomised Selection, Linear Worst-Case Selection [Ottman/Widmayer, Kap. 3.1, Cormen et al, Kap. 9]

## The Problem of Selection

Input

- unsorted array $A = (A_1, \ldots, A_n)$ with pairwise different values
- Number $1 \leq k \leq n$.

Output $A[i]$ with $|\{j : A[j] < A[i]\}| = k - 1$

**Special cases**

$k = 1$: Minimum: Algorithm with $n$ comparison operations trivial.
$k = n$: Maximum: Algorithm with $n$ comparison operations trivial.
$k = \lfloor n/2 \rfloor$: Median.

## Naive Algorithm

Repeatedly find and remove the minimum $\Theta(k \cdot n)$.
$\rightarrow$ Median in $\Theta(n^2)$

## Min and Max

? To separately find minimum an maximum in $(A[1], \dots, A[n])$, $2n$ comparisons are required. (How) can an algorithm with less than $2n$ comparisons for both values at a time can be found?

! Possible with $\frac{3}{2}n$ comparisons: compare 2 elements each and then the smaller one with min and the greater one with max.[4]

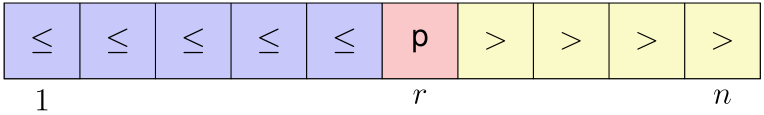
[4] An indication that the naive algorithm can be improved.

## Better Approaches

- Sorting (covered soon): $\Theta(n \log n)$
- Use a pivot: $\Theta(n)$ !

## Use a pivot

1. Choose a (an arbitrary) *pivot* $p$
2. Partition $A$ in two parts, and determine the rank of $p$ by counting the indices $i$ with $A[i] \leq p$.
3. Recursion on the relevant part. If $k = r$ then found.



## Algorithmus Partition($A[l..r], p$)

**Input:** Array $A$, that contains the pivot $p$ in the interval $[l, r]$ at least once.
**Output:** Array $A$ partitioned in $[l..r]$ around $p$. Returns position of $p$.

```
while $l \leq r$ do
    while $A[l] < p$ do
        $l \leftarrow l + 1$
    while $A[r] > p$ do
        $r \leftarrow r - 1$
    swap($A[l]$, $A[r]$)
    if $A[l] = A[r]$ then
        $l \leftarrow l + 1$
return l-1
```

## Correctness: Invariant

Invariant $I$: $A_i \leq p \ \forall i \in [0, l)$, $A_i \geq p \ \forall i \in (r, n]$, $\exists k \in [l, r] : A_k = p$.

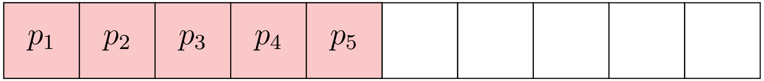
```
while $l \leq r$ do
                                        $I$
    while $A[l] < p$ do
        $l \leftarrow l + 1$
                                        $I$ und $A[l] \geq p$
    while $A[r] > p$ do
        $r \leftarrow r - 1$
                                        $I$ und $A[r] \leq p$
    swap($A[l]$, $A[r]$)
                                        $I$ und $A[l] \leq p \leq A[r]$
    if $A[l] = A[r]$ then
        $l \leftarrow l + 1$
                                        $I$
return l-1
```

## Correctness: progress

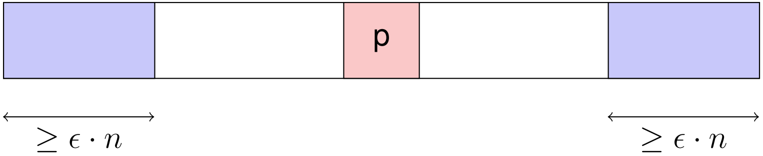
```
while $l \leq r$ do
    while $A[l] < p$ do                progress if $A[l] < p$
        $l \leftarrow l + 1$
    while $A[r] > p$ do                progress if $A[r] > p$
        $r \leftarrow r - 1$
    swap($A[l]$, $A[r]$)               progress if $A[l] > p$ oder $A[r] < p$
    if $A[l] = A[r]$ then              progress if $A[l] = A[r] = p$
        $l \leftarrow l + 1$
return l-1
```

## Choice of the pivot.

The minimum is a bad pivot: worst case $\Theta(n^2)$



A good pivot has a linear number of elements on both sides.



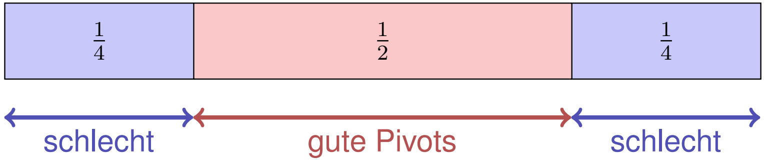
## Analysis

Partitioning with factor $q$ ($0 < q < 1$): two groups with $q \cdot n$ and $(1-q) \cdot n$ elements (without loss of generality $g \geq 1 - q$).

$$\begin{aligned} T(n) &\leq T(q \cdot n) + c \cdot n \\ &= c \cdot n + q \cdot c \cdot n + T(q^2 \cdot n) = \ldots = c \cdot n \sum_{i=0}^{\log_q(n)-1} q^i + T(1) \\ &\leq c \cdot n \underbrace{\sum_{i=0}^{\infty} q^i}_{\text{geom. Reihe}} + d = c \cdot n \cdot \frac{1}{1-q} + d = \mathcal{O}(n) \end{aligned}$$

## How can we achieve this?

Randomness to our rescue (Tony Hoare, 1961). In each step choose a random pivot.



Probability for a good pivot in one trial: $\frac{1}{2} =: \rho$.

Probability for a good pivot after $k$ trials: $(1-\rho)^{k-1} \cdot \rho$.

Expected number of trials: $1/\rho = 2$ (Expected value of the geometric distribution:)

## Algorithm Quickselect ($A[l..r], k$)

**Input:** Array $A$ with length $n$. Indices $1 \leq l \leq k \leq r \leq n$, such that for all $x \in A[l..r] : |\{j|A[j] \leq x\}| \geq l$ and $|\{j|A[j] \leq x\}| \leq r$.
**Output:** Value $x \in A[l..r]$ with $|\{j|A[j] \leq x\}| \geq k$ and $|\{j|x \leq A[j]\}| \geq n - k + 1$

```
if l=r then
    return $A[l]$;
$x \leftarrow$ RandomPivot($A[l..r]$)
$m \leftarrow$ Partition($A[l..r], x$)
if $k < m$ then
    return QuickSelect($A[l..m-1], k$)
else if $k > m$ then
    return QuickSelect($A[m+1..r], k$)
else
    return $A[k]$
```

## Algorithm RandomPivot ($A[l..r]$)

```
Input: Array $A$ with length $n$. Indices $1 \leq l \leq i \leq r \leq n$
Output: Random “good” pivot $x \in A[l..r]$
repeat
    choose a random pivot $x \in A[l..r]$
    $p \leftarrow l$
    for $j = l$ to $r$ do
        if $A[j] \leq x$ then $p \leftarrow p + 1$
until $\left\lfloor \frac{3l+r}{4} \right\rfloor \leq p \leq \left\lceil \frac{l+3r}{4} \right\rceil$
return $x$
```

*This algorithm is only of theoretical interest and delivers a good pivot in 2 expected iterations. Practically, in algorithm QuickSelect a uniformly chosen random pivot can be chosen or a deterministic one such as the median of three elements.*

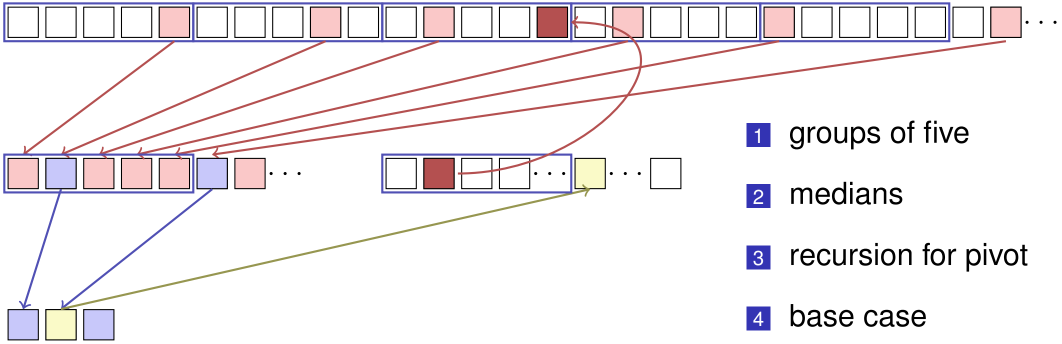
## Median of medians

Goal: find an algorithm that even in worst case requires only linearly many steps.

Algorithm Select ($k$-smallest)

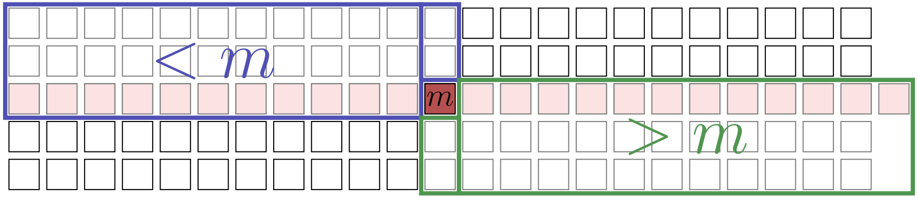
- Consider groups of five elements.
- Compute the median of each group (straighforward)
- Apply Select recursively on the group medians.
- Partition the array around the found median of medians. Result: $i$
- If $i = k$ then result. Otherwise: select recursively on the proper side.

## Median of medians



1. groups of five
2. medians
3. recursion for pivot
4. base case
5. pivot (level 1)
6. partition (level 1)
7. median = pivot level 0
8. 2. recursion starts

## How good is this?



Number points left / right of the median of medians (without median group and the rest group) $\geq 3 \cdot (\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2) \geq \frac{3n}{10} - 6$

Second call with maximally $\lceil \frac{7n}{10} + 6 \rceil$ elements.

## Analysis

Recursion inequality:

$$T(n) \leq T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\left\lceil \frac{7n}{10} + 6 \right\rceil\right) + d \cdot n.$$

with some constant $d$.

Claim:

$$T(n) = \mathcal{O}(n).$$

## Proof

Base clause: choose $c$ large enough such that

$$T(n) \leq c \cdot n \text{ für alle } n \leq n_0.$$

Induction hypothesis:

$$T(i) \leq c \cdot i \text{ für alle } i < n.$$

Induction step:

$$\begin{aligned} T(n) &\leq T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\left\lceil \frac{7n}{10} + 6 \right\rceil\right) + d \cdot n \\ &= c \cdot \left\lceil \frac{n}{5} \right\rceil + c \cdot \left\lceil \frac{7n}{10} + 6 \right\rceil + d \cdot n. \end{aligned}$$

## Proof

Induction step:

$$T(n) \leq c \cdot \left\lceil \frac{n}{5} \right\rceil + c \cdot \left\lceil \frac{7n}{10} + 6 \right\rceil + d \cdot n$$
$$\leq c \cdot \frac{n}{5} + c + c \cdot \frac{7n}{10} + 6c + c + d \cdot n = \frac{9}{10} \cdot c \cdot n + 8c + d \cdot n.$$

Choose $c \geq 80 \cdot d$ and $n_0 = 91$.

$$T(n) \leq \frac{72}{80} \cdot c \cdot n + 8c + \frac{1}{80} \cdot c \cdot n = c \cdot \underbrace{\left(\frac{73}{80} n + 8\right)}_{\leq n \text{ für } n > n_0} \leq c \cdot n.$$

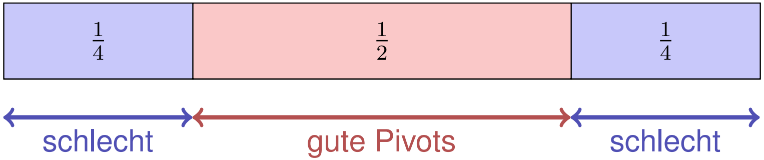
## Result

**Theorem**

*The $k$-th element of a sequence of $n$ elements can, in the worst case, be found in $\Theta(n)$ steps.*

## Overview

| | | |
|---|---|---|
| 1. | Repeatedly find minimum | $\mathcal{O}(n^2)$ |
| 2. | Sorting and choosing $A[i]$ | $\mathcal{O}(n \log n)$ |
| 3. | Quickselect with random pivot | $\mathcal{O}(n)$ expected |
| 4. | Median of Medians (Blum) | $\mathcal{O}(n)$ worst case |



## 5.1 Appendix

Derivation of some mathemmatical formulas

## [Expected value of the Geometric Distribution]

Random variable $X \in \mathbb{N}^+$ with $\mathbb{P}(X = k) = (1-p)^{k-1} \cdot p$.

Expected value

$$
\begin{aligned}
\mathbb{E}(X) &= \sum_{k=1}^{\infty} k \cdot (1-p)^{k-1} \cdot p = \sum_{k=1}^{\infty} k \cdot q^{k-1} \cdot (1-q) \\
&= \sum_{k=1}^{\infty} k \cdot q^{k-1} - k \cdot q^k = \sum_{k=0}^{\infty} (k+1) \cdot q^k - k \cdot q^k \\
&= \sum_{k=0}^{\infty} q^k = \frac{1}{1-q} = \frac{1}{p}.
\end{aligned}
$$

# 6. C++ advanced (I)

Repetition: vectors, pointers and iterators, range for, keyword auto, a class for vectors, subscript-operator, move-construction, iterators

## What do we learn today?

- Keyword `auto`
- Ranged `for`
- Short recap of the Rule of Three
- Subscript operator
- Move Semantics, X-Values and the Rule of Five
- Custom Iterators

## We look back...

```
#include <iostream>
#include <vector>
using iterator = std::vector<int>::iterator;

int main(){
  // Vector of length 10
  std::vector<int> v(10);
  // Input
  for (int i = 0; i < v.size(); ++i)
    std::cin >> v[i];
  // Output
  for (iterator it = v.begin(); it != v.end(); ++it)
    std::cout << *it << " ";
}
```

We want to understand this in depth!

This looks too pedestrian

## Useful tools (1): `auto` (C++11)

The keyword `auto`:

The type of a variable is inferred from the initializer.

**Examples**

```
int x = 10;
auto y = x; // int
auto z = 3; // int
std::vector<double> v(5);
auto i = v[3]; // double
```

## Slightly better...

```
#include <iostream>
#include <vector>

int main(){
  std::vector<int> v(10); // Vector of length 10

  for (int i = 0; i < v.size(); ++i)
    std::cin >> v[i];

  for (auto it = v.begin(); it != v.end(); ++it){
    std::cout << *it << " ";
  }
}
```

## Useful tools (2): range `for` (C++11)

```
for (range-declaration : range-expression)
    statement;
```

*range-declaration:* named variable of element type specified via the sequence in range-expression

*range-expression:* Expression that represents a sequence of elements via iterator pair `begin()`, `end()` or in the form of an intializer list.

**Examples**

```
std::vector<double> v(5);
for (double x: v) std::cout << x; // 00000
for (int x: {1,2,5}) std::cout << x; // 125
for (double& x: v) x=5;
```

## That is indeed cool!

```
#include <iostream>
#include <vector>

int main(){
  std::vector<int> v(10); // Vector of length 10

  for (auto& x: v)
    std::cin >> x;

  for (const auto x: v)
    std::cout << x << " ";
}
```

## For our detailed understanding

*We build a vector class with the same capabilities ourselves!*

On the way we learn about

- RAII (Resource Acquisition is Initialization) and move construction
- Subscript operators and other utilities
- Templates
- Exception Handling
- Functors and lambda expressions

## A class for (double) vectors

```
class Vector{
public:
    // constructors
    Vector(): sz{0}, elem{nullptr} {};
    Vector(std::size_t s): sz{s}, elem{new double[s]} {}
    // destructor
    ~Vector(){
        delete[] elem;
    }
    // (something is missing here)
private:
  std::size_t sz;
  double* elem;
}
```

## Element access

```
class Vector{
    ...
    // getter. pre: 0 <= i < sz;
    double get(std::size_t i) const{
        return elem[i];
    }
    // setter. pre: 0 <= i < sz;
    void set(std::size_t i, double d){
        elem[i] = d;
    }
    // size property
    std::size_t size() const {
        return sz;
    }
}
```

```
class Vector{
public:
  Vector ();
  Vector(std :: size_t s);
  ~Vector();
  double get(std :: size_t i ) const;
  void set(std :: size_t i , double d);
  std :: size_t size () const;
}
```

(Vector Interface)

## What’s the problem here?

```
int main(){
  Vector v(32);
  for (std::size_t i = 0; i!=v.size(); ++i)
    v.set(i, i);
  Vector w = v;
  for (std::size_t i = 0; i!=w.size(); ++i)
    w.set(i, i*i);
  return 0;
}
```

```
class Vector{
public:
  Vector();
  Vector(std::size_t s);
  ~Vector();
  double get(std::size_t i) const;
  void set(std::size_t i, double d);
  std::size_t size() const;
}
```

(Vector Interface)

```
*** Error in ‘vector1’: double free or corruption
(!prev): 0x0000000000d23c20 ***
======= Backtrace: =========
/lib/x86_64-linux-gnu/libc.so.6(+0x777e5)[0x7fe5a5ac97e5]
```

## Rule of Three!

```
class Vector{
...
  public:
  // copy constructor
  Vector(const Vector &v)
    : sz{v.sz}, elem{new double[v.sz]} {
    std::copy(v.elem, v.elem + v.sz, elem);
  }
}
```

```
class Vector{
public:
  Vector();
  Vector(std::size_t s);
  ~Vector();
  Vector(const Vector &v);
  double get(std::size_t i) const;
  void set(std::size_t i, double d);
  std::size_t size() const;
}
```

(Vector Interface)

## Rule of Three!

```
class Vector{
...
  // assignment operator
  Vector& operator=(const Vector& v){
    if (v.elem == elem) return *this;
    if (elem != nullptr) delete[] elem;
    sz = v.sz;
    elem = new double[sz];
    std::copy(v.elem, v.elem+v.sz, elem);
    return *this;
  }
}
```

```
class Vector{
public:
  Vector();
  Vector(std::size_t s);
  ~Vector();
  Vector(const Vector &v);
  Vector& operator=(const Vector&v);
  double get(std::size_t i) const;
  void set(std::size_t i, double d);
  std::size_t size() const;
}
```

(Vector Interface)

Now it is correct, but cumbersome.

## More elegant this way (part 1):

```
public:
// copy constructor
// (with constructor delegation)
Vector(const Vector &v): Vector(v.sz)
{
  std::copy(v.elem, v.elem + v.sz, elem);
}
```

## More elegant this way (part 2):

```
class Vector{
...
  // Assignment operator
  Vector& operator= (const Vector&v){
    Vector cpy(v);
    swap(cpy);
    return *this;
  }
private:
  // helper function
  void swap(Vector& v){
    std::swap(sz, v.sz);
    std::swap(elem, v.elem);
  }
}
```

copy-and-swap idiom: all members of `*this` are exchanged with members of `cpy`. When leaving `operator=`, `cpy` is cleaned up (deconstructed), while the copy of the data of `v` stay in `*this`.

## Syntactic sugar.

Getters and setters are poor. We want a subscript (index) operator.

Overloading! So?

```
class Vector{
...
  double operator[] (std::size_t pos) const{
    return elem[pos];
  }
```

```
~~  void operator[] (std::size_t pos, double value){~~
~~    elem[pos] = value;~~
~~  }~~
~~}~~
```

No!

## Reference types!

```
class Vector{
...
  // for non−const objects
  double& operator[] (std::size_t pos){
    return elem[pos]; // return by reference!
  }
  // for const objects
  const double& operator[] (std::size_t pos) const{
    return elem[pos];
  }
}
```

## So far so good.

```
int main(){
  Vector v(32); // constructor
  for (int i = 0; i<v.size(); ++i)
    v[i] = i; // subscript operator

  Vector w = v; // copy constructor
  for (int i = 0; i<w.size(); ++i)
    w[i] = i*i;

  const auto u = w;
  for (int i = 0; i<u.size(); ++i)
    std::cout << v[i] << ":" << u[i] << " "; // 0:0 1:1 2:4 ...
  return 0;
}
```

```
class Vector{
public:
  Vector();
  Vector(std::size_t s);
  ~Vector();
  Vector(const Vector &v);
  Vector& operator=(const Vector&v);
  const double& operator[] (std::size_t pos) const;
  double& operator[] (std::size_t pos);
  std::size_t size() const;
}
```

## Number copies

How often is v being copied?

```
Vector operator+ (const Vector& l, double r){
    Vector result (l); // copy of l to result
    for (std::size_t i = 0; i < l.size(); ++i)
        result[i] = l[i] + r;
    return result; // deconstruction of result after assignment
}
int main(){
    Vector v(16); // allocation of elems[16]
    v = v + 1;   // copy when assigned!
    return 0;    // deconstruction of v
}
```

v is copied (at least) twice

## Move construction and move assignment

```
class Vector{
...
    // move constructor
    Vector (Vector&& v): Vector() {
        swap(v);
    };
    // move assignment
    Vector& operator=(Vector&& v){
        swap(v);
        return *this;
    };
}
```

```
class Vector{
public:
   Vector ();
   Vector(std::size_t s);
   ~Vector();
   Vector(const Vector &v);
   Vector& operator=(const Vector&v);
   Vector (Vector&& v);
   Vector& operator=(Vector&& v);
   const double& operator[] (std::size_t pos) const;
   double& operator[] (std::size_t pos);
   std::size_t size () const;
}
```

## Explanation

When the source object of an assignment will not continue existing after an assignment the compiler can use the move assignment instead of the assignment operator.[5] Expensive copy operations are then avoided.

Number of copies in the previous example goes down to $1$.

[5]Analogously so for the copy-constructor and the move constructor

## Illustration of the Move-Semantics

```
// nonsense implementation of a "vector" for demonstration purposes
class Vec{
public:
  Vec () {
        std::cout << "default constructor\n";}
  Vec (const Vec&) {
        std::cout << "copy constructor\n";}
  Vec& operator = (const Vec&) {
        std::cout << "copy assignment\n"; return *this;}
  ~Vec() {}
};
```

## How many Copy Operations?

```
Vec operator + (const Vec& a, const Vec& b){
    Vec tmp = a;
    // add b to tmp
    return tmp;
}

int main (){
    Vec f;
    f = f + f + f + f;
}
```

Output

```
default constructor
copy constructor
copy constructor
copy constructor
copy assignment
```

4 copies of the vector

## Illustration of the Move-Semantics

```
// nonsense implementation of a "vector" for demonstration purposes
class Vec{
public:
  Vec () { std::cout << "default constructor\n";}
  Vec (const Vec&) { std::cout << "copy constructor\n";}
  Vec& operator = (const Vec&) {
        std::cout << "copy assignment\n"; return *this;}
  ~Vec() {}
  // new: move constructor and assignment
  Vec (Vec&&) {
        std::cout << "move constructor\n";}
  Vec& operator = (Vec&&) {
        std::cout << "move assignment\n"; return *this;}
};
```

## How many Copy Operations?

```
Vec operator + (const Vec& a, const Vec& b){
    Vec tmp = a;
    // add b to tmp
    return tmp;
}

int main (){
    Vec f;
    f = f + f + f + f;
}
```

Output
```
default constructor
copy constructor
copy constructor
copy constructor
move assignment
```

3 copies of the vector

## How many Copy Operations?

```
Vec operator + (Vec a, const Vec& b){
    // add b to a
    return a;
}

int main (){
    Vec f;
    f = f + f + f + f;
}
```

Output

```
default constructor
copy constructor
move constructor
move constructor
move constructor
move assignment

1 copy of the vector
```

Explanation: move semantics are applied when an x-value (expired value) is assigned. R-value return values of a function are examples of x-values.

http://en.cppreference.com/w/cpp/language/value_category

# How many Copy Operations?

```
void swap(Vec& a, Vec& b){
    Vec tmp = a;
    a=b;
    b=tmp;
}

int main (){
    Vec f;
    Vec g;
    swap(f,g);
}
```

Output

```
default constructor
default constructor
copy constructor
copy assignment
copy assignment
```

3 copies of the vector

# Forcing x-values

```
void swap(Vec& a, Vec& b){
    Vec tmp = std::move(a);
    a=std::move(b);
    b=std::move(tmp);
}
int main (){
    Vec f;
    Vec g;
    swap(f,g);
}
```

Output

```
default constructor
default constructor
move constructor
move assignment
move assignment

0 copies of the vector
```

Explanation: With std::move an l-value expression can be forced into an x-value. Then move-semantics are applied. `http://en.cppreference.com/w/cpp/utility/move`

## `std::swap` & `std::move`

`std::swap` is implemented as above (using templates)

`std::move` can be used to move the elements of a container into another

```
        std::move(va.begin(),va.end(),vb.begin())
```

## Range `for`

We wanted this:

```
Vector v = ...;
for (auto x: v)
  std::cout << x << " ";
```

In order to support this, an iterator must be provided via `begin` and `end` .

## Iterator for the vector

```
class Vector{
...
    // Iterator
    double∗ begin(){
        return elem;
    }
    double∗ end(){
        return elem+sz;
    }
}
```

```
class Vector{
public:
   Vector ();
   Vector(std :: size_t s);
   ~Vector();
   Vector(const Vector &v);
   Vector& operator=(const Vector&v);
   Vector (Vector&& v);
   Vector& operator=(Vector&& v);
   const double& operator[] (std :: size_t pos) const;
   double& operator[] (std :: size_t pos);
   std :: size_t size () const;
   double∗ begin();
   double∗ end();
}
```

(Pointers support iteration)

## Const Iterator for the vector

```
class Vector{
...
      // Const−Iterator
   const double∗ begin() const{
      return elem;
   }
   const double∗ end() const{
      return elem+sz;
   }

}
```

```
class Vector{
public:
  Vector();
  Vector(std::size_t s);
  ~Vector();
  Vector(const Vector &v);
  Vector& operator=(const Vector&v);
  Vector (Vector&& v);
  Vector& operator=(Vector&& v);
  const double& operator[] (std::size_t pos) const;
  double& operator[] (std::size_t pos);
  std::size_t size() const;
  double∗ begin();
  double∗ end();
  const double∗ begin() const;
  const double∗ end() const;
}
```

## Intermediate result

```
Vector Natural(int from, int to){
  Vector v(to-from+1);
  for (auto& x: v) x = from++;
  return v;
}

int main(){
  auto v = Natural(5,12);
  for (auto x: v)
    std::cout << x << " "; // 5 6 7 8 9 10 11 12
  std::cout << std::endl;
            << "sum = "
            << std::accumulate(v.begin(), v.end(),0); // sum = 68
  return 0;
}
```

## Today’s Conclusion

- Use `auto` to infer a type from the initializer.
- X-values are values where the compiler can determine that they go out of scope.
- Use move constructors in order to move X-values instead of copying.
- When you know what you are doing then you can enforce the use of X-Values.
- Subscript operators can be overloaded. In order to write, references are used.
- Behind a ranged `for` there is an iterator working.
- Iteration is supported by implementing an iterator following the syntactic convention of the standard library.

# 7. Sorting I

Simple Sorting

## 7.1 Simple Sorting

Selection Sort, Insertion Sort, Bubblesort [Ottman/Widmayer, Kap. 2.1, Cormen et al, Kap. 2.1, 2.2, Exercise 2.2-2, Problem 2-2

## Problem

**Input:** An array $A = (A[1], ..., A[n])$ with length $n$.

**Output:** a permutation $A'$ of $A$, that is sorted: $A'[i] \leq A'[j]$ for all $1 \leq i \leq j \leq n$.

# Algorithm: IsSorted($A$)

**Input**: Array $A = (A[1], ..., A[n])$ with length $n$.
**Output**: Boolean decision “sorted” or “not sorted”

```
for $i \leftarrow 1$ to $n - 1$ do
    if $A[i] > A[i+1]$ then
        return “not sorted”;
return “sorted”;
```

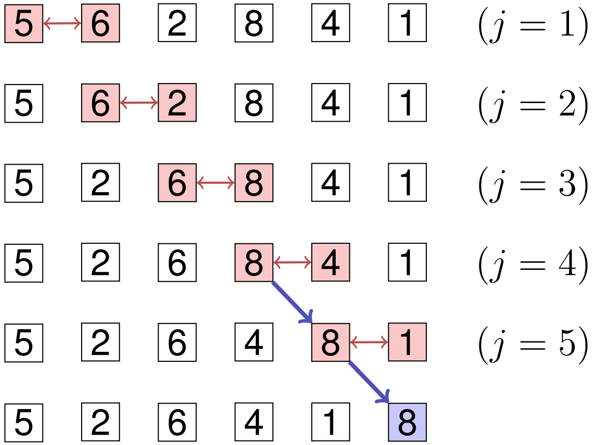
## Observation

IsSorted($A$):“not sorted”, if $A[i] > A[i+1]$ for any $i$.

$\Rightarrow$ idea:

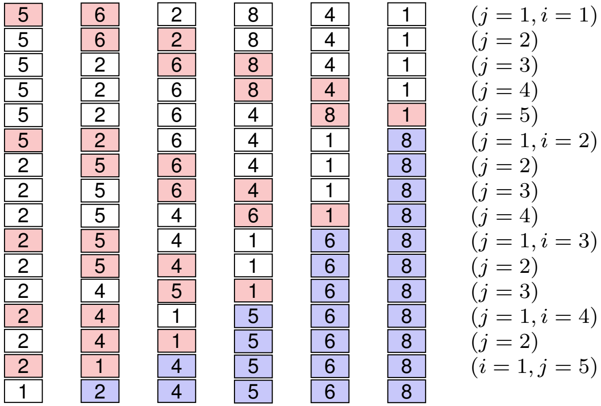
```
for $j \leftarrow 1$ to $n-1$ do
    if $A[j] > A[j+1]$ then
        swap($A[j], A[j+1]$);
```

## Give it a try



- Not sorted! ☹.
- But the greatest element moves to the right
  $\Rightarrow$ new idea! ☺

# Try it out



| | | | | | | |
|---|---|---|---|---|---|---|
| 5 | 6 | 2 | 8 | 4 | 1 | $(j = 1, i = 1)$ |
| 5 | 6 | 2 | 8 | 4 | 1 | $(j = 2)$ |
| 5 | 2 | 6 | 8 | 4 | 1 | $(j = 3)$ |
| 5 | 2 | 6 | 8 | 4 | 1 | $(j = 4)$ |
| 5 | 2 | 6 | 4 | 8 | 1 | $(j = 5)$ |
| 5 | 2 | 6 | 4 | 1 | 8 | $(j = 1, i = 2)$ |
| 2 | 5 | 6 | 4 | 1 | 8 | $(j = 2)$ |
| 2 | 5 | 6 | 4 | 1 | 8 | $(j = 3)$ |
| 2 | 5 | 4 | 6 | 1 | 8 | $(j = 4)$ |
| 2 | 5 | 4 | 1 | 6 | 8 | $(j = 1, i = 3)$ |
| 2 | 5 | 4 | 1 | 6 | 8 | $(j = 2)$ |
| 2 | 4 | 5 | 1 | 6 | 8 | $(j = 3)$ |
| 2 | 4 | 1 | 5 | 6 | 8 | $(j = 1, i = 4)$ |
| 2 | 4 | 1 | 5 | 6 | 8 | $(j = 2)$ |
| 2 | 1 | 4 | 5 | 6 | 8 | $(i = 1, j = 5)$ |
| 1 | 2 | 4 | 5 | 6 | 8 | |

- Apply the procedure iteratively.
- For $A[1, \ldots, n]$, then $A[1, \ldots, n-1]$, then $A[1, \ldots, n-2]$, etc.

## Algorithm: Bubblesort

**Input**: Array $A = (A[1], \dots, A[n])$, $n \geq 0$.
**Output**: Sorted Array $A$

```
for $i \leftarrow 1$ to $n - 1$ do
    for $j \leftarrow 1$ to $n - i$ do
        if $A[j] > A[j+1]$ then
            swap($A[j], A[j+1]$);
```

## Analysis

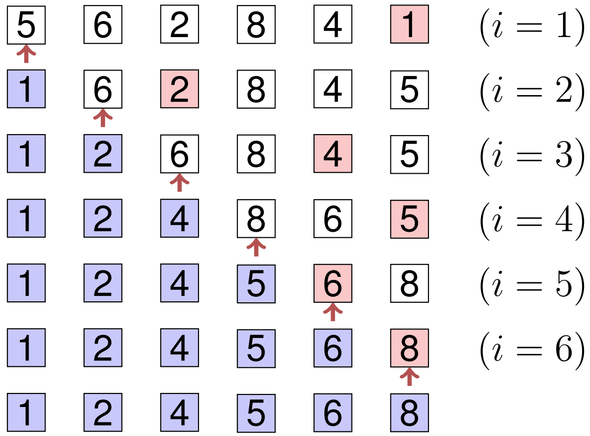
Number key comparisons $\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} = \Theta(n^2)$.

Number swaps in the worst case: $\Theta(n^2)$

? What is the worst case?

! If $A$ is sorted in decreasing order.

## Selection Sort



- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element ($i \to i + 1$). Repeat until all is sorted. ($i = n$)

## Algorithm: Selection Sort

**Input**: Array $A = (A[1], \dots, A[n])$, $n \geq 0$.
**Output**: Sorted Array $A$

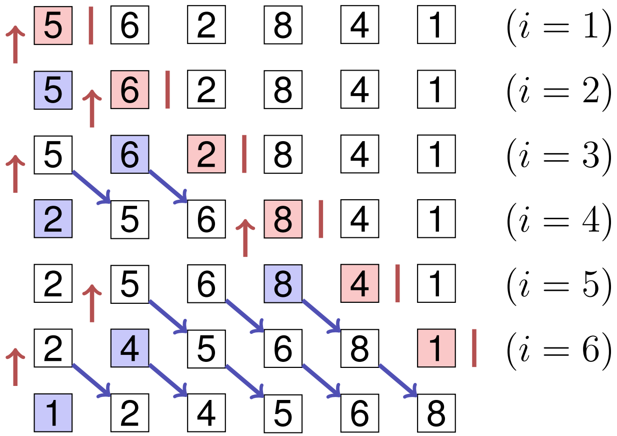
```
for $i \leftarrow 1$ to $n-1$ do
    $p \leftarrow i$
    for $j \leftarrow i+1$ to $n$ do
        if $A[j] < A[p]$ then
            $p \leftarrow j$;
    swap($A[i], A[p]$)
```

## Analysis

Number comparisons in worst case: $\Theta(n^2)$.

Number swaps in the worst case: $n - 1 = \Theta(n)$

# Insertion Sort



- Iterative procedure: $i = 1...n$
- Determine insertion position for element $i$.
- Insert element $i$ array block movement potentially required

## Insertion Sort

? What is the disadvantage of this algorithm compared to sorting by selection?

! Many element movements in the worst case.

? What is the advantage of this algorithm compared to selection sort?

! The search domain (insertion interval) is already sorted. Consequently: binary search possible.

# Algorithm: Insertion Sort

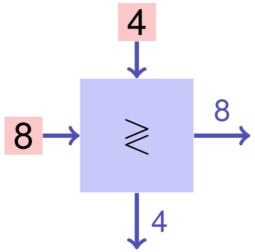
```
Input:     Array $A = (A[1], \dots, A[n])$, $n \geq 0$.
Output:    Sorted Array $A$
for $i \leftarrow 2$ to $n$ do
    $x \leftarrow A[i]$
    $p \leftarrow$ BinarySearch($A[1...i-1], x$); // Smallest $p \in [1, i]$ with $A[p] \geq x$
    for $j \leftarrow i-1$ downto $p$ do
        $A[j+1] \leftarrow A[j]$
    $A[p] \leftarrow x$
```

## Analysis

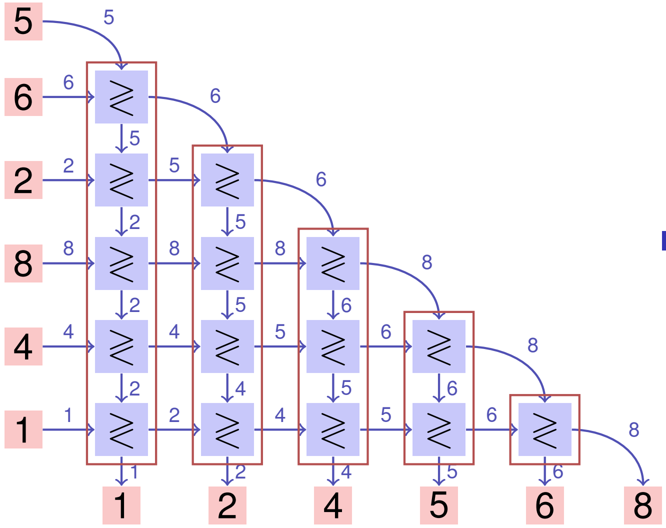
Number comparisons in the worst case:
$\sum_{k=1}^{n-1} a \cdot \log k = a \log((n-1)!) \in \mathcal{O}(n \log n)$.
Number swaps in the worst case $\sum_{k=2}^{n}(k-1) \in \Theta(n^2)$

## Different point of view

Sorting node:

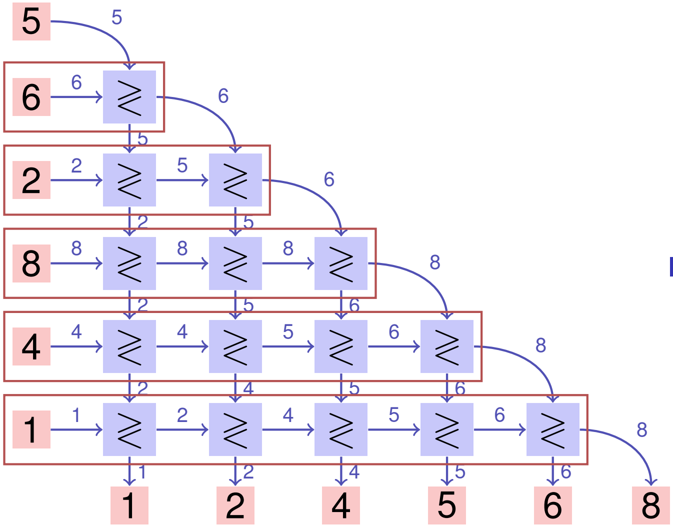


## Different point of view



- Like selection sort
  [and like Bubblesort]

## Different point of view



- Like insertion sort

## Conclusion

In a certain sense, Selection Sort, Bubble Sort and Insertion Sort provide the same kind of sort strategy. Will be made more precise. [6]

[6] In the part about parallel sorting networks. For the sequential code of course the observations as described above still hold.

## Shellsort (Donald Shell 1959)

Insertion sort on subsequences of the form $(A_{k\cdot i})$ ($i \in \mathbb{N}$) with decreasing distances $k$. Last considered distance must be $k = 1$.

Worst-case performance critically depends on the chosen subsequences

- Original concept with sequence $1, 2, 4, 8, ..., 2^k$. Running time: $\mathcal{O}(n^2)$
- Sequence $1, 3, 7, 15, ..., 2^{k-1}$ (Hibbard 1963). $\mathcal{O}(n^{3/2})$
- Sequence $1, 2, 3, 4, 6, 8, ..., 2^p 3^q$ (Pratt 1971). $\mathcal{O}(n \log^2 n)$

## Shellsort

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 1 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 9 | 0 | insertion sort, $k = 4$ |
| 1 | 0 | 7 | 6 | 5 | 4 | 3 | 2 | 9 | 8 | |
| 1 | 0 | 3 | 6 | 5 | 4 | 7 | 2 | 9 | 8 | |
| 1 | 0 | 3 | 2 | 5 | 4 | 7 | 6 | 9 | 8 | |
| 1 | 0 | 3 | 2 | 5 | 4 | 7 | 6 | 9 | 8 | insertion sort, $k = 2$ |
| 1 | 0 | 3 | 2 | 5 | 4 | 7 | 6 | 9 | 8 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | insertion sort, $k = 1$ |

# 8. Sorting II

Heapsort, Quicksort, Mergesort

## 8.1 Heapsort

[Ottman/Widmayer, Kap. 2.3, Cormen et al, Kap. 6]

## Heapsort

Inspiration from selectsort: fast insertion

Inspiration from insertion sort: fast determination of position

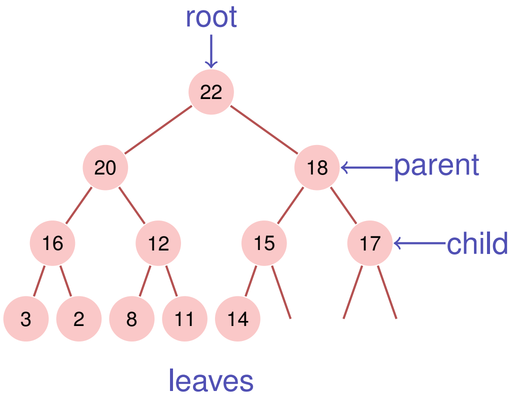
? Can we have the best of both worlds?

! Yes, but it requires some more thinking...

# [Max-]Heap[7]

Binary tree with the following properties

1. complete up to the lowest level
2. Gaps (if any) of the tree in the last level to the right
3. *Heap-Condition:* Max-(Min-)Heap: key of a child smaller (greater) that that of the parent node

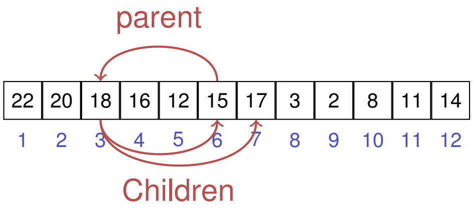


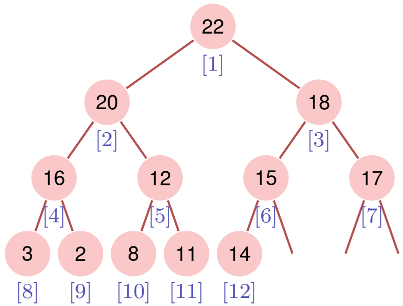
[7]Heap(data structure), not: as in "heap and stack" (memory allocation)

## Heap as Array

Tree $\rightarrow$ Array:

- children$(i) = \{2i, 2i+1\}$
- parent$(i) = \lfloor i/2 \rfloor$



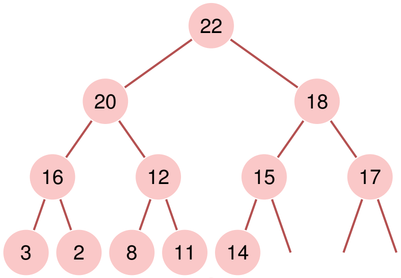


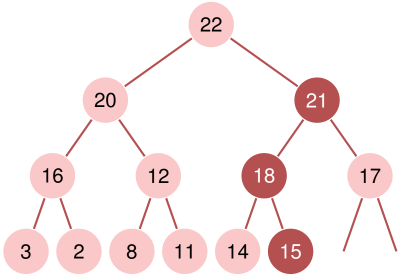
Depends on the starting index[8]

[8]For array that start at $0$: $\{2i, 2i+1\} \rightarrow \{2i+1, 2i+2\}$, $\lfloor i/2 \rfloor \rightarrow \lfloor (i-1)/2 \rfloor$

# Insert

- Insert new element at the first free position. Potentially violates the heap property.
- Reestablish heap property: climb successively
- Worst case number of operations: $\mathcal{O}(\log n)$





## Algorithm Sift-Up($A, m$)

**Input**: Array $A$ with at least $m+1$ and Max-Heap-Structure on $A[0, \dots, m-1]$
**Output**: Array $A$ with Max-Heap-Structure on $A[0, \dots, m]$.

```
$v \leftarrow A[m]$ // value
$c \leftarrow m$ // current position
$p \leftarrow \lfloor (c-1)/2 \rfloor$ // parent node
**while** $c > 0$ and $v > A[p]$ **do**
    $A[c] \leftarrow A[p]$ // Value parent node $\rightarrow$ current node
    $c \leftarrow p$ // parent node $\rightarrow$ current node
    $p \leftarrow \lfloor (c-1)/2 \rfloor$
$A[c] \leftarrow v$ // value $\rightarrow$ current node
```

## Height of a Heap

A complete binary tree with height[9] $h$ provides

$$1 + 2 + 4 + 8 + ... + 2^{h-1} = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

nodes. Thus for a heap with height $h$:

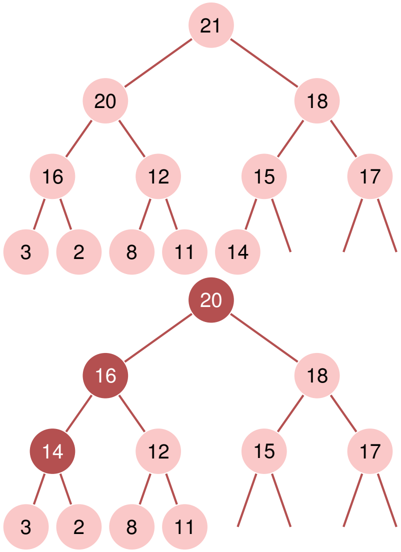
$$\begin{aligned} 2^{h-1} - 1 &< n \leq 2^h - 1 \\ \Leftrightarrow \qquad 2^{h-1} &< n + 1 \leq 2^h \end{aligned}$$

Particularly $h(n) = \lceil \log_2(n+1) \rceil$ and $h(n) \in \Theta(\log n)$.

[9]here: number of edges from the root to a leaf

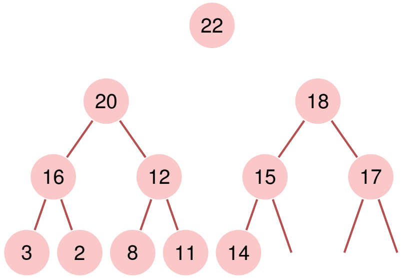
## Remove the maximum

- Replace the maximum by the lower right element
- Reestablish heap property: sink successively (in the direction of the greater child)
- Worst case number of operations: $\mathcal{O}(\log n)$



## Why this is correct: Recursive heap structure

A heap consists of two heaps:



## Algorithm SiftDown($A, i, m$)

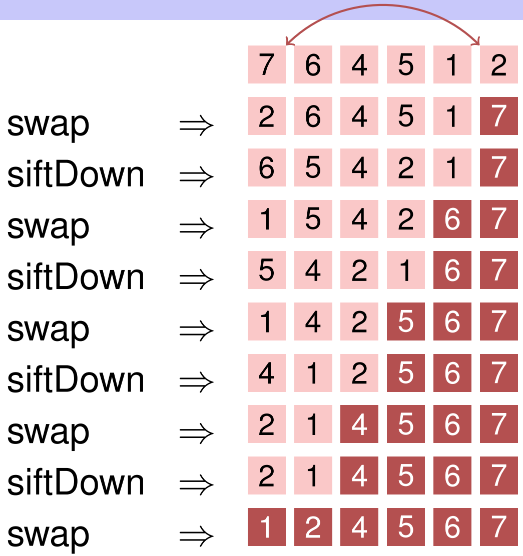
**Input**: Array $A$ with heap structure for the children of $i$. Last element $m$.
**Output**: Array $A$ with heap structure for $i$ with last element $m$.

```
while $2i \leq m$ do
    $j \leftarrow 2i$; // $j$ left child
    if $j < m$ and $A[j] < A[j+1]$ then
        $j \leftarrow j + 1$; // $j$ right child with greater key
    if $A[i] < A[j]$ then
        swap($A[i], A[j]$)
        $i \leftarrow j$; // keep sinking down
    else
        $i \leftarrow m$; // sift down finished
```

# Sort heap

$A[1, ..., n]$ is a Heap.
While $n > 1$

- swap($A[1]$, $A[n]$)
- SiftDown($A, 1, n-1$);
- $n \leftarrow n - 1$



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | 7 | 6 | 4 | 5 | 1 | 2 |
| swap | $\Rightarrow$ | 2 | 6 | 4 | 5 | 1 | 7 |
| siftDown | $\Rightarrow$ | 6 | 5 | 4 | 2 | 1 | 7 |
| swap | $\Rightarrow$ | 1 | 5 | 4 | 2 | 6 | 7 |
| siftDown | $\Rightarrow$ | 5 | 4 | 2 | 1 | 6 | 7 |
| swap | $\Rightarrow$ | 1 | 4 | 2 | 5 | 6 | 7 |
| siftDown | $\Rightarrow$ | 4 | 1 | 2 | 5 | 6 | 7 |
| swap | $\Rightarrow$ | 2 | 1 | 4 | 5 | 6 | 7 |
| siftDown | $\Rightarrow$ | 2 | 1 | 4 | 5 | 6 | 7 |
| swap | $\Rightarrow$ | 1 | 2 | 4 | 5 | 6 | 7 |

## Heap creation

Observation: Every leaf of a heap is trivially a correct heap.

Consequence: Induction from below!

## Algorithm HeapSort($A, n$)

```
Input:    Array $A$ with length $n$.
Output:   $A$ sorted.
// Build the heap.
for $i \leftarrow n/2$ downto $1$ do
    SiftDown($A, i, n$);
// Now $A$ is a heap.
for $i \leftarrow n$ downto $2$ do
    swap($A[1], A[i]$)
    SiftDown($A, 1, i-1$)
// Now $A$ is sorted.
```

## Analysis: sorting a heap

SiftDown traverses at most $\log n$ nodes. For each node $2$ key comparisons. $\Rightarrow$ sorting a heap costs in the worst case $2 \log n$ comparisons.

Number of memory movements of sorting a heap also $\mathcal{O}(n \log n)$.

## Analysis: creating a heap

Calls to siftDown: $n/2$. Thus number of comparisons and movements: $v(n) \in \mathcal{O}(n \log n)$.

But mean length of the sift-down paths is much smaller:

$$v(n) = \sum_{l=0}^{\lfloor \log n \rfloor} \underbrace{2^l}_{\text{number heaps on level l}} \cdot \underbrace{(\lfloor \log n \rfloor - l)}_{\text{height heaps on level l}} = \sum_{k=0}^{\lfloor \log n \rfloor} 2^{\lfloor \log n \rfloor - k} \cdot k$$

$$\leq \sum_{k=0}^{\lfloor \log n \rfloor} \frac{n}{2^k} \cdot k = n \cdot \sum_{k=0}^{\lfloor \log n \rfloor} \frac{k}{2^k} \in \mathcal{O}(\mathbf{n})$$

with $s(x) := \sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2} \quad (0 < x < 1)$ [10] and $s(\frac{1}{2}) = 2$

[10] $f(x) = \frac{1}{1-x} = 1 + x + x^2 ... \Rightarrow f'(x) = \frac{1}{(1-x)^2} = 1 + 2x + ...$

## Intermediate result

Heapsort: $\mathcal{O}(n \log n)$ Comparisons and movements.

? Disadvantages of heapsort?

! Missing locality: heapsort jumps around in the sorted array (negative cache effect).

! Two comparisons required before each necessary memory movement.

## 8.2 Mergesort

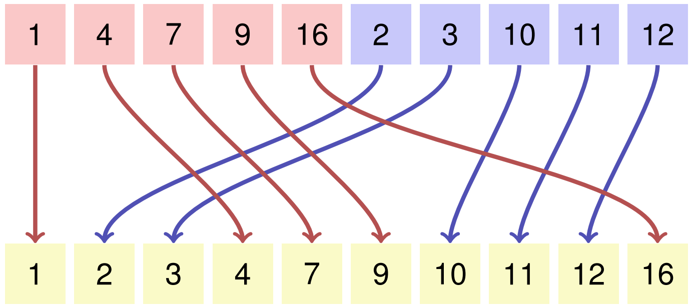
[Ottman/Widmayer, Kap. 2.4, Cormen et al, Kap. 2.3],

## Mergesort

Divide and Conquer!

- Assumption: two halves of the array $A$ are already sorted.
- Minimum of $A$ can be evaluated with two comparisons.
- Iteratively: merge the two presorted halves of $A$ in $\mathcal{O}(n)$.

## Merge



## Algorithm Merge($A, l, m, r$)

```
Input:   Array $A$ with length $n$, indexes $1 \le l \le m \le r \le n$.
         $A[l, \dots, m]$, $A[m+1, \dots, r]$ sorted
Output:  $A[l, \dots, r]$ sorted
$B \leftarrow$ new Array$(r - l + 1)$
$i \leftarrow l$; $j \leftarrow m + 1$; $k \leftarrow 1$
while $i \le m$ and $j \le r$ do
    if $A[i] \le A[j]$ then $B[k] \leftarrow A[i]$; $i \leftarrow i + 1$
    else $B[k] \leftarrow A[j]$; $j \leftarrow j + 1$
    $k \leftarrow k + 1$;
while $i \le m$ do $B[k] \leftarrow A[i]$; $i \leftarrow i + 1$; $k \leftarrow k + 1$
while $j \le r$ do $B[k] \leftarrow A[j]$; $j \leftarrow j + 1$; $k \leftarrow k + 1$
for $k \leftarrow l$ to $r$ do $A[k] \leftarrow B[k - l + 1]$
```

## Correctness

Hypothesis: after $k$ iterations of the loop in line 3 $B[1,\dots,k]$ is sorted and $B[k] \le A[i]$, if $i \le m$ and $B[k] \le A[j]$ if $j \le r$.

Proof by induction:
*Base case:* the empty array $B[1,\dots,0]$ is trivially sorted.
*Induction step* ($k \to k+1$):

- wlog $A[i] \le A[j]$, $i \le m, j \le r$.
- $B[1,\dots,k]$ is sorted by hypothesis and $B[k] \le A[i]$.
- After $B[k+1] \leftarrow A[i]$ $B[1,\dots,k+1]$ is sorted.
- $B[k+1] = A[i] \le A[i+1]$ (if $i+1 \le m$) and $B[k+1] \le A[j]$ if $j \le r$.
- $k \leftarrow k+1, i \leftarrow i+1$: Statement holds again.

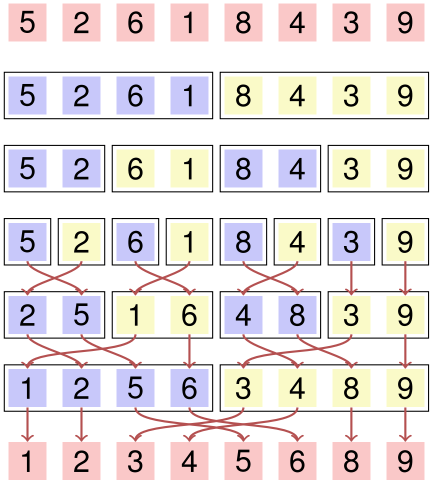
## Analysis (Merge)

**Lemma**

*If: array $A$ with length $n$, indexes $1 \leq l < r \leq n$. $m = \lfloor (l+r)/2 \rfloor$ and $A[l, \dots, m]$, $A[m+1, \dots, r]$ sorted.*
*Then: in the call of Merge($A, l, m, r$) a number of $\Theta(r - l)$ key movements and comparisons are executed.*

Proof: straightforward(Inspect the algorithm and count the operations.)

## Mergesort



Split

Split

Split

Merge

Merge

Merge

## Algorithm (recursive 2-way) Mergesort($A, l, r$)

**Input**: Array $A$ with length $n$. $1 \leq l \leq r \leq n$
**Output**: Array $A[l, \dots, r]$ sorted.

```
if $l < r$ then
    $m \leftarrow \lfloor (l + r)/2 \rfloor$      // middle position
    Mergesort($A, l, m$)          // sort lower half
    Mergesort($A, m + 1, r$)      // sort higher half
    Merge($A, l, m, r$)           // Merge subsequences
```

## Analysis

Recursion equation for the number of comparisons and key movements:

$$T(n) = T(\left\lceil \frac{n}{2} \right\rceil) + T(\left\lfloor \frac{n}{2} \right\rfloor) + \Theta(n) \in \Theta(n \log n)$$

# Algorithm StraightMergesort($A$)

*Avoid recursion:* merge sequences of length $1, 2, 4, \ldots$ directly

```
Input:    Array $A$ with length $n$
Output:   Array $A$ sorted
$length \leftarrow 1$
while $length < n$ do                       // Iterate over lengths $n$
    $r \leftarrow 0$
    while $r + length < n$ do               // Iterate over subsequences
        $l \leftarrow r + 1$
        $m \leftarrow l + length - 1$
        $r \leftarrow \min(m + length, n)$
        Merge($A, l, m, r$)
    $length \leftarrow length \cdot 2$
```

## Analysis

Like the recursive variant, the straight 2-way mergesort always executes a number of $\Theta(n \log n)$ key comparisons and key movements.

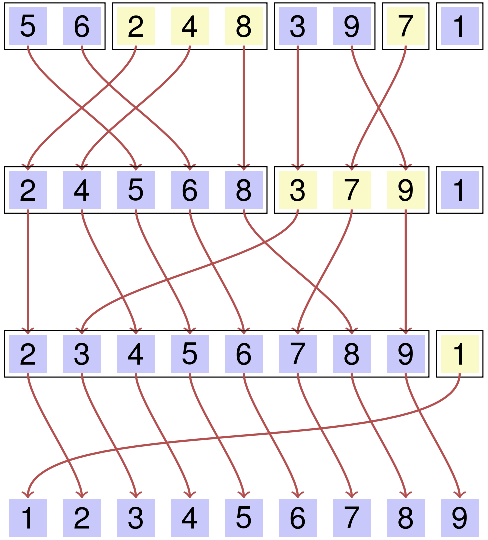
## Natural 2-way mergesort

Observation: the variants above do not make use of any presorting and always execute $\Theta(n \log n)$ memory movements.

? How can partially presorted arrays be sorted better?

! Recursive merging of previously sorted parts (*runs) of* $A$.

## Natural 2-way mergesort



## Algorithm NaturalMergesort($A$)

```
Input:      Array $A$ with length $n > 0$
Output:     Array $A$ sorted
repeat
    $r \leftarrow 0$
    while $r < n$ do
        $l \leftarrow r + 1$
        $m \leftarrow l$; while $m < n$ and $A[m+1] \geq A[m]$ do $m \leftarrow m + 1$
        if $m < n$ then
            $r \leftarrow m + 1$; while $r < n$ and $A[r+1] \geq A[r]$ do $r \leftarrow r + 1$
            Merge($A, l, m, r$);
        else
            $r \leftarrow n$
until $l = 1$
```

## Analysis

? Is it also asymptotically better than StraightMergesort on average?

! No. Given the assumption of pairwise distinct keys, on average there are $n/2$ positions $i$ with $k_i > k_{i+1}$, i.e. $n/2$ runs. Only one iteration is saved on average.

Natural mergesort executes in the worst case and on average a number of $\Theta(n \log n)$ comparisons and memory movements.

## 8.3 Quicksort

[Ottman/Widmayer, Kap. 2.2, Cormen et al, Kap. 7]

## Quicksort

? What is the disadvantage of Mergesort?

! Requires additional $\Theta(n)$ storage for merging.

? How could we reduce the merge costs?

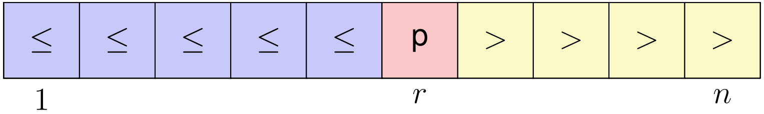
! Make sure that the left part contains only smaller elements than the right part.

? How?

! Pivot and Partition!

## Use a pivot

1. Choose a (an arbitrary) *pivot* $p$
2. Partition $A$ in two parts, one part $L$ with the elements with $A[i] \leq p$ and another part $R$ with $A[i] > p$
3. Quicksort: Recursion on parts L and R



## Algorithm Partition($A[l..r], p$)

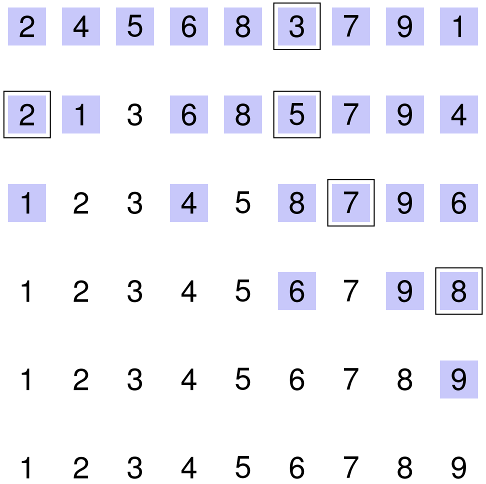
**Input:** Array $A$, that contains the pivot $p$ in the interval $[l, r]$ at least once.
**Output:** Array $A$ partitioned in $[l..r]$ around $p$. Returns position of $p$.

```
while $l \leq r$ do
    while $A[l] < p$ do
        $l \leftarrow l + 1$
    while $A[r] > p$ do
        $r \leftarrow r - 1$
    swap($A[l]$, $A[r]$)
    if $A[l] = A[r]$ then
        $l \leftarrow l + 1$
return l-1
```

## Algorithm Quicksort($A[l,\dots,r]$)

```
Input:    Array $A$ with length $n$. $1 \le l \le r \le n$.
Output:   Array $A$, sorted between $l$ and $r$.
if $l < r$ then
    Choose pivot $p \in A[l,\dots,r]$
    $k \leftarrow$ Partition($A[l,\dots,r], p$)
    Quicksort($A[l,\dots,k-1]$)
    Quicksort($A[k+1,\dots,r]$)
```

# Quicksort (arbitrary pivot)



## Analysis: number comparisons

*Worst case.* Pivot = min or max; number comparisons:

$$T(n) = T(n-1) + c \cdot n,\ T(1) = 0 \quad \Rightarrow \quad T(n) \in \Theta(n^2)$$

## Analysis: number swaps

Result of a call to partition (pivot 3):

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 3 | 6 | 8 | 5 | 7 | 9 | 4 |

? How many swaps have taken place?

! 2. The maximum number of swaps is given by the number of keys in the smaller part.

## Analysis: number swaps

*Thought experiment*

- Each key from the smaller part pays a coin when it is being swapped.
- After a key has paid a coin the domain containing the key decreases to half its previous size.
- Every key needs to pay at most $\log n$ coins. But there are only $n$ keys.

*Consequence:* there are $\mathcal{O}(n \log n)$ key swaps in the worst case.

## Randomized Quicksort

Despite the worst case running time of $\Theta(n^2)$, quicksort is used practically very often.

Reason: quadratic running time unlikely provided that the choice of the pivot and the pre-sorting are not very disadvantageous.

Avoidance: randomly choose pivot. Draw uniformly from $[l, r]$.

## Analysis (randomized quicksort)

Expected number of compared keys with input length $n$:

$$T(n) = (n-1) + \frac{1}{n}\sum_{k=1}^{n}\left(T(k-1) + T(n-k)\right),\ T(0) = T(1) = 0$$

Claim $T(n) \leq 4n\log n$.

Proof by induction:

*Base case* straightforward for $n = 0$ (with $0\log 0 := 0$) and for $n = 1$.

*Hypothesis:* $T(n) \leq 4n\log n$ for some $n$.

*Induction step:* $(n-1 \rightarrow n)$

## Analysis (randomized quicksort)

$$
\begin{aligned}
T(n) &= n-1+\frac{2}{n}\sum_{k=0}^{n-1}T(k) \overset{\mathsf{H}}{\leq} n-1+\frac{2}{n}\sum_{k=0}^{n-1}4k\log k\\
&= n-1+\sum_{k=1}^{n/2}4k\underbrace{\log k}_{\leq \log n-1}+\sum_{k=n/2+1}^{n-1}4k\underbrace{\log k}_{\leq \log n}\\
&\leq n-1+\frac{8}{n}\left((\log n-1)\sum_{k=1}^{n/2}k+\log n\sum_{k=n/2+1}^{n-1}k\right)\\
&= n-1+\frac{8}{n}\left((\log n)\cdot\frac{n(n-1)}{2}-\frac{n}{4}\left(\frac{n}{2}+1\right)\right)\\
&= 4n\log n-4\log n-3 \leq 4n\log n
\end{aligned}
$$

## Analysis (randomized quicksort)

**Theorem**

*On average randomized quicksort requires $\mathcal{O}(n \cdot \log n)$ comparisons.*

## Practical Considerations

Worst case recursion depth $n-1$[11]. Then also a memory consumption of $\mathcal{O}(n)$.

Can be avoided: recursion only on the smaller part. Then guaranteed $\mathcal{O}(\log n)$ worst case recursion depth and memory consumption.

[11] stack overflow possible!

## Quicksort with logarithmic memory consumption

```
Input:    Array $A$ with length $n$. $1 \le l \le r \le n$.
Output:   Array $A$, sorted between $l$ and $r$.
while $l < r$ do
    Choose pivot $p \in A[l,\dots,r]$
    $k \leftarrow$ Partition$(A[l,\dots,r], p)$
    if $k - l < r - k$ then
        Quicksort$(A[l,\dots,k-1])$
        $l \leftarrow k + 1$
    else
        Quicksort$(A[k+1,\dots,r])$
        $r \leftarrow k - 1$
```

The call of Quicksort($A[l,\dots,r]$) in the original algorithm has moved to iteration (tail recursion!): the if-statement became a while-statement.

## Practical Considerations.

- Practically the pivot is often the median of three elements. For example: Median3($A[l], A[r], A[\lfloor l + r/2 \rfloor]$).
- There is a variant of quicksort that requires only constant storage. Idea: store the old pivot at the position of the new pivot.
- Complex divide-and-conquer algorithms often use a trivial ($\Theta(n^2)$) algorithm as base case to deal with small problem sizes.

## 8.4 Appendix

Derivation of some mathematical formulas

## $\log n! \in \Theta(n \log n)$

$$\log n! = \sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n = n \log n$$

$$\begin{aligned}
\sum_{i=1}^{n} \log i &= \sum_{i=1}^{\lfloor n/2 \rfloor} \log i + \sum_{\lfloor n/2 \rfloor + 1}^{n} \log i \\
&\geq \sum_{i=2}^{\lfloor n/2 \rfloor} \log 2 + \sum_{\lfloor n/2 \rfloor + 1}^{n} \log \frac{n}{2} \\
&= (\underbrace{\lfloor n/2 \rfloor}_{> n/2 - 1} - 2 + 1) + (\underbrace{n - \lfloor n/2 \rfloor}_{\geq n/2})(\log n - 1) \\
&> \frac{n}{2} \log n - 2.
\end{aligned}$$

## $[n! \in o(n^n)]$

$$
\begin{aligned}
n \log n &\geq \sum_{i=1}^{\lfloor n/2 \rfloor} \log 2i + \sum_{i=\lfloor n/2 \rfloor+1}^{n} \log i \\
&= \sum_{i=1}^{n} \log i + \left\lfloor \frac{n}{2} \right\rfloor \log 2 \\
&> \sum_{i=1}^{n} \log i + n/2 - 1 = \log n! + n/2 - 1
\end{aligned}
$$

$$
n^n = 2^{n \log_2 n} \geq 2^{\log_2 n!} \cdot 2^{n/2} \cdot 2^{-1} = n! \cdot 2^{n/2-1}
$$

$$
\Rightarrow \frac{n!}{n^n} \leq 2^{-n/2+1} \xrightarrow{n \to \infty} 0 \Rightarrow n! \in o(n^n) = \mathcal{O}(n^n) \backslash \Omega(n^n)
$$

## [Even $n! \in o((n/c)^n)\ \forall\, 0 < c < e$ ]

Konvergenz oder Divergenz von $f_n = \frac{n!}{(n/c)^n}$.

Ratio Test

$$\frac{f_{n+1}}{f_n} = \frac{(n+1)!}{\left(\frac{n+1}{c}\right)^{n+1}} \cdot \frac{\left(\frac{n}{c}\right)^n}{n!} = c \cdot \left(\frac{n}{n+1}\right)^n \longrightarrow c \cdot \frac{1}{e} \lessgtr 1 \text{ if } c \lessgtr e$$

because $\left(1+\frac{1}{n}\right)^n \to e$. Even the series $\sum_{i=1}^{n} f_n$ converges / diverges for $c \lessgtr e$.

$f_n$ diverges for $c = e$, because (Stirling): $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.

## [ Ratio Test]

Ratio test for a sequence $(f_n)_{n\in\mathbb{N}}$: If $\frac{f_{n+1}}{f_n} \xrightarrow[n\to\infty]{} \lambda$, then the sequence $f_n$ and the series $\sum_{i=1}^{n} f_i$

- converge, if $\lambda < 1$ and
- diverge, if $\lambda > 1$.

## [ Ratio Test Derivation ]

Ratio test is implied by Geometric Series

$$S_n(r) := \sum_{i=0}^{n} r^i = \frac{1-r^{n+1}}{1-r}.$$

converges for $n \to \infty$ if and only if $-1 < r < 1$.

Let $0 \leq \lambda < 1$:

$$\begin{aligned} &\forall \varepsilon > 0\, \exists n_0 : f_{n+1}/f_n < \lambda + \varepsilon\, \forall n \geq n_0 \\ \Rightarrow &\exists \varepsilon > 0, \exists n_0 : f_{n+1}/f_n \leq \mu < 1\, \forall n \geq n_0 \end{aligned}$$

Thus

$$\sum_{n=n_0}^{\infty} f_n \leq f_{n_0} \cdot \sum_{n=n_0}^{\infty} \cdot \mu^{n-n_0} \quad \text{konvergiert.}$$

(Analogously for divergence)

# 9. C++ advanced (II): Templates

## What do we learn today?

- templates of classes
- function templates
- Specialization
- templates with values

## Motivation

Goal: generic vector class and functionality.

**Examples**

```
Vector<double> vd(10);
Vector<int> vi(10);
Vector<char> vi(20);

auto nd = vd * vd; // norm (vector of double)
auto ni = vi * vi; // norm (vector of int)
```

## Types as Template Parameters

1. In the concrete implementation of a class replace the type that should become generic (in our example: `double`) by a representative element, e.g. `T`.
2. Put in front of the class the construct `template<typename T>`[12] Replace `T` by the representative name).

The construct `template<typename T>` can be understood as “for all types `T`”.

[12] equally: `template<class T>`

## Types as Template Parameters

```
template <typename ElementType>
class Vector{
    std::size_t size;
    ElementType* elem;
public:
    ...
    Vector(std::size_t s):
        size{s},
        elem{new ElementType[s]}{}
    ...
    ElementType& operator[](std::size_t pos){
        return elem[pos];
    }
    ...
}
```

## Template Instances

`Vector<typeName>` generates a type instance `Vector` with `ElementType=typeName`.
Notation: Instantiation

**Examples**

```
Vector<double> x;          // vector of double
Vector<int> y;             // vector of int
Vector<Vector<double>> x; // vector of vector of double
```

## Type-checking

Templates are basically replacement rules at instantiation time and during compilation. The compiler always checks as little as necessary and as much as possible.

## Example

```
template <typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r):left{l}, right{r}{}
    T min(){
        return left < right ? left : right;
    }
};
```

```
Pair<int> a(10,20); // ok
auto m = a.min(); // ok
Pair<Pair<int>> b(a,Pair<int>(20,30)); // ok
auto n = b.min(); no match for operator< !
```

## Generic Programming

Generic components should be developed rather as a generalization of one or more examples than from first principles.

```
template <typename T>
class Vector{
public:
   Vector ();
   Vector(std::size_t);
   ~Vector();
   Vector(const Vector&);
   Vector& operator=(const Vector&);
   Vector (Vector&&);
   Vector& operator=(Vector&&);
   const T& operator[] (std::size_t) const;
   T& operator[] (std::size_t);
   std::size_t size () const;
   T* begin();
   T* end();
   const T* begin() const;
   const T* end() const;
}
```

## Function Templates

1. To make a concrete implementation generic, replace the specific type (e.g. int) with a name, e.g. `T`,
2. Put in front of the function the construct `template<typename T>`[13] (Replace `T` by the chosen name)

[13] equally: `template<class T>`

## Function Templates

```
template <typename T>
void swap(T& x, T&y){
    T temp = x;
    x = y;
    y = temp;
}
```

The actual parameters' types determine the version of the function that is (compiled) and used:

```
int x=5;
int y=6;
swap(x,y); // calls swap with T=int
```

## Limits of Magic

```
template <typename T>
void swap(T& x, T&y){
    T temp = x;
    x = y;
    y = temp;
}
```

An inadmissible version of the function is not generated:

```
int x=5;
double y=6;
swap(x,y); // error:  no matching function for ...
```

## .. also with operators

```
template <typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r):left{l}, right{r}{}
    T min(){ return left < right? left: right; }
    std::ostream& print (std::ostream& os) const{
        return os << "("<< left << "," << right<< ")";
    }
};

template <typename T>
std::ostream& operator<< (std::ostream& os, const Pair<T>& pair){
    return pair.print(os);
}
```

```
Pair<int> a(10,20); // ok
std::cout << a; // ok
```

## Useful!

```
// Output of an arbitrary container
template <typename T>
void output(const T& t){
    for (auto x: t)
        std::cout << x << " ";
    std::cout << "\n";
}

int main(){
  std::vector<int> v={1,2,3};
  output(v); // 1 2 3
}
```

## Explicit Type

```
// input of an arbitrary pair
template <typename T>
Pair<T> read(){
       T left;
       T right;
       std::cin << left << right;
       return Pair<T>(left,right);
}
...

auto p = read<double>();
```

If the type of a template instantiation cannot be inferred, it has to be provided explicitly.

## Powerful!

```
template <typename T> // square number
T sq(T x){
    return x*x;
}
template <typename Container, typename F>
void  apply(Container& c, F f){ // x <- f(x) forall x in c
    for(auto& x: c)
        x = f(x);
}
int main(){
  std::vector<int> v={1,2,3};
  apply(v,sq<int>);
  output(v); // 1 4 9
}
```

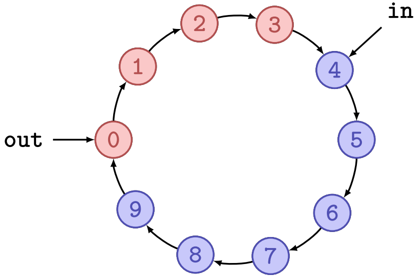
## Specialization

```
template <>
class Pair<bool>{
    short both;
public:
    Pair(bool l, bool r):both{(l?1:0) + (r?2:0)} {};
    std::ostream& print (std::ostream& os) const{
        return os << "("<< both % 2 << "," << both /2 << ")";
    }
};
```

```
Pair<int> i(10,20); // ok -- generic template
std::cout << i << std::endl; // (10,20);
Pair<bool> b(true, false); // ok -- special bool version
std::cout << b << std::endl; // (1,0)
```

## Template Parameterization with Values

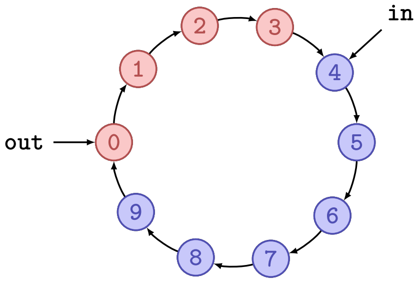
```
template <typename T, int size>
class CircularBuffer{
  T buf[size] ;
  int in; int out;
public:
  CircularBuffer():in{0},out{0}{};
  bool empty(){
    return in == out;
  }
  bool full(){
    return (in + 1) % size == out;
  }
  void put(T x); // declaration
  T get();        // declaration
};
```



## Template Parameterization with Values

```
template <typename T, int size>
void CircularBuffer<T,size>::put(T x){
    assert(!full());
    buf[in] = x;
    in = (in + 1) % size;
}

template <typename T, int size>
T CircularBuffer<T,size>::get(){
    assert(!empty());
    T x = buf[out];
    out = (out + 1) % size;
    return x;
}
```

$\longleftarrow$ Potential for optimization if size $= 2^k$.



## Memory Management

**Guideline “Dynamic Memory”**

For each `new` there is a matching `delete`!

Avoid:

- Memory leaks: old objects that occupy memory
- Pointer to released objects: *dangling pointers*
- Releasing an object more than once using `delete`.

How?

## Smart Pointers

- Can make sure that an object is deleted if and only if it is not used any more
- Are based on the RAII (Resouce Acquisition is Initialization) paradigm.
- Can be used instead of a normal pointer: are implemented as class templates.
- There are `std::unique_ptr<>`, `std::shared_ptr<>` (and `std::weak_ptr<>`)

```
std::unique_ptr<Node> nodeU(new Node()); // unique pointer
std::shared_ptr<Node> nodeS(new Node()); // shared pointer
```

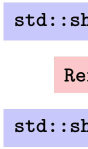
## Unique Pointer

- The deconstructor of a `std::unique_ptr<T>` deletes the pointer contained.
- `std::unique_ptr<T>` has exclusive ownership for the contained pointer on `T`.
- Copy constructor and assignment operator are deleted. A unique pointer cannot be copied by value. The move constructor is implemented: the pointer can be moved.
- No additional runtime overhead in comparison to a normal pointer

```
std::unique_ptr<Node> nodeU(new Node()); // unique pointer
std::unique_ptr<Node> node2 = std::move(nodeU); // ok
std::unique_ptr<Node> node3 = nodeU; // error
```

## Shared Pointer

- `std::shared_ptr<T>` Counts the numbers of owners of a pointer (reference count). When reference count goes to 0, the pointer is deleted.
- Shared pointers can be copied.
- Shared pointers provide additional space- and runtime overhead: they manage the reference counter at runtime and contain a pointer to the reference.



```
std::shared_ptr<Node> nodeS(new Node()); // shared pointer, rc = 1
std::shared_ptr<Node> node2 = std::move(nodeS); // ok, rc unchanged
std::shared_ptr<Node> node3 = node2; // ok, rc = 2
```

## Smart Pointers

Some rules

- Never call `delete` on a pointer contained in a smart pointer.
- Avoid `new`, instead:

  ```
  std::unique_ptr<Node> nodeU = std::make_unique<Node>()
  std::shared_ptr<Node> nodeS = std::make_shared<Node>()
  ```

- Where possible, use `std::unique_ptr`
- If using `std::shared_ptr` make sure there are no cycles in the pointer graph.

# 10. Sorting III

Lower bounds for the comparison based sorting, radix- and bucket-sort

## 10.1 Lower bounds for comparison based sorting

[Ottman/Widmayer, Kap. 2.8, Cormen et al, Kap. 8.1]

## Lower bound for sorting

Up to here: worst case sorting takes $\Omega(n \log n)$ steps.

Is there a better way? No:

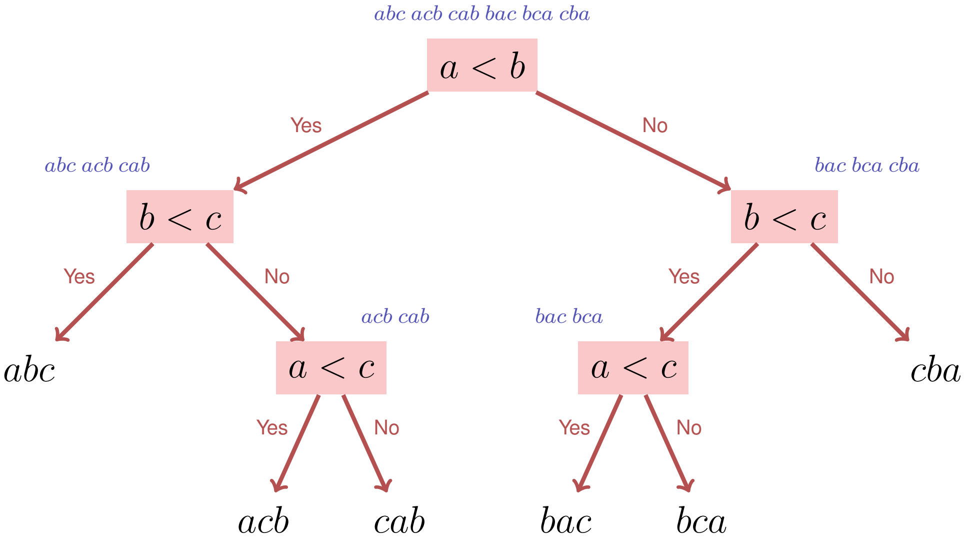
**Theorem**

*Sorting procedures that are based on comparison require in the worst case and on average at least $\Omega(n \log n)$ key comparisons.*

## Comparison based sorting

- An algorithm must identify the correct one of $n!$ permutations of an array $(A_i)_{i=1,\dots,n}$ .
- At the beginning the algorithm know nothing about the array structure.
- We consider the knowledge gain of the algorithm in the form of a decision tree:
  - Nodes contain the remaining possibilities.
  - Edges contain the decisions.

## Decision tree



## Decision tree

A binary tree with $L$ leaves provides $K = L - 1$ inner nodes.[14]

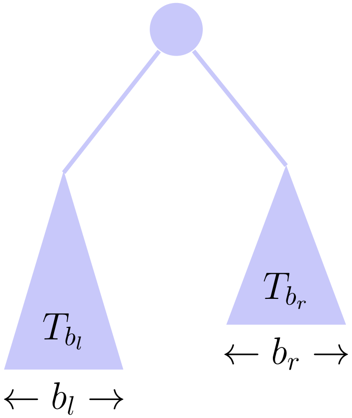
The height of a binary tree with $L$ leaves is at least $\log_2 L$. $\Rightarrow$ The heigh of the decision tree $h \geq \log n! \in \Omega(n \log n)$.

Thus the length of the longest path in the decision tree $\in \Omega(n \log n)$.

Remaining to show: mean length $M(n)$ of a path $M(n) \in \Omega(n \log n)$.

[14]Proof: start with emtpy tree ($K = 0, L = 1$). Each added node replaces a leaf by two leaves, i.e.}
$K \to K + 1 \Rightarrow L \to L + 1$.

# Average lower bound



- Decision tree $T_n$ with $n$ leaves, average height of a leaf $m(T_n)$
- Assumption $m(T_n) \geq \log n$ not for all $n$.
- Choose smalles $b$ with $m(T_b) < \log b \Rightarrow b \geq 2$
- $b_l + b_r = b$ with $b_l > 0$ und $b_r > 0 \Rightarrow$ $b_l < b, b_r < b \Rightarrow m(T_{b_l}) \geq \log b_l$ und $m(T_{b_r}) \geq \log b_r$

## Average lower bound

Average height of a leaf:

$$\begin{aligned}
m(T_b) &= \frac{b_l}{b}(m(T_{b_l})+1) + \frac{b_r}{b}(m(T_{b_r})+1) \\
&\geq \frac{1}{b}(b_l(\log b_l + 1) + b_r(\log b_r + 1)) = \frac{1}{b}(b_l \log 2b_l + b_r \log 2b_r) \\
&\geq \frac{1}{b}(b \log b) = \log b.
\end{aligned}$$

Contradiction. ■

The last inequality holds because $f(x) = x \log x$ is convex ($f''(x) = 1/x > 0$) and for a convex function it holds that $f((x+y)/2) \leq 1/2 f(x) + 1/2 f(y)$ ($x = 2b_l$, $y = 2b_r$ ).[15] Enter $x = 2b_l$, $y = 2b_r$, and $b_l + b_r = b$.

[15]generally $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y)$ for $0 \leq \lambda \leq 1$.

## 10.2 Radixsort and Bucketsort

Radixsort, Bucketsort [Ottman/Widmayer, Kap. 2.5, Cormen et al, Kap. 8.3]

## Radix Sort

*Sorting based on comparison:* comparable keys ($<$ or $>$, often $=$).
No further assumptions.

*Different idea:* use more information about the keys.

## Assumptions

Assumption: keys representable as words from an alphabet containing $m$ elements.

**Examples**

| | | |
|---|---|---|
| $m = 10$ | decimal numbers | $183 = 183_{10}$ |
| $m = 2$ | dual numbers | $101_2$ |
| $m = 16$ | hexadecimal numbers | $A0_{16}$ |
| $m = 26$ | words | ``INFORMATIK'' |

$m$ is called the radix of the representation.

## Assumptions

- keys = $m$-adic numbers with same length.
- Procedure $z$ for the extraction of digit $k$ in $\mathcal{O}(1)$ steps.

**Example**

$z_{10}(0,85) = 5$
$z_{10}(1,85) = 8$
$z_{10}(2,85) = 0$

## Radix-Exchange-Sort

Keys with radix $2$.

Observation: if for some $k \geq 0$:

$$z_2(i, x) = z_2(i, y) \text{ for all } i > k$$

and

$$z_2(k, x) < z_2(k, y),$$

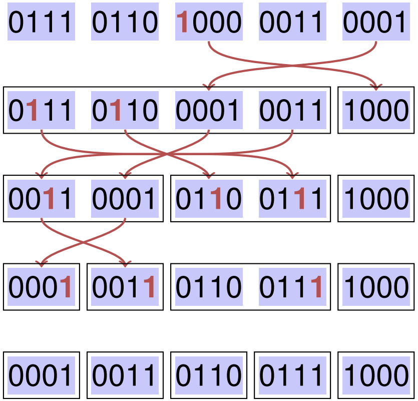
then it holds that $x < y$.

## Radix-Exchange-Sort

Idea:

- Start with a maximal $k$.
- Binary partition the data sets with $z_2(k, \cdot) = 0$ vs. $z_2(k, \cdot) = 1$ like with quicksort.
- $k \leftarrow k - 1$.

# Radix-Exchange-Sort



## Algorithm RadixExchangeSort($A, l, r, b$)

**Input**: Array $A$ with length $n$, left and right bounds $1 \leq l \leq r \leq n$, bit position $b$

**Output**: Array $A$, sorted in the domain $[l, r]$ by bits $[0, \ldots, b]$ .

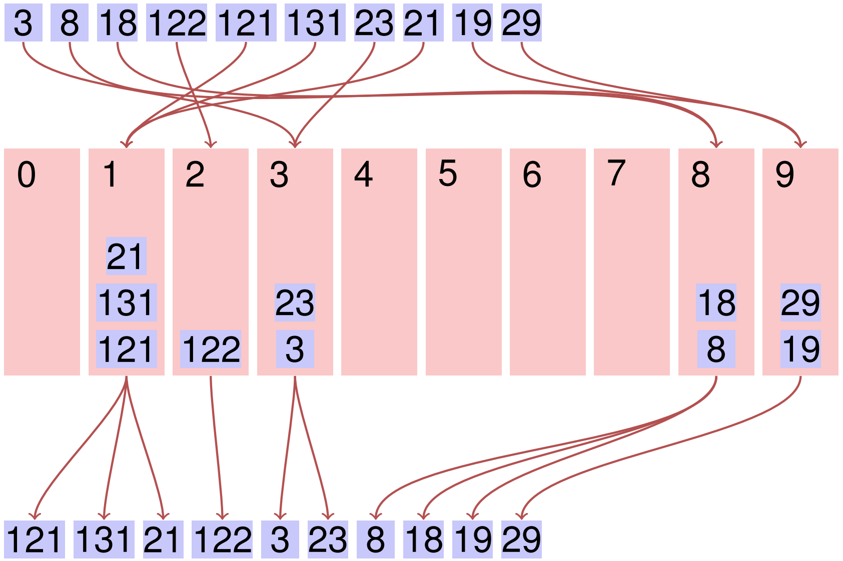
```
if $l < r$ and $b \geq 0$ then
    $i \leftarrow l - 1$
    $j \leftarrow r + 1$
    repeat
        repeat $i \leftarrow i + 1$ until $z_2(b, A[i]) = 1$ or $i \geq j$
        repeat $j \leftarrow j - 1$ until $z_2(b, A[j]) = 0$ or $i \geq j$
        if $i < j$ then swap($A[i], A[j]$)
    until $i \geq j$
    RadixExchangeSort($A, l, i - 1, b - 1$)
    RadixExchangeSort($A, i, r, b - 1$)
```

## Analysis

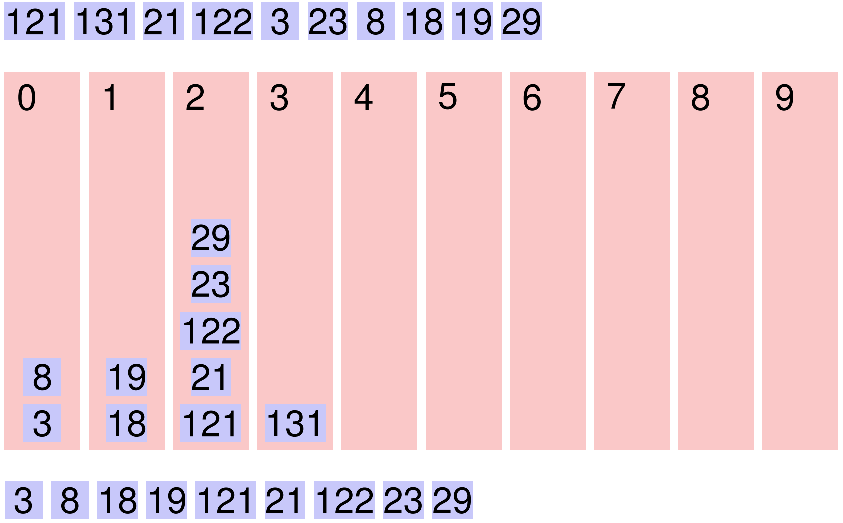
RadixExchangeSort provides recursion with maximal recursion depth = maximal number of digits $p$.

Worst case run time $\mathcal{O}(p \cdot n)$.

## Bucket Sort

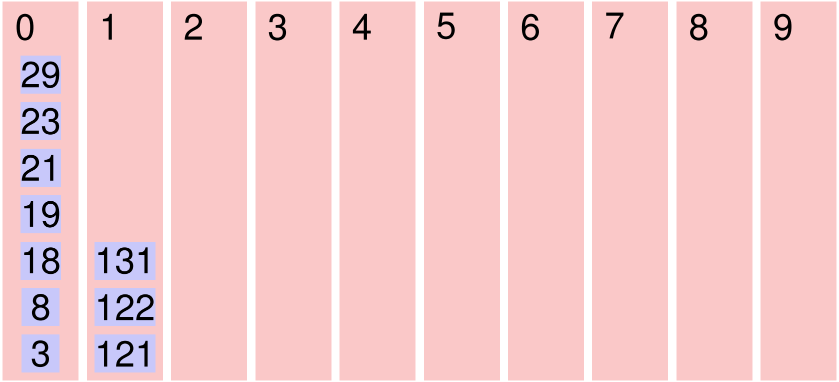

3 8 18 122 121 131 23 21 19 29
0 1 2 3 4 5 6 7 8 9
21
131
121
122
23
3
18
8
29
19
121 131 21 122 3 23 8 18 19 29

## Bucket Sort


121 131 21 122 3 23 8 18 19 29
0
1
2
3
4
5
6
7
8
9
8
3
19
18
29
23
122
21
121
131
3 8 18 19 121 21 122 23 29

# Bucket Sort

3 8 18 19 121 21 122 23 29



3 8 18 19 21 23 29 121 122 131

## implementation details

Bucket size varies greatly. Possibilities

- Linked list or dynamic array for each digit.
- One array of length $n$. compute offsets for each digit in the first iteration.

Assumptions: Input length $n$ , Number bits / integer: $k$ , Number Buckets: $2^b$

Asymptotic running time $\mathcal{O}(\frac{k}{b} \cdot (n + 2^b)$.

For Example: $k = 32$, $2^b = 256$ : $\frac{k}{b} \cdot (n + 2^b) = 4n + 1024$.

# 11. Fundamental Data Structures

Abstract data types stack, queue, implementation variants for linked lists [Ottman/Widmayer, Kap. 1.5.1-1.5.2, Cormen et al, Kap. 10.1.-10.2]

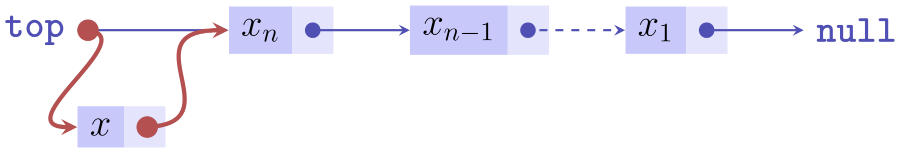
## Abstract Data Types

We recall

A *stack* is an abstract data type (ADR) with operations

- `push`$(x, S)$: Puts element $x$ on the stack $S$.
- `pop`$(S)$: Removes and returns top most element of $S$ or `null`
- `top`$(S)$: Returns top most element of $S$ or `null`.
- `isEmpty`$(S)$: Returns `true` if stack is empty, `false` otherwise.
- `emptyStack`$()$: Returns an empty stack.

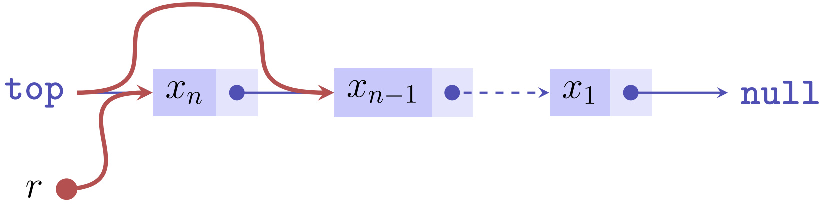
## Implementation Push



**push**$(x, S)$:

1. Create new list element with $x$ and pointer to the value of **top**.
2. Assign the node with $x$ to **top**.

## Implementation Pop



pop$(S)$:

1. If top=null, then return null
2. otherwise memorize pointer $p$ of top in $r$.
3. Set top to $p.next$ and return $r$

## Analysis

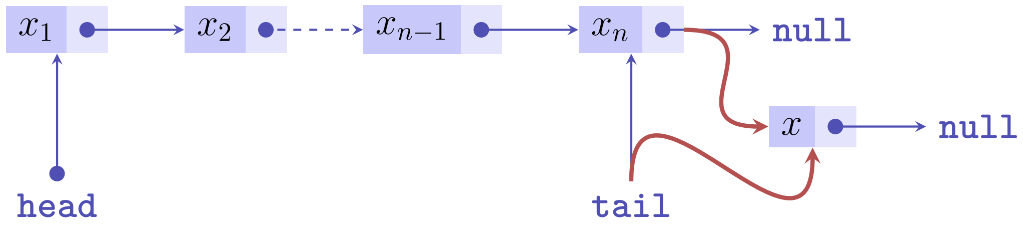
Each of the operations `push`, `pop`, `top` and `isEmpty` on a stack can be executed in $\mathcal{O}(1)$ steps.

# Queue (fifo)

A queue is an ADT with the following operations

- `enqueue`$(x, Q)$: adds $x$ to the tail (=end) of the queue.
- `dequeue`$(Q)$: removes $x$ from the head of the queue and returns $x$ (`null` otherwise)
- `head`$(Q)$: returns the object from the head of the queue (`null` otherwise)
- `isEmpty`$(Q)$: return `true` if the queue is empty, otherwise `false`
- `emptyQueue`$()$: returns empty queue.

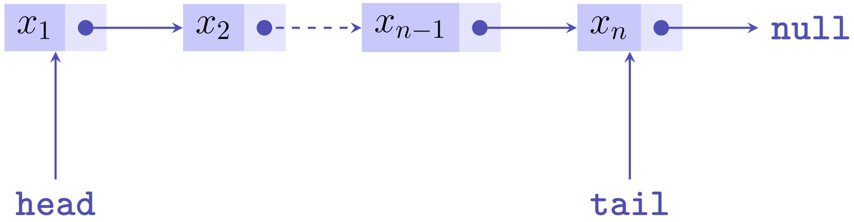
## Implementation Queue



`enqueue`$(x, S)$:

1. Create a new list element with $x$ and pointer to `null`.
2. If `tail` $\neq$ `null`, then set `tail.next` to the node with $x$.
3. Set `tail` to the node with $x$.
4. If `head` $=$ `null`, then set `head` to `tail`.

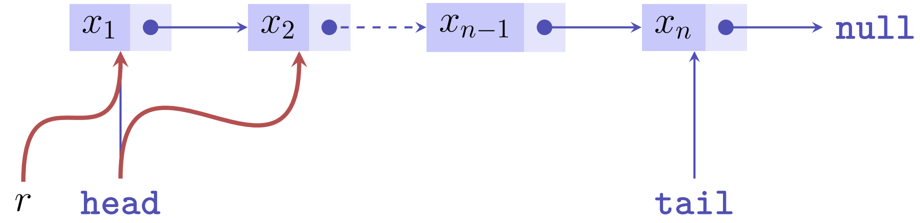
## Invariants



With this implementation it holds that

- either `head` $=$ `tail` $=$ `null`,
- or `head` $=$ `tail` $\neq$ `null` and `head.next` $=$ `null`
- or `head` $\neq$ `null` and `tail` $\neq$ `null` and `head` $\neq$ `tail` and `head.next` $\neq$ `null`.

## Implementation Queue



**dequeue**$(S)$:

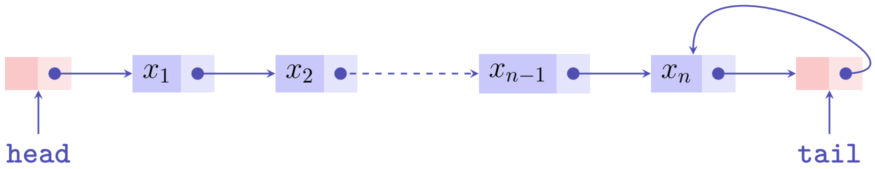
1. Store pointer to **head** in $r$. If $r =$ **null**, then return $r$ .
2. Set the pointer of **head** to **head.next**.
3. Is now **head** $=$ **null** then set tail to **null**.
4. Return the value of $r$.

## Analysis

Each of the operations `enqueue`, `dequeue`, `head` and `isEmpty` on the queue can be executed in $\mathcal{O}(1)$ steps.

## Implementation Variants of Linked Lists

List with dummy elements (sentinels).

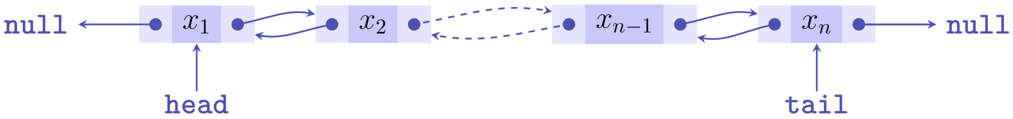


Advantage: less special cases

Variant: like this with pointer of an element stored singly indirect. (Example: pointer to $x_3$ points to $x_2$.)

# Implementation Variants of Linked Lists

Doubly linked list



## Overview

| | enqueue | delete | search | concat |
|---|---|---|---|---|
| (A) | $\Theta(1)$ | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ |
| (B) | $\Theta(1)$ | $\Theta(n)$ | $\Theta(n)$ | $\Theta(1)$ |
| (C) | $\Theta(1)$ | $\Theta(1)$ | $\Theta(n)$ | $\Theta(1)$ |
| (D) | $\Theta(1)$ | $\Theta(1)$ | $\Theta(n)$ | $\Theta(1)$ |

(A) = singly linked
(B) = Singly linked with dummy element at the beginning and the end
(C) = Singly linked with indirect element addressing
(D) = doubly linked

## priority queue

Priority Queue

Operations

- `insert(x,p,Q)`: Enter object $x$ with priority $p$.
- `extractMax(Q)`: Remove and return object $x$ with highest priority.

## Implementation Priority Queue

With a Max Heap

Thus

- `insert` in $\mathcal{O}(\log n)$ and
- `extractMax` in $\mathcal{O}(\log n)$.

# 12. Amortized Analyis

Amortized Analysis: Aggregate Analysis, Account-Method, Potential-Method [Ottman/Widmayer, Kap. 3.3, Cormen et al, Kap. 17]

## Multistack

Multistack adds to the stack operations `push` und `pop`

`multipop(s,S)`: remove the $\min(\text{size}(S), k)$ most recently inserted objects and return them.

Implementation as with the stack. Runtime of `multipop` is $\mathcal{O}(k)$.

## Academic Question

If we execute on a stack with $n$ elements a number of $n$ times `multipop(k,S)` then this costs $\mathcal{O}(n^2)$?

Certainly correct because each `multipop` may take $\mathcal{O}(n)$ steps.

How to make a better estimation?

## Amortized Analysis

- Upper bound: *average* performance of each considered operation in the *worst case*.

$$\frac{1}{n}\sum_{i=1}^{n}\mathrm{cost}(\mathrm{op}_i)$$

- Makes use of the fact that a few expensive operations are opposed to many cheap operations.
- In amortized analysis we search for a credit or a potential function that captures how the cheap operations can “compensate” for the expensive ones.

## Aggregate Analysis

Direct argument: compute a bound for the total number of elementary operations and divide by the total number of operations.

## Aggregate Analysis: (Stack)

- 
-

$$\sum_{i=1}^{n} \text{cost}(\text{op}_i) \leq 2n$$

$$\textit{amortized cost}(\text{op}_i) \leq 2 \in \mathcal{O}(1)$$

## Accounting Method

Model

- The computer is driven with coins: each elementary operation of the machine costs a coin.
- For each operation $op_k$ of a data structure, a number of coins $a_k$ has to be put on an account $A$: $A_k = A_{k-1} + a_k$
- Use the coins from the account $A$ to pay the true costs $t_k$ of each operation.
- The account $A$ needs to provide enough coins in order to pay each of the ongoing operations $op_k$: $A_k - t_k \geq 0 \, \forall k$.

$\Rightarrow a_k$ are the amortized costs of $op_k$.

## Accounting Method (Stack)

- Each call of `push` costs 1 CHF and additionally 1 CHF will be deposited on the account. ($a_k = 2$)
- Each call to `pop` costs 1 CHF and will be paid from the account. ($a_k = 0$)

Account will never have a negative balance.

$a_k \leq 2 \,\forall\, k$, thus: constant amortized costs.

## Potential Method

Slightly different model

- Define a *potential* $\Phi_i$ that is *associated to the state of a data structure* at time $i$.
- The potential shall be used to level out expensive operations und therefore needs to be chosen such that it is increased during the (frequent) cheap operations while it decreases for the (rare) expensive operations.

## Potential Method (Formal)

Let $t_i$ denote the real costs of the operation $op_i$.

Potential function $\Phi_i \geq 0$ to the data structure after $i$ operations.
Requirement: $\Phi_i \geq \Phi_0 \; \forall i$.

of the $i$th operation:

$$a_i := t_i + \Phi_i - \Phi_{i-1}.$$

It holds

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (t_i + \Phi_i - \Phi_{i-1}) = \left( \sum_{i=1}^{n} t_i \right) + \Phi_n - \Phi_0 \geq \sum_{i=1}^{n} t_i.$$

## Example stack

Potential function $\Phi_i$ = number element on the stack.

- `push`$(x, S)$: real costs $t_i = 1$. $\Phi_i - \Phi_{i-1} = 1$. Amortized costs $a_i = 2$.
- `pop`$(S)$: real costs $t_i = 1$. $\Phi_i - \Phi_{i-1} = -1$. Amortized costs $a_i = 0$.
- `multipop`$(k, S)$: real costs $t_i = k$. $\Phi_i - \Phi_{i-1} = -k$. amortized costs $a_i = 0$.

All operations have *constant amortized cost*! Therefore, on average Multipop requires a constant amount of time. [16]

[16] Note that we are not talking about the probabilistic mean but the (worst-case) average of the costs.

## Example Binary Counter

Binary counter with $k$ bits. In the worst case for each count operation maximally $k$ bitflips. Thus $\mathcal{O}(n \cdot k)$ bitflips for counting from 1 to $n$. Better estimation?

Real costs $t_i$ = number bit flips from 0 to 1 plus number of bit-flips from $1$ to $0$.

$$...0\underbrace{1111111}_{l \text{ Einsen}}+1 = ...1\underbrace{0000000}_{l \text{ Zeroes}}.$$

$$\Rightarrow t_i = l + 1$$

## Binary Counter: Aggregate Analysis

Count the number of bit flips when counting from $0$ to $n-1$.

Observation

- Bit $0$ flips for each $k-1 \to k$
- Bit $1$ flips for each $2k-1 \to 2k$
- Bit $2$ flips for each $4k-1 \to 4k$

Total number bit flips $\sum_{i=0}^{n-1} \frac{n}{2^i} \leq n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$

Amortized cost for each increase: $\mathcal{O}(1)$ bit flips.

## Binary Counter: Account Method

Observation: for each increment exactly one bit is incremented to $1$, while many bits may be reset to $0$. Only a bit that had previously been set to $1$ can be reset to $0$.

$a_i = 2$: $1$ CHF real cost for setting $0 \rightarrow 1$ plus $1$ CHF to deposit on the account. Every reset $1 \rightarrow 0$ can be paid from the account.

## Binary Counter: Potential Method

$$\ldots 0\underbrace{1111111}_{l \text{ ones}} + 1 = \ldots 1\underbrace{0000000}_{l \text{ zeros}}$$

potential function $\Phi_i$: number of $1$-bits of $x_i$.

$$\Rightarrow \Phi_0 = 0 \leq \Phi_i \; \forall i$$

$$\Rightarrow \Phi_i - \Phi_{i-1} = 1 - l,$$

$$\Rightarrow a_i = t_i + \Phi_i - \Phi_{i-1} = l + 1 + (1 - l) = 2.$$

Amortized constant cost for each count operation. ☺

# 13. Dictionaries

Dictionary, Self-ordering List, Implementation of Dictionaries with Array / List /Skip lists. [Ottman/Widmayer, Kap. 3.3,1.7, Cormen et al, Kap. Problem 17-5]

## Dictionary

ADT to manage keys from a set $\mathcal{K}$ with operations

- `insert`$(k, D)$: Insert $k \in \mathcal{K}$ to the dictionary $D$. Already exists $\Rightarrow$ error messsage.
- `delete`$(k, D)$: Delete $k$ from the dictionary $D$. Not existing $\Rightarrow$ error message.
- `search`$(k, D)$: Returns `true` if $k \in D$, otherwise `false`

## Idea

Implement dictionary as sorted array

Worst case number of fundamental operations

|  |  |  |
|---|---|---|
| Search | $\mathcal{O}(\log n)$ | 🙂 |
| Insert | $\mathcal{O}(n)$ | 🙁 |
| Delete | $\mathcal{O}(n)$ | 🙁 |

## Other idea

Implement dictionary as a linked list

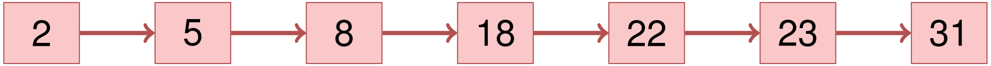
Worst case number of fundamental operations

| | | |
|---|---|---|
| Search | $\mathcal{O}(n)$ | ☹ |
| Insert | $\mathcal{O}(1)$[17] | ☺ |
| Delete | $\mathcal{O}(n)$ | ☹ |

[17] Provided that we do not have to check existence.

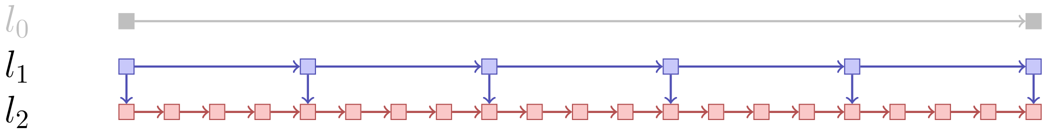
## 13.1 Skip Lists

## Sorted Linked List



Search for element / insertion position: *worst-case* $n$ Steps.

## Sorted Linked List with two Levels



- Number elements: $n_0 := n$
- Stepsize on level $1$: $n_1$
- Stepsize on level $2$: $n_2 = 1$

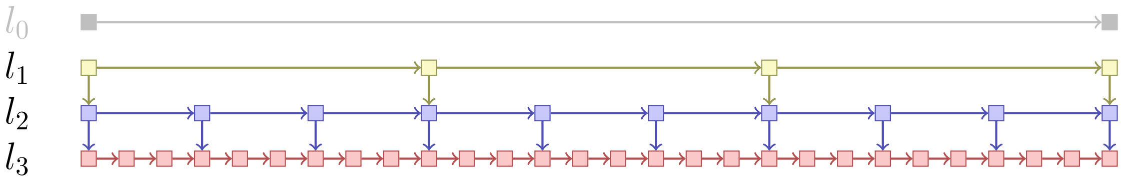
$\Rightarrow$ Search for element / insertion position: worst-case $\frac{n_0}{n_1} + \frac{n_1}{n_2}$.

$\Rightarrow$ Best Choice for[18] $n_1$: $n_1 = \frac{n_0}{n_1} = \sqrt{n_0}$.

Search for element / insertion position: *worst-case* $2\sqrt{n}$ steps.

[18] Differentiate and set to zero, cf. appendix

# Sorted Linked List with two Levels



- Number elements: $n_0 := n$
- Stepsizes on levels $0 < i < 3$: $n_i$
- Stepsize on level $3$: $n_3 = 1$

$\Rightarrow$ Best Choice for $(n_1, n_2)$: $n_2 = \frac{n_0}{n_1} = \frac{n_1}{n_2} = \sqrt[3]{n_0}$.

Search for element / insertion position: *worst-case* $3 \cdot \sqrt[3]{n}$ steps.

## Sorted Linked List with $k$ Levels (Skiplist)

- Number elements: $n_0 := n$
- Stepsizes on levels $0 < i < k$: $n_i$
- Stepsize on level $k$: $n_k = 1$

$\Rightarrow$ Best Choice for $(n_1, \dots, n_k)$: $n_{k-1} = \frac{n_0}{n_1} = \frac{n_1}{n_2} = \dots = \sqrt[k]{n_0}$.

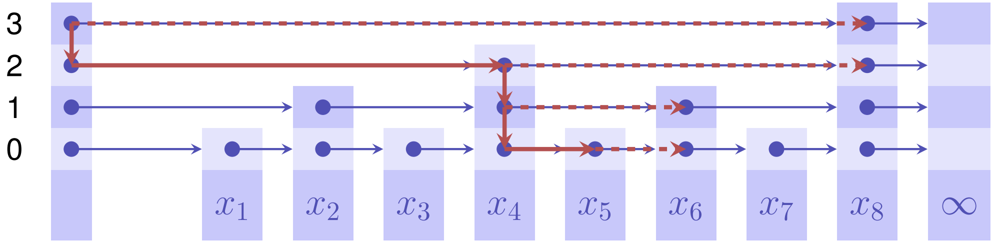
Search for element / insertion position: *worst-case* $k \cdot \sqrt[k]{n}$ steps[19](Derivation: Appendix).

Assumption $n = 2^k$
$\Rightarrow$ worst case $\log_2 n \cdot 2$ steps and $\frac{n_i}{n_{i+1}} = 2 \, \forall \, 0 \leq i < \log_2 n$.

[19](Herleitung: Anhang)

## Search in a Skiplist

Perfect skip list



$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

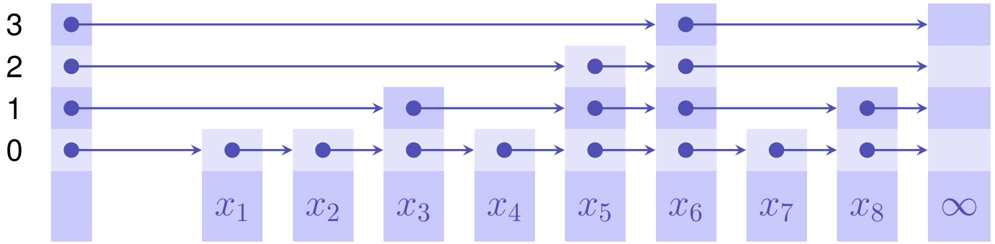
Example: search for a key $x$ with $x_5 < x < x_6$.

## Analysis perfect skip list (worst cases)

Search in $\mathcal{O}(\log n)$. Insert in $\mathcal{O}(n)$.

## Randomized Skip List

Idea: insert a key with random height $H$ with $\mathbb{P}(H = i) = \frac{1}{2^{i+1}}$.



## Analysis Randomized Skip List

**Theorem**

*The expected number of fundamental operations for Search, Insert and Delete of an element in a randomized skip list is $\mathcal{O}(\log n)$.*

The lengthy proof that will not be presented in this courseobserves the length of a path from a searched node back to the starting point in the highest level.

## 13.2 [Self Ordering]

not covered in class

## Self Ordered Lists

Problematic with the adoption of a linked list: linear search time

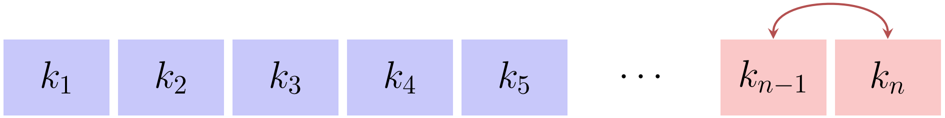
*Idea:* Try to order the list elements such that accesses over time are possible in a faster way

For example

- Transpose: For each access to a key, the key is moved one position closer to the front.
- Move-to-Front (MTF): For each access to a key, the key is moved to the front of the list.

## Transpose

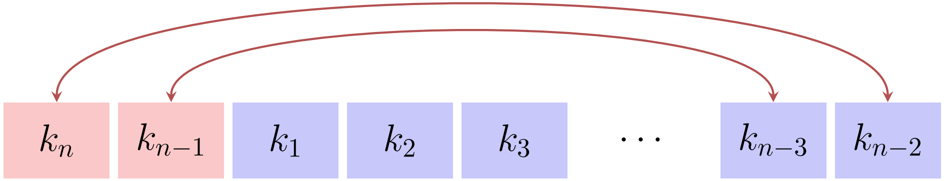
Transpose:



Worst case: Alternating sequence of $n$ accesses to $k_{n-1}$ and $k_n$.
Runtime: $\Theta(n^2)$

## Move-to-Front

Move-to-Front:



Alternating sequence of $n$ accesses to $k_{n-1}$ and $k_n$. Runtime: $\Theta(n)$

Also here we can provide a sequence of accesses with quadratic runtime, e.g. access to the last element. But there is no obvious strategy to counteract much better than MTF..

## Analysis

Compare MTF with the best-possible competitor (algorithm) A. How much better can A be?

Assumptions:

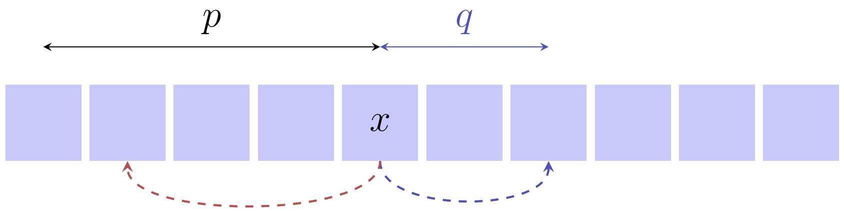
- MTF and A may only move the accessed element.
- MTF and A start with the same list.

Let $M_k$ and $A_k$ designate the lists after the $k$th step. $M_0 = A_0$.

## Analysis

Costs:

- Access to $x$: position $p$ of $x$ in the list.
- No further costs, if $x$ is moved before $p$
- Further costs $q$ for each element that $x$ is moved back starting from $p$.



## Amortized Analysis

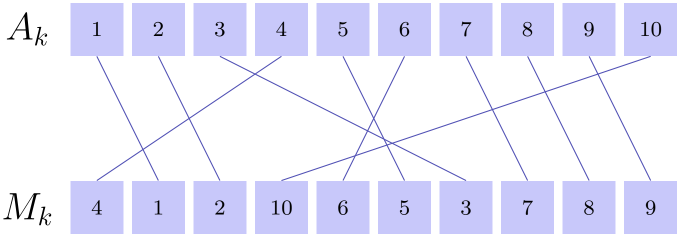
Let an arbitrary sequence of search requests be given and let $G_k^{(M)}$ and $G_k^{(A)}$ the costs in step $k$ for Move-to-Front and A, respectively. Want estimation of $\sum_k G_k^{(M)}$ compared with $\sum_k G_k^{(A)}$.

$\Rightarrow$ Amortized analysis with potential function $\Phi$.

## Potential Function

Potential function $\Phi =$ Number of inversions of A vs. MTF.

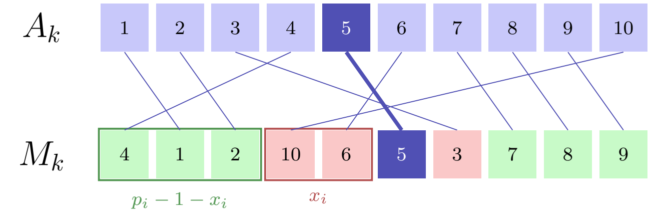
Inversion = Pair $x$, $y$ such that for the positions of $a$ and $y$
$\left(p^{(A)}(x) < p^{(A)}(y)\right) \neq \left(p^{(M)}(x) < p^{(M)}(y)\right)$

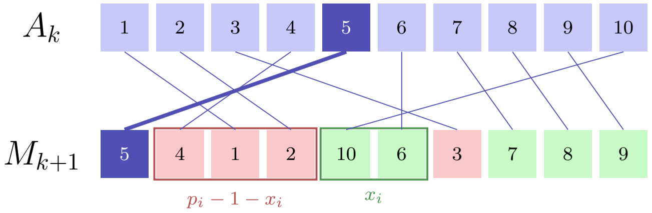


#inversion = #crossings

# Estimating the Potential Function: MTF

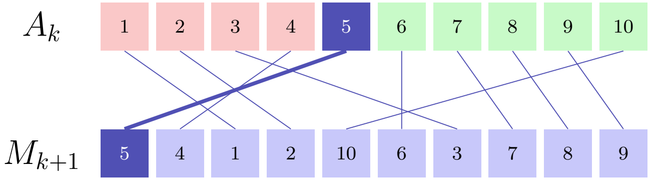
- Element $i$ at position $p_i := p^{(M)}(i)$.
- access costs $C_k^{(M)} = p_i$.
- $x_i$: Number elements that are in M before $p_i$ and in A after $i$ .
- MTF removes $x_i$ inversions.
- $p_i - x_i - 1$: Number elements that in M are before $p_i$ and in A are before $i$.
- MTF generates $p_i - 1 - x_i$ inversions.

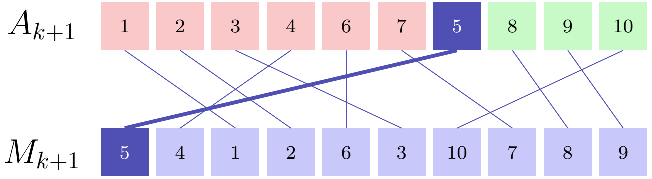




## Estimating the Potential Function: A

- Wlog element $i$ at position $p^{(A)}(i)$.
- $X_k^{(A)}$: number movements to the back (otherwise 0).
- access costs for $i$: $C_k^{(A)} = p^{(A)}(i) \geq p^{(M)}(i) - x_i$.
- A increases the number of inversions maximally by $X_k^{(A)}$.





## Estimation

$$\Phi_{k+1} - \Phi_k \le -x_i + (p_i - 1 - x_i) + X_k^{(A)}$$

Amortized costs of MTF in step $k$:

$$\begin{aligned} a_k^{(M)} &= C_k^{(M)} + \Phi_{k+1} - \Phi_k \\ &\le p_i - x_i + (p_i - 1 - x_i) + X_k^{(A)} \\ &= (p_i - x_i) + (p_i - x_i) - 1 + X_k^{(A)} \\ &\le C_k^{(A)} + C_k^{(A)} - 1 + X_k^{(A)} \le 2 \cdot C_k^{(A)} + X_k^{(A)}. \end{aligned}$$

## Estimation

Summing up costs

$$\begin{aligned}\sum_k G_k^{(M)} = \sum_k C_k^{(M)} &\leq \sum_k a_k^{(M)} \leq \sum_k 2 \cdot C_k^{(A)} + X_k^{(A)} \\ &\leq 2 \cdot \sum_k C_k^{(A)} + X_k^{(A)} \\ &= 2 \cdot \sum_k G_k^{(A)}\end{aligned}$$

In the worst case MTF requires at most twice as many operations as the optimal strategy.

## 13.3 Appendix

Mathematik zur Skipliste

## [$k$-Level Skiplist Math]

Let the number of data points $n_0$ and number levels $k > 0$ be given and let $n_l$ be the numbers of elements skipped per level $l$, $n_k = 1$. Maximum number of total steps in the skip list:

$$f(\vec{n}) = \frac{n_0}{n_1} + \frac{n_1}{n_2} + \ldots \frac{n_{k-1}}{n_k}$$

Minimize $f$ for $(n_1, \ldots, n_{k-1})$: $\frac{\partial f(\vec{n})}{\partial n_t} = 0$ for all $0 < t < k$,
$\frac{\partial f(\vec{n})}{\partial n_t} = -\frac{n_{t-1}}{{n_t}^2} + \frac{1}{n_{t+1}} = 0 \Rightarrow n_{t+1} = \frac{n_t^2}{n_{t-1}}$ and $\frac{n_{t+1}}{n_t} = \frac{n_t}{n_{t-1}}$.

## [$k$-Level Skiplist Math]

Previous slide $\Rightarrow \frac{n_t}{n_0} = \frac{n_t}{n_{t-1}} \frac{n_{t-1}}{n_{t-2}} \ldots \frac{n_1}{n_0} = \left(\frac{n_1}{n_0}\right)^t$

Particularly $1 = n_k = \frac{n_1^k}{n_0^{k-1}} \Rightarrow n_1 = \sqrt[k]{n_0^{k-1}}$

Thus $n_{k-1} = \frac{n_0}{n_1} = \sqrt[k]{\frac{n_0^k}{n_0^{k-1}}} = \sqrt[k]{n_0}$.

Maximum number of total steps in the skip list: $f(\vec{n}) = k \cdot (\sqrt[k]{n_0})$

Assume $n_0 = 2^k$, then $\frac{n_l}{n_{l+1}} = 2$ for all $0 \leq l < k$ (skiplist halves data in each step) and $f(n) = k \cdot 2 = 2 \log_2 n \in \Theta(\log n)$.

# 14. Hashing

Hash Tables, Pre-Hashing, Hashing, Resolving Collisions using Chaining, Simple Uniform Hashing, Popular Hash Functions, Table-Doubling, Open Addressing: Probing, Uniform Hashing, Universal Hashing, Perfect Hashing [Ottman/Widmayer, Kap. 4.1-4.3.2, 4.3.4, Cormen et al, Kap. 11-11.4]

## Motivating Example

*Gloal:* Efficient management of a table of all $n$ ETH-students of

*Possible Requirement:* fast access (insertion, removal, find) of a dataset by name

## Dictionary

Abstract Data Type (ADT) $D$ to manage items[20] $i$ with keys $k \in \mathcal{K}$ with operations

- $D.\texttt{insert}(i)$: Insert or replace $i$ in the dictionary $D$.
- $D.\texttt{delete}(i)$: Delete $i$ from the dictionary $D$. Not existing $\Rightarrow$ error message.
- $D.\texttt{search}(k)$: Returns item with key $k$ if it exists.

[20] Key-value pairs $(k, v)$, in the following we consider mainly the keys

## Dictionary in C++

*Associative Container* `std::unordered_map<>`

```
// Create an unordered_map of strings that map to strings
std::unordered_map<std::string, std::string> u = {
       {"RED","#FF0000"}, {"GREEN","#00FF00"}
};

u["BLUE"] = "#0000FF"; // Add

std::cout << "The HEX of color RED is: " << u["RED"] << "\n";

for( const auto& n : u ) // iterate over key—value pairs
  std::cout << n.first << ":" << n.second << "\n";
```

## Motivation / Use

Perhaps *the* most popular data structure.

- Supported in many programming languages (C++, Java, Python, Ruby, Javascript, C# ...)
- Obvious use
    - Databases, Spreadsheets
    - Symbol tables in compilers and interpreters
- Less obvious
    - Substrin Search (Google, grep)
    - String commonalities (Document distance, DNA)
    - File Synchronisation
    - Cryptography: File-transfer and identification

## 1. Idea: Direct Access Table (Array)

| Index | Item |
|---|---|
| 0 | - |
| 1 | - |
| 2 | - |
| 3 | [3,value(3)] |
| 4 | - |
| 5 | - |
| $\vdots$ | $\vdots$ |
| k | [k,value(k)] |
| $\vdots$ | $\vdots$ |

*Problems*

1. Keys must be non-negative integers
2. Large key-range $\Rightarrow$ large array

## Solution to the first problem: Pre-hashing

Prehashing: Map keys to positive integers using a function $ph : \mathcal{K} \to \mathbb{N}$

- Theoretically always possible because each key is stored as a bit-sequence in the computer
- Theoretically also: $x = y \Leftrightarrow ph(x) = ph(y)$
- Practically: APIs offer functions for pre-hashing. (Java: **`object.hashCode()`**, C++: **`std::hash<>`**, Python: **`hash(object)`**)
- APIs map the key from the key set to an integer with a restricted size.[21]

[21] Therefore the implication $ph(x) = ph(y) \Rightarrow x = y$ does **not** hold any more for all $x$,$y$.

## Prehashing Example : String

Mapping Name $s = s_1 s_2 \dots s_{l_s}$ to key

$$ph(s) = \left( \sum_{i=1}^{l_s} s_{l_s - i + 1} \cdot b^i \right) \bmod 2^w$$

$b$ so that different names map to different keys as far as possible.

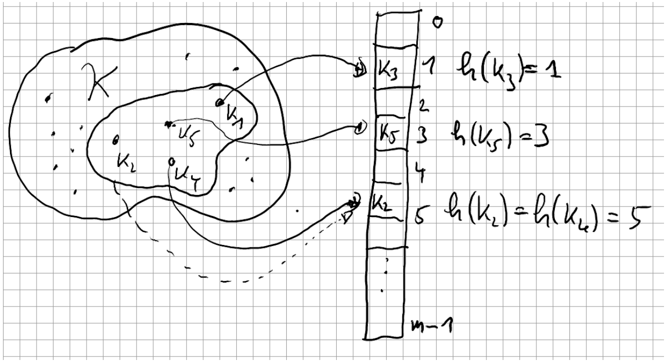
$b$ Word-size of the system (e.g. 32 or 64)

**Example (Java) with $b = 31$, $w = 32$. Ascii-Values $s_i$.**

Anna $\mapsto 2045632$
Jacqueline $\mapsto 42042089953442505 \bmod 2^{32} = 507919049$

## Lösung zum zweiten Problem: Hashing

Reduce the universe. Map (hash-function) $h : \mathcal{K} \to \{0, ..., m-1\}$ ($m \approx n =$ number entries of the table)



Collision: $h(k_i) = h(k_j)$.

## Nomenclature

*Hash funtion* $h$: Mapping from the set of keys $\mathcal{K}$ to the index set $\{0, 1, \ldots, m-1\}$ of an array (*hash table*).

$$h : \mathcal{K} \to \{0, 1, \ldots, m-1\}.$$

Normally $|\mathcal{K}| \gg m$. There are $k_1, k_2 \in \mathcal{K}$ with $h(k_1) = h(k_2)$ (*collision*).

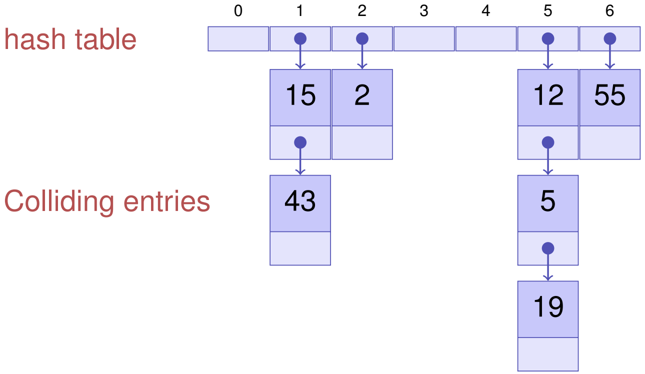
A hash function should map the set of keys as uniformly as possible to the hash table.

## Resolving Collisions: Chaining

Example $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.

Keys 12 , 55 , 5 , 15 , 2 , 19 , 43

Direct Chaining of the Colliding entries



## Algorithm for Hashing with Chaining

- `insert`$(i)$ Check if key $k$ of item $i$ is in list at position $h(k)$. If no, then append $i$ to the end of the list. Otherwise replace element by $i$.
- `find`$(k)$ Check if key $k$ is in list at position $h(k)$. If yes, return the data associated to key $k$, otherwise return empty element `null`.
- `delete`$(k)$ Search the list at position $h(k)$ for $k$. If successful, remove the list element.

## Worst-case Analysis

Worst-case: all keys are mapped to the same index.

$\Rightarrow \Theta(n)$ per operation in the worst case. ☹

## Simple Uniform Hashing

*Strong Assumptions:* Each key will be mapped to one of the $m$ available slots

- with equal probability (Uniformity)
- and independent of where other keys are hashed (Independence).

## Simple Uniform Hashing

Under the assumption of simple uniform hashing:
*Expected length* of a chain when $n$ elements are inserted into a hash table with $m$ elements

$$\mathbb{E}(\text{Länge Kette j}) = \mathbb{E}\left(\sum_{i=0}^{n-1} \mathbb{1}(k_i = j)\right) = \sum_{i=0}^{n-1} \mathbb{P}(k_i = j)$$
$$= \sum_{i=1}^{n} \frac{1}{m} = \frac{n}{m}$$

$\alpha = n/m$ is called *load factor* of the hash table.

## Simple Uniform Hashing

**Theorem**

*Let a hash table with chaining be filled with load-factor $\alpha = \frac{n}{m} < 1$. Under the assumption of simple uniform hashing, the next operation has expected costs of $\leq 1 + \alpha$.*

Consequence: if the number slots $m$ of the hash table is always at least proportional to the number of elements $n$ of the hash table, $n \in \mathcal{O}(m) \Rightarrow$ Expected Running time of Insertion, Search and Deletion is $\mathcal{O}(1)$.

## Further Analysis (directly chained list)

1. Unsuccesful search. The average list lenght is $\alpha = \frac{n}{m}$. The list has to be traversed completely.
   $\Rightarrow$ Average number of entries considered
   $$C'_n = \alpha.$$
2. Successful search Consider the insertion history: key $j$ sees an average list length of $(j-1)/m$.
   $\Rightarrow$ Average number of considered entries
   $$C_n = \frac{1}{n}\sum_{j=1}^{n}(1+(j-1)/m)) = 1 + \frac{1}{n}\frac{n(n-1)}{2m} \approx 1 + \frac{\alpha}{2}.$$

## Advantages and Disadvantages of Chaining

Advantages

- Possible to overcommit: $\alpha > 1$ allowed
- Easy to remove keys.

Disadvantages

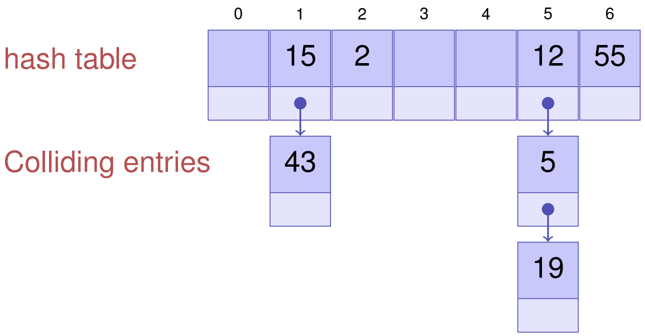
- Memory consumption of the chains-

## [Variant:Indirect Chaining]

Example $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.

Keys 12 , 55 , 5 , 15 , 2 , 19 , 43

Indirect chaining the Collisions



## Examples of popular Hash Functions

$$h(k) = k \bmod m$$

Ideal: $m$ prime, not too close to powers of $2$ or $10$

But often: $m = 2^k - 1$ ($k \in \mathbb{N}$)

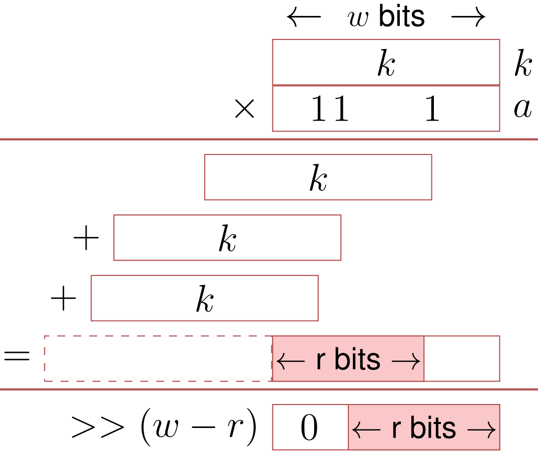
## Examples of popular Hash Functions

*Multiplication method*

$$h(k) = \left\lfloor (a \cdot k \bmod 2^w)/2^{w-r} \right\rfloor \bmod m$$

- $m = 2^r$, $w =$ size of the machine word in bits.
- Multiplication adds $k$ along all bits of $a$, integer division with $2^{w-r}$ and $\operatorname{mod} m$ extract the upper $r$ bits.
- Written as code `a * k >> (w−r)`
- A good value of $a$: $\left\lfloor \frac{\sqrt{5}-1}{2} \cdot 2^w \right\rfloor$: Integer that represents the first $w$ bits of the fractional part of the irrational number.

## Illustration



$\leftarrow$ $w$ bits $\rightarrow$
$k$
$k$
$\times$
11 1
$a$
$k$
$+$ $k$
$+$ $k$
$=$
$\leftarrow$ r bits $\rightarrow$
$>> (w-r)$
0
$\leftarrow$ r bits $\rightarrow$

## Table size increase

- We do not know beforehand how large $n$ will be
- Require $m = \Theta(n)$ at all times.

Table size needs to be adapted. Hash-Function changes $\Rightarrow$ *rehashing*

- Allocate array $A'$ with size $m' > m$
- Insert each entry of $A$ into $A'$ (with re-hashing the keys)
- Set $A \leftarrow A'$.
- Costs $\mathcal{O}(n + m + m')$.

How to choose $m'$?

# Table size increase

- 1.Idea $n = m \Rightarrow m' \leftarrow m + 1$
  Increase for each insertion: Costs $\Theta(1 + 2 + 3 + \cdots + n) = \Theta(n^2)$
  ☹
- 2.Idea $n = m \Rightarrow m' \leftarrow 2m$ Increase only if$m = 2^i$:
  $\Theta(1 + 2 + 4 + 8 + \cdots + n) = \Theta(n)$
  Few insertions cost linear time but on average we have $\Theta(1)$ ☺

Jede Operation vom Hashing mit Verketten hat erwartet amortisierte Kosten $\Theta(1)$.

($\Rightarrow$ Amortized Analysis)

## Open Addressing[22]

Store the colliding entries directly in the hash table using a *probing function* $s : \mathcal{K} \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$

Key table position along a *probing sequence*

$$S(k) := (s(k, 0), s(k, 1), \ldots, s(k, m-1)) \mod m$$

Probing sequence must for each $k \in \mathcal{K}$ be a permutation of $\{0, 1, \ldots, m-1\}$

[22]Notational clarification: this method uses *open addressing*(meaning that the positions in the hashtable are not fixed) but it is a *closed hashing* procedure (because the entries stay in the hashtable)

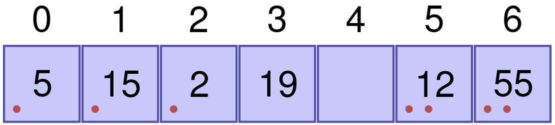
## Algorithms for open addressing

- `insert`$(i)$ Search for kes $k$ of $i$ in the table according to $S(k)$. If $k$ is not present, insert $k$ at the first free position in the probing sequence. Otherwise error message.
- `find`$(k)$ Traverse table entries according to $S(k)$. If $k$ is found, return data associated to $k$. Otherwise return an empty element `null`.
- `delete`$(k)$ Search $k$ in the table according to $S(k)$. If $k$ is found, replace it with a special key `removed`.

## Linear Probing

$s(k, j) = h(k) + j \Rightarrow S(k) = (h(k), h(k) + 1, \ldots, h(k) + m - 1) \bmod m$

Example $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.
Key 12 , 55 , 5 , 15 , 2 , 19



## [Analysis linear probing (without proof)]

1. Unsuccessful search. Average number of considered entries

$$C'_n \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

2. Successful search. Average number of considered entries

$$C_n \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right).$$

## Discussion

**Example $\alpha = 0.95$**

The unsuccessful search consideres 200 table entries on average! (here without derivation).

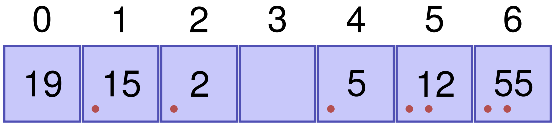
**? Disadvantage of the method?**

! *Primary clustering:* similar hash addresses have similar probing sequences $\Rightarrow$ long contiguous areas of used entries.

## Quadratic Probing

$$s(k,j) = h(k) + \lceil j/2 \rceil^2 (-1)^{j+1}$$
$$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \mod m$$

Example $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.
Keys 12 , 55 , 5 , 15 , 2 , 19



## [Analysis Quadratic Probing (without Proof)]

1. Unsuccessful search. Average number of entries considered

$$C'_n \approx \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right)$$

2. Successful search. Average number of entries considered

$$C_n \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2}.$$

## Discussion

**Example $\alpha = 0.95$**

Unsuccessfuly search considers 22 entries on average (here without derivation)

**? Problems of this method?**

! *Secondary clustering:* Synonyms $k$ and $k'$ (with $h(k) = h(k')$) travers the same probing sequence.

## Double Hashing

Two hash functions $h(k)$ and $h'(k)$. $s(k,j) = h(k) + j \cdot h'(k)$.

$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m-1)h'(k)) \mod m$

Example:

$m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod 7$, $h'(k) = 1 + k \bmod 5$.

Keys 12 , 55 , 5 , 15 , 2 , 19

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 5 | 15 | 2 | 19 |  | 12 | 55 |

## Double Hashing

- Probing sequence must permute all hash addresses. Thus $h'(k) \neq 0$ and $h'(k)$ may not divide $m$, for example guaranteed with $m$ prime.
- $h'$ should be as independent of $h$ as possible (to avoid secondary clustering)

Independence:

$$\mathbb{P}\left((h(k) = h(k')) \ \wedge \ (h'(k) = h'(k'))\right) = \mathbb{P}\left(h(k) = h(k')\right) \cdot \mathbb{P}\left(h'(k) = h'(k')\right).$$

Independence largely fulfilled by $h(k) = k \bmod m$ and $h'(k) = 1 + k \bmod (m-2)$ ($m$ prime).

## [Analysis Double Hashing]

Let $h$ and $h'$ be independent, then:

1. Unsuccessful search. Average number of considered entries:

$$C'_n \approx \frac{1}{1-\alpha}$$

2. Successful search. Average number of considered entries:

$$C_n \approx \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$$

## Uniform Hashing

Strong assumption: the probing sequence $S(k)$ of a key $l$ is equaly likely to be any of the $m!$ permutations of $\{0, 1, \ldots, m-1\}$

(Double hashing is reasonably close)

## Analysis of Uniform Hashing with Open Addressing

**Theorem**

*Let an open-addressing hash table be filled with load-factor $\alpha = \frac{n}{m} < 1$. Under the assumption of uniform hashing, the next operation has expected costs of $\leq \frac{1}{1-\alpha}$.*

## Analysis of Uniform Hashing with Open Addressing

Proof of the Theorem: Random Variable $X$: Number of probings when searching without success.

$$\begin{aligned}\mathbb{P}(X \geq i) &\overset{*}{=} \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2}\\ &\overset{**}{\leq} \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}. \qquad (1 \leq i \leq m)\end{aligned}$$

*: $A_j$:Slot used during step $j$.
$\mathbb{P}(A_1 \cap \cdots \cap A_{i-1}) = \mathbb{P}(A_1) \cdot \mathbb{P}(A_2|A_1) \cdot \ldots \cdot \mathbb{P}(A_{i-1}|A_1 \cap \cdots \cap A_{i-2})$,
**: $\frac{n-1}{m-1} < \frac{n}{m}$ because[23] $n < m$.

Moreover $\mathbb{P}(x \geq i) = 0$ for $i \geq m$. Therefore

$$\mathbb{E}(X) \overset{\text{Appendix}}{=} \sum_{i=1}^{\infty} \mathbb{P}(X \geq i) \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha}.$$

23 $\frac{n-1}{m-1} < \frac{n}{m} \Leftrightarrow \frac{n-1}{n} < \frac{m-1}{m} \Leftrightarrow 1 - \frac{1}{n} < 1 - \frac{1}{m} \Leftrightarrow n < m \; (n > 0, m > 0)$

## [Successful search of Uniform Open Hashing]

**Theorem**

*Let an open-addressing hash table be filled with load-factor $\alpha = \frac{n}{m} < 1$. Under the assumption of uniform hashing, the successful search has expected costs of* $\leq \frac{1}{\alpha} \cdot \log \frac{1}{1-\alpha}$.

Proof: Cormen et al, Kap. 11.4

## Overview

| | $\alpha = 0.50$ | | $\alpha = 0.90$ | | $\alpha = 0.95$ | |
|---|---|---|---|---|---|---|
| | $C_n$ | $C'_n$ | $C_n$ | $C'_n$ | $C_n$ | $C'_n$ |
| (Direct) Chaining | $1.25$ | $0.50$ | $1.45$ | $0.90$ | $1.48$ | $0.95$ |
| Linear Probing | $1.50$ | $2.50$ | $5.50$ | $50.50$ | $10.50$ | $200.50$ |
| Quadratic Probing | $1.44$ | $2.19$ | $2.85$ | $11.40$ | $3.52$ | $22.05$ |
| Uniform Hashing | $1.39$ | $2.00$ | $2.56$ | $10.00$ | $3.15$ | $20.00$ |

: $C_n$: Anzahl Schritte erfolgreiche Suche, $C'_n$: Anzahl Schritte erfolglose Suche, Belegungsgrad $\alpha$.

## Universal Hashing

- $|\mathcal{K}| > m \Rightarrow$ Set of “similar keys” can be chosen such that a large number of collisions occur.
- Impossible to select a “best” hash function for all cases.
- Possible, however[24]: randomize!

*Universal hash class* $\mathcal{H} \subseteq \{h : \mathcal{K} \to \{0, 1, \ldots, m-1\}\}$ is a family of hash functions such that

$$\forall\, k_1 \neq k_2 \in \mathcal{K} \text{ it holds that } |\{h \in \mathcal{H} \text{ with } h(k_1) = h(k_2)\}| \leq \frac{|\mathcal{H}|}{m}.$$

[24] Similar as for quicksort

## Universal Hashing

**Theorem**

*A function $h$ randomly chosen from a universal class $\mathcal{H}$ of hash functions randomly distributes an arbitrary sequence of keys from $\mathcal{K}$ as uniformly as possible on the available slots.*

*When using hashing with chaining, the expected chain length for an element that is not contained in the table is $\leq \alpha = n/m$. The expected chain length for an element contained is $\leq 1 + \alpha$.*

## Universal Hashing

Initial remark for the proof of the theorem:

Define with $x, y \in \mathcal{K}$, $h \in \mathcal{H}$, $Y \subseteq \mathcal{K}$:

$$\delta(h, x, y) = \begin{cases} 1, & \text{if } h(x) = h(y) \\ 0, & \text{otherwise,} \end{cases}$$

is $h(x) = h(y)$ (0 or 1)?

$$\delta(h, x, Y) = \sum_{y \in Y} \delta(x, y, h),$$

for how many $y \in Y$ is $h(x) = h(y)$?

$$\delta(\mathcal{H}, x, y) = \sum_{h \in \mathcal{H}} \delta(x, y, h)$$

for how many $h \in \mathcal{H}$ is $h(x) = h(y)$?.

$\mathcal{H}$ is universal if for all $x, y \in \mathcal{K}$, $x \neq y$ : $\delta(\mathcal{H}, x, y) \leq |\mathcal{H}|/m$.

## Universal Hashing

Proof of the theorem

$S \subseteq \mathcal{K}$: keys stored up to now. $x$ is added now: ($x \notin S$)

Expected number of collisions of $x$ with $S$

$$\begin{aligned}
\mathbb{E}_{\mathcal{H}}(\delta(h,x,S)) &= \sum_{h\in\mathcal{H}} \delta(h,x,S)/|\mathcal{H}| \\
&= \frac{1}{|\mathcal{H}|}\sum_{h\in\mathcal{H}}\sum_{y\in S}\delta(h,x,y) = \frac{1}{|\mathcal{H}|}\sum_{y\in S}\sum_{h\in\mathcal{H}}\delta(h,x,y) \\
&= \frac{1}{|\mathcal{H}|}\sum_{y\in S}\delta(\mathcal{H},x,y) \\
&\leq \frac{1}{|\mathcal{H}|}\sum_{y\in S}\frac{|\mathcal{H}|}{m} = \frac{|S|}{m} = \alpha.
\end{aligned}$$

■

## Universal Hashing

$S \subseteq \mathcal{K}$: keys stored up to now, now $x \in S$.

Expected number of collisions of $x$ with $S$

$$\begin{aligned}
\mathbb{E}_{\mathcal{H}}(\delta(x,S,h)) &= \sum_{h\in\mathcal{H}} \delta(x,S,h)/|\mathcal{H}| \\
&= \frac{1}{|\mathcal{H}|}\sum_{h\in\mathcal{H}}\sum_{y\in S}\delta(h,x,y) = \frac{1}{|\mathcal{H}|}\sum_{y\in S}\sum_{h\in\mathcal{H}}\delta(h,x,y) \\
&= \frac{1}{|\mathcal{H}|}\left(\delta(\mathcal{H},x,x) + \sum_{y\in S-\{x\}}\delta(\mathcal{H},x,y)\right) \\
&\le \frac{1}{|\mathcal{H}|}\left(|\mathcal{H}| + \sum_{y\in S-\{x\}}|\mathcal{H}|/m\right) = 1 + \frac{|S|-1}{m} = 1 + \frac{n-1}{m} \le 1+\alpha.
\end{aligned}$$

■

## Construction Universal Class of Hashfunctions

Let key set be $\mathcal{K} = \{0, \ldots, u-1\}$ and $p \geq u$ be prime. With $a \in \mathcal{K} \setminus \{0\}$, $b \in \mathcal{K}$ define

$$h_{ab} : \mathcal{K} \rightarrow \{0, \ldots, m-1\}, h_{ab}(x) = ((ax+b) \bmod p) \bmod m.$$

Then the following theorem holds:

**Theorem**

*The class $\mathcal{H} = \{h_{ab} | a, b \in \mathcal{K}, a \neq 0\}$ is a universal class of hash functions.*

(Here without proof, see e.g. Cormen et al, Kap. 11.3.3)

## Perfect Hashing

If the set of used keys is known up-front, the hash function can be chosen perfectly, i.e. such that there are no collisions.

Example: table of key words of a compiler.

## Observation (Birthday Paradox Reversed)

- $h$ be chosen at random from universal hashclass $\mathcal{H}$.
- $n$ keys $S \subset \mathcal{K}$
- Random variable $X$ : number collisionsof the $n$ keys from$S$

$\Rightarrow$

$$\mathbb{E}(X) = \mathbb{E}\left(\sum_{i \neq j} \mathbb{1}(h(k_i) = h(k_j)\right) = \sum_{i \neq j} \mathbb{E}\left(\mathbb{1}(h(k_i) = h(k_j))\right)$$

$$\overset{*}{=} \binom{n}{2}\frac{1}{m} \leq \frac{n^2}{2m}$$

* # Unordered Pairs $\sum_{i \neq j} 1 = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} (n-1-i) = n(n-1) - n(n-1)/2 = n(n-1)/2$

## Perfect Hashing with memory space $\Theta(n^2)$

if $m = n^2 \Rightarrow \mathbb{E}(X) \leq \frac{1}{2}$.

Markov-Inequality[25] $\mathbb{P}(X \geq 1) \leq \frac{\mathbb{E}(X)}{1} \leq \frac{1}{2}$

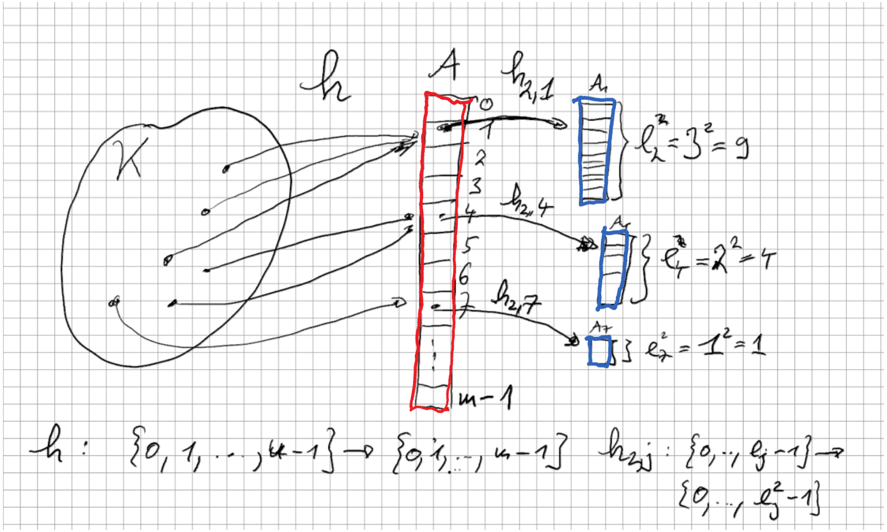
Thus

$$\mathbb{E}(X < 1) = \mathbb{E}(\text{no Collision}) \geq \frac{1}{2}.$$

Consequence: for $n$ keys, in expected $2 \cdot n$ steps, a collision free hash-table of size $m = n^2$ can be constructed by choosing from a universal hash class at random.

[25] Appendix

## Perfect Hashing Idea



h
A
K
$h_{2,1}$
$A_1$
$\ell_1^2 = 3^2 = 9$
$h_{2,4}$
$A_4$
$\ell_4^2 = 2^2 = 4$
$h_{2,7}$
$A_7$
$\ell_7^2 = 1^2 = 1$
0
1
2
3
4
5
6
7
m−1
$h: \{0, 1, \ldots, u-1\} \to \{0, 1, \ldots, m-1\}$
$h_{2,j}: \{0, \ldots, \ell_j - 1\} \to \{0, \ldots, \ell_j^2 - 1\}$

# Perfect Hashing with $\Theta(n)$ memory consumption.

Two-level hashing

1. Choose $m = n$ and $h : \{0, 1, \dots, u-1\} \to \{0, 1, \dots, m-1\}$ from a universal hash-class. Insert all $n$ keys into the hash table using chaining. Let $l_i$ be the length of a chain at index $i$. If $\sum_{i=0}^{m-1} l_i^2 > 4n$, then repeat this step 1.
2. For each index $i = 1, \dots, m-1$ with $l_i > 0$ construct, for the $l_i$ contained keys, hash tables of length $l_i^2$ using universal hashing (hash function $h_{2,i}$) until there are no collisions.

Memory consumption $\Theta(n)$.

## Expected Running times

- For Step 1: hash table of size $m = n$.
  We show on the next page that $\mathbb{E}\left(\sum_{j=0}^{m-1} l_j^2\right) \leq 2n$. Consequently (Markov): $\mathbb{P}\left(\sum_{j=0}^{m-1} l_j^2 \geq 4n\right) \leq \frac{2n}{4n} = \frac{1}{2}$.
  $\Rightarrow$ Expected two retries of step 1.
- For Step 2: $\sum l_i^2 \leq 4n$. For each $i$ expected two trials with running time $l_i^2$. Overal $\mathcal{O}(n)$

$\Rightarrow$ The perfect hash tables can be constructed in expected $\mathcal{O}(n)$ steps.

## Expected Memory Space 2nd Level Hash Tables

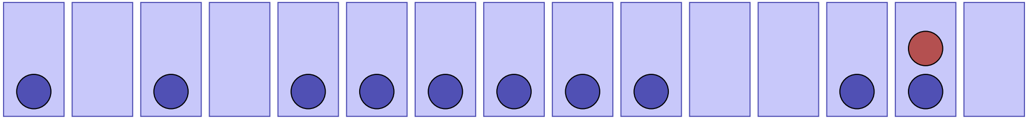
$$\begin{aligned}
\mathbb{E}\left(\sum_{j=0}^{m-1} l_j^2\right) &= \mathbb{E}\left(\sum_{j=0}^{m-1}\sum_{i=0}^{n-1}\sum_{i'=0}^{n-1} \mathbb{1}(h(k_i) = h(k_{i'}) = j)\right) \\
&= \mathbb{E}\left(\sum_{i=0}^{n-1}\sum_{i'=0}^{n-1} \mathbb{1}(h(k_i) = h(k_{i'}))\right) \\
&= \mathbb{E}\left(\sum_{i=i'} \mathbb{1}(h(k_i) = h(k_{i'})) + 2\cdot\sum_{i\neq i'} \mathbb{1}(h(k_i) = h(k_{i'}))\right) \\
&= n + 2\cdot\sum_{i\neq i'} \mathbb{E}\left(\mathbb{1}(h(k_i) = h(k_{i'}))\right) \\
&= n + 2\binom{n}{2}\frac{1}{m} \overset{m=n}{=} 2n - 1 \leq 2n.
\end{aligned}$$

## 14.9 Appendix

Some mathematical formulas

## [Birthday Paradox]

Assumption: $m$ urns, $n$ balls (wlog $n \leq m$).
$n$ balls are put uniformly distributed into the urns



What is the collision probability?

Birthdayparadox: with how many people ($n$) the probability that two of them share the same birthday ($m = 365$) is larger than $50\%$?

## [Birthday Paradox]

$\mathbb{P}(\text{no collision}) = \frac{m}{m} \cdot \frac{m-1}{m} \cdot \cdots \cdot \frac{m-n+1}{m} = \frac{m!}{(m-n)! \cdot m^m}$.

Let $a \ll m$. With $e^x = 1 + x + \frac{x^2}{2!} + \ldots$ approximate $1 - \frac{a}{m} \approx e^{-\frac{a}{m}}$. This yields:

$$1 \cdot \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{2}{m}\right) \cdot \ldots \cdot \left(1 - \frac{n-1}{m}\right) \approx e^{-\frac{1+\cdots+n-1}{m}} = e^{-\frac{n(n-1)}{2m}}.$$

Thus

$$\mathbb{P}(\text{Kollision}) = 1 - e^{-\frac{n(n-1)}{2m}}.$$

Puzzle answer: with 23 people the probability for a birthday collision is $50.7\%$. Derived from the slightly more accurate Stirling formula. $n! \approx \sqrt{2\pi n} \cdot n^n \cdot e^{-n}$

## [Formula for Expected Value]

$X \geq 0$ discrete random variable with $\mathbb{E}(X) < \infty$

$$\mathbb{E}(X) \overset{(def)}{=} \sum_{x=0}^{\infty} x \mathbb{P}(X = x)$$

$$\overset{\text{Counting}}{=} \sum_{x=1}^{\infty} \sum_{y=x}^{\infty} \mathbb{P}(X = y)$$

$$= \sum_{x=0}^{\infty} \mathbb{P}(X \geq x)$$

## [Markov Inequality]

discrete Version

$$\begin{aligned}\mathbb{E}(X) &= \sum_{x=-\infty}^{\infty} x\mathbb{P}(X=x)\\ &\geq \sum_{x=a}^{\infty} x\mathbb{P}(X=x)\\ &\geq a\sum_{x=a}^{\infty} \mathbb{P}(X=x)\\ &= a\cdot\mathbb{P}(X\geq a)\end{aligned}$$

$\Rightarrow$

$$\mathbb{P}(X\geq a)\leq\frac{\mathbb{E}(X)}{a}$$

# 15. C++ advanced (III): Functors and Lambda

## What do we learn today?

- Functors: objects with overloaded function operator `()`.
- Closures
- Lambda-Expressions: syntactic sugar
- Captures
- Function type variables

## Functors: Motivation

A simple output filter

```
template <typename T, typename Function>
void filter(const T& collection, Function f){
    for (const auto& x: collection)
        if (f(x)) std::cout << x << " ";
    std::cout << "\n";
}
```

( `filter` works if the first argument offers an iterator and if the second argument can be applied to elements with a result that can be converted to bool. )

## Functors: Motivation

```
template <typename T, typename Function>
void filter(const T& collection, Function f);

template <typename T>
bool even(T x){
    return x % 2 == 0;
}

std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
filter(a,even<int>); // output: 2,4,6,16
```

## Functor: Object with Overloaded Operator ( )

```
class GreaterThan{
  int value; // state
  public:
  GreaterThan(int x):value{x}{}

  bool operator() (int par) const {
    return par > value;
  }
};
```

A Functor is a callable object. Can be understood as a stateful function.

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int value=8;
filter(a,GreaterThan(value)); // 9,11,16,19
```

## Functor: object with overloaded operator ()

```
template <typename T>
class GreaterThan{
    T value;
public:
    GreaterThan(T x):value{x}{}

    bool operator() (T par) const{
        return par > value;
    }
};
```

(this also works with a template, of course)

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int value=8;
filter(a,GreaterThan<int>(value)); // 9,11,16,19
```

## The same with a Lambda-Expression

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int value=8;

filter(a, [value](int x) {return x > value;} );
```

## Sum of Elements – Old School

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int sum = 0;
for (auto x: a)
  sum += x;
std::cout << sum << std::endl; // 83
```

## Sum of Elements – with Functor

```
template <typename T>
struct Sum{
    T value = 0;

    void operator() (T par){ value += par; }
};

std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
Sum<int> sum;
// for_each copies sum: we need to copy the result back
sum = std::for_each(a.begin(), a.end(), sum);
std::cout << sum.value << std::endl; // 83
```

# Sum of Elements – with References[26]

```
template <typename T>
struct SumR{
    T& value;
    SumR (T& v):value{v} {}

    void operator() (T par){ value += par; }
};

std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int s=0;
SumR<int> sum{s};
// cannot (and do not need to) assign to sum here
std::for_each(a.begin(), a.end(), sum);
std::cout << s << std::endl; // 83
```

[26] Of course this works, very similarly, using pointers

## Sum of Elements – with $\Lambda$

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};

int s=0;

std::for_each(a.begin(), a.end(),  [&s] (int x) {s += x;}  );

std::cout << s << std::endl;
```

## Sorting by Different Order

```
// pre: i >= 0
// post: returns sum of digits of i
int q(int i){
    int res =0;
    for(;i>0;i/=10)
        res += i % 10;
    return res;
}

std::vector<int> v {10,12,9,7,28,22,14};
std::sort (v.begin(), v.end(),
  [] (int i, int j) { return q(i) < q(j);}
);
```

Now $v = 10, 12, 22, 14, 7, 9, 28$ (sorted by sum of digits)

## Lambda-Expressions in Detail

$$\underbrace{\texttt{[value]}}_{\text{capture}}\ \underbrace{\texttt{(int x)}}_{\text{parameters}}\ \underbrace{\texttt{->bool}}_{\substack{\text{return}\\\text{type}}}\ \underbrace{\texttt{\{return x > value;\}}}_{\text{statement}}$$

## Closure

```
[value] (int x) ->bool {return x > value;}
```

- Lambda expressions evaluate to a temporary object – a closure
- The closure retains the execution context of the function - the captured objects.
- Lambda expressions can be implemented as functors.

# Simple Lambda Expression

```
[]()->void {std::cout << "Hello World";}
```

call:

```
[]()->void {std::cout << "Hello World";}();
```

assignment:

```
auto f = []()->void {std::cout << "Hello World";};
```

## Minimal Lambda Expression

```
[]{}
```

- Return type can be inferred if no or only one return statement is present.[27]

```
[]() {std::cout << "Hello World";}
```

- If no parameters and no explicit return type, then () can be omitted.

```
[]{std::cout << "Hello World";}
```

- `[...]` can never be omitted.

[27]Since C++14 also several returns possible, provided that the same return type is deduced

## Examples

```
[](int x, int y) {std::cout << x * y;} (4,5);
```

Output: 20

## Examples

```
int k = 8;
auto f = [](int& v) {v += v;};
f(k);
std::cout << k;
```

Output: 16

## Examples

```
int k = 8;
auto f = [](int v) {v += v;};
f(k);
std::cout << k;
```

Output: 8

## Capture – Lambdas

For Lambda-expressions the capture list determines the context accessible

Syntax:

- `[x]`: Access a copy of x (read-only)
- `[&x]`: Capture x by reference
- `[&x,y]`: Capture x by reference and y by value
- `[&]`: Default capture all objects by reference in the scope of the lambda expression
- `[=]`: Default capture all objects by value in the context of the Lambda-Expression

## Capture – Lambdas

```
int elements=0;
int sum=0;
std::for_each(v.begin(), v.end(),
  [&] (int k) {sum += k; elements++;} // capture all by reference
)
```

## Capture – Lambdas

```
template <typename T>
void sequence(vector<int> & v, T done){
  int i=0;
  while (!done()) v.push_back(i++);
}

vector<int> s;
sequence(s, [&] {return s.size() >= 5;} )
```

now v = 0 1 2 3 4

The capture list refers to the context of the lambda expression.

## Capture – Lambdas

When is the value captured?

```
int v = 42;
auto func = [=] {std::cout << v << "\n"};
v = 7;
func();
```

Output: 42

Values are assigned when the lambda-expression is created.

## Capture – Lambdas

(Why) does this work?

```
class Limited{
  int limit = 10;
 public:
  // count entries smaller than limit
  int count(const std::vector<int>& a){
    int c = 0;
    std::for_each(a.begin(), a.end(),
        [=,&c] (int x) {if (x < limit) c++;}
    );
    return c;
  }
};
```

The `this` pointer is implicitly copied by value

## Capture – Lambdas

```
struct mutant{
  int i = 0;
  void do(){ [=] {i=42;}();}
};

mutant m;
m.do();
std::cout << m.i;
```

Output: 42

The `this` *pointer* is implicitly copied by value

## Lambda Expressions are Functors

```
[x, &y] () {y = x;}
```

can be implemented as

```
unnamed {x,y};
```

with

```
class unnamed {
  int x; int& y;
  unnamed (int x_, int& y_) : x (x_), y (y_) {}
  void operator () () {y = x;}
};
```

## Lambda Expressions are Functors

```
[=] () {return x + y;}
```

can be implemented as

```
unnamed {x,y};
```

with

```
class unnamed {
  int x; int y;
  unnamed (int x_, int y_) : x (x_), y (y_) {}
  int operator () () const {return x + y;}
};
```

## Polymorphic Function Wrapper `std::function`

```
#include <functional>

int k= 8;
std::function<int(int)> f;
f = [k](int i){ return i+k; };
std::cout << f(8); // 16
```

can be used in order to store lambda expressions.

Other Examples

`std::function<int(int,int)>`;

`std::function<void(double)>` ...

http://en.cppreference.com/w/cpp/utility/functional/function

## Example

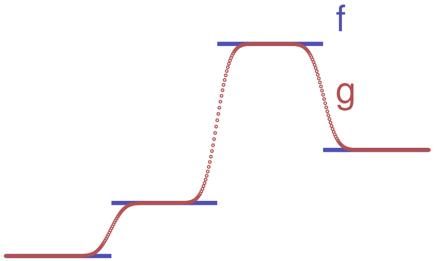
```
template <typename T>
auto toFunction(std::vector<T> v){
  return [v] (T x) -> double {
    int index = (int)(x+0.5);
    if (index < 0) index = 0;
    if (index >= v.size()) index = v.size()-1;
    return v[index];
  };
}
```

## Example

```
auto Gaussian(double mu, double sigma){
    return [mu,sigma](double x) {
        const double a = ( x − mu ) / sigma;
        return std::exp( −0.5 ∗ a ∗ a );
    };
}

template <typename F, typename Kernel>
auto smooth(F f, Kernel kernel){
  return [kernel,f] (auto x) {
        // compute convolution ...
        // and return result
  };
}
```

## Example

```
std::vector<double> v {1,2,5,3};
auto f = toFunction(v);
auto k = Gaussian(0,0.1);
auto g = smooth(f,k);
```



## Conclusion

- Functors allow to write functional programs in C++. Lambdas are syntactic sugar to simplify this.
- With functors/lambdas classic patters from functional programming (e.g. map / filter /reduce) can be applied in C++.
- In combination with templates and the type inference (`auto`) very powerful functions can be stored in variables. Functions can even return functions (so called higher order functions).

# 16. Binary Search Trees

[Ottman/Widmayer, Kap. 5.1, Cormen et al, Kap. 12.1 - 12.3]

## Dictionary implementation

Hashing: implementation of dictionaries with expected very fast access times.

Disadvantages of hashing: linear access time in worst case. Some operations not supported at all:

- enumerate keys in increasing order
- next smallest key to given key
- Key $k$ in given interval $k \in [l, r]$

## Trees

Trees are

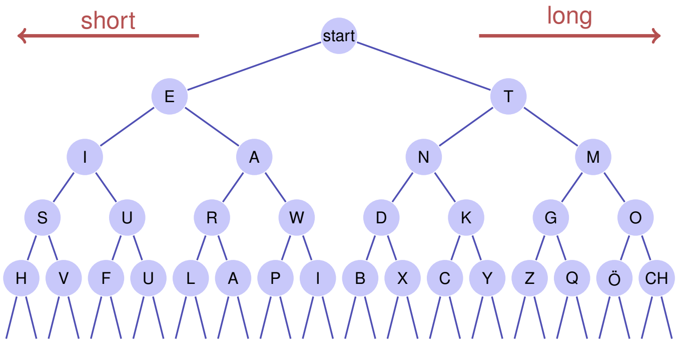
- Generalized lists: nodes can have more than one successor
- Special graphs: graphs consist of nodes and edges. A tree is a fully connected, directed, acyclic graph.

## Trees

Use

- Decision trees: hierarchic representation of decision rules
- syntax trees: parsing and traversing of expressions, e.g. in a compiler
- Code tress: representation of a code, e.g. morse alphabet, huffman code
- Search trees: allow efficient searching for an element by value

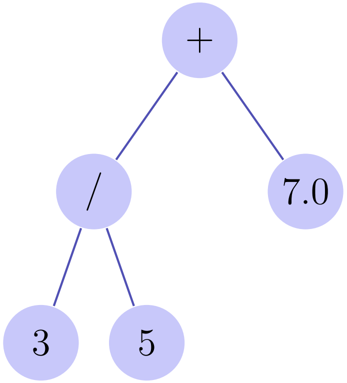
## Examples



Morsealphabet

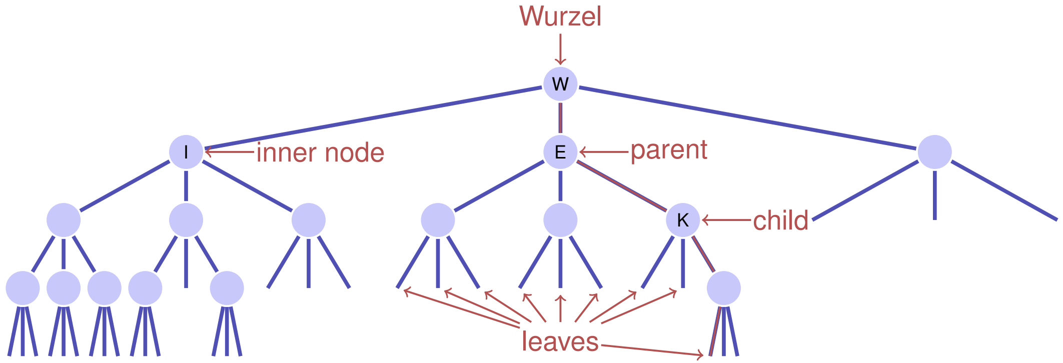
## Examples

$$3/5 + 7.0$$



Expression tree

## Nomenclature



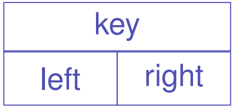
- Order of the tree: maximum number of child nodes, here: 3
- Height of the tree: maximum path length root – leaf (here: 4)

## Binary Trees

A binary tree is

- either a leaf, i.e. an empty tree,
- or an inner leaf with two trees $T_l$ (left subtree) and $T_r$ (right subtree) as left and right successor.

In each inner node `v` we store



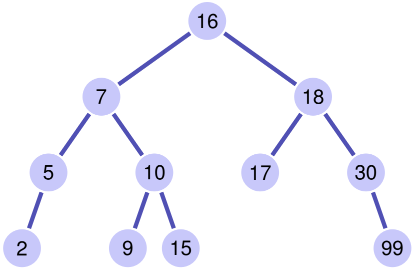
- a key `v.key` and
- two nodes `v.left` and `v.right` to the roots of the left and right subtree.

a leaf is represented by the **null**-pointer

## Binary search tree

A binary search tree is a binary tree that fulfils the *search tree property*:

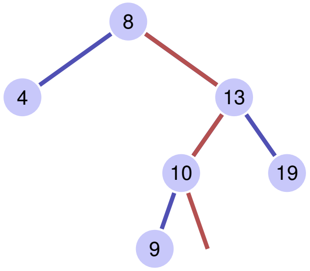
- Every node `v` stores a key
- Keys in left subtree `v.left` are smaller than `v.key`
- Keys in right subtree `v.right` are greater than `v.key`



## Searching

**Input:** Binary search tree with root $r$, key $k$
**Output:** Node $v$ with $v.\mathrm{key} = k$ or **null**

```
$v \leftarrow r$
while $v \neq$ null do
    if $k = v.\mathrm{key}$ then
        return $v$
    else if $k < v.\mathrm{key}$ then
        $v \leftarrow v.\mathrm{left}$
    else
        $v \leftarrow v.\mathrm{right}$
return null
```



Search (12) $\rightarrow$ **null**

# Height of a tree

The height $h(T)$ of a binary tree $T$ with root $r$ is given by

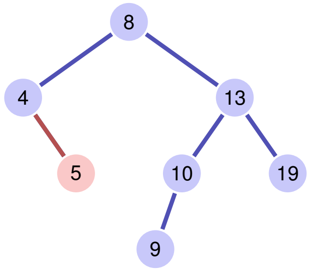
$$h(r) = \begin{cases} 0 & \text{if } r = \textbf{null} \\ 1 + \max\{h(r.\text{left}), h(r.\text{right})\} & \text{otherwise.} \end{cases}$$

The worst case run time of the search is thus $\mathcal{O}(h(T))$

## Insertion of a key

Insertion of the key $k$

- Search for $k$
- If successful search: output error
- Of no success: insert the key at the leaf reached



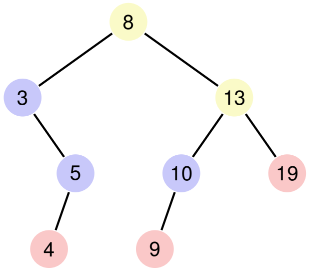
Insert (5)

## Remove node

Three cases possible:

- Node has no children
- Node has one child
- Node has two children

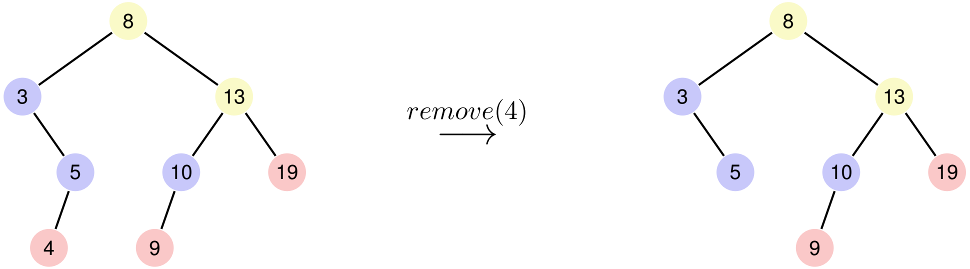
[Leaves do not count here]



## Remove node

Node has no children

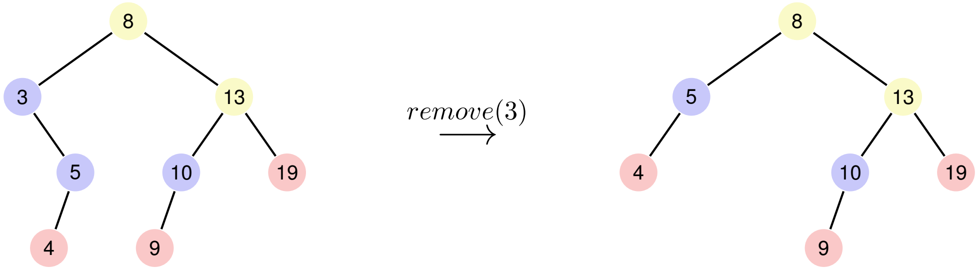
Simple case: replace node by leaf.



## Remove node

Node has one child

Also simple: replace node by single child.



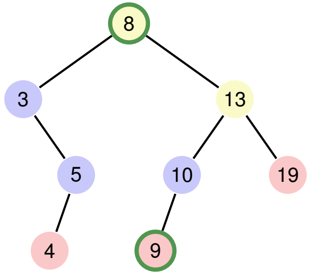
## Remove node

### Node `v` has two children

The following observation helps: the smallest key in the right subtree `v.right` (the *symmetric successor* of `v`)

- is smaller than all keys in `v.right`
- is greater than all keys in `v.left`
- and cannot have a left child.

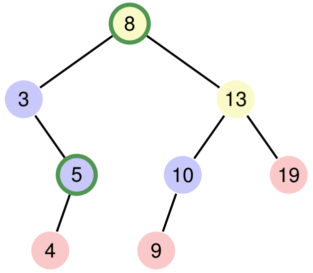
Solution: replace `v` by its symmetric successor.



## By symmetry...

Node $v$ has two children

Also possible: replace $v$ by its symmetric predecessor.



# Algorithm SymmetricSuccessor($v$)

**Input:** Node $v$ of a binary search tree.
**Output:** Symmetric successor of $v$

```
$w \leftarrow v.\mathrm{right}$
$x \leftarrow w.\mathrm{left}$
while $x \neq$ null do
    $w \leftarrow x$
    $x \leftarrow x.\mathrm{left}$
return w
```

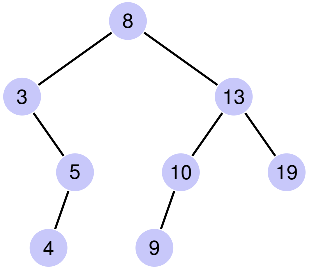
## Analysis

Deletion of an element $v$ from a tree $T$ requires $\mathcal{O}(h(T))$ fundamental steps:

- Finding $v$ has costs $\mathcal{O}(h(T))$
- If $v$ has maximal one child unequal to **null**then removal takes $\mathcal{O}(1)$ steps
- Finding the symmetric successor $n$ of $v$ takes $\mathcal{O}(h(T))$ steps. Removal and insertion of $n$ takes $\mathcal{O}(1)$ steps.

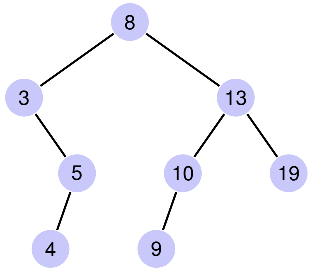
## Traversal possibilities

- preorder: $v$, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- postorder: $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$, then $v$.
  4, 5, 3, 9, 10, 19, 13, 8
- inorder: $T_{\text{left}}(v)$, then $v$, then $T_{\text{right}}(v)$.
  3, 4, 5, 8, 9, 10, 13, 19

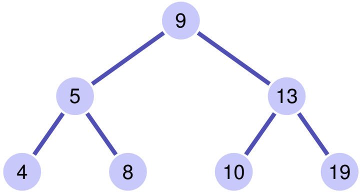


## Further supported operations

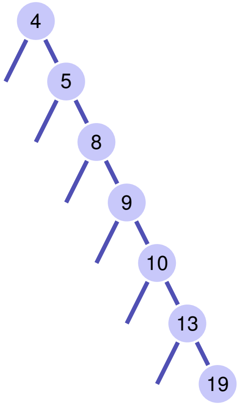
- Min($T$): Read-out minimal value in $\mathcal{O}(h)$
- ExtractMin($T$): Read-out and remove minimal value in $\mathcal{O}(h)$
- List($T$): Output the sorted list of elements
- Join($T_1, T_2$): Merge two trees with $\max(T_1) < \min(T_2)$ in $\mathcal{O}(n)$.



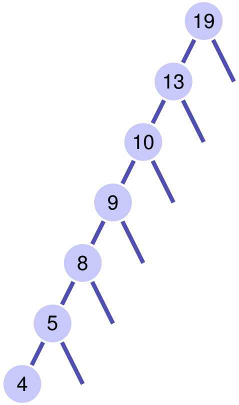
## Degenerated search trees



Insert 9,5,13,4,8,10,19
ideally balanced



Insert 4,5,8,9,10,13,19
linear list



Insert 19,13,10,9,8,5,4
linear list

## Probabilistically

A search tree constructed from a random sequence of numbers provides an an expected path length of $\mathcal{O}(\log n)$.

Attention: this only holds for insertions. If the tree is constructed by random insertions and deletions, the expected path length is $\mathcal{O}(\sqrt{n})$.

*Balanced* trees make sure (e.g. with *rotations*) during insertion or deletion that the tree stays balanced and provide a $\mathcal{O}(\log n)$ Worst-case guarantee.

# 17. AVL Trees

Balanced Trees [Ottman/Widmayer, Kap. 5.2-5.2.1, Cormen et al, Kap. Problem 13-3]

## Objective

Searching, insertion and removal of a key in a tree generated from $n$ keys inserted in random order takes expected number of steps $\mathcal{O}(\log_2 n)$.

But worst case $\Theta(n)$ (degenerated tree).

**Goal:** avoidance of degeneration. Artificial balancing of the tree for each update-operation of a tree.

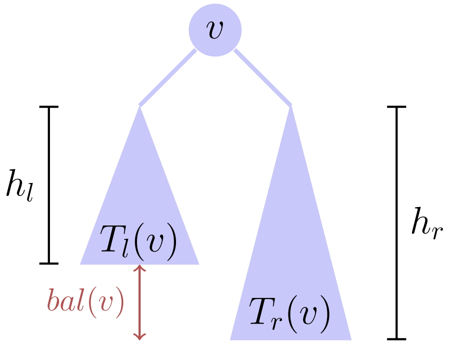
Balancing: guarantee that a tree with $n$ nodes always has a height of $\mathcal{O}(\log n)$.

**Adelson-Venskii and Landis (1962): AVL-Trees**

## Balance of a node

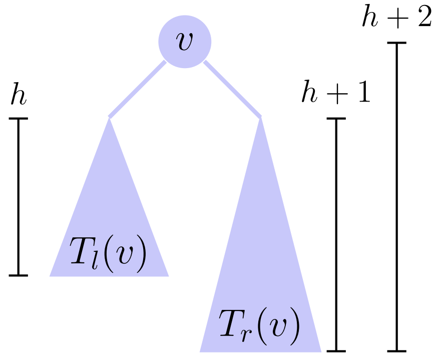
The height *balance* of a node $v$ is defined as the height difference of its sub-trees $T_l(v)$ and $T_r(v)$

$$\text{bal}(v) := h(T_r(v)) - h(T_l(v))$$

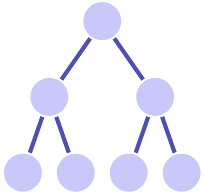


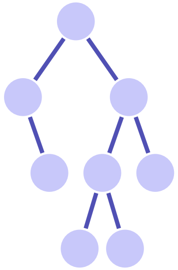
## AVL Condition

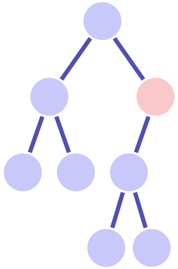
*AVL Condition: for eacn node $v$ of a tree* $\mathrm{bal}(v) \in \{-1, 0, 1\}$



# (Counter-)Examples


AVL tree with height 2

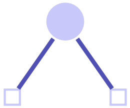

AVL tree with height 3


No AVL tree

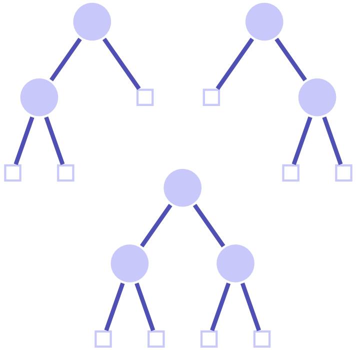
## Number of Leaves

- 1. observation: a binary search tree with $n$ keys provides exactly $n+1$ leaves. Simple induction argument.
  - The binary search tree with $n = 0$ keys has $m = 1$ leaves
  - When a key is added ($n \to n+1$), then it replaces a leaf and adds two new leafs ($m \to m - 1 + 2 = m + 1$).
- 2. observation: a lower bound of the number of leaves in a search tree with given height implies an upper bound of the height of a search tree with given number of keys.

## Lower bound of the leaves



AVL tree with height $1$ has $N(1) := 2$ leaves.



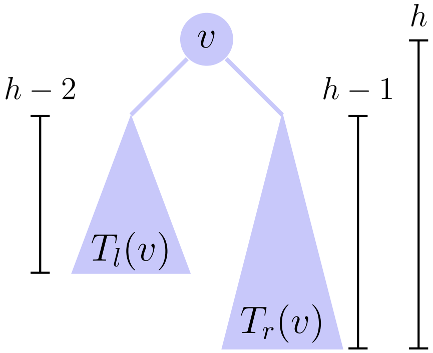
AVL tree with height $2$ has at least $N(2) := 3$ leaves.

## Lower bound of the leaves for $h > 2$

- Height of one subtree $\geq h-1$.
- Height of the other subtree $\geq h-2$.

Minimal number of leaves $N(h)$ is

$$N(h) = N(h-1) + N(h-2)$$



Overal we have $N(h) = F_{h+2}$ with *Fibonacci-numbers* $F_0 := 0$, $F_1 := 1$, $F_n := F_{n-1} + F_{n-2}$ for $n > 1$.

## Fibonacci Numbers, closed Form

It holds that[28]

$$F_i = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)$$

with the roots $\phi, \hat{\phi}$ of the golden ratio equation $x^2 - x - 1 = 0$:

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$$
$$\hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -0.618$$

[28] Derivation using generating functions (power series) in the appendix.

## Fibonacci Numbers, Inductive Proof

$F_i \overset{!}{=} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i) \qquad [*] \qquad \left(\phi = \frac{1+\sqrt{5}}{2}, \hat{\phi} = \frac{1-\sqrt{5}}{2}\right)$.

1. Immediate for $i = 0, i = 1$.
2. Let $i > 2$ and claim $[*]$ true for all $F_j$, $j < i$.

$$F_i \overset{def}{=} F_{i-1} + F_{i-2} \overset{[*]}{=} \frac{1}{\sqrt{5}}(\phi^{i-1} - \hat{\phi}^{i-1}) + \frac{1}{\sqrt{5}}(\phi^{i-2} - \hat{\phi}^{i-2})$$

$$= \frac{1}{\sqrt{5}}(\phi^{i-1} + \phi^{i-2}) - \frac{1}{\sqrt{5}}(\hat{\phi}^{i-1} + \hat{\phi}^{i-2}) = \frac{1}{\sqrt{5}}\phi^{i-2}(\phi + 1) - \frac{1}{\sqrt{5}}\hat{\phi}^{i-2}(\hat{\phi} + 1)$$

($\phi, \hat{\phi}$ fulfil $x + 1 = x^2$)

$$= \frac{1}{\sqrt{5}}\phi^{i-2}(\phi^2) - \frac{1}{\sqrt{5}}\hat{\phi}^{i-2}(\hat{\phi}^2) = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i).$$

## Tree Height

Because $|\hat{\phi}| < 1$, overal we have

$$N(h) \in \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^h\right) \subseteq \Omega(1.618^h)$$

and thus

$$\begin{aligned} N(h) &\geq c \cdot 1.618^h \\ \Rightarrow \quad h &\leq 1.44 \log_2 n + c'. \end{aligned}$$

An AVL tree is asymptotically not more than $44\%$ higher than a perfectly balanced tree.[29]

[29] The perfectly balanced tree has a height of $\lceil \log_2 n + 1 \rceil$

## Insertion

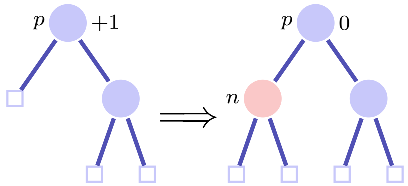
Balance

- Keep the balance stored in each node
- Re-balance the tree in each update-operation

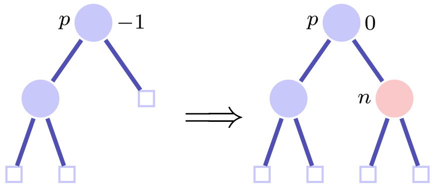
New node $n$ is inserted:

- Insert the node as for a search tree.
- Check the balance condition increasing from $n$ to the root.

# Balance at Insertion Point



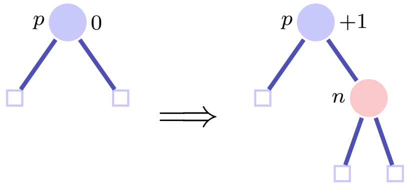
case 1: $\mathrm{bal}(p) = +1$



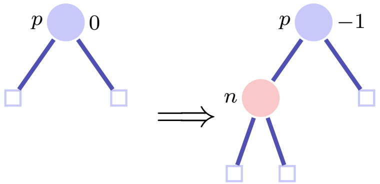
case 2: $\mathrm{bal}(p) = -1$

Finished in both cases because the subtree height did not change

## Balance at Insertion Point



case 3.1: $\mathrm{bal}(p) = 0$ right



case 3.2: $\mathrm{bal}(p) = 0$, left

Not finished in both case. Call of `upin(p)`

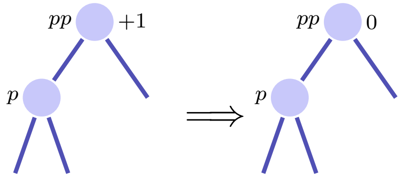
## upin(p) - invariant

When `upin(p)` is called it holds that

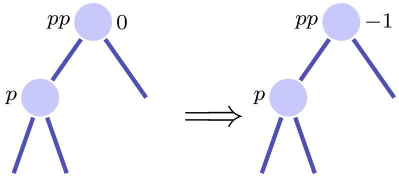
- the subtree from $p$ is grown and
- $\mathrm{bal}(p) \in \{-1, +1\}$

# upin(p)

Assumption: $p$ is left son of $pp$[30]



case 1: $\mathrm{bal}(pp) = +1$, done.



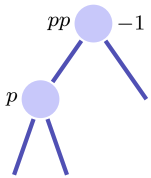
case 2: $\mathrm{bal}(pp) = 0$, `upin(pp)`

In both cases the AVL-Condition holds for the subtree from $pp$

[30] If $p$ is a right son: symmetric cases with exchange of $+1$ and $-1$

## upin(p)

Assumption: $p$ is left son of $pp$



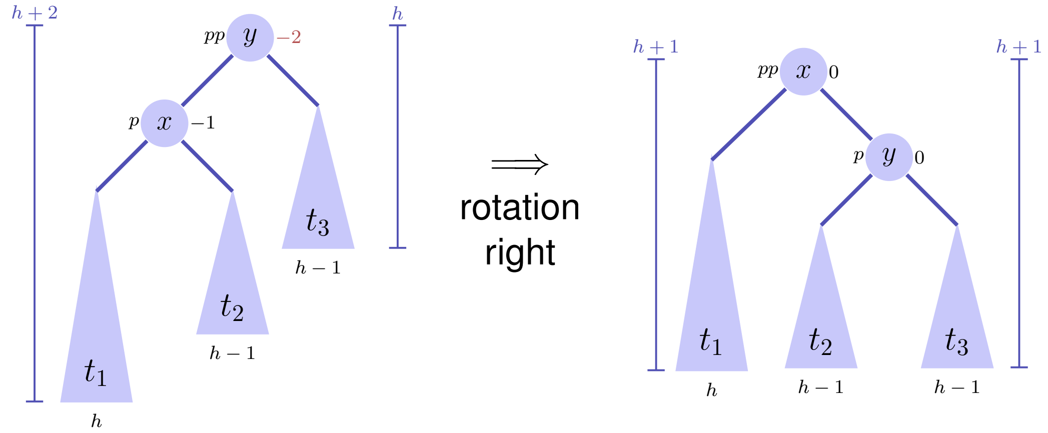
case 3: $\mathrm{bal}(pp) = -1$,

This case is problematic: adding $n$ to the subtree from $pp$ has violated the AVL-condition. Re-balance!

Two cases $\mathrm{bal}(p) = -1$, $\mathrm{bal}(p) = +1$

## Rotations

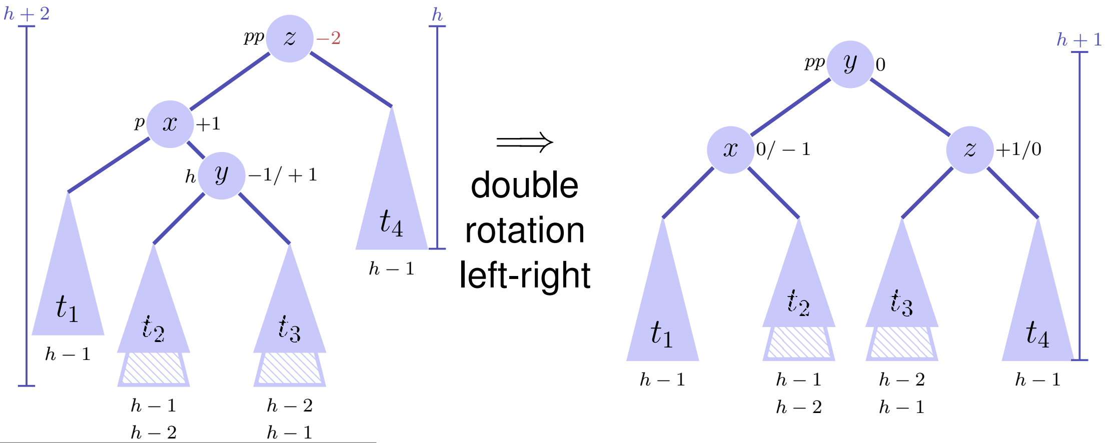
case 1.1 $\mathrm{bal}(p) = -1.$ [31]



[31] $p$ right son: $\Rightarrow \mathrm{bal}(pp) = \mathrm{bal}(p) = +1$, left rotation

## Rotations

case 1.1 $\mathrm{bal}(p) = -1.$ [32]



[32] $p$ right son $\Rightarrow \mathrm{bal}(pp) = +1$, $\mathrm{bal}(p) = -1$, double rotation right left

## Analysis

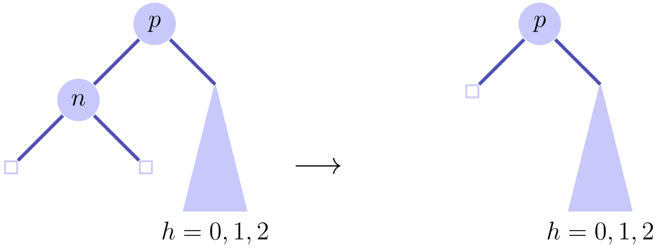
- Tree height: $\mathcal{O}(\log n)$.
- Insertion like in binary search tree.
- Balancing via recursion from node to the root. Maximal path lenght $\mathcal{O}(\log n)$.

Insertion in an AVL-tree provides run time costs of $\mathcal{O}(\log n)$.

## Deletion

Case 1: Children of node $n$ are both leaves Let $p$ be parent node of $n$. $\Rightarrow$ Other subtree has height $h' = 0$, $1$ or $2$.

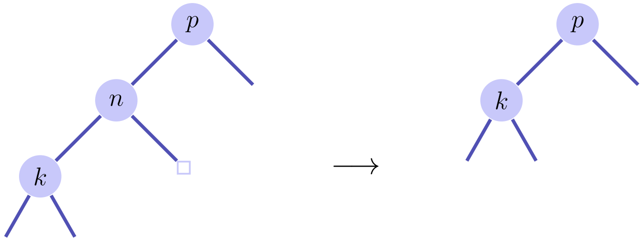
- $h' = 1$: Adapt $\mathrm{bal}(p)$.
- $h' = 0$: Adapt $\mathrm{bal}(p)$. Call `upout(p)`.
- $h' = 2$: Rebalanciere des Teilbaumes. Call `upout(p)`.



## Deletion

Case 2: one child $k$ of node $n$ is an inner node

- Replace $n$ by $k$. `upout(k)`



## Deletion

Case 3: both children of node $n$ are inner nodes

- Replace $n$ by symmetric successor. `upout(k)`
- Deletion of the symmetric successor is as in case 1 or 2.

## `upout(p)`

Let $pp$ be the parent node of $p$.

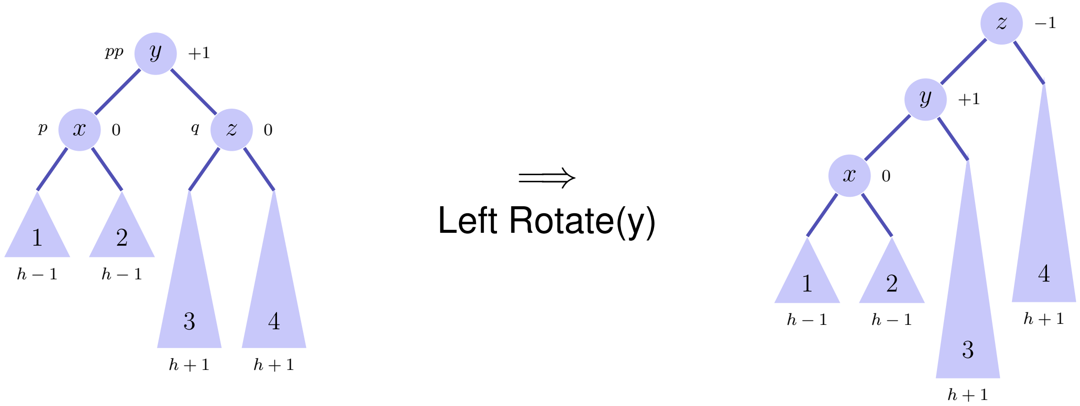
(a) $p$ left child of $pp$

1. $\mathrm{bal}(pp) = -1 \;\Rightarrow\; \mathrm{bal}(pp) \leftarrow 0$. `upout(pp)`
2. $\mathrm{bal}(pp) = 0 \;\Rightarrow\; \mathrm{bal}(pp) \leftarrow +1$.
3. $\mathrm{bal}(pp) = +1 \Rightarrow$ next slides.

(b) $p$ right child of $pp$: Symmetric cases exchanging $+1$ and $-1$.

## upout(p)

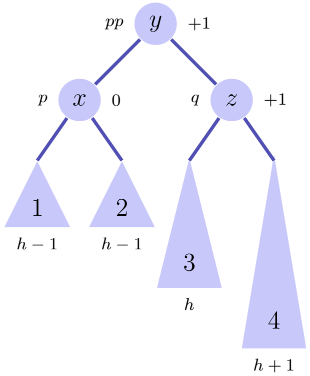
Case (a).3: $\mathrm{bal}(pp) = +1$. Let $q$ be brother of $p$
(a).3.1: $\mathrm{bal}(q) = 0$.[33]



[33] (b).3.1: $\mathrm{bal}(pp) = -1$, $\mathrm{bal}(q) = -1$, Right rotation

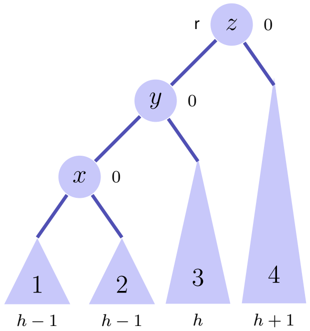
## upout(p)

Case (a).3: $\text{bal}(pp) = +1$. (a).3.2: $\text{bal}(q) = +1$.[34]



$\Longrightarrow$

Left Rotate(y)

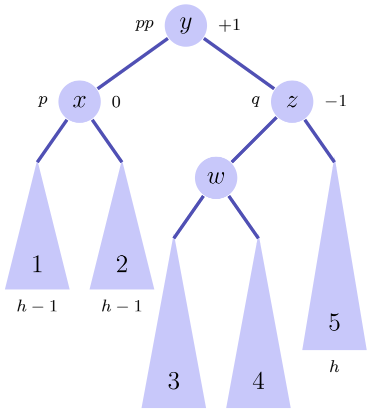


plus `upout(r)`.

[34] (b).3.2: $\text{bal}(pp) = -1$, $\text{bal}(q) = +1$, Right rotation+upout

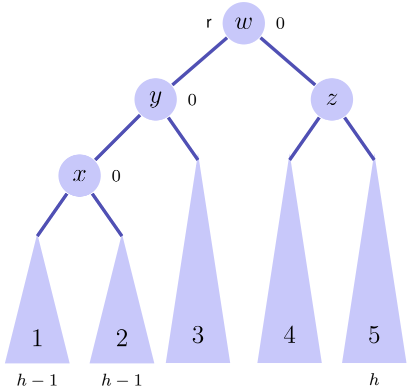
## upout(p)

Case (a).3: $\mathrm{bal}(pp) = +1$. (a).3.3: $\mathrm{bal}(q) = -1$.[35]



$\Longrightarrow$

Rotate right (z) left (y)



plus `upout(r)`.

[35](b).3.3: $\mathrm{bal}(pp) = -1$, $\mathrm{bal}(q) = -1$, left-right rotation + upout

## Conclusion

- AVL trees have worst-case asymptotic runtimes of $\mathcal{O}(\log n)$ for searching, insertion and deletion of keys.
- Insertion and deletion is relatively involved and an overkill for really small problems.

## 17.5 Appendix

Derivation of some mathemmatical formulas

## [Fibonacci Numbers: closed form]

Closed form of the Fibonacci numbers: computation via generation functions:

1. Power series approach

$$f(x) := \sum_{i=0}^{\infty} F_i \cdot x^i$$

## [Fibonacci Numbers: closed form]

2 For Fibonacci Numbers it holds that $F_0 = 0$, $F_1 = 1$, $F_i = F_{i-1} + F_{i-2} \; \forall \; i > 1$. Therefore:

$$\begin{aligned} f(x) = x + \sum_{i=2}^{\infty} F_i \cdot x^i &= x + \sum_{i=2}^{\infty} F_{i-1} \cdot x^i + \sum_{i=2}^{\infty} F_{i-2} \cdot x^i \\ &= x + x \sum_{i=2}^{\infty} F_{i-1} \cdot x^{i-1} + x^2 \sum_{i=2}^{\infty} F_{i-2} \cdot x^{i-2} \\ &= x + x \sum_{i=0}^{\infty} F_i \cdot x^i + x^2 \sum_{i=0}^{\infty} F_i \cdot x^i \\ &= x + x \cdot f(x) + x^2 \cdot f(x). \end{aligned}$$

## [Fibonacci Numbers: closed form]

3 Thus:

$$\begin{aligned} & f(x)\cdot(1-x-x^2) = x. \\ \Leftrightarrow \quad & f(x) = \frac{x}{1-x-x^2} = -\frac{x}{x^2+x-1} \end{aligned}$$

with the roots $-\phi$ and $-\hat{\phi}$ of $x^2+x-1$,

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.6, \qquad \hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -0.6.$$

it holds that $\phi\cdot\hat{\phi} = -1$ and thus

$$f(x) = -\frac{x}{(x+\phi)\cdot(x+\hat{\phi})} = \frac{x}{(1-\phi x)\cdot(1-\hat{\phi}x)}$$

## [Fibonacci Numbers: closed form]

4 It holds that:

$$(1-\hat{\phi}x)-(1-\phi x)=\sqrt{5}\cdot x.$$

Damit:

$$\begin{aligned} f(x) &= \frac{1}{\sqrt{5}}\frac{(1-\hat{\phi}x)-(1-\phi x)}{(1-\phi x)\cdot(1-\hat{\phi}x)} \\ &= \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi x}-\frac{1}{1-\hat{\phi}x}\right) \end{aligned}$$

## [Fibonacci Numbers: closed form]

5 Power series of $g_a(x) = \frac{1}{1-a\cdot x}$ ($a \in \mathbb{R}$):

$$\frac{1}{1-a\cdot x} = \sum_{i=0}^{\infty} a^i \cdot x^i.$$

E.g. Taylor series of $g_a(x)$ at $x = 0$ or like this: Let $\sum_{i=0}^{\infty} G_i \cdot x^i$ a power series of $g$. By the identity $g_a(x)(1 - a \cdot x) = 1$ it holds that for all $x$ (within the radius of convergence)

$$1 = \sum_{i=0}^{\infty} G_i \cdot x^i - a \cdot \sum_{i=0}^{\infty} G_i \cdot x^{i+1} = G_0 + \sum_{i=1}^{\infty} (G_i - a \cdot G_{i-1}) \cdot x^i$$

For $x = 0$ it follows $G_0 = 1$ and for $x \neq 0$ it follows then that $G_i = a \cdot G_{i-1} \Rightarrow G_i = a^i$.

## [Fibonacci Numbers: closed form]

6 Fill in the power series:

$$f(x) = \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi x} - \frac{1}{1-\hat{\phi} x}\right) = \frac{1}{\sqrt{5}}\left(\sum_{i=0}^{\infty} \phi^i x^i - \sum_{i=0}^{\infty} \hat{\phi}^i x^i\right)$$

$$= \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i) x^i$$

Comparison of the coefficients with $f(x) = \sum_{i=0}^{\infty} F_i \cdot x^i$ yields

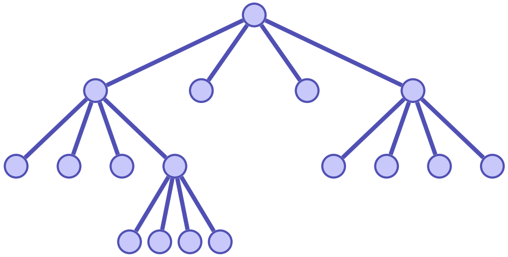
$$F_i = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i).$$

# 18. Quadtrees

Quadtrees, Collision Detection, Image Segmentation

## Quadtree

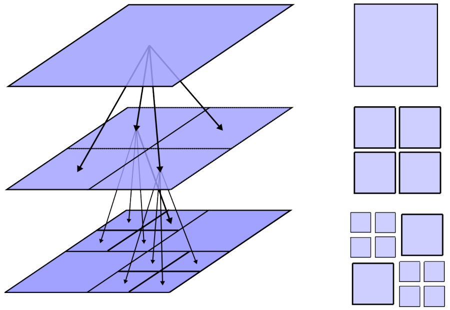
A quad tree is a tree of order 4.



... and as such it is not particularly interesting except when it is used for ...

## Quadtree - Interpretation und Nutzen

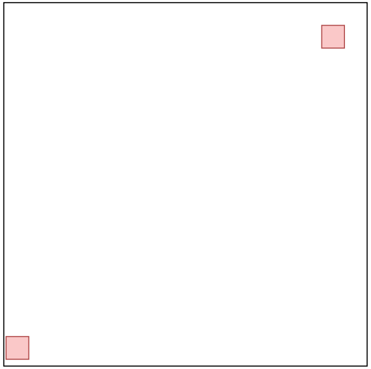
Separation of a two-dimensional range into 4 equally sized parts.



[analogously in three dimensions with an *octtree* (tree of order 8)]

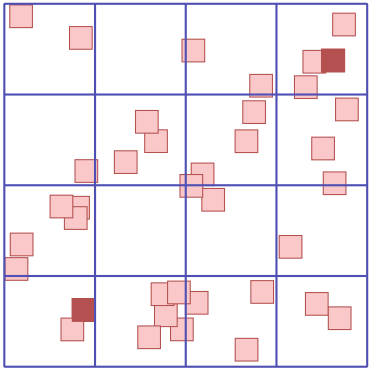
## Example 1: Collision Detection

- Objects in the 2D-plane, e.g. particle simulation on the screen.
- Goal: collision detection



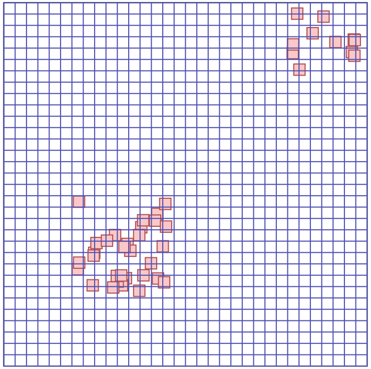
## Idea

- Many objects: $n^2$ detections (naively)
- Improvement?
- Obviously: collision detection not required for objects far away from each other
- What is „far away”?
- Grid ($m \times m$)
- Collision detection per grid cell



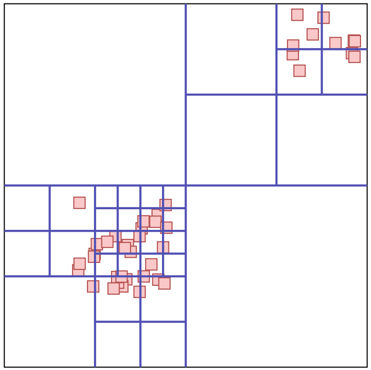
## Grids

- A grid often helps, but not always
- Improvement?
- More finegrained grid?
- Too many grid cells!



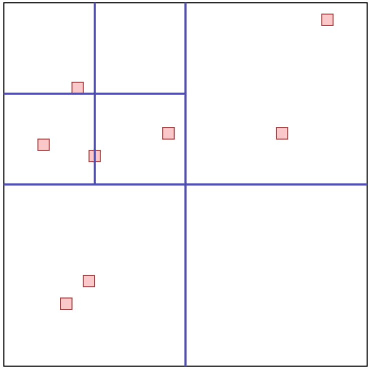
## Adaptive Grids

- A grid often helps, but not always
- Improvement?
- *Adaptively* refine grid
- Quadtree!



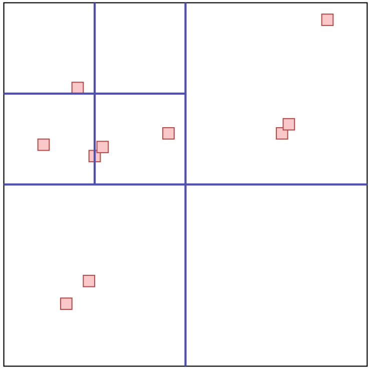
## Algorithm: Insertion

- Quadtree starts with a single node
- Objects are added to the node. When a node contains too many objects, the node is split.
- Objects that are on the boundary of the quadtree remain in the higher level node.

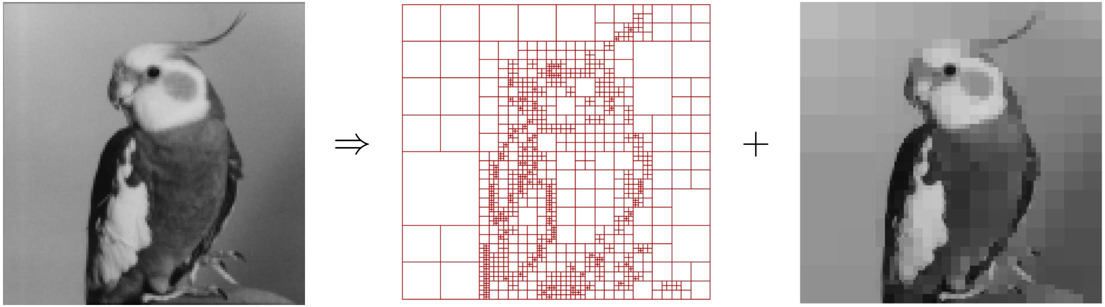


## Algorithm: Collision Detection

- Run through the quadtree in a recursive way. For each node test collision with all objects contained in the same or (recursively) contained nodes.

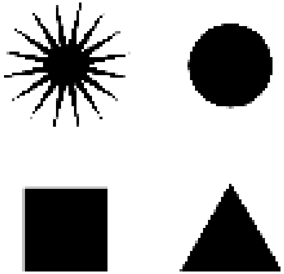


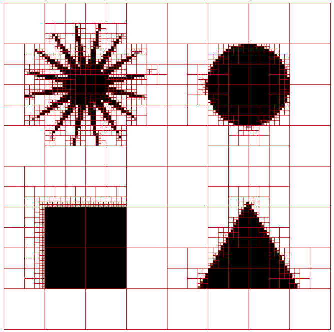
## Example 2: Image Segmentation



(Possible applications: compression, denoising, edge detection)

## Quadtree on Monochrome Bitmap

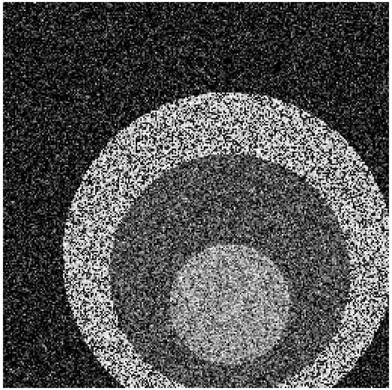


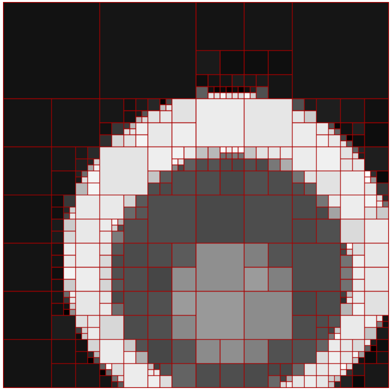


Similar procedure to generate the quadtree: split nodes recursively until each node only contains pixels of the same color.

## Quadtree with Approximation

When there are more than two color values, the quadtree can get very large. $\Rightarrow$ Compressed representation: *approximate* the image piecewise constant on the rectangles of a quadtree.





## Piecewise Constant Approximation

(Grey-value) Image $z \in \mathbb{R}^S$ on pixel indices $S$. [36]

Rectangle $r \subset S$.

Goal: determine

$$\arg\min_{x \in r} \sum_{s \in r} (z_s - x)^2$$

Solution: the arithmetic mean $\mu_r = \frac{1}{|r|} \sum_{s \in r} z_s$

[36] we assume that $S$ is a square with side length $2^k$ for some $k \geq 0$

## Intermediate Result

The (w.r.t. mean squared error) best approximation

$$\mu_r = \frac{1}{|r|} \sum_{s \in r} z_s$$

and the corresponding error

$$\sum_{s \in r} (z_s - \mu_r)^2 =: \|z_r - \mu_r\|_2^2$$

can be computed quickly after a $\mathcal{O}(|S|)$ tabulation: prefix sums!

## Which Quadtree?

Conflict

- *As close as possible to the data* $\Rightarrow$ small rectangles, large quadtree . Extreme case: one node per pixel. Approximation = original
- *Small amount of nodes* $\Rightarrow$ large rectangles, small quadtree Extreme case: a single rectangle. Approximation = a single grey value.

## Which Quadtree?

Idea: choose between data fidelity and complexity with a regularisation parameter $\gamma \geq 0$

Choose quadtree $T$ with leaves[37] $L(T)$ such that it minimizes the following function

$$H_\gamma(T, z) := \gamma \cdot \underbrace{|L(T)|}_{\text{Number of Leaves}} + \underbrace{\sum_{r \in L(T)} \|z_r - \mu_r\|_2^2}_{\text{Cummulative approximation error of all leaves}} .$$

[37] here: leaf: node with null-children

## Regularisation

Let $T$ be a quadtree over a rectangle $S_T$ and let $T_{ll}, T_{lr}, T_{ul}, T_{ur}$ be the four possible sub-trees and

$$\widehat{H}_\gamma(T, z) := \min_T \gamma \cdot |L(T)| + \sum_{r \in L(T)} \|z_r - \mu_r\|_2^2$$

Extreme cases:
$\gamma = 0 \Rightarrow$ original data;
$\gamma \to \infty \Rightarrow$ a single rectangle

## Observation: Recursion

- If the (sub-)quadtree $T$ represents only one pixel, then it cannot be split and it holds that

$$\widehat{H}_\gamma(T, z) = \gamma$$

- Let, otherwise,

$$\begin{aligned} M_1 &:= \gamma + \|z_{S_T} - \mu_{S_T}\|_2^2 \\ M_2 &:= \widehat{H}_\gamma(T_{ll}, z) + \widehat{H}_\gamma(T_{lr}, z) + \widehat{H}_\gamma(T_{ul}, z) + \widehat{H}_\gamma(T_{ur}, z) \end{aligned}$$

then

$$\widehat{H}_\gamma(T, z) = \min\{\underbrace{M_1(T, \gamma, z)}_{\text{no split}}, \underbrace{M_2(T, \gamma, z)}_{\text{split}}\}$$

## Algorithmus: Minimize($z$,$r$,$\gamma$)

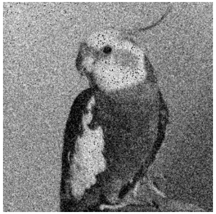
**Input:** Image data $z \in \mathbb{R}^S$, rectangle $r \subset S$, regularization $\gamma > 0$
**Output:** $\min_T \gamma|L(T)| + \|z - \mu_{L(T)}\|_2^2$

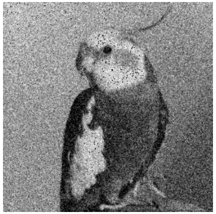
```
if $|r| = 0$ then return $0$
$m \leftarrow \gamma + \sum_{s\in r} (z_s - \mu_r)^2$
if $|r| > 1$ then
    Split $r$ into $r_{ll}$,$r_{lr}$,$r_{ul}$,$r_{ur}$
    $m_1 \leftarrow \mathrm{Minimize}(z, r_{ll}, \gamma)$; $m_2 \leftarrow \mathrm{Minimize}(z, r_{lr}, \gamma)$
    $m_3 \leftarrow \mathrm{Minimize}(z, r_{ul}, \gamma)$; $m_4 \leftarrow \mathrm{Minimize}(z, r_{ur}, \gamma)$
    $m' \leftarrow m_1 + m_2 + m_3 + m_4$
else
    $m' \leftarrow \infty$
if $m' < m$ then $m \leftarrow m'$
return $m$
```

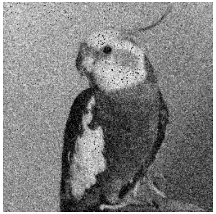
## Analysis

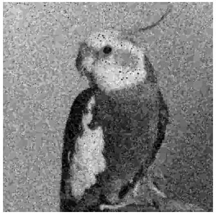
The minimization algorithm over dyadic partitions (quadtrees) takes $\mathcal{O}(|S| \log |S|)$ steps.

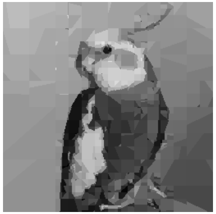
# Application: Denoising (with addditional Wedgelets)

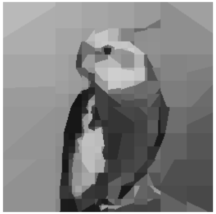

noised

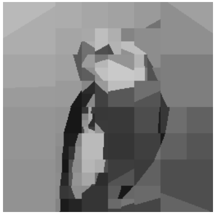

$\gamma = 0.003$

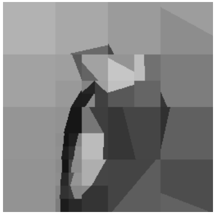

$\gamma = 0.01$

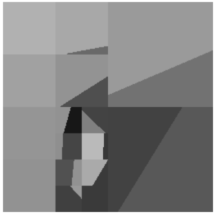

$\gamma = 0.03$


$\gamma = 0.1$

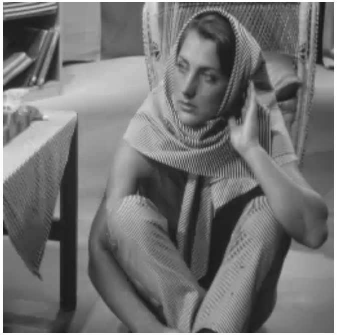
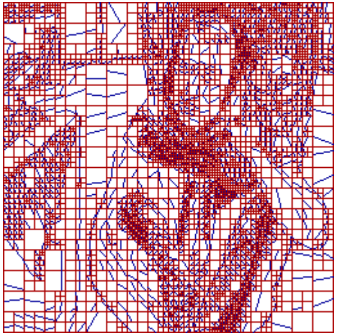

$\gamma = 0.3$


$\gamma = 1$


$\gamma = 3$

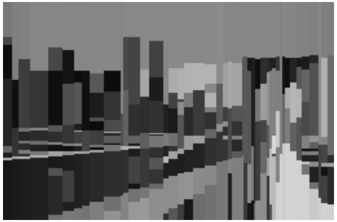

$\gamma = 10$

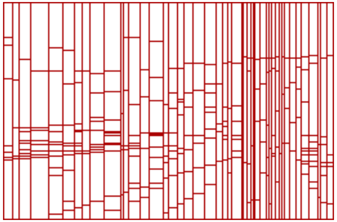
## Extensions: Affine Regression + Wedgelets

## Other ideas

no quadtree: hierarchical one-dimensional modell (requires dynamic programming)





# 19. Dynamic Programming I

Memoization, Optimal Substructure, Overlapping Sub-Problems, Dependencies, General Procedure. Examples: Fibonacci, Rod Cutting, Longest Ascending Subsequence, Longest Common Subsequence, Edit Distance, Matrix Chain Multiplication (Strassen)

[Ottman/Widmayer, Kap. 1.2.3, 7.1, 7.4, Cormen et al, Kap. 15]

## Fibonacci Numbers

(again)

$$F_n := \begin{cases} n & \text{if } n < 2 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

Analysis: why ist the recursive algorithm so slow?

## Algorithm FibonacciRecursive($n$)

```
Input: $n \geq 0$
Output: $n$-th Fibonacci number
if $n < 2$ then
    $f \leftarrow n$
else
    $f \leftarrow$ FibonacciRecursive$(n-1)$ + FibonacciRecursive$(n-2)$
return $f$
```

## Analysis

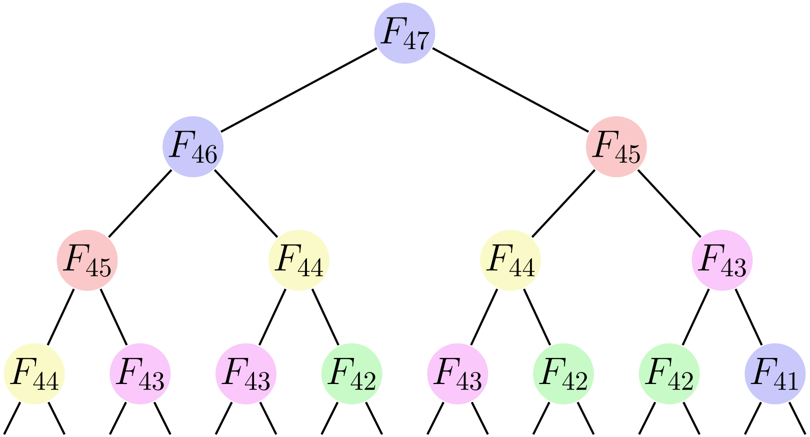
$T(n)$: Number executed operations.

- $n = 0, 1$: $T(n) = \Theta(1)$
- $n \geq 2$: $T(n) = T(n-2) + T(n-1) + c$.

$$T(n) = T(n-2) + T(n-1) + c \geq 2T(n-2) + c \geq 2^{n/2}c' = (\sqrt{2})^n c'$$

Algorithm is *exponential* in $n$.

## Reason (visual)



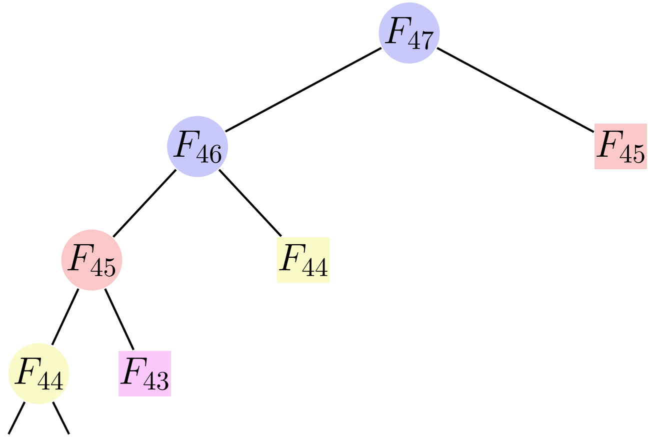
Nodes with same values are evaluated (too) often.

## Memoization

*Memoization* (sic) saving intermediate results.

- Before a subproblem is solved, the existence of the corresponding intermediate result is checked.
- If an intermediate result exists then it is used.
- Otherwise the algorithm is executed and the result is saved accordingly.

## Memoization with Fibonacci



Rechteckige Knoten wurden bereits ausgewertet.

## Algorithm FibonacciMemoization($n$)

**Input:** $n \geq 0$
**Output:** $n$-th Fibonacci number

```
if $n \leq 2$ then
    $f \leftarrow 1$
else if $\exists$memo$[n]$ then
    $f \leftarrow$ memo$[n]$
else
    $f \leftarrow$ FibonacciMemoization$(n-1)$ + FibonacciMemoization$(n-2)$
    memo$[n] \leftarrow f$
return $f$
```

## Analysis

Computational complexity:

$$T(n) = T(n-1) + c = ... = \mathcal{O}(n).$$

because after the call to $f(n-1)$, $f(n-2)$ has already been computed.

A different argument: $f(n)$ is computed exactly once recursively for each $n$. Runtime costs: $n$ calls with $\Theta(1)$ costs per call $n \cdot c \in \Theta(n)$. The recursion vanishes from the running time computation.

Algorithm requires $\Theta(n)$ memory.[38]

[38] But the naive recursive algorithm also requires $\Theta(n)$ memory implicitly.

## Looking closer ...

... the algorithm computes the values of $F_1$, $F_2$, $F_3$,. . . in the *top-down* approach of the recursion.

Can write the algorithm *bottom-up*. This is characteristic for *dynamic programming*.

# Algorithm FibonacciBottomUp(n)

```
Input: $n \geq 0$
Output: $n$-th Fibonacci number
$F[1] \leftarrow 1$
$F[2] \leftarrow 1$
for $i \leftarrow 3, \dots, n$ do
    $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$
```

## Dynamic Programming: Idea

- Divide a complex problem into a reasonable number of sub-problems
- The solution of the sub-problems will be used to solve the more complex problem
- Identical problems will be computed only once

## Dynamic Programming Consequence

Identical problems will be computed only once

$\Rightarrow$ Results are saved

We trade spee against memory consumption

## Dynamic Programming: Description

1. Use a *DP-table* with information to the subproblems.
   Dimension of the entries? Semantics of the entries?
2. Computation of the *base cases*
   Which entries do not depend on others?
3. Determine *computation order*.
   In which order can the entries be computed such that dependencies are fulfilled?
4. Read-out the *solution*
   How can the solution be read out from the table?

Runtime (typical) = number entries of the table times required operations per entry.

## Dynamic Programing: Description with the example

1. **Dimension of the table? Semantics of the entries?**

   $n \times 1$ table. $n$th entry contains $n$th Fibonacci number.

2. **Which entries do not depend on other entries?**

   Values $F_1$ and $F_2$ can be computed easily and independently.

3. **What is the execution order such that required entries are always available?**

   $F_i$ with increasing $i$.

4. **Wie kann sich Lösung aus der Tabelle konstruieren lassen?**

   $F_n$ ist die $n$-te Fibonacci-Zahl.

## Dynamic Programming = Divide-And-Conquer ?

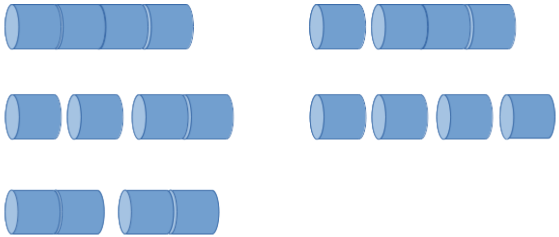
- In both cases the original problem can be solved (more easily) by utilizing the solutions of sub-problems. The problem provides *optimal substructure*.
- Divide-And-Conquer algorithms (such as Mergesort): sub-problems are independent; their solutions are required only once in the algorithm.
- DP: sub-problems are dependent. The problem is said to have *overlapping sub-problems* that are required multiple-times in the algorithm.
- In order to avoid redundant computations, results are tabulated. For *sub-problems there must not be any circular dependencies*.

## Rod Cutting

- Rods (metal sticks) are cut and sold.
- Rods of length $n \in \mathbb{N}$ are available. A cut does not provide any costs.
- For each length $l \in \mathbb{N}$, $l \leq n$ known is the value $v_l \in \mathbb{R}^+$
- Goal: cut the rods such (into $k \in \mathbb{N}$ pieces) that

$$\sum_{i=1}^{k} v_{l_i} \text{ is maximized subject to } \sum_{i=1}^{k} l_i = n.$$

## Rod Cutting: Example


Possibilities to cut a rod of length 4 (without permutations)

| Length | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Price | 0 | 2 | 3 | 8 | 9 |

$\Rightarrow$ Best cut: 3 + 1 with value 10.

## Wie findet man den DP Algorithms

0. Exact formulation of the wanted solution
1. Define sub-problems (and compute the cardinality)
2. Guess / Enumerate (and determine the running time for guessing)
3. Recursion: relate sub-problems
4. Memoize / Tabularize. Determine the dependencies of the sub-problems
5. Solve the problem
   Running time = #sub-problems $\times$ time/sub-problem

## Structure of the problem

0. *Wanted:* $r_n$ = maximal value of rod (cut or as a whole) with length $n$.
1. *sub-problems*: maximal value $r_k$ for each $0 \le k < n$
2. *Guess* the length of the first piece
3. *Recursion*

$$
\begin{aligned}
r_k &= \max\{v_i + r_{k-i} : 0 < i \le k\}, \quad k > 0 \\
r_0 &= 0
\end{aligned}
$$

4. *Dependency:* $r_k$ depends (only) on values $v_i$, $1 \le i \le k$ and the optimal cuts $r_i$, $i < k$
5. *Solution* in $r_n$

## Algorithm RodCut($v$,$n$)

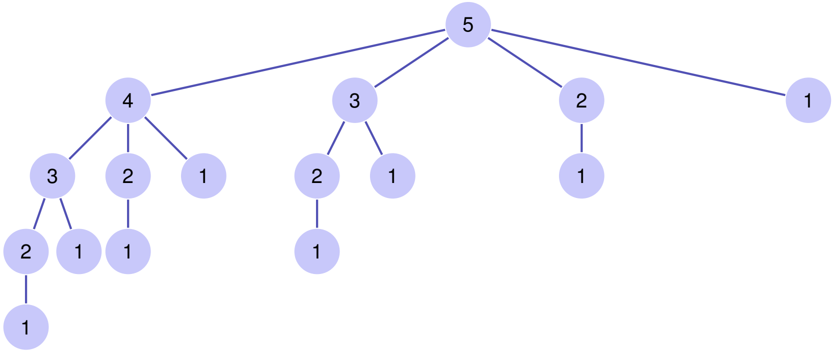
**Input:** $n \geq 0$, Prices $v$
**Output:** best value

```
$q \leftarrow 0$
if $n > 0$ then
    for $i \leftarrow 1, \dots, n$ do
        $q \leftarrow \max\{q, v_i + \text{RodCut}(v, n-i)\}$;
return $q$
```

Running time $T(n) = \sum_{i=0}^{n-1} T(i) + c \quad \Rightarrow$[39] $\quad T(n) \in \Theta(2^n)$

[39] $T(n) = T(n-1) + \sum_{i=0}^{n-2} T(i) + c = T(n-1) + (T(n-1) - c) + c = 2T(n-1) \quad (n > 0)$

## Recursion Tree



## Algorithm RodCutMemoized($m, v, n$)

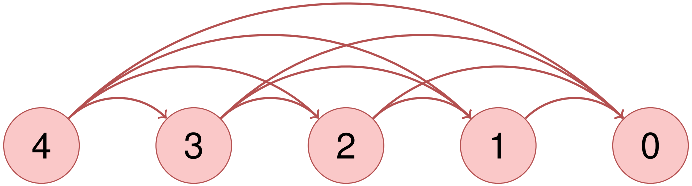
**Input:** $n \geq 0$, Prices $v$, Memoization Table $m$
**Output:** best value

```
$q \leftarrow 0$
**if** $n > 0$ **then**
    **if** $\exists\ m[n]$ **then**
        $q \leftarrow m[n]$
    **else**
        **for** $i \leftarrow 1, \dots, n$ **do**
            $q \leftarrow \max\{q, v_i + \text{RodCutMemoized}(m, v, n-i)\}$;
        $m[n] \leftarrow q$
**return** $q$
```

Running time $\sum_{i=1}^{n} i = \Theta(n^2)$

## Subproblem-Graph

Describes the mutual dependencies of the subproblems


$$\begin{tikzcd}
4 \arrow[r, bend left] \arrow[rr, bend left] \arrow[rrr, bend left] \arrow[rrrr, bend left] & 3 \arrow[r, bend left] \arrow[rr, bend left] \arrow[rrr, bend left] & 2 \arrow[r, bend left] \arrow[rr, bend left] & 1 \arrow[r, bend left] & 0
\end{tikzcd}$$

and must not contain cycles

## Construction of the Optimal Cut

- During the (recursive) computation of the optimal solution for each $k \leq n$ the recursive algorithm determines the optimal length of the first rod
- Store the lenght of the first rod in a separate table of length $n$

## Bottom-up Description with the example

1. **Dimension of the table? Semantics of the entries?**

   $n \times 1$ table. $n$th entry contains the best value of a rod of length $n$.

2. **Which entries do not depend on other entries?**

   Value $r_0$ is $0$

3. **What is the execution order such that required entries are always available?**

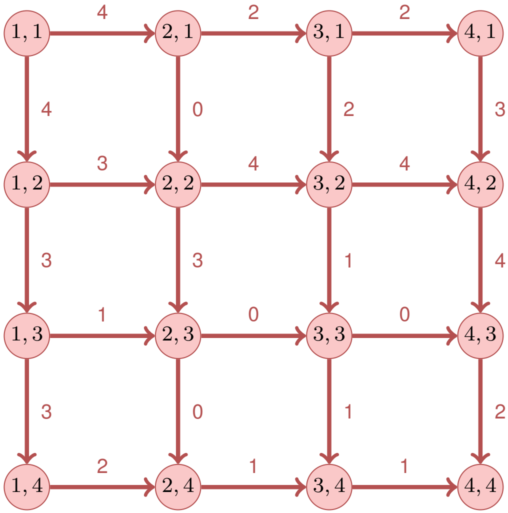
   $r_i$, $i = 1, \ldots, n$.

4. **Wie kann sich Lösung aus der Tabelle konstruieren lassen?**

   $r_n$ is the best value for the rod of length $n$.

## Rabbit!

A rabbit sits on cite $(1, 1)$ of an $n \times n$ grid. It can only move to east or south. On each pathway there is a number of carrots. How many carrots does the rabbit collect maximally?

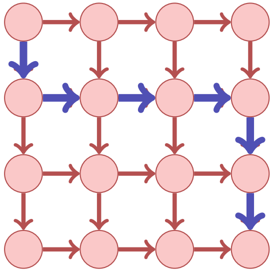


# Rabbit!

Number of possible paths?

- Choice of $n-1$ ways to south out of $2n-2$ ways overal.
- $$\binom{2n-2}{n-1} \in \Omega(2^n)$$

$\Rightarrow$ No chance for a naive algorithm


The path $100011$
(1:to south, 0: to east)

## Recursion

Wanted: *$T_{0,0}$ = maximal number carrots from $(0,0)$ to $(n,n)$.*

Let $w_{(i,j)-(i',j')}$ number of carrots on egde from $(i,j)$ to $(i',j')$.

Recursion (maximal number of carrots from $(i,j)$ to $(n,n)$)

$$T_{ij} = \begin{cases} \max\{w_{(i,j)-(i,j+1)} + T_{i,j+1}, w_{(i,j)-(i+1,j)} + T_{i+1,j}\}, & i < n, j < n \\ w_{(i,j)-(i,j+1)} + T_{i,j+1}, & i = n, j < n \\ w_{(i,j)-(i+1,j)} + T_{i+1,j}, & i < n, j = n \\ 0 & i = j = n \end{cases}$$

## Graph of Subproblem Dependencies

$$\begin{tikzcd}
(1,1) \arrow[r] \arrow[d] & (2,1) \arrow[r] \arrow[d] & (3,1) \arrow[r] \arrow[d] & (4,1) \arrow[d] \\
(1,2) \arrow[r] \arrow[d] & (2,2) \arrow[r] \arrow[d] & (3,2) \arrow[r] \arrow[d] & (4,2) \arrow[d] \\
(1,3) \arrow[r] \arrow[d] & (2,3) \arrow[r] \arrow[d] & (3,3) \arrow[r] \arrow[d] & (4,3) \arrow[d] \\
(1,4) \arrow[r] & (2,4) \arrow[r] & (3,4) \arrow[r] & (4,4)
\end{tikzcd}$$

## Bottom-up Description with the example

1. **Dimension of the table? Semantics of the entries?**

   Table $T$ with size $n \times n$. Entry at $i, j$ provides the maximal number of carrots from $(i, j)$ to $(n, n)$.

2. **Which entries do not depend on other entries?**

   Value $T_{n,n}$ is $0$

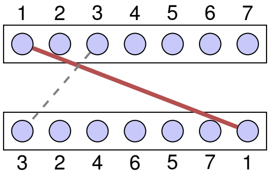
3. **What is the execution order such that required entries are always available?**

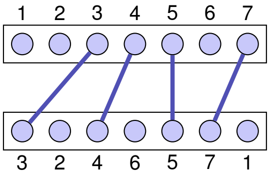
   $T_{i,j}$ with $i = n \searrow 1$ and for each $i$: $j = n \searrow 1$, (or vice-versa: $j = n \searrow 1$ and for each $j$: $i = n \searrow 1$).

4. **Wie kann sich Lösung aus der Tabelle konstruieren lassen?**

   $T_{1,1}$ provides the maximal number of carrots.

## Longest Ascending Sequence (LAS)

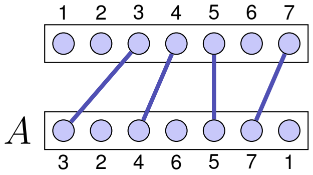




Connect as many as possible fitting ports without lines crossing.

## Formally

- Consider Sequence $A_n = (a_1, \ldots, a_n)$.
- Search for a longest increasing subsequence of $A_n$.
- Examples of increasing subsequences: $(3, 4, 5)$, $(2, 4, 5, 7)$, $(3, 4, 5, 7)$, $(3, 7)$.



**Generalization:** allow any numbers, even with duplicates (still only strictly increasing subsequences permitted). Example: $(2, 3, 3, 3, 5, 1)$ with increasing subsequence $(2, 3, 5)$.

## First idea

Let *$L_i$ = longest ascending subsequence of $A_i$ $(1 \leq i \leq n)$*

Assumption: LAS $L_k$ of $A_k$ known for Now want to compute $L_{k+1}$ for $A_{k+1}$ .

If $a_{k+1}$ fits to $L_k$, then $L_{k+1} = L_k \oplus a_{k+1}$?

Counterexample $A_5 = (1, 2, 5, 3, 4)$. Let $A_3 = (1, 2, 5)$ with $L_3 = A$. Determine $L_4$ from $L_3$?

It does not work this way, we cannot infer $L_{k+1}$ from $L_k$.

## Second idea.

Let $L_i$ = *longest ascending subsequence of* $A_i$ $(1 \leq i \leq n)$

Assumption: a LAS $L_j$ is known for each $j \leq k$. Now compute LAS $L_{k+1}$ for $k+1$.

Look at all fitting $L_{k+1} = L_j \oplus a_{k+1}$ ($j \leq k$) and choose a longest sequence.

Counterexample: $A_5 = (1, 2, 5, 3, 4)$. Let $A_4 = (1, 2, 5, 3)$ with $L_1 = (1)$, $L_2 = (1, 2)$, $L_3 = (1, 2, 5)$, $L_4 = (1, 2, 5)$. Determine $L_5$ from $L_1, \ldots, L_4$?

That does not work either: cannot infer $L_{k+1}$ from only *an arbitrary solution* $L_j$. We need to consider all LAS. Too many.

## Third approach

Let $M_{n,i}$ = *longest ascending subsequence of* $A_i$ $(1 \leq i \leq n)$

Assumption: the LAS $M_j$ for $A_k$, *that end with smallest element* are known for each of the lengths $1 \leq j \leq k$.

Consider all fitting $M_{k,j} \oplus a_{k+1}$ ($j \leq k$) and update the table of the LAS,that end with smallest possible element.

# Third approach Example

Example: $A = (1, 1000, 1001, 4, 5, 2, 6, 7)$

| $A$ | LAT $M_{k,\cdot}$ |
|---|---|
| $1$ | $(1)$ |
| $+\,1000$ | $(1), (1, 1000)$ |
| $+\,1001$ | $(1), (1, 1000), (1, 1000, 1001)$ |
| $+\,4$ | $(1), (1, 4), (1, 1000, 1001)$ |
| $+\,5$ | $(1), (1, 4), (1, 4, 5)$ |
| $+\,2$ | $(1), (1, 2), (1, 4, 5)$ |
| $+\,6$ | $(1), (1, 2), (1, 4, 5), (1, 4, 5, 6)$ |
| $+\,7$ | $(1), (1, 2), (1, 4, 5), (1, 4, 5, 6), (1, 4, 5, 6, 7)$ |

## DP Table

- Idea: save the last element of the increasing sequence $M_{k,j}$ at slot $j$.
- Example: 3 2 5 1 6 4
- Problem: Table does not contain the subsequence, only the last value.
- Solution: second table with the predecessors.

| Index | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Wert | 3 | 2 | 5 | 1 | 6 | 4 |
| Predecessor | $-\infty$ | $-\infty$ | 2 | $-\infty$ | 5 | 1 |

| Index | 0 | 1 | 2 | 3 | 4 | ... |
|---|---|---|---|---|---|---|
| $(L_j)_j$ | -$\infty$ | 1 | 4 | 6 | $\infty$ | |

## Dynamic Programming Algorithm LAS

**Table dimension? Semantics?**

1 Two tables $T[0,\dots,n]$ and $V[1,\dots,n]$.
$T[j]$: last Element of the increasing subequence $M_{n,j}$
$V[j]$: Value of the predecessor of $a_j$.
Start with $T[0] \leftarrow -\infty$, $T[i] \leftarrow \infty\ \forall i > 1$

**Computation of an entry**

2 Entries in $T$ sorted in ascending order. For each new entry $a_{k+1}$ binary search for $l$, such that $T[l] < a_k < T[l+1]$. Set $T[l+1] \leftarrow a_{k+1}$. Set $V[k] = T[l]$.

# Dynamic Programming algorithm LAS

3 **Computation order**

Traverse the list anc compute $T[k]$ and $V[k]$ with ascending $k$

**How can the solution be determined from the table?**

4 Search the largest $l$ with $T[l] < \infty$. $l$ is the last index of the LAS. Starting at $l$ search for the index $i < l$ such that $V[l] = a_i$, $i$ is the predecessor of $l$. Repeat with $l \leftarrow i$ until $T[l] = -\infty$

## Analysis

- Computation of the table:
  - Initialization: $\Theta(n)$ Operations
  - Computation of the $k$th entry: binary search on positions $\{1, \ldots, k\}$ plus constant number of assignments.

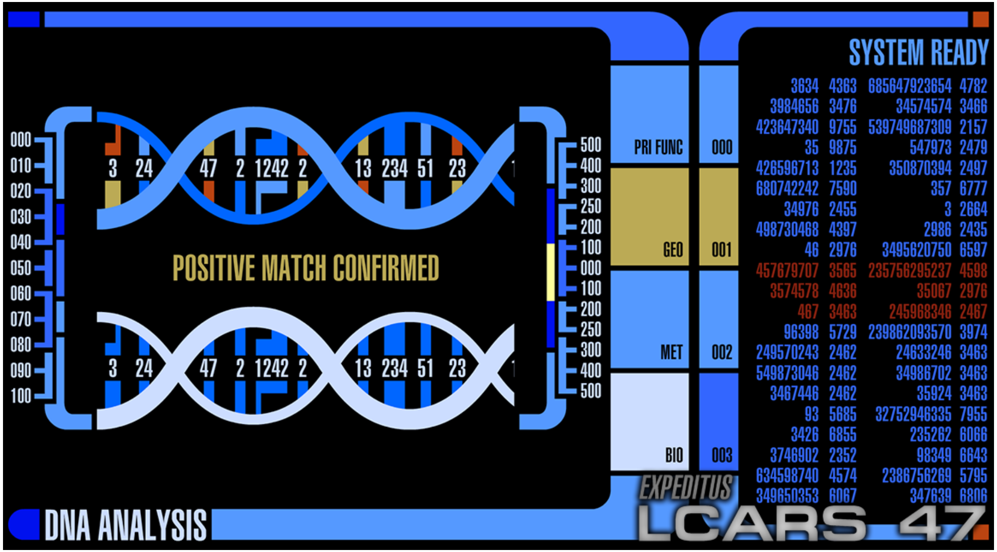
$$\sum_{k=1}^{n} (\log k + \mathcal{O}(1)) = \mathcal{O}(n) + \sum_{k=1}^{n} \log(k) = \Theta(n \log n).$$

- Reconstruction: traverse $A$ from right to left: $\mathcal{O}(n)$.

Overal runtime:

$$\Theta(n \log n).$$

# DNA - Comparison (Star Trek)



## DNA - Comparison

- DNA consists of sequences of four different nucleotides **A**denine **G**uanine **T**hymine **C**ytosine
- DNA sequences (genes) thus can be described with strings of A, G, T and C.
- Possible comparison of two genes: determine the **longest common subsequence**

The longest common subsequence problem is a special case of the minimal edit distance problem. The following slides are therefore not presented in the lectures.

## [Longest common subsequence]

Subsequences of a string:

*Subsequences(KUH): (), (K), (U), (H), (KU), (KH), (UH), (KUH)*

Problem:

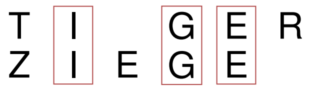
- Input: two strings $A = (a_1, \dots, a_m)$, $B = (b_1, \dots, b_n)$ with lengths $m > 0$ and $n > 0$.
- Wanted: Longest common subsequecnes (LCS) of $A$ and $B$.

## [Longest Common Subsequence]

Examples:

*LGT(IGEL,KATZE)=E, LGT(TIGER,ZIEGE)=IGE*

Ideas to solve?



## [Recursive Procedure]

**Assumption**: solutions $L(i,j)$ known for $A[1,\dots,i]$ and $B[1,\dots,j]$ for all $1 \le i \le m$ and $1 \le j \le n$, but not for $i = m$ and $j = n$.

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| T | I |   | G | E | R |
| Z | I | E | G | E |   |

Consider characters $a_m$, $b_n$. Three possibilities:

1. $A$ is enlarged by one whitespace. $L(m,n) = L(m, n-1)$
2. $B$ is enlarged by one whitespace. $L(m,n) = L(m-1, n)$
3. $L(m,n) = L(m-1, n-1) + \delta_{mn}$ with $\delta_{mn} = 1$ if $a_m = b_n$ and $\delta_{mn} = 0$ otherwise

## [Recursion]

$$L(m,n) \leftarrow \max\{L(m-1,n-1)+\delta_{mn}, L(m,n-1), L(m-1,n)\}$$

for $m,n > 0$ and base cases $L(\cdot,0) = 0$, $L(0,\cdot) = 0$.

|  | ∅ | Z | I | E | G | E |
|---|---|---|---|---|---|---|
| ∅ | 0 | 0 | 0 | 0 | 0 | 0 |
| T | 0 | 0 | 0 | 0 | 0 | 0 |
| I | 0 | 0 | 1 | 1 | 1 | 1 |
| G | 0 | 0 | 1 | 1 | 2 | 2 |
| E | 0 | 0 | 1 | 2 | 2 | 3 |
| R | 0 | 0 | 1 | 2 | 2 | 3 |

## [Dynamic Programming algorithm LCS]

**Dimension of the table? Semantics?**

1 Table $L[0,\dots,m][0,\dots,n]$. $L[i,j]$: length of a LCS of the strings $(a_1,\dots,a_i)$ and $(b_1,\dots,b_j)$

**Computation of an entry**

2 $L[0,i] \leftarrow 0\ \forall 0 \le i \le m$, $L[j,0] \leftarrow 0\ \forall 0 \le j \le n$. Computation of $L[i,j]$ otherwise via $L[i,j] = \max(L[i-1,j-1] + \delta_{ij}, L[i,j-1], L[i-1,j])$.

## [Dynamic Programming algorithm LCS]

3 **Computation order**

Rows increasing and within columns increasing (or the other way round).

4 **Reconstruct solution?**

Start with $j = m$, $i = n$. If $a_i = b_j$ then output $a_i$ and continue with $(j, i) \leftarrow (j - 1, i - 1)$; otherwise, if $L[i, j] = L[i, j - 1]$ continue with $j \leftarrow j - 1$ otherwise, if $L[i, j] = L[i - 1, j]$ continue with $i \leftarrow i - 1$ . Terminate for $i = 0$ or $j = 0$.

## [Analysis LCS]

- Number table entries: $(m+1) \cdot (n+1)$.
- Constant number of assignments and comparisons each. Number steps: $\mathcal{O}(mn)$
- Determination of solition: decrease $i$ or $j$. Maximally $\mathcal{O}(n+m)$ steps.

Runtime overal:

$$\mathcal{O}(mn).$$

## Minimal Editing Distance

Editing distance of two sequences $A_n = (a_1, \ldots, a_m)$, $B_m = (b_1, \ldots, b_m)$.

**Editing operations**:

- Insertion of a character
- Deletion of a character
- Replacement of a character

Question: how many editing operations at least required in order to transform string $A$ into string $B$.

*TIGER ZIGER ZIEGER ZIEGE*

## Minimal Editing Distance

Wanted: cheapest character-wise transformation $A_n \to B_m$ with costs

| operation | Levenshtein | LCS[40] | general |
|---|---|---|---|
| Insert $c$ | $1$ | $1$ | ins$(c)$ |
| Delete $c$ | $1$ | $1$ | del$(c)$ |
| Replace $c \to c'$ | $\mathbb{1}(c \neq c')$ | $\infty \cdot \mathbb{1}(c \neq c')$ | repl$(c, c')$ |

Beispiel

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T | I | G | E | R |  | T | I | _ | G | E | R |  | T$\to$Z | +E | -R |
| Z | I | E | G | E |  | Z | I | E | G | E | _ |  | Z$\to$T | -E | +R |

[40]Longest common subsequence – A special case of an editing problem

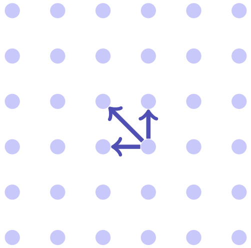
## DP

0. $E(n,m)$ = mimimum number edit operations (ED cost) $a_{1\dots n} \to b_{1\dots m}$
1. Subproblems $E(i,j)$ = ED von $a_{1\dots i}$. $b_{1\dots j}$. #SP $= n \cdot m$
2. Guess Costs$\Theta(1)$
   - $a_{1..i} \to a_{1\dots i-1}$ (delete)
   - $a_{1..i} \to a_{1\dots i} b_j$ (insert)
   - $a_{1..i} \to a_{1\dots i_1} b_j$ (replace)
3. Rekursion

$$E(i,j) = \min \begin{cases} \text{del}(a_i) + E(i-1,j), \\ \text{ins}(b_j) + E(i,j-1), \\ \text{repl}(a_i,b_j) + E(i-1,j-1) \end{cases}$$

# DP

4. Dependencies



$\Rightarrow$ Computation from left top to bottom right. Row- or column-wise.

5. Solution in $E(n, m)$

## Example (Levenshtein Distance)

$$E[i,j] \leftarrow \min\left\{E[i-1,j]+1, E[i,j-1]+1, E[i-1,j-1]+\mathbb{1}(a_i \neq b_j)\right\}$$

|   | ∅ | Z | I | E | G | E |
|---|---|---|---|---|---|---|
| ∅ | 0 | 1 | 2 | 3 | 4 | 5 |
| T | 1 | 1 | 2 | 3 | 4 | 5 |
| I | 2 | 2 | 1 | 2 | 3 | 4 |
| G | 3 | 3 | 2 | 2 | 2 | 3 |
| E | 4 | 4 | 3 | 2 | 3 | 2 |
| R | 5 | 5 | 4 | 3 | 3 | 3 |

Editing steps: from bottom right to top left, following the recursion.
Bottom-Up description of the algorithm: exercise

## Bottom-Up DP algorithm ED]

**Dimension of the table? Semantics?**

1 Table $E[0,\ldots,m][0,\ldots,n]$. $E[i,j]$: minimal edit distance of the strings $(a_1,\ldots,a_i)$ and $(b_1,\ldots,b_j)$

**Computation of an entry**

2 $E[0,i] \leftarrow i \; \forall 0 \leq i \leq m$, $E[j,0] \leftarrow i \; \forall 0 \leq j \leq n$. Computation of $E[i,j]$ otherwise via $E[i,j] =$ $\min\{\mathsf{del}(a_i)+E(i-1,j), \mathsf{ins}(b_j)+E(i,j-1), \mathsf{repl}(a_i,b_j)+E(i-1,j-1)\}$

## Bottom-Up DP algorithm ED

3 **Computation order**

Rows increasing and within columns increasing (or the other way round).

4 **Reconstruct solution?**

Start with $j = m$, $i = n$. If $E[i, j] = \text{repl}(a_i, b_j) + E(i-1, j-1)$ then output $a_i \rightarrow b_j$ and continue with $(j, i) \leftarrow (j-1, i-1)$; otherwise, if $E[i, j] = \text{del}(a_i) + E(i-1, j)$ output $\text{del}(a_i)$ and continue with $j \leftarrow j-1$ otherwise, if $E[i, j] = \text{ins}(b_j) + E(i, j-1)$, continue with $i \leftarrow i-1$ . Terminate for $i = 0$ and $j = 0$.

## Matrix-Chain-Multiplication

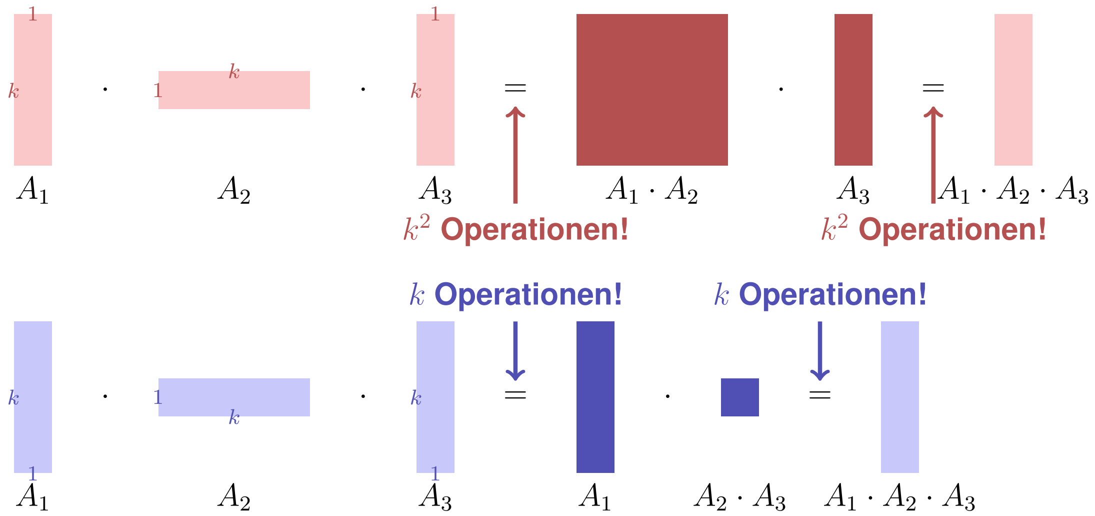
Task: Computation of the product $A_1 \cdot A_2 \cdot ... \cdot A_n$ of matrices $A_1$, ..., $A_n$.

Matrix multiplication is associative, i.e. the order of evalution can be chosen arbitrarily

Goal: efficient computation of the product.

Assumption: multiplicaiton of an $(r \times s)$-matrix with an $(s \times u)$-matrix provides costs $r \cdot s \cdot u$.

## Does it matter?



1
$k$
$k$
1
1
$k$
$A_1 \cdot A_2 \cdot A_3 = A_1 \cdot A_2 \cdot A_3 = A_1 \cdot A_2 \cdot A_3$
$k^2$ Operationen!
$k^2$ Operationen!
$k$ Operationen!
$k$ Operationen!
$k$
1
1
$k$
$k$
1
$A_1 \cdot A_2 \cdot A_3 = A_1 \cdot A_2 \cdot A_3 = A_1 \cdot A_2 \cdot A_3$

## Recursion

- Assume that the best possible computation of $(A_1 \cdot A_2 \cdots A_i)$ and $(A_{i+1} \cdot A_{i+2} \cdots A_n)$ is known for each $i$.
- Compute best $i$, done.

$n \times n$-table $M$. entry $M[p,q]$ provides costs of the best possible bracketing $(A_p \cdot A_{p+1} \cdots A_q)$.

$$M[p,q] \leftarrow \min_{p \le i < q} \left( M[p,i] + M[i+1,q] + \text{costs of the last multiplication} \right)$$

## Computation of the DP-table

- Base cases $M[p,p] \leftarrow 0$ for all $1 \leq p \leq n$.
- Computation of $M[p,q]$ depends on $M[i,j]$ with $p \leq i \leq j \leq q$, $(i,j) \neq (p,q)$.
  In particular $M[p,q]$ depends at most from entries $M[i,j]$ with $i-j < q-p$.
  Consequence: fill the table from the diagonal.

## Analysis

DP-table has $n^2$ entries. Computation of an entry requires considering up to $n-1$ other entries.

Overal runtime $\mathcal{O}(n^3)$.

Readout the order from $M$: exercise!

## Digression: matrix multiplication

Consider the mutliplicaiton of two $n \times n$ matrices.

Let

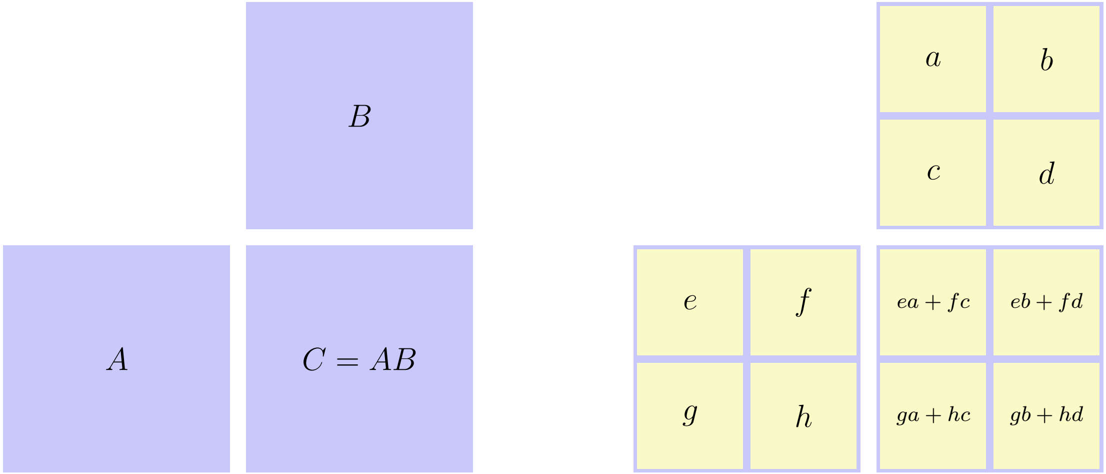
$$A = (a_{ij})_{1 \le i,j \le n}, B = (b_{ij})_{1 \le i,j \le n}, C = (c_{ij})_{1 \le i,j \le n},$$
$$C = A \cdot B$$

then

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

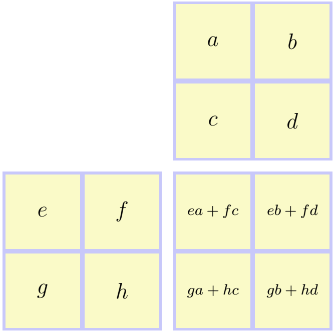
Naive algorithm requires $\Theta(n^3)$ elementary multiplications.

## Divide and Conquer



## Divide and Conquer

- Assumption $n = 2^k$.
- Number of elementary multiplications: $M(n) = 8M(n/2)$, $M(1) = 1$.
- yields $M(n) = 8^{\log_2 n} = n^{\log_2 8} = n^3$. No advantage ☹



## Strassen's Matrix Multiplication

- Nontrivial observation by Strassen (1969):

  It suffices to compute the seven products

  $A = (e+h)\cdot(a+d)$, $B = (g+h)\cdot a$,

  $C = e\cdot(b-d)$, $D = h\cdot(c-a)$, $E = (e+f)\cdot d$,

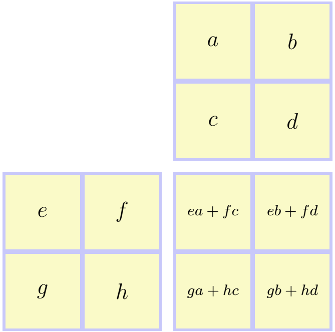
  $F = (g-e)\cdot(a+b)$, $G = (f-h)\cdot(c+d)$. Denn:

  $ea + fc = A + D - E + G$, $eb + fd = C + E$,

  $ga + hc = B + D$, $gb + hd = A - B + C + F$.

- This yields $M'(n) = 7M(n/2)$, $M'(1) = 1$. Thus $M'(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}$.

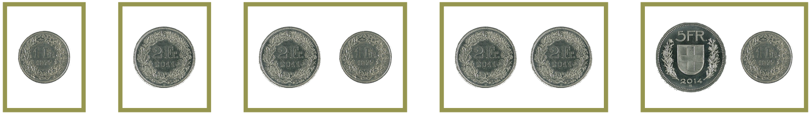
- Fastest currently known algorithm: $\mathcal{O}(n^{2.37})$



# 20. Dynamic Programming II

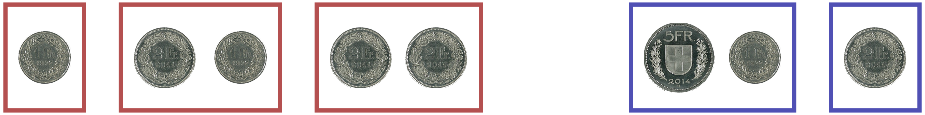
Subset sum problem, knapsack problem, greedy algorithm vs dynamic programming [Ottman/Widmayer, Kap. 7.2, 7.3, 5.7, Cormen et al, Kap. 15,35.5]

## Task



Partition the set of the “item” above into two set such that both sets have the same value.

A solution:



## Subset Sum Problem

Consider $n \in \mathbb{N}$ numbers $a_1, \ldots, a_n \in \mathbb{N}$.

Goal: decide if a selection $I \subseteq \{1, \ldots, n\}$ exists such that

$$\sum_{i \in I} a_i = \sum_{i \in \{1,\ldots,n\} \setminus I} a_i.$$

## Naive Algorithm

Check for each bit vector $b = (b_1, \ldots, b_n) \in \{0, 1\}^n$, if

$$\sum_{i=1}^{n} b_i a_i \overset{?}{=} \sum_{i=1}^{n} (1 - b_i) a_i$$

Worst case: $n$ steps for each of the $2^n$ bit vectors $b$. Number of steps: $\mathcal{O}(n \cdot 2^n)$.

## Algorithm with Partition

- Partition the input into two equally sized parts $a_1, \ldots, a_{n/2}$ and $a_{n/2+1}, \ldots, a_n$.
- Iterate over all subsets of the two parts and compute partial sum $S^k_1, \ldots, S^k_{2^{n/2}}$ ($k = 1, 2$).
- Sort the partial sums: $S^k_1 \leq S^k_2 \leq \cdots \leq S^k_{2^{n/2}}$.
- Check if there are partial sums such that $S^1_i + S^2_j = \frac{1}{2}\sum_{i=1}^n a_i =: h$

  - Start with $i = 1, j = 2^{n/2}$.
  - If $S^1_i + S^2_j = h$ then finished
  - If $S^1_i + S^2_j > h$ then $j \leftarrow j - 1$
  - If $S^1_i + S^2_j < h$ then $i \leftarrow i + 1$

## Example

Set $\{1, 6, 2, 3, 4\}$ with value sum $16$ has $32$ subsets.

Partitioning into $\{1, 6\}$ , $\{2, 3, 4\}$ yields the following $12$ subsets with value sums:

| $\{1,6\}$ | | | | $\{2,3,4\}$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $\{\}$ | $\{1\}$ | $\{6\}$ | $\{1,6\}$ | $\{\}$ | $\{2\}$ | $\{3\}$ | $\{4\}$ | $\{2,3\}$ | $\{2,4\}$ | $\{3,4\}$ | $\{2,3,4\}$ |
| $0$ | $\boxed{1}$ | $6$ | $7$ | $0$ | $2$ | $3$ | $4$ | $5$ | $6$ | $\boxed{7}$ | $9$ |

$\Leftrightarrow$ One possible solution: $\{1, 3, 4\}$

## Analysis

- Generate partial sums for each part: $\mathcal{O}(2^{n/2} \cdot n)$.
- Each sorting: $\mathcal{O}(2^{n/2} \log(2^{n/2})) = \mathcal{O}(n 2^{n/2})$.
- Merge: $\mathcal{O}(2^{n/2})$

Overal running time

$$\mathcal{O}\left(n \cdot 2^{n/2}\right) = \mathcal{O}\left(n \left(\sqrt{2}\right)^n\right).$$

Substantial improvement over the naive method – but still exponential!

## Dynamic programming

**Task**: let $z = \frac{1}{2}\sum_{i=1}^{n} a_i$. Find a selection $I \subset \{1, \dots, n\}$, such that $\sum_{i \in I} a_i = z$.

**DP-table**: $[0, \dots, n] \times [0, \dots, z]$-table $T$ with boolean entries. $T[k, s]$ specifies if there is a selection $I_k \subset \{1, \dots, k\}$ such that $\sum_{i \in I_k} a_i = s$.

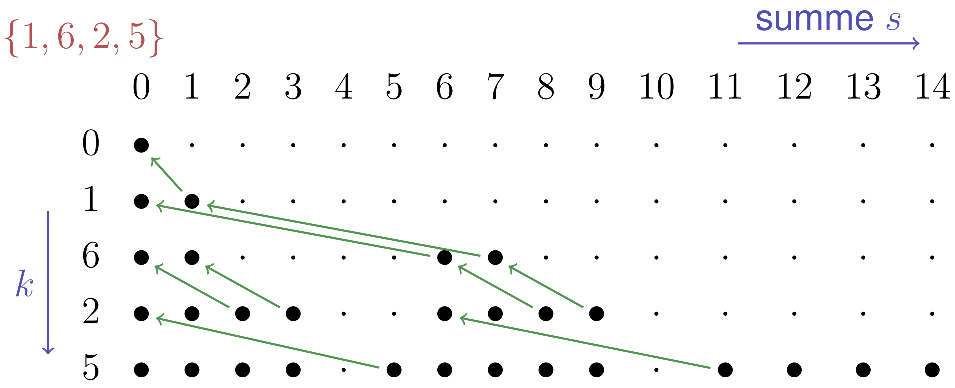
**Initialization**: $T[0, 0] =$ true. $T[0, s] =$ false for $s > 1$.

**Computation**:

$$T[k, s] \leftarrow \begin{cases} T[k-1, s] & \text{if } s < a_k \\ T[k-1, s] \vee T[k-1, s-a_k] & \text{if } s \geq a_k \end{cases}$$

for increasing $k$ and then within $k$ increasing $s$.

# Example



Determination of the solution: if $T[k, s] = T[k-1, s]$ then $a_k$ unused and continue with $T[k-1, s]$ , otherwise $a_k$ used and continue with $T[k-1, s-a_k]$ .

## That is mysterious

The algorithm requires a number of $\mathcal{O}(n \cdot z)$ fundamental operations.

What is going on now? Does the algorithm suddenly have polynomial running time?

## Explained

The algorithm does not necessarily provide a polynomial run time. $z$ is an *number* and not a *quantity*!

Input length of the algorithm $\cong$ number bits to *reasonably* represent the data. With the number $z$ this would be $\zeta = \log z$.

Consequently the algorithm requires $\mathcal{O}(n \cdot 2^{\zeta})$ fundamental operations and has a run time exponential in $\zeta$.

If, however, $z$ is polynomial in $n$ then the algorithm has polynomial run time in $n$. This is called *pseudo-polynomial*.

## NP

It is known that the subset-sum algorithm belongs to the class of *NP*-complete problems (and is thus *NP-hard*).

*P*: Set of all problems that can be solved in polynomial time.

*NP*: Set of all problems that can be solved Nondeterministically in Polynomial time.

Implications:

- NP contains P.
- Problems can be *verified* in polynomial time.
- Under the not (yet?) proven assumption[41] that NP $\neq$ P, there is *no algorithm with polynomial run time* for the problem considered above.

[41] The most important unsolved question of theoretical computer science.

## The knapsack problem

We pack our suitcase with ...

- toothbrush
- dumbell set
- coffee machine
- uh oh – too heavy.

- Toothbrush
- Air balloon
- Pocket knife
- identity card
- dumbell set
- Uh oh – too heavy.

- toothbrush
- coffe machine
- pocket knife
- identity card
- Uh oh – too heavy.

Aim to take as much as possible with us. But some things are more valuable than others!

## Knapsack problem

Given:

- set of $n \in \mathbb{N}$ items $\{1, \dots, n\}$.
- Each item $i$ has value $v_i \in \mathbb{N}$ and weight $w_i \in \mathbb{N}$.
- Maximum weight $W \in \mathbb{N}$.
- Input is denoted as $E = (v_i, w_i)_{i=1,\dots,n}$.

Wanted:

a selection $I \subseteq \{1, \dots, n\}$ that maximises $\sum_{i \in I} v_i$ under $\sum_{i \in I} w_i \leq W$.

## Greedy heuristics

Sort the items decreasingly by value per weight $v_i/w_i$: Permutation $p$ with $v_{p_i}/w_{p_i} \geq v_{p_{i+1}}/w_{p_{i+1}}$

Add items in this order ($I \leftarrow I \cup \{p_i\}$), if the maximum weight is not exceeded.

That is fast: $\Theta(n \log n)$ for sorting and $\Theta(n)$ for the selection. But is it good?

## Counterexample

$$\begin{array}{lll} v_1 = 1 & w_1 = 1 & v_1/w_1 = 1 \\ v_2 = W-1 & w_2 = W & v_2/w_2 = \frac{W-1}{W} \end{array}$$

Greed algorithm chooses $\{v_1\}$ with value $1$.
Best selection: $\{v_2\}$ with value $W-1$ and weight $W$.

Greedy heuristics can be arbitrarily bad.

## Dynamic Programming

Partition the maximum weight.

Three dimensional table $m[i, w, v]$ ("doable") of boolean values.

$m[i, w, v] =$ true if and only if

- A selection of the first $i$ parts exists ($0 \leq i \leq n$)
- with overal weight $w$ ($0 \leq w \leq W$) and
- a value of at least $v$ ($0 \leq v \leq \sum_{i=1}^{n} v_i$) .

## Computation of the DP table

Initially

- $m[i, w, 0] \leftarrow$ true für alle $i \geq 0$ und alle $w \geq 0$.
- $m[0, w, v] \leftarrow$ false für alle $w \geq 0$ und alle $v > 0$.

Computation

$$m[i, w, v] \leftarrow \begin{cases} m[i-1, w, v] \vee m[i-1, w-w_i, v-v_i] & \text{if } w \geq w_i \text{ und } v \geq v_i \\ m[i-1, w, v] & \text{otherwise.} \end{cases}$$

increasing in $i$ and for each $i$ increasing in $w$ and for fixed $i$ and $w$ increasing by $v$.

Solution: largest $v$, such that $m[i, w, v] =$ true for some $i$ and $w$.

## Observation

The definition of the problem obviously implies that

- for $m[i,w,v] =$ true it holds:
  $m[i',w,v] =$ true $\forall i' \geq i$ ,
  $m[i,w',v] =$ true $\forall w' \geq w$ ,
  $m[i,w,v'] =$ true $\forall v' \leq v$.
- fpr $m[i,w,v] =$ false it holds:
  $m[i',w,v] =$ false $\forall i' \leq i$ ,
  $m[i,w',v] =$ false $\forall w' \leq w$ ,
  $m[i,w,v'] =$ false $\forall v' \geq v$.

This strongly suggests that we do not need a 3d table!

## 2d DP table

Table entry $t[i, w]$ contains, instead of boolean values, the largest $v$, that can be achieved[42] with

- items $1, \dots, i$ $(0 \le i \le n)$
- at maximum weight $w$ $(0 \le w \le W)$.

[42] We could have followed a similar idea in order to reduce the size of the sparse table.

## Computation

Initially

- $t[0, w] \leftarrow 0$ for all $w \geq 0$.

We compute

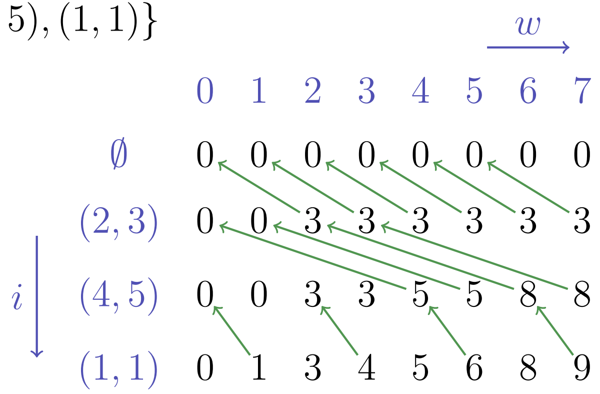
$$t[i, w] \leftarrow \begin{cases} t[i-1, w] & \text{if } w < w_i \\ \max\{t[i-1, w], t[i-1, w-w_i] + v_i\} & \text{otherwise.} \end{cases}$$

increasing by $i$ and for fixed $i$ increasing by $w$.

Solution is located in $t[n, w]$

## Example

$E = \{(2,3),(4,5),(1,1)\}$



$$
\begin{array}{c|cccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\emptyset & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
(2,3) & 0 & 0 & 3 & 3 & 3 & 3 & 3 & 3 \\
(4,5) & 0 & 0 & 3 & 3 & 5 & 5 & 8 & 8 \\
(1,1) & 0 & 1 & 3 & 4 & 5 & 6 & 8 & 9
\end{array}
$$

Reading out the solution: if $t[i,w] = t[i-1,w]$ then item $i$ unused and continue with $t[i-1,w]$ otherwise used and continue with $t[i-1,s-w_i]$ .

## Analysis

The two algorithms for the knapsack problem provide a run time in $\Theta(n \cdot W \cdot \sum_{i=1}^{n} v_i)$ (3d-table) and $\Theta(n \cdot W)$ (2d-table) and are thus both pseudo-polynomial, but they deliver the best possible result.

The greedy algorithm is very fast butmight deliver an arbitrarily bad result.

Now we consider a solution between the two extremes.

# 21. Dynamic Programming III

FPTAS [Ottman/Widmayer, Kap. 7.2, 7.3, Cormen et al, Kap. 15,35.5]

## Approximation

Let $\varepsilon \in (0,1)$ given. Let $I_{\text{opt}}$ an optimal selection.

No try to find a valid selection $I$ with

$$\sum_{i \in I} v_i \geq (1-\varepsilon) \sum_{i \in I_{\text{opt}}} v_i .$$

Sum of weights may not violate the weight limit.

## Different formulation of the algorithm

**Before**: weight limit $w \rightarrow$ maximal value $v$

**Reversed**: value $v \rightarrow$ minimal weight $w$

$\Rightarrow$ **alternative table** $g[i, v]$ provides the minimum weight with

- a selection of the first $i$ items ($0 \leq i \leq n$) that
- provide a value of exactly $v$ ($0 \leq v \leq \sum_{i=1}^{n} v_i$).

## Computation

### Initially

- $g[0,0] \leftarrow 0$
- $g[0,v] \leftarrow \infty$ (Value $v$ cannot be achieved with $0$ items.).

### Computation

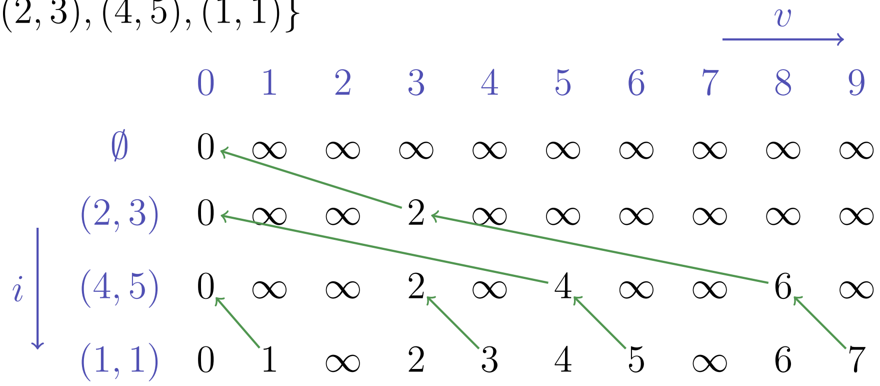
$$g[i,v] \leftarrow \begin{cases} g[i-1,v] & \text{falls } v < v_i \\ \min\{g[i-1,v], g[i-1,v-v_i]+w_i\} & \text{sonst.} \end{cases}$$

incrementally in $i$ and for fixed $i$ increasing in $v$.

Solution can be found at largest index $v$ with $g[n,v] \leq w$.

## Example

$E = \{(2,3),(4,5),(1,1)\}$



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $\emptyset$ | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| $(2,3)$ | 0 | $\infty$ | $\infty$ | 2 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| $(4,5)$ | 0 | $\infty$ | $\infty$ | 2 | $\infty$ | 4 | $\infty$ | $\infty$ | 6 | $\infty$ |
| $(1,1)$ | 0 | 1 | $\infty$ | 2 | 3 | 4 | 5 | $\infty$ | 6 | 7 |

Read out the solution: if $g[i,v] = g[i-1,v]$ then item $i$ unused and continue with $g[i-1,v]$ otherwise used and continue with $g[i-1,b-v_i]$ .

## The approximation trick

Pseuduopolynomial run time gets polynmial if the number of occuring values can be bounded by a polynom of the input length.

Let $K > 0$ be chosen *appropriately*. Replace values $v_i$ by “rounded values” $\tilde{v}_i = \lfloor v_i/K \rfloor$ delivering a new input $E' = (w_i, \tilde{v}_i)_{i=1\dots n}$.

Apply the algorithm on the input $E'$ with the same weight limit $W$.

## Idea

**Example** $K = 5$

Values

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots, 98, 99, 100$$
$$\rightarrow$$
$$0, 0, 0, 0, 1, 1, 1, 1, 1, 2, \ldots, 19, 19, 20$$

Obviously less different values

## Properties of the new algorithm

- Selection of items in $E'$ is also admissible in $E$. Weight remains unchanged!
- Run time of the algorithm is bounded by $\mathcal{O}(n^2 \cdot v_{\max}/K)$ ($v_{\max} := \max\{v_i | 1 \leq i \leq n\}$)

## How good is the approximation?

It holds that

$$v_i - K \leq K \cdot \left\lfloor \frac{v_i}{K} \right\rfloor = K \cdot \tilde{v}_i \leq v_i$$

Let $I'_{opt}$ be an optimal solution of $E'$. Then

$$\left(\sum_{i \in I_{\text{opt}}} v_i\right) - n \cdot K \overset{|I_{\text{opt}}| \leq n}{\leq} \sum_{i \in I_{\text{opt}}} (v_i - K) \leq \sum_{i \in I_{\text{opt}}} (K \cdot \tilde{v}_i) = K \sum_{i \in I_{\text{opt}}} \tilde{v}_i$$

$$\underset{I'_{\text{opt}} \text{optimal}}{\leq} K \sum_{i \in I'_{\text{opt}}} \tilde{v}_i = \sum_{i \in I'_{\text{opt}}} K \cdot \tilde{v}_i \leq \sum_{i \in I'_{\text{opt}}} v_i.$$

## Choice of $K$

Requirement:

$$\sum_{i \in I'} v_i \geq (1 - \varepsilon) \sum_{i \in I_{\text{opt}}} v_i.$$

Inequality from above:

$$\sum_{i \in I'_{\text{opt}}} v_i \geq \left( \sum_{i \in I_{\text{opt}}} v_i \right) - n \cdot K$$

thus: $K = \varepsilon \frac{\sum_{i \in I_{\text{opt}}} v_i}{n}$.

## Choice of $K$

Choose $K = \varepsilon \frac{\sum_{i \in I_{\text{opt}}} v_i}{n}$. The optimal sum is unknown. Therefore we choose $K' = \varepsilon \frac{v_{\max}}{n}$.[43]

It holds that $v_{\max} \leq \sum_{i \in I_{\text{opt}}} v_i$ and thus $K' \leq K$ and the approximation is even slightly better.

The run time of the algorithm is bounded by

$$\mathcal{O}(n^2 \cdot v_{\max}/K') = \mathcal{O}(n^2 \cdot v_{\max}/(\varepsilon \cdot v_{\max}/n)) = \mathcal{O}(n^3/\varepsilon).$$

[43] We can assume that items $i$ with $w_i > W$ have been removed in the first place.

## FPTAS

Such a family of algorithms is called an *approximation scheme*: the choice of $\varepsilon$ controls both running time and approximation quality.

The runtime $\mathcal{O}(n^3/\varepsilon)$ is a polynom in $n$ and in $\frac{1}{\varepsilon}$. The scheme is therefore also called a *FPTAS - Fully Polynomial Time Approximation Scheme*

# 21. Dynamic Programming III

Optimal Search Tree [Ottman/Widmayer, Kap. 5.7]

## Optimal binary Search Trees

Given: search probabilities $p_i$ for each key $k_i$ ($i = 1, \ldots, n$) and $q_i$ of each interval $d_i$ ($i = 0, \ldots, n$) between search keys of a binary search tree. $\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1$.

Wanted: optimal search tree $T$ with key depths $\operatorname{depth}(\cdot)$, that minimizes the expected search costs

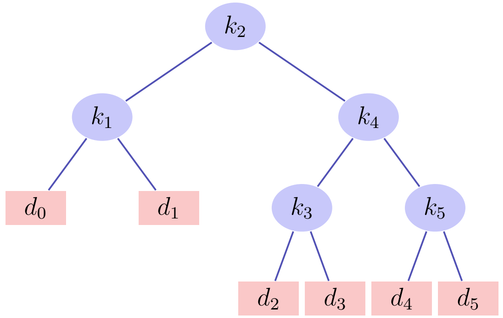
$$\begin{aligned} C(T) &= \sum_{i=1}^n p_i \cdot (\operatorname{depth}(k_i) + 1) + \sum_{i=0}^n q_i \cdot (\operatorname{depth}(d_i) + 1) \\ &= 1 + \sum_{i=1}^n p_i \cdot \operatorname{depth}(k_i) + \sum_{i=0}^n q_i \cdot \operatorname{depth}(d_i) \end{aligned}$$

## Example

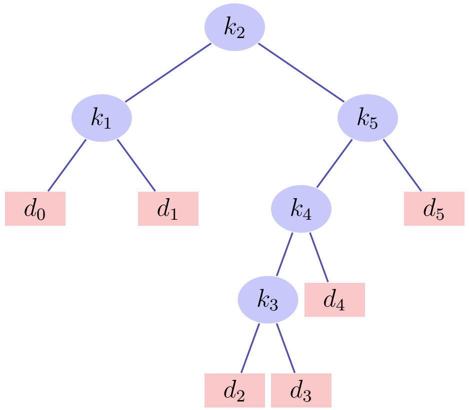
**Expected Frequencies**

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $p_i$ |  | 0.15 | 0.10 | 0.05 | 0.10 | 0.20 |
| $q_i$ | 0.05 | 0.10 | 0.05 | 0.05 | 0.05 | 0.10 |

## Example



Search tree with expected costs 2.8



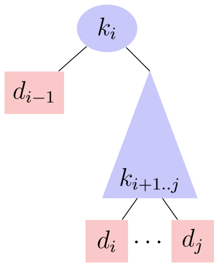
Search tree with expected costs 2.75

## Structure of a optimal binary search tree

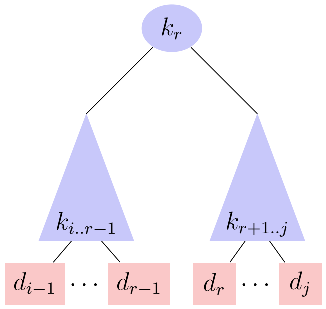
- Subtree with keys $k_i, \dots, k_j$ and intervals $d_{i-1}, \dots, d_j$ must be optimal for the respective sub-problem.[44]
- Consider all subtrees with roots $k_r$ and optimal subtrees for keys $k_i, \dots, k_{r-1}$ and $k_{r+1}, \dots, k_j$

[44] The usual argument: if it was not optimal, it could be replaced by a better solution improving the overal solution.

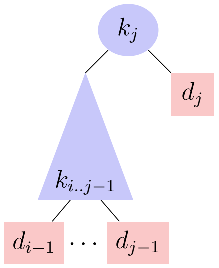
# Sub-trees for Searching



empty left subtree



non-empty left and right subtrees



empty right subtree

## Expected Search Costs

Let $\text{depth}_T(k)$ be the depth of a node $k$ in the sub-tree $T$. Let $k$ be the root of subtrees $T_r$ and $T_{L_r}$ and $T_{R_r}$ be the left and right sub-tree of $T_r$. Then

$$\text{depth}_T(k_i) = \text{depth}_{T_{L_r}}(k_i) + 1,\ (i < r)$$
$$\text{depth}_T(k_i) = \text{depth}_{T_{R_r}}(k_i) + 1,\ (i > r)$$

## Expected Search Costs

Let $e[i,j]$ be the costs of an optimal search tree with nodes $k_i, \ldots, k_j$.

Base case $e[i,i-1]$, expected costs $d_{i-1}$

Let $w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$.

If $k_r$ is the root of an optimal search tree with keys $k_i, \ldots, k_j$, then

$$e[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j))$$

with $w(i,j) = w(i,r-1) + p_r + w(r+1,j)$:

$$e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j).$$

## Dynamic Programming

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1, \\ \min_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + w[i,j]\} & \text{if } i \le j \end{cases}$$

## Computation

Tables $e[1 \ldots n+1, 0 \ldots n], w[1 \ldots n+1, 0 \ldots m], r[1 \ldots n, 1 \ldots n]$
Initially

- $e[i, i-1] \leftarrow q_{i-1}$, $w[i, i-1] \leftarrow q_{i-1}$ for all $1 \leq i \leq n+1$.

We compute

$$\begin{aligned} w[i,j] &= w[i,j-1] + p_j + q_j \\ e[i,j] &= \min_{i \leq r \leq j} \{e[i,r-1] + e[r+1,j] + w[i,j]\} \\ r[i,j] &= \arg\min_{i \leq r \leq j} \{e[i,r-1] + e[r+1,j] + w[i,j]\} \end{aligned}$$

for intervals $[i,j]$ with increasing lengths $l = 1, \ldots, n$, each for $i = 1, \ldots, n-l+1$. Result in $e[1,n]$, reconstruction via $r$. Runtime $\Theta(n^3)$.

## Example

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $p_i$ | | 0.15 | 0.10 | 0.05 | 0.10 | 0.20 |
| $q_i$ | 0.05 | 0.10 | 0.05 | 0.05 | 0.05 | 0.10 |

$w$

| $j$ \ $i$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 0 | 0.05 | | | | | |
| 1 | 0.30 | 0.10 | | | | |
| 2 | 0.45 | 0.25 | 0.05 | | | |
| 3 | 0.55 | 0.35 | 0.15 | 0.05 | | |
| 4 | 0.70 | 0.50 | 0.30 | 0.20 | 0.05 | |
| 5 | 1.00 | 0.80 | 0.60 | 0.50 | 0.35 | 0.10 |

$e$

| $j$ \ $i$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 0 | 0.05 | | | | | |
| 1 | 0.45 | 0.10 | | | | |
| 2 | 0.90 | 0.40 | 0.05 | | | |
| 3 | 1.25 | 0.70 | 0.25 | 0.05 | | |
| 4 | 1.75 | 1.20 | 0.60 | 0.30 | 0.05 | |
| 5 | 2.75 | 2.00 | 1.30 | 0.90 | 0.50 | 0.10 |

$r$

| $j$ \ $i$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 1 | | | | |
| 2 | 1 | 2 | | | |
| 3 | 2 | 2 | 3 | | |
| 4 | 2 | 2 | 4 | 4 | |
| 5 | 2 | 4 | 5 | 5 | 5 |

# 22. Greedy Algorithms

Fractional Knapsack Problem, Huffman Coding [Cormen et al, Kap. 16.1, 16.3]

## The Fractional Knapsack Problem

set of $n \in \mathbb{N}$ items $\{1, \dots, n\}$ Each item $i$ has value $v_i \in \mathbb{N}$ and weight $w_i \in \mathbb{N}$. The maximum weight is given as $W \in \mathbb{N}$. Input is denoted as $E = (v_i, w_i)_{i=1,\dots,n}$.

Wanted: Fractions $0 \leq q_i \leq 1$ ($1 \leq i \leq n$) that maximise the sum $\sum_{i=1}^n q_i \cdot v_i$ under $\sum_{i=1}^n q_i \cdot w_i \leq W$.

## Greedy heuristics

Sort the items decreasingly by value per weight $v_i/w_i$.

Assumption $v_i/w_i \geq v_{i+1}/w_{i+1}$

Let $j = \max\{0 \leq k \leq n : \sum_{i=1}^{k} w_i \leq W\}$. Set

- $q_i = 1$ for all $1 \leq i \leq j$.
- $q_{j+1} = \frac{W - \sum_{i=1}^{j} w_i}{w_{j+1}}$.
- $q_i = 0$ for all $i > j + 1$.

That is fast: $\Theta(n \log n)$ for sorting and $\Theta(n)$ for the computation of the $q_i$.

## Correctness

Assumption: optimal solution $(r_i)$ ($1 \leq i \leq n$).

The knapsack is full: $\sum_i r_i \cdot w_i = \sum_i q_i \cdot w_i = W$.

Consider $k$: smallest $i$ with $r_i \neq q_i$ Definition of greedy: $q_k > r_k$. Let $x = q_k - r_k > 0$.

Construct a new solution $(r'_i)$: $r'_i = r_i \forall i < k$. $r'_k = q_k$. Remove weight $\sum_{i=k+1}^{n} \delta_i = x \cdot w_k$ from items $k+1$ to $n$. This works because $\sum_{i=k}^{n} r_i \cdot w_i = \sum_{i=k}^{n} q_i \cdot w_i$.

## Correctness

$$\begin{aligned}
\sum_{i=k}^{n} r'_i v_i &= r_k v_k + x w_k \frac{v_k}{w_k} + \sum_{i=k+1}^{n} (r_i w_i - \delta_i) \frac{v_i}{w_i} \\
&\geq r_k v_k + x w_k \frac{v_k}{w_k} + \sum_{i=k+1}^{n} r_i w_i \frac{v_i}{w_i} - \delta_i \frac{v_k}{w_k} \\
&= r_k v_k + x w_k \frac{v_k}{w_k} - x w_k \frac{v_k}{w_k} + \sum_{i=k+1}^{n} r_i w_i \frac{v_i}{w_i} = \sum_{i=k}^{n} r_i v_i .
\end{aligned}$$

Thus $(r'_i)$ is also optimal. Iterative application of this idea generates the solution $(q_i)$.

## Huffman-Codes

Goal: memory-efficient saving of a sequence of characters using a binary code with code words..

**Example**

File consisting of 100.000 characters from the alphabet $\{a, \dots, f\}$.

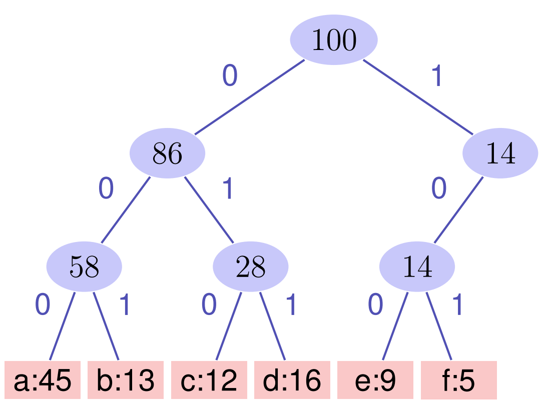
| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| Frequency (Thousands) | 45 | 13 | 12 | 16 | 9 | 5 |
| Code word with fix length | 000 | 001 | 010 | 011 | 100 | 101 |
| Code word variable length | 0 | 101 | 100 | 111 | 1101 | 1100 |

File size (code with fix length): $300.000$ bits.
File size (code with variable length): $224.000$ bits.

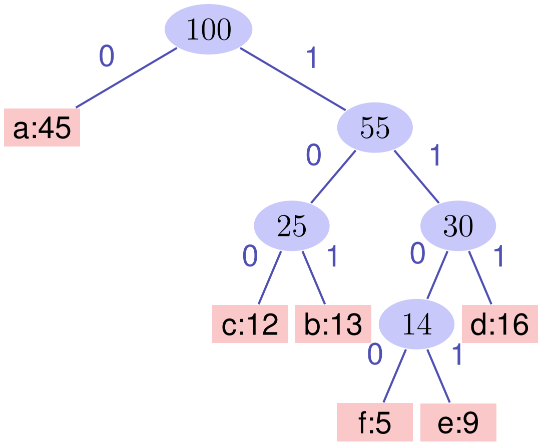
## Huffman-Codes

- Consider prefix-codes: no code word can start with a different codeword.
- Prefix codes can, compared with other codes, achieve the optimal *data compression* (without proof here).
- Encoding: concatenation of the code words without stop character (difference to morsing).
  $affe \to 0 \cdot 1100 \cdot 1100 \cdot 1101 \to 0110011001101$
- Decoding simple because prefixcode
  $0110011001101 \to 0 \cdot 1100 \cdot 1100 \cdot 1101 \to affe$

# Code trees



Code words with fixed length



Code words with variable length

## Properties of the Code Trees

- An optimal coding of a file is alway represented by a complete binary tree: every inner node has two children.
- Let $C$ be the set of all code words, $f(c)$ the frequency of a codeword $c$ and $d_T(c)$ the depth of a code word in tree $T$. Define the cost of a tree as

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c).$$

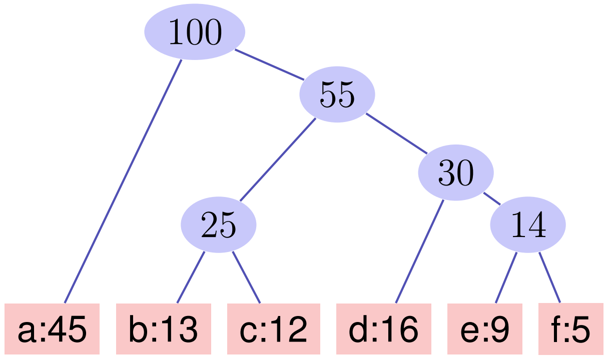
  (cost = number bits of the encoded file)

In the following a code tree is called optimal when it minimizes the costs.

## Algorithm Idea

Tree construction bottom up

- Start with the set $C$ of code words
- Replace iteriatively the two nodes with smallest frequency by a new parent node.



## Algorithm Huffman($C$)

**Input**: code words $c \in C$
**Output**: Root of an optimal code tree

```
$n \leftarrow |C|$
$Q \leftarrow C$
for $i = 1$ to $n - 1$ do
    allocate a new node $z$
    $z$.left $\leftarrow$ ExtractMin($Q$)          // extract word with minimal frequency.
    $z$.right $\leftarrow$ ExtractMin($Q$)
    $z$.freq $\leftarrow$ $z$.left.freq $+$ $z$.right.freq
    Insert($Q, z$)
return ExtractMin($Q$)
```

## Analyse

Use a heap: build Heap in $\mathcal{O}(n)$. Extract-Min in $O(\log n)$ for $n$ Elements. Yields a runtime of $O(n \log n)$.

## The greedy approach is correct

**Theorem**

*Let $x, y$ be two symbols with smallest frequencies in $C$ and let $T'(C')$ be an optimal code tree to the alphabet $C' = C - \{x, y\} + \{z\}$ with a new symbol $z$ with $f(z) = f(x) + f(y)$. Then the tree $T(C)$ that is constructed from $T'(C')$ by replacing the node $z$ by an inner node with children $x$ and $y$ is an optimal code tree for the alphabet $C$.*

## Proof

It holds that $f(x) \cdot d_T(x) + f(y) \cdot d_T(y) = (f(x) + f(y)) \cdot (d_{T'}(z) + 1) = f(z) \cdot d_{T'}(x) + f(x) + f(y)$. Thus $B(T') = B(T) - f(x) - f(y)$.

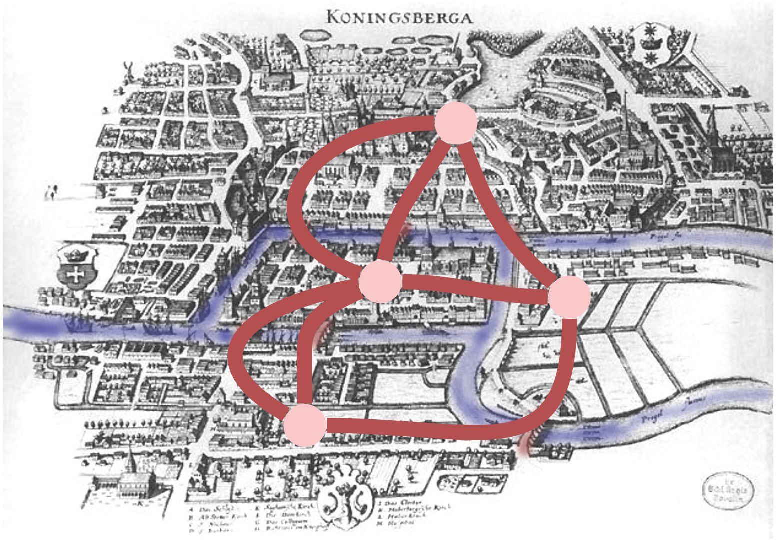
Assumption: $T$ is not optimal. Then there is an optimal tree $T''$ with $B(T'') < B(T)$. We assume that $x$ and $y$ are brothers in $T''$. Let $T'''$ be the tree where the inner node with children $x$ and $y$ is replaced by $z$. Then it holds that $B(T''') = B(T'') - f(x) - f(y) < B(T) - f(x) - f(y) = B(T')$. Contradiction to the optimality of $T'$.

The assumption that $x$ and $y$ are brothers in $T''$ can be justified because a swap of elements with smallest frequency to the lowest level of the tree can at most decrease the value of $B$.

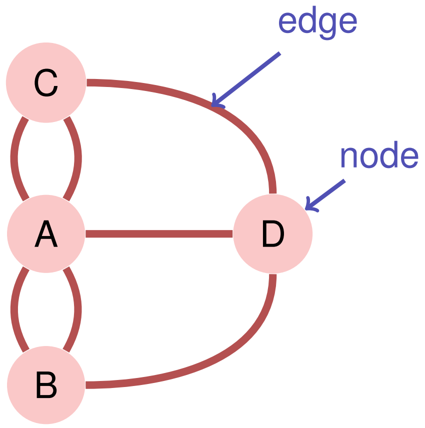
# 23. Graphs

Notation, Representation, Graph Traversal (DFS, BFS), Topological Sorting , Reflexive transitive closure, Connected components [Ottman/Widmayer, Kap. 9.1 - 9.4,Cormen et al, Kap. 22]

## Königsberg 1736



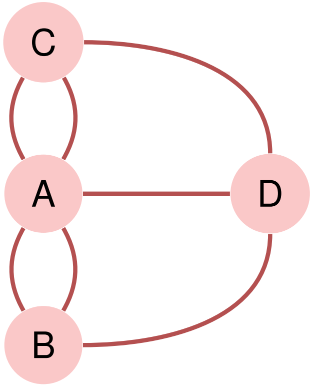
## [Multi]Graph



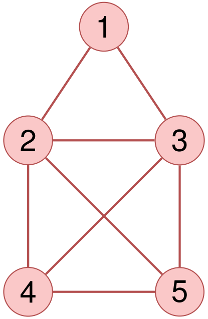
## Cycles

- Is there a cycle through the town (the graph) that uses each bridge (each edge) exactly once?
- Euler (1736): no.
- Such a *cycle* is called *Eulerian path*.
- Eulerian path $\Leftrightarrow$ each node provides an even number of edges (each node is of an *even degree*).

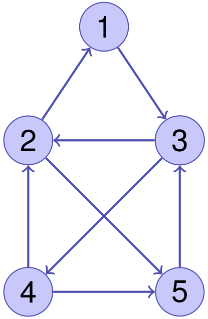
'$\Rightarrow$" is straightforward, "$\Leftarrow$" ist a bit more difficult but still elementary.



## Notation



undirected



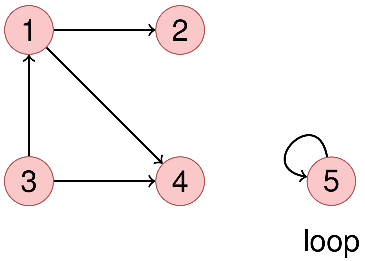
directed

$$\begin{aligned} V &= \{1,2,3,4,5\} \\ E &= \{\{1,2\},\{1,3\},\{2,3\},\{2,4\}, \\ &\quad \{2,5\},\{3,4\},\{3,5\},\{4,5\}\} \end{aligned}$$

$$\begin{aligned} V &= \{1,2,3,4,5\} \\ E &= \{(1,3),(2,1),(2,5),(3,2), \\ &\quad (3,4),(4,2),(4,5),(5,3)\} \end{aligned}$$

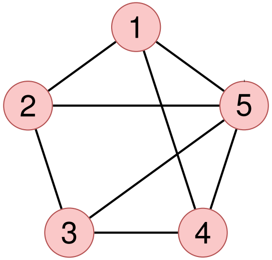
## Notation

A *directed graph* consists of a set $V = \{v_1, \dots, v_n\}$ of nodes (*Vertices) and a set* $E \subseteq V \times V$ *of* Edges. The same edges may not be contained more than once.



## Notation

An *undirected graph* consists of a set $V = \{v_1, \ldots, v_n\}$ of nodes a and a set $E \subseteq \{\{u, v\} | u, v \in V\}$ of edges. Edges may bot be contained more than once.[45]

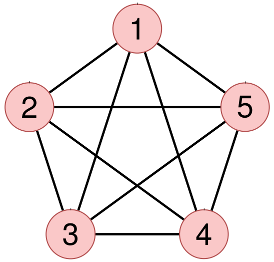


undirected graph

[45] As opposed to the introductory example – it is then called multi-graph.

## Notation

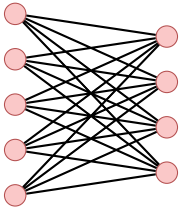
An undirected graph $G = (V, E)$ without loops where $E$ comprises all edges between pairwise different nodes is called *complete*.



a complete undirected graph

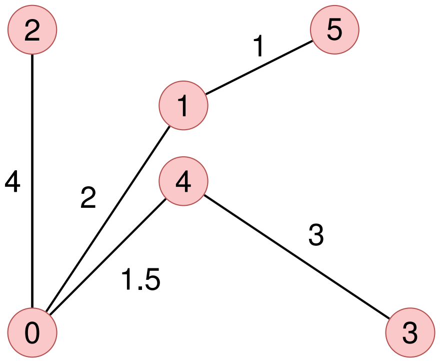
## Notation

A graph where $V$ can be partitioned into disjoint sets $U$ and $W$ such that each $e \in E$ provides a node in $U$ and a node in $W$ is called *bipartite*.



## Notation

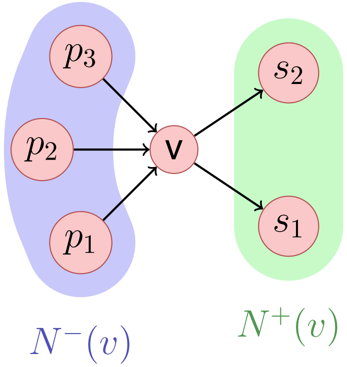
A *weighted graph* $G = (V, E, c)$ is a graph $G = (V, E)$ with an *edge weight function* $c : E \to \mathbb{R}$. $c(e)$ is called *weight* of the edge $e$.



## Notation

For directed graphs $G = (V, E)$

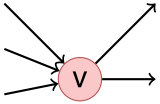
- $w \in V$ is called adjacent to $v \in V$, if $(v, w) \in E$
- *Predecessors* of $v \in V$: $N^-(v) := \{u \in V | (u, v) \in E\}$.
  *Successors*: $N^+(v) := \{u \in V | (v, u) \in E\}$

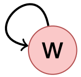


## Notation

For directed graphs $G = (V, E)$

- *In-Degree*: $\deg^-(v) = |N^-(v)|$,
  *Out-Degree*: $\deg^+(v) = |N^+(v)|$



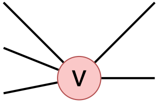


$\deg^-(v) = 3$, $\deg^+(v) = 2$ $\deg^-(w) = 1$, $\deg^+(w) = 1$

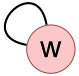
## Notation

For undirected graphs $G = (V, E)$:

- $w \in V$ is called *adjacent* to $v \in V$, if $\{v, w\} \in E$
- *Neighbourhood* of $v \in V$: $N(v) = \{w \in V | \{v, w\} \in E\}$
- *Degree* of $v$: $\deg(v) = |N(v)|$ with a special case for the loops: increase the degree by $2$.



$\deg(v) = 5$



$\deg(w) = 2$

## Relationship between node degrees and number of edges

For each graph $G = (V, E)$ it holds

1. $\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$, for $G$ directed
2. $\sum_{v \in V} \deg(v) = 2|E|$, for $G$ undirected.

## Paths

- *Path*: a sequence of nodes $\langle v_1, \dots, v_{k+1} \rangle$ such that for each $i \in \{1 \dots k\}$ there is an edge from $v_i$ to $v_{i+1}$ .
- *Length* of a path: number of contained edges $k$.
- *Weight* of a path (in weighted graphs): $\sum_{i=1}^{k} c((v_i, v_{i+1}))$ (bzw. $\sum_{i=1}^{k} c(\{v_i, v_{i+1}\})$)
- *Simple path*: path without repeating vertices

## Connectedness

- An undirected graph is called *connected*, if for eacheach pair $v, w \in V$ there is a connecting path.
- A directed graph is called *strongly connected*, if for each pair $v, w \in V$ there is a connecting path.
- A directed graph is called *weakly connected*, if the corresponding undirected graph is connected.

## Simple Observations

- generally: $0 \leq |E| \in \mathcal{O}(|V|^2)$
- connected graph: $|E| \in \Omega(|V|)$
- complete graph: $|E| = \frac{|V| \cdot (|V|-1)}{2}$ (undirected)
- Maximally $|E| = |V|^2$ (directed ),$|E| = \frac{|V| \cdot (|V|+1)}{2}$ (undirected)

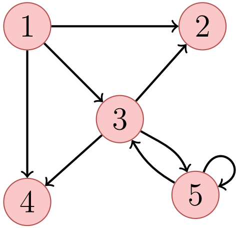
## Cycles

- *Cycle*: path $\langle v_1, \ldots, v_{k+1} \rangle$ with $v_1 = v_{k+1}$
- *Simple cycle*: Cycle with pairwise different $v_1, \ldots, v_k$, that does not use an edge more than once.
- *Acyclic*: graph without any cycles.

Conclusion: undirected graphs cannot contain cycles with length 2 (loops have length 1)

# Representation using a Matrix

Graph $G = (V, E)$ with nodes $v_1 \ldots, v_n$ stored as *adjacency matrix* $A_G = (a_{ij})_{1 \le i,j \le n}$ with entries from $\{0, 1\}$. $a_{ij} = 1$ if and only if edge from $v_i$ to $v_j$.

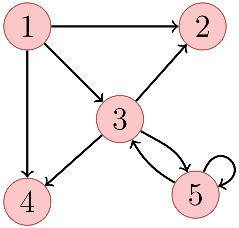


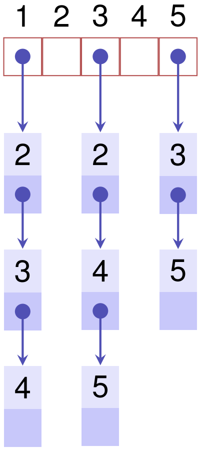
$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Memory consumption $\Theta(|V|^2)$. $A_G$ is symmetric, if $G$ undirected.

## Representation with a List

Many graphs $G = (V, E)$ with nodes $v_1, \dots, v_n$ provide much less than $n^2$ edges. Representation with *adjacency list*: Array $A[1], \dots, A[n]$, $A_i$ comprises a linked list of nodes in $N^+(v_i)$.



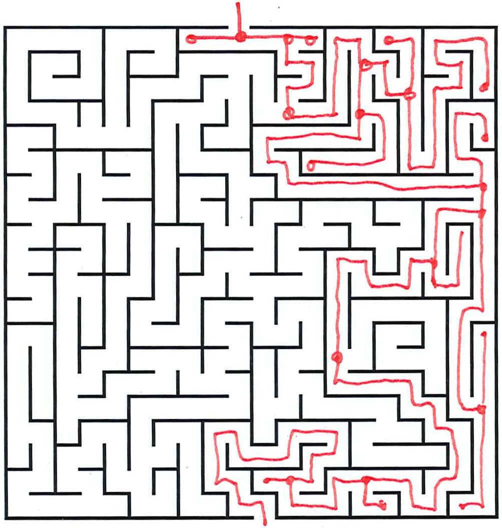


Memory Consumption $\Theta(|V| + |E|)$.

## Runtimes of simple Operations

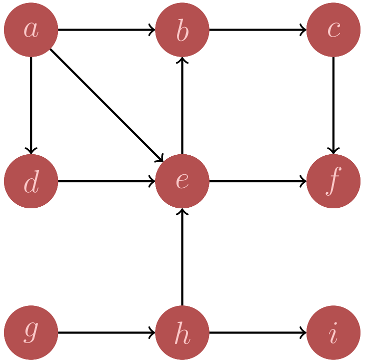
| **Operation** | Matrix | List |
|---|---|---|
| Find neighbours/successors of $v \in V$ | $\Theta(n)$ | $\Theta(\deg^+ v)$ |
| find $v \in V$ without neighbour/successor | $\Theta(n^2)$ | $\Theta(n)$ |
| $(u, v) \in E$ ? | $\Theta(1)$ | $\Theta(\deg^+ v)$ |
| Insert edge | $\Theta(1)$ | $\Theta(1)$ |
| Delete edge | $\Theta(1)$ | $\Theta(\deg^+ v)$ |

## Depth First Search



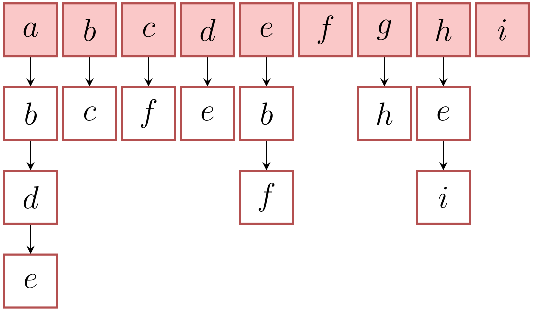
## Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.



Order $a, b, c, f, d, e, g, h, i$

Adjazenzliste



## Colors

Conceptual coloring of nodes

- **white:** node has not been discovered yet.
- **grey:** node has been discovered and is marked for traversal / being processed.
- **black:** node was discovered and entirely processed.

## Algorithm Depth First visit DFS-Visit($G, v$)

```
Input: graph $G = (V, E)$, Knoten $v$.

$v.color \leftarrow$ grey
foreach $w \in N^+(v)$ do
    if $w.color =$ white then
        DFS-Visit($G, w$)
$v.color \leftarrow$ black
```

Depth First Search starting from node $v$. Running time (without recursion): $\Theta(\deg^+ v)$

## Algorithm Depth First visit DFS-Visit($G$)

```
Input: graph $G = (V, E)$
foreach $v \in V$ do
    $v.color \leftarrow$ white
foreach $v \in V$ do
    if $v.color =$ white then
        DFS-Visit(G,v)
```

Depth First Search for all nodes of a graph. Running time: $\Theta(|V| + \sum_{v \in V}(\deg^+(v) + 1)) = \Theta(|V| + |E|)$.

# Iterative DFS-Visit($G, v$)

**Input:** graph $G = (V, E)$, $v \in V$ with $v.color$ = white

```
Stack $S \leftarrow \emptyset$
$v.color \leftarrow$ grey; $S$.push($v$)                // invariant: grey nodes always on stack
while $S \neq \emptyset$ do
    $w \leftarrow$ nextWhiteSuccessor($v$)                        // code: next slide
    if $w \neq$ null then
        $w.color \leftarrow$ grey; $S$.push($w$)
        $v \leftarrow w$                        // work on $w$. parent remains on the stack
    else
        $v.color \leftarrow$ black                  // no grey successors, $v$ becomes black
        if $S \neq \emptyset$ then
            $v \leftarrow S$.pop()                            // visit/revisit next node
            if $v.color$ = grey then  $S$.push($v$)
```

Memory Consumption Stack $\Theta(|V|)$

## nextWhiteSuccessor($v$)

**Input:** node $v \in V$
**Output:** Successor node $u$ of $v$ with $u.color =$ white, null otherwise

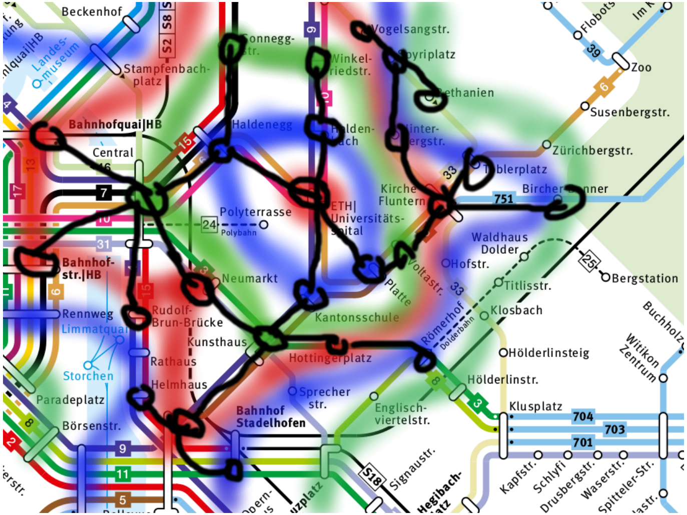
```
foreach $u \in N^+(v)$ do
    if $u.color =$ white then
        return $u$
return null
```

## Interpretation of the Colors

When traversing the graph, a tree (or Forest) is built. When nodes are discovered there are three cases

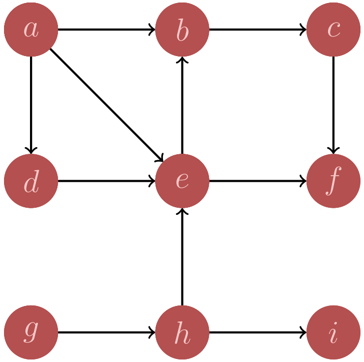
- White node: new tree edge
- Grey node: Zyklus (“back-egde”)
- Black node: forward- / cross edge

## Breadth First Search



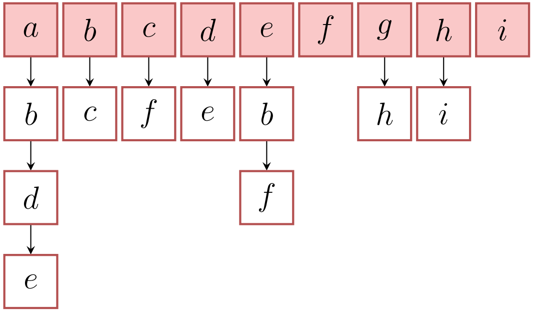
# Graph Traversal: Breadth First Search

Follow the path in breadth and only then descend into depth.



Order $a, b, d, e, c, f, g, h, i$

Adjazenzliste



## (Iterative) BFS-Visit($G, v$)

**Input:** graph $G = (V, E)$

```
Queue $Q \leftarrow \emptyset$
$v.color \leftarrow$ grey
enqueue($Q, v$)
**while** $Q \neq \emptyset$ **do**
    $w \leftarrow$ dequeue($Q$)
    **foreach** $c \in N^+(w)$ **do**
        **if** $c.color =$ white **then**
            $c.color \leftarrow$ grey
            enqueue($Q, c$)
    $w.color \leftarrow$ black
```

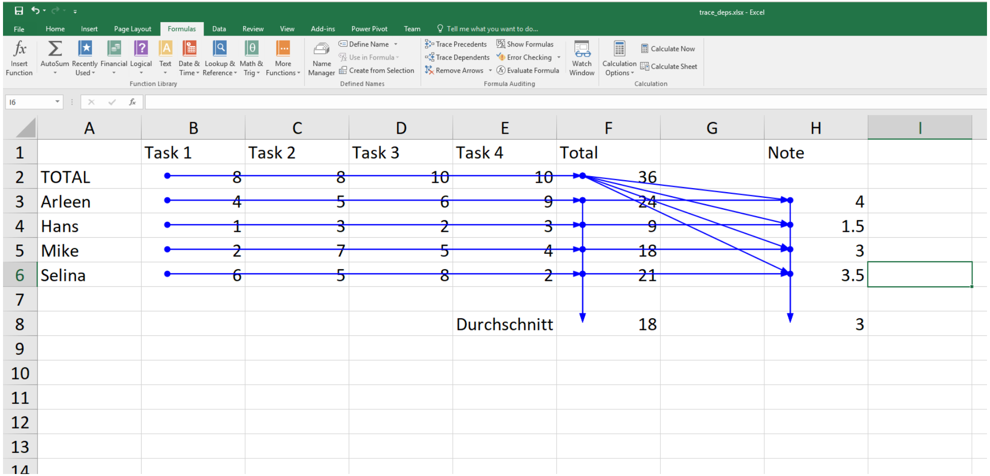
Algorithm requires extra space of $\mathcal{O}(|V|)$.

## Main program BFS-Visit($G$)

```
Input: graph $G = (V, E)$
foreach $v \in V$ do
    $v.color \leftarrow$ white
foreach $v \in V$ do
    if $v.color =$ white then
        BFS-Visit(G,v)
```

Breadth First Search for all nodes of a graph. Running time: $\Theta(|V| + |E|)$.

# Topological Sorting



|   | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| 1 |  | Task 1 | Task 2 | Task 3 | Task 4 | Total |  | Note |  |
| 2 | TOTAL | 8 | 8 | 10 | 10 | 36 |  |  |  |
| 3 | Arleen | 4 | 5 | 6 | 9 | 24 |  | 4 |  |
| 4 | Hans | 1 | 3 | 2 | 3 | 9 |  | 1.5 |  |
| 5 | Mike | 2 | 7 | 5 | 4 | 18 |  | 3 |  |
| 6 | Selina | 6 | 5 | 8 | 2 | 21 |  | 3.5 |  |
| 7 |  |  |  |  |  |  |  |  |  |
| 8 |  |  |  |  | Durchschnitt | 18 |  | 3 |  |
| 9 |  |  |  |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |  |  |  |
| 11 |  |  |  |  |  |  |  |  |  |
| 12 |  |  |  |  |  |  |  |  |  |
| 13 |  |  |  |  |  |  |  |  |  |
| 14 |  |  |  |  |  |  |  |  |  |

Evaluation Order?

## Topological Sorting

*Topological Sorting* of an acyclic directed graph $G = (V, E)$:

Bijective mapping

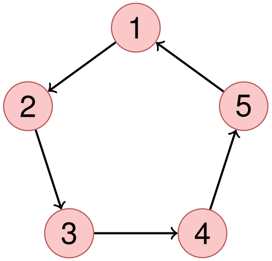
$$\text{ord} : V \to \{1, \ldots, |V|\}$$

such that

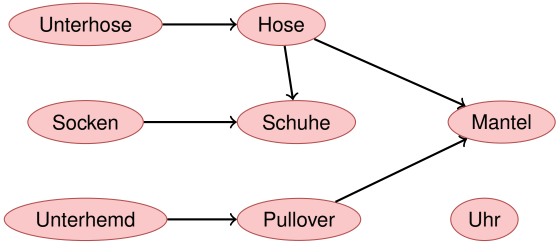
$$\text{ord}(v) < \text{ord}(w) \;\forall\; (v, w) \in E.$$

Identify $i$ with Element $v_i := \text{ord}^1(i)$. Topological sorting $\widehat{=}$ $\langle v_1, \ldots, v_{|V|} \rangle$.

## (Counter-)Examples



Cyclic graph: cannot be sorted topologically.



A possible toplogical sorting of the graph:
Unterhemd,Pullover,Unterhose,Uhr,Hose,Mantel,Socken,Schuhe

## Observation

**Theorem**

*A directed graph $G = (V, E)$ permits a topological sorting if and only if it is acyclic.*

Proof "$\Rightarrow$": If $G$ contains a cycle it cannot permit a topological sorting, because in a cycle $\langle v_{i_1}, \ldots, v_{i_m} \rangle$ it would hold that $v_{i_1} < \cdots < v_{i_m} < v_{i_1}$.

## Inductive Proof Opposite Direction

- Base case ($n = 1$): Graph with a single node without loop can be sorted topologically, set$\mathrm{ord}(v_1) = 1$.
- Hypothesis: Graph with $n$ nodes can be sorted topologically
- Step ($n \to n + 1$):
  1. $G$ contains a node $v_q$ with in-degree $\deg^-(v_q) = 0$. Otherwise iteratively follow edges backwards – after at most $n + 1$ iterations a node would be revisited. Contradiction to the cycle-freeness.
  2. Graph without node $v_q$ and without its edges can be topologically sorted by the hypothesis. Now use this sorting and set $\mathrm{ord}(v_i) \leftarrow \mathrm{ord}(v_i) + 1$ for all $i \neq q$ and set $\mathrm{ord}(v_q) \leftarrow 1$.

## Preliminary Sketch of an Algorithm

Graph $G = (V, E)$. $d \leftarrow 1$

1. Traverse backwards starting from any node until a node $v_q$ with in-degree $0$ is found.
2. If no node with in-degree $0$ found after $n$ stepsm, then the graph has a cycle.
3. Set $\mathrm{ord}(v_q) \leftarrow d$.
4. Remove $v_q$ and his edges from $G$.
5. If $V \neq \emptyset$ , then $d \leftarrow d + 1$, go to step $1$.

Worst case runtime: $\Theta(|V|^2)$.

## Improvement

Idea?

Compute the in-degree of all nodes in advance and traverse the nodes with in-degree 0 while correcting the in-degrees of following nodes.

# Algorithm Topological-Sort($G$)

**Input:** graph $G = (V, E)$.
**Output:** Topological sorting $\mathrm{ord}$

```
Stack $S \leftarrow \emptyset$
**foreach** $v \in V$ **do** $A[v] \leftarrow 0$
**foreach** $(v, w) \in E$ **do** $A[w] \leftarrow A[w] + 1$ // Compute in-degrees
**foreach** $v \in V$ with $A[v] = 0$ **do** push$(S, v)$ // Memorize nodes with in-degree 0
$i \leftarrow 1$
**while** $S \neq \emptyset$ **do**
    $v \leftarrow$ pop$(S)$; $\mathrm{ord}[v] \leftarrow i$; $i \leftarrow i + 1$ // Choose node with in-degree 0
    **foreach** $(v, w) \in E$ **do** // Decrease in-degree of successors
        $A[w] \leftarrow A[w] - 1$
        **if** $A[w] = 0$ **then** push$(S, w)$
**if** $i = |V| + 1$ **then return** $\mathrm{ord}$ **else return** “Cycle Detected”
```

## Algorithm Correctness

**Theorem**

*Let $G = (V, E)$ be a directed acyclic graph. Algorithm TopologicalSort($G$) computes a topological sorting $\mathrm{ord}$ for $G$ with runtime $\Theta(|V| + |E|)$.*

Proof: follows from previous theorem:

1. Decreasing the in-degree corresponds with node removal.
2. In the algorithm it holds for each node $v$ with $A[v] = 0$ that either the node has in-degree 0 or that previously all predecessors have been assigned a value $\mathrm{ord}[u] \leftarrow i$ and thus $\mathrm{ord}[v] > \mathrm{ord}[u]$ for all predecessors $u$ of $v$. Nodes are put to the stack only once.
3. Runtime: inspection of the algorithm (with some arguments like with graph traversal)

## Algorithm Correctness

**Theorem**

*Let $G = (V, E)$ be a directed graph containing a cycle. Algorithm TopologicalSort($G$) terminates within $\Theta(|V| + |E|)$ steps and detects a cycle.*

Proof: let $\langle v_{i_1}, \ldots, v_{i_k}\rangle$ be a cycle in $G$. In each step of the algorithm remains $A[v_{i_j}] \geq 1$ for all $j = 1, \ldots, k$. Thus $k$ nodes are never pushed on the stack und therefore at the end it holds that $i \leq V + 1 - k$.

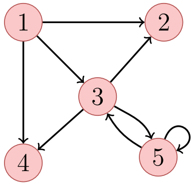
The runtime of the second part of the algorithm can become shorter. But the computation of the in-degree costs already $\Theta(|V| + |E|)$.

## Alternative: Algorithm DFS-Topsort($G, v$)

```
Input: graph $G = (V, E)$, node $v$, node list $L$.

if $v.color$ = grey then
    stop (Cycle)
if $v.color$ = black then
    return
$v.color \leftarrow$ grey
foreach $w \in N^+(v)$ do
    DFS-Topsort($G, w$)
$v.color \leftarrow$ black
Add $v$ to head of $L$
```

Call this algorithm for each node that has not yet been visited.
Asymptotic Running Time $\Theta(|V| + |E|)$.

## Adjacency Matrix Product



$$B := A_G^2 = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}^2 = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 2 \end{pmatrix}$$

## Interpretation

**Theorem**

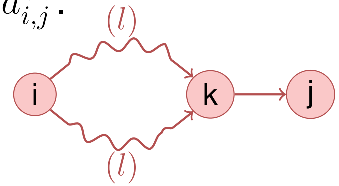
*Let $G = (V, E)$ be a graph and $k \in \mathbb{N}$. Then the element $a_{i,j}^{(k)}$ of the matrix $(a_{i,j}^{(k)})_{1\leq i,j\leq n} = (A_G)^k$ provides the number of paths with length $k$ from $v_i$ to $v_j$ .*

## Proof

By Induction.

**Base case:** straightforward for $k = 1$. $a_{i,j} = a^{(1)}_{i,j}$.
**Hypothesis:** claim is true for all $k \leq l$
**Step ($l \rightarrow l+1$):**

$$a^{(l+1)}_{i,j} = \sum_{k=1}^{n} a^{(l)}_{i,k} \cdot a_{k,j}$$



$a_{k,j} = 1$ iff egde $k$ to $j$, 0 otherwise. Sum counts the number paths of length $l$ from node $v_i$ to all nodes $v_k$ that provide a direct direction to node $v_j$, i.e. all paths with length $l + 1$.

## Example: Shortest Path

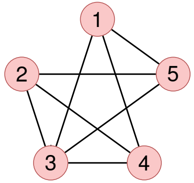
*Question:* is there a path from $i$ to $j$? How long is the shortest path?

*Answer:* exponentiate $A_G$ until for some $k < n$ it holds that $a_{i,j}^{(k)} > 0$. $k$ provides the path length of the shortest path. If $a_{i,j}^{(k)} = 0$ for all $1 \leq k < n$, then there is no path from $i$ to $j$.

## Example: Number triangles

*Question:* How many triangular path does an undirected graph contain?

*Answer:* Remove all cycles (diagonal entries). Compute $A_G^3$. $a_{ii}^{(3)}$ determines the number of paths of length $3$ that contain $i$. There are $6$ different permutations of a triangular path. Thus for the number of triangles: $\sum_{i=1}^{n} a_{ii}^{(3)}/6$.



$$\begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix}^3 = \begin{pmatrix} 4 & 4 & 8 & 8 & 8 \\ 4 & 4 & 8 & 8 & 8 \\ 8 & 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 4 & 4 \\ 8 & 8 & 8 & 4 & 4 \end{pmatrix}$$

$\Rightarrow 24/6 = 4$ Dreiecke.

## Relation

Given a finite set $V$

(Binary) **Relation** $R$ on $V$: Subset of the cartesian product $V \times V = \{(a,b) | a \in V, b \in V\}$

Relation $R \subseteq V \times V$ is called

- *reflexive*, if $(v,v) \in R$ for all $v \in V$
- *symmetric*, if $(v,w) \in R \Rightarrow (w,v) \in R$
- *transitive*, if $(v,x) \in R$, $(x,w) \in R \Rightarrow (v,w) \in R$

The (Reflexive) Transitive Closure $R^*$ of $R$ is the smallest extension $R \subseteq R^* \subseteq V \times V$ such that $R^*$ is reflexive and transitive.

## Graphs and Relations

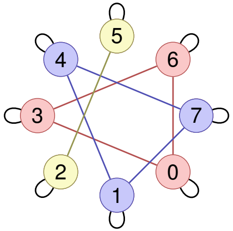
Graph $G = (V, E)$
adjacencies $A_G \widehat{=}$ Relation $E \subseteq V \times V$ over $V$

- *reflexive* $\Leftrightarrow a_{i,i} = 1$ for all $i = 1, \ldots, n$. (loops)
- *symmetric* $\Leftrightarrow a_{i,j} = a_{j,i}$ for all $i, j = 1, \ldots, n$ (undirected)
- *transitive* $\Leftrightarrow (u, v) \in E$, $(v, w) \in E \Rightarrow (u, w) \in E$. (reachability)

## Example: Equivalence Relation

Equivalence relation $\Leftrightarrow$ symmetric, transitive, reflexive relation $\Leftrightarrow$ collection of complete, undirected graphs where each element has a loop.

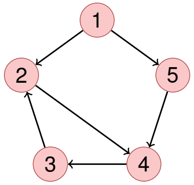
**Example:** Equivalence classes of the numbers $\{0, ..., 7\}$ modulo $3$



# Reflexive Transitive Closure

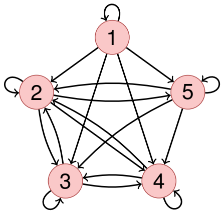
Reflexive transitive closure of $G$ $\Leftrightarrow$ *Reachability relation* $E^*$:
$(v, w) \in E^*$ iff $\exists$ path from node $v$ to $w$.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \qquad \Rightarrow \qquad \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$



$G = (V, E)$

$\Rightarrow$



$G^* = (V, E^*)$

## Computation of the Reflexive Transitive Closure

*Goal:* computation of $B = (b_{ij})_{1\leq i,j\leq n}$ with $b_{ij} = 1 \Leftrightarrow (v_i, v_j) \in E^*$

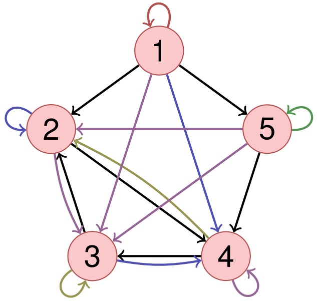
*Observation:* $a_{ij} = 1$ already implies $(v_i, v_j) \in E^*$.

First idea:

- Start with $B \leftarrow A$ and set $b_{ii} = 1$ for each $i$ (Reflexivity.).
- Iterate over $i, j, k$ and set $b_{ij} = 1$, if $b_{ik} = 1$ and $b_{kj} = 1$. Then all paths with lenght 1 and 2 taken into account.
- Repeated iteration $\Rightarrow$ all paths with length $1 \dots 4$ taken into account.
- $\lceil \log_2 n \rceil$ iterations required. $\Rightarrow$ running time $n^3 \lceil \log_2 n \rceil$

# Improvement: Algorithm of Warshall (1962)

Inductive procedure: all paths known over nodes from $\{v_i : i < k\}$. Add node $v_k$.



$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

## Algorithm TransitiveClosure($A_G$)

**Input:** Adjacency matrix $A_G = (a_{ij})_{i,j=1\dots n}$
**Output:** Reflexive transitive closure $B = (b_{ij})_{i,j=1\dots n}$ of $G$

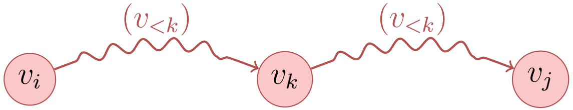
```
$B \leftarrow A_G$
for $k \leftarrow 1$ to $n$ do
    $a_{kk} \leftarrow 1$                                        // Reflexivity
    for $i \leftarrow 1$ to $n$ do
        for $j \leftarrow 1$ to $n$ do
            $b_{ij} \leftarrow \max\{b_{ij}, b_{ik} \cdot b_{kj}\}$          // All paths via $v_k$
return $B$
```

Runtime $\Theta(n^3)$.

# Correctness of the Algorithm (Induction)

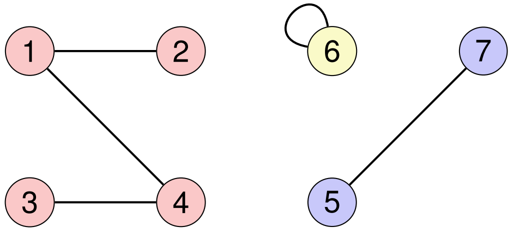
**Invariant ($k$)**: all paths via nodes with maximal index $< k$ considered.

- **Base case ($k = 1$)**: All directed paths (all edges) in $A_G$ considered.
- **Hypothesis**: invariant ($k$) fulfilled.
- **Step** ($k \to k + 1$): For each path from $v_i$ to $v_j$ via nodes with maximal index $k$: by the hypothesis $b_{ik} = 1$ and $b_{kj} = 1$. Therefore in the $k$-th iteration: $b_{ij} \leftarrow 1$.



# Connected Components

Connected components of an undirected graph $G$: equivalence classes of the reflexive, transitive closure of $G$. Connected component = subgraph $G' = (V', E')$, $E' = \{\{v,w\} \in E | v,w \in V'\}$ with
$\{\{v,w\} \in E | v \in V' \vee w \in V'\} = E = \{\{v,w\} \in E | v \in V' \wedge w \in V'\}$



Graph with connected components $\{1,2,3,4\}$, $\{5,7\}$, $\{6\}$.

## Computation of the Connected Components

- Computation of a partitioning of $V$ into pairwise disjoint subsets $V_1, \ldots, V_k$
- such that each $V_i$ contains the nodes of a connected component.
- Algorithm: depth-first search or breadth-first search. Upon each new start of DFSSearch($G, v$) or BFSSearch($G, v$) a new empty connected component is created and all nodes being traversed are added.

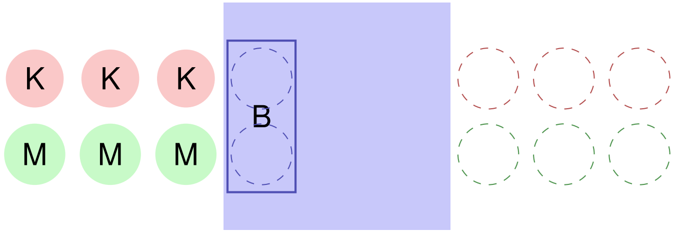
# 24. Shortest Paths

Motivation, Dijkstra’s algorithm on distance graphs, Bellman-Ford Algorithm, Floyd-Warshall Algorithm

[Ottman/Widmayer, Kap. 9.5 Cormen et al, Kap. 24.1-24.3, 25.2-25.3]

## River Crossing (Missionaries and Cannibals)

Problem: Three cannibals and three missionaries are standing at a river bank. The available boat can carry two people. At no time may at any place (banks or boat) be more cannibals than missionaries. How can the missionaries and cannibals cross the river as fast as possible? [46]

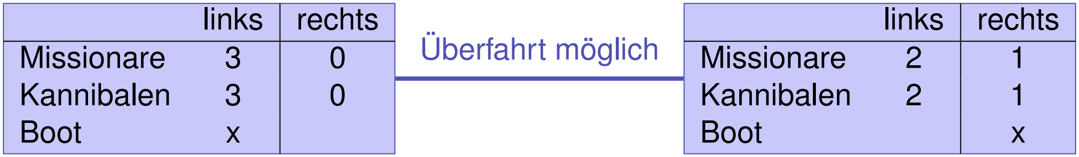


[46]There are slight variations of this problem. It is equivalent to the jealous husbands problem.

## Problem as Graph

Enumerate permitted configurations as nodes and connect them with an edge, when a crossing is allowed. The problem then becomes a shortest path problem.

Example



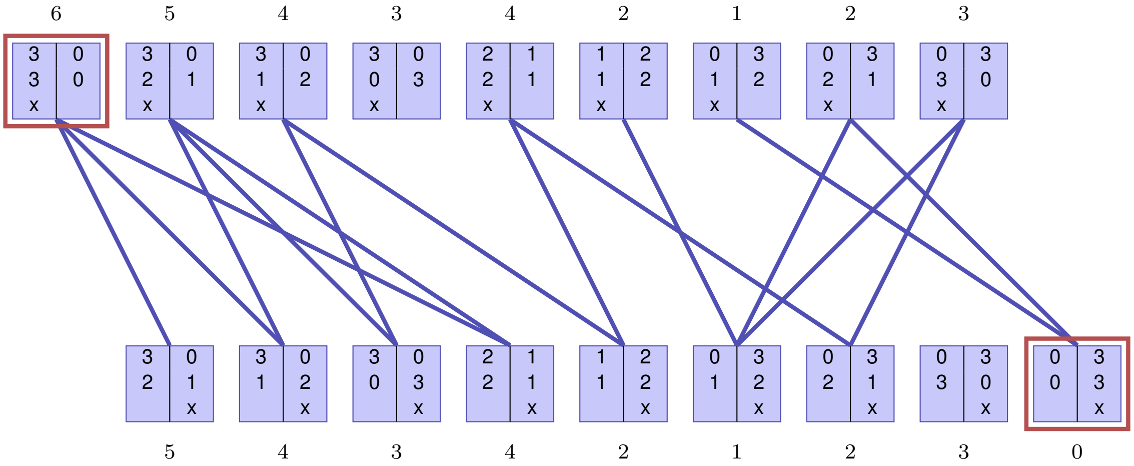
| | links | rechts |
|---|---|---|
| Missionare | 3 | 0 |
| Kannibalen | 3 | 0 |
| Boot | x | |

$6$ Personen am linken Ufer

| | links | rechts |
|---|---|---|
| Missionare | 2 | 1 |
| Kannibalen | 2 | 1 |
| Boot | | x |

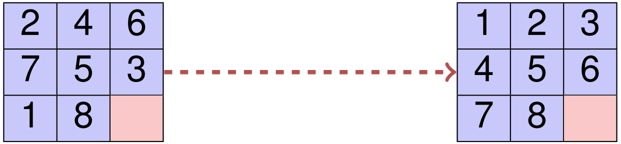
$4$ Personen am linken Ufer

## The whole problem as a graph

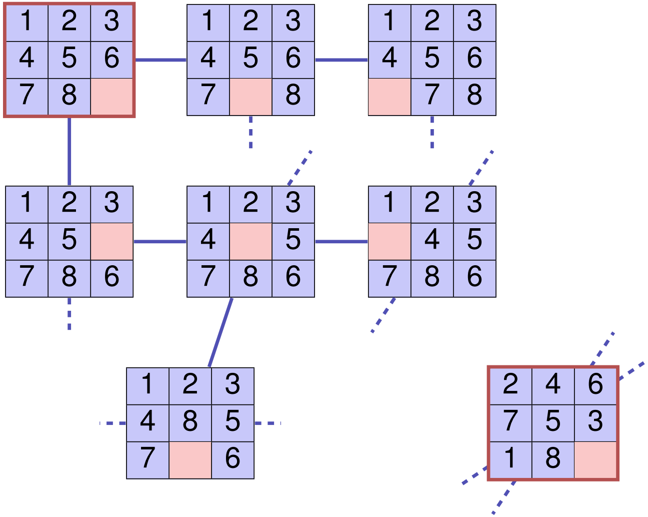


## Another Example: Mystic Square

Want to find the fastest solution for

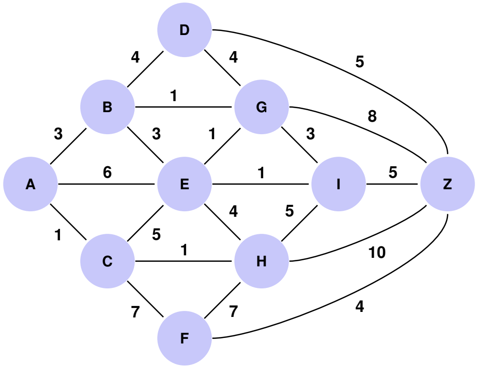


## Problem as Graph



# Route Finding

Provided cities A - Z and Distances between cities.

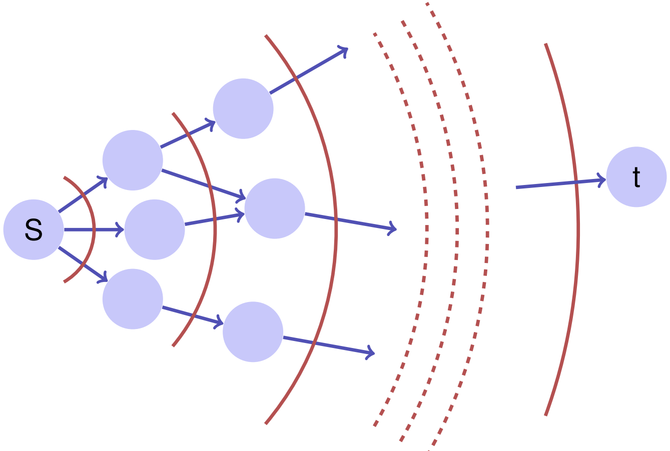


What is the shortest path from A to Z?

## Simplest Case

Constant edge weight $1$ (wlog)

Solution: Breadth First Search



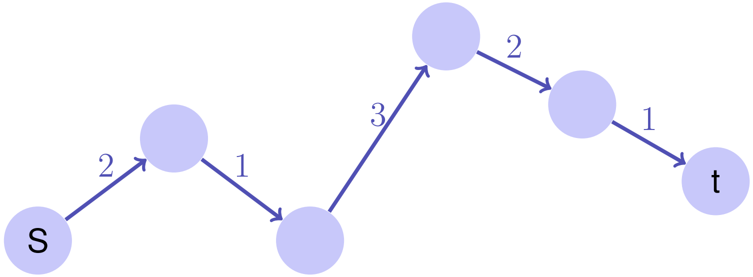
## Weighted Graphs

*Given:* $G = (V, E, c)$, $c : E \to \mathbb{R}$, $s, t \in V$.

*Wanted:* Length (weight) of a shortest path from $s$ to $t$.

*Path:* $p = \langle s = v_0, v_1, \ldots, v_k = t \rangle$, $(v_i, v_{i+1}) \in E$ $(0 \le i < k)$

*Weight:* $c(p) := \sum_{i=0}^{k-1} c((v_i, v_{i+1}))$.



Path with weight $9$

## Shortest Paths

**Notation**: we write

$$u \overset{p}{\rightsquigarrow} v \qquad \text{oder} \qquad p : u \rightsquigarrow v$$

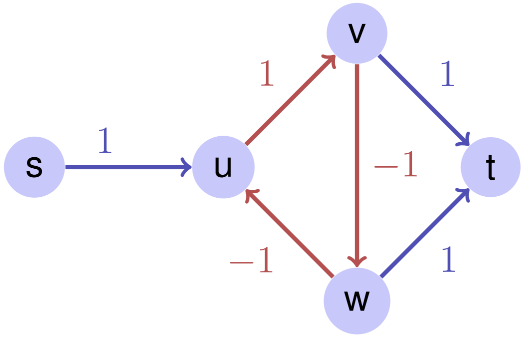
and mean a path $p$ from $u$ to $v$

**Notation**: $\delta(u, v)$ = weight of a shortest path from $u$ to $v$:

$$\delta(u, v) = \begin{cases} \infty & \text{no path from } u \text{ to } v \\ \min\{c(p) : u \overset{p}{\rightsquigarrow} v\} & \text{otherwise} \end{cases}$$

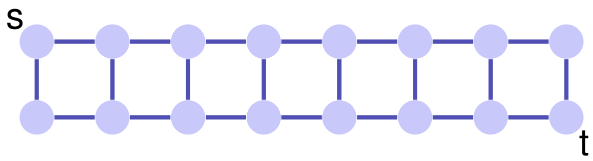
## Observations (1)

It may happen that a shortest paths does not exist: negative cycles can occur.



## Observations (2)

There can be exponentially many paths.



(at least $2^{|V|/2}$ paths from $s$ to $t$)

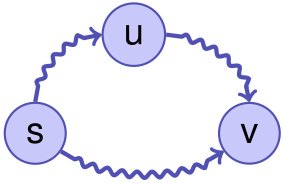
$\Rightarrow$ To try all paths is too inefficient

## Observations (3)

*Triangle Inequality*

For all $s, u, v \in V$:

$$\delta(s, v) \leq \delta(s, u) + \delta(u, v)$$

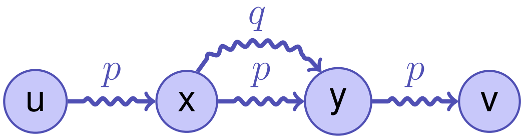


A shortest path from $s$ to $v$ cannot be longer than a shortest path from $s$ to $v$ that has to include $u$

## Observations (4)

*Optimal Substructure*

Sub-paths of shortest paths are shortest paths. Let $p = \langle v_0, \ldots, v_k \rangle$ be a shortest path from $v_0$ to $v_k$. Then each of the sub-paths $p_{ij} = \langle v_i, \ldots, v_j \rangle$ $(0 \le i < j \le k)$ is a shortest path from $v_i$ to $v_j$.



If not, then one of the sub-paths could be shortened which immediately leads to a contradiction.

## Observations (5)

Shortest paths do not contain cycles

1. Shortest path contains a negative cycle: there is no shortest path, contradiction
2. Path contains a positive cycle: removing the cycle from the path will reduce the weight. Contradiction.
3. Path contains a cycle with weight $0$: removing the cycle from the path will not change the weight. Remove the cycle (convention).

## Ingredients of an Algorithm

Wanted: shortest paths from a starting node $s$.

- Weight of the shortest path found so far

$$d_s : V \to \mathbb{R}$$

  *At the beginning:* $d_s[v] = \infty$ for all $v \in V$.
  *Goal:* $d_s[v] = \delta(s, v)$ for all $v \in V$.

- Predecessor of a node

$$\pi_s : V \to V$$

  Initially $\pi_s[v]$ undefined for each node $v \in V$

## General Algorithm

1. Initialise $d_s$ and $\pi_s$: $d_s[v] = \infty$, $\pi_s[v] =$ null for each $v \in V$
2. Set $d_s[s] \leftarrow 0$
3. Choose an edge $(u, v) \in E$

```
Relaxiere $(u,v)$:
        if $d_s[v] > d[u] + c(u,v)$ then
                $d_s[v] \leftarrow d_s[u] + c(u,v)$
                $\pi_s[v] \leftarrow u$
```

4. Repeat 3 until nothing can be relaxed any more.
   (until $d_s[v] \leq d_s[u] + c(u,v) \quad \forall (u,v) \in E$)

## It is Safe to Relax

At any time in the algorithm above it holds

$$d_s[v] \geq \delta(s,v) \quad \forall v \in V$$

In the relaxation step:

$$\begin{aligned} \delta(s,v) &\leq \delta(s,u) + \delta(u,v) && \text{[Triangle Inequality].} \\ \delta(s,u) &\leq d_s[u] && \text{[Induction Hypothesis].} \\ \delta(u,v) &\leq c(u,v) && \text{[Minimality of } \delta\text{]} \\ \Rightarrow \quad d_s[u] + c(u,v) &\geq \delta(s,v) \end{aligned}$$

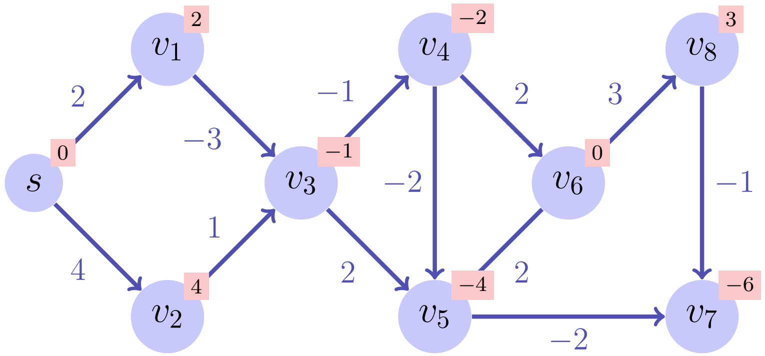
$$\Rightarrow \min\{d_s[v], d_s[u] + c(u,v)\} \geq \delta(s,v)$$

## Central Question

How / in which order should edges be chosen in above algorithm?

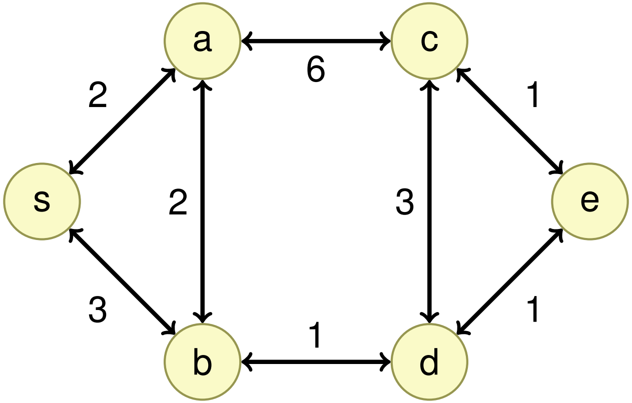
## Special Case: Directed Acyclic Graph (DAG)

DAG $\Rightarrow$ topological sorting returns optimal visiting order



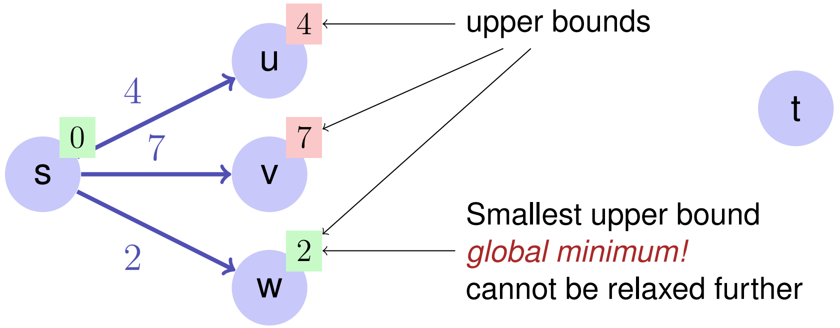
Top. Sort: $\Rightarrow$ Order $s, v_1, v_2, v_3, v_4, v_6, v_5, v_8, v_7$.

## Assumption (preliminary)



All weights of $G$ are *positive*.

## Observation (Dijkstra)



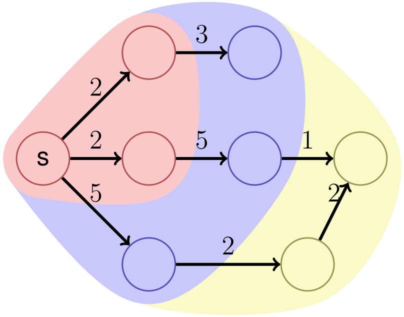
upper bounds

Smallest upper bound
*global minimum!*
cannot be relaxed further

## Basic Idea

Set $V$ of nodes is partitioned into

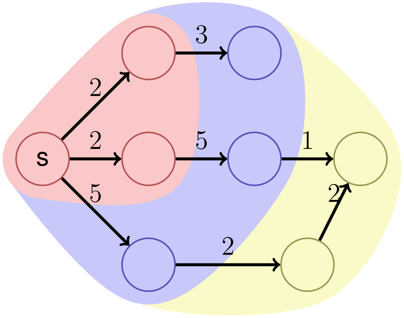
- the set $M$ of nodes for which a shortest path from $s$ is already known,
- the set $R = \bigcup_{v \in M} N^+(v) \setminus M$ of nodes where a shortest path is not yet known but that are accessible directly from $M$,
- the set $U = V \setminus (M \cup R)$ of nodes that have not yet been considered.



# Induction

Induction over $|M|$: choose nodes from $R$ with smallest upper bound. Add $r$ to $M$ and update $R$ and $U$ accordingly.

Correctness: if within the “wavefront” a node with minimal weight $w$ has been found then no path over later nodes (providing weight $\geq d$) can provide any improvement.

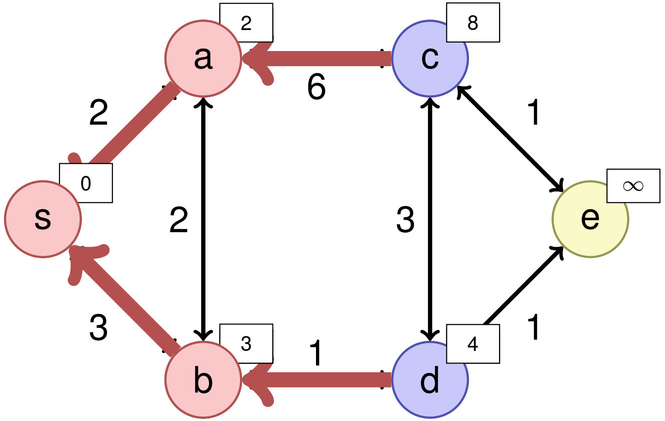


## Algorithm Dijkstra($G, s$)

**Input:** Positively weighted Graph $G = (V, E, c)$, starting point $s \in V$,
**Output:** Minimal weights $d$ of the shortest paths and corresponding predecessor node for each node.

```
foreach $u \in V$ do
    $d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow$ null
$d_s[s] \leftarrow 0$; $R \leftarrow \{s\}$
while $R \neq \emptyset$ do
    $u \leftarrow$ ExtractMin($R$)
    foreach $v \in N^+(u)$ do
        if $d_s[u] + c(u, v) < d_s[v]$ then
            $d_s[v] \leftarrow d_s[u] + c(u, v)$
            $\pi_s[v] \leftarrow u$
            $R \leftarrow R \cup \{v\}$
```

## Example



$$M = \{s, a, b\}$$

$$R = \{c, d\}$$

$$U = \{e\}$$

## Implementation: Data Structure for $R$?

Required operations:

- Insert (add to $R$)
- ExtractMin (over $R$) and DecreaseKey (Update in $R$)

```
foreach $v \in N^+(u)$ do
    if $d_s[u] + c(u,v) < d_s[v]$ then
        $d_s[v] \leftarrow d_s[u] + c(u,v)$
        $\pi_s[v] \leftarrow u$
        if $v \in R$ then
            DecreaseKey($R, v$)          // Update of a $d(v)$ in the heap of $R$
        else
            $R \leftarrow R \cup \{v\}$          // Update of $d(v)$ in the heap of $R$
```

MinHeap!

## DecreaseKey

- DecreaseKey: climbing in MinHeap in $\mathcal{O}(\log |V|)$
- Position in the heap?
    - alternative (a): Store position at the nodes
    - alternative (b): Hashtable of the nodes
    - alterantive (c): re-insert node after successful relax operation and mark it "deleted" once extracted (Lazy Deletion).[47]

[47] For lazy deletion a pair of egde (or target node) and distance is required.

## Runtime

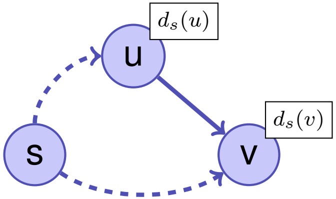
- $|V|\times$ ExtractMin: $\mathcal{O}(|V|\log|V|)$
- $|E|\times$ Insert or DecreaseKey: $\mathcal{O}(|E|\log|V|)$
- $1\times$ Init: $\mathcal{O}(|V|)$
- Overal: $\mathcal{O}(|E|\log|V|)$.

Can be improved when a data structure optimized for ExtractMin and DecreaseKey ist used (Fibonacci Heap), then runtime $\mathcal{O}(|E|+|V|\log|V|)$.

## General Weighted Graphs

Relaxing Step as before but with a return value:

```
Relax($u, v$) ($u, v \in V$, $(u, v) \in E$)
if $d_s[u] + c(u, v) < d_s[v]$ then
    $d_s[v] \leftarrow d_s[u] + c(u, v)$
    $\pi_s[v] \leftarrow u$
    return true
return false
```



Problem: cycles with negative weights can shorten the path, a shortest path is not guaranteed to exist.

## Dynamic Programming Approach (Bellman)

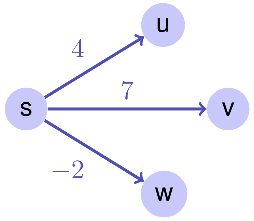
Induction over number of edges $d_s[i, v]$: Shortest path from $s$ to $v$ via maximally $i$ edges.

$$d_s[i, v] = \min\{d_s[i-1, v], \min_{(u,v)\in E}(d_s[i-1, u] + c(u, v))$$

$$d_s[0, s] = 0, d_s[0, v] = \infty \; \forall v \neq s.$$

## Dynamic Programming Approach (Bellman)

| | $s$ | $\cdots$ | $v$ | $\cdots$ | $w$ |
|---|---|---|---|---|---|
| $0$ | $0$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| $1$ | $0$ | $\infty$ | $7$ | $\infty$ | $-2$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $n-1$ | $0$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |



Algorithm: Iterate over last row until the relaxation steps do not provide any further changes, maximally $n-1$ iterations. If still changes, then there is no shortest path.

## Algorithm Bellman-Ford($G$, $s$)

**Input:** Graph $G = (V, E, c)$, starting point $s \in V$
**Output:** If return value true, minimal weights $d$ for all shortest paths from $s$, otherwise no shortest path.

```
**foreach** $u \in V$ **do**
    $d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow$ **null**
$d_s[s] \leftarrow 0$;
**for** $i \leftarrow 1$ **to** $|V|$ **do**
    $f \leftarrow$ false
    **foreach** $(u, v) \in E$ **do**
        $f \leftarrow f \vee \mathrm{Relax}(u, v)$
    **if** $f =$ false **then return** true
**return** false;
```

## All shortest Paths

Compute the weight of a shortest path for each pair of nodes.

- $|V|\times$ Application of Dijkstra's Shortest Path algorithm $\mathcal{O}(|V|\cdot|E|\cdot\log|V|)$ (with Fibonacci Heap: $\mathcal{O}(|V|^2\log|V|+|V|\cdot|E|)$)
- $|V|\times$ Application of Bellman-Ford: $\mathcal{O}(|E|\cdot|V|^2)$
- There are better ways!

## Induction via node number[48]

Consider weights of all shortest paths $S^k$ with intermediate nodes in $V^k := \{v_1, \dots, v_k\}$, provided that weights for all shortest paths $S^{k-1}$ with intermediate nodes in $V^{k-1}$ are given.

- $v_k$ no intermediate node of a shortest path of $v_i \rightsquigarrow v_j$ in $V^k$: Weight of a shortest path $v_i \rightsquigarrow v_j$ in $S^{k-1}$ is then also weight of shortest path in $S^k$.
- $v_k$ intermediate node of a shortest path $v_i \rightsquigarrow v_j$ in $V^k$: Sub-paths $v_i \rightsquigarrow v_k$ and $v_k \rightsquigarrow v_j$ contain intermediate nodes only from $S^{k-1}$.

[48]like for the algorithm of the reflexive transitive closure of Warshall

## DP Induction

$d^k(u,v)$ = Minimal weight of a path $u \rightsquigarrow v$ with intermediate nodes in $V^k$

Induktion

$$d^k(u,v) = \min\{d^{k-1}(u,v), d^{k-1}(u,k) + d^{k-1}(k,v)\}(k \geq 1)$$
$$d^0(u,v) = c(u,v)$$

## DP Algorithm Floyd-Warshall($G$)

```
Input: Acyclic Graph $G = (V, E, c)$
Output: Minimal weights of all paths $d$
$d^0 \leftarrow c$
for $k \leftarrow 1$ to $|V|$ do
    for $i \leftarrow 1$ to $|V|$ do
        for $j \leftarrow 1$ to $|V|$ do
            $d^k(v_i, v_j) = \min\{d^{k-1}(v_i, v_j), d^{k-1}(v_i, v_k) + d^{k-1}(v_k, v_j)\}$
```

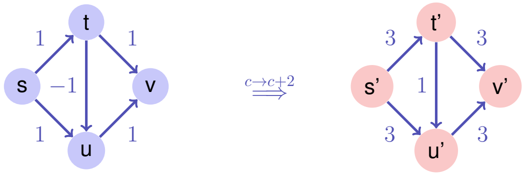
Runtime: $\Theta(|V|^3)$

Remark: Algorithm can be executed with a single matrix $d$ (in place).

## Reweighting

Idea: Reweighting the graph in order to apply Dijkstra’s algorithm.

The following does *not* work. The graphs are not equivalent in terms of shortest paths.



## Reweighting

Other Idea: “Potential” (Height) on the nodes

- $G = (V, E, c)$ a weighted graph.
- Mapping $h : V \to \mathbb{R}$
- New weights

$$\tilde{c}(u, v) = c(u, v) + h(u) - h(v), \ (u, v \in V)$$

## Reweighting

*Observation:* A path $p$ is shortest path in in $G = (V, E, c)$ iff it is shortest path in in $\tilde{G} = (V, E, \tilde{c})$

$$\begin{aligned}\tilde{c}(p) &= \sum_{i=1}^{k} \tilde{c}(v_{i-1}, v_i) = \sum_{i=1}^{k} c(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i) \\ &= h(v_0) - h(v_k) + \sum_{i=1}^{k} c(v_{i-1}, v_i) = c(p) + h(v_0) - h(v_k)\end{aligned}$$

Thus $\tilde{c}(p)$ minimal in all $v_0 \rightsquigarrow v_k \iff c(p)$ minimal in all $v_0 \rightsquigarrow v_k$.

Weights of cycles are invariant: $\tilde{c}(v_0, \ldots, v_k = v_0) = c(v_0, \ldots, v_k = v_0)$

# Johnson's Algorithm

Add a new node $s \notin V$:

$$\begin{aligned} G' &= (V', E', c') \\ V' &= V \cup \{s\} \\ E' &= E \cup \{(s, v) : v \in V\} \\ c'(u, v) &= c(u, v),\ u \neq s \\ c'(s, v) &= 0 (v \in V) \end{aligned}$$

## Johnson's Algorithm

If no negative cycles, choose as height function the weight of the shortest paths from $s$,

$$h(v) = d(s, v).$$

For a minimal weight $d$ of a path the following triangular inequality holds:

$$d(s, v) \leq d(s, u) + c(u, v).$$

Substitution yields $h(v) \leq h(u) + c(u, v)$. Therefore

$$\tilde{c}(u, v) = c(u, v) + h(u) - h(v) \geq 0.$$

# Algorithm Johnson($G$)

**Input:** Weighted Graph $G = (V, E, c)$
**Output:** Minimal weights of all paths $D$.

```
New node $s$. Compute $G' = (V', E', c')$
if BellmanFord($G', s$) = false then return "graph has negative cycles"
foreach $v \in V'$ do
    $h(v) \leftarrow d(s, v)$ // $d$ aus BellmanFord Algorithmus
foreach $(u, v) \in E'$ do
    $\tilde{c}(u, v) \leftarrow c(u, v) + h(u) - h(v)$
foreach $u \in V$ do
    $\tilde{d}(u, \cdot) \leftarrow$ Dijkstra($\tilde{G}', u$)
    foreach $v \in V$ do
        $D(u, v) \leftarrow \tilde{d}(u, v) + h(v) - h(u)$
```

## Analysis

Runtimes

- Computation of $G'$: $\mathcal{O}(|V|)$
- Bellman Ford $G'$: $\mathcal{O}(|V| \cdot |E|)$
- $|V|\times$ Dijkstra $\mathcal{O}(|V| \cdot |E| \cdot \log |V|)$
  (with Fibonacci Heap: $\mathcal{O}(|V|^2 \log |V| + |V| \cdot |E|)$)

Overal $\mathcal{O}(|V| \cdot |E| \cdot \log |V|)$
($\mathcal{O}(|V|^2 \log |V| + |V| \cdot |E|)$)

# 25. Minimum Spanning Trees

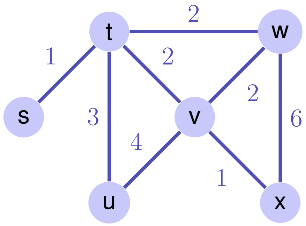
Motivation, Greedy, Algorithm Kruskal, General Rules, ADT Union-Find, Algorithm Jarnik, Prim, Dijkstra, ,Algorithm Jarnik, Prim, Dijkstra ,Fibonacci Heaps

[Ottman/Widmayer, Kap. 9.6, 6.2, 6.1, Cormen et al, Kap. 23, 19]

## Problem

*Given:* Undirected, weighted, connected graph $G = (V, E, c)$.

*Wanted:* Minimum Spanning Tree $T = (V, E')$: connected, cycle-free subgraph $E' \subset E$, such that $\sum_{e \in E'} c(e)$ minimal.



## Application Examples

- Network-Design: find the cheapest / shortest network that connects all nodes.
- Approximation of a solution of the travelling salesman problem: find a round-trip, as short as possible, that visits each node once.

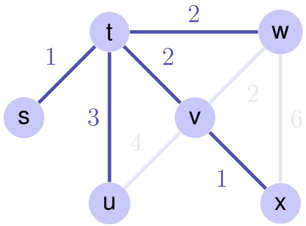
## Greedy Procedure

Recall:

- Greedy algorithms compute the solution stepwise choosing locally optimal solutions.
- Most problems cannot be solved with a greedy algorithm.
- The Minimum Spanning Tree problem can be solved with a greedy strategy.

## Greedy Idea (Kruskal, 1956)

Construct $T$ by adding the cheapest edge that does not generate a cycle.



(Solution is not unique.)

## Algorithm MST-Kruskal($G$)

**Input:** Weighted Graph $G = (V, E, c)$
**Output:** Minimum spanning tree with edges $A$.

```
Sort edges by weight $c(e_1) \leq ... \leq c(e_m)$
$A \leftarrow \emptyset$
for $k = 1$ to $|E|$ do
    if $(V, A \cup \{e_k\})$ acyclic then
        $A \leftarrow A \cup \{e_k\}$
return $(V, A, c)$
```

## Correctness

At each point in the algorithm $(V, A)$ is a forest, a set of trees.

MST-Kruskal considers each edge $e_k$ exactly once and either chooses or rejects $e_k$

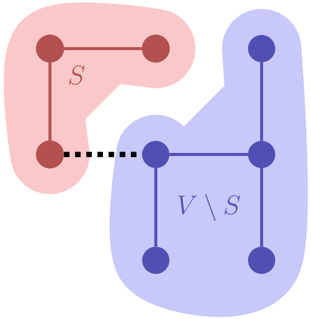
Notation (snapshot of the state in the running algorithm)

- $A$: Set of selected edges
- $R$: Set of rejected edges
- $U$: Set of yet undecided edges

## Cut

A cut of $G$ is a partition $S, V - S$ of $V$. ($S \subseteq V$).

An edge crosses a cut when one of its endpoints is in $S$ and the other is in $V \setminus S$.



## Rules

1. Selection rule: choose a cut that is not crossed by a selected edge. Of all undecided edges that cross the cut, select the one with minimal weight.
2. Rejection rule: choose a cycle without rejected edges. Of all undecided edges of the cycle, reject those with maximal weight.

## Rules

Kruskal applies both rules:

1. A selected $e_k$ connects two connection components, otherwise it would generate a cycle. $e_k$ is minimal, i.e. a cut can be chosen such that $e_k$ crosses and $e_k$ has minimal weight.
2. A rejected $e_k$ is contained in a cycle. Within the cycle $e_k$ has minimal weight.

## Correctness

**Theorem**

*Every algorithm that applies the rules above in a step-wise manner until $U = \emptyset$ is correct.*

Consequence: MST-Kruskal is correct.

## Selection invariant

*Invariant:* At each step there is a minimal spanning tree that contains all selected and none of the rejected edges.

If both rules satisfy the invariant, then the algorithm is correct. Induction:

- At beginning: $U = E$, $R = A = \emptyset$. Invariant obviously holds.
- Invariant is preserved at each step of the algorithm.
- At the end: $U = \emptyset$, $R \cup A = E \Rightarrow (V, A)$ is a spanning tree.

Proof of the theorem: show that both rules preserve the invariant.

## Selection rule preserves the invariant

At each step there is a minimal spanning tree $T$ that contains all selected and none of the rejected edges.

Choose a cut that is not crossed by a selected edge. Of all undecided edges that cross the cut, select the egde $e$ with minimal weight.

- Case 1: $e \in T$ (done)
- Case 2: $e \notin T$. Then $T \cup \{e\}$ contains a cycle that contains $e$ Cycle must have a second edge $e'$ that also crosses the cut.[49] Because $e' \notin R$ , $e' \in U$. Thus $c(e) \leq c(e')$ and $T' = T \setminus \{e'\} \cup \{e\}$ is also a minimal spanning tree (and $c(e) = c(e')$).

[49]Such a cycle contains at least one node in $S$ and one node in $V \setminus S$ and therefore at lease to edges between $S$ and $V \setminus S$.

## Rejection rule preserves the invariant

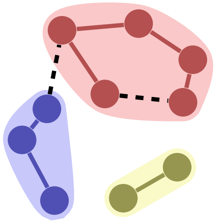
At each step there is a minimal spanning tree $T$ that contains all selected and none of the rejected edges.

Choose a cycle without rejected edges. Of all undecided edges of the cycle, reject an edge $e$ with maximal weight.

- Case 1: $e \notin T$ (done)
- Case 2: $e \in T$. Remove $e$ from $T$, This yields a cut. This cut must be crossed by another edge $e'$ of the cycle. Because $c(e') \leq c(e)$ , $T' = T \setminus \{e\} \cup \{e'\}$ is also minimal (and $c(e) = c(e')$).

## Implementation Issues

Consider a set of sets $i \equiv A_i \subset V$. To identify cuts and cycles: membership of the both ends of an edge to sets?



## Implementation Issues

General problem: partition (set of subsets) .e.g.
$\{\{1,2,3,9\},\{7,6,4\},\{5,8\},\{10\}\}$

Required: Abstract data type “Union-Find” with the following operations

- Make-Set($i$): create a new set represented by $i$.
- Find($e$): name of the set $i$ that contains $e$ .
- Union($i,j$): union of the sets with names $i$ and $j$.

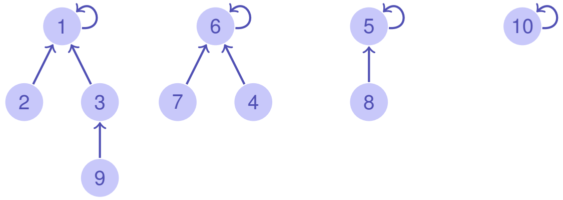
## Union-Find Algorithm MST-Kruskal($G$)

**Input:** Weighted Graph $G = (V, E, c)$
**Output:** Minimum spanning tree with edges $A$.

```
Sort edges by weight $c(e_1) \leq ... \leq c(e_m)$
$A \leftarrow \emptyset$
for $k = 1$ to $|V|$ do
    MakeSet($k$)
for $k = 1$ to $m$ do
    $(u, v) \leftarrow e_k$
    if Find($u$) $\neq$ Find($v$) then
        Union(Find($u$), Find($v$))
        $A \leftarrow A \cup e_k$
    else                                        // conceptual: $R \leftarrow R \cup e_k$
return $(V, A, c)$
```

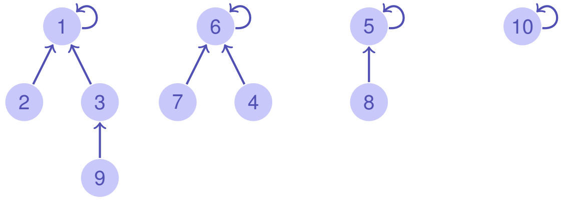
## Implementation Union-Find

Idea: tree for each subset in the partition,e.g.
$\{\{1,2,3,9\},\{7,6,4\},\{5,8\},\{10\}\}$



roots = names (representatives) of the sets,
trees = elements of the sets

## Implementation Union-Find



Representation as array:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Parent | **1** | 1 | 1 | 6 | **5** | **6** | 5 | 5 | 3 | **10** |

## Implementation Union-Find

| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Parent | **1** | 1 | 1 | 6 | **5** | **6** | 5 | 5 | 3 | **10** |

| | |
|---|---|
| Make-Set($i$) | $p[i] \leftarrow i$; **return** $i$ |
| Find($i$) | **while** $(p[i] \neq i)$ **do** $i \leftarrow p[i]$<br>**return** $i$ |
| Union($i, j$) [50] | $p[j] \leftarrow i$; |

[50] $i$ and $j$ need to be names (roots) of the sets. Otherwise use Union(Find($i$),Find($j$))

## Optimisation of the runtime for Find

Tree may degenerate. Example: Union$(8, 7)$, Union$(7, 6)$, Union$(6, 5)$, ...

| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | .. |
|---|---|---|---|---|---|---|---|---|---|
| Parent | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | .. |

Worst-case running time of Find in $\Theta(n)$.

## Optimisation of the runtime for Find

Idea: always append smaller tree to larger tree. Requires additional size information (array) $g$

```
Make-Set($i$)    $p[i] \leftarrow i$; $g[i] \leftarrow 1$; **return** $i$

Union($i, j$)    **if** $g[j] > g[i]$ **then** swap($i, j$)
                 $p[j] \leftarrow i$
                 **if** $g[i] = g[j]$ **then** $g[i] \leftarrow g[i] + 1$
```

$\Rightarrow$ Tree depth (and worst-case running time for Find) in $\Theta(\log n)$

## [Observation]

**Theorem**

*The method above (union by size) preserves the following property of the trees: a tree of height $h$ has at least $2^h$ nodes.*

Immediate consequence: runtime Find = $\mathcal{O}(\log n)$.

## [Proof]

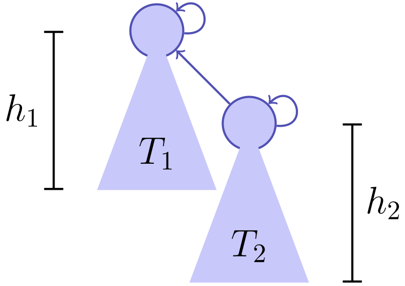
Induction: by assumption, sub-trees have at least $2^{h_i}$ nodes. WLOG: $h_2 \leq h_1$

- $h_2 < h_1$:

$$h(T_1 \oplus T_2) = h_1 \Rightarrow g(T_1 \oplus T_2) \geq 2^h$$

- $h_2 = h_1$:

$$g(T_1) \geq g(T_2) \geq 2^{h_2}$$
$$\Rightarrow g(T_1 \oplus T_2) = g(T_1) + g(T_2) \geq 2 \cdot 2^{h_2} = 2^{h(T_1 \oplus T_2)}$$



## Further improvement

Link all nodes to the root when Find is called.

Find($i$):

```
$j \leftarrow i$
**while** $(p[i] \neq i)$ **do** $i \leftarrow p[i]$
**while** $(j \neq i)$ **do**
    $t \leftarrow j$
    $j \leftarrow p[j]$
    $p[t] \leftarrow i$
**return** $i$
```

Cost: amortised *nearly* constant (inverse of the Ackermann-function).[51]

[51] We do not go into details here.

## Running time of Kruskal's Algorithm

- Sorting of the edges: $\Theta(|E| \log |E|) = \Theta(|E| \log |V|)$. [52]
- Initialisation of the Union-Find data structure $\Theta(|V|)$
- $|E| \times$ Union(Find($x$),Find($y$)): $\mathcal{O}(|E| \log |E|) = \mathcal{O}(|E| \log |V|)$.

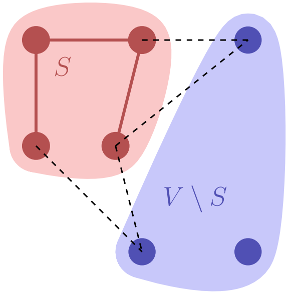
Overal $\Theta(|E| \log |V|)$.

[52]because $G$ is connected: $|V| \leq |E| \leq |V|^2$

## Algorithm of Jarnik (1930), Prim, Dijkstra (1959)

Idea: start with some $v \in V$ and grow the spanning tree from here by the acceptance rule.

```
$A \leftarrow \emptyset$
$S \leftarrow \{v_0\}$
for $i \leftarrow 1$ to $|V|$ do
    Choose cheapest $(u, v)$ mit $u \in S$, $v \notin S$
    $A \leftarrow A \cup \{(u, v)\}$
    $S \leftarrow S \cup \{v\}$ // (Coloring)
```



Remark: a union-Find data structure is not required. It suffices to color nodes when they are added to $S$.

## Running time

Trivially $\mathcal{O}(|V| \cdot |E|)$.

Improvement (like with Dijkstra's ShortestPath)

- With Min-Heap: costs
    - Initialization (node coloring) $\mathcal{O}(|V|)$
    - $|V|\times$ ExtractMin = $\mathcal{O}(|V| \log |V|)$,
    - $|E|\times$ Insert or DecreaseKey: $\mathcal{O}(|E| \log |V|)$,

  $\mathcal{O}(|E| \cdot \log |V|)$
- With a Fibonacci-Heap: $\mathcal{O}(|E| + |V| \cdot \log |V|)$.

## Fibonacci Heaps

Data structure for elements with key with operations

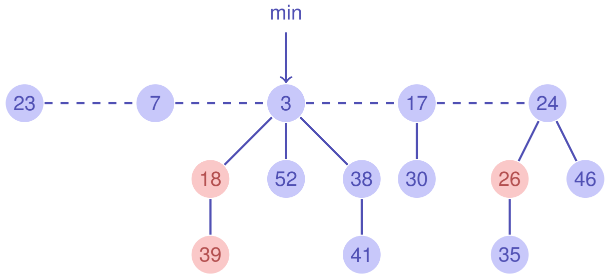
- MakeHeap(): Return new heap without elements
- Insert$(H, x)$: Add $x$ to $H$
- Minimum$(H)$: return a pointer to element $m$ with minimal key
- ExtractMin$(H)$: return and remove (from $H$) pointer to the element $m$
- Union$(H_1, H_2)$: return a heap merged from $H_1$ and $H_2$
- DecreaseKey$(H, x, k)$: decrease the key of $x$ in $H$ to $k$
- Delete $(H, x)$: remove element $x$ from $H$

## Advantage over binary heap?

| | Binary Heap (worst-Case) | Fibonacci Heap (amortized) |
|---|---|---|
| MakeHeap | $\Theta(1)$ | $\Theta(1)$ |
| Insert | $\Theta(\log n)$ | $\Theta(1)$ |
| Minimum | $\Theta(1)$ | $\Theta(1)$ |
| ExtractMin | $\Theta(\log n)$ | $\Theta(\log n)$ |
| Union | $\Theta(n)$ | $\Theta(1)$ |
| DecreaseKey | $\Theta(\log n)$ | $\Theta(1)$ |
| Delete | $\Theta(\log n)$ | $\Theta(\log n)$ |

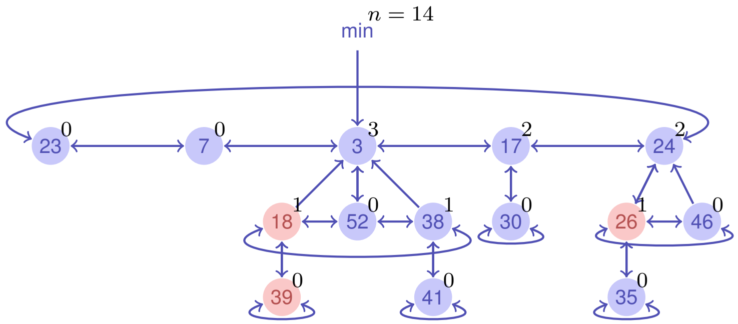
## Structure

Set of trees that respect the Min-Heap property. Nodes that can be marked.



## Implementation

Doubly linked lists of nodes with a marked-flag and number of children. Pointer to minimal Element and number nodes.



## Simple Operations

- MakeHeap (trivial)
- Minimum (trivial)
- Insert($H, e$)

  1. Insert new element into root-list
  2. If key is smaller than minimum, reset min-pointer.

- Union ($H_1, H_2$)

  1. Concatenate root-lists of $H_1$ and $H_2$
  2. Reset min-pointer.

- Delete($H, e$)

  1. DecreaseKey($H, e, -\infty$)
  2. ExtractMin($H$)

## ExtractMin

1. Remove minimal node $m$ from the root list
2. Insert children of $m$ into the root list
3. Merge heap-ordered trees with the same degrees until all trees have a different degree:
   Array of degrees $a[0, \dots, n]$ of elements, empty at beginning.
   For each element $e$ of the root list:
   a. Let $g$ be the degree of $e$
   b. If $a[g] = nil$: $a[g] \leftarrow e$.
   c. If $e' := a[g] \neq nil$: Merge $e$ with $e'$ resutling in $e''$ and set $a[g] \leftarrow nil$. Set $e''$ unmarked. Re-iterate with $e \leftarrow e''$ having degree $g + 1$.

## DecreaseKey $(H, e, k)$

1. Remove $e$ from its parent node $p$ (if existing) and decrease the degree of $p$ by one.
2. Insert$(H, e)$
3. Avoid too thin trees:
   a. If $p = nil$ then done.
   b. If $p$ is unmarked: mark $p$ and done.
   c. If $p$ marked: unmark $p$ and cut $p$ from its parent $pp$. Insert $(H, p)$. Iterate with $p \leftarrow pp$.

## Estimation of the degree

**Theorem**

*Let $p$ be a node of a F-Heap $H$. If child nodes of $p$ are sorted by time of insertion (Union), then it holds that the $i$th child node has a degree of at least $i-2$.*

Proof: $p$ may have had more children and lost by cutting. When the $i$th child $p_i$ was linked, $p$ and $p_i$ must at least have had degree $i-1$. $p_i$ may have lost at least one child (marking!), thus at least degree $i-2$ remains.

## Estimation of the degree

**Theorem**

*Every node $p$ with degree $k$ of a F-Heap is the root of a subtree with at least $F_{k+1}$ nodes. ($F$: Fibonacci-Folge)*

Proof: Let $S_k$ be the minimal number of successors of a node of degree $k$ in a F-Heap plus $1$ (the node itself). Clearly $S_0 = 1$, $S_1 = 2$. With the previous theorem $S_k \geq 2 + \sum_{i=0}^{k-2} S_i, k \geq 2$ ($p$ and nodes $p_1$ each 1). For Fibonacci numbers it holds that (induction) $F_k \geq 2 + \sum_{i=2}^{k} F_i, k \geq 2$ and thus (also induction) $S_k \geq F_{k+2}$.

Fibonacci numbers grow exponentially fast ($\mathcal{O}(\varphi^k)$) Consequence: maximal degree of an arbitrary node in a Fibonacci-Heap with $n$ nodes is $\mathcal{O}(\log n)$.

# Amortized worst-case analysis Fibonacci Heap

$t(H)$: number of trees in the root list of $H$, $m(H)$: number of marked nodes in $H$ not within the root-list, Potential function $\Phi(H) = t(H) + 2 \cdot m(H)$. At the beginnning $\Phi(H) = 0$. Potential always non-negative.

Amortized costs:

- Insert($H, x$): $t'(H) = t(H) + 1$, $m'(H) = m(H)$, Increase of the potential: $1$, Amortized costs $\Theta(1) + 1 = \Theta(1)$
- Minimum($H$): Amortized costs = real costs = $\Theta(1)$
- Union($H_1, H_2$): Amortized costs = real costs = $\Theta(1)$

## Amortized costs of ExtractMin

- Number trees in the root list $t(H)$.
- Real costs of ExtractMin operation $\mathcal{O}(\log n + t(H))$.
- When merged still $\mathcal{O}(\log n)$ nodes.
- Number of markings can only get smaller when trees are merged
- Thus maximal amortized costs of ExtractMin

$$\mathcal{O}(\log n + t(H)) + \mathcal{O}(\log n) - \mathcal{O}(t(H)) = \mathcal{O}(\log n).$$

## Amortized costs of DecreaseKey

- Assumption: DecreaseKey leads to $c$ cuts of a node from its parent node, real costs $\mathcal{O}(c)$
- $c$ nodes are added to the root list
- Delete $(c-1)$ mark flags, addition of at most one mark flag
- Amortized costs of DecreaseKey:

$$\mathcal{O}(c) + (t(H) + c) + 2 \cdot (m(H) - c + 2)) - (t(H) + 2m(H)) = \mathcal{O}(1)$$

# 26. Flow in Networks

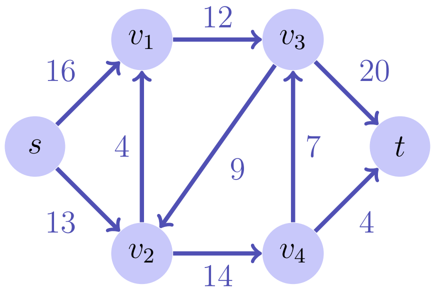
Flow Network, Maximal Flow, Cut, Rest Network, Max-flow Min-cut Theorem, Ford-Fulkerson Method, Edmonds-Karp Algorithm, Maximal Bipartite Matching [Ottman/Widmayer, Kap. 9.7, 9.8.1], [Cormen et al, Kap. 26.1-26.3]

## Motivation

- Modelling flow of fluents, components on conveyors, current in electrical networks or information flow in communication networks.
- Connectivity of Communication Networks, Bipartite Matching, Circulation, Scheduling, Image Segmentation, Baseball Eliminination...

# Flow Network

- *Flow network* $G = (V, E, c)$: directed graph with *capacities*
- Antiparallel edges forbidden: $(u, v) \in E \;\Rightarrow\; (v, u) \notin E$.
- Model a missing edge $(u, v)$ by $c(u, v) = 0$.
- *Source* $s$ and *sink* $t$: special nodes. Every node $v$ is on a path between $s$ and $t$ : $s \rightsquigarrow v \rightsquigarrow t$

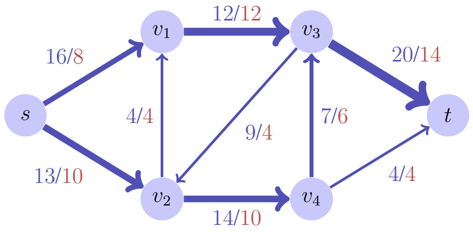


## Flow

A *Flow* $f : V \times V \to \mathbb{R}$ fulfills the following conditions:

- *Bounded Capacity*:
  For all $u, v \in V$: $f(u,v) \le c(u,v)$.
- *Skew Symmetry*:
  For all $u, v \in V$: $f(u,v) = -f(v,u)$.
- *Conservation of flow*:
  For all $u \in V \setminus \{s,t\}$:

$$\sum_{v \in V} f(u,v) = 0.$$



*Value* of the flow:
$|f| = \sum_{v \in V} f(s,v)$.
Here $|f| = 18$.

## How large can a flow possibly be?

Limiting factors: cuts

- *cut separating $s$ from $t$*: Partition of $V$ into $S$ and $T$ with $s \in S$, $t \in T$.
- *Capacity* of a cut: $c(S,T) = \sum_{v \in S, v' \in T} c(v,v')$
- *Minimal cut*: cut with minimal capacity.
- *Flow over the cut*: $f(S,T) = \sum_{v \in S, v' \in T} f(v,v')$

## Implicit Summation

Notation: Let $U, U' \subseteq V$

$$f(U,U') := \sum_{\substack{u \in U \\ u' \in U'}} f(u,u'), \qquad f(u,U') := f(\{u\},U')$$

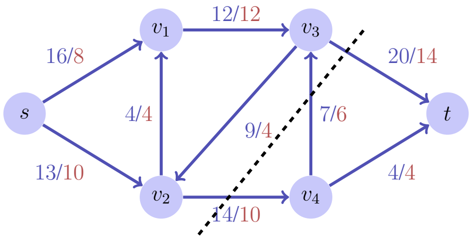
Thus

- $|f| = f(s,V)$
- $f(U,U) = 0$
- $f(U,U') = -f(U',U)$
- $f(X \cup Y, Z) = f(X,Z) + f(Y,Z)$, if $X \cap Y = \emptyset$.
- $f(R,V) = 0$ if $R \cap \{s,t\} = \emptyset$. [flow conversation!]

# How large can a flow possibly be?

For each flow and each cut it holds that $f(S,T) = |f|$:

$$f(S,T) = f(S,V) - \underbrace{f(S,S)}_{0} = f(S,V)$$
$$= f(s,V) + f(\underbrace{S - \{s\}}_{\not\ni t, \not\ni s}, V) = |f|.$$

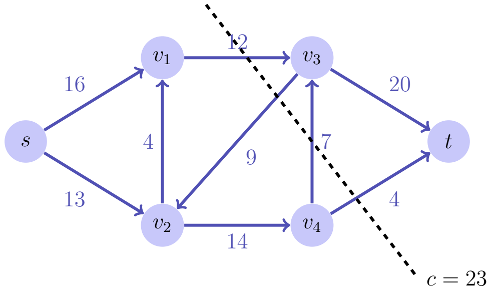


## Maximal Flow ?

In particular, for each cut $(S, T)$ of $V$.

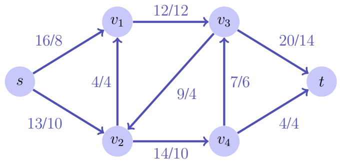
$$|f| \leq \sum_{v \in S, v' \in T} c(v, v') = c(S, T)$$

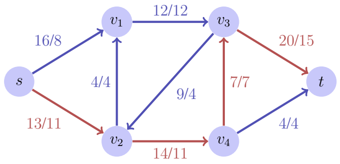
Will discover that equality holds for $\min_{S,T} c(S, T)$.

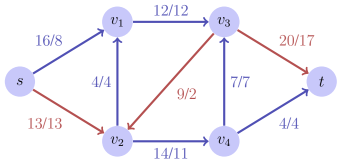


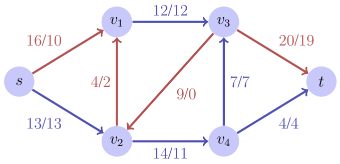
## Maximal Flow ?

Naive Procedure









Conclusion: greedy increase of flow does not solve the problem.

## The Method of Ford-Fulkerson

- Start with $f(u,v) = 0$ for all $u, v \in V$
- Determine rest network* $G_f$ and expansion path in $G_f$
- Increase flow via expansion path*
- Repeat until no expansion path available.

$$\begin{aligned} G_f &:= (V, E_f, c_f) \\ c_f(u,v) &:= c(u,v) - f(u,v) \quad \forall u, v \in V \\ E_f &:= \{(u,v) \in V \times V | c_f(u,v) > 0\} \end{aligned}$$

*Will now be explained

## Increase of flow, negative!

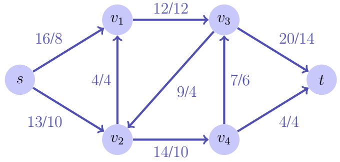
Let some flow $f$ in the network be given.

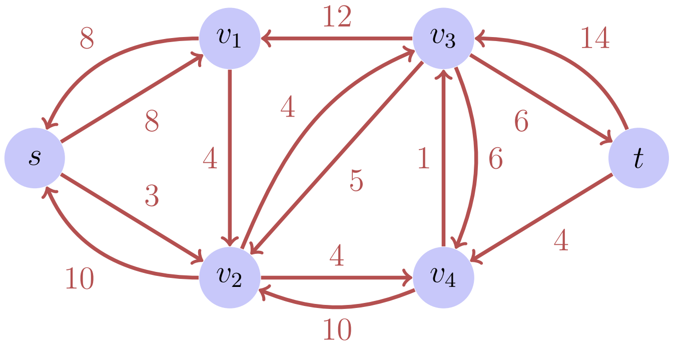
Finding:

- Increase of the flow along some edge possible, when flow can be increased along the edge,i.e. if $f(u,v) < c(u,v)$.
  Rest capacity $c_f(u,v) = c(u,v) - f(u,v) > 0$.
- Increase of flow *against the direction* of the edge possible, if flow can be reduced along the edge, i.e. if $f(u,v) > 0$.
  Rest capacity $c_f(v,u) = f(u,v) > 0$.

## Rest Network

*Rest network* $G_f$ provided by the edges with positive rest capacity:





Rest networks provide the same kind of properties as flow networks with the exception of permitting antiparallel capacity-edges

## Observation

**Theorem**

*Let $G = (V, E, c)$ be a flow network with source $s$ and sink $t$ and $f$ a flow in $G$. Let $G_f$ be the corresponding rest networks and let $f'$ be a flow in $G_f$. Then $f \oplus f'$ with*

$$(f \oplus f')(u,v) = f(u,v) + f'(u,v)$$

*defines a flow in $G$ with value $|f| + |f'|$.*

## Proof

$f \oplus f'$ defines a flow in $G$:

- capacity limit

$$(f \oplus f')(u,v) = f(u,v) + \underbrace{f'(u,v)}_{\leq c(u,v) - f(u,v)} \leq c(u,v)$$

- skew symmetry

$$(f \oplus f')(u,v) = -f(v,u) + -f'(v,u) = -(f \oplus f')(v,u)$$

- flow conservation $u \in V - \{s,t\}$:

$$\sum_{v \in V} (f \oplus f')(u,v) = \sum_{v \in V} f(u,v) + \sum_{v \in V} f'(u,v) = 0$$

## Proof

Value of $f \oplus f'$

$$\begin{aligned}
|f \oplus f'| &= (f \oplus f')(s, V) \\
&= \sum_{u \in V} f(s, u) + f'(s, u) \\
&= f(s, V) + f'(s, V) \\
&= |f| + |f'|
\end{aligned}$$

■

## Augmenting Paths

*expansion path* $p$: simple path from $s$ to $t$ in the rest network $G_f$.

*Rest capacity* $c_f(p) = \min\{c_f(u,v) : (u,v)$ edge in $p\}$

## Flow in $G_f$

**Theorem**

*The mapping $f_p : V \times V \to \mathbb{R}$,*

$$f_p(u,v) = \begin{cases} c_f(p) & \textit{if } (u,v) \textit{ edge in } p \\ -c_f(p) & \textit{if } (v,u) \textit{ edge in } p \\ 0 & \textit{otherwise} \end{cases}$$

*provides a flow in $G_f$ with value $|f_p| = c_f(p) > 0$.*

$f_p$ is a flow (easy to show). there is one and only one $u \in V$ with $(s,u) \in p$. Thus $|f_p| = \sum_{v \in V} f_p(s,v) = f_p(s,u) = c_f(p)$.

## Consequence

Strategy for an algorithm:

With an expansion path $p$ in $G_f$ the flow $f \oplus f_p$ defines a new flow with value $|f \oplus f_p| = |f| + |f_p| > |f|$.

## Max-Flow Min-Cut Theorem

**Theorem**

*Let $f$ be a flow in a flow network $G = (V, E, c)$ with source $s$ and sink $t$. The following statementsa are equivalent:*

1. *$f$ is a maximal flow in $G$*
2. *The rest network $G_f$ does not provide any expansion paths*
3. *It holds that $|f| = c(S, T)$ for a cut $(S, T)$ of $G$.*

## Proof

- $(3) \Rightarrow (1)$:
  It holds that $|f| \leq c(S,T)$ for all cuts $S, T$. From $|f| = c(S,T)$ it follows that $|f|$ is maximal.
- $(1) \Rightarrow (2)$:
  $f$ maximal Flow in $G$. Assumption: $G_f$ has some expansion path $|f \oplus f_p| = |f| + |f_p| > |f|$. Contradiction.

## Proof $(2) \Rightarrow (3)$

Assumption: $G_f$ has no expansion path

Define $S = \{v \in V : \text{ there is a path } s \rightsquigarrow v \text{ in } G_f\}$.

$(S, T) := (S, V \setminus S)$ is a cut: $s \in S, t \in T$.

Let $u \in S$ and $v \in T$. Then $c_f(u, v) = 0$, also
$c_f(u, v) = c(u, v) - f(u, v) = 0$. Somit $f(u, v) = c(u, v)$.

Thus

$$|f| = f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) = \sum_{u \in S} \sum_{v \in T} c(u, v) = C(S, T).$$

■

## Algorithm Ford-Fulkerson($G, s, t$)

**Input:** Flow network $G = (V, E, c)$
**Output:** Maximal flow $f$.

```
for $(u, v) \in E$ do
    $f(u, v) \leftarrow 0$
while Exists path $p : s \rightsquigarrow t$ in rest network $G_f$ do
    $c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \in p\}$
    foreach $(u, v) \in p$ do
        $f(u, v) \leftarrow f(u, v) + c_f(p)$
        $f(v, u) \leftarrow f(v, u) - c_f(p)$
```

## Practical Consideration

In an implementation of the Ford-Fulkerson algorithm the negative flow egdes are usually not stored because their value always equals the negated value of the antiparallel edge.

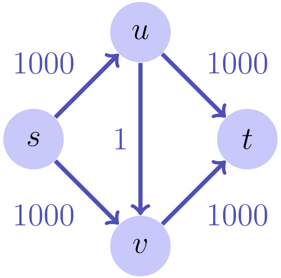
$f(u,v) \leftarrow f(u,v) + c_f(p)$
$f(v,u) \leftarrow f(v,u) - c_f(p)$

is then transformed to

```
if $(u,v) \in E$ then
    $f(u,v) \leftarrow f(u,v) + c_f(p)$
else
    $f(v,u) \leftarrow f(v,u) - c_f(p)$
```

## Analysis

- The Ford-Fulkerson algorithm does not necessarily have to converge for irrational capacities. For integers or rational numbers it terminates.
- For an integer flow, the algorithms requires maximally $|f_{\max}|$ iterations of the while loop (because the flow increases minimally by 1). Search a single increasing path (e.g. with DFS or BFS) $\mathcal{O}(|E|)$ Therefore $\mathcal{O}(f_{\max}|E|)$.



With an unlucky choice the algorithm may require up to 2000 iterations here.

## Edmonds-Karp Algorithm

Choose in the Ford-Fulkerson-Method for finding a path in $G_f$ the expansion path of shortest possible length (e.g. with BFS)

## Edmonds-Karp Algorithm

**Theorem**

*When the Edmonds-Karp algorithm is applied to some integer valued flow network $G = (V, E)$ with source $s$ and sink $t$ then the number of flow increases applied by the algorithm is in $\mathcal{O}(|V| \cdot |E|)$.*
*$\Rightarrow$ Overal asymptotic runtime: $\mathcal{O}(|V| \cdot |E|^2)$*

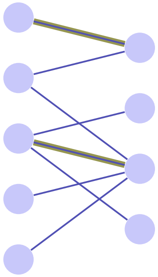
[Without proof]

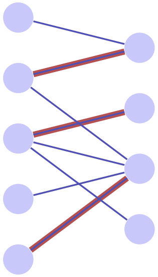
## Application: maximal bipartite matching

Given: bipartite undirected graph $G = (V, E)$.

Matching $M$: $M \subseteq E$ such that $|\{m \in M : v \in m\}| \leq 1$ for all $v \in V$.

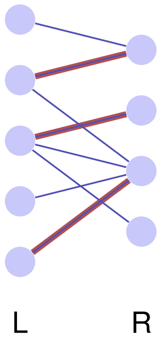
Maximal Matching $M$: Matching $M$, such that $|M| \geq |M'|$ for each matching $M'$.

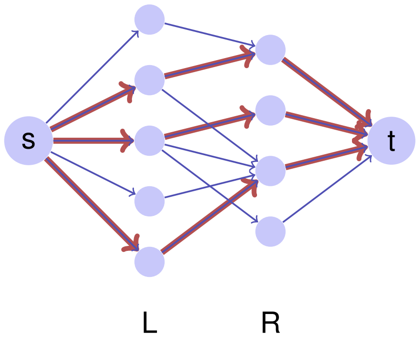




## Corresponding flow network

Construct a flow network that corresponds to the partition $L, R$ of a bipartite graph with source $s$ and sink $t$, with directed edges from $s$ to $L$, from $L$ to $R$ and from $R$ to $t$. Each edge has capacity $1$.





## Integer number theorem

**Theorem**

*If the capacities of a flow network are integers, then the maximal flow generated by the Ford-Fulkerson method provides integer numbers for each $f(u, v)$, $u, v \in V$.*

[without proof]

Consequence: Ford-Fulkerson generates for a flow network that corresponds to a bipartite graph a maximal matching $M = \{(u, v) : f(u, v) = 1\}$.

# 27. Parallel Programming I

Moore’s Law and the Free Lunch, Hardware Architectures, Parallel Execution, Flynn’s Taxonomy, Multi-Threading, Parallelism and Concurrency, C++ Threads, Scalability: Amdahl and Gustafson, Data-parallelism, Task-parallelism, Scheduling

[Task-Scheduling: Cormen et al, Kap. 27] [Concurrency, Scheduling: Williams, Kap. 1.1 – 1.2]

## The Free Lunch

The free lunch is over [53]

[53] "The Free Lunch is Over", a fundamental turn toward concurrency in software, Herb Sutter, Dr. Dobb's Journal, 2005

## Moore’s Law

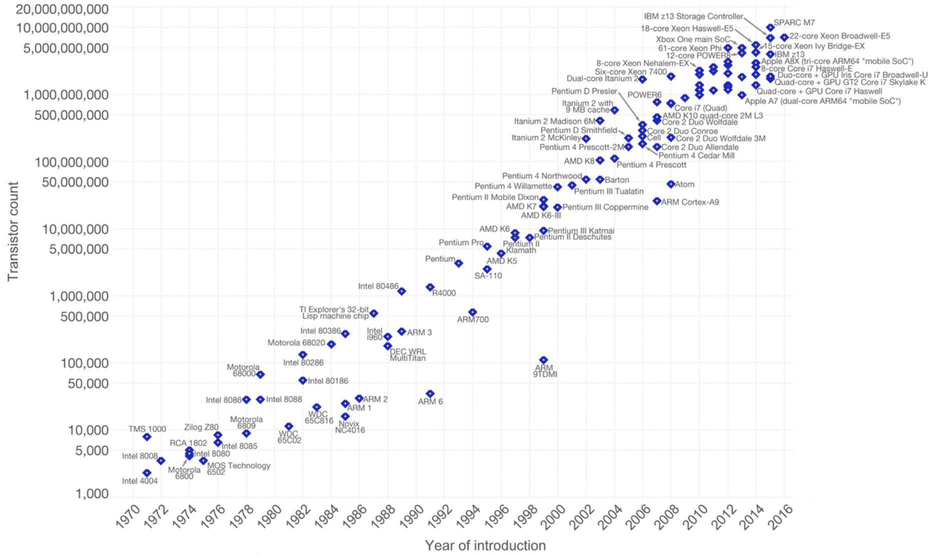
Gordon E. Moore (1929)

Observation by Gordon E. Moore:

The number of transistors on integrated circuits doubles approximately every two years.

# Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

ourworldindata.org, https://en.wikipedia.org/wiki/Transistor_count

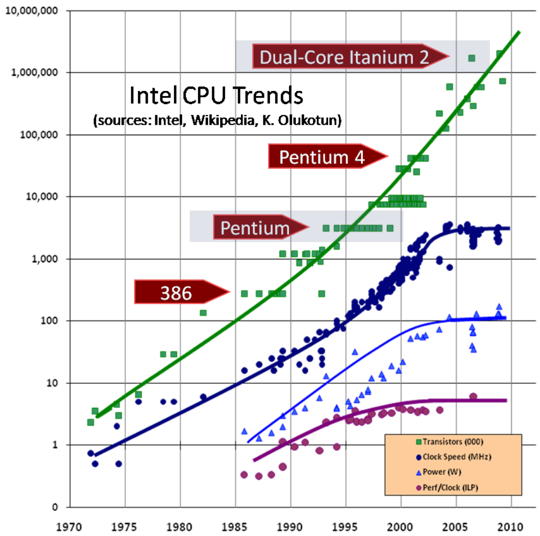
## For a long time...

- the sequential execution became faster ("Instruction Level Parallelism", "Pipelining", Higher Frequencies)
- more and smaller transistors = more performance
- programmers simply waited for the next processor generation

## Today

- the frequency of processors does not increase significantly and more (heat dissipation problems)
- the instruction level parallelism does not increase significantly any more
- the execution speed is dominated by memory access times (but caches still become larger and faster)

## Trends



http://www.gotw.ca/publications/concurrency-ddj.htm

## Multicore

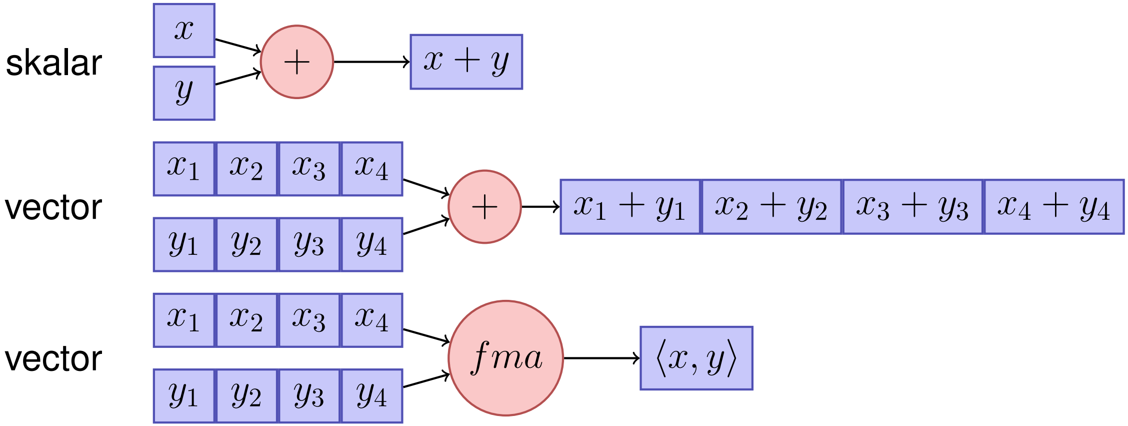
- Use transistors for more compute cores
- Parallelism in the software
- Programmers have to write parallel programs to benefit from new hardware

## Forms of Parallel Execution

- Vectorization
- Pipelining
- Instruction Level Parallelism
- Multicore / Multiprocessing
- Distributed Computing

## Vectorization

Parallel Execution of the same operations on elements of a vector (register)



## Pipelining in CPUs

Fetch | Decode | Execute | Data Fetch | Writeback

Multiple Stages

- Every instruction takes 5 time units (cycles)
- In the best case: 1 instruction per cycle, not always possible (“stalls”)

*Paralellism* (several functional units) leads to *faster execution.*

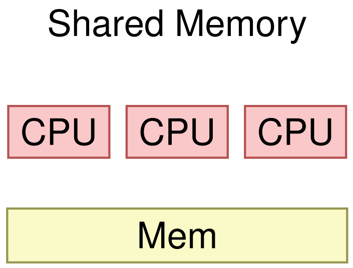
## ILP – Instruction Level Parallelism

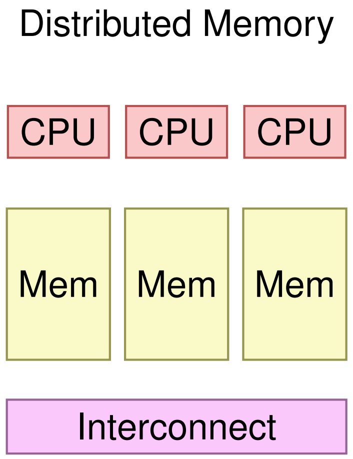
Modern CPUs provide several hardware units and execute independent instructions in parallel.

- Pipelining
- Superscalar CPUs (multiple instructions per cycle)
- Out-Of-Order Execution (Programmer observes the sequential execution)
- Speculative Execution ()

## 27.2 Hardware Architectures

## Shared vs. Distributed Memory





## Shared vs. Distributed Memory Programming

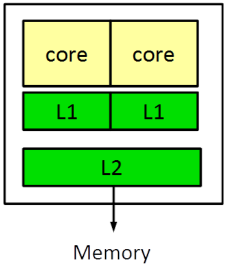
- Categories of programming interfaces
  - Communication via message passing
  - Communication via memory sharing
- It is possible:
  - to program shared memory systems as distributed systems (e.g. with message passing MPI)
  - program systems with distributed memory as shared memory systems (e.g. partitioned global address space PGAS)

## Shared Memory Architectures

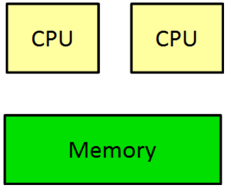
- Multicore (Chip Multiprocessor - CMP)
- Symmetric Multiprocessor Systems (SMP)
- Simultaneous Multithreading (SMT = Hyperthreading)
  - one physical core, Several Instruction Streams/Threads: several virtual cores
  - Between ILP (several units for a stream) and multicore (several units for several streams). Limited parallel performance.
- Non-Uniform Memory Access (NUMA)

Same programming interface

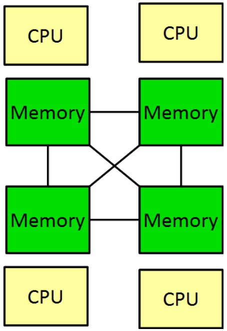
## Overview



CMP



SMP

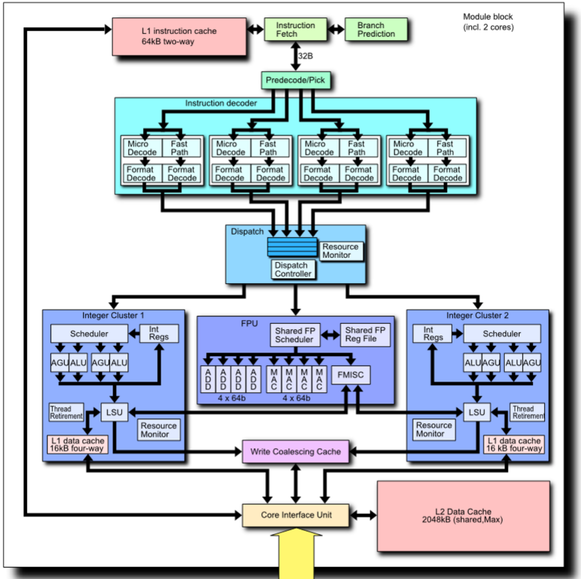


NUMA

## An Example

AMD Bulldozer: between CMP and SMT

- 2x integer core
- 1x floating point core

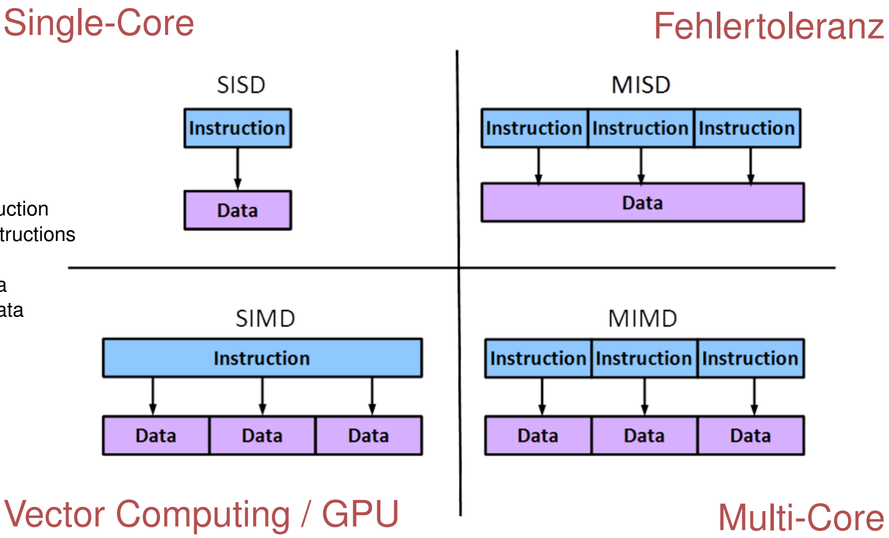


©Wikipedia

## Flynn’s Taxonomy

SI = Single Instruction
MI = Multiple Instructions

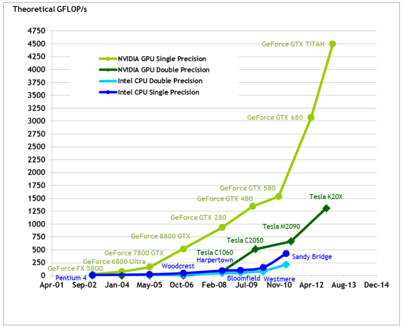
SD = Single Data
MD = Multiple Data



## Massively Parallel Hardware

[General Purpose] Graphical Processing Units ([GP]GPUs)

- Revolution in High Performance Computing
  - Calculation 4.5 TFlops vs. 500 GFlops
  - Memory Bandwidth 170 GB/s vs. 40 GB/s
- SIMD
  - High data parallelism
  - Requires own programming model. Z.B. CUDA / OpenCL



## 27.3 Multi-Threading, Parallelism and Concurrency

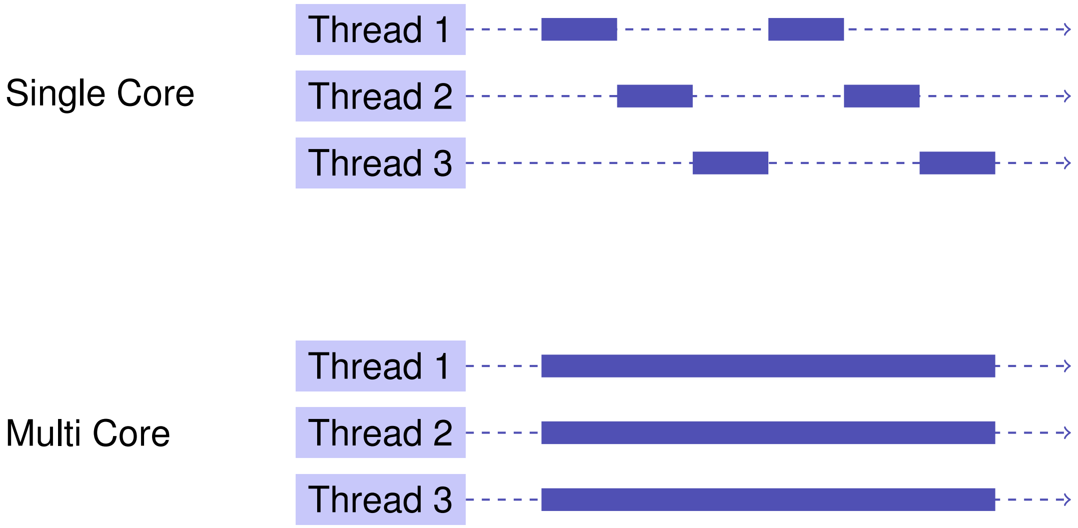
## Processes and Threads

- Process: instance of a program
  - each process has a separate context, even a separate address space
  - OS manages processes (resource control, scheduling, synchronisation)
- Threads: threads of execution of a program
  - Threads share the address space
  - fast context switch between threads

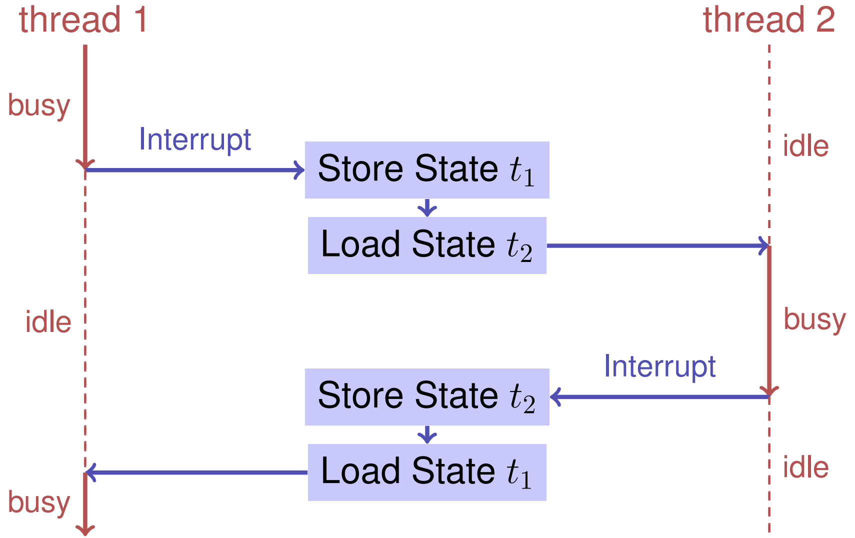
## Why Multithreading?

- Avoid “polling” resources (files, network, keyboard)
- Interactivity (e.g. responsivity of GUI programs)
- Several applications / clients in parallel
- Parallelism (performance!)

## Multithreading conceptually

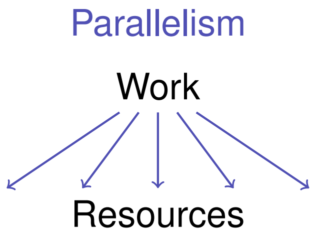


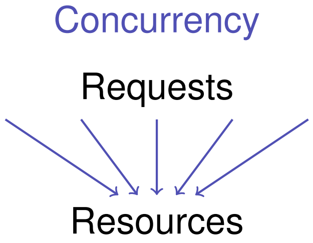
## Thread switch on one core (Preemption)



# Parallelität vs. Concurrency

- *Parallelism:* Use extra resources to solve a problem faster
- *Concurrency:* Correctly and efficiently manage access to shared resources
- Begriffe überlappen offensichtlich. Bei parallelen Berechnungen besteht fast immer Synchronisierungsbedarf.





## Thread Safety

Thread Safety means that in a concurrent application of a program this always yields the desired results.

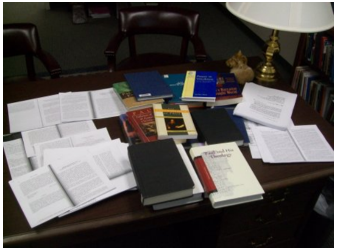
Many optimisations (Hardware, Compiler) target towards the correct execution of a *sequential* program.

Concurrent programs need an annotation that switches off certain optimisations selectively.

## Example: Caches

- Access to registers faster than to shared memory.
- Principle of locality.
- Use of Caches (transparent to the programmer)

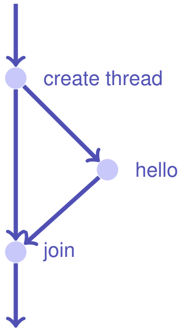
If and how far a cache coherency is guaranteed depends on the used system.



## 27.4 C++ Threads

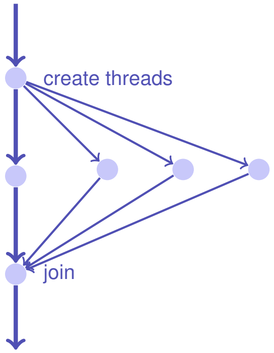
## C++11 Threads

```
#include <iostream>
#include <thread>

void hello(){
  std::cout << "hello\n";
}

int main(){
  // create and launch thread t
  std::thread t(hello);
  // wait for termination of t
  t.join();
  return 0;
}
```



## C++11 Threads

```
void hello(int id){
  std::cout << "hello from " << id << "\n";
}

int main(){
  std::vector<std::thread> tv(3);
  int id = 0;
  for (auto & t:tv)
    t = std::thread(hello, ++id);
  std::cout << "hello from main \n";
  for (auto & t:tv)
        t.join();
  return 0;
}
```



## Nondeterministic Execution!

One execution:

```
hello  from main
hello  from 2
hello  from 1
hello  from 0
```

Other execution:

```
hello  from 1
hello  from main
hello  from 0
hello  from 2
```

Other execution:

```
hello  from main
hello  from 0
hello  from hello  from 1
2
```

## Technical Detail

To let a thread continue as background thread:

```
void background();

void someFunction(){
  ...
  std::thread t(background);
  t.detach();
  ...
} // no problem here, thread is detached
```

## More Technical Details

- With allocating a thread, reference parameters are copied, except explicitly std::ref is provided at the construction.
- Can also run Functor or Lambda-Expression on a thread
- In exceptional circumstances, joining threads should be executed in a catch block

More background and details in chapter 2 of the book *C++ Concurrency in Action*, Anthony Williams, Manning 2012. also available online at the ETH library.

## 27.5 Scalability: Amdahl and Gustafson

## Scalability

In parallel Programming:

- Speedup when increasing number $p$ of processors
- What happens if $p \to \infty$?
- Program scales linearly: Linear speedup.

## Parallel Performance

Given a fixed amount of computing work $W$ (number computing steps)

Sequential execution time $T_1$

Parallel execution time on $p$ CPUs

- Perfection: $T_p = T_1/p$
- Performance loss: $T_p > T_1/p$ (usual case)
- Sorcery: $T_p < T_1/p$

## Parallel Speedup

Parallel speedup $S_p$ on $p$ CPUs:

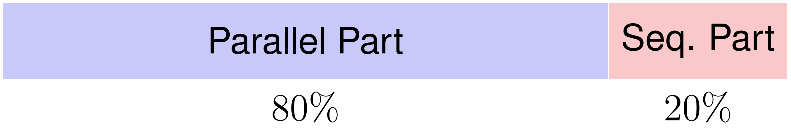
$$S_p = \frac{W/T_p}{W/T_1} = \frac{T_1}{T_p}.$$

- Perfection: linear speedup $S_p = p$
- Performance loss: sublinear speedup $S_p < p$ (the usual case)
- Sorcery: superlinear speedup $S_p > p$

Efficiency:$E_p = S_p/p$

## Reachable Speedup?

Parallel Program



$$T_1 = 10$$

$$T_8 = \frac{10 \cdot 0.8}{8} + 10 \cdot 0.2 = 1 + 2 = 3$$

$$S_8 = \frac{T_1}{T_8} = \frac{10}{3} \approx 3.3 < 8 \quad (!)$$

## Amdahl's Law: Ingredients

Computational work $W$ falls into two categories

- Paralellisable part $W_p$
- Not parallelisable, sequential part $W_s$

Assumption: $W$ can be processed sequentially by *one* processor in $W$ time units ($T_1 = W$):

$$T_1 = W_s + W_p$$
$$T_p \geq W_s + W_p/p$$

## Amdahl's Law

$$S_p = \frac{T_1}{T_p} \leq \frac{W_s + W_p}{W_s + \frac{W_p}{p}}$$

## Amdahl's Law

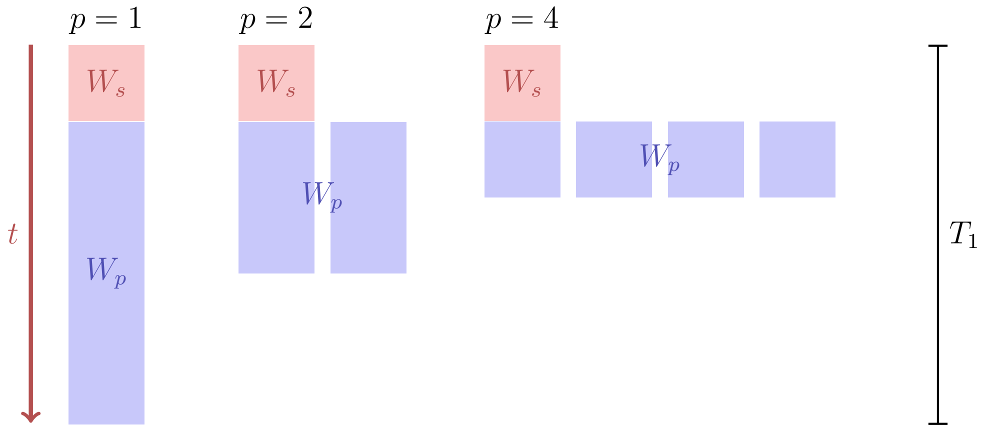
With sequential, not parallelizable fraction $\lambda$: $W_s = \lambda W$, $W_p = (1-\lambda)W$:

$$S_p \leq \frac{1}{\lambda + \frac{1-\lambda}{p}}$$

Thus

$$S_\infty \leq \frac{1}{\lambda}$$

# Illustration Amdahl's Law



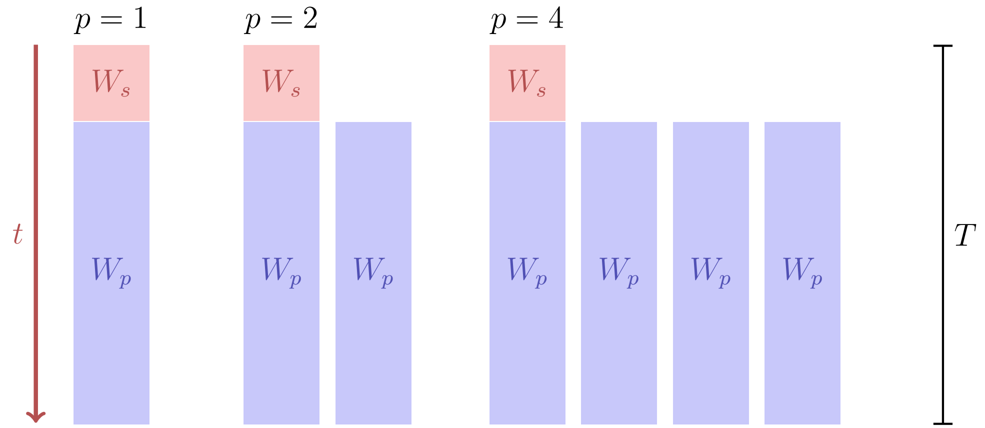
## Amdahl’s Law is bad news

All non-parallel parts of a program can cause problems

## Gustafson’s Law

- Fix the time of execution
- Vary the problem size.
- Assumption: the sequential part stays constant, the parallel part becomes larger

## Illustration Gustafson's Law



## Gustafson's Law

Work that can be executed by one processor in time $T$:

$$W_s + W_p = T$$

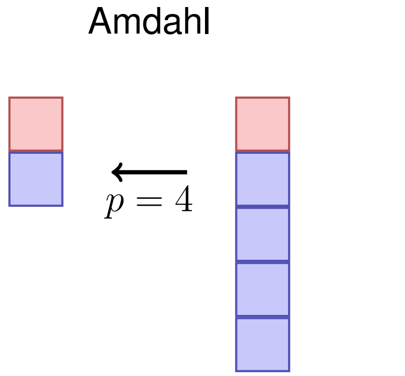
Work that can be executed by $p$ processors in time $T$:

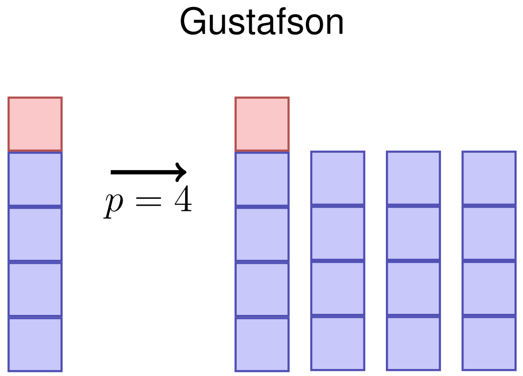
$$W_s + p \cdot W_p = \lambda \cdot T + p \cdot (1 - \lambda) \cdot T$$

Speedup:

$$\begin{aligned} S_p = \frac{W_s + p \cdot W_p}{W_s + W_p} &= p \cdot (1 - \lambda) + \lambda \\ &= p - \lambda(p - 1) \end{aligned}$$

## Amdahl vs. Gustafson





## Amdahl vs. Gustafson

The laws of Amdahl and Gustafson are models of speedup for parallelization.

Amdahl assumes a fixed *relative* sequential portion, Gustafson assumes a fixed *absolute* sequential part (that is expressed as portion of the work $W_1$ and that does not increase with increasing work).

The two models do not contradict each other but describe the runtime speedup of different problems and algorithms.

## 27.6 Task- and Data-Parallelism

## Parallel Programming Paradigms

- *Task Parallel:* Programmer explicitly defines parallel tasks.
- *Data Parallel:* Operations applied simulatenously to an aggregate of individual items.

## Example Data Parallel (OMP)

```
double sum = 0, A[MAX];
#pragma omp parallel for reduction (+:ave)
for (int i = 0; i< MAX; ++i)
  sum += A[i];
return sum;
```

## Example Task Parallel (C++11 Threads/Futures)

```
double sum(Iterator from, Iterator to)
{
  auto len = from - to;
  if (len > threshold){
    auto future = std::async(sum, from, from + len / 2);
    return sumS(from + len / 2, to) + future.get();
  }
  else
    return sumS(from, to);
}
```

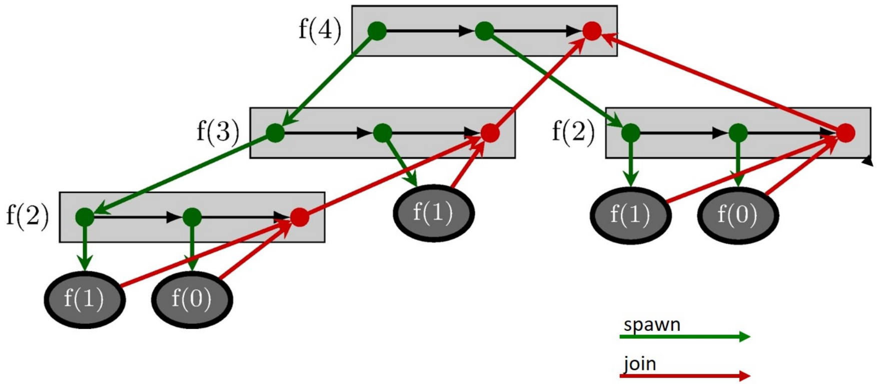
## Work Partitioning and Scheduling

- Partitioning of the work into parallel task (programmer or system)
  - One task provides a unit of work
  - Granularity?
- Scheduling (Runtime System)
  - Assignment of tasks to processors
  - Goal: full resource usage with little overhead

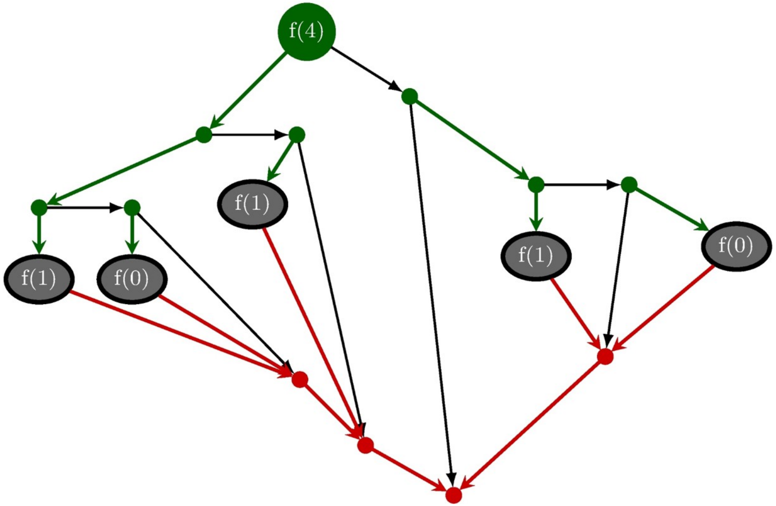
## Example: Fibonacci P-Fib

```
if $n \leq 1$ then
    return $n$
else
    $x \leftarrow$ spawn P-Fib$(n-1)$
    $y \leftarrow$ spawn P-Fib$(n-2)$
    sync
    return $x + y$;
```

## P-Fib Task Graph

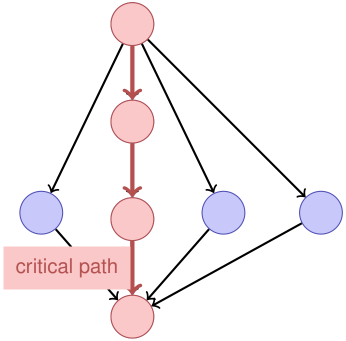


# P-Fib Task Graph



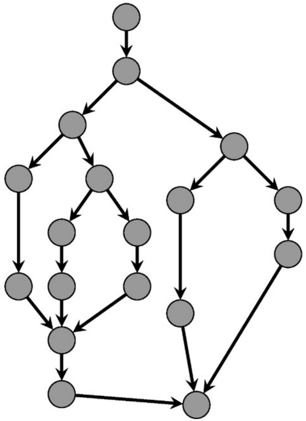
## Question

- Each Node (task) takes 1 time unit.
- Arrows depict dependencies.
- Minimal execution time when number of processors = $\infty$?



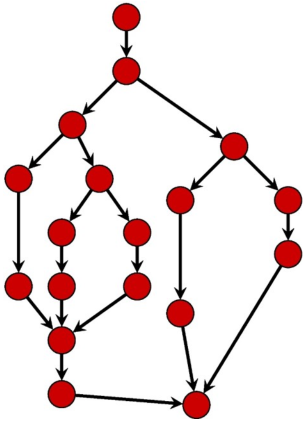
## Performance Model

- $p$ processors
- Dynamic scheduling
- $T_p$: Execution time on $p$ processors



## Performance Model

- $T_p$: Execution time on $p$ processors
- $T_1$: *work*: time for executing total work on one processor
- $T_1/T_p$: Speedup

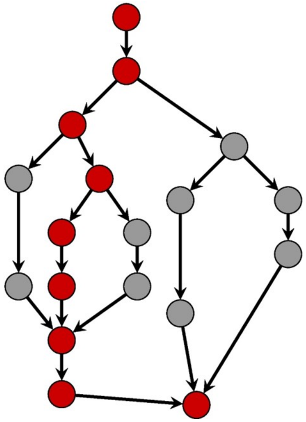


## Performance Model

- $T_\infty$: *span*: critical path, execution time on $\infty$ processors. Longest path from root to sink.
- $T_1/T_\infty$: *Parallelism:* wider is better
- Lower bounds:

$T_p \geq T_1/p$ Work law

$T_p \geq T_\infty$ Span law



## Greedy Scheduler

Greedy scheduler: at each time it schedules as many as availbale tasks.

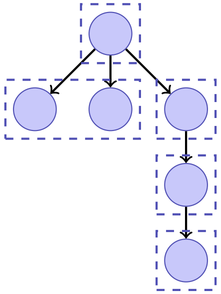
**Theorem**

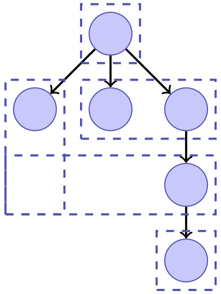
*On an ideal parallel computer with $p$ processors, a greedy scheduler executes a multi-threaded computation with work $T_1$ and span $T_\infty$ in time*

$$T_p \leq T_1/p + T_\infty$$

## Beispiel

Assume $p = 2$.





$T_p = 5$

$T_p = 4$

## Proof of the Theorem

Assume that all tasks provide the same amount of work.

- Complete step: $p$ tasks are available.
- incomplete step: less than $p$ steps available.

Assume that number of complete steps larger than $\lfloor T_1/p \rfloor$. Executed work $\geq \lfloor T_1/p \rfloor \cdot p + p = T_1 - T_1 \mod p + p > T_1$. Contradiction. Therefore maximally $\lfloor T_1/p \rfloor$ complete steps.

We now consider the graph of tasks to be done. Any maximal (critical) path starts with a node $t$ with $\deg^-(t) = 0$. An incomplete step executes all available tasks $t$ with $\deg^-(t) = 0$ and thus decreases the length of the span. Number incomplete steps thus limited by $T_\infty$.

## Consequence

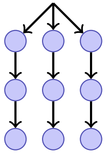
if $p \ll T_1/T_\infty$, i.e. $T_\infty \ll T_1/p$, then $T_p \approx T_1/p$.

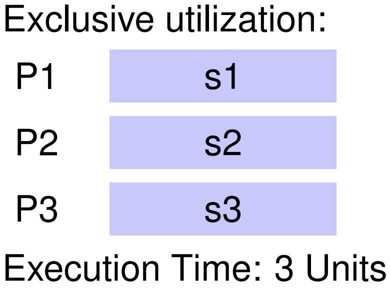
**Example Fibonacci**

$T_1(n)/T_\infty(n) = \Theta(\phi^n/n)$. For moderate sizes of $n$ we can use a lot of processors yielding linear speedup.

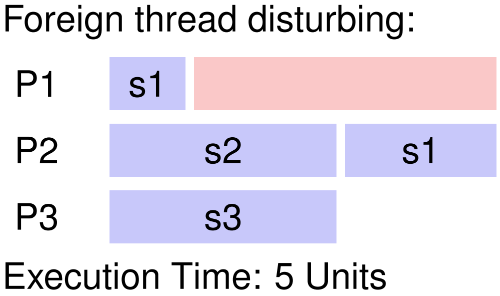
# Granularity: how many tasks?

- #Tasks = #Cores?
- Problem if a core cannot be fully used
- Example: 9 units of work. 3 core. Scheduling of 3 sequential tasks.




Exclusive utilization:

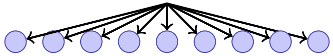
Execution Time: 3 Units


Foreign thread disturbing:

Execution Time: 5 Units

## Granularity: how many tasks?

- #Tasks = Maximum?
- Example: 9 units of work. 3 cores. Scheduling of 9 sequential tasks.



Exclusive utilization:

| | | | |
|---|---|---|---|
| P1 | s1 | s4 | s7 |
| P2 | s2 | s5 | s8 |
| P3 | s3 | s6 | s9 |

Execution Time: $3 + \varepsilon$ Units

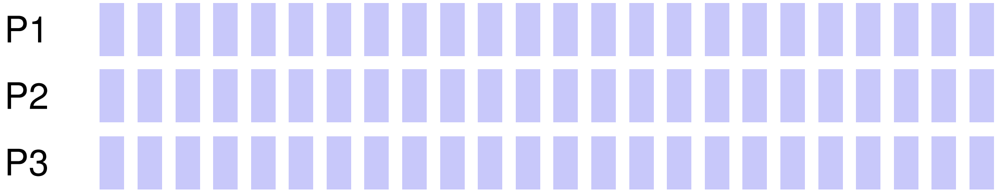
Foreign thread disturbing:

| | | | | |
|---|---|---|---|---|
| P1 | s1 | 🟥 | 🟥 | 🟥 |
| P2 | s2 | s4 | s5 | s8 |
| P3 | s3 | s6 | s7 | s9 |

Execution Time: 4 Units. Full utilization.

## Granularity: how many tasks?

- #Tasks = Maximum?
- Example: $10^6$ tiny units of work.



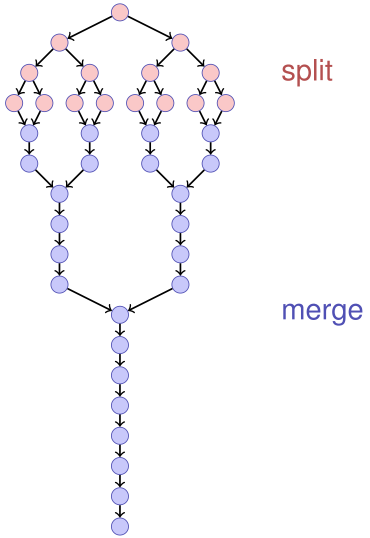
Execution time: dominiert vom Overhead.

## Granularity: how many tasks?

Answer: as many tasks as possible with a sequential cutoff such that the overhead can be neglected.

## Example: Parallelism of Mergesort

- Work (sequential runtime) of Mergesort $T_1(n) = \Theta(n \log n)$.
- Span $T_\infty(n) = \Theta(n)$
- Parallelism $T_1(n)/T_\infty(n) = \Theta(\log n)$ (Maximally achievable speedup with $p = \infty$ processors)



# 28. Parallel Programming II

Shared Memory, Concurrency, Excursion: lock algorithm (Peterson), Mutual Exclusion Race Conditions [C++ Threads: Williams, Kap. 2.1-2.2], [C++ Race Conditions: Williams, Kap. 3.1] [C++ Mutexes: Williams, Kap. 3.2.1, 3.3.3]

## 28.1 Shared Memory, Concurrency

## Sharing Resources (Memory)

- Up to now: fork-join algorithms: data parallel or divide-and-conquer
- Simple structure (data independence of the threads) to avoid race conditions
- Does not work any more when threads access shared memory.

## Managing state

Managing state: Main challenge of concurrent programming.

Approaches:

- Immutability, for example constants.
- Isolated Mutability, for example thread-local variables, stack.
- Shared mutable data, for example references to shared memory, global variables

## Protect the shared state

- Method 1: locks, guarantee exclusive access to shared data.
- Method 2: lock-free data structures, exclusive access with a much finer granularity.
- Method 3: transactional memory (not treated in class)

## Canonical Example

```
class BankAccount {
  int balance = 0;
public:
  int  getBalance(){ return balance; }
  void setBalance(int x) { balance = x; }
  void withdraw(int amount) {
    int b = getBalance();
    setBalance(b - amount);
  }
  // deposit etc.
};
```

(correct in a single-threaded world)

## Bad Interleaving

Parallel call to `widthdraw(100)` on the same account

Thread 1

```
int b = getBalance();



setBalance(b-amount);
```

Thread 2

```

int b = getBalance();

setBalance(b-amount);
```

$t \downarrow$

## Tempting Traps

WRONG:

```
void withdraw(int amount) {
  int b = getBalance();
  if (b==getBalance())
       setBalance(b - amount);
}
```

Bad interleavings cannot be solved with a repeated reading

## Tempting Traps

also WRONG:

```
void withdraw(int amount) {
        setBalance(getBalance() - amount);
}
```

Assumptions about atomicity of operations are almost always wrong

## Mutual Exclusion

We need a concept for mutual exclusion

*Only one thread* may execute the operation withdraw *on the same account* at a time.

The programmer has to make sure that mutual exclusion is used.

## More Tempting Traps

```
class BankAccount {
  int balance = 0;
  bool busy = false;
public:
  void withdraw(int amount) {
    while (busy); // spin wait
    busy = true;
    int b = getBalance();
    setBalance(b - amount);
    busy = false;
  }

  // deposit would spin on the same boolean
};
```

does not work!

## Just moved the problem!

Thread 1                              Thread 2

```
while (busy); //spin
                                  while (busy); //spin
busy = true;
                                  busy = true;
int b = getBalance();
                                  int b = getBalance();
                                  setBalance(b - amount);
setBalance(b - amount);
```

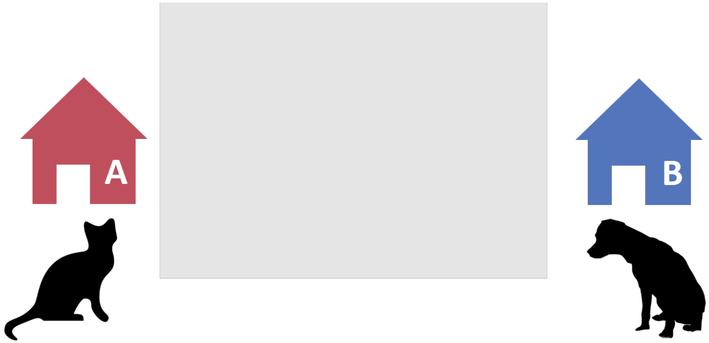
$t$

## How ist this correctly implemented?

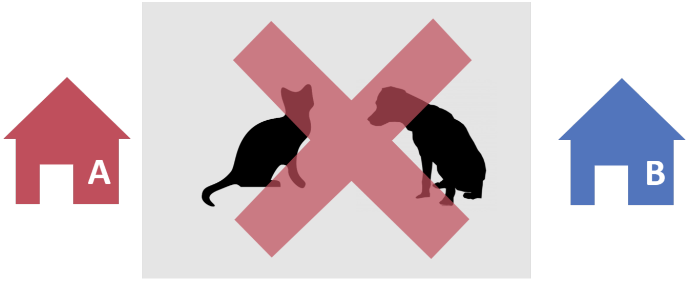
- We use *locks* (mutexes) from libraries
- They use hardware primitives, *Read-Modify-Write* (RMW) operations that can, in an atomic way, read and write depending on the read result.
- Without RMW Operations the algorithm is non-trivial and requires at least atomic access to variable of primitive type.

## 28.2 Excursion: lock algorithm

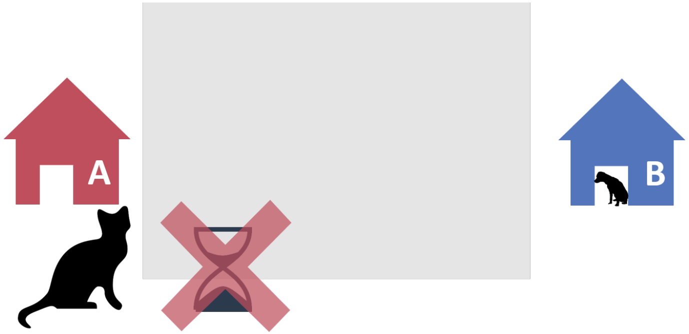
## Alice’s Cat vs. Bob’s Dog



## Required: Mutual Exclusion

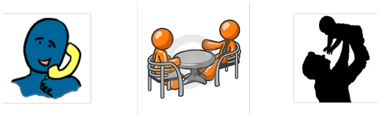


## Required: No Lockout When Free

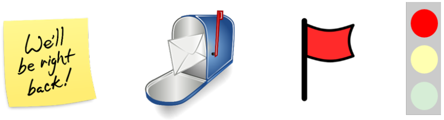


## Communication Types

- Transient: Parties participate at the same time

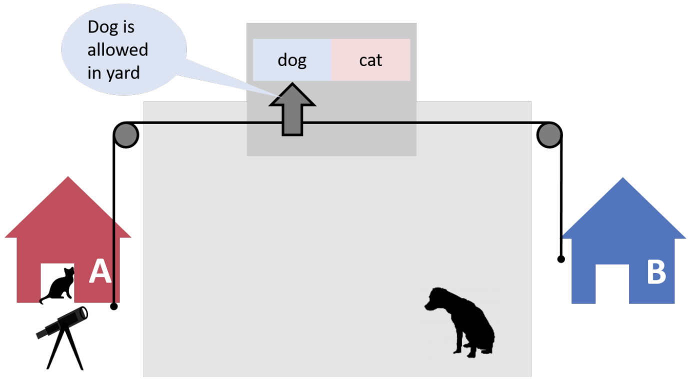


- Persistent: Parties participate at different times

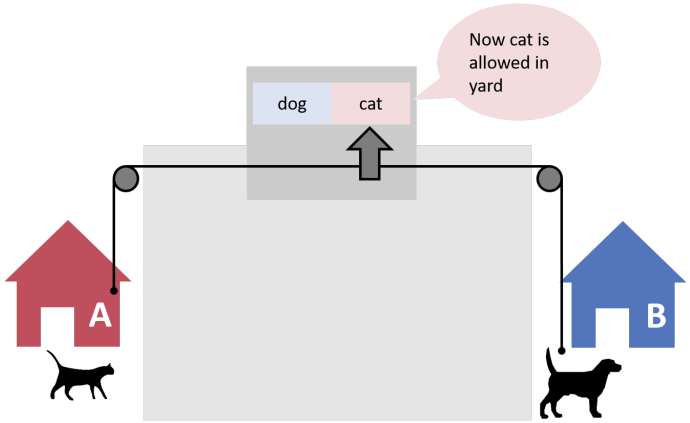


Mutual exclusion: persistent communication

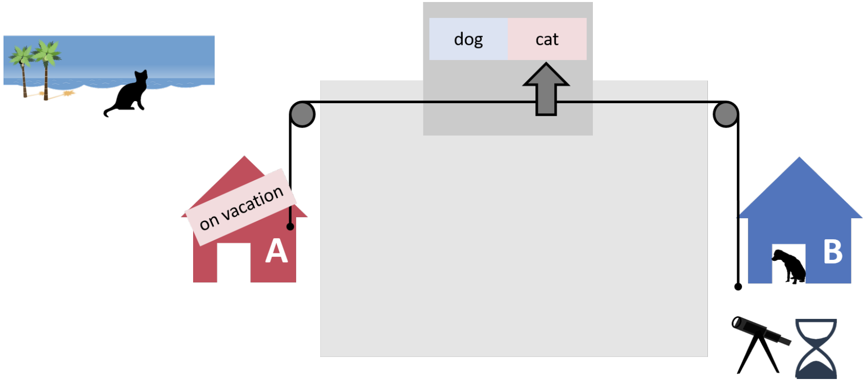
# Communication Idea 1



# Access Protocol



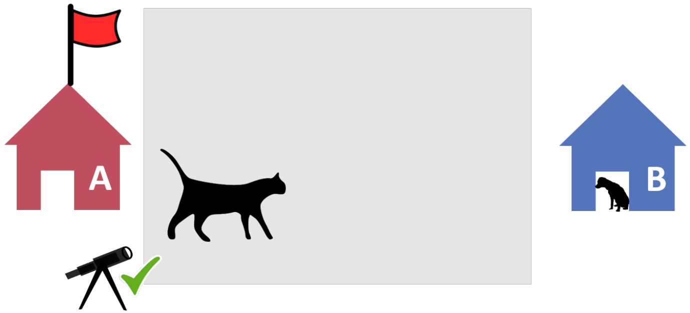
## Problem!



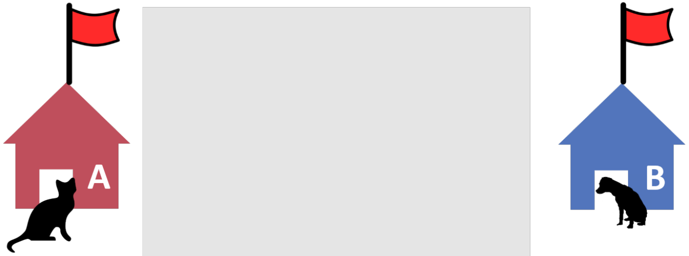
## Communication Idea 2

My pet is in the yard
A
B

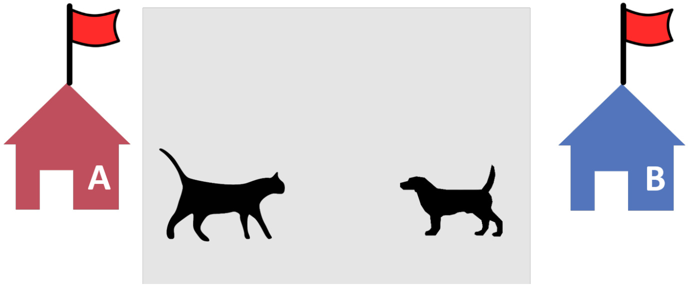
## Access Protocol 2.1



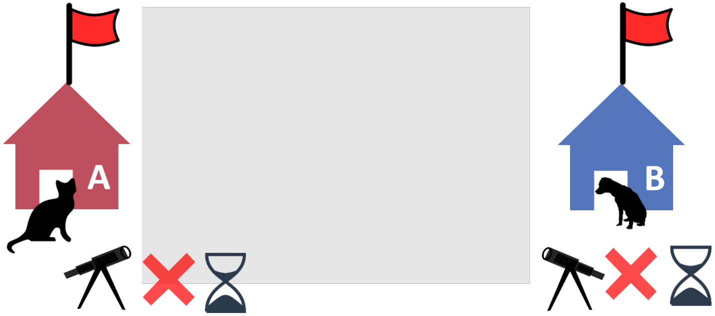
## Different Scenario



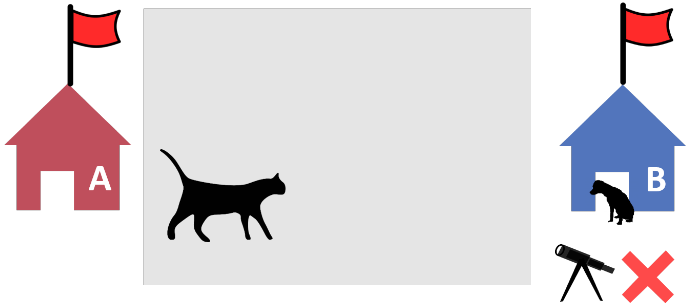
## Problem: No Mutual Exclusion



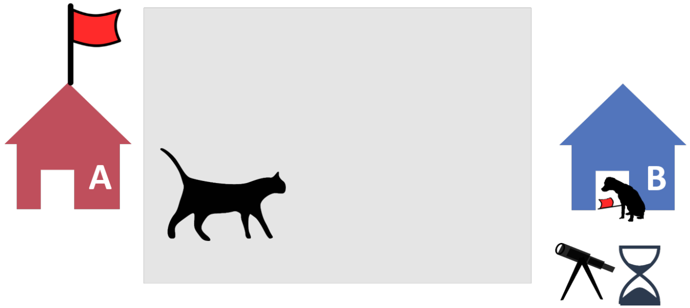
## Checking Flags Twice: Deadlock



## Access Protocol 2.2

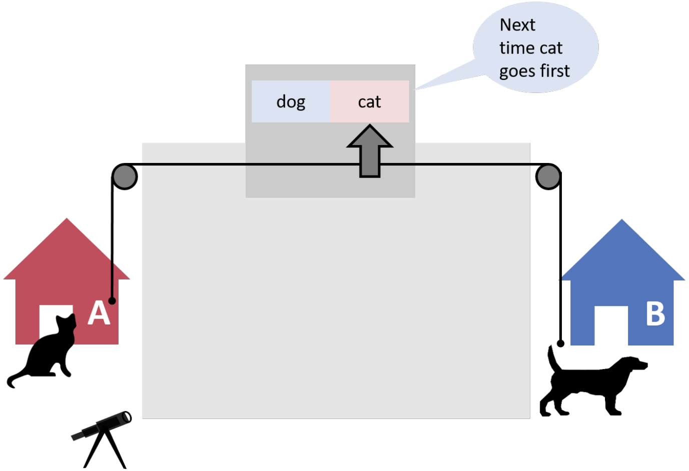


## Access Protocol 2.2:provably correct



## Weniger schwerwiegend: Starvation

## Final Solution



## General Problem of Locking remains

## Peterson's Algorithm[54]

for two processes is provable correct and free from starvation

```
non-critical section

flag[me] = true // I am interested
victim = me // but you go first
// spin while we are both interested and you go first:
while (flag[you] && victim == me) {};

critical section

flag[me] = false
```

The code assumes that the access to flag / victim is atomic and particularly linearizable or sequential consistent. An assumption that – as we will see below – is not necessarily given for normal variables. The Peterson-lock is not used on modern hardware.

[54] not relevant for the exam

## 28.3 Mutual Exclusion

## Critical Sections and Mutual Exclusion

*Critical Section*

Piece of code that may be executed by at most one process (thread) at a time.

*Mutual Exclusion*

Algorithm to implement a critical section

```
acquire_mutex();      // entry algorithm\\
 ...                  //  critical  section
release_mutex();      // exit algorithm
```

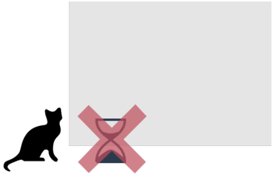
## Required Properties of Mutual Exclusion

Correctness (Safety)

- At most one process executes the critical section code

Liveness

- Acquiring the mutex must terminate in finite time when no process executes in the critical section



## Almost Correct

```
class BankAccount {
  int balance = 0;
  std::mutex m; // requires #include <mutex>
public:
  ...
  void withdraw(int amount) {
    m.lock();
    int b = getBalance();
    setBalance(b - amount);
    m.unlock();
  }
};
```

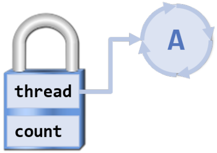
What if an exception occurs?

## RAII Approach

```
class BankAccount {
  int balance = 0;
  std::mutex m;
public:
  ...
  void withdraw(int amount) {
    std::lock_guard<std::mutex> guard(m);
    int b = getBalance();
    setBalance(b - amount);
  } // Destruction of guard leads to unlocking m
};
```

What about getBalance / setBalance?

## Reentrant Locks



Reentrant Lock (recursive lock)

- remembers the currently affected thread;
- provides a counter
  - Call of lock: counter incremented
  - Call of unlock: counter is decremented. If counter = 0 the lock is released.

## Account with reentrant lock

```
class BankAccount {
  int balance = 0;
  std::recursive_mutex m;
  using guard = std::lock_guard<std::recursive_mutex>;
public:
  int  getBalance(){ guard g(m); return balance;
  }
  void setBalance(int x) { guard g(m); balance = x;
  }
  void withdraw(int amount) { guard g(m);
    int b = getBalance();
    setBalance(b - amount);
  }
};
```

## 28.4 Race Conditions

## Race Condition

- A *race condition* occurs when the result of a computation depends on scheduling.
- We make a distinction between *bad interleavings* and *data races*
- *Bad interleavings* can occur even when a mutex is used.

## Example: Stack

Stack with correctly synchronized access:

```
template <typename T>
class stack{
  ...
  std::recursive_mutex m;
  using guard = std::lock_guard<std::recursive_mutex>;
public:
  bool isEmpty(){ guard g(m); ...   }
  void push(T value){ guard g(m);   ... }
  T pop(){ guard g(m); ...}
};
```

## Peek

Forgot to implement peek. Like this?

```
template <typename T>
T peek (stack<T> &s){
  T value = s.pop();
  s.push(value);
  return value;
}
```

not thread-safe!

Despite its questionable style the code is correct in a sequential world. Not so in concurrent programming.

## Bad Interleaving!

Initially empty stack $s$, only shared between threads 1 and 2.

Thread 1 pushes a value and checks that the stack is then non-empty. Thread 2 reads the topmost value using peek().

```
     Thread 1                         Thread 2

     s.push(5);
                                      int value = s.pop();
 t   assert(!s.isEmpty());
                                      s.push(value);
                                      return value;
```

## The fix

Peek must be protected with the same lock as the other access methods

## Bad Interleavings

Race conditions as bad interleavings can happen on a high level of abstraction

In the following we consider a different form of race condition: data race.

## How about this?

```
class counter{
  int count = 0;
  std::recursive_mutex m;
  using guard = std::lock_guard<std::recursive_mutex>;
public:
  int increase(){
    guard g(m); return ++count;
  }
  int get(){
    return count;
  }
}
```

not thread-safe!

## Why wrong?

It looks like nothing can go wrong because the update of count happens in a “tiny step”.

But this code is still wrong and depends on language-implementation details you cannot assume.

This problem is called *Data-Race*

Moral: *Do not introduce a data race, even if every interleaving you can think of is correct. Don’t make assumptions on the memory order.*

## A bit more formal

*Data Race* (low-level Race-Conditions) Erroneous program behavior caused by insufficiently synchronized accesses of a shared resource by multiple threads, e.g. Simultaneous read/write or write/write of the same memory location

*Bad Interleaving* (High Level Race Condition) Erroneous program behavior caused by an unfavorable execution order of a multithreaded algorithm, even if that makes use of otherwise well synchronized resources.

# We look deeper

```
 class C {
   int x = 0;
   int y = 0;
 public:
   void f() {
Ⓐ    x = 1;
Ⓑ    y = 1;
   }
   void g() {
Ⓒ    int a = y;
Ⓓ    int b = x;
     assert(b >= a);
   }
 }
```

Can this fail?

There is no interleaving of f and g that would cause the assertion to fail:

- A B C D ✓
- A C B D ✓
- A C D B ✓
- C A B D ✓
- C C D B ✓
- C D A B ✓

**It can nevertheless fail!**

## One Resason: Memory Reordering

*Rule of thumb:* Compiler and hardware allowed to make changes that do not affect the *semantics of a sequentially* executed program

```
void f() {
       x = 1;
       y = x+1;
       z = x+1;
}
```

$\Longleftrightarrow$
sequentially equivalent

```
void f() {
       x = 1;
       z = x+1;
       y = x+1;
}
```

## From a Software-Perspective

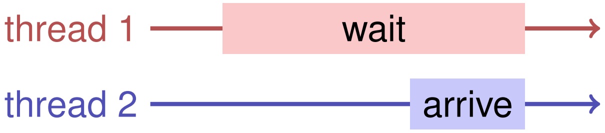
Modern compilers do not give guarantees that a global ordering of memory accesses is provided as in the sourcecode:

- Some memory accesses may be even optimized away completely!
- Huge potential for optimizations – and for errors, when you make the wrong assumptions

## Example: Self-made Rendevouz

```
int x; // shared

void wait(){
  x = 1;
  while(x == 1);
}

void arrive(){
  x = 2;
}
```

Assume thread 1 calls wait, later thread 2 calls arrive. What happens?



## Compilation

### Source

```
int x; // shared

void wait(){
  x = 1;
  while(x == 1);
}



void arrive(){
  x = 2;
}
```

### Without optimisation

```
wait:
movl $0x1, x
test:
mov x, %eax
cmp $0x1, %eax
je  test

arrive:
movl $0x2, x
```

if equal

### With optimisation

```
wait:
movl $0x1, x
test:
jmp test

arrive
movl $0x2, x
```

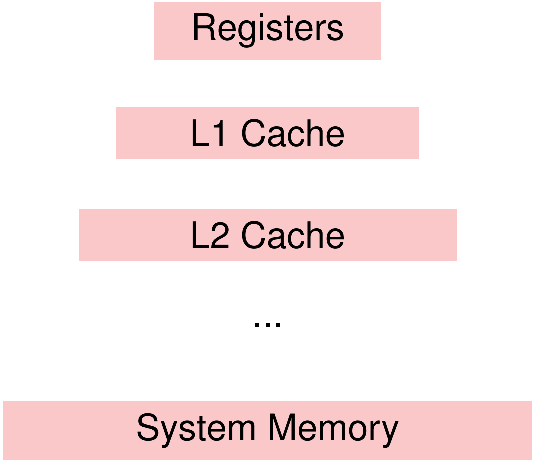
always

## Hardware Perspective

Modern multiprocessors do not enforce global ordering of all instructions for performance reasons:

- Most processors have a pipelined architecture and can execute (parts of) multiple instructions simultaneously. They can even reorder instructions internally.
- Each processor has a local cache, and thus loads/stores to shared memory can become visible to other processors at different times

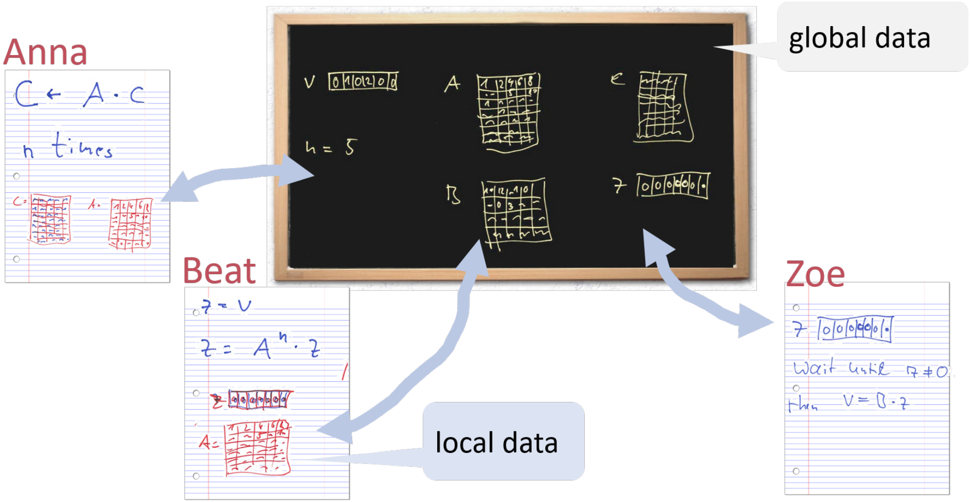
## Memory Hierarchy



fast,low latency, high cost, low capacity

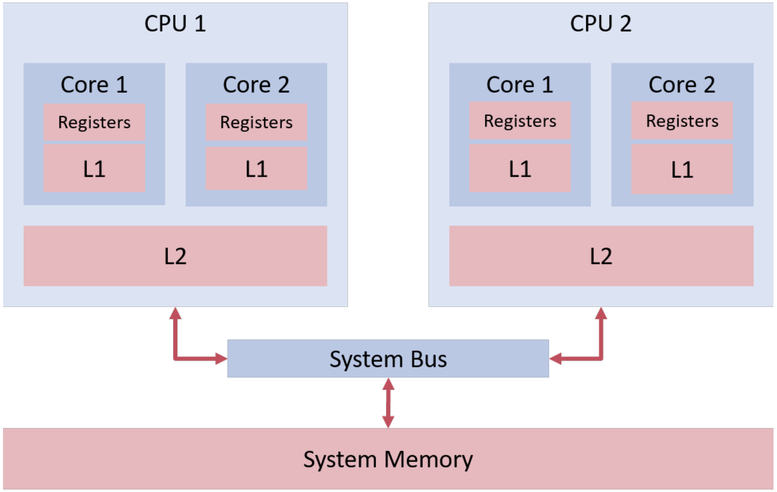
slow,high latency,low cost,high capacity

# An Analogy



Anna
C ← A · c
n times
C=
A=
Beat
z = v
z = A^n · z
z
A=
v
n = 5
A
B
c
z
global data
local data
Zoe
z
wait until z ≠ 0
then v = B · z

## Schematic


CPU 1
Core 1
Registers
L1
Core 2
Registers
L1
L2
CPU 2
Core 1
Registers
L1
Core 2
Registers
L1
L2
System Bus
System Memory

## Memory Models

When and if effects of memory operations become visible for threads, depends on hardware, runtime system and programming language.

A *memory model* (e.g. that of C++) provides minimal guarantees for the effect of memory operations

- leaving open possibilities for optimisation
- containing guidelines for writing thread-safe programs

For instance, C++ provides *guarantees when synchronisation with a mutex* is used.

## Fixed

```
class C {
  int x = 0;
  int y = 0;
  std::mutex m;
public:
  void f() {
    m.lock(); x = 1; m.unlock();
    m.lock(); y = 1; m.unlock();
  }
  void g() {
    m.lock(); int a = y; m.unlock();
    m.lock(); int b = x; m.unlock();
    assert(b >= a); // cannot fail
  }
};
```

## Atomic

Here also possible:

```
class C {
  std::atomic_int x{0}; // requires #include <atomic>
  std::atomic_int y{0};
public:
  void f() {
    x = 1;
    y = 1;
  }
  void g() {
    int a = y;
    int b = x;
    assert(b >= a); // cannot fail
  }
};
```

# 29. Parallel Programming III

Deadlock and Starvation Producer-Consumer, The concept of the monitor, Condition Variables [Deadlocks : Williams, Kap. 3.2.4-3.2.5] [Condition Variables: Williams, Kap. 4.1]

## Deadlock Motivation

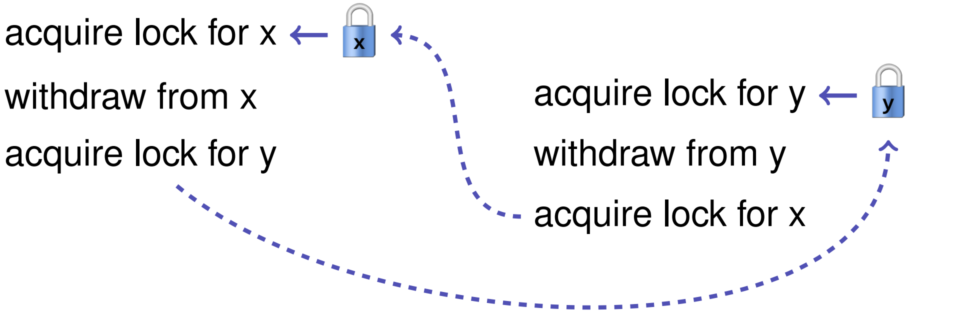
```
class BankAccount {
  int balance = 0;
  std::recursive_mutex m;
  using guard = std::lock_guard<std::recursive_mutex>;
public:
  ...
  void withdraw(int amount) { guard g(m); ... }
  void deposit(int amount){ guard g(m); ... }

  void transfer(int amount, BankAccount& to){
      guard g(m);
      withdraw(amount);
      to.deposit(amount);
  }
};
```

Problem?

## Deadlock Motivation

Suppose BankAccount instances `x` and `y`

```
Thread 1: x.transfer(1,y);
```



acquire lock for x ← x

withdraw from x

acquire lock for y

```
Thread 2: y.transfer(1,x);
```

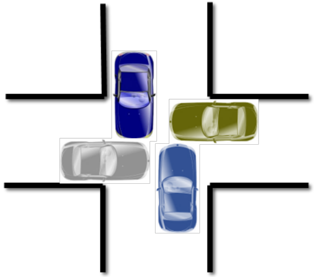
acquire lock for y ← y

withdraw from y

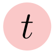
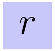
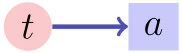
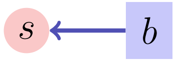
acquire lock for x

## Deadlock

*Deadlock:* two or more processes are mutually blocked because each process waits for another of these processes to proceed.

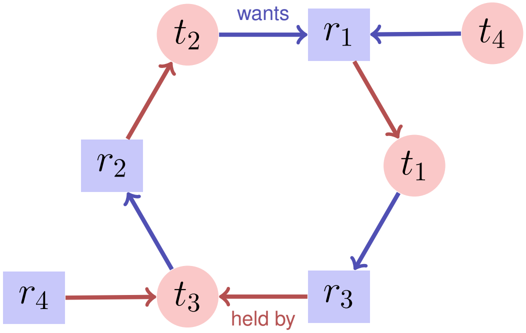


## Threads and Resources

- Grafically  and Resources (Locks) 
- Thread $t$ attempts to acquire resource $a$: 
- Resource $b$ is held by thread $q$: 

## Deadlock – Detection

A deadlock for threads $t_1, \dots, t_n$ occurs when the graph describing the relation of the $n$ threads and resources $r_1, \dots, r_m$ contains a cycle.



## Techniques

- *Deadlock detection* detects cycles in the dependency graph. Deadlocks can in general not be healed: releasing locks generally leads to inconsistent state
- *Deadlock avoidance* amounts to techniques to ensure a cycle can never arise
  - Coarser granularity “one lock for all”
  - Two-phase locking with retry mechanism
  - Lock Hierarchies
  - ...
  - Resource Ordering

## Back to the Example

```
class BankAccount {
  int id; // account number, also used for locking order
  std::recursive_mutex m; ...
public:
  ...
   void transfer(int amount, BankAccount& to){
      if (id < to.id){
        guard g(m); guard h(to.m);
        withdraw(amount); to.deposit(amount);
      } else {
        guard g(to.m); guard h(m);
        withdraw(amount); to.deposit(amount);
      }
  }
};
```

## C++11 Style

```
class BankAccount {
  ...
  std::recursive_mutex m;
  using guard = std::lock_guard<std::recursive_mutex>;
public:
  ...
   void transfer(int amount, BankAccount& to){
      std::lock(m,to.m); // lock order done by C++
      // tell the guards that the lock is already taken:
      guard g(m,std::adopt_lock); guard h(to.m,std::adopt_lock);
      withdraw(amount);
      to.deposit(amount);
  }
};
```

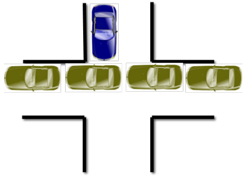
# By the way...

```
class BankAccount {
  int balance = 0;
  std::recursive_mutex m;
  using guard = std::lock_guard<std::recursive_mutex>;
public:
  ...
  void withdraw(int amount) { guard g(m); ... }
  void deposit(int amount){ guard g(m); ... }

  void transfer(int amount, BankAccount& to){
      withdraw(amount);
      to.deposit(amount);
  }
};
```

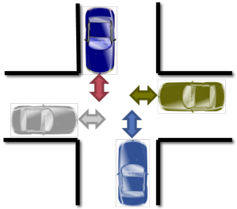
This would have worked here also. But then for a very short amount of time, money disappears, which does not seem acceptable (transient incon-

## Starvation und Livelock

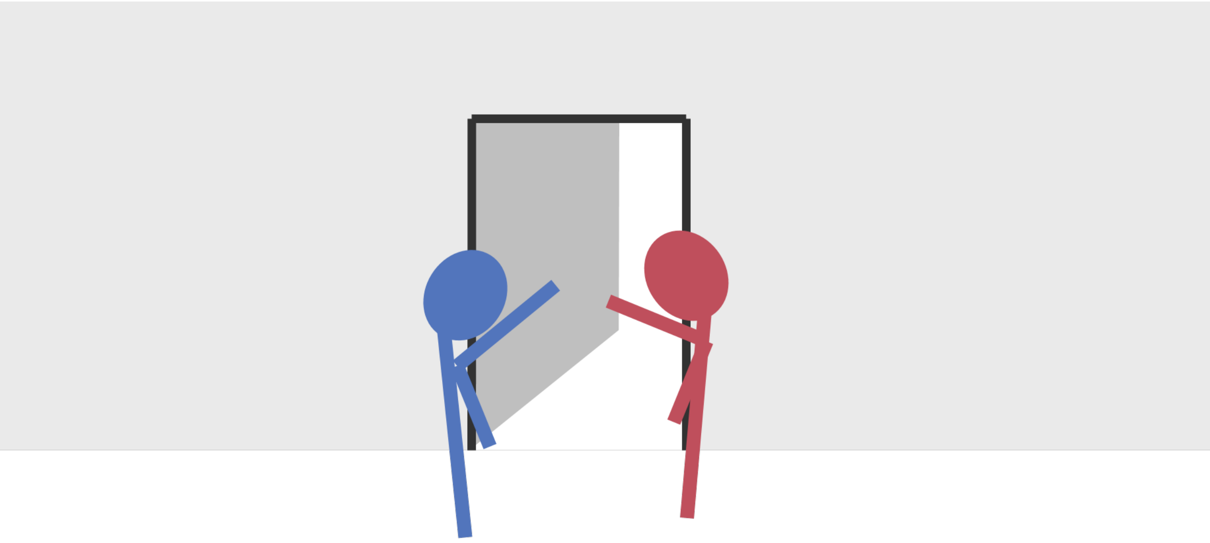
*Starvation:* the repeated but unsuccessful attempt to acquire a resource that was recently (transiently) free.



*Livelock:* competing processes are able to detect a potential deadlock but make no progress while trying to resolve it.



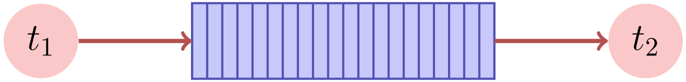
## Politelock



## Producer-Consumer Problem

Two (or more) processes, producers and consumers of data should become decoupled by some data structure.

Fundamental Data structure for building pipelines in software.



# Sequential implementation (unbounded buffer)

```
class BufferS {
  std::queue<int> buf;
public:
    void put(int x){
        buf.push(x);
    }

    int get(){
        while (buf.empty()){} // wait until data arrive
        int x = buf.front();
        buf.pop();
        return x;
    }
};
```

not thread-safe

## How about this?

```
class Buffer {
  std::recursive_mutex m;
  using guard = std::lock_guard<std::recursive_mutex>;
  std::queue<int> buf;
public:
    void put(int x){ guard g(m);
        buf.push(x);
    }
    int get(){ guard g(m);
        while (buf.empty()){}
        int x = buf.front();
        buf.pop();
        return x;
    }
};
```

Deadlock

## Well, then this?

```
void put(int x){
    guard g(m);
    buf.push(x);
}
int get(){
    m.lock();
    while (buf.empty()){
        m.unlock();
        m.lock();
    }
    int x = buf.front();
    buf.pop();
    m.unlock();
    return x;
}
```

Ok this works, but it wastes CPU time.

## Better?

```
void put(int x){
  guard g(m);
  buf.push(x);
}
int get(){
  m.lock();
  while (buf.empty()){
    m.unlock();
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
    m.lock();
  }
  int x = buf.front(); buf.pop();
  m.unlock();
  return x;
}
```

Ok a little bit better, limits reactivity though.

## Moral

We do not want to implement waiting on a condition ourselves.

There already is a mechanism for this: *condition variables*.

The underlying concept is called *Monitor*.

## Monitor

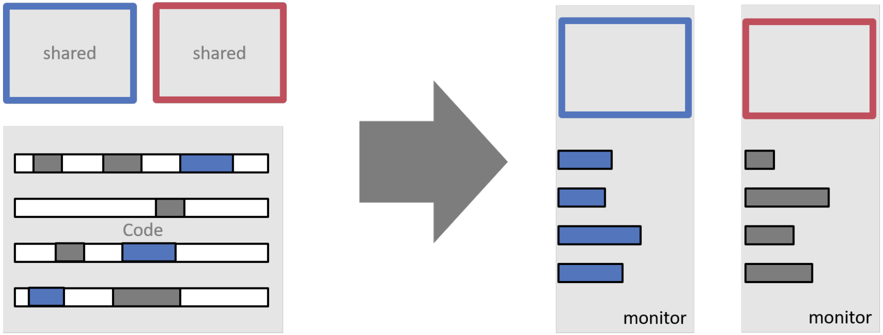
*Monitor* abstract data structure equipped with a set of operations that run in mutual exclusion and that can be synchronized.

Invented by C.A.R. Hoare and Per Brinch Hansen (cf. Monitors – An Operating System Structuring Concept, C.A.R. Hoare 1974)

C.A.R. Hoare, *1934

Per Brinch Hansen (1938-2007)

# Monitors vs. Locks



## Monitor and Conditions

Monitors provide, in addition to mutual exclusion, the following mechanism:

*Waiting on conditions:* If a condition does not hold, then

- Release the monitor lock
- Wait for the condition to become true
- Check the condition when a signal is raised

*Signalling:* Thread that might make the condition true:

- Send signal to potentially waiting threads

## Condition Variables

```
#include <mutex>
#include <condition_variable>
...

class Buffer {
  std::queue<int> buf;

  std::mutex m;
  // need unique_lock guard for conditions
  using guard = std::unique_lock<std::mutex>;
  std::condition_variable cond;
public:
       ...
};
```

## Condition Variables

```
class Buffer {
...
public:
    void put(int x){
        guard g(m);
        buf.push(x);
        cond.notify_one();
    }
    int get(){
        guard g(m);
        cond.wait(g, [&]{return !buf.empty();});
        int x = buf.front(); buf.pop();
        return x;
    }
};
```

## Technical Details

- A thread that waits using `cond.wait` runs at most for a short time on a core. After that it does not utilize compute power and “sleeps”.
- The notify (or signal-) mechanism wakes up sleeping threads that subsequently check their conditions.
  - `cond.notify_one` signals *one* waiting thread
  - `cond.notify_all` signals *all* waiting threads. Required when waiting thrads wait potentially on *different* conditions.

## Technical Details

- Many other programming langauges offer the same kind of mechanism. The checking of conditions (in a loop!) has to be usually implemented by the programmer.

### Java Example

```
synchronized long get() {
  long x;
  while (isEmpty())
    try {
      wait ();
      } catch (InterruptedException e) { }
  x = doGet();
  return x;
}

synchronized put(long x){
  doPut(x);
  notify ();
}
```

## By the way, using a bounded buffer..

```
class Buffer {
  ...
  CircularBuffer<int,128> buf; // from lecture 6
public:
    void put(int x){ guard g(m);
        cond.wait(g, [&]{return !buf.full();});
        buf.put(x);
        cond.notify_all();
    }
    int get(){ guard g(m);
        cond.wait(g, [&]{return !buf.empty();});
        cond.notify_all();
        return buf.get();
    }
};
```

# 30. Parallel Programming IV

Futures, Read-Modify-Write Instructions, Atomic Variables, Idea of lock-free programming

[C++ Futures: Williams, Kap. 4.2.1-4.2.3] [C++ Atomic: Williams, Kap. 5.2.1-5.2.4, 5.2.7] [C++ Lockfree: Williams, Kap. 7.1.-7.2.1]

## Futures: Motivation

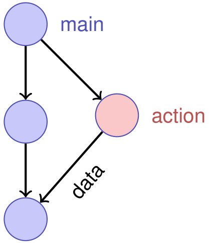
Up to this point, threads have been functions without a result:

```
void action(some parameters){
  ...
}

std::thread t(action, parameters);
...
t.join();
// potentially read result written via ref-parameters
```

## Futures: Motivation

Now we would like to have the following

```
T action(some parameters){
  ...
  return value;
}

std::thread t(action, parameters);
...
value = get_value_from_thread();
```



## We can do this already!

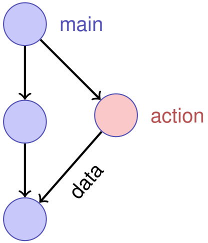
- We make use of the producer/consumer pattern, implemented with condition variables
- Start the thread with reference to a buffer
- We get the result from the buffer.
- Synchronisation is already implemented

## Reminder

```
template <typename T>
class Buffer {
  std::queue<T> buf;
  std::mutex m;
  std::condition_variable cond;
public:
  void put(T x){ std::unique_lock<std::mutex> g(m);
    buf.push(x);
    cond.notify_one();
  }
  T get(){ std::unique_lock<std::mutex> g(m);
    cond.wait(g, [&]{return (!buf.empty());});
    T x = buf.front(); buf.pop(); return x;
  }
};
```

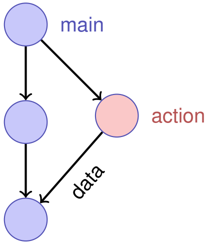
## Application

```
void action(Buffer<int>& c){
  // some long lasting operation ...
  c.put(42);
}

int main(){
  Buffer<int> c;
  std::thread t(action, std::ref(c));
  t.detach(); // no join required for free running thread
  // can do some more work here in parallel
  int val = c.get();
  // use result
  return 0;
}
```



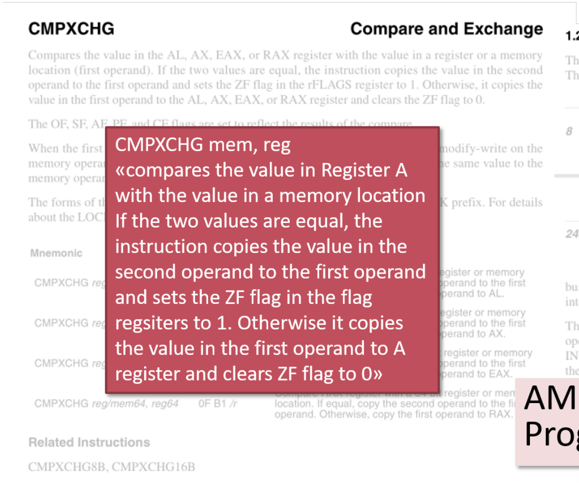
## With features of C++11

```
int action(){
  // some long lasting operation
  return 42;
}

int main(){
  std::future<int> f = std::async(action);
  // can do some work here in parallel
  int val = f.get();
  // use result
  return 0;
}
```



## 30.2 Read-Modify-Write

# Example: Atomic Operations in Hardware



**CMPXCHG** **Compare and Exchange**

Compares the value in the AL, AX, EAX, or RAX register with the value in a register or a memory location (first operand). If the two values are equal, the instruction copies the value in the second operand to the first operand and sets the ZF flag in the rFLAGS register to 1. Otherwise, it copies the value in the first operand to the AL, AX, EAX, or RAX register and clears the ZF flag to 0.

The OF, SF, AF, PF, and CF flags are set to reflect the results of the compare.

When the first … modify-write on the memory opera… e same value to the memory opera…

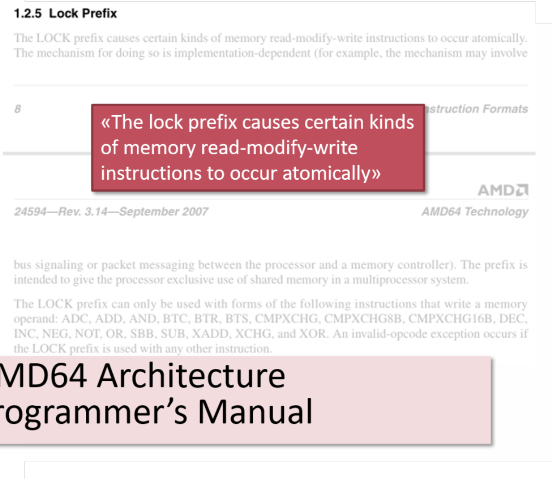
The forms of t… K prefix. For details about the LOC…

| Mnemonic | | |
|---|---|---|
| CMPXCHG *reg* | | egister or memory perand to the first perand to AL. |
| CMPXCHG *reg* | | egister or memory perand to the first perand to AX. |
| CMPXCHG *reg* | | register or memory perand to the first perand to EAX. |
| CMPXCHG *reg/mem64, reg64* | 0F B1 /r | Compare RAX register with a 64-bit register or mem location. If equal, copy the second operand to the fi operand. Otherwise, copy the first operand to RAX. |

**Related Instructions**

CMPXCHG8B, CMPXCHG16B

CMPXCHG mem, reg
«compares the value in Register A with the value in a memory location If the two values are equal, the instruction copies the value in the second operand to the first operand and sets the ZF flag in the flag regsiters to 1. Otherwise it copies the value in the first operand to A register and clears ZF flag to 0»



**1.2.5 Lock Prefix**

The LOCK prefix causes certain kinds of memory read-modify-write instructions to occur atomically. The mechanism for doing so is implementation-dependent (for example, the mechanism may involve

8 …struction Formats

AMD

24594—Rev. 3.14—September 2007 AMD64 Technology

bus signaling or packet messaging between the processor and a memory controller). The prefix is intended to give the processor exclusive use of shared memory in a multiprocessor system.

The LOCK prefix can only be used with forms of the following instructions that write a memory operand: ADC, ADD, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XADD, XCHG, and XOR. An invalid-opcode exception occurs if the LOCK prefix is used with any other instruction.

«The lock prefix causes certain kinds of memory read-modify-write instructions to occur atomically»

AMD64 Architecture Programmer’s Manual

## Read-Modify-Write

Concept of Read-Modify-Write: The effect of reading, modifying and writing back becomes visible at one point in time (happens atomically).

## Psudocode for CAS – Compare-And-Swap

```
bool CAS(int& variable, int& expected, int desired){
  if (variable == expected){
    variable = desired;
    return true;
  }
  else{
    expected = variable;
    return false;
  }
}
```

atomic

## Application example CAS in C++11

We build our own (spin-)lock:

```
class Spinlock{
  std::atomic<bool> taken {false};
public:
  void lock(){
    bool old = false;
    while (!taken.compare_exchange_strong(old=false, true)){}
  }
  void unlock(){
    bool old = true;
    assert(taken.compare_exchange_strong(old, false));
  }
};
```

## 30.3 Lock-Free Programming

Ideas

## Lock-free programming

Data structure is called

- *lock-free*: at least one thread always makes progress in bounded time even if other algorithms run concurrently. Implies system-wide progress but not freedom from starvation.
- *wait-free*: all threads eventually make progress in bounded time. Implies freedom from starvation.

## Progress Conditions

| | Non-Blocking | Blocking |
|---|---|---|
| Everyone makes progress | Wait-free | Starvation-free |
| Someone makes progress | Lock-free | Deadlock-free |

## Implication

- Programming with locks: each thread can block other threads indefinitely.
- Lock-free: failure or suspension of one thread cannot cause failure or suspension of another thread !

## Lock-free programming: how?

Beobachtung:

- RMW-operations are implemented *wait-free* by hardware.
- Every thread sees his result of a CAS or TAS in bounded time.

Idea of lock-free programming: read the state of a data sructure and change the data structure *atomically* if and only if the previously read state remained unchanged meanwhile.

## Example: lock-free stack

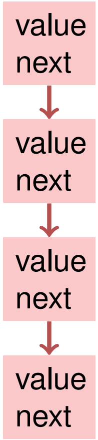
Simplified variant of a stack in the following

- pop prüft nicht, ob der Stack leer ist
- pop gibt nichts zurück

## (Node)

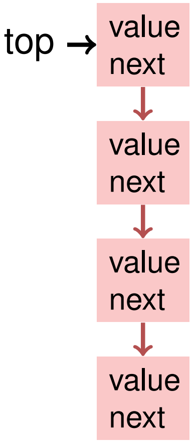
Nodes:

```
struct Node {
  T value;

  Node<T>* next;
  Node(T v, Node<T>* nxt): value(v), next(nxt) {}
};
```



## (Blocking Version)

```
template <typename T>
class Stack {
    Node<T> *top=nullptr;
    std::mutex m;
public:
    void push(T val){ guard g(m);
        top = new Node<T>(val, top);
    }
    void pop(){ guard g(m);
        Node<T>* old_top = top;
        top = top->next;
        delete old_top;
    }
};
```



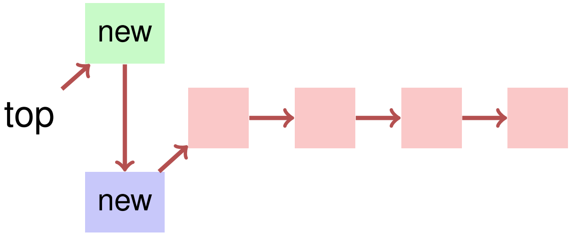
## Lock-Free

```
template <typename T>
class Stack {
  std::atomic<Node<T>*> top {nullptr};
public:
  void push(T val){
    Node<T>* new_node = new Node<T> (val, top);
    while (!top.compare_exchange_weak(new_node->next, new_node));
  }
  void pop(){
    Node<T>* old_top = top;
    while (!top.compare_exchange_weak(old_top, old_top->next));
    delete old_top;
  }
};
```

## Push

```
void push(T val){
  Node<T>* new_node = new Node<T> (val, top);
  while (!top.compare_exchange_weak(new_node->next, new_node));
}
```

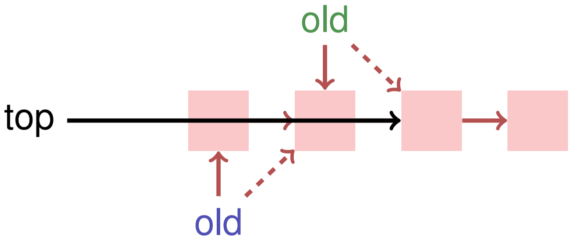
2 Threads:



## Pop

```
void pop(){
  Node<T>* old_top = top;
  while (!top.compare_exchange_weak(old_top, old_top->next));
  delete old_top;
}
```

2 Threads:



## Lock-Free Programming – Limits

- Lock-Free Programming is complicated.
- If more than one value has to be changed in an algorithm (example: queue), it is becoming even more complicated: threads have to “help each other” in order to make an algorithm lock-free.
- The *ABA problem* can occur if memory is reused in an algorithm. A solution of this problem can be quite expensive.