Felix Friedrich

Data Structures and Algorithms

Course at D-MATH of ETH Zurich

Spring 2022

Welcome!

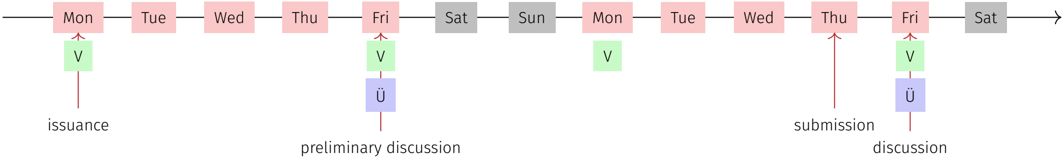
Course homepage

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

The team:

Assistants	Ciril Humbel	Kamelia Ivanova
	Ivana Klasovita	Leonhard Knirsch
	Vedran Mihal	Harun Mustafa
	Bastian Seifert	Felix Vittori
Backoffice	Ulysse Schaller	
Head-Assistant	Julia Chatain	
Lecturer	Felix Friedrich	

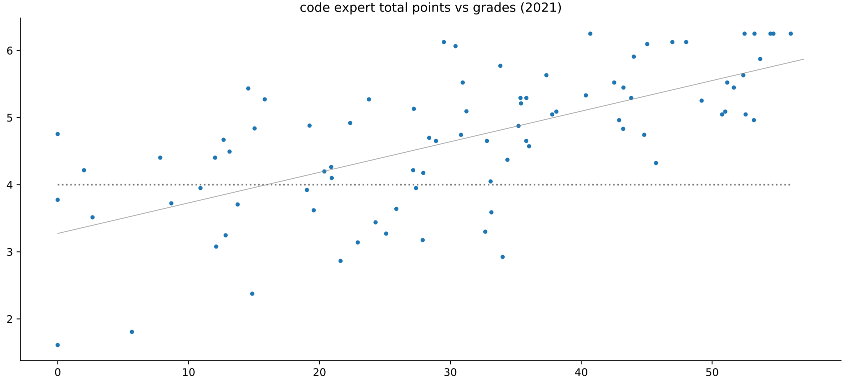
Exercises



- 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 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 (150 min). Examination aids: four a4 pages. No constraints regarding content and layout (text, images, single/double page, margins, font size, etc.).

The exam will most likely be performed in hybrid form (on paper and at the computer).

Offer

- Doing the weekly exercise series → bonus of maximally 0.25 of a grade point 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)

- 3 bonus exercises in total; 2/3 of the points suffice for the exam bonus of 0.25 marks
- You can, e.g. fully solve 2 bonus exercises, or solve 3 bonus exercises to 66% 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

We encourage you explicitly to discuss solution ideas and approaches with your colleagues. Teamwork is important, also in computer science. It is, however, also important that you learn actively and do not only reproduce. Therefore:

Rules for Bonus Tasks

You submit only solutions that you have written yourself and that you have understood. Copy-paste is not permitted, neither are team implementations.

Should there be any Problems ...

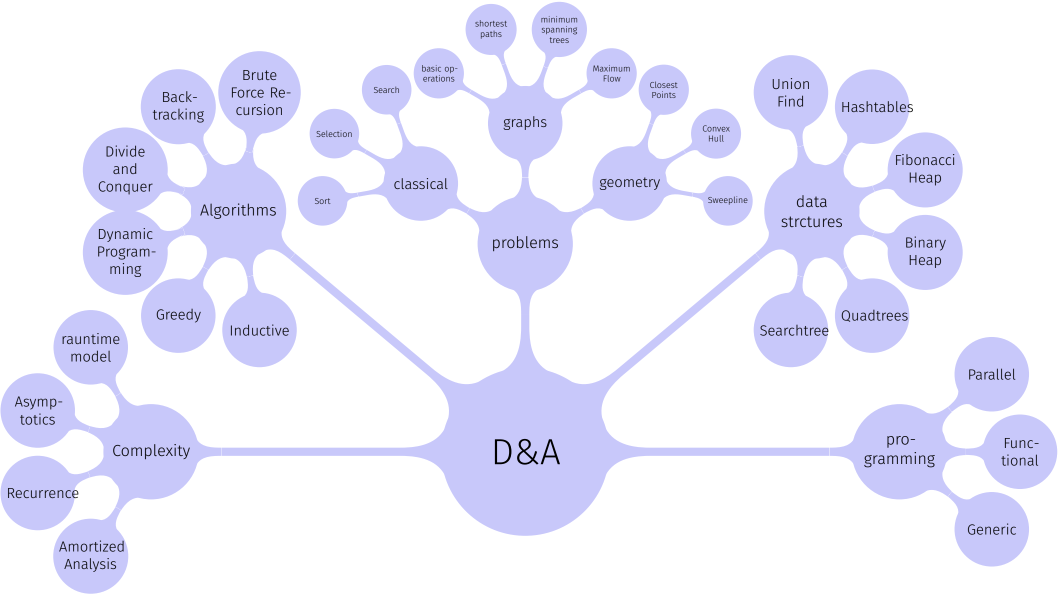
- with the course content
  - definitely attend all recitation sessions
  - ask questions there
  - and/or contact the assistant
- further problems
  - Email to chef assistant (Julia Chatain) or lecturer (Felix Friedrich)
- We are definitely willing to help (!)

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.


D&A
Complexity
rauntime model
Asymp-totics
Recurrence
Amortized Analysis
Algorithms
Back-tracking
Brute Force Re-cursion
Divide and Conquer
Dynamic Program-ming
Greedy
Inductive
problems
classical
Selection
Search
Sort
graphs
basic op-erations
shortest paths
minimum spanning trees
Maximum Flow
geometry
Closest Points
Convex Hull
Sweepline
data strctures
Union Find
Hashtables
Fibonacci Heap
Binary Heap
Quadtrees
Searchtree
pro-gramming
Parallel
Func-tional
Generic

1.2 Algorithms

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

Algorithm

Algorithm

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

Example Problem: Sorting

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

Possible input

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

Every example represents a **problem instance**

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

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

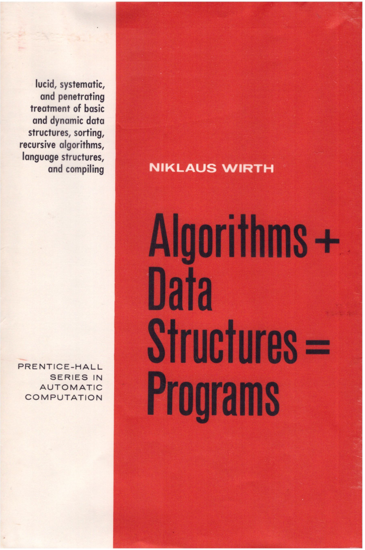
Possible solution

How many times are the lines executed each?

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

Data Structures

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



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

PRENTICE-HALL
SERIES IN
AUTOMATIC
COMPUTATION

NIKLAUS WIRTH

Algorithms +
Data
Structures =
Programs

Typical Algorithm Design Steps: Example

Route planning

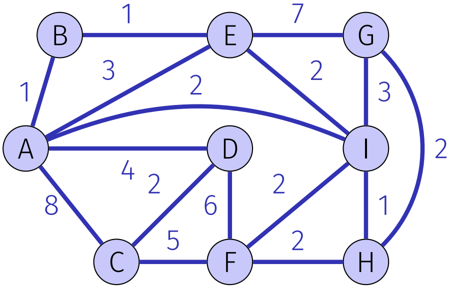
NEL
VOG
WEV
CLI
STC
STE
HAW
HAD
Haldenbachstrasse
Universitätstrasse
Sonneggstrasse
Clausiusstrasse
Gladbachstrasse
UNO
STD
WET
RZ
IFW
WES
CLW
SOL
SOI
CLV
CLP
SON
SOP
SOK
WEH
UNH
Stampfenbachstrasse
SOH
Universitätsspital Zürich
CLX
CLT
CLD
SOQ
UNG
CHN
Spöndlistrasse
ETL
HCA
HCW
HCB
WPW
NW
NO
CAB
HCH
AMA
VOC
WEC
LEE
CNB
STW
STS
ETA
ML
LFG
LFH
LFV
LFO
VOB
VOA
LEO
CLA
ETF
ETZ
GLC
LFW
Sternwartstrasse
Schmelzbergstrasse
HAC
TAN
Leonhardstrasse
Tannenstr.
HAA
LEH
GEP
HG
Rämistrasse
Gloriastrasse
Universitätsspital Zürich
Limmat
HRS
Poly-terrasse
MM
Karl-Schmid-Str.
HRG
Universität Zürich

Typical Design Steps

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

Difficult Problem: Travelling Salesman

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



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

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

Hard problems.

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

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

Efficiency

Resources are bounded and do not come for free:

- Computing time → Efficiency
- Storage space → Efficiency

Actually, this course is nearly only about efficiency.

2. Efficiency of algorithms

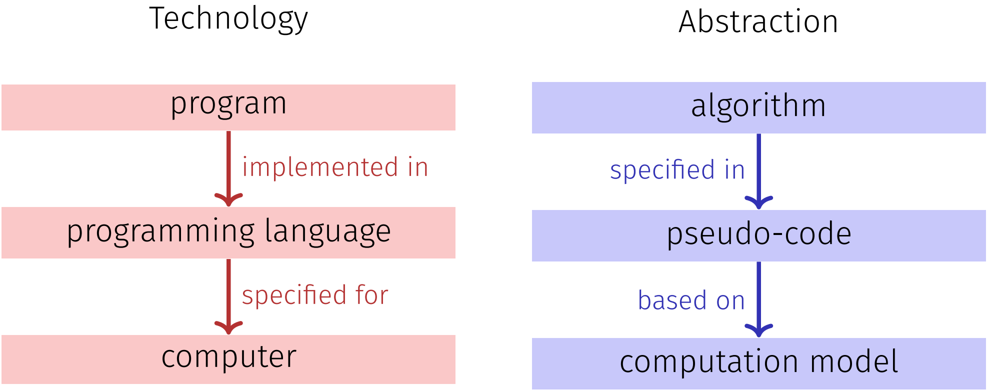
Efficiency of Algorithms, Random Access Machine Model, Function Growth, Asymptotics [Cormen et al, Kap. 2.2,3,4.2-4.4 | Ottman/Widmayer, Kap. 1.1]

Efficiency of Algorithms

Goals

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

Programs and Algorithms



Technology

program

implemented in

programming language

specified for

computer

Abstraction

algorithm

specified in

pseudo-code

based on

computation model

Technology Model

Random Access Machine (RAM) Model

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

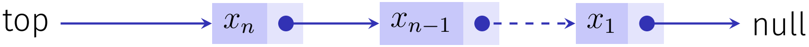
Size of the Input Data

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

For Dynamic Data Strcutures

Pointer Machine Model

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



Asymptotic behavior

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

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

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

Algorithms, Programs and Execution Time

Program: concrete implementation of an algorithm.

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

Example 1

3GHz computer. Maximal number of operations per cycle (e.g. 8). ⇒ lower bound.

A single operations does never take longer than a day ⇒ upper bound.

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

2.2 Function growth

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

Superficially

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

More precise: asymptotic upper bound

provided: a function $g : \mathbb{N} \to \mathbb{R}$.
Definition:[1]

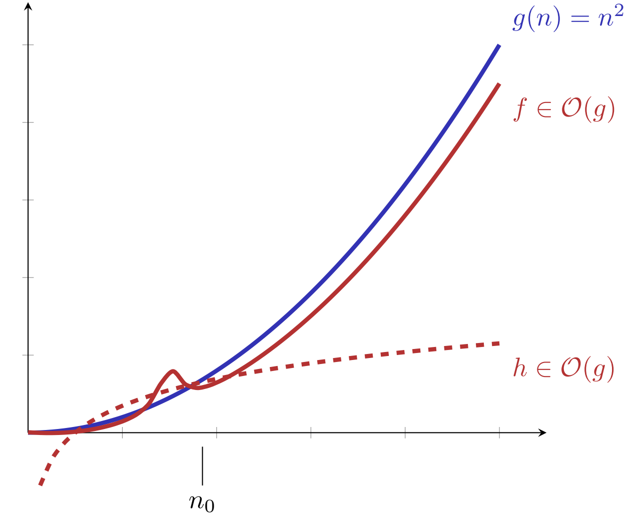
$$\begin{aligned}\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R}| \\ &\exists\, c > 0, \exists n_0 \in \mathbb{N} : \\ &\forall\, n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}\end{aligned}$$

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

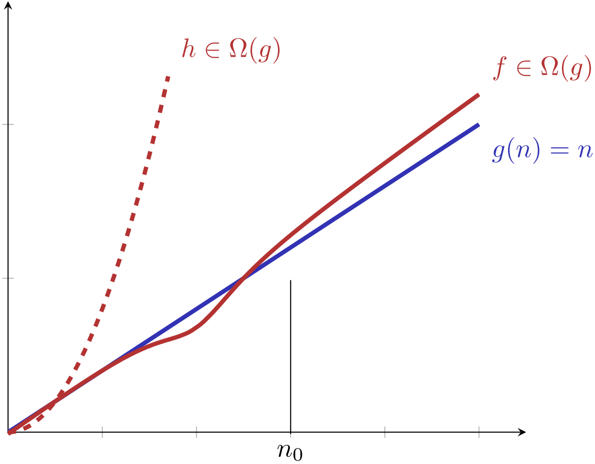


Converse: asymptotic lower bound

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

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

Example



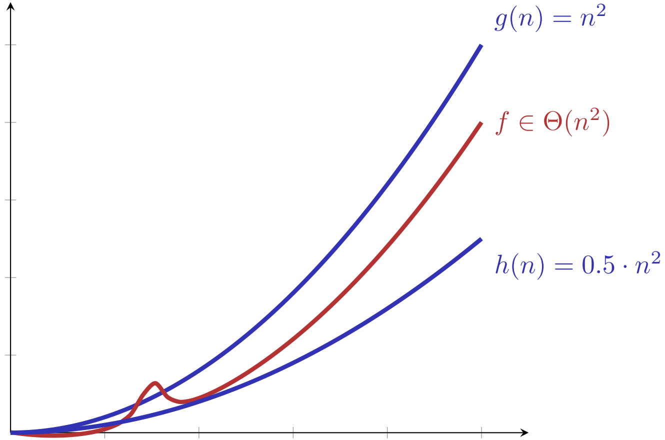
Asymptotic tight bound

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

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

Simple, closed form: exercise.

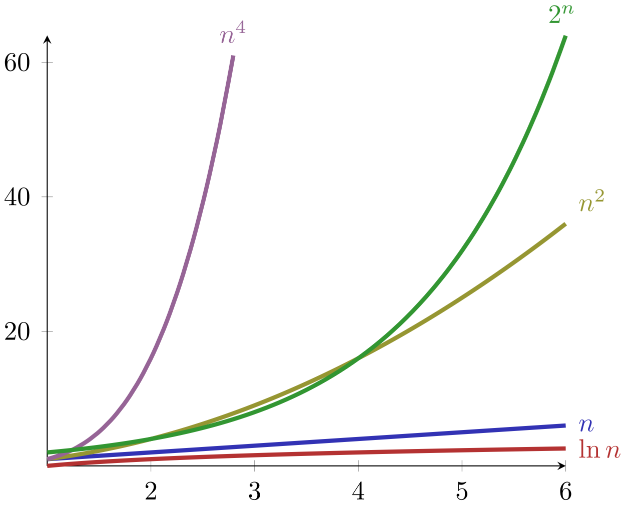
Example



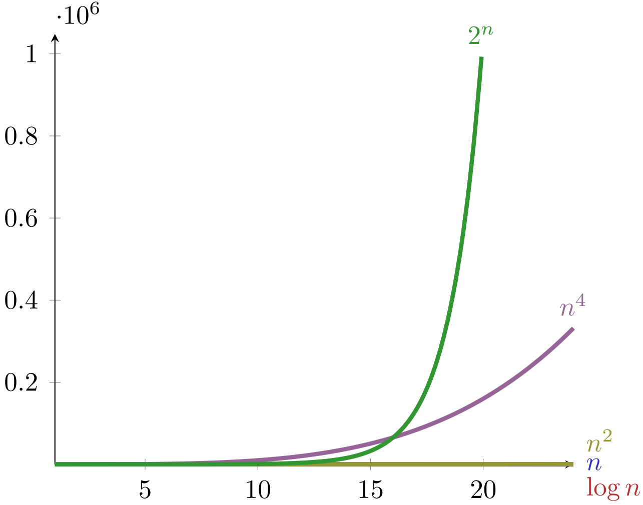
Notions of Growth

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

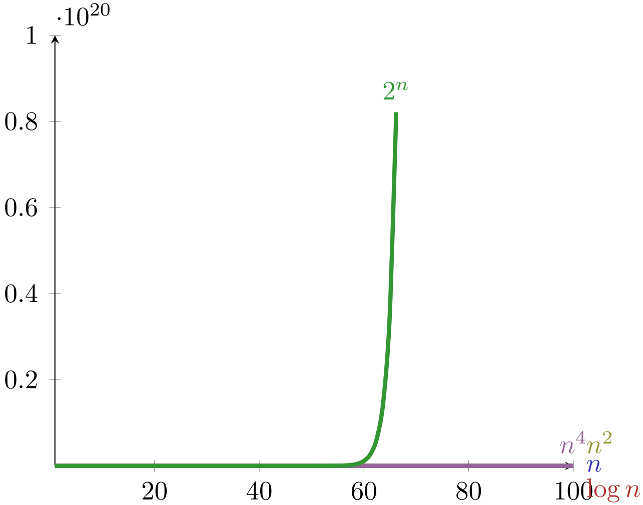
Small n



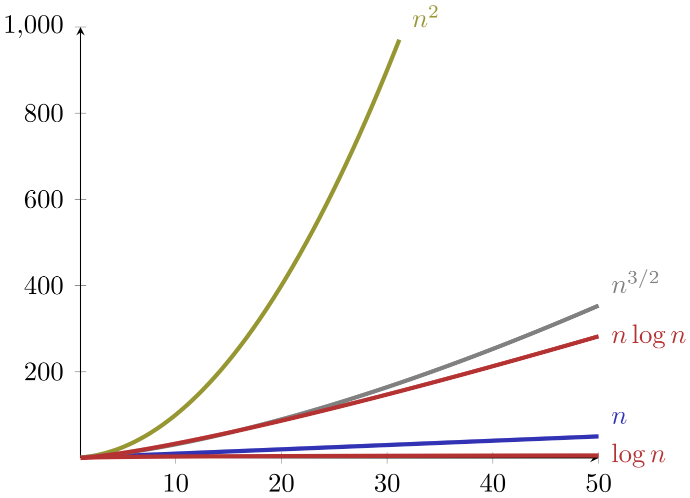
Larger n



“Large” n



Logarithms



Time Consumption

Assumption 1 Operation = $1\mu s$.

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

Useful Tool

Theorem 2

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

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

About the Notation

Common casual notation

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

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

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

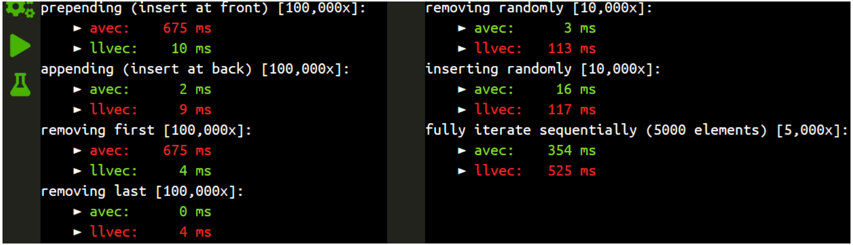
$n = \mathcal{O}(n^2), n^2 = \mathcal{O}(n^2)$ but naturally $n \neq n^2$.

We avoid this notation where it could lead to ambiguities.

Reminder: Efficiency: Arrays vs. Linked Lists

- Memory: our **`avec`** requires roughly n ints (vector size n), our **`llvec`** roughly $3n$ ints (a pointer typically requires 8 byte)

- Runtime (with **`avec`** = **`std::vector`**, **`llvec`** = **`std::list`**):



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

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

Asymptotic Runtimes

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

Typical asymptotic running times (Anticipation!)

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

A = amortized, P=expected, otherwise worst case

Complexity

Complexity of a problem P

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

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

Complexity

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

[2] Number fundamental operations

3. Examples

Show Correctnes of an Algorithm or its Implementation, 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

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 its efficiency?

The Essentials

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

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

Termination

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

Recursively, Functional

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

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

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

■

[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);
}
```

→

```
// 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 ⇒ 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);
}
```

→

```
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; // new a
    b /= 2; // new b
  }
  res += a; // base case 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;  → part of the division
      z = a; → directly in res
    }
    res += z;
    a *= 2;
    b /= 2;
  }
  res += a;  → into the loop
  return res;
}
```

→

```
// 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;
}
```

let $x := a \cdot b$.

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: $x = a \cdot b + res$ and $b = 0$

therefore $res = x$.

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 atomic, 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;
}
```

→

```
// 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:

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

$2 \cdot 2 = 4$ 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.

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

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

Algorithm Karatsuba Ofman

```
**Input**: Two positive integers $x$ and $y$ with $n$ decimal digits each: $(x_i)_{1 \le i \le n}$,
           $(y_i)_{1 \le i \le n}$
**Output**: Product $x \cdot y$
if $n = 1$ then
    return $x_1 \cdot y_1$
else
    Let $m := \lfloor \frac{n}{2} \rfloor$
    Divide $a := (x_1, \dots, x_m)$, $b := (x_{m+1}, \dots, x_n)$, $c := (y_1, \dots, y_m)$,
    $d := (y_{m+1}, \dots, y_n)$
    Compute recursively $A := a \cdot c$, $B := b \cdot d$, $C := (a - b) \cdot (d - c)$
    Compute $R := 10^n \cdot A + 10^m \cdot A + 10^m \cdot B + B + 10^m \cdot C$
    return $R$
```

Analysis

$M(n)$: Number of single-digit multiplications.
Recursive application of the algorithm from above ⇒ recursion equality:

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

Iterative Substition

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

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

Proof: induction

Hypothesis $H(k)$:

$$M(2^k) = F(k) := 3^k. \tag{H}$$

Claim:

$$H(k) \text{ holds for all } \; k \in \mathbb{N}_0.$$

Base clause $k = 0$:

$$M(2^0) \overset{R}{=} 1 = F(0). \quad \checkmark$$

Induction step $H(k) \Rightarrow H(k+1)$**:**

$$M(2^{k+1}) \overset{R}{=} 3 \cdot M(2^k) \overset{H(k)}{=} 3 \cdot F(k) = 3^{k+1} = F(k+1). \quad \checkmark$$

■

Comparison

Traditionally n^2 single-digit multiplications.
Karatsuba/Ofman:

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

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. [4]

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

[4] In March 2019, David Harvey and Joris van der Hoeven have shown an $\mathcal{O}(n \log n)$ algorithm that is practically irrelevent yet. It is conjectured, but yet unproven that this is the best lower bound we can get.

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 $c > 1$, $d > 0$) by the following recurrence

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

Appendix: Asymptotics with Addition and Shifts

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

$$\begin{aligned}
T(2^k) &= 3 \cdot T\left(2^{k-1}\right) + c \cdot 2^k \\
&= 3 \cdot (3 \cdot T(2^{k-2}) + c \cdot 2^{k-1}) + c \cdot 2^k \\
&= 3 \cdot (3 \cdot (3 \cdot T(2^{k-3}) + c \cdot 2^{k-2}) + c \cdot 2^{k-1}) + c \cdot 2^k \\
&= 3 \cdot (3 \cdot (...(3 \cdot T(2^{k-k}) + c \cdot 2^1)...) + c \cdot 2^{k-1}) + c \cdot 2^k \\
&= 3^k \cdot d + c \cdot 2^k \sum_{i=0}^{k-1} \frac{3^i}{2^i} = 3^k \cdot d + c \cdot 2^k \, \frac{\frac{3^k}{2^k} - 1}{\frac{3}{2} - 1} \\
&= 3^k(d + 2c) - 2c \cdot 2^k
\end{aligned}$$

Thus $T(2^k) = 3^k(d + 2c) - 2c \cdot 2^k \in \Theta(3^k) = \Theta(3^{\log_2 n}) = \Theta(n^{\log_2 3})$.

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

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

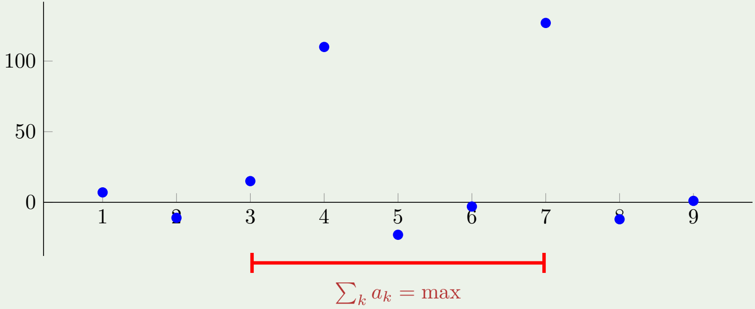
Goal: development of the asymptotically most efficient (correct) algorithm.

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

Maximum Subarray Problem

Given: an array of n real numbers $(a_1, \dots, a_n)$.
Wanted: interval $[i, j]$, $1 \le i \le j \le n$ with maximal positive sum $\sum_{k=i}^{j} a_k$.

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



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 3

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

Proof:

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

■

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 $(a_1, a_2, \dots, a_n)$
Output: I, J such that $\sum_{k=J}^{J} a_k$ maximal.

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

Analysis

Theorem 4

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

Proof:

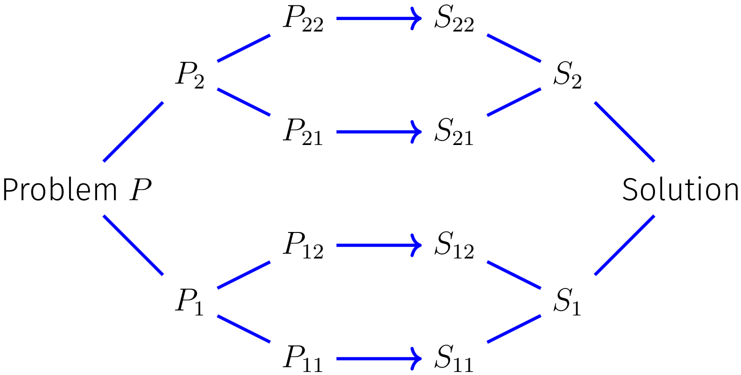
$$\sum_{i=1}^{n} 1 + \sum_{i=1}^{n} \sum_{j=i}^{n} 1 = n + \sum_{i=1}^{n} (n - i + 1) = n + \sum_{i=1}^{n} i = \Theta(n^2)$$

■

divide et impera

Divide and Conquer

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



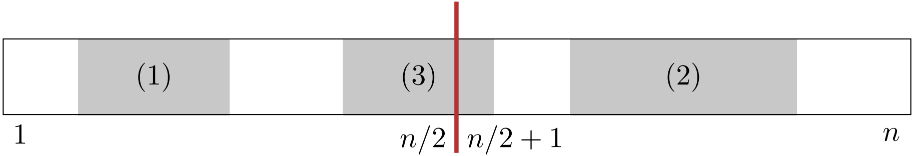
Maximum Subarray – Divide

- 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 $1 \le i \le n/2 < j \le n$

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

Maximum Subarray Divide and Conquer Algorithm

```
Input:  A sequence of $n$ numbers $(a_1, a_2, \ldots, a_n)$
Output: Maximal $\sum_{k=i'}^{j'} a_k$.
if $n = 1$ then
    return $\max\{a_1, 0\}$
else
    Divide $a = (a_1, \ldots, a_n)$ in $A_1 = (a_1, \ldots, a_{n/2})$ und $A_2 = (a_{n/2+1}, \ldots, a_n)$
    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 5

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 $(a_1, a_2, \dots, a_n)$
Output: Maximal $\sum_{k=i'}^{j'} a_k$.
if $n = 1$ then
    $\Theta(1)$ return $\max\{a_1, 0\}$
else
    $\Theta(1)$ Divide $a = (a_1, \dots, a_n)$ in $A_1 = (a_1, \dots, a_{n/2})$ und $A_2 = (a_{n/2+1}, \dots, a_n)$
    $T(n/2)$ 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 $n = 2^k$:

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

Solution:

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

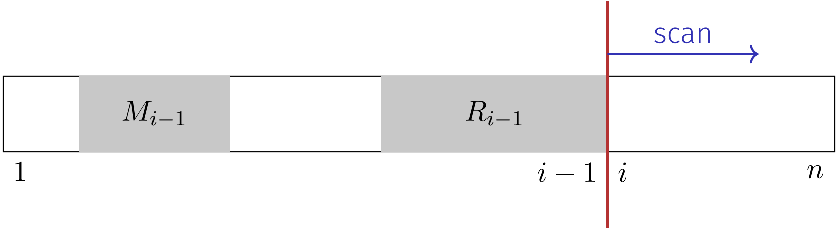
also

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

■

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 $(a_1, a_2, \ldots, a_n)$.
Output: $\max\{0, \max_{i,j} \sum_{k=i}^{j} a_k\}$.
$M \leftarrow 0$
$R \leftarrow 0$
for $i = 1 \ldots n$ do
    $R \leftarrow R + a_i$
    if $R < 0$ then
        $R \leftarrow 0$
    if $R > M$ then
        $M \leftarrow R$
return $M$;
```

Analysis

Theorem 6

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 7

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

Proof: 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 and repetition of some mathematical formulas

Logarithms

$$\log_a y = x \Leftrightarrow a^x = y \quad (a > 0, y > 0)$$

$$\log_a(x \cdot y) = \log_a x + \log_a y \qquad a^x \cdot a^y = a^{x+y}$$

$$\log_a \frac{x}{y} = \log_a x - \log_a y \qquad \frac{a^x}{a^y} = a^{x-y}$$

$$\log_a x^y = y \log_a x \qquad a^{x \cdot y} = (a^x)^y$$

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

$$\log_b x = \log_b a \cdot \log_a x \qquad a^{\log_b x} = x^{\log_b a}$$

To see the last line, replace $x \to a^{\log_a x}$

Sums

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

Trick

$$\begin{aligned}\sum_{i=0}^{n} i = \frac{1}{2}\left(\sum_{i=0}^{n} i + \sum_{i=0}^{n} n - i\right) = \frac{1}{2}\sum_{i=0}^{n} i + n - i \\ = \frac{1}{2}\sum_{i=0}^{n} n = \frac{1}{2}(n+1) \cdot n\end{aligned}$$

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

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

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

For $0 \leq \rho < 1$:

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

4. C++ advanced (I)

Repetition: Vectors, Pointers and Iterators,
Range for, Keyword auto, a Class for Vectors, Subscript-operator,
Move-construction, Iterators

Learning objectives

- 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!

Not as good as it could be!

4.1 Useful Tools

On our way to elegant, less complicated code.

auto

The keyword **auto** (from C++11):
The type of a variable is inferred from the initializer.

```cpp
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 << " ";
  }
}
```

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.

```
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;
```

Cool!

```cpp
#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 << " ";
}
```

4.2 Memory Allocation

Construction of a vector class

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.

Constructor Delegation

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

Copy-&-Swap Idiom

```
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`.

Clarification of Terms: Idioms and Patterns

Idiom and **(Design) Pattern** are notions coming from **software engineering**

- Design patterns are general, reusable solutions to commonly occurring design problems. They capture best practices and usually describe relationships and interactions among classes and objects, e.g. the visitor pattern.
- Idioms are language-specific ways of implementing certain tasks/steps, e.g. the RAII idiom, or the Copy-&-Swap Idiom in C++. Idioms usually require less code than patterns.

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;
}
```

4.3 Iterators

How to support the range **`for`**

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;
    }
}
```

(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;
    }

}
```

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;
}
```

Vector Interface

```
class Vector{
public:
  Vector(); // Default Constructor
  Vector(std::size_t s); // Constructor
  ~Vector(); // Destructor
  Vector(const Vector &v); // Copy Constructor
  Vector& operator=(const Vector&v); // Assignment Operator
  double& operator[] (std::size_t pos); // Subscript operator (read/write)
  const double& operator[] (std::size_t pos) const; // Subscript operator
  std::size_t size() const;
  double* begin(); // iterator begin
  double* end(); // iterator end
  const double* begin() const; // const iterator begin
  const double* end() const; // const iterator end
}
```

4.4 Efficient Memory-Management*

How to avoid copies

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;
    };
}
```

Vector Interface

```
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())
```

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.

5. C++ advanced (II): Templates

Some (converted) slides from Malte Schwerhoff

Generic Programming

Goal: Make code (functions, classes) as widely usable as possible

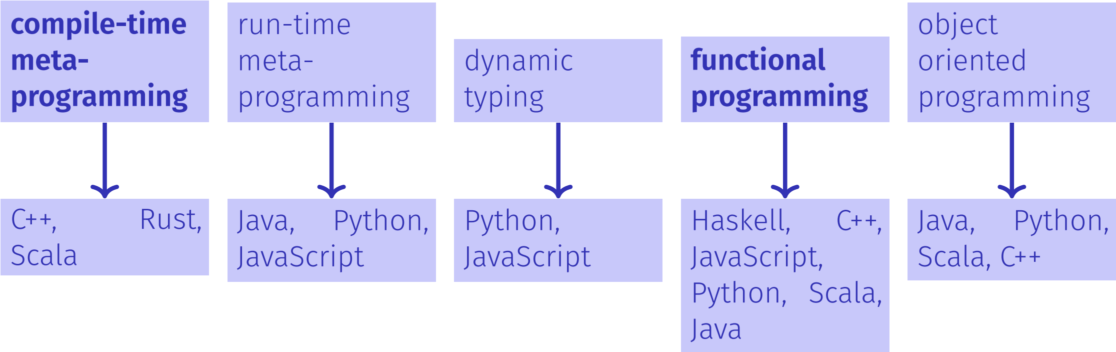
Familiar use-cases:

- `vector`: store objects of any type, e.g. ints, strings, ...
- `std::max(e1, e2)`: return the greater of any two objects
- `Exp* e = new ...; e.eval()`: evaluate any expression (variable, constant, sum, product, ...)

Generic Programming

How: Different languages implement different approaches (and typically more than one), but high-level ideas are often similar.

Examples:



compile-time meta-programming → C++, Rust, Scala

run-time meta-programming → Java, Python, JavaScript

dynamic typing → Python, JavaScript

functional programming → Haskell, C++, JavaScript, Python, Scala, Java

object oriented programming → Java, Python, Scala, C++

Motivation

Goal: generic vector class and functionality.

```
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)
```

Parametric Polymorphism

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>`** Replace **`T`** by the representative name).

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

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

Notation: **Type Instantiation**.

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< !
```

Parametric Polymorphism II

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>`** (Replace **`T`** by the chosen name)

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
```

Safety

```
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 (const auto& x: t)
        std::cout << x << " ";
    std::cout << "\n";
}

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

Neat: a generic function that can be applied to all containers. What are the limits of this function?

- Requirements for T?
- Requirements on Elements of T?

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.

Type Inference

- Upon instantiation, template parameters must be provided explicitly – except if the compiler can infer them from provided arguments
- Type inference improved with C++17

```
auto p1 = Pair<int>(1,2); // OK
Pair<int> p2 = Pair(1,2); // C++14: error; C++17: OK
auto p3 = Pair(1,2); // C++14: error; C++17: OK
```

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
}
```

Conclusion

- Templates improve code reusability, by making classes and functions parametric w.r.t. types
- Implemented via static code-generation (compile-time meta-programming)
  - Advantages
    - No runtime overhead (compared to dynamic solutions, e.g. inheritance or dynamic typing)
    - Compiler can optimise generated code as usual
    - Type-dependent static specialisation possible (see below)
  - Disadvantages
    - Separate compilation (.h vs. .cpp) no longer possible
    - Resulting binary (machine code) larger
    - Delayed type checking, more complex error messages (mitigatable via concepts, see next)

Concepts (C++20)

```
struct Student { ... }; // Lacks == and < operators

template <typename K>
struct BSTNode {
  K key;
  ...
  bool contains(K search_key) {
    if (search_key < key) ...
    else if (search_key == key) ...
    else ...
  }
};

auto n1 = BSTNode(8);
auto n2 = BSTNode("Howdy!");
auto n3 = BSTNode(new int(8));
auto n4 = BSTNode(Student("Omar"));
```

Code compiles just fine

- But should it?
- If it shouldn't, why does it compile?

Concepts (C++20)

```cpp
auto n4 = BSTNode(Student("Omar"));
n4.contains(Student("Ida")); // Error!
```

Code fails to compile

```
test.cpp:29:
  In instantiation of
  'bool BSTNode<K>::contains(K)
  [with K = Student]':
test.cpp:21:
  no match for 'operator<' (operand types
  are 'Student' and 'Student')
      21 |     if (search_key < key)
           |
```

Problems:

- Instantiation of **`BSTNode<Student>`** already nonsensical, but accepted by the compiler
- Requirement on concrete `K`s scattered across code, not part of declaration

Concepts (C++20)

Better: explicit requirement, prevent nonsensical instantiations.

```
struct Student { ... }; // Lacks < operators

template <typename T>
concept Comparable = requires(T t1, T t2) {
  { t1 < t2 };
};
template <Comparable K>
struct BSTNode {
  K key;
      ...
    bool contains(K search_key) { ... }
};

auto n4 = BSTNode(Student("Omar")); // Error
```

```
test.cpp:43:
  In substitution of ’template<class K>
  BSTNode(K)-> BSTNode<K> [with K = Student]:
test.cpp:23:
  template constraint failure for
  ’template<class K> requires Comparable<K>
  struct BSTNode
test.cpp:4:
  required for the satisfaction of Comparable<K>
  [with K = Student]
```

Concepts (C++20)

```cpp
#include <algorithm>
#include <vector>
#include <list>

template <typename Iter>
void clever_sort(Iter begin, Iter end) {
  if (!is_sorted(begin, end))
    std::sort(begin, end);
}

int main() {
  std::vector<int> data1 = {3,-1,10};
  clever_sort(data1.begin(), data1.end());

  std::list<int> data2 = {3,-1,10};
  clever_sort(data2.begin(), data2.end());
}
```

```
In file included from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/algorithm:62,
               from <source>:1:
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_algo.h: In instantiation of ’void std::__sort(
      _RandomAccessIterator, _RandomAccessIterator, _Compare) [with _RandomAccessIterator = std::_List_iterator<int>;
      _Compare = __gnu_cxx::__ops::_Iter_less_iter]’:
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_algo.h:4842:18: required from ’void std::sort(_RAIter, _RAIter)
       [with _RAIter = std::_List_iterator<int>]’
<source>:10:14: required from ’void clever_sort(Iter, Iter) [with Iter = std::_List_iterator<int>]’
<source>:18:14: required from here
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_algo.h:1955:50: error: no match for ’operator-’ (operand types
      are ’std::_List_iterator<int>’ and ’std::_List_iterator<int>’)
 1955 |                              std::__lg(__last - __first) * 2,
      |                                        ~~~~~~~^~~~~~~~~
In file included from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_algobase.h:67,
               from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/algorithm:61,
               from <source>:1:
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_iterator.h:560:5: note: candidate: ’template<class _IteratorL,
      class _IteratorR> constexpr decltype ((__y.base() - __x.base())) std::operator-(const std::reverse_iterator<_Iterator
      >&, const std::reverse_iterator<_IteratorR>&)’
  560 |    operator-(const reverse_iterator<_IteratorL>& __x,
      |    ^~~~~~~~
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_iterator.h:560:5: note: template argument deduction/
      substitution failed:
In file included from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/algorithm:62,
               from <source>:1:
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_algo.h:1955:50: note: ’std::_List_iterator<int>’ is not derived
       from ’const std::reverse_iterator<_Iterator>’
 1955 |                              std::__lg(__last - __first) * 2,
      |                                        ~~~~~~~^~~~~~~~~
In file included from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_algobase.h:67,
               from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/algorithm:61,
               from <source>:1:
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_iterator.h:1639:5: note: candidate: ’template<class _IteratorL,
       class _IteratorR> constexpr decltype ((__x.base() - __y.base())) std::operator-(const std::move_iterator<_IteratorL>&,
       const std::move_iterator<_IteratorR>&)’
 1639 |    operator-(const move_iterator<_IteratorL>& __x,
      |    ^~~~~~~~
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_iterator.h:1639:5: note: template argument deduction/
      substitution failed:
In file included from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/algorithm:62,
               from <source>:1:
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_algo.h:1955:50: note: ’std::_List_iterator<int>’ is not derived
       from ’const std::move_iterator<_IteratorL>’
 1955 |                              std::__lg(__last - __first) * 2,
      |                                        ~~~~~~~^~~~~~~~~
```

Concepts (C++20)

```cpp
#include <algorithm>
#include <vector>
#include <list>
#include <iterator>

template<std::random_access_iterator Iter>
void clever_sort(Iter begin, Iter end) {
  if (!is_sorted(begin, end))
    std::sort(begin, end);
}

int main() {
  std::vector<int> data1 = {3,-1,10};
  clever_sort(data1.begin(), data1.end());

  std::list<int> data2 = {3,-1,10};
  clever_sort(data2.begin(), data2.end());
}
```

```
In function 'int main()':
<source>:18:14: error: no matching function for call to 'clever_sort(std::__cxx11::list<int>::
    iterator, std::__cxx11::list<int>::iterator)'
   18 |   clever_sort(data2.begin(), data2.end());
      |   ~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~
<source>:8:6: note: candidate: 'template<class Iter> requires random_access_iterator<Iter>
    void clever_sort(Iter, Iter)'
    8 | void clever_sort(Iter begin, Iter end) {
      |      ^~~~~~~~~~~
<source>:8:6: note: template argument deduction/substitution failed:
<source>:8:6: note: constraints not satisfied
In file included from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/compare:39,
                 from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/stl_pair.h:65,
                 from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/utility:70,
                 from /opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/algorithm:60,
                 from <source>:1:
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/concepts: In substitution of 'template<
    class Iter> requires random_access_iterator<Iter> void clever_sort(Iter, Iter) [with Iter
     = std::_List_iterator<int>]':
<source>:18:14: required from here
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/concepts:67:13: required for the
    satisfaction of 'derived_from<typename std::__detail::__iter_concept_impl<_Iter>::type,
    std::random_access_iterator_tag>' [with _Iter = std::_List_iterator<int>]
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/bits/iterator_concepts.h:660:13:
    required for the satisfaction of 'random_access_iterator<Iter>' [with Iter = std::
    _List_iterator<int>]
/opt/compiler-explorer/gcc-11.2.0/include/c++/11.2.0/concepts:67:28: note: 'std::
    random_access_iterator_tag' is not a base of 'std::bidirectional_iterator_tag'
   67 |     concept derived_from = __is_base_of(_Base, _Derived)
      |                            ^~~~~~~~~~~~~~~~~~~~~~~~~~~~
          ~~~~~~~~^~~~~~~~~
```

Specialization

- Basic idea: general implementation for arbitrary types, but specialised (usually: more efficient) implementation for specific types
- Applicable to data structures (e.g. more space-efficient internal representation) and algorithms (e.g. more efficient sorting)
- Examples:
  - General **`vector<T>`**, but space-compressed **`vector<bool>`**
  - Quicksort for a random-access container, Mergesort otherwise

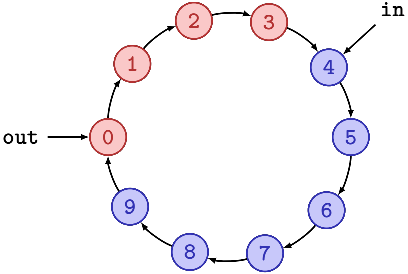
Specialization

```
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)
```

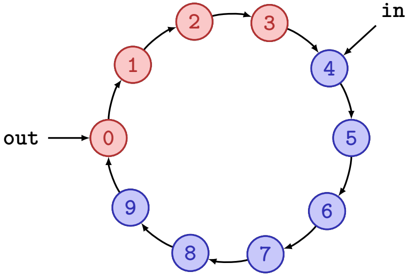
Template Parameterization with Values

```
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;
}
```

← Potential for optimization if size $= 2^k$.

Specialisation

Specialisation has many different use cases, and can come in different forms:

- Optimization by specialisation of types: e.g. **`vector<T>`** generic, and **`vector<bool>`** specific
- Optimisation by specialisation for values: e.g. **`std::array<T, n>`**
- Type-specifict implementation via specialisations: e.g. **`std::hash<T>`**, used e.g. by **`std::unordered_set`**.
  This idea is the base of traits.

6. 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

  > 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, $1 \le k \le n$ 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

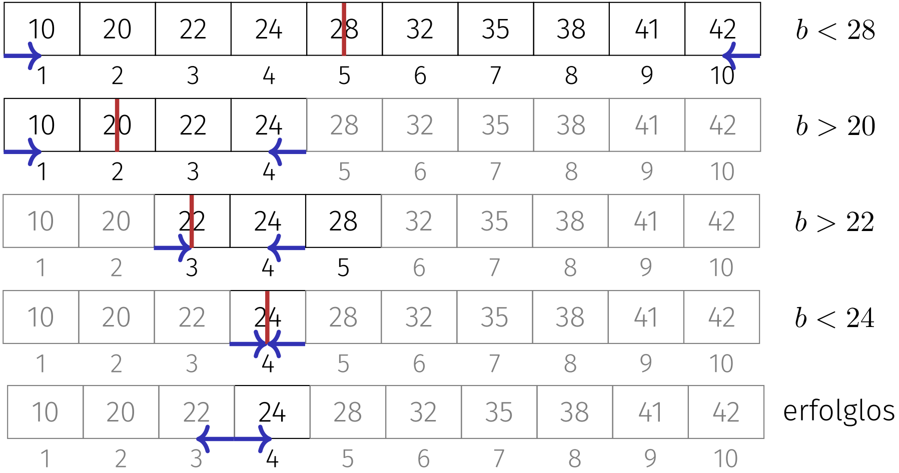
- Sorted array A with n elements $(A[1],\dots,A[n])$ with $A[1] \leq A[2] \leq \cdots \leq A[n]$.
- Key b

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 $b = 23$.



10	20	22	24	28	32	35	38	41	42	$b < 28$
1	2	3	4	5	6	7	8	9	10	

10	20	22	24	28	32	35	38	41	42	$b > 20$
1	2	3	4	5	6	7	8	9	10	

10	20	22	24	28	32	35	38	41	42	$b > 22$
1	2	3	4	5	6	7	8	9	10	

10	20	22	24	28	32	35	38	41	42	$b < 24$
1	2	3	4	5	6	7	8	9	10	

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

Binary Search Algorithm BSearch(A, l, r, b)

```
Input:  Sorted array $A$ of $n$ keys. Key $b$. Bounds $1 \le l, r \le n$ mit $l \le r$ or
        $l = r + 1$.
Output: Index $m \in [l, \dots, r + 1]$, such that $A[i] \le b$ for all $l \le i < m$ and
        $A[i] \ge b$ for all $m < i \le r$.
$m \leftarrow \lfloor (l + r)/2 \rfloor$
if $l > r$ then // Unsuccessful search
    return l
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 8

The binary sorted search algorithm requires $\Theta(\log n)$ fundamental operations in the worst case.

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?

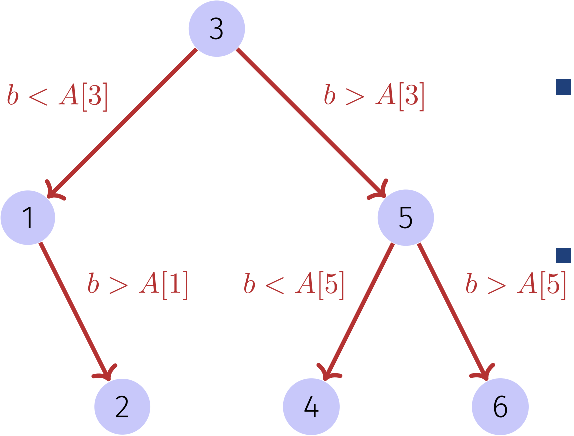
No: worst case number of comparisons $\Omega(n)$.

Lower Bounds

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

Does for *any* comparison-based 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 ⇒ 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 9

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 10

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

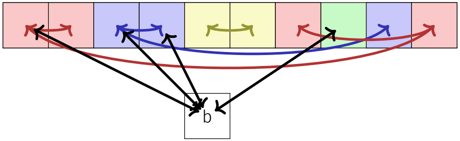
Attempt

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 $i + e \geq n - g + g = n$. ■

7. 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.[6]

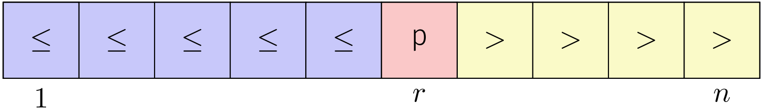
[6] An indication that the naive algorithm can be improved.

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.



$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	p	$>$	$>$	$>$	$>$
1					r				n

Algorithm **`Partition`**(A, l, r, p)

Input: Array A, that contains the pivot p in $A[l, \dots, r]$ at least once.
Output: Array A partitioned in $[l, \dots, r]$ 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: $A_i \leq p \ \forall i \in [0, l)$, $A_i \geq p \ \forall i \in (r, n]$, $\exists k \in [l, r] : A_k = p$.

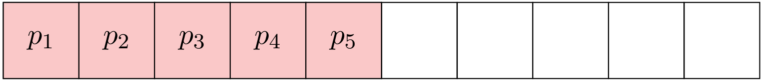
```
while $l \leq r$ do
                                                  $I$
    while $A[l] < p$ do
        $l \leftarrow l + 1$
                                                  $I$ und $A[l] \geq p$
    while $A[r] > p$ do
        $r \leftarrow r - 1$
                                                  $I$ 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

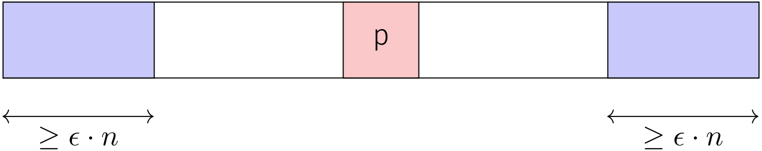
```
while $l \leq r$ do
    while $A[l] < p$ do                progress if $A[l] < p$
        $l \leftarrow l + 1$
    while $A[r] > p$ do                progress if $A[r] > p$
        $r \leftarrow r - 1$
    swap($A[l]$, $A[r]$)               progress if $A[l] > p$ oder $A[r] < p$
    if $A[l] = A[r]$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



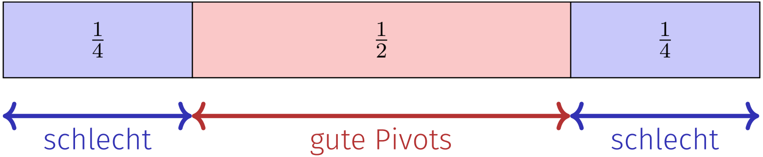
Analysis

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

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

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 $1 \leq l \leq r \leq n$
Output: Random "good" pivot $x \in A[l, \dots, r]$
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.

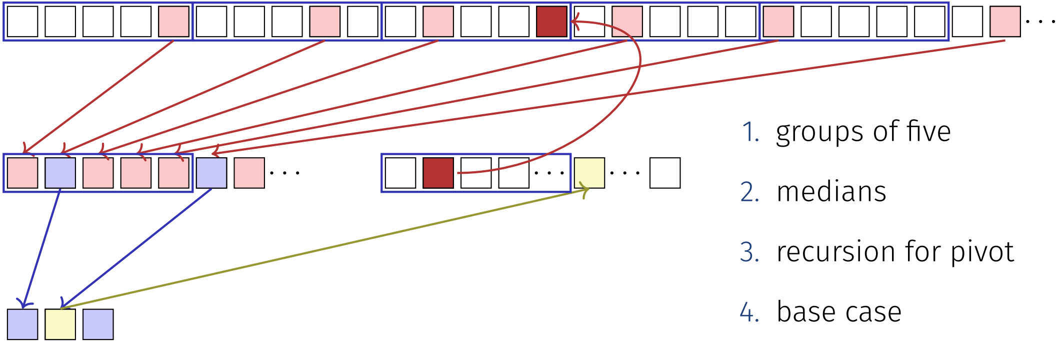
Median of medians

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

Algorithmus `MMSelect`(A, l, r, k)

Input: Array A with length n with pair-wise different entries. $1 \le l \le k \le r \le n$, $A[i] < A[k]\ \forall\ 1 \le i < l$, $A[i] > A[k]\ \forall\ r < i \le n$

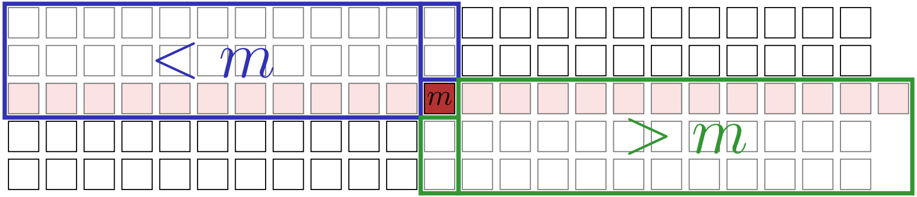
Output: Value $x \in A$ with $|\{j | A[j] \le x\}| = k$

```
$m \leftarrow$ MMChoose$(A, l, r)$
$i \leftarrow$ Partition$(A, l, r, m)$
if $k < i$ then
    return MMSelect$(A, l, i-1, k)$
else if $k > i$ then
    return MMSelect$(A, i+1, r, k)$
else
    return $A[i]$
```

Algorithmus $\texttt{MMChoose}(A, l, r)$

```
Input: Array $A$ with length $n$ with pair-wise different entries. $1 \le l \le r \le n$.
Output: Median $m$ of medians
if $r - l \le 5$ then
    return MedianOf5($A[l, \dots, r]$)
else
    $A' \leftarrow$ MedianOf5Array($A[l, \dots, r]$)
    return MMSelect($A', 1, |A'|, \left\lfloor \frac{|A'|}{2} \right\rfloor$)
```

How good is this?



- Number groups of five: $\lceil \frac{n}{5} \rceil$, without median group: $\lceil \frac{n}{5} \rceil - 1$
- Minimal number groups left / right of Mediangroup $\left\lfloor \frac{1}{2}\left(\left\lceil \frac{n}{5} \right\rceil - 1\right)\right\rfloor$
- Minimal number of points less than / greater than m

$$3\left\lfloor \frac{1}{2}\left(\left\lceil \frac{n}{5} \right\rceil - 1\right)\right\rfloor \geq 3\left\lfloor \frac{1}{2}\left(\frac{n}{5} - 1\right)\right\rfloor \geq 3\left(\frac{n}{10} - \frac{1}{2} - 1\right) > \frac{3n}{10} - 6$$

(Fill rest group with points from the median group)

$\Rightarrow$ Recursive 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:[7] choose c large enough such that

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

Induction hypothesis: $H(n)$

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

Induction step: $H(k)_{k<n} \to H(n)$

$$\begin{aligned} T(n) &\leq T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\left\lceil \frac{7n}{10} + 6 \right\rceil\right) + d \cdot n \\ &\leq c \cdot \left\lceil \frac{n}{5} \right\rceil + c \cdot \left\lceil \frac{7n}{10} + 6 \right\rceil + d \cdot n \qquad \text{(for } n > 20\text{).} \end{aligned}$$

[7] It will turn out in the induction step that the base case has to hold of some fixed $n_0 > 0$. Because an arbitrarily large value can be chosen for c and because there is a limited number of terms, this is a simple extension of the base case for $n = 1$

Proof

Induction step:

$$T(n) \overset{n>20}{\leq} c \cdot \left\lceil \frac{n}{5} \right\rceil + c \cdot \left\lceil \frac{7n}{10} + 6 \right\rceil + d \cdot n$$
$$\leq c \cdot \frac{n}{5} + c + c \cdot \frac{7n}{10} + 6c + c + d \cdot n = \frac{9}{10} \cdot c \cdot n + 8c + d \cdot n.$$

To show

$$\exists n_0, \exists c \quad | \quad \frac{9}{10} \cdot c \cdot n + 8c + d \cdot n \leq cn \quad \forall n \geq n_0$$

thus

$$8c + d \cdot n \leq \frac{1}{10} cn \quad \Leftrightarrow \quad n \geq \frac{80c}{c - 10d}$$

Set, for example $c = 90d, n_0 = 91$ $\quad \Rightarrow T(n) \leq cn \; \forall \; n \geq n_0$ ■

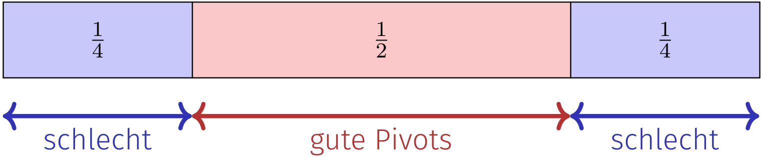
Result

Theorem 11

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



$\frac{1}{4}$ $\frac{1}{2}$ $\frac{1}{4}$

schlecht gute Pivots schlecht

7.1 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

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

8. Sorting I

Simple Sorting

8.1 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";
```

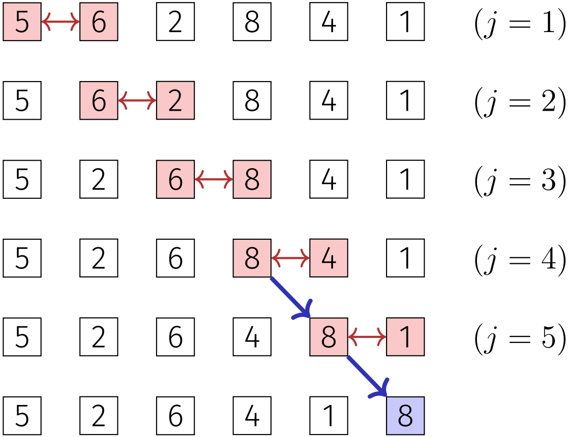
Observation

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

⇒ 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



5 ↔ 6 2 8 4 1 ($j = 1$)

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

- Not sorted! ☹.
- But the greatest element moves to the right
  ⇒ 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($A[j], A[j+1]$);
```

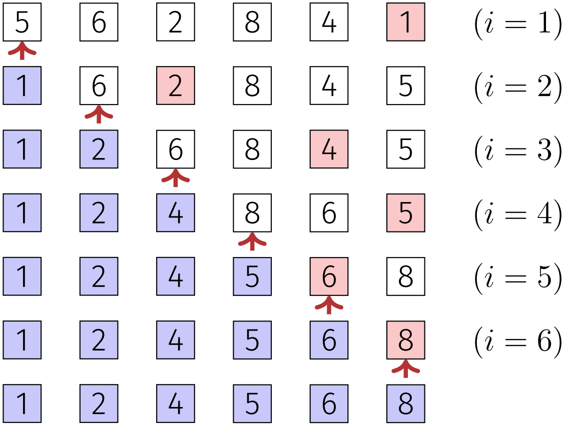
Analysis

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



5 6 2 8 4 1 $(i = 1)$

1 6 2 8 4 5 $(i = 2)$

1 2 6 8 4 5 $(i = 3)$

1 2 4 8 6 5 $(i = 4)$

1 2 4 5 6 8 $(i = 5)$

1 2 4 5 6 8 $(i = 6)$

1 2 4 5 6 8

- 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. $(i = n)$

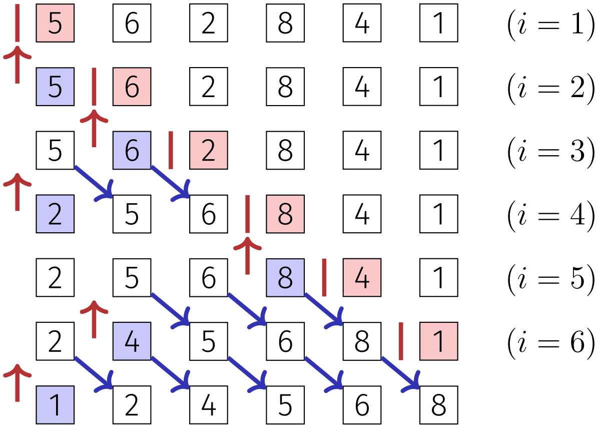
Algorithm: Selection Sort

```
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



5 6 2 8 4 1 $(i = 1)$

5 6 2 8 4 1 $(i = 2)$

5 6 2 8 4 1 $(i = 3)$

2 5 6 8 4 1 $(i = 4)$

2 5 6 8 4 1 $(i = 5)$

2 4 5 6 8 1 $(i = 6)$

1 2 4 5 6 8

- 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

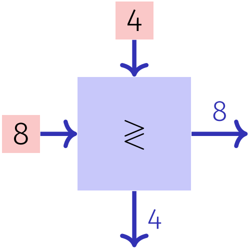
```
Input:  Array $A = (A[1], \dots, A[n])$, $n \geq 0$.
Output: Sorted Array $A$
for $i \leftarrow 2$ to $n$ do
    $x \leftarrow A[i]$
    $p \leftarrow$ BinarySearch($A, 1, i-1, x$); // Smallest $p \in [1, i]$ with $A[p] \geq x$
    for $j \leftarrow i-1$ downto $p$ do
        $A[j+1] \leftarrow A[j]$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

Analysis

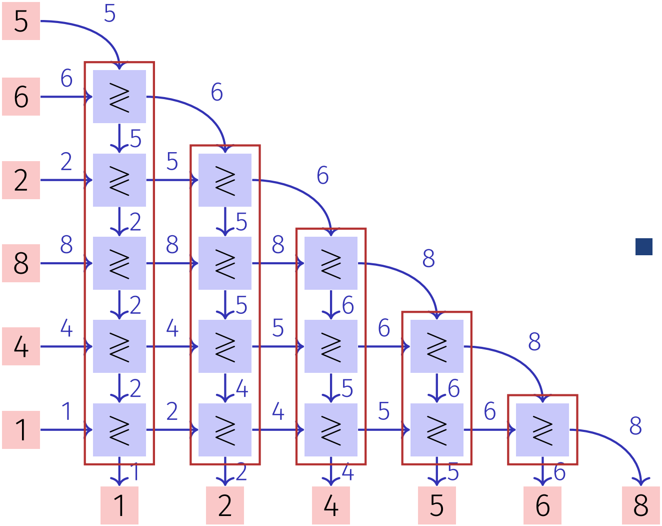
Number comparisons in the worst case:
$\sum_{k=1}^{n-1} a \cdot \log k = a \log((n-1)!) \in \Theta(n \log n)$.
Number swaps in the worst case $\sum_{k=2}^{n}(k-1) \in \Theta(n^2)$

Different point of view

Sorting node:

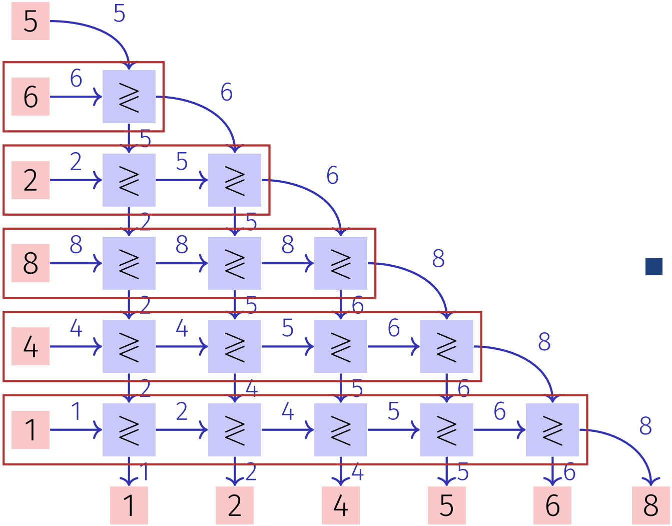


Different point of view



- 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. [8]

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

Shellsort (Donald Shell 1959)

Intuition: moving elements far apart takes many steps in the naive methods from abobe

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

9. Sorting II

Mergesort, Quicksort

9.1 Mergesort

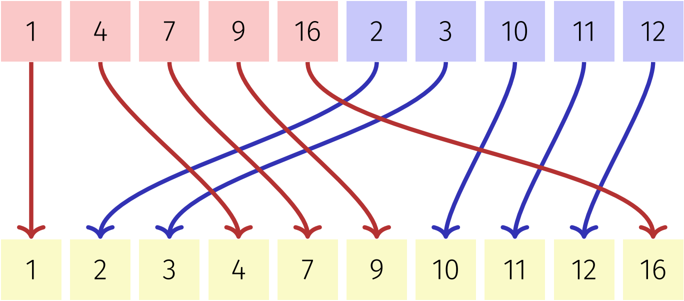
[Ottman/Widmayer, Kap. 2.4, Cormen et al, Kap. 2.3],

Mergesort

Divide and Conquer!

- Assumption: two halves of the array A are already sorted.
- Minimum of A can be evaluated with a single element comparison.
- 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 $B[1,\ldots,k]$ is sorted and $B[k] \leq A[i]$, if $i \leq m$ and $B[k] \leq A[j]$ if $j \leq r$.

Proof by induction:

Base case: the empty array $B[1,\ldots,0]$ is trivially sorted.

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

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

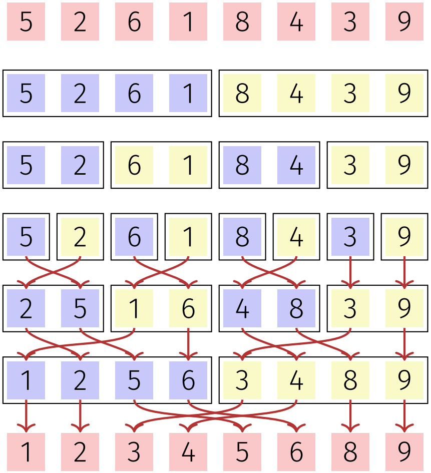
Analysis (Merge)

Lemma 12

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



5 2 6 1 8 4 3 9

5 2 6 1 | 8 4 3 9

5 2 | 6 1 | 8 4 | 3 9

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

2 5 | 1 6 | 4 8 | 3 9

1 2 5 6 | 3 4 8 9

1 2 3 4 5 6 8 9

Split

Split

Split

Merge

Merge

Merge

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

```
Input:  Array $A$ with length $n$. $1 \le l \le r \le n$
Output: $A[l, \dots, r]$ 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.

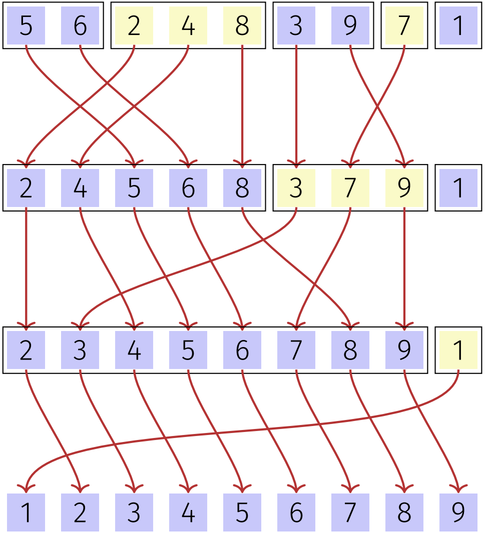
Natural 2-way mergesort

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.

9.2 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?

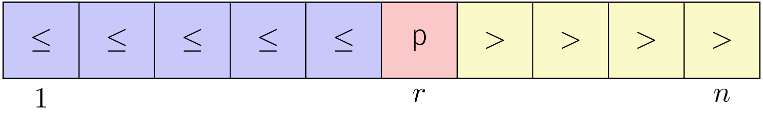
Make sure that the left part contains only smaller elements than the right part.

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



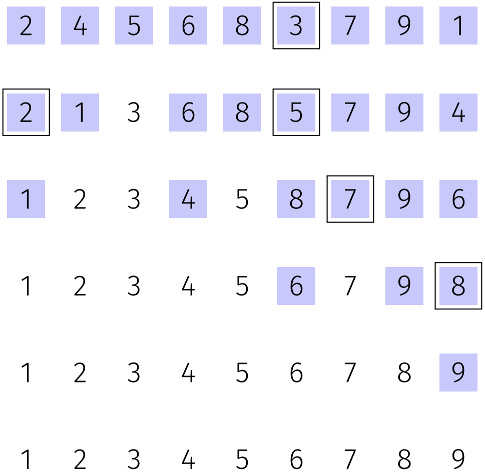
Algorithm `Partition`(A, l, r, p)

```
Input: Array $A$, that contains the pivot $p$ in $A[l, \dots, r]$ at least once.
Output: Array $A$ partitioned in $[l, \dots, r]$ 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, r)

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

Quicksort (arbitrary pivot)



2 4 5 6 8 $\boxed{3}$ 7 9 1

$\boxed{2}$ 1 3 6 8 $\boxed{5}$ 7 9 4

1 2 3 4 5 8 $\boxed{7}$ 9 6

1 2 3 4 5 6 7 9 $\boxed{8}$

1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9

Analysis: number comparisons

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

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

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:

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

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: $(n-1 \to n)$

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 13

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

Practical Considerations

Worst case recursion depth $n - 1$[9]. 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.

[9] stack overflow possible!

Quicksort with logarithmic memory consumption

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

The call of Quicksort($A[l, \ldots, r]$) 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.

9.3 Appendix

Derivation of some mathematical formulas

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

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

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

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

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

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

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

[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 $\frac{f_{n+1}}{f_n} \underset{n\to\infty}{\longrightarrow} \lambda$, 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)

L'Hospital's rule

Theorem 14

Let $f, g : \mathbb{R}^+ \to \mathbb{R}$ differentiable functions with $g'(x) \neq 0 \; \forall x > 0$.
If

$$\lim_{x \to \infty} f(x) = \lim_{x \to \infty} g(x) = 0,$$

or

$$\lim_{x \to \infty} f(x) = \pm\infty \textit{ and } \lim_{x \to \infty} g(x) = \pm\infty,$$

then

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = \lim_{x \to \infty} \frac{f'(x)}{g'(x)}$$

if the limit of $f'(x)/g'(x)$ exists

L'Hospital's rule

Example

Es gilt $\log^k(n) \in o(n)$, because with $f(x) = \log^k(x)$, $g(n) = x$, we can apply L'Hospital's rule and get

$$\lim_{x\to\infty} \frac{\log^k(x)}{x} = \lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)} = \lim_{x\to\infty} k\frac{\log^{k-1}(x)}{x}$$

After k iterations we get

$$\lim_{x\to\infty} \frac{\log^k(x)}{x} = \lim_{x\to\infty} k!\frac{1}{x} = 0.$$

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:

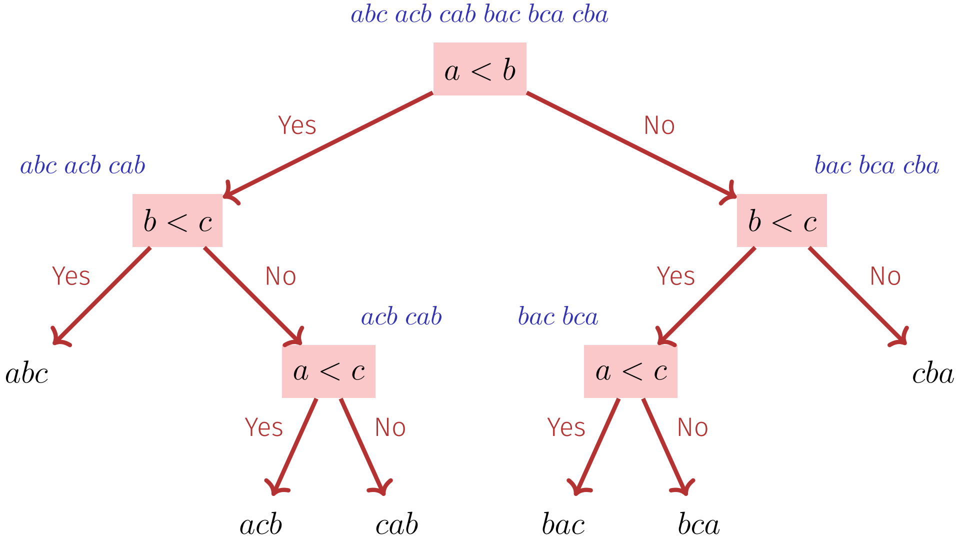
Theorem 15

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 knows 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.[10]

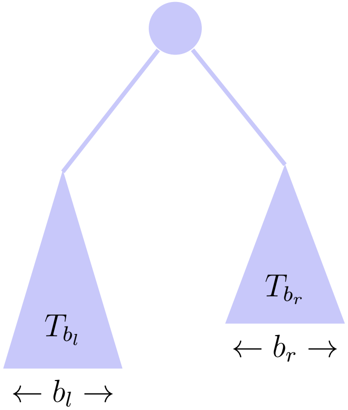
The height of a binary tree with L leaves is at least $\log_2 L$. $\Rightarrow$ The height of the decision tree $h \geq \log n! \in \Omega(n \log n)$.

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 $M(n) \in \Omega(n \log n)$.

[10] 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 smallest b with $m(T_b) < \log b \Rightarrow b \geq 2$
- $b_l + b_r = b$ with $b_l > 0$ and $b_r > 0 \Rightarrow$ $b_l < b, b_r < b \Rightarrow m(T_{b_l}) \geq \log b_l$ 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 $\left(f''(x) = 1/x > 0\right)$ and for a convex function it holds that $f((x+y)/2) \leq 1/2 f(x) + 1/2 f(y)$ $\left(x = 2b_l,\right.$ $\left.y = 2b_r\right)$.[11] Enter $x = 2b_l$, $y = 2b_r$, and $b_l + b_r = b$.

[11] 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

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

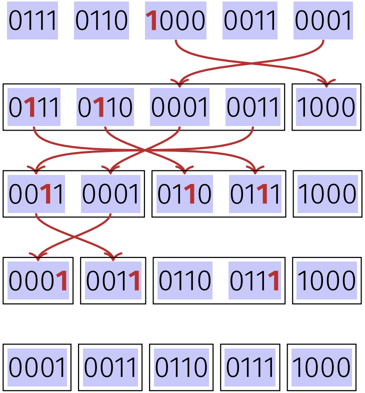
then it holds that $x < y$.

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



0111 0110 1000 0011 0001

0111 0110 0001 0011 1000

0011 0001 0110 0111 1000

0001 0011 0110 0111 1000

0001 0011 0110 0111 1000

Algorithm RadixExchangeSort(A, l, r, b)

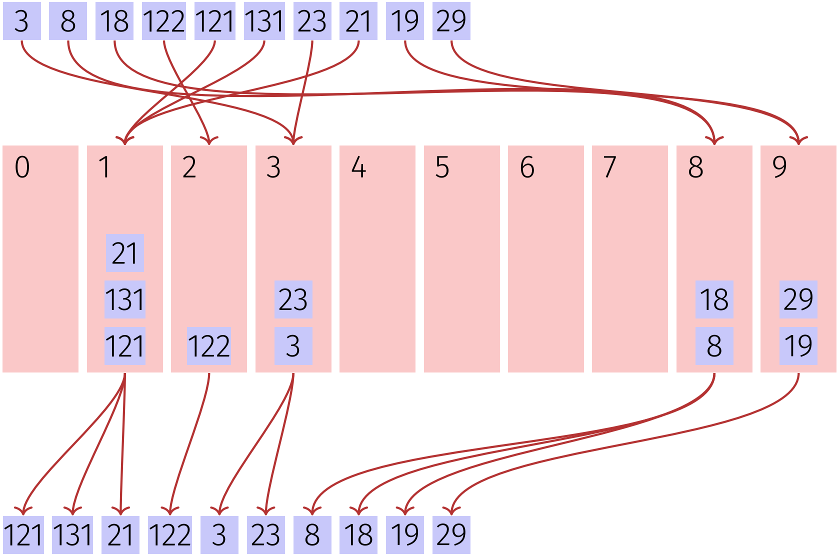
```
Input:  Array $A$ with length $n$, left and right bounds $1 \leq l \leq r \leq n$, bit position $b$
Output: Array $A$, sorted in the domain $[l, r]$ by bits $[0, \ldots, b]$ .
if $l < r$ and $b \geq 0$ then
    $i \leftarrow l - 1$
    $j \leftarrow r + 1$
    repeat
        repeat $i \leftarrow i + 1$ 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

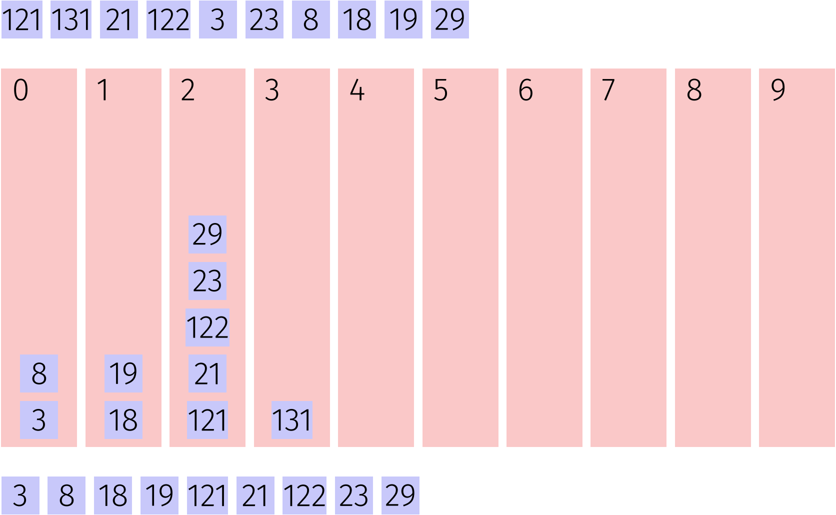
RadixExchangeSort provides recursion with maximal recursion depth = maximal number of digits p.

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

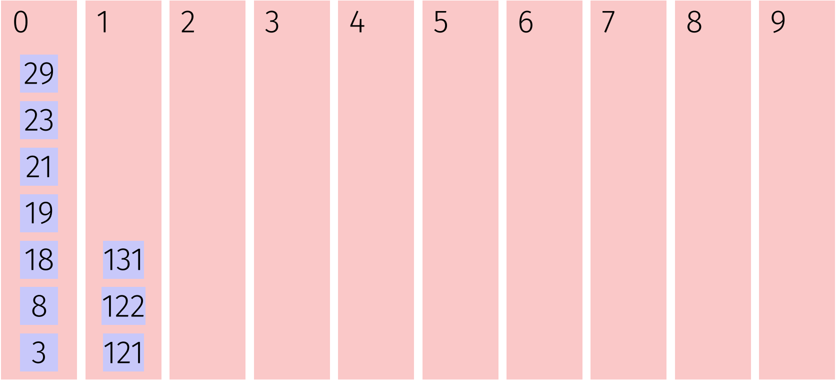
0 1 2 3 4 5 6 7 8 9

29
23
122
8 19 21
3 18 121 131

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

Bucket Sort – Different Assumption

Hypothesis: uniformly distributed data e.g. from $[0, 1)$

```
Input:  Array $A$ with length $n$, $A_i \in [0, 1)$, constant $M \in \mathbb{N}^+$
Output: Sorted array
$k \leftarrow \lceil n/M \rceil$
$B \leftarrow$ new array of $k$ empty lists
for $i \leftarrow 1$ to $n$ do
    $B[\lfloor A_i \cdot k \rfloor]$.append($A[i]$)
for $i \leftarrow 1$ to $k$ do
    sort $B[i]$ // e.g. insertion sort, running time $\mathcal{O}(M^2)$
return $B[0] \circ B[1] \circ \cdots \circ B[k]$ // concatenated
```

Expected asymptotic running time $\mathcal{O}(n)$ (Proof in Cormen et al, Kap. 8.4)

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]

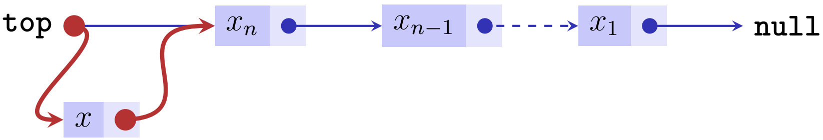
Abstract Data Types

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.

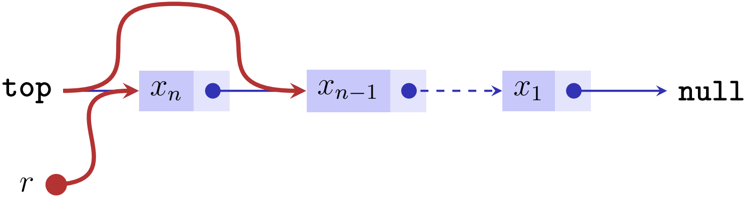
Implementation Push



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

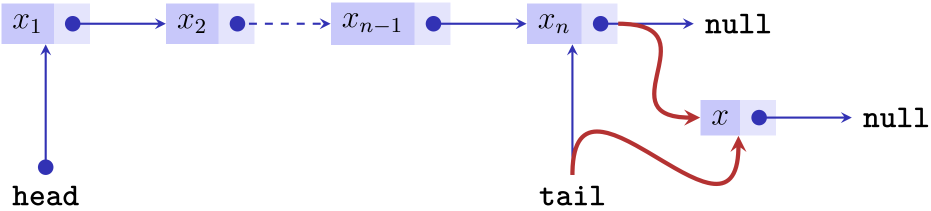
Each of the operations **`push`**, **`pop`**, **`top`** and **`isEmpty`** on a stack can be executed in $\mathcal{O}(1)$ steps.

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.

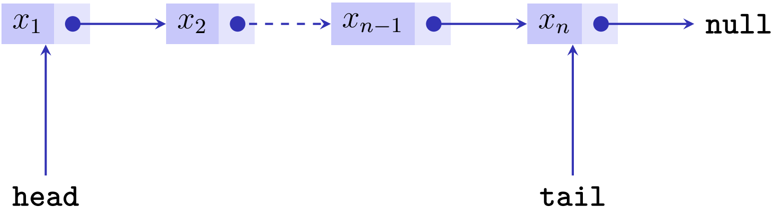
Implementation Queue



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**.

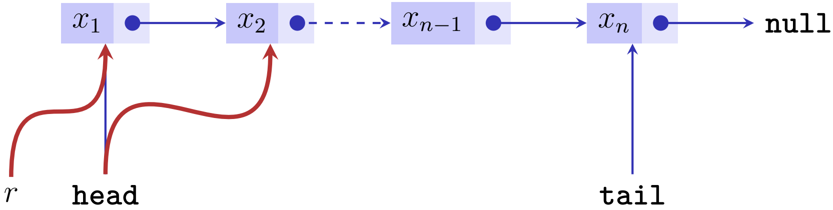
Invariants



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):

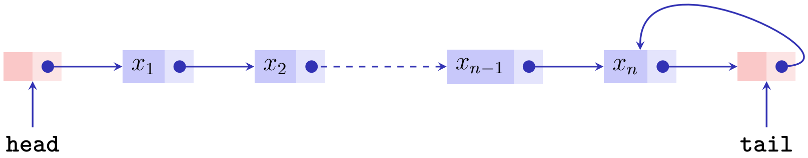
1. Store pointer to **head** in r. If $r =$ **null**, then return r .
2. Set the pointer of **head** to **head.next**.
3. Is now **head** $=$ **null** then set tail to **null**.
4. Return the value of r.

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).

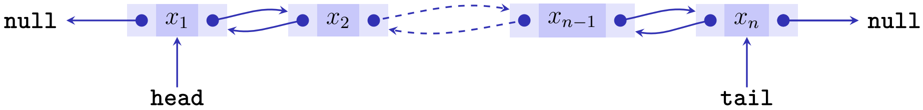


Advantage: less special cases

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

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(k, 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)

- With n operations at most n elements can be pushed onto the stack. Therefore a maximum of n elements can be removed from the stack
- For the total costs we get

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

and thus

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

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** Φ_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: $\Phi_i \geq \Phi_0 \; \forall i$.
Amortized costs of the ith 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 Φ_i = number element on the stack.

- $\texttt{push}(x, S)$: real costs $t_i = 1$. $\Phi_i - \Phi_{i-1} = 1$. Amortized costs $a_i = 2$.
- $\texttt{pop}(S)$: real costs $t_i = 1$. $\Phi_i - \Phi_{i-1} = -1$. Amortized costs $a_i = 0$.
- $\texttt{multipop}(k, S)$: 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. [12]

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

Example Binary Counter

Given: counter with k bits. We count from 0 to $n-1$ where $n=2^k$.

Wanted: average counting costs as number bit-flips per operation

Naive: per operation maximally k bit-flips. Thus, in total $\mathcal{O}(n \cdot k)$ operations or $\mathcal{O}(\log n)$ bitflips per count operation on average.

Example Binary Counter

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

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

$$\Rightarrow t_i = l + 1$$

Binary Counter: Aggregate Analysis

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

Observation

- Bit 0 flips each time
- Bit 1 flips each 2. time
- Bit 2 flips each 4. time
- ...

Thus, 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 operation: $\mathcal{O}(1)$ bit flips.

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

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.

Amortised costs $a_i = 2$:

- 1 CHF real cost for setting $0 \mapsto 1$
- plus 1 CHF to deposit on the account.
- Every reset $1 \to 0$ can be paid from the account.

Binary Counter: Potential Method

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

potential function Φ_i: number of 1-bits of x_i.

$$\begin{aligned}
0 = \Phi_0 &\leq \Phi_i \, \forall i, \\
\Phi_i - \Phi_{i-1} &= 1 - l_i, \\
\Rightarrow a_i &= t_i + \Phi_i - \Phi_{i-1} \\
&= l_i + 1 + (1 - l_i) \\
&= 2.
\end{aligned}$$

Amortized constant cost per 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 ⇒ error messsage.
- **`delete`**(k, D): Delete k from the dictionary D. Not existing ⇒ 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	$\mathcal{O}(1)$[13]	☺
Delete	$\mathcal{O}(n)$	☹

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

13.1 Self Ordering

Self Ordered Lists

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

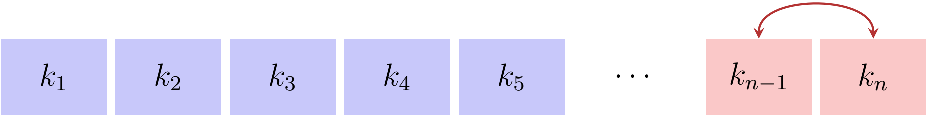
Idea: Try to order the list elements such that accesses over time are possible in a faster way

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

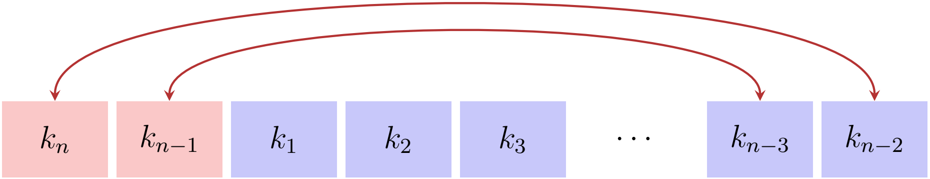
Transpose:



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:

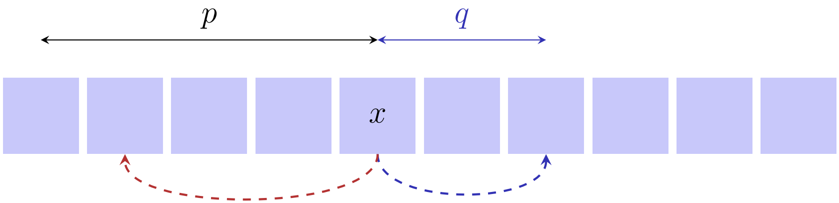
- MTF and A may only move the accessed element.
- MTF and A start with the same list.

Let M_k and A_k designate the lists after the kth 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

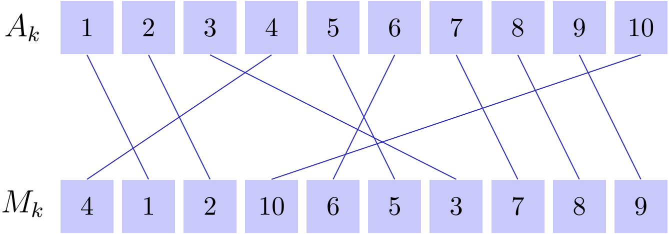
Let an arbitrary sequence of search requests be given and let $G_k^{(M)}$ and $G_k^{(A)}$ the costs in step k for Move-to-Front and A, respectively. Want estimation of $\sum_k G_k^{(M)}$ compared with $\sum_k G_k^{(A)}$.

$\Rightarrow$ Amortized analysis with potential function Φ.

Potential Function

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

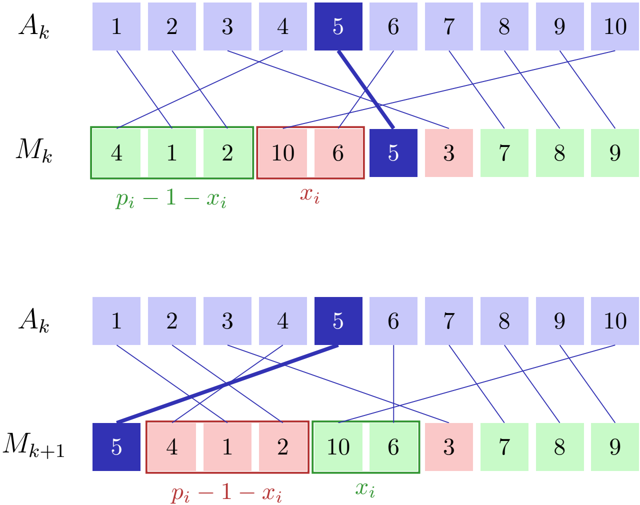
Inversion = Pair x, y such that for the positions of a and y
$\left(p^{(A)}(x) < p^{(A)}(y)\right) \neq \left(p^{(M)}(x) < p^{(M)}(y)\right)$



#inversion = #crossings

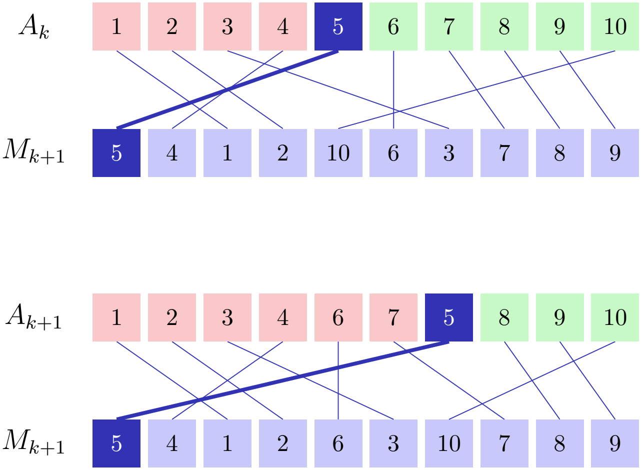
Estimating the Potential Function: MTF

- 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

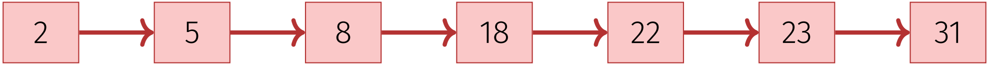
Summing up costs

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

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

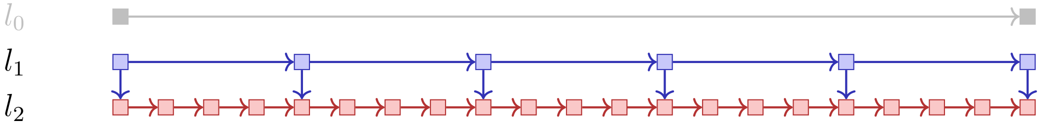
13.2 Skip Lists

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: $n_2 = 1$

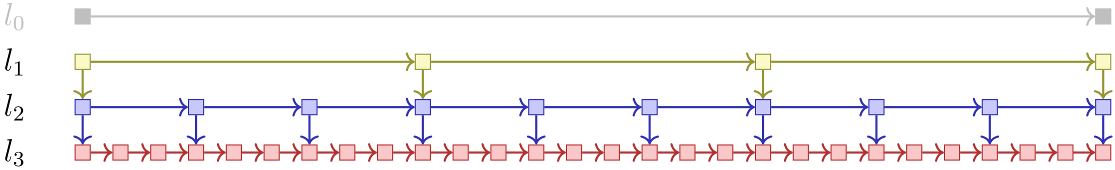
⇒ Search for element / insertion position: worst-case $\frac{n_0}{n_1} + \frac{n_1}{n_2}$.

⇒ Best Choice for[14] n_1: $n_1 = \frac{n_0}{n_1} = \sqrt{n_0}$.

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

[14] 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[15].

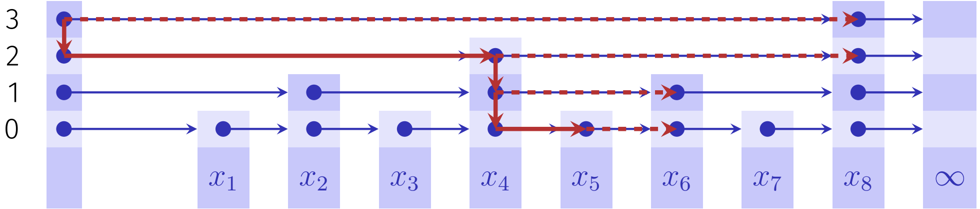
Assumption $n = 2^k$

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

[15] (Derivation: Appendix)

Search in a Skiplist

Perfect skip list



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

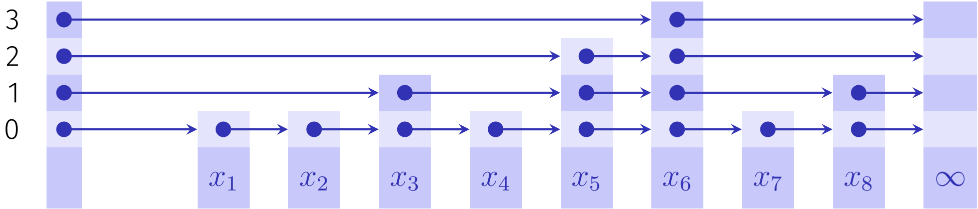
Example: search for a key x with $x_5 < x < x_6$.

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 16

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.

13.3 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:

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

Minimize f for $(n_1, \dots, n_{k-1})$: $\frac{\partial f(\vec{n})}{\partial n_t} = 0$ 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 $\Rightarrow \frac{n_t}{n_0} = \frac{n_t}{n_{t-1}} \frac{n_{t-1}}{n_{t-2}} \dots \frac{n_1}{n_0} = \left(\frac{n_1}{n_0}\right)^t$
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 [Ottman/Widmayer, Kap. 4.1-4.3.2, 4.3.4, Cormen et al, Kap. 11-11.4]

Introductory Example

Gloal: Efficient management of a table of all n ETH students.

Requirement: Fast access (insertion, removal, find) of a dataset by name.

Dictionary

Abstract Data Type (ADT) D to manage items[16] $i = (k, v)$ with keys $k \in \mathcal{K}$ with operations:

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

[16] 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> colours = {
  {"RED","#FF0000"}, {"GREEN","#00FF00"}
};

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

std::cout << "The hex value of color red is: "
          << colours["RED"] << "\n";

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

Motivation/Applications

Perhaps *the* most popular data structure.

- Supported in many programming languages ($\mathrm{C}{+}{+}$, Python, Javascript, Java, C#, Ruby, . . .)
- Obvious use
  - Databases
  - Symbol tables in compilers and interpreters
  - Objects in dynamically typed languages, e.g. Python, Javascript
- Less obvious
  - Substring search (z.B. Rabin-Karp)
  - String similarity (e.g. comparing documents, DNA)
  - File synchronisation (e.g. git, rsync)
  - Cryptography (e.g. identification, authentification)

Idea: Keys as Indices

Index	Item
0	-
1	-
2	-
3	[3,value(3)]
4	-
5	-
⋮	⋮
k	[k,value(k)]
⋮	⋮

Problems

1. Keys must be non-negative integers
2. Large key-range ⇒ large array

Solution to the first problem: Prehashing

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: $x = y \Leftrightarrow ph(x) = ph(y)$
- In practice: 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[17]

[17] 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

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

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

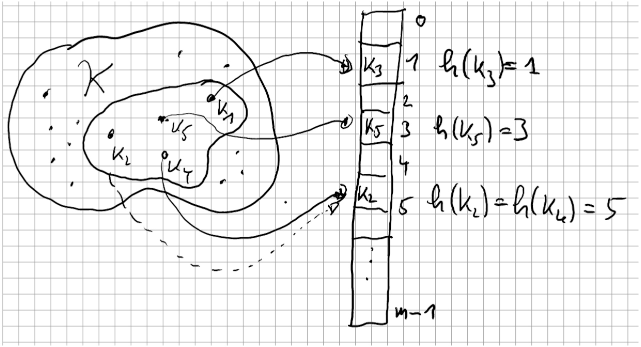
b Word-size of the system (e.g. 32 or 64)

Example with $b = 31$, $w = 32$, ASCII values s_i

Anna $\mapsto 92966272$
Anne $\mapsto 96660356$
Heinz-Harald $\mapsto 81592996699304236533 \bmod 2^{32} = 631641589$

Solution to the second problem: Hashing

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



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

Nomenclature

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

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

Usually $|\mathcal{K}| \gg m$. 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.

Examples of popular Hash Functions

Division method

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

Ideal: m prime number, not too close to powers of 2 or 10 (see e.g. Cormen et al. “Introduction to Algorithms”, Donald E. Knuth “The Art of Computer Programming”).

But often: $m = 2^r - 1$ ($r \in \mathbb{N}$), due to growing tables by doubling (more later).

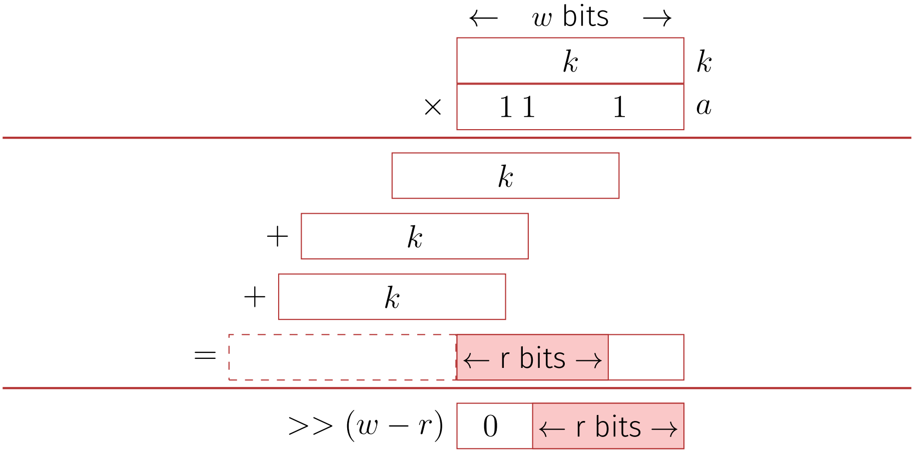
Examples of popular Hash Functions

Multiplication method

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

- 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.
- Table size $m = 2^r$, $w =$ size of the machine word in bits.
- Multiplication adds k along all bits of a, integer division by 2^{w-r} and $\bmod m$ extract the upper r bits.
- Written as code very simple: `a * k >> (w-r)`

Illustration



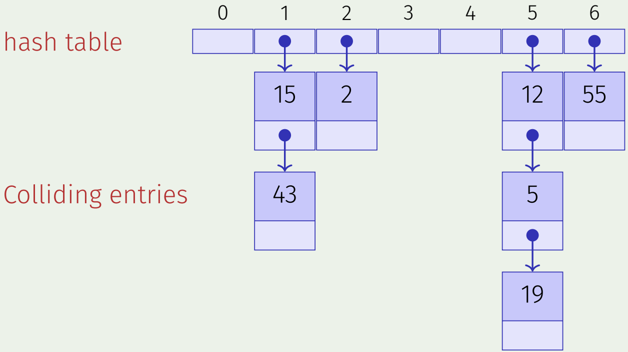
$\leftarrow$ w bits $\rightarrow$
k
k
$\times$
1 1 1
a
k
$+$
k
$+$
k
$=$
$\leftarrow$ r bits $\rightarrow$
$>> (w - r)$
0
$\leftarrow$ r bits $\rightarrow$

Resolving Collisions: Chaining

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

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

Direct Chaining of the Colliding entries



Algorithm for Hashing with Chaining

Let H be a hash table with collision lists.

- **`insert`**(H, 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`**(H, 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`**(H, 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

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

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

Simple Uniform Hashing

Theorem 17

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 $\Theta(1+\alpha)$.

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 entirely.
   $\Rightarrow$ 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:

- Load factor greater 1 possible (more entries than hash table slots)
- Removing keys is straightforward (relative to alternative introduced later)

Disadvantages:

- Linear runtime in case of degenerated hash tables with long collision chains
- (Memory consumption of the chains)

Better: reduce probability of collisions

[Variant:Indirect Chaining]

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

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

Indirect chaining of colliding entries

hash table

Colliding entries

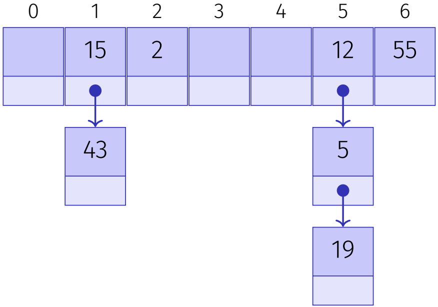


Table size increase

- We do not know beforehand how large n will be
- We would like $m = \Theta(n)$ at all times (hash table size m linearly dependent on no. of entries n, i.e. not arbitrarily large)

Adjust table size $\rightarrow$ 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 $\Theta(n + m + m')$

How to choose m'?

Table size increase

Double the table size, depending on the load factor.

$\Rightarrow$ Amortized analysis yields: Each operation of hashing with chaining has expected amortized costs $\Theta(1)$.

Open Addressing

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 $\{0, 1, \ldots, m-1\}$

Notational clarification: this method uses *open addressing* (meaning that the positions in the hash table are not fixed), but it is nonetheless a *closed hashing* procedure (entries stay in the hash table).

Algorithms for open addressing

Let H be a hash table (without collision lists).

- **`insert`**(H, 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`**(H, k) Traverse table entries according to $S(k)$. If k is found, return data associated to k. Otherwise return an empty element **`null`**.
- **`delete`**(H, k) Search k in the table according to $S(k)$. If k is found, replace it with a special key **`removed`**.

Linear Probing

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

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

Key 12 , 55 , 5 , 15 , 2 , 19

0	1	2	3	4	5	6
● 5	● 15	● 2	19		● ● 12	● ● 55

[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?

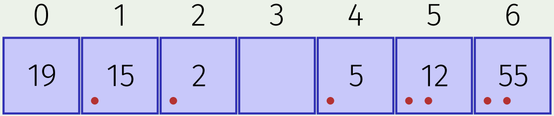
Primary clustering: similar hash addresses have similar probing sequences ⇒ long contiguous areas of used entries.

Quadratic Probing

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

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

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



0	1	2	3	4	5	6
19	15	2		5	12	55
	●	●		●	● ●	● ●

[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 $h'(k)$. $s(k,j) = h(k) + j \cdot h'(k)$.
$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m-1)h'(k)) \mod m$

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

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:

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

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 18

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: Proof of the theorem

Random Variable X: Number of probings when searching without success.

$$\mathbb{P}(X \geq i) \overset{*}{=} \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2}$$

$$\overset{**}{\leq} \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}. \qquad (1 \leq i \leq m)$$

* : Event A_j: slot used during step j.
$\mathbb{P}(A_1 \cap \cdots \cap A_{i-1}) = \mathbb{P}(A_1) \cdot \mathbb{P}(A_2 | A_1) \cdot ... \cdot \mathbb{P}(A_{i-1} | A_1 \cap \cdots \cap A_{i-2})$,

** : $\frac{n-1}{m-1} < \frac{n}{m}$ because $n < m$: $\frac{n-1}{m-1} < \frac{n}{m} \Leftrightarrow \frac{n-1}{n} < \frac{m-1}{m} \Leftrightarrow 1 - \frac{1}{n} < 1 - \frac{1}{m} \Leftrightarrow n < m$
$(n > 0, m > 0)$

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

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

[Successful search of Uniform Open Hashing]

Theorem 19

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

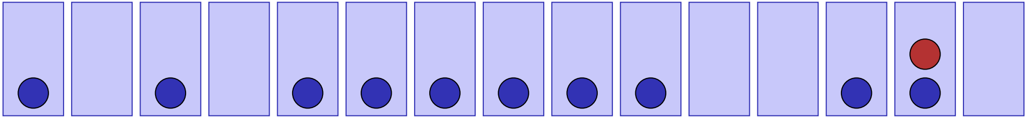
α: load factor.
C_n: Number steps successful search,
C'_n: Number steps unsuccessful search

14.8 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]

$\mathbb{P}(\text{no collision}) = \frac{m}{m} \cdot \frac{m-1}{m} \cdot \dots \cdot \frac{m-n+1}{m} = \frac{m!}{(m-n)! \cdot m^m}$.

Let $a \ll m$. With $e^x = 1 + x + \frac{x^2}{2!} + \dots$ approximate $1 - \frac{a}{m} \approx e^{-\frac{a}{m}}$. This yields:

$$1 \cdot \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{2}{m}\right) \cdot \dots \cdot \left(1 - \frac{n-1}{m}\right) \approx e^{-\frac{1+\dots+n-1}{m}} = e^{-\frac{n(n-1)}{2m}}.$$

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 $\mathbb{E}(X) < \infty$

$$\begin{aligned}
\mathbb{E}(X) &\overset{(def)}{=} \sum_{x=0}^{\infty} x\mathbb{P}(X = x) \\
&\overset{\text{Counting}}{=} \sum_{x=1}^{\infty} \sum_{y=x}^{\infty} \mathbb{P}(X = y) \\
&= \sum_{x=0}^{\infty} \mathbb{P}(X > x)
\end{aligned}$$

15. C++ advanced (III): Functors and Lambda

Generic Programming: higher order functions

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";
}
```

Question: when does a call of the **filter** template function work?

Answer: works if the first argument offers an iterator and if the second argument can be applied to elements of the iterator with a return value 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
```

Context and Overview

- Recent C++-lecture: make code parametric on the data it operates
  - `Pair<T>` for element types `T`
  - `print<C>` for iterable containers `C`
- Now: make code parameteric on a (part of) the algorithm
  - `filter(container, predicate)`
  - `apply(signal, transformation/filter)`
- We learn about
  - Higher-order functions: fucntions that take functions as arguments
  - Functors: objects with overloaded function operator `()`.
  - Lambda-Expressions: anonymous functors (syntactic sugar)
  - Closures: lambdas that capture their environment

Functors: Motivation

```
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";
}
```

- Requirements on `f`: must be callable/applicable (...)
- `f` must be a kind of function ⇒ `filter` is a function that takes a function as argument
- A function taking (or returning) a function is called a **higher order function**
- Higher order functions are parameteric in their functionality (or they generate functions)

What if...

the filter should be more flexible:

```
template <typename T, typename function>
void filter(const T& collection, function f);

template <typename T>
bool largerThan(T x, T y){
    return x > y;
}

std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int val = 8;
~~filter(a,largerThan<int>( ? ,val));~~ (No, this does not exist)
```

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
```

Observations

```
template <typename T>
class GreaterThan{
    T value;
public:
    GreaterThan(T x):value{x}{}

    bool operator() (T par) const{
        return par > value;
    }
};
```

Need to give the predicate a **name**

- Often unnecessary, many are unsed only once
- Descriptive names not always possible
- Distance (in the code) between declaration and use

Overhead: stateful predicates as functors

- cumbersome for what is ultimately only `par > value`

The same with a Lambda-Expression

Anonymous functions with **lambda-expressions**:

```
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;} );
```

That is just **syntactic sugar**, from which the compiler generates a suitable functor.

Interlude: Sorting with Custom Comparator

- **`std::sort`** is generic
  - in the iterator-type
  - in the values iterated over
  - in the used comparator

```
std::sort (v.begin(), v.end(), std::less());
```

- The comparator returns **`true`** if the elements are ordered as wished.

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.[18]

```
[]() {std::cout << "Hello World";}
```

- If no parameters and no explicit return type, then () can be omitted.

```
[]{std::cout << "Hello World";}
```

- `[...]` can never be omitted.

[18] 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

```cpp
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

Commonly Used: `std::foreach`

- Common task: iterate over a container and do something with each element

```
for (auto& name: names) {
  std::cout << name << ’ ’;
}
```

- This pattern is typically implemented as higher-order function **`for_each`**

```
for_each(names, [](auto name) {std::cout << name << " ";});
```

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 – Old School

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int sum = 0;
....
std::cout << sum << std::endl; // 83
```

- Task: increase **`sum`** for each call in **`for_each`**
- Problem: **`for_each`** requires accesst to the context (here **`sum`**)

Other example **`filter`**: store each element of a vector into a different vector according to some condition.

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
```

- Ok: solves the problem but does not operate on the **sum** variable

Sum of Elements – with References

```
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
```

Of course this works, very similarly, using pointers

Sum of Elements – with Λ

```
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;
```

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

```
std::vector<int> v = {1,2,4,8,16};

int elements=0;
int sum=0;
std::for_each(v.begin(), v.end(),
  [&] (int k) {sum += k; elements++;} // capture all by reference
)

std::cout << "sum=" << sum << " elements=" << elements << std::endl;
```

Output: `sum=31 elements=5`

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 action(){ [=] {i=42;}();}
};

mutant m;
m.action();
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

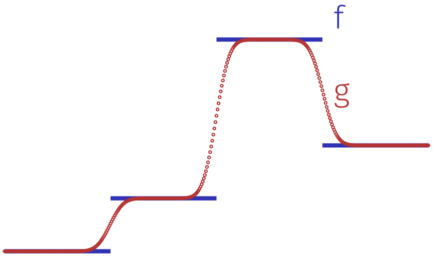
```
template <typename T>
auto toFunction(std::vector<T> v){
  return [v] (T x) -> double {
    int index = (int)(x+0.5);
    if (index < 0) index = 0;
    if (index >= v.size()) index = v.size()-1;
    return v[index];
  };
}
```

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

- Higher-order functions are parametric in their functionality: more flexible → more widely applicable → more code reuse
- Being able to pass around functions means being able to pass around entire computations.
- Lambda expressions facilitate the use of “one-off” functions, which are often used in combination with higher-order functions.
- Returning lambdas enables implementing function generators, that can generate whole families of functions.
- In C++, lambda expressions desugare into function objects (functors).
- Higher-order functions and lambdas are an important, and nowadays mainstream, building block of the functional programming paradigm.

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

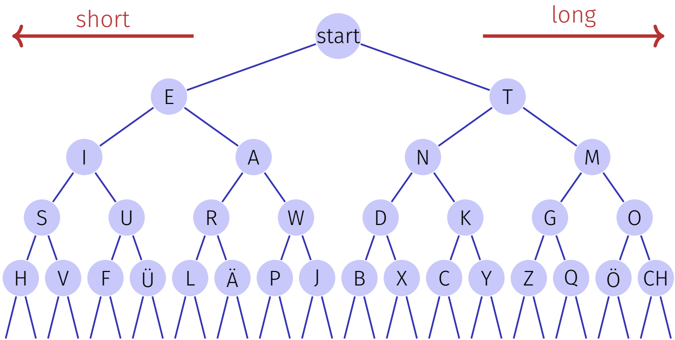
- Generalized lists: nodes can have more than one successor
- Special graphs: graphs consist of nodes and edges. A tree is a fully connected, directed, acyclic graph.

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

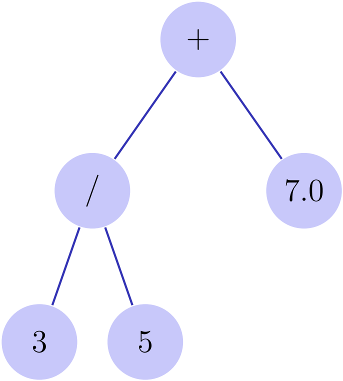
Examples



Morsealphabet

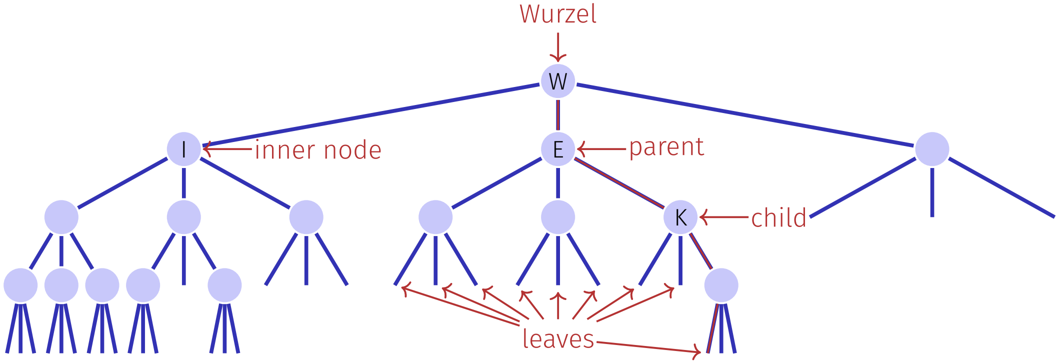
Examples

$3/5 + 7.0$



Expression tree

Nomenclature



- *Order* of the tree: maximum number of child nodes (here: 3)
- *Height* of the tree: maximum path length root to leaf (here: 4)

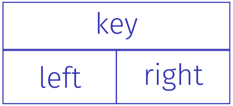
Binary Trees

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.



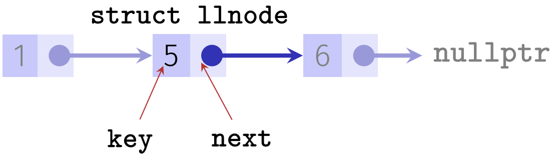
key

left

right

a leaf is represented by the **null**-pointer

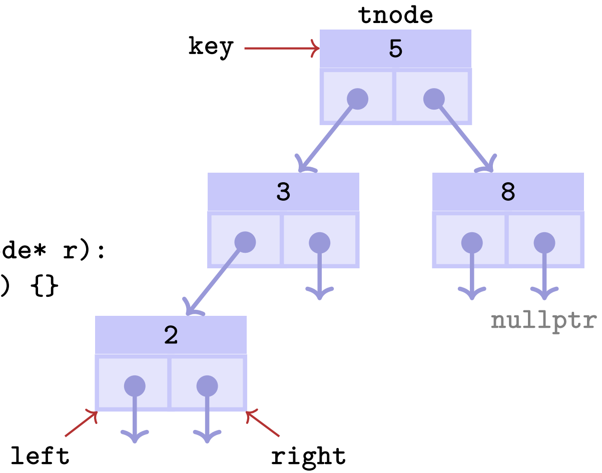
Recap: Linked-list Node in C++



```
struct llnode {
  int key;
  llnode* next;
  llnode(int k, llnode* n): key(k), next(n) {} // Constructor
};
```

Recap: Tree Nodes in $\mathrm{C}{+}{+}$

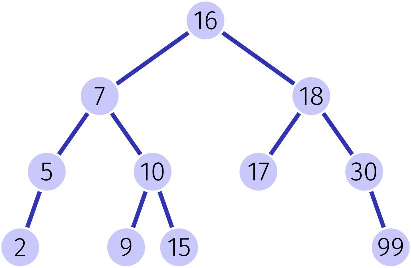
```
struct tnode {
  int key;
  tnode* left;
  tnode* right;
  tnode(int k, tnode* l, tnode* r):
    key(k), left(l), right(r) {}
};
```



Binary search tree

A *binary search tree* is a binary tree that fulfils the **search tree property**:

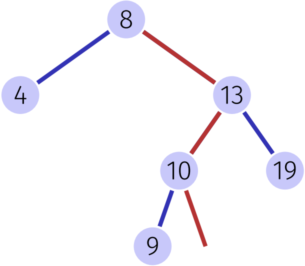
- Every node **`v`** stores a key
- Keys in left subtree **`v.left`** are smaller than **`v.key`**
- Keys in right subtree **`v.right`** are greater than **`v.key`**



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) → **null**

Searching in C++

```
bool contains(const llnode* root, int search_key) {
  while (root != nullptr) {
    if (search_key == root->key) return true;
    else if (search_key < root->key) root = root->left;
    else root = root->right;
  }

  return false;
}
```

Remarks (pot. also for subsequent code):

- `contains` would typically be a member of function of `struct tnode` or `class bin_search_tree` (→ slightly different signature)
- Recursive implementation also possible

Height of a tree

The height $h(T)$ of a binary tree T with root r is given by

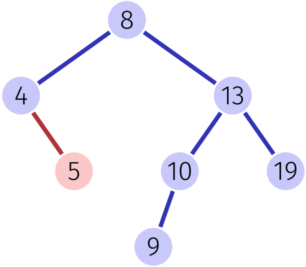
$$h(r) = \begin{cases} 0 & \text{if } r = \textbf{null} \\ 1 + \max\{h(r.\mathrm{left}), h(r.\mathrm{right})\} & \text{otherwise.} \end{cases}$$

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: e.g. output error
- If no success: insert the key at the leaf reached



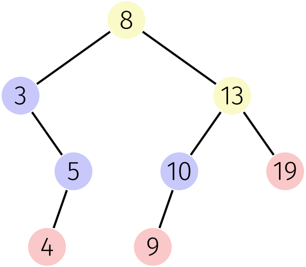
Insert (5)

Remove node

Three cases possible:

- Node has no children
- Node has one child
- Node has two children

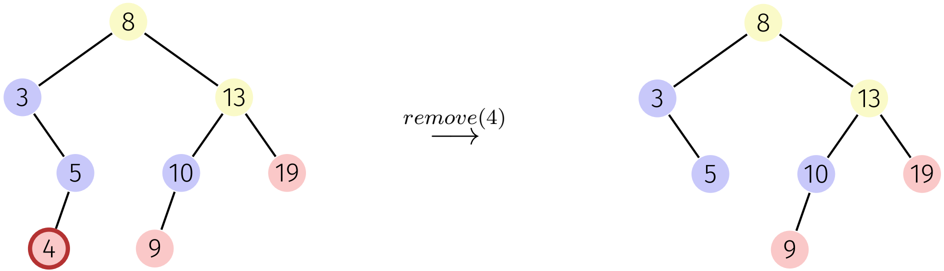
[Leaves do not count here]



Remove node

Node has no children

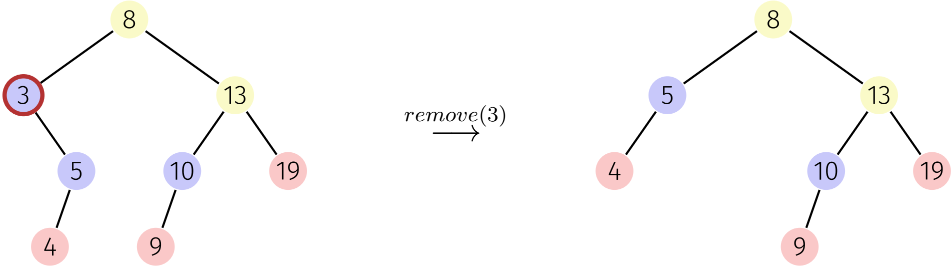
Simple case: replace node by leaf.



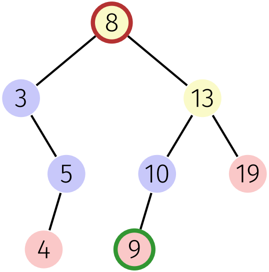
Remove node

Node has one child

Also simple: replace node by single child.



Remove node



Node `v` has two children

Requirements for replacement node `w`:

1. `w.key` is larger than all keys in `v.left`
2. `w.key` is smaller than all keys in `v.right`
3. ideally has not children

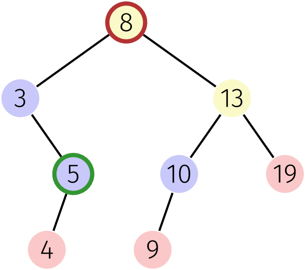
Observation: the smallest key in the right subtree `v.right` (here: 9) meets requirements 1, 2; and has at most one (right) child.

Solution: replace `v` by exactly this *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
```

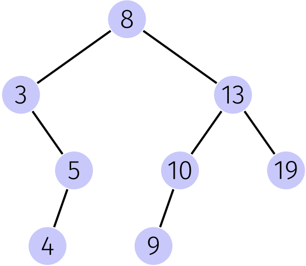
Analysis

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.

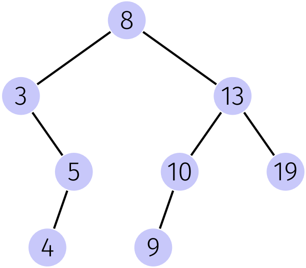
Traversal possibilities

- *preorder*:
  v, then $T_{\mathrm{left}}(v)$, then $T_{\mathrm{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- *postorder*:
  $T_{\mathrm{left}}(v)$, then $T_{\mathrm{right}}(v)$, then v.
  4, 5, 3, 9, 10, 19, 13, 8
- *inorder*:
  $T_{\mathrm{left}}(v)$, then v, then $T_{\mathrm{right}}(v)$.
  3, 4, 5, 8, 9, 10, 13, 19

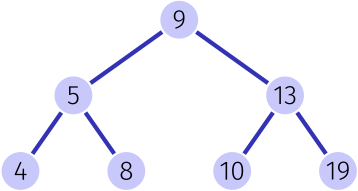


Further supported operations

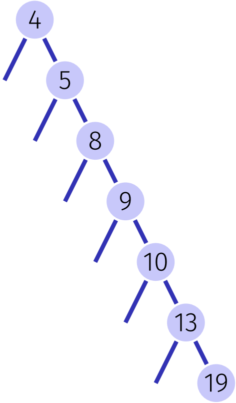
- *Min/Max(T)*: Query minimal/maximal value in $\mathcal{O}(h(T))$
- *ExtractMin/Max(T)*: Query and remove remove min/max in $\mathcal{O}(h(T))$
- *List(T)*: Output the sorted list of elements
- *Join(T_1, T_2)*: Merge two trees with *Max(T_1)* < *Min(T_2)* in $\mathcal{O}(h(T_1, T_2))$



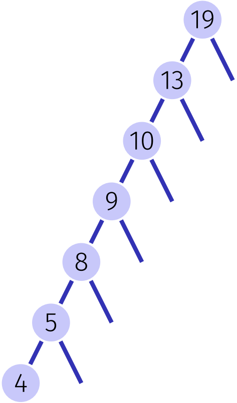
Search Trees: Balanced vs. Degenerated



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.

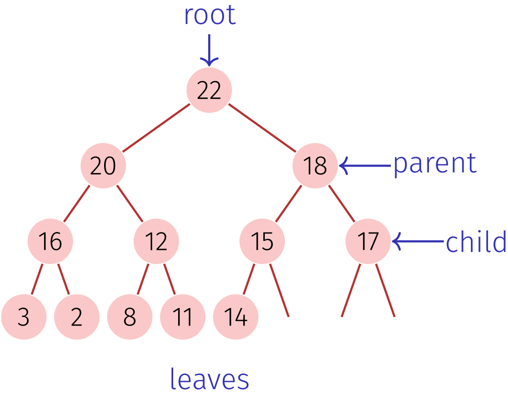
17. Heaps

Data structure optimized for fast extraction of minimum or maximum and for sorting. [Ottman/Widmayer, Kap. 2.3, Cormen et al, Kap. 6]

[Max-]Heap*

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) than that of the parent node

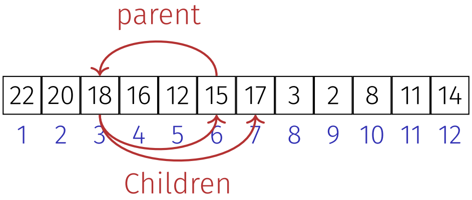


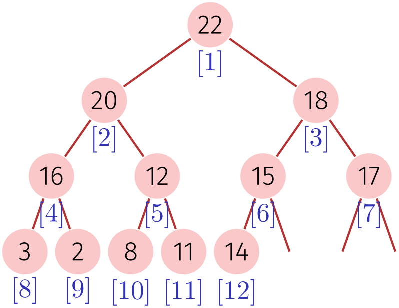
*Heap(data structure), not as in “heap and stack” (memory allocation)

Heap as Array

Tree → Array:

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





Depends on the starting index[19]

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

Height of a Heap

What is the height $H(n)$ of Heap with n nodes? On the i-th level of a binary tree there are at most 2^i nodes. Modulo the last level of a heap, all levels are filled with values.

$$H(n) = \min\{h \in \mathbb{N} : \sum_{i=0}^{h-1} 2^i \geq n\}$$

with $\sum_{i=0}^{h-1} 2^i = 2^h - 1$:

$$H(n) = \min\{h \in \mathbb{N} : 2^h \geq n + 1\},$$

thus

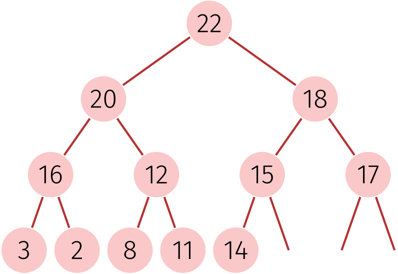
$$H(n) = \lceil \log_2(n+1) \rceil.$$

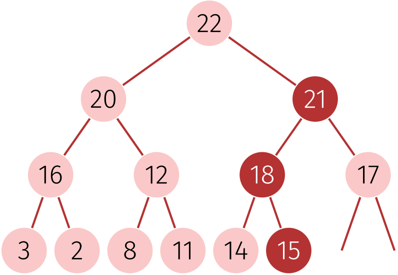
Heap in C++

```
class MaxHeap {
  int* keys; // Pointer to first key
  unsigned int capacity; // Length of key array
  unsigned int count; // Keys in use <= capacity
  // Or even better: build on top of std::vector

public:
  MaxHeap(unsigned int initial_capacity):
      keys(new int[initial_capacity]),
      capacity(initial_capacity),
      count(0)
  {}

  void insert(unsigned int key) { ...}
  int remove_max() { ...}
  ...
}
```

Insert

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





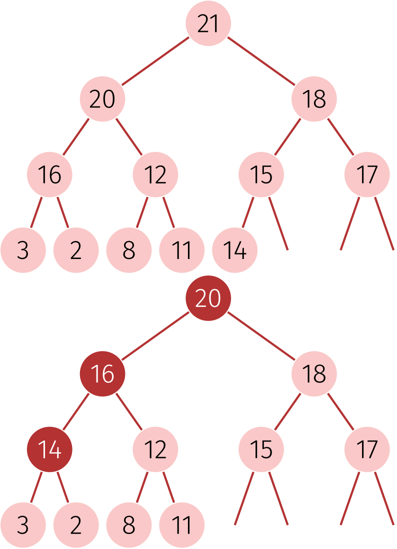
Algorithm Sift-Up(A, m)

Input: Array A with at least m keys and heap structure on $A[1, \dots, m-1]$
Output: Array A with heap structure on $A[1, \dots, m]$

```
$v \leftarrow A[m]$ // new key
$c \leftarrow m$ // index current node (child)
$p \leftarrow \lfloor c/2 \rfloor$ // index parent node
while $c > 1$ and $v > A[p]$ do
    $A[c] \leftarrow A[p]$ // key parent node → key current node
    $c \leftarrow p$ // parent node → current node
    $p \leftarrow \lfloor c/2 \rfloor$
$A[c] \leftarrow v$ // place new key
```

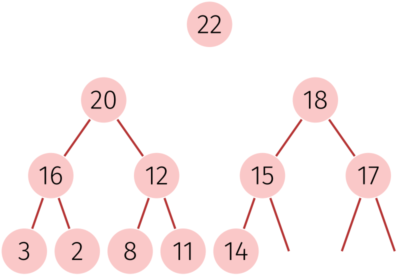
Remove the Maximum

- 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)

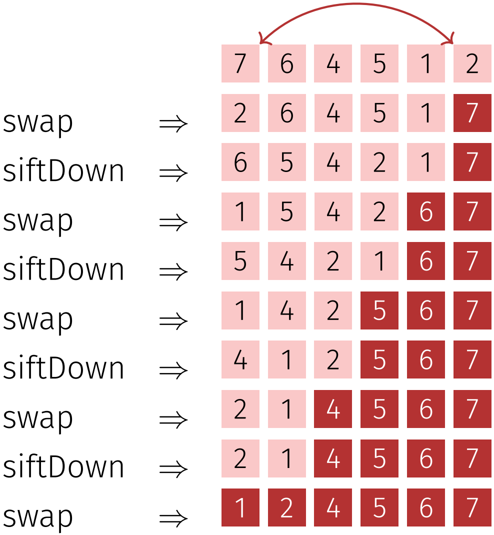
```
Input:  Array $A$ with heap structure for the children of $i$. Last element $m$.
Output: Array $A$ with heap structure for $i$ with last element $m$.
while $2i \leq m$ do
    $j \leftarrow 2i$; // $j$ left child
    if $j < m$ and $A[j] < A[j+1]$ then
        $j \leftarrow j+1$; // $j$ right child 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
```

Sorting Heaps

Let $A[1, ..., n]$ be a heap.

While $n > 1$:

1. *Swap*$(A[1], A[n])$
2. *SiftDown*$(A, 1, n-1)$
3. $n \leftarrow n - 1$



		7	6	4	5	1	2
swap	⇒	2	6	4	5	1	7
siftDown	⇒	6	5	4	2	1	7
swap	⇒	1	5	4	2	6	7
siftDown	⇒	5	4	2	1	6	7
swap	⇒	1	4	2	5	6	7
siftDown	⇒	4	1	2	5	6	7
swap	⇒	2	1	4	5	6	7
siftDown	⇒	2	1	4	5	6	7
swap	⇒	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 $2 \log n$ comparisons in the worst case.
Number of memory movements while 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:

We use that $h(n) = \lceil \log_2 n + 1 \rceil = \lfloor \log_2 n \rfloor + 1$ für $n > 0$

$$v(n) = \sum_{l=0}^{\lfloor \log_2 n \rfloor} \underbrace{2^l}_{\text{number heaps on level l}} \cdot (\underbrace{\lfloor \log_2 n \rfloor + 1 - l}_{\text{height heaps on level l}} -1) = \sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^{\lfloor \log_2 n \rfloor - k} \cdot k$$

$$= 2^{\lfloor \log_2 n \rfloor} \cdot \sum_{k=0}^{\lfloor \log_2 n \rfloor} \frac{k}{2^k} \leq n \cdot \sum_{k=0}^{\infty} \frac{k}{2^k} \leq n \cdot 2 \in \mathcal{O}(n)$$

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

Disadvantages

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.

18. AVL Trees

Balanced Trees [Ottman/Widmayer, Kap. 5.2-5.2.1, Cormen et al, Kap. Problem 13-3]

Background

- Search tree: Search, insertion and removal of a key in average in $\mathcal{O}(\log n)$ steps (given n keys in the tree)
- Worst case, though: $\Theta(n)$ (degenerated tree)

Goal: Avoid degeneration, by balancing the tree after each update operation.

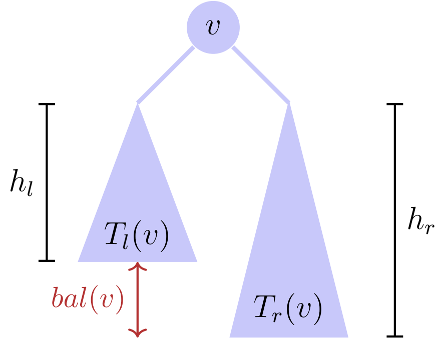
Balancing: guarantee that a tree with n nodes always has a height of $\mathcal{O}(\log n)$.

Adelson-Velsky and Landis (1962): AVL-Trees

Balance of a node

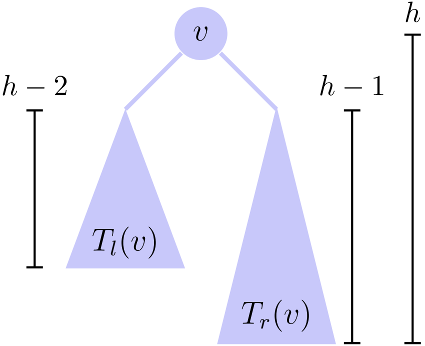
The *balance* of a node v is defined as the height difference of its sub-trees $T_l(v)$ and $T_r(v)$

$$\mathrm{bal}(v) := h(T_r(v)) - h(T_l(v))$$

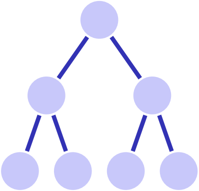


AVL Condition

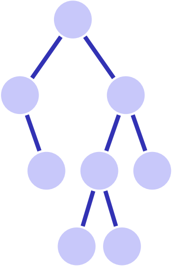
AVL Condition: for each node v of a tree $\mathrm{bal}(v) \in \{-1, 0, 1\}$



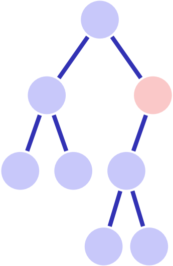
(Counter-)Examples



AVL tree with height 3



AVL tree with height 4

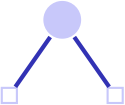


No AVL tree

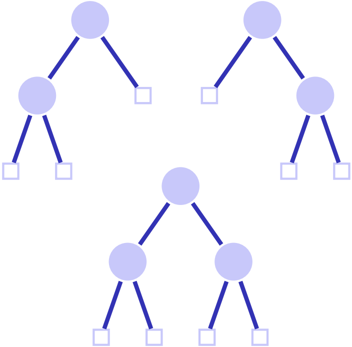
Number of Leaves

- 1. observation: a binary tree with n keys provides exactly $n+1$ leaves. Simple induction argument.
  - The binary 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 binary tree with given height implies an upper bound of the height of a binary tree with given number of keys.

Lower bound of the leaves



AVL tree with height 1 has $N(1) := 2$ leaves.



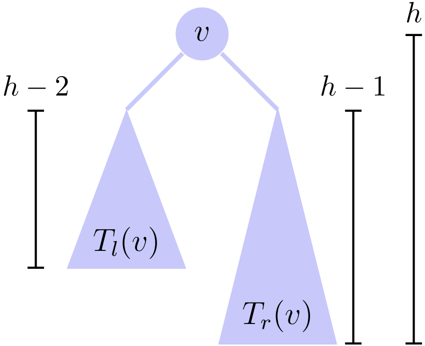
AVL tree with height 2 has at least $N(2) := 3$ leaves.

Lower bound on the leaves for $h > 2$ in AVL trees

- 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[20]

$$F_i = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)$$

with the roots $\phi, \hat{\phi}$ of the golden ratio equation $x^2 - x - 1 = 0$:

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$$
$$\hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -0.618$$

[20] Derivation using generating functions (power series) in the appendix.

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

$$N(h) \geq c \cdot 1.618^h \quad \Rightarrow \quad h \leq 1.44 \log_2 n + c'.$$

- I.e. an AVL tree has, as desired, a height of $\mathcal{O}(\log n)$
- and is asymptotically not more than 44% higher than a perfectly balanced tree (height $\lceil \log_2 n + 1 \rceil$)

Insertion and Balancing

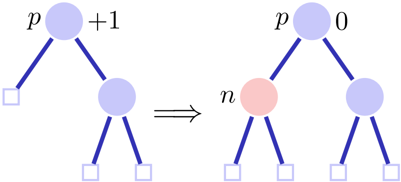
Balance:

- Insertion potentially violates AVL condition → balancing
- For that, we store the balance in each node

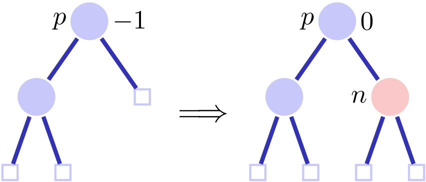
Insert:

- Insert new node n, as done for search trees
- Check, and potentially restore, balance of all nodes from n upwards to the root

Balance at Insertion Point



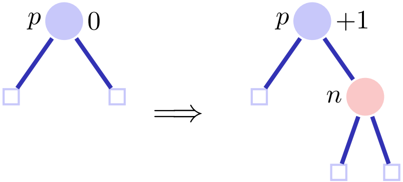
case 1: $\mathrm{bal}(p) = +1$



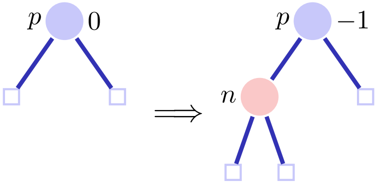
case 2: $\mathrm{bal}(p) = -1$

Directly done in both cases because the height of subtree p did not change. Balance of parent node thus also unchanged.

Balance at Insertion Point



case 3.1: $\mathrm{bal}(p) = 0$ right



case 3.2: $\mathrm{bal}(p) = 0$, left

Not yet done in both case, since parent node potentially no longer balanced → Invocation of function **`upin(p)`** (upwards + insert)

upin(p): Recursive Invocation Requirement

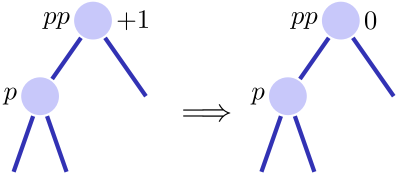
For every call **`upin(p)`** it must hold that

- the subtree p grew and thereby
- changed $\mathrm{bal}(p)$ from 0 to $\in \{-1, +1\}$.

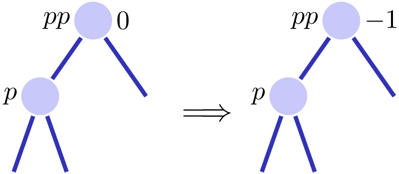
Because only in this situation can the newly developed imbalance of p ($\mathrm{bal}(p) \neq 0$) affect the tree structure above.

upin(p)

Assumption: p is left son of pp[21]



case 1: $\mathrm{bal}(pp) = +1$, done.



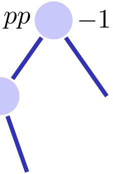
case 2: $\mathrm{bal}(pp) = 0$, **`upin(pp)`**

In both cases the AVL-Condition holds for the subtree from pp

[21] If p is a right son: symmetric cases with exchange of $+1$ and -1

upin(p)

Assumption: p is left son of pp



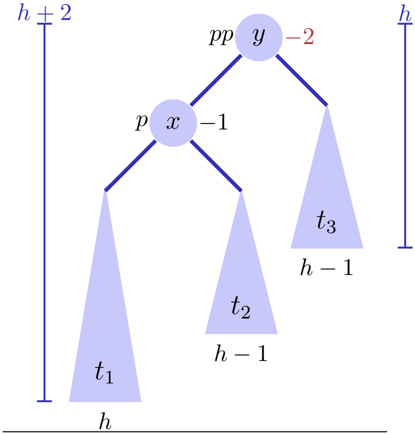
case 3: $\mathrm{bal}(pp) = -1$,

This case is problematic: adding n to the subtree from pp has violated the AVL-condition. Re-balance!

Two cases $\mathrm{bal}(p) = -1$, $\mathrm{bal}(p) = +1$

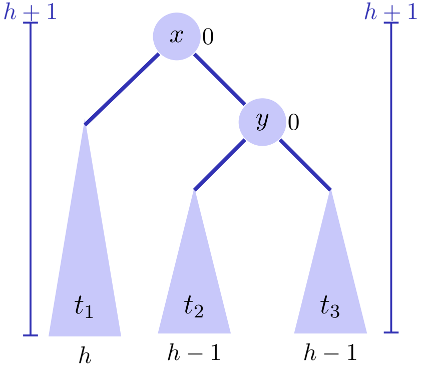
Rotations

case 1.1 $\mathrm{bal}(p) = -1.$[22]



$\Longrightarrow$

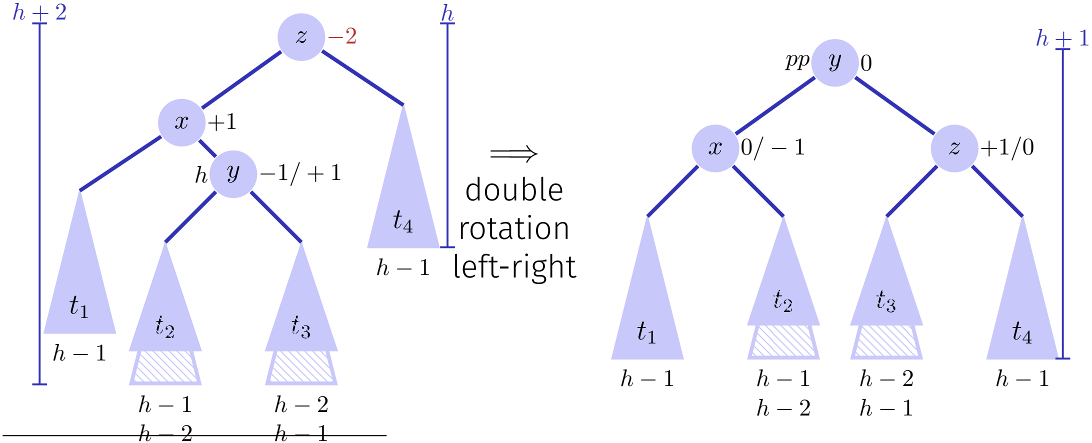
rotation right



[22] p right son: $\Rightarrow \mathrm{bal}(pp) = \mathrm{bal}(p) = +1$, left rotation

Rotations

case 1.2 $\mathrm{bal}(p) = +1$. [23]



[23] p right son $\Rightarrow \mathrm{bal}(pp) = +1$, $\mathrm{bal}(p) = -1$, double rotation right left

Analysis

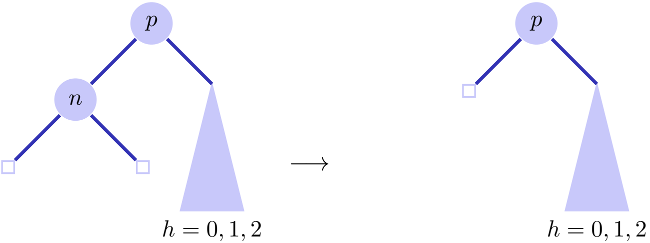
- Tree height: $\mathcal{O}(\log n)$.
- Insertion like in binary search tree.
- Balancing via recursion from node to the root (during recursive ascend). Maximal path lenght $\mathcal{O}(\log n)$.

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.

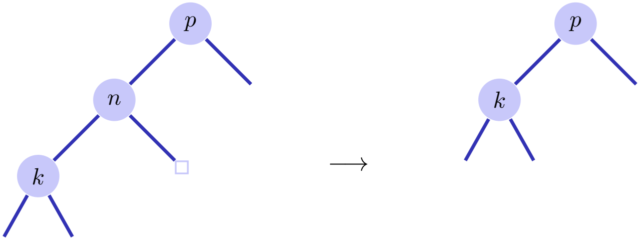
- $h' = 1$: Adapt $\text{bal}(p)$.
- $h' = 0$: Adapt $\text{bal}(p)$. Call **`upout(p)`**.
- $h' = 2$: Rebalanciere des Teilbaumes. Call **`upout(p)`**.



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 $\Rightarrow$ **`upout(k)`**
- Deletion of the symmetric successor is as in case 1 or 2.

`upout(p)`

Let pp be the parent node of p.

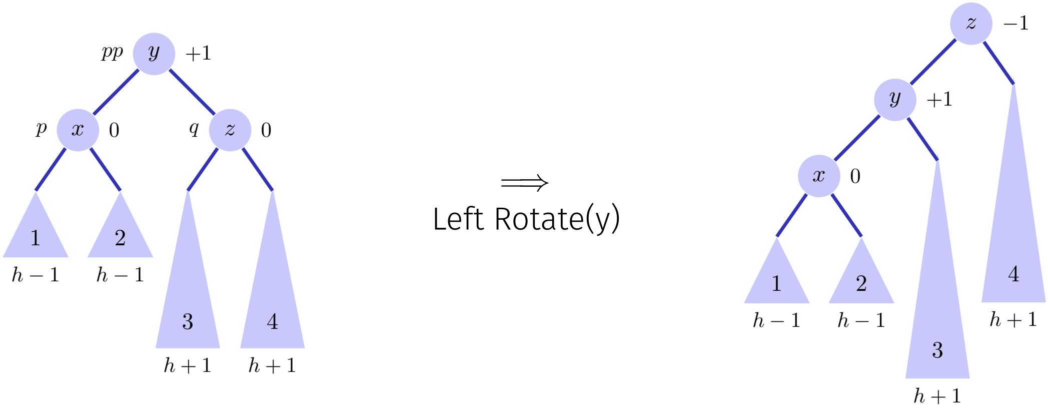
(a) p left child of pp

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: $\mathrm{bal}(q) = 0$.[24]

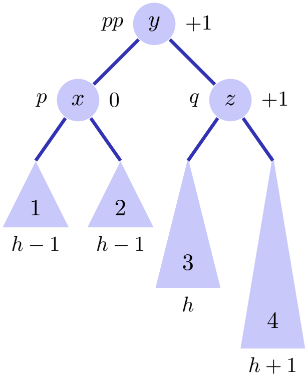


Left Rotate(y)

[24] (b).3.1: $\mathrm{bal}(pp) = -1$, $\mathrm{bal}(q) = 0$, Right rotation

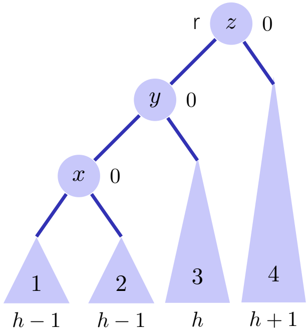
`upout(p)`

Case (a).3: $\mathrm{bal}(pp) = +1$. (a).3.2: $\mathrm{bal}(q) = +1$.[25]



$\Longrightarrow$

Left Rotate(y)

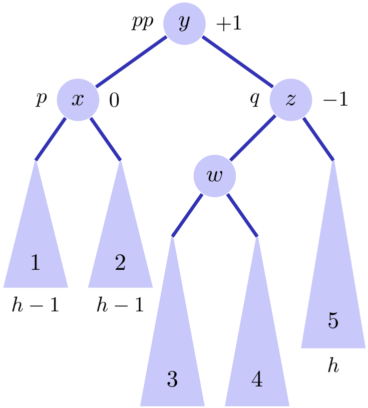


plus **`upout(r)`**.

[25] (b).3.2: $\mathrm{bal}(pp) = -1$, $\mathrm{bal}(q) = +1$, Right rotation+upout

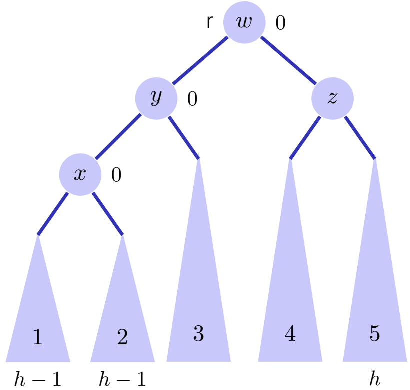
upout(p)

Case (a).3: $\mathrm{bal}(pp) = +1$. (a).3.3: $\mathrm{bal}(q) = -1$.[26]



$\Longrightarrow$

Rotate right (z)
left (y)



plus **upout(r)**.

[26] (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. For small trees (key sets), the costs of balancing outweighs the gain of $\mathcal{O}(\log n)$ height.
- Several other balanced trees exist: Red-Black tree (**`std::map`** in $\mathrm{C}{+}{+}$), B-tree (**`std::collections::BTreeMap`** in Rust), Splay tree; Treap (balanced with high probability)

18.6 Appendix

Derivation of some mathemmatical formulas

Fibonacci Numbers, Inductive Proof

$F_i \overset{!}{=} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i) \qquad [*]$ $\left(\phi = \frac{1+\sqrt{5}}{2}, \hat{\phi} = \frac{1-\sqrt{5}}{2}\right)$.

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 $x + 1 = x^2$)

$$= \frac{1}{\sqrt{5}}\phi^{i-2}(\phi^2) - \frac{1}{\sqrt{5}}\hat{\phi}^{i-2}(\hat{\phi}^2) = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i).$$

[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:

$$f(x) \cdot (1 - x - x^2) = x.$$

$$\Leftrightarrow \quad f(x) = \frac{x}{1 - x - x^2} = -\frac{x}{x^2 + x - 1}$$

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

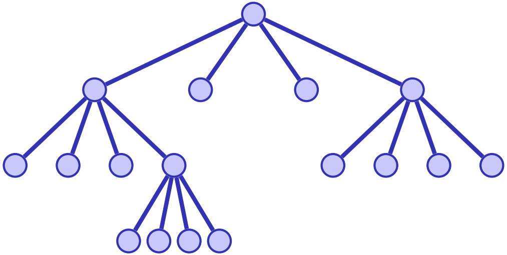
$$F_i = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i).$$

19. Quadtrees

Quadtrees, Collision Detection, Image Segmentation

Quadtree

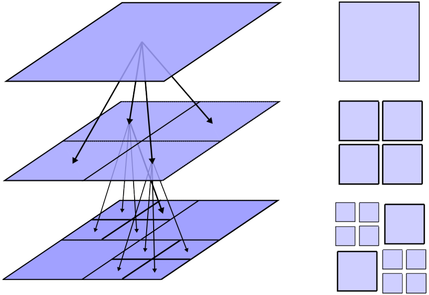
A quad tree is a tree of order 4.



… and as such it is not particularly interesting except when it is used for …

Quadtree - Interpretation und Nutzen

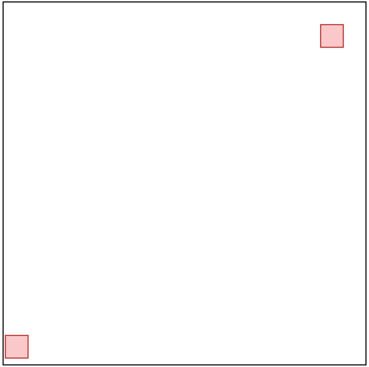
Separation of a two-dimensional range into 4 equally sized parts.



[analogously in three dimensions with an *octtree* (tree of order 8)]

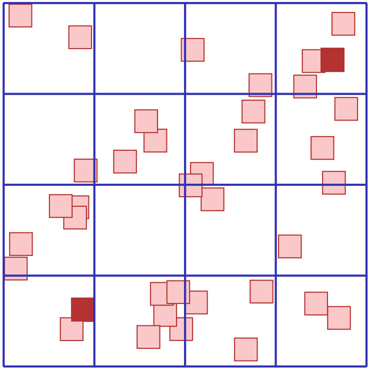
Example 1: Collision Detection

- Objects in the 2D-plane, e.g. particle simulation on the screen.
- Goal: collision detection



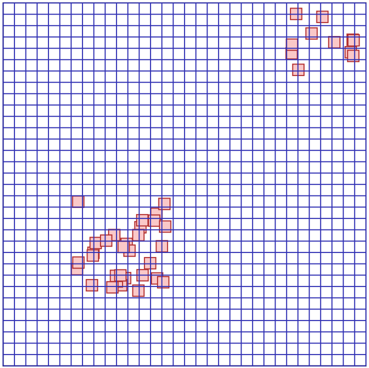
Idea

- 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



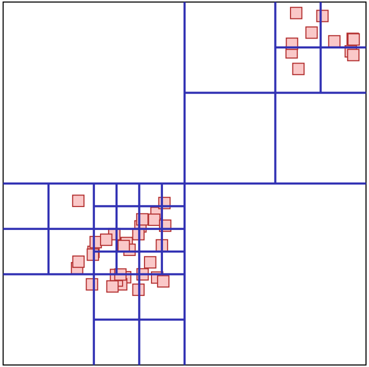
Grids

- A grid often helps, but not always
- Improvement?
- More finegrained grid?
- Too many grid cells!



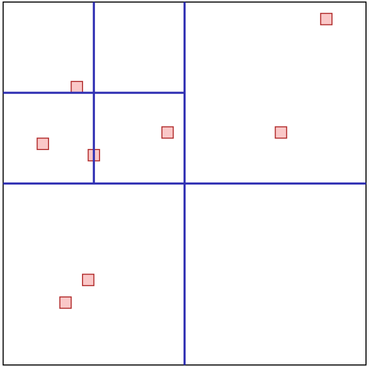
Adaptive Grids

- A grid often helps, but not always
- Improvement?
- *Adaptively* refine grid
- Quadtree!



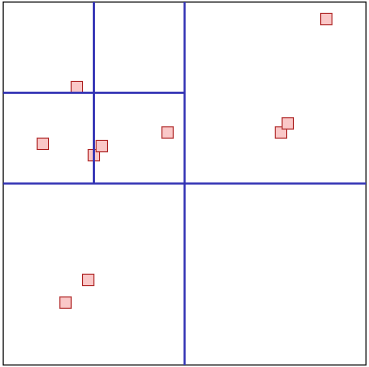
Algorithm: Insertion

- 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.

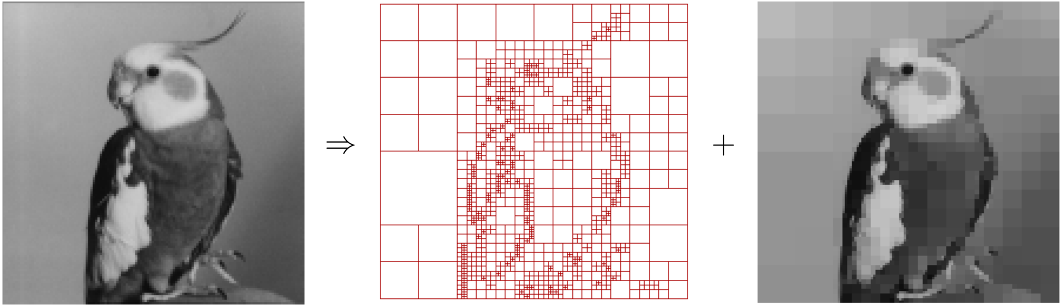


Algorithm: Collision Detection

- Run through the quadtree in a recursive way. For each node test collision with all objects contained in the same or (recursively) contained nodes.

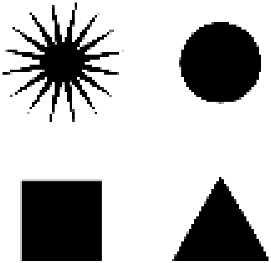


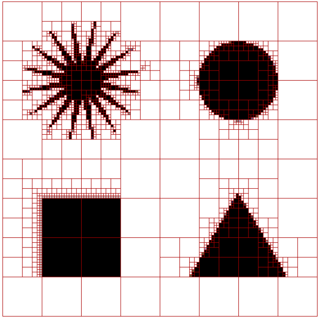
Example 2: Image Segmentation



(Possible applications: compression, denoising, edge detection)

Quadtree on Monochrome Bitmap

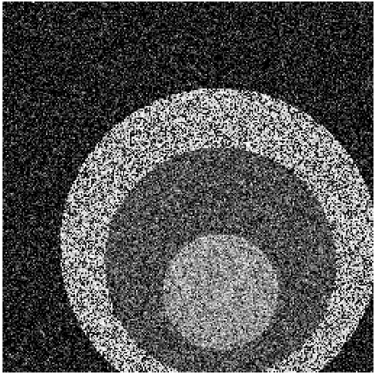


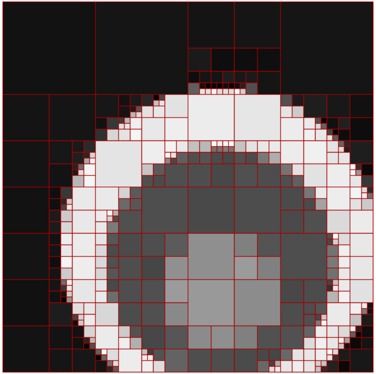


Similar procedure to generate the quadtree: split nodes recursively until each node only contains pixels of the same color.

Quadtree with Approximation

When there are more than two color values, the quadtree can get very large. ⇒ Compressed representation: *approximate* the image piecewise constant on the rectangles of a quadtree.





Piecewise Constant Approximation

(Grey-value) Image $\boldsymbol{y} \in \mathbb{R}^S$ on pixel indices S. [27]

Rectangle $r \subset S$.

Goal: determine

$$\arg\min_{v \in \mathbb{R}} \sum_{s \in r} (y_s - v)^2$$

Solution: the arithmetic mean $\mu_r = \frac{1}{|r|} \sum_{s \in r} y_s$

[27] 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

$$\mu_r = \frac{1}{|r|} \sum_{s \in r} y_s$$

and the corresponding error

$$\sum_{s \in r} (y_s - \mu_r)^2 =: \|\boldsymbol{y}_r - \boldsymbol{\mu}_r\|_2^2$$

can be computed quickly after a $\mathcal{O}(|S|)$ tabulation: prefix sums!

Which Quadtree?

Conflict

- **As close as possible to the data** ⇒ small rectangles, large quadtree . Extreme case: one node per pixel. Approximation = original
- **Small amount of nodes** ⇒ 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[28] $L(T)$ such that it minimizes the following function

$$H_\gamma(T, \boldsymbol{y}) := \gamma \cdot \underbrace{|L(T)|}_{\text{Number of Leaves}} + \underbrace{\sum_{r \in L(T)} \|y_r - \mu_r\|_2^2}_{\text{Cummulative approximation error of all leaves}} .$$

[28] 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

$$\widehat{H}_\gamma(T, y) := \min_T \gamma \cdot |L(T)| + \sum_{r \in L(T)} \|y_r - \mu_r\|_2^2$$

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

$$\widehat{H}_\gamma(T, \boldsymbol{y}) = \gamma$$

- Let, otherwise,

$$M_1 := \gamma + \|\boldsymbol{y}_{S_T} - \boldsymbol{\mu}_{S_T}\|_2^2$$
$$M_2 := \widehat{H}_\gamma(T_{ll}, \boldsymbol{y}) + \widehat{H}_\gamma(T_{lr}, \boldsymbol{y}) + \widehat{H}_\gamma(T_{ul}, \boldsymbol{y}) + \widehat{H}_\gamma(T_{ur}, \boldsymbol{y})$$

then

$$\widehat{H}_\gamma(T, y) = \min\{\underbrace{M_1(T, \gamma, \boldsymbol{y})}_{\text{no split}}, \underbrace{M_2(T, \gamma, \boldsymbol{y})}_{\text{split}}\}$$

Algorithmus: Minimize($\boldsymbol{y}$,r,γ)

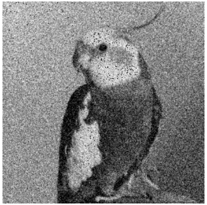
Input: Image data $\boldsymbol{y} \in \mathbb{R}^S$, rectangle $r \subset S$, regularization $\gamma > 0$
Output: $\min_T \gamma |L(T)| + \|\boldsymbol{y} - \boldsymbol{\mu}_{L(T)}\|_2^2$

```
if $|r| = 0$ then return $0$
$m \leftarrow \gamma + \sum_{s \in r} (y_s - \mu_r)^2$
if $|r| > 1$ then
    Split $r$ into $r_{ll}$,$r_{lr}$,$r_{ul}$,$r_{ur}$
    $m_1 \leftarrow \mathrm{Minimize}(\boldsymbol{y}, r_{ll}, \gamma)$; $m_2 \leftarrow \mathrm{Minimize}(\boldsymbol{y}, r_{lr}, \gamma)$
    $m_3 \leftarrow \mathrm{Minimize}(\boldsymbol{y}, r_{ul}, \gamma)$; $m_4 \leftarrow \mathrm{Minimize}(\boldsymbol{y}, r_{ur}, \gamma)$
    $m' \leftarrow m_1 + m_2 + m_3 + m_4$
else
    $m' \leftarrow \infty$
if $m' < m$ then $m \leftarrow m'$
return $m$
```

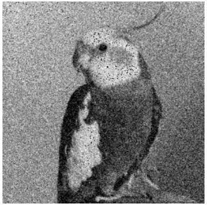
Analysis

The minimization algorithm over dyadic partitions (quadtrees) takes $\mathcal{O}(|S| \log |S|)$ steps.

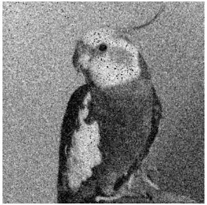
Application: Denoising (with addditional Wedgelets)



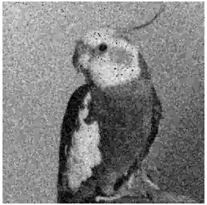
noised



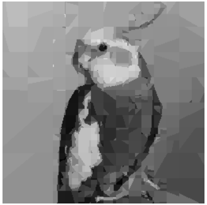
$\gamma = 0.003$



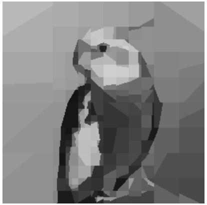
$\gamma = 0.01$



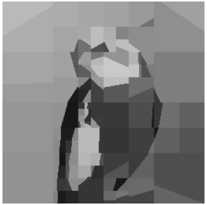
$\gamma = 0.03$



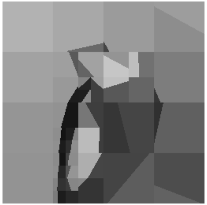
$\gamma = 0.1$



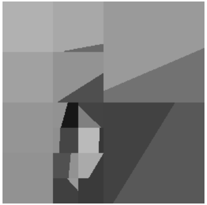
$\gamma = 0.3$



$\gamma = 1$

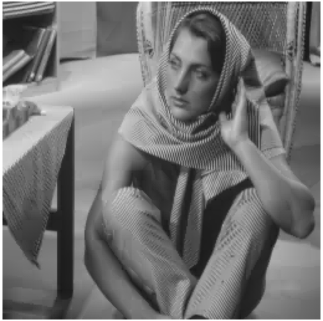


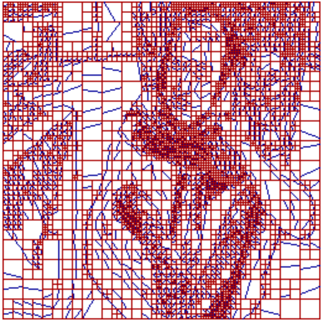
$\gamma = 3$



$\gamma = 10$

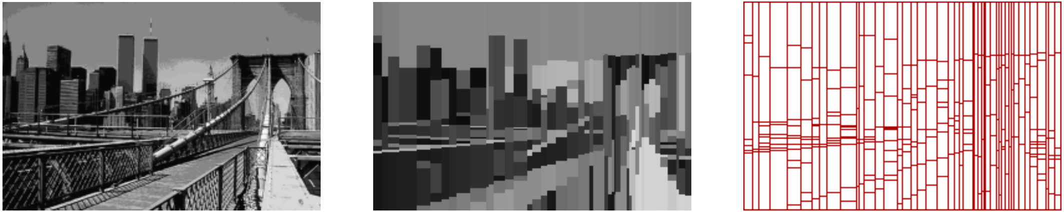
Extensions: Affine Regression + Wedgelets





Other ideas

no quadtree: hierarchical one-dimensional modell (requires dynamic programming)



19.1 Appendix

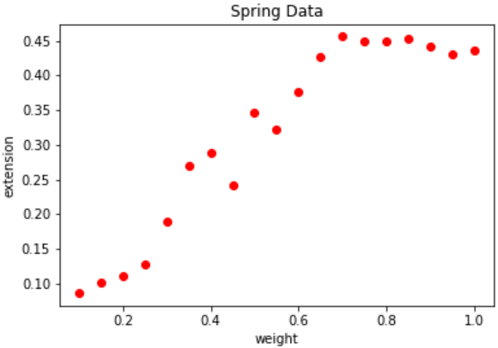
Linear Regression

The Learning Problem

Setup

- We observe N data points
- Input examples: $\boldsymbol{X} = (\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N)^\top$
- Output examples: $\boldsymbol{y} = (y_1, \ldots, y_N)^\top$
- Assupmtion: there is an underlying truth

$$f : \mathcal{X} \to \mathcal{Y}$$



Goal: find a good approximation $h \approx g$ to make predictions $h(\boldsymbol{x})$ for new data points or to explain the data in order to find a compressed representation, for instance.

Here $\mathcal{X} = \mathbb{R}^d$. $\mathcal{Y} = \mathbb{R}$ (Regression).

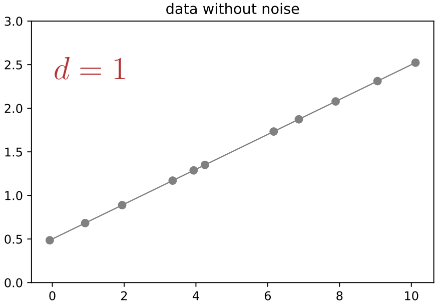
Model: Linear Regression

Assumption: The underlying truth can be represented as

$$h_{\boldsymbol{w}}(\boldsymbol{x}) = w_0 + w_1 x_1 + \cdots + w_d x_d = w_0 + \sum_{i=1}^{d} w_i x_i .$$

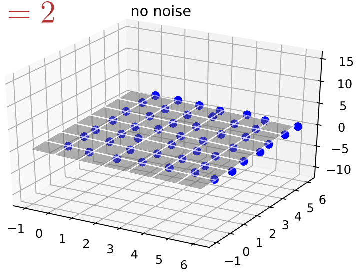
linear in $\boldsymbol{w}$!

$\Rightarrow$ We search for $\boldsymbol{w}$ (sometimes also d).



$d = 1$

$d = 2$



Trick for simplified notation

$$\boldsymbol{x} = (x_1, \ldots, x_d) \rightarrow (\underbrace{x_0}_{\equiv 1}, x_1, \ldots, x_d)$$

$$\begin{aligned} h_{\boldsymbol{w}}(\boldsymbol{x}) &= w_0 x_0 + w_1 x_1 + \cdots + w_d x_d \\ &= \sum_{i=0}^{d} w_i x_i \\ &= \boldsymbol{w}^\top \boldsymbol{x} \end{aligned}$$

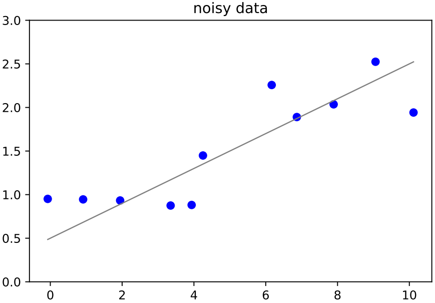
Data matrix

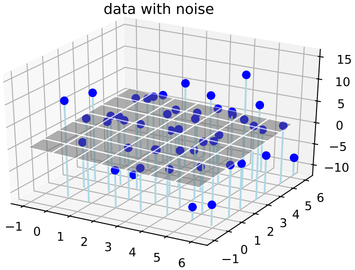
$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{X}_1 \\ \boldsymbol{X}_2 \\ \vdots \\ \boldsymbol{X}_n \end{bmatrix} = \begin{bmatrix} \overset{\overset{\equiv 1}{\downarrow}}{X_{1,0}} & X_{1,1} & X_{1,2} & \dots & X_{1,d} \\ X_{2,0} & X_{2,1} & X_{2,2} & \dots & X_{2,d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ X_{n,0} & X_{n,1} & X_{n,2} & \dots & X_{n,d} \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \qquad \boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_d \end{bmatrix}$$

$$\boldsymbol{X}\boldsymbol{w} \approx \boldsymbol{y}?$$

Imprecise observations

Reality: the data are imprecise or the model is only a model.





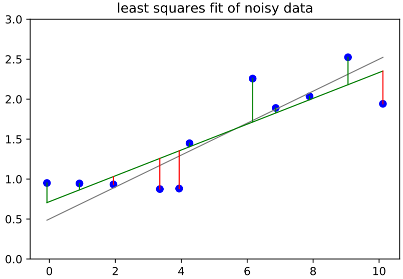
What to do?

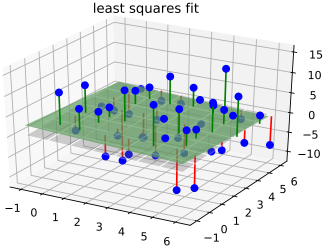
Error function

$$E(\boldsymbol{w}) = \sum_{i=1}^{N} (h_{\boldsymbol{w}}(\boldsymbol{X}_i) - y_i)^2$$

Want a $\widehat{\boldsymbol{w}}$ that minimizes E

Linarity of $h_{\boldsymbol{w}}$ in $\boldsymbol{w}$ ⇒ solution with linear algebra.





Solution from Linear Algebra

$$\widehat{\boldsymbol{w}} = \underbrace{\left(\boldsymbol{X}^\top \boldsymbol{X}\right)^{-1} \boldsymbol{X}^\top}_{=:\boldsymbol{X}^\dagger} \boldsymbol{y}.$$

$\boldsymbol{X}^\dagger$: *Moore-Penroe Pseudo-Inverse*

Fitting Polynomials

Also works with linear regression.

$$h_{\boldsymbol{w}}(x) = w_0 + w_1 x^1 + w_2 x^2 + \cdots + w_d x^d = w_0 + \sum_{i=1}^{d} w_i x^i.$$

because $h_{\boldsymbol{w}}(x)$ remains being linear in $\boldsymbol{w}$!

$$\boldsymbol{X} = \begin{bmatrix} 1 & x_1 & (x_1)^2 & \dots & (x_1)^d \\ 1 & x_2 & (x_2)^2 & \dots & (x_2)^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & (x_n)^2 & \dots & (x_n)^d \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \qquad \boldsymbol{w} = \begin{bmatrix} w_0 \\ \vdots \\ w_d \end{bmatrix}$$

Example: Constant Approximation

$$\boldsymbol{X} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \qquad \boldsymbol{w} = \begin{bmatrix} w_0 \end{bmatrix}$$

$$\widehat{\boldsymbol{w}} = \left(\boldsymbol{X}^\top \boldsymbol{X}\right)^{-1} \boldsymbol{X}^\top \boldsymbol{y} = \left[\frac{1}{n} \sum y_i\right].$$

Example: Linear Approximation

$$\boldsymbol{X} = \begin{bmatrix} 1 & x_1^{(1)} & x_1^{(2)} \\ 1 & x_2^{(1)} & x_2^{(2)} \\ \vdots & & \\ 1 & x_n^{(1)} & x_n^{(2)} \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \qquad \boldsymbol{w} = \begin{bmatrix} w_0 \end{bmatrix}$$

$$\widehat{\boldsymbol{w}} = \left(\boldsymbol{X}^\top \boldsymbol{X}\right)^{-1} \boldsymbol{X}^\top \boldsymbol{y} = \begin{bmatrix} N & \sum x_i^{(1)} & \sum x_i^{(2)} \\ \sum x_i^{(1)} & \sum \left(x_i^{(1)}\right)^2 & \sum x_i^{(1)} \cdot x_i^{(2)} \\ \sum x_i^{(2)} & \sum x_i^{(1)} \cdot x_i^{(2)} & \sum \left(x_i^{(2)}\right)^2 \end{bmatrix}^{-1} \cdot \begin{bmatrix} \sum y_i \\ \sum y_i \cdot x_i^{(1)} \\ \sum y_i \cdot x_i^{(2)} \end{bmatrix}$$

20. 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

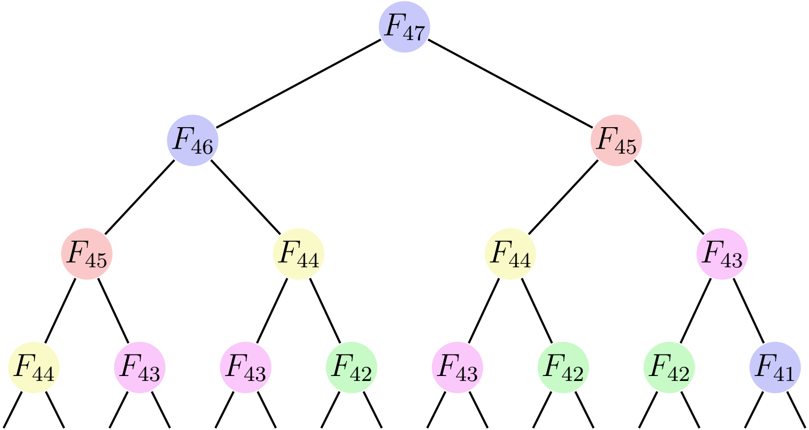
$T(n)$: Number executed operations.

- $n = 0, 1$: $T(n) = \Theta(1)$
- $n \geq 2$: $T(n) = T(n-2) + T(n-1) + c$.

$$T(n) = T(n-2) + T(n-1) + c \geq 2T(n-2) + c \geq 2^{n/2}c' = (\sqrt{2})^n c'$$

Algorithm is **exponential** in n.

Reason (visual)



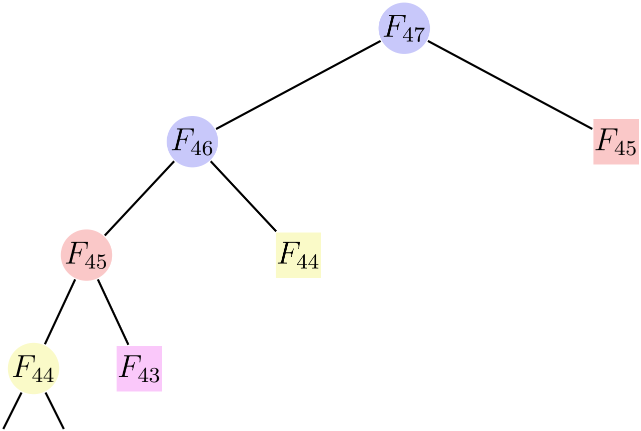
Nodes with same values are evaluated (too) often.

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



Rectangular nodes have been computed before.

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.[29]

[29] 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,$\ldots$ 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

Arbeitsspeicher ✓

192.–

HyperX Fury (2x, 8GB, DDR4-2400, DIMM 288)

★★★★⯪ 16

We trade speed against memory consumption

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**.
- Classical 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**.

Dynamic Programming: Description

1. Use a **DP-table** with information to the subproblems.
   Dimension of the table? 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 (Fibonacci)

1. Dimension of the table? Semantics of the entries?

   $n \times 1$ table. nth entry contains nth Fibonacci number.

2. Which entries do not depend on other entries?

   Values F_1 and F_2 can be computed easily and independently.

3. Computation order?

   F_i with increasing i.

4. Reconstruction of a solution?

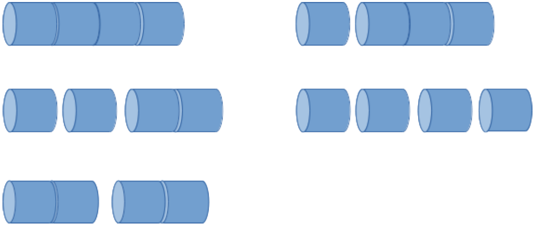
   F_n is the n-th Fibonacci number.

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.

How to Find the DP Algorithm.

0. Exact formulation of the wanted solution
1. Define sub-problems, reformulate (0.) as sub-problem
2. Recursion: relate subproblems by enumerating of local properties
3. Determine the dependencies of the sub-problems
4. Solve the problem
   Running time = #sub-problems × 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 $0 \leq k < n$
2. Local property: length of the first piece
   Recursion

$$
\begin{aligned}
r_k &= \max\{v_i + r_{k-i} : 0 < i \leq k\}, \quad k > 0 \\
r_0 &= 0
\end{aligned}
$$

3. **Dependency:** r_k depends (only) on values v_i, $1 \leq i \leq k$ and the optimal cuts r_i, $i < k$.
4. **Solution** in r_n. DP running time: $\Theta(n^2)$

Algorithm RodCut(v,n) (without memoization)

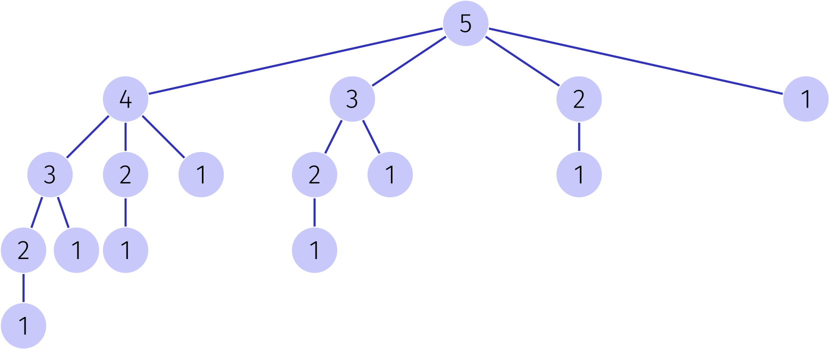
Input: $n \geq 0$, Prices v
Output: best value

```
$q \leftarrow 0$
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 $T(n) = \sum_{i=0}^{n-1} T(i) + c \quad \Rightarrow$[30] $\quad T(n) \in \Theta(2^n)$

[30] $T(n) = T(n-1) + \sum_{i=0}^{n-2} T(i) + c = T(n-1) + (T(n-1) - c) + c = 2T(n-1) \quad (n > 0)$

Recursion Tree



Algorithm RodCutMemoized(m, v, n)

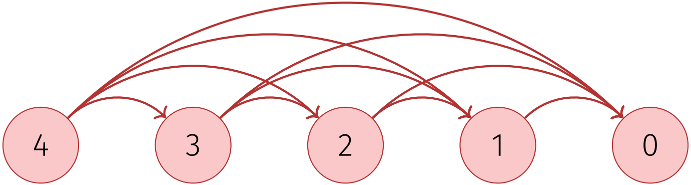
Input: $n \geq 0$, Prices v, Memoization Table m
Output: best value

```
$q \leftarrow 0$
if $n > 0$ then
    if $\exists\ m[n]$ then
        $q \leftarrow m[n]$
    else
        for $i \leftarrow 1, \ldots, n$ do
            $q \leftarrow \max\{q, v_i +$ 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



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. nth 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. Computation order?

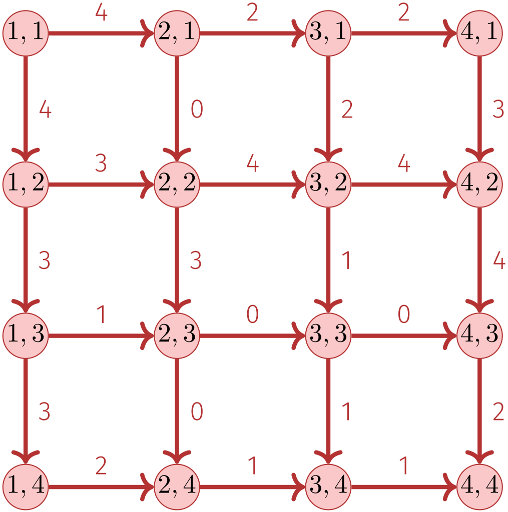
   r_i, $i = 1, \ldots, n$.

4. Reconstruction of a solution?

   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?

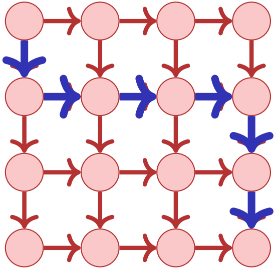


Rabbit!

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



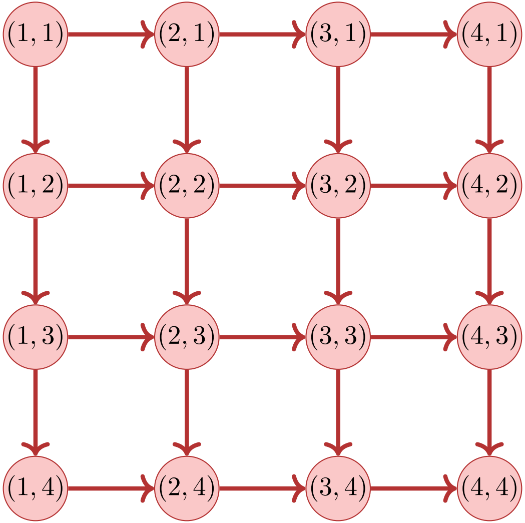
The path 100011
(1:to south, 0: to east)

Recursion

Wanted: $T_{1,1}$ **= maximal number carrots from** $(1,1)$ **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



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

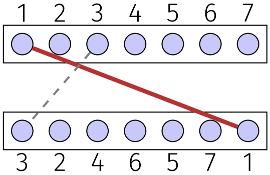
3. Computation order?

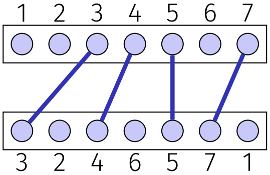
   $T_{i,j}$ with $i = n \searrow 1$ and for each i: $j = n \searrow 1$, (or vice-versa: $j = n \searrow 1$ and for each j: $i = n \searrow 1$).

4. Reconstruction of a solution?

   $T_{1,1}$ provides the maximal number of carrots.

Longest Ascending Sequence (LAS)

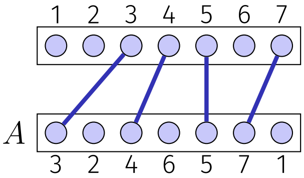




Connect as many as possible fitting ports without lines crossing.

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 (Greedy)

Let L_i **= longest ascending subsequence of** A_i $(1 \leq i \leq n)$

Assumption: LAS L_k of A_k known. Compute L_{k+1} for A_{k+1}.

Idea

$$L_{k+1} = \begin{cases} L_k \oplus a_{k+1} & \text{if } a_k > \max(L_k) \\ L_k & \text{otherwise?} \end{cases}$$

Counterexample

$A_5 = (1, 2, 5, 3, 4)$.
$A_3 = (1, 2, 5)$ with $L_3 = A_3$ and $L_4 = A_3$.

Greedy idea fails here: we cannot directly infer L_{k+1} from L_k.

Second idea. (Prefix)

Let L_i **= longest ascending subsequence of** A_i $(1 \leq i \leq n)$

Assumption: a LAS L_j that ends in a_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.

Example

$A_5 = (1, 2, 5, 3, 4)$.
$L_1 = (1)$, $L_2 = (1, 2)$, $L_3 = (1, 2, 5)$, $L_4 = (1, 2, 3)$, $L_5 = (1, 2, 3, 4)$.

This works with running time n^2 (and requires access to all sequences L_i.

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	$(\mathbf{1})$
$+\ 1000$	$(1), (1, \mathbf{1000})$
$+\ 1001$	$(1), (1, 1000), (1, 1000, \mathbf{1001})$
$+\ 4$	$(1), (1, \mathbf{4}), (1, 1000, 1001)$
$+\ 5$	$(1), (1, 4), (1, 4, \mathbf{5})$
$+\ 2$	$(1), (1, \mathbf{2}), (1, 4, 5)$
$+\ 6$	$(1), (1, 2), (1, 4, 5), (1, 4, 5, \mathbf{6})$
$+\ 7$	$(1), (1, 2), (1, 4, 5), (1, 4, 5, 6), (1, 4, 5, 6, \mathbf{7})$

DP Table

- Idea: save the last element of the increasing sequence $M_{k,j}$ at slot j.
- Example:
  13 12 15 11 16 14
- Problem: Table does not contain the subsequence, only the last value.
- Solution: second table with the values of the predecessors.

i	1	2	3	4	5	6
value a_i	13	12	15	11	16	14
Predecessor	$-\infty$	$-\infty$	12	$-\infty$	15	11

j	0	1	2	3	4	...
$(L_j)_j$	-∞	11	14	16	∞	

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 binary search for l, such that $T[l] < a_k < T[l+1]$. Set $T[l+1] \leftarrow a_k$. 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

4. Reconstruction of a solution?

   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 kth entry: binary search on positions $\{1, \dots, k\}$ 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:

$$\Theta(n \log n).$$

20.7 Editing Distance

Minimal Editing Distance

Editing distance of two sequences $A_n = (a_1, \dots, a_n)$, $B_m = (b_1, \dots, b_m)$.

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[31]	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→Z +E -R
Z I E G E	Z I E G E _	Z→T -E +R

[31] Longest common subsequence – A special case of an editing problem

Idea

Z I E G E → T I G E R

Possibilities

1.
$$c(\texttt{'ZIEG'} \to \texttt{'TIGE'}) + c(\texttt{'E'} \to \texttt{'R'})$$
Z I E G **E** → T I G E **R**

2.
$$c(\texttt{'ZIEGE'} \to \texttt{'TIGE'}) + c(\mathsf{ins}(\texttt{'R'}))$$
Z I E G E → T I G E **+ R**

3.
$$c(\texttt{'ZIEG'} \to \texttt{'TIGER'}) + c(\mathsf{del}(\texttt{'E'}))$$
Z I E G E **- E** → T I G E R

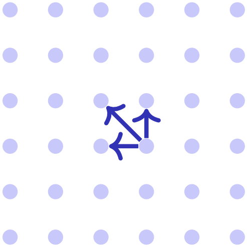
DP

0. $E(n,m)$ = mimimum number edit operations (ED cost) $a_{1...n} \rightarrow b_{1...m}$
1. Subproblems $E(i,j)$ = ED of $a_{1...i}$, $b_{1...j}$. #SP $= n \cdot m$
2. Guess CostsΘ(1)
   - $a_{1..i} \rightarrow a_{1...i-1}$ (delete)
   - $a_{1..i} \rightarrow a_{1...i}b_j$ (insert)
   - $a_{1..i} \rightarrow a_{1...i-1}b_j$ (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 DP algorithm ED

Dimension of the table? Semantics?

1. Table $E[0,\dots,m][0,\dots,n]$. $E[i,j]$: minimal edit distance of the strings $(a_1,\dots,a_i)$ and $(b_1,\dots,b_j)$

Computation of an entry

2. $E[0,i] \leftarrow i \; \forall 0 \le i \le m$, $E[j,0] \leftarrow i \; \forall 0 \le j \le n$. Computation of $E[i,j]$ otherwise via $E[i,j] = \min\{$del$(a_i)+E(i-1,j)$, ins$(b_j)+E(i,j-1)$, 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).

Reconstruction of a solution?

4.

Start with $j = m, i = n$. If $E[i, j] = \text{repl}(a_i, b_j) + E(i-1, j-1)$ then output $a_i \to b_j$ 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 $j = 0$.

Analysis ED

- 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:

$$\mathcal{O}(mn).$$

Matrix-Chain-Multiplication

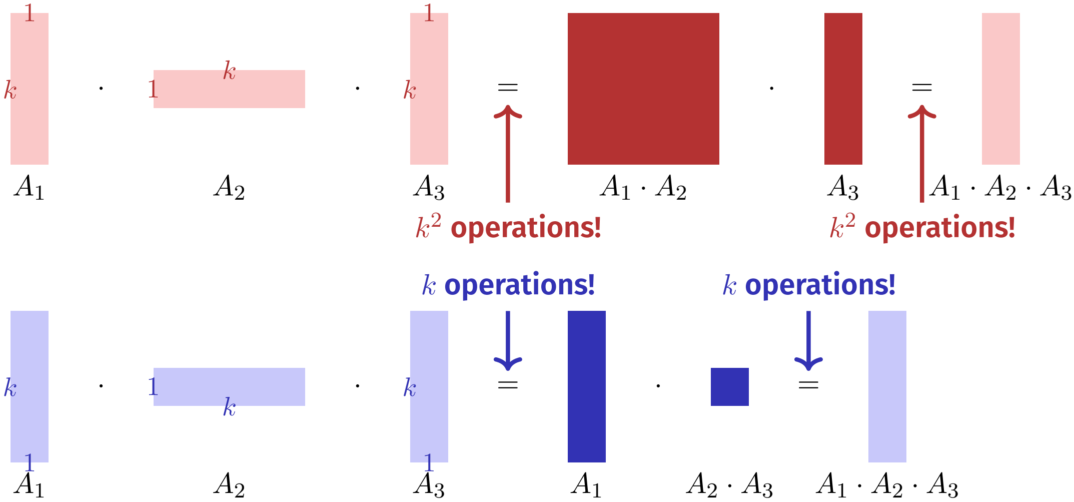
Task: Computation of the product $A_1 \cdot A_2 \cdot ... \cdot A_n$ of matrices $A_1, \ldots, A_n$.

Matrix multiplication is associative, i.e. the order of evaluation can be chosen arbitrarily

Goal: efficient computation of the product.

Assumption: multiplication of an $(r \times s)$-matrix with an $(s \times u)$-matrix provides costs $r \cdot s \cdot u$.

Does it matter?



k^2 operations!

k^2 operations!

k operations!

k operations!

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 $(A_p \cdot A_{p+1} \cdots A_q)$.

$$M[p, q] \leftarrow \min_{p \leq i < q} (M[p, i] + M[i + 1, q] + \text{costs of the last multiplication})$$

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 multiplication of two $n \times n$ matrices.
Let

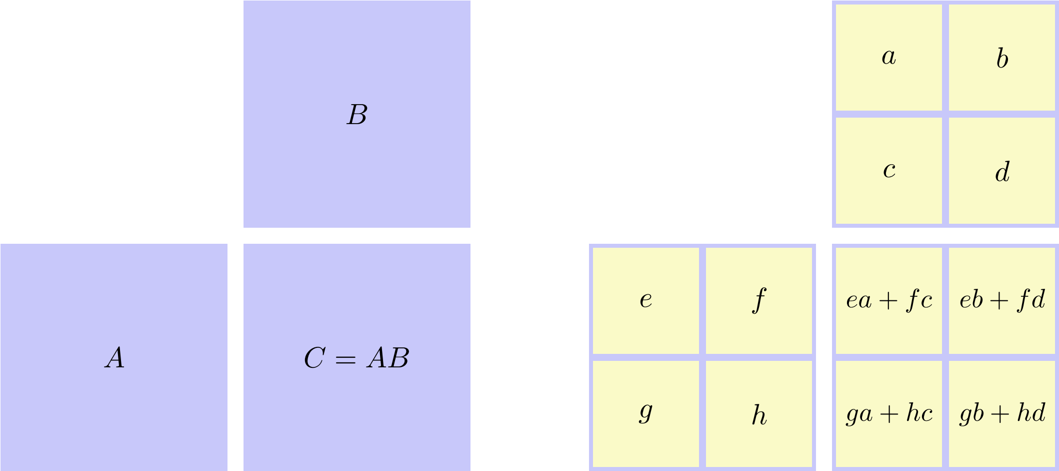
$$\begin{aligned}&A = (a_{ij})_{1 \le i,j \le n}, B = (b_{ij})_{1 \le i,j \le n}, C = (c_{ij})_{1 \le i,j \le n},\\&C = A \cdot B\end{aligned}$$

then

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Naive algorithm requires $\Theta(n^3)$ elementary multiplications.

Divide and Conquer



B

A

$C = AB$

a b

c d

e f

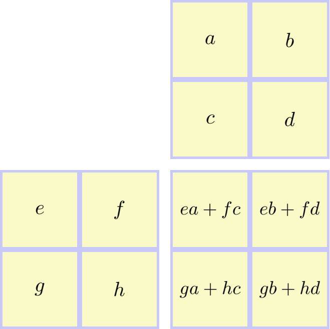
g h

$ea + fc$ $eb + fd$

$ga + hc$ $gb + hd$

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 ☹

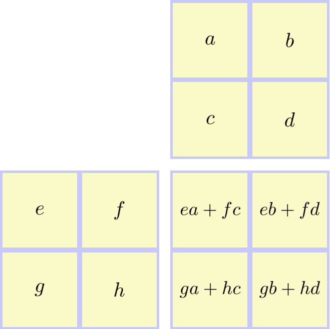


$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

$$\begin{pmatrix} e & f \\ g & h \end{pmatrix} \quad \begin{pmatrix} ea + fc & eb + fd \\ ga + hc & gb + hd \end{pmatrix}$$

Strassen’s Matrix Multiplication

- Nontrivial observation by Strassen (1969): It suffices to compute the seven products $A = (e+h)\cdot(a+d)$, $B = (g+h)\cdot a$, $C = e\cdot(b-d)$, $D = h\cdot(c-a)$, $E = (e+f)\cdot d$, $F = (g-e)\cdot(a+b)$, $G = (f-h)\cdot(c+d)$. Because: $ea+fc = A+D-E+G$, $eb+fd = C+E$, $ga+hc = B+D$, $gb+hd = A-B+C+F$.
- 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: $\mathcal{O}(n^{2.37})$



a b
c d

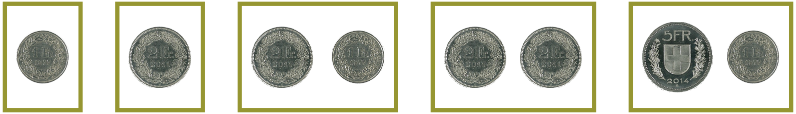
e f
g h

$ea+fc$ $eb+fd$
$ga+hc$ $gb+hd$

21. Dynamic Programming II

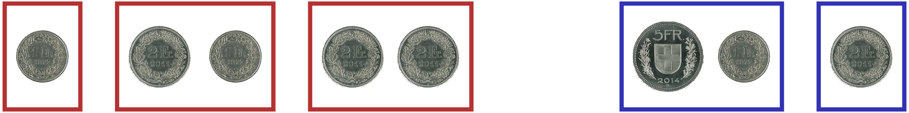
Subset sum problem, knapsack problem, greedy algorithm vs dynamic programming [Ottman/Widmayer, Kap. 7.2, 7.3, 5.7, Cormen et al, Kap. 15,35.5]

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 $b = (b_1, \dots, b_n) \in \{0,1\}^n$, 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 $a_1, \dots, a_{n/2}$ and $a_{n/2+1}, \dots, a_n$.
- Iterate over all subsets of the two parts and compute partial sum $S_1^k, \dots, S_{2^{n/2}}^k$ $(k = 1, 2)$.
- Sort the partial sums: $S_1^k \le S_2^k \le \dots \le S_{2^{n/2}}^k$.
- Check if there are partial sums such that $S_i^1 + S_j^2 = \frac{1}{2}\sum_{i=1}^n a_i =: h$
  - Start with $i = 1, j = 2^{n/2}$.
  - If $S_i^1 + S_j^2 = h$ then finished
  - If $S_i^1 + S_j^2 > h$ then $j \leftarrow j - 1$
  - If $S_i^1 + S_j^2 < h$ 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: $\mathcal{O}(2^{n/2} \log(2^{n/2})) = \mathcal{O}(n2^{n/2})$.
- 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!

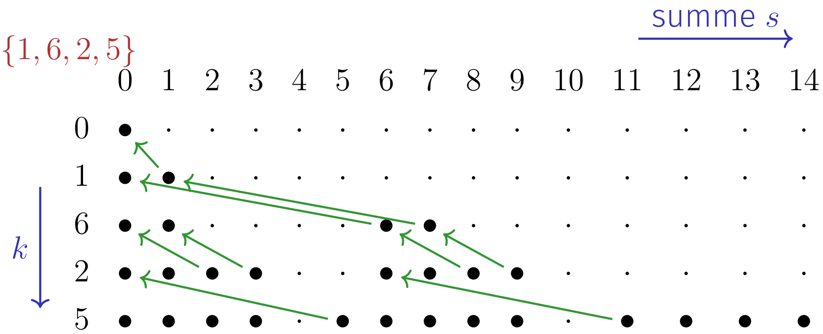
Dynamic programming

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



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	●	·	·	·	·	·	·	·	·	·	·	·	·	·	·
1	●	●	·	·	·	·	·	·	·	·	·	·	·	·	·
6	●	●	·	·	·	·	●	●	·	·	·	·	·	·	·
2	●	●	●	●	·	·	●	●	●	●	·	·	·	·	·
5	●	●	●	●	·	●	●	●	●	●	·	●	●	●	●

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 ζ.

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[32] that NP $\neq$ P, there is **no algorithm with polynomial run time** for the problem considered above.

[32] 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 $\sum_{i \in I} w_i \le W$.

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 $m[i, w, v] = \text{true}$ it holds:
  $m[i', w, v] = \text{true } \forall i' \geq i$,
  $m[i, w', v] = \text{true } \forall w' \geq w$,
  $m[i, w, v'] = \text{true } \forall v' \leq v$.
- fpr $m[i, w, v] = \text{false}$ it holds:
  $m[i', w, v] = \text{false } \forall i' \leq i$,
  $m[i, w', v] = \text{false } \forall w' \leq w$,
  $m[i, w, v'] = \text{false } \forall v' \geq v$.

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[33] with

- items $1, \ldots, i$ $(0 \leq i \leq n)$
- at maximum weight w $(0 \leq w \leq W)$.

[33] We could have followed a similar idea in order to reduce the size of the sparse table for subset sum.

Computation

Initially

- $t[0, w] \leftarrow 0$ for all $w \geq 0$.

We compute

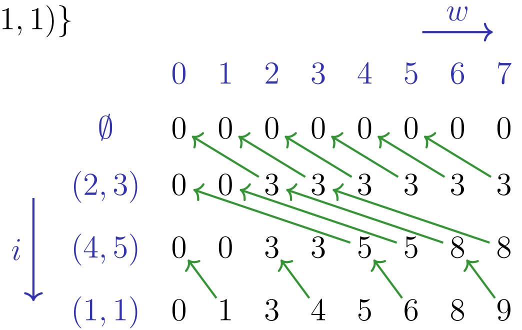
$$t[i, w] \leftarrow \begin{cases} t[i-1, w] & \text{if } w < w_i \\ \max\{t[i-1, w], t[i-1, w-w_i] + v_i\} & \text{otherwise.} \end{cases}$$

increasing by i and for fixed i increasing by w.

Solution is located in $t[n, w]$

Example

$E = \{(2,3),(4,5),(1,1)\}$



	0	1	2	3	4	5	6	7
$\emptyset$	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

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 but can yield an arbitrarily bad result.

22. Dynamic Programming III

Optimal Search Tree [Ottman/Widmayer, Kap. 5.7]

22.1 Optimal Search Trees

Optimal binary Search Trees

Given: n keys $k_1, k_2 \ldots k_n$ (wlog $k_1 < k_2 < ... < k_n$) with weights (search probabilities[34]) $p_1, p_2, \ldots, p_n$.

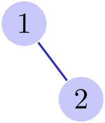
Wanted: optimal search tree T with key depths[35] $\mathrm{d}(\cdot)$, that minimizes the expected search costs

$$C(T) = \sum_{i=1}^{n} (\mathrm{d}(k_i) + 1) \cdot p_i$$

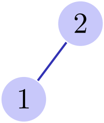
[34] It is possible to model unsuccesful search additionally, omitted for brevity here

[35] $d(k)$: Length of the path from the root to the node k

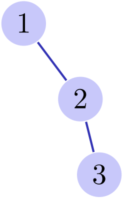
Examples



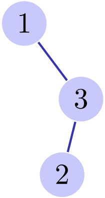
$2p_1 + p_2$



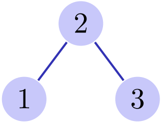
$p_1 + 2p_2$



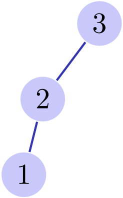
$p_1 + 2p_2 + 3p_3$



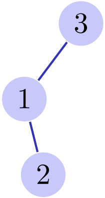
$p_1 + 3p_2 + 2p_2$



$2p_1 + p_2 + 2p_3$



$3p_1 + 2p_2 + p_3$

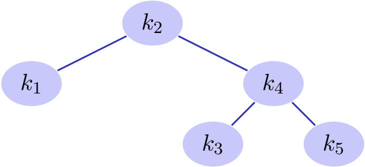


$2p_1 + 3p_3 + 1p_3$

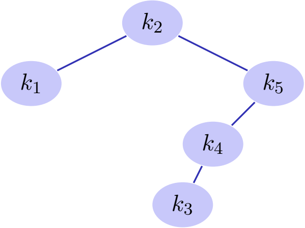
Example

Expected Frequencies

i	1	2	3	4	5
p_i	0.25	0.10	0.05	0.20	0.40

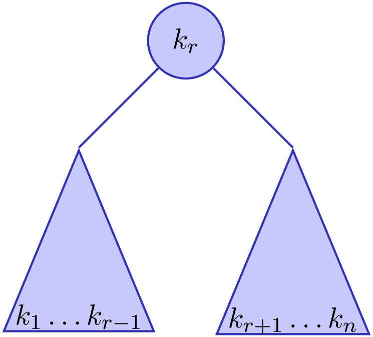


Search tree with expected costs 2.35



Search tree with expected costs 2.2

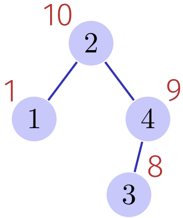
Sub-trees for Searching



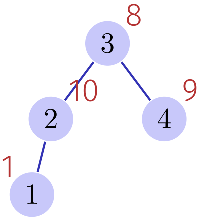
Which r to choose?

Greedy?

Scenario $p_1 = 1, p_2 = 10, p_3 = 8, p_4 = 9$



$c(T) = 54$



$c(T) = 49$

Structure of a optimal binary search tree

- Consider all subtrees with roots k_r and optimal subtrees for keys $k_i, \ldots, k_{r-1}$ and $k_{r+1}, \ldots, k_j$
- Subtrees with keys $k_i, \ldots, k_{r-1}$ and $k_{r+1}, \ldots, k_j$ must be optimal for the respective sub-problems.[36]

$$E(i, j) = \text{Costs of optimal search tree with nodes } k_i, k_{i+1}, \ldots, k_j$$

[36] The usual argument: if it was not optimal, it could be replaced by a better solution improving the overal solution.

Rekursion

With

$$p(i,j) := p_i + p_{i+1} + \cdots + p_j \qquad i \le j$$

it holds that

$$E(i,j) = \begin{cases} 0 & \text{if } i > j \\ p(i) & \text{if } i = j \\ p(i,j) + \min\{E(i,k-1) + E(k+1,j), i \le k \le j\} & \text{otherwise.} \end{cases}$$

DP

0. $E(1, n)$: Costs of optimal search tree with nodes $k_1, \ldots, k_n$ with search frequencies $p_1, \ldots, p_n$
1. $E(i, j), 1 \leq i \leq j \leq n$ # sub-problems $\Theta(n^2)$
2. Enumerate roots of subtree of $k_i, \ldots, k_j$, # possibilities: $j - i + 1$
3. Dependencies $E(i, j)$ depend on $E(i, k), E(k, j)$ $i < k < j$. Computation of the off-diagonals of E, starting with the diagonal of E
4. Solution is in $E(1, n)$, Reconstruction: store the arg-mins of the recursion in a separate table V.
5. Running time $\Theta(n^3)$. Memory $\Theta(n^2)$.

Example

i	1	2	3	4	5
p_i	0.25	0.10	0.05	0.20	0.40

E

i \ j	0	1	2	3	4	5
1	0	0.25	0.45	0.60	1.15	2.00
2		0	0.10	0.20	0.55	1.30
3			0	0.05	0.30	0.95
4				0	0.20	0.80
5					0	0.40
6						0

p

i \ j	1	2	3	4	5
1	0.25	0.35	0.40	0.60	1.00
2		0.10	0.15	0.35	0.75
3			0.05	0.25	0.65
4				0.20	0.60
5					0.40

V

i \ j	1	2	3	4	5
1	1	1	1	1	4
2		2	2	4	5
3			3	4	5
4				4	5
5					5

23. Greedy Algorithms

Fractional Knapsack Problem, Huffman Coding [Cormen et al, Kap. 16.1, 16.3]

Greedy Choice

A problem with a recursive solution can be solved with a **greedy algorithm** if it has the following properties:

- The problem has **optimal substructure**: the solution of a problem can be constructed with a combination of solutions of sub-problems.
- The problem has the **greedy choice property**: The solution to a problem can be constructed, by using a local criterion that is not depending on the solution of the sub-problems.

Examples: fractional knapsack, Huffman-Coding (below)
Counter-Example: knapsack problem, Optimal Binary Search Tree

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 $\{a, \ldots, f\}$.

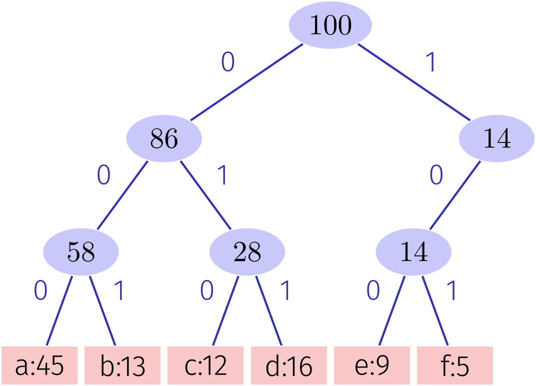
	a	b	c	d	e	f
Frequency (Thousands)	45	13	12	16	9	5
Code word with fix length	000	001	010	011	100	101
Code word variable length	0	101	100	111	1101	1100

File size (code with fix length): 300.000 bits.
File size (code with variable length): 224.000 bits.

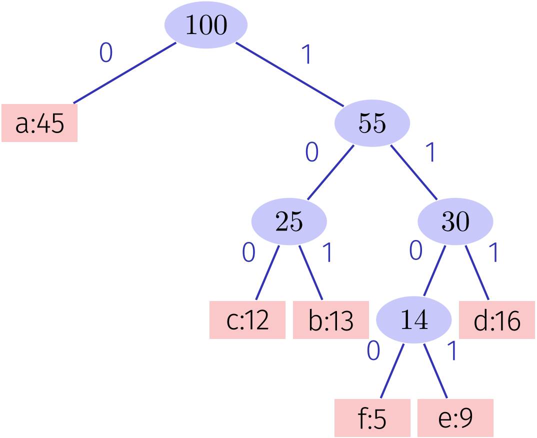
Huffman-Codes

- 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 $\rightarrow 0 \cdot 1100 \cdot 1100 \cdot 1101 \rightarrow 0110011001101$
- Decoding simple because prefixcode
  $0110011001101 \rightarrow 0 \cdot 1100 \cdot 1100 \cdot 1101 \rightarrow$ 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

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c).$$

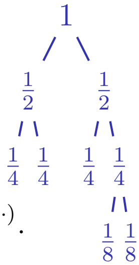
(cost = number bits of the encoded file)

In the following a code tree is called optimal when it minimizes the costs.

Probabilitiy Distributions

The sum to be minimized

$$\sum_{c \in C} f(c) \cdot d_T(c)$$



1
$\frac{1}{2}$ $\frac{1}{2}$
$\frac{1}{4}$ $\frac{1}{4}$ $\frac{1}{4}$ $\frac{1}{4}$
$\frac{1}{8}$ $\frac{1}{8}$

can be written as

$$-\sum_{c \in C} f(c) \cdot \log_2 g_T(c), \text{ where } g_T(\cdot) := 2^{-d_T(\cdot)}.$$

$g_T(\cdot)$ can be understood as discrete probability distribution because it holds that $\sum_c g_T(c) = 1$. That is a property of a complete binary tree because each inner node has two child nodes.

Probabilitiy Distributions

For two discrete proability distributions f and g over C the **Gibbs inequality** holds

$$\underbrace{-\sum_{c\in C} f(c)\log f(c)}_{\text{Entropy of } f} \leq -\sum_{c\in C} f(x)\log g(c)$$

with equality if and only if $f(c) = g(c)$ for each $c \in C$.

Consequence if $f(c) \in \{2^{-k}, k \in \mathbb{N}\}$ for all $c \in C$, then the optimal code tree can be formed easily with $d_T(c) = -\log_2 f(c)$.

Shannon Fano Coding

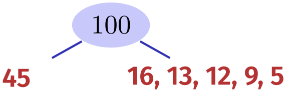
Approximative algorithm of Shannon and Fano

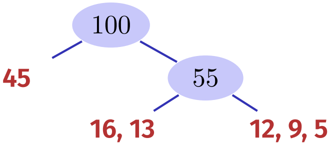
1. Sort the keys by frequency, wlog $p_1 \leq p_2 \leq ... \leq p_n$
2. Partition the keys into two sets of almost equal weight, i.e. into sets $A = \{1, \dots, k\}$ and $B = \{k+1, \dots, n\}$ such that $\sum_{i \in A} p_i \approx \sum_{i \in B} p_i$. Recursion until all sets contain a single element.

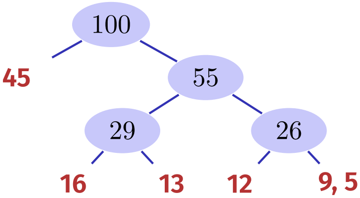
Running Time: $\Theta(n \log n)$

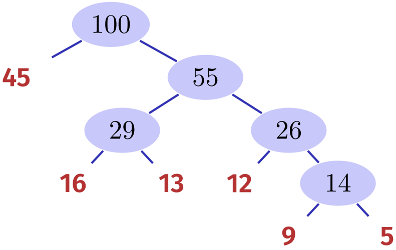
Shannon Fano Coding

45, 16, 13, 12, 9, 5





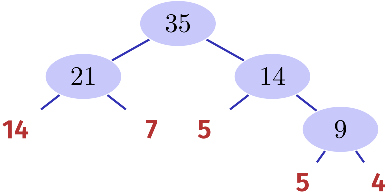




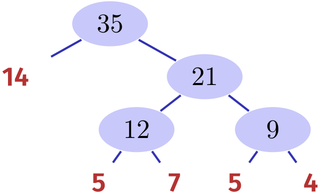
Problem

The approximate algorithm of Shannon and Fano does not always provide the optimal result

Example $\{14, 7, 5, 5, 4\}$ with lower bound (entropy) $B(T) \geq 75.35$



Shannon-Fano Coding, $B(T) = 79$

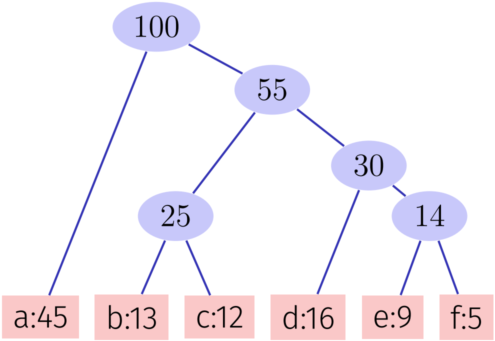


Optimal, $B(T) = 77$

Huffman’s 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 ← ExtractMin($Q$)                // extract word with minimal frequency.
    $z$.right ← ExtractMin($Q$)
    $z$.freq ← $z$.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 20

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.

Recursive Problem-Solving Strategies

Brute Force Enumeration	**Backtracking**	**Divide and Conquer**	**Dynamic Programming**	**Greedy**
Recursive Enumerability	Constraint Satisfaction, Partial Validation	Optimal Substructure	Optimal Substructure, Overlapping Subproblems	Optimal Substructure, Greedy Choice Property
DFS, BFS, all Permutations, Tree Traversal	n-Queen, Sudoku, m-Coloring, SAT-Solving, naive TSP	Binary Search, Mergesort, Quicksort, Hanoi Towers, FFT	Bellman Ford, Warshall, Rod-Cutting, LAS, Editing Distance, Knapsack Problem DP	Dijkstra, Kruskal, Huffmann Coding

24. Geometric Algorithms

Properties of Line Segments, Intersection of Line Segments, Convex Hull, Closest Point Pair [Ottman/Widmayer, Kap. 8.2,8.3,8.8.2, Cormen et al, Kap. 33]

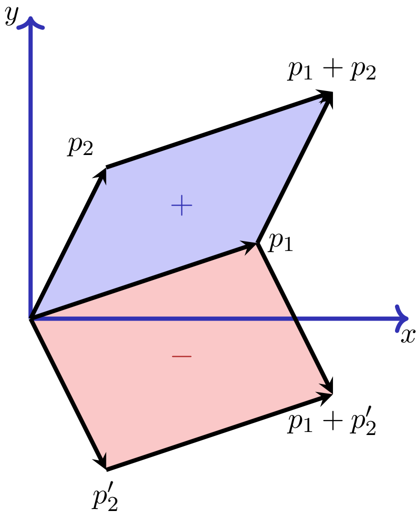
24.1 Properties of Line Segments

Properties of line segments.

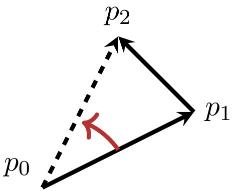
Cross-Product of two vectors $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ in the plane

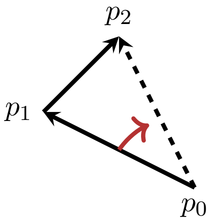
$$p_1 \times p_2 = \det \begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \end{bmatrix} = x_1 y_2 - x_2 y_1$$

Signed area of the parallelogram



Turning direction

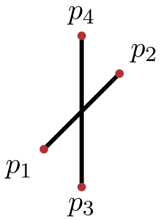




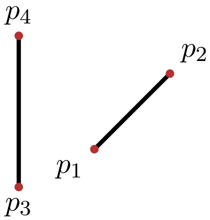
nach links:
$(p_1 - p_0) \times (p_2 - p_0) > 0$

nach rechts:
$(p_1 - p_0) \times (p_2 - p_0) < 0$

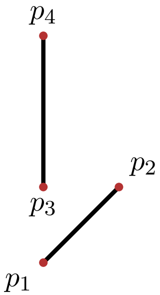
Intersection of two line segments



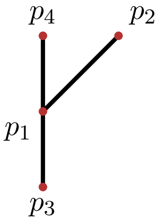
Intersection: p_1 and p_2 opposite w.r.t $\overline{p_3p_4}$ and p_3, p_4 opposite w.r.t. $\overline{p_1p_2}$



No intersection: p_1 and p_2 on the same side of $\overline{p_3p_4}$



No intersection: p_3 and p_4 on the same side of $\overline{p_1p_2}$



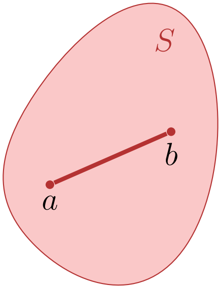
Intersection: p_1 on $\overline{p_3p_4}$

24.2 Convex Hull

Convex Hull

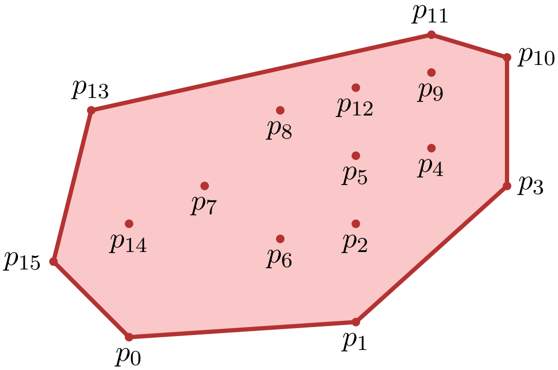
Subset S of a real vector space is called **convex**, if for all $a, b \in S$ and all $\lambda \in [0, 1]$:

$$\lambda a + (1 - \lambda)b \in S$$



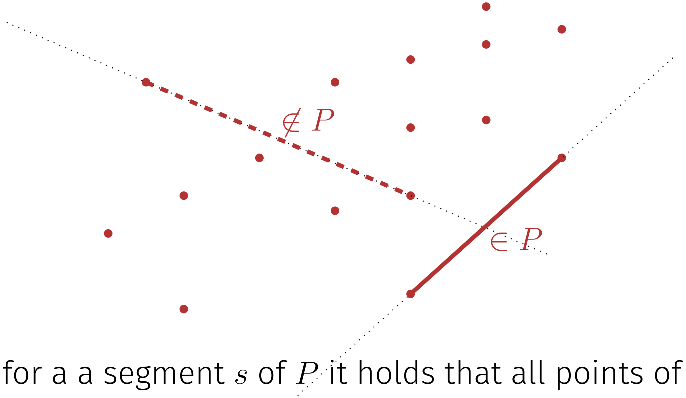
Convex Hull

Convex Hull $H(Q)$ of a set Q of points: smallest convex polygon P such that each point of Q is on P or in the interior of P.



Convex Hull

Identify segments of P

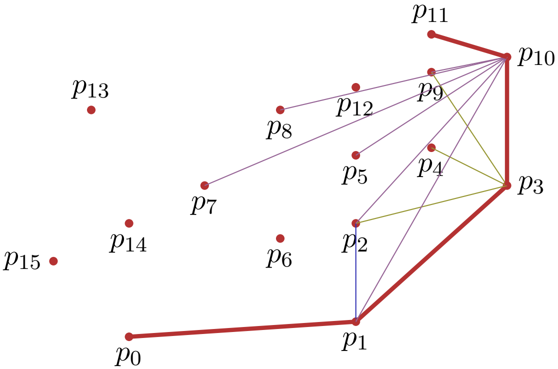


Observation: for a a segment s of P it holds that all points of Q not on the line through s are either on the left or on the right of s.

Jarvis Marsch / Gift Wrapping algorithm

1. Start with an extremal point (e.g. lowest point) $p = p_0$
2. Search point q, such that $\overline{pq}$ is a line to the right of all other points (or other points are on this line closer to p.
3. Continue with $p \leftarrow q$ at (2) until $p = p_0$.

Illustration Jarvis

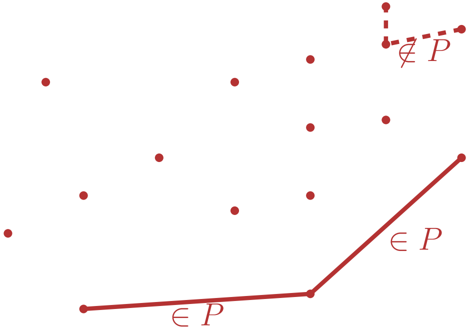


Analysis Gift-Wrapping

- Let h be the number of corner points of the convex hull.
- Runtime of the algorithm $\mathcal{O}(h \cdot n)$.

Convex Hull

Identify segments of P

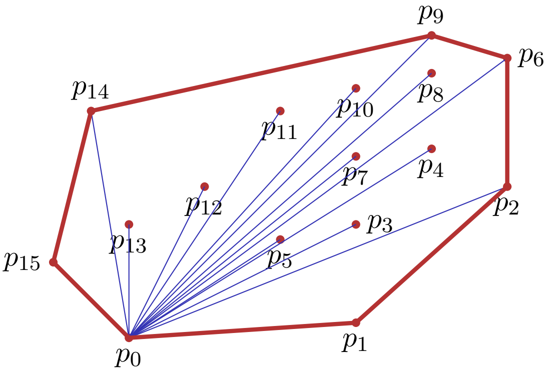


Observation: if the points of the polygon are ordered anti-clockwise then subsequent segments of the polygon P only make left turns.

Algorithm Graham-Scan

```
Input: Set of points $Q$
Output: Stack $S$ of points of the convex hull of $Q$
$p_0$: point with minimal $y$ coordinate (if required, additionally minimal $x$-)
  coordinate
$(p_1, \ldots, p_m)$ remaining points sorted by polar angle counter-clockwise in relation to
  $p_0$; if points with same polar angle available, discard all except the one with
  maximal distance from $p_0$
$S \leftarrow \emptyset$
if $m < 2$ then return S
Push($S, p_0$); Push($S, p_1$); Push($S, p_2$)
for $i \leftarrow 3$ to $m$ do
    while Winkel (NextToTop($S$), Top($S$), $p_i$) nicht nach links gerichtet do
        Pop($S$);
    Push($S, p_i$)
return $S$
```

Illustration Graham-Scan



Stack:

p_{15}

p_{14}

p_9

p_6

p_2

p_1

p_0

Analysis

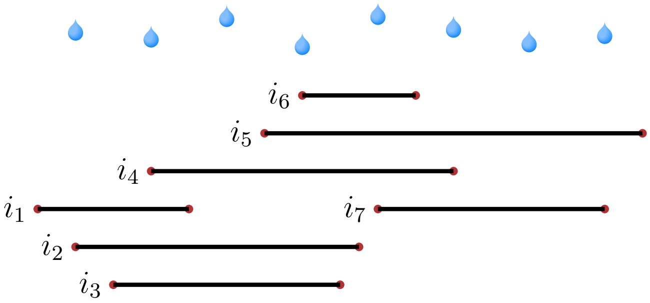
Runtime of the algorithm Graham-Scan

- Sorting $\mathcal{O}(n \log n)$
- n Iterations of the for-loop
- Amortized analysis of the multipop on a stack: amortized constant runtime of mulitpop, same here: amortized constant runtime of the While-loop.

Overal $\mathcal{O}(n \log n)$

24.3 Intersection of Line Segments

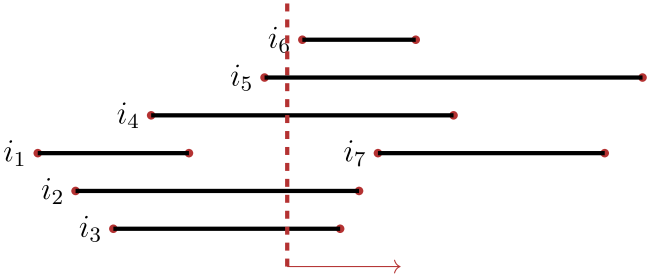
Preparation: Overlapping Intervals



Questions:

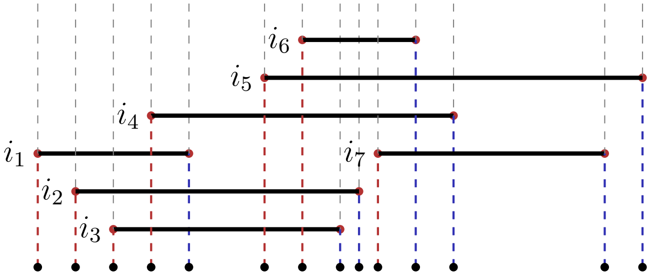
- How many intervals overlap maximally?
- Which intervals (don't) get wet?
- Which intervals are directly on top of each other?

Preparation: Overlapping Intervals



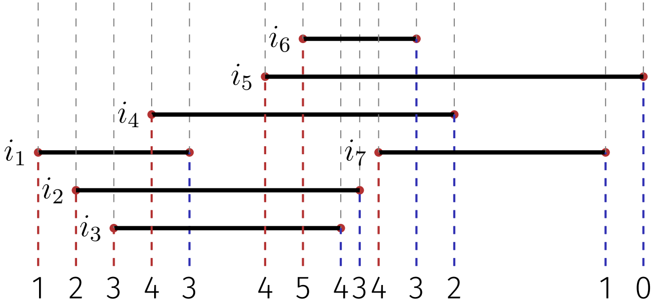
Idea of a sweep line: vertical line, moving in x-direction, observes the geometric objects.

Preparation: Overlapping Intervals



Event list: list of points where the state observed by the sweepline changes.

Preparation: Overlapping Intervals

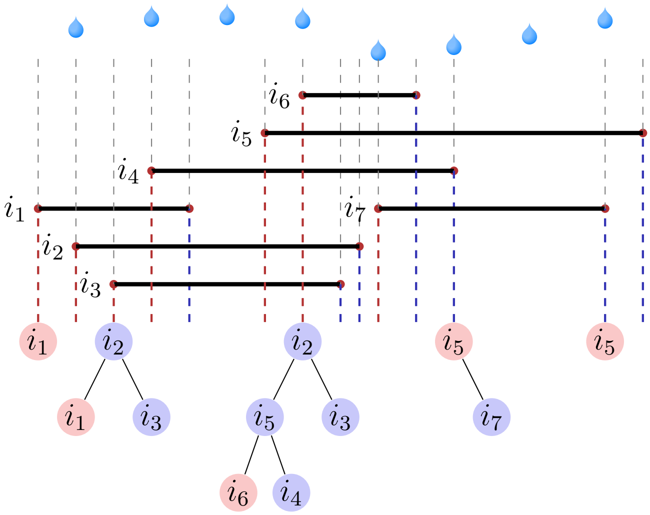


Q: How many intervals overlap maximally?

Sweep line controls a counter that is incremented (decremented) at the left (right) end point of an interval.

A: maximum counter value

Preparation: Overlapping Intervals

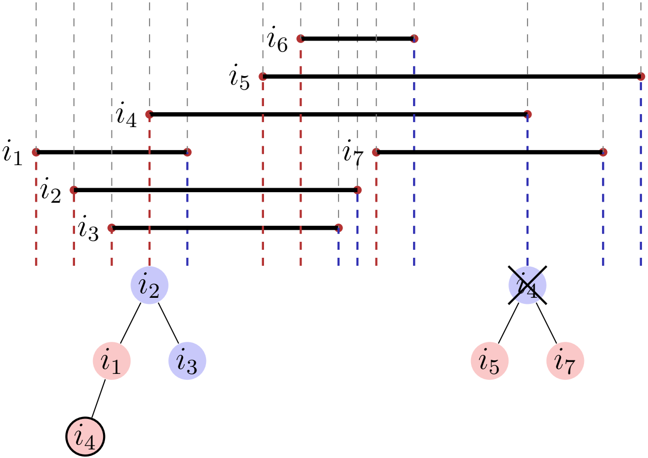


Q: Which intervals get wet?

Sweep line controls a binary search tree that comprises the intervals according to their vertical ordering.

A: intervalls on the very left of the tree.

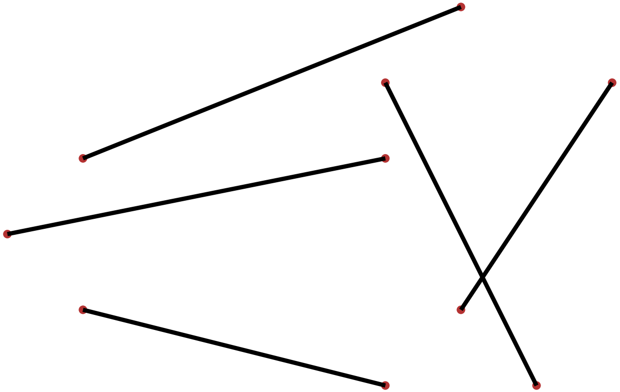
Preparation: Overlapping Intervals



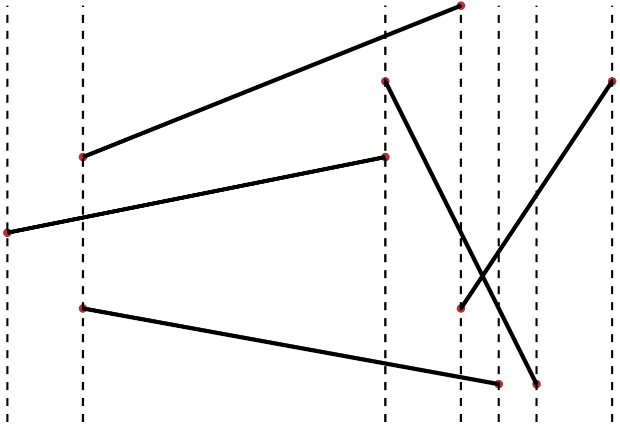
Q: Which intervals are neighbours?

A: intervalls on the very left of the tree.

Cutting many line segments



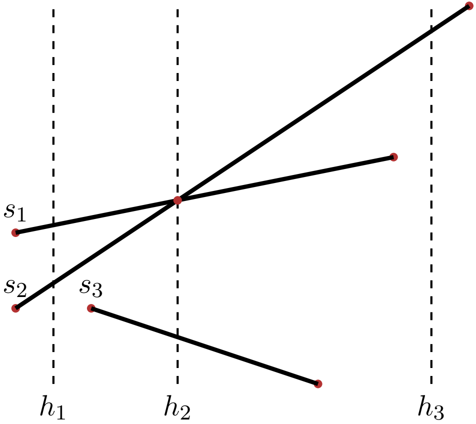
Sweepline Principle



Simplifying Assumptions

- No vertical line segments
- Each intersection is formed by at most two line segments.

(Vertical) Ordering line segments



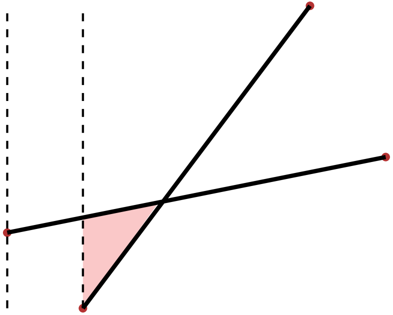
Preorder (partial order without anti-symmetry)[37]

$$s_2 \preccurlyeq_{h_1} s_1$$
$$s_1 \preccurlyeq_{h_2} s_2$$
$$s_2 \preccurlyeq_{h_2} s_1$$
$$s_3 \preccurlyeq_{h_2} s_2$$

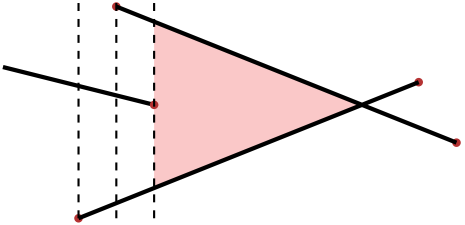
W.r.t. h_3 the line segments are uncomparable.

[37] No anti-symmetry: $s \preccurlyeq t \wedge t \preccurlyeq s \not\Rightarrow s = t$

Observation: two cases

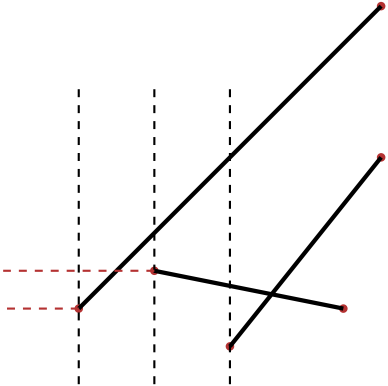


(a) Intersecting line segments are neighbours w.r.t. quasi-order from above directly from the start.



(b) Intersecting line segments are neighbours w.r.t. quasi-order from above after the last segment between them ends.

Observation: possible misunderstanding



It does not suffice to compare the y-coordinates of starting points of lines.
Positions on the sweep line have to be compared.

Moving the sweepline

- **Sweep-Line Status** : Relationship of all objects intersected by sweep-line
- **Event List** : Series of event positions, sorted by x-coordinate. Sweep-line travels from left to right and stops at each event position.

Sweep-Line Status

Preorder T of the intersected line segments

Required operations:

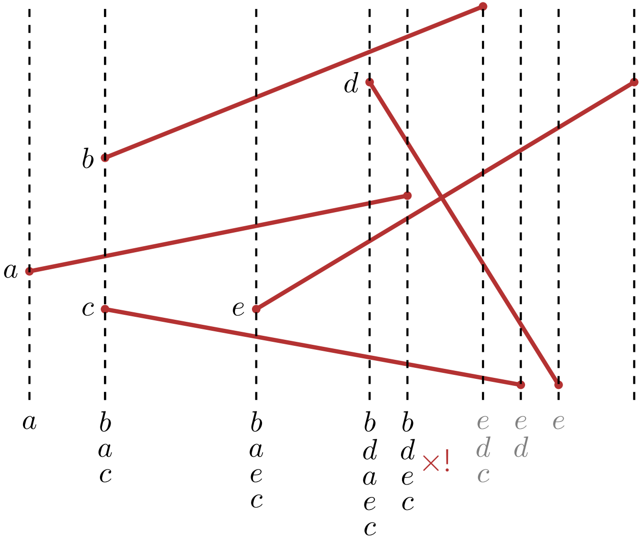
- **Insert(**T, s**)** Insert line segment s in T
- **Delete(**T, s**)** Remove s from T
- **Above(**T, s**)** Return line segment immediately above of s in T
- **Below(**T, s**)** Return line segment immediately below of s in T

Possible Implementation: Blanced tree (AVL-Tree, Red-Black Tree etc.)

Algorithm Any-Segments-Intersect(S)

```
Input: List of $n$ line segments $S$
Output: Returns if $S$ contains intersecting segments
$T \leftarrow \emptyset$
Sort endpoints of line segments in $S$ from left to right (left before right and lower
  before upper)
for Sorted end points $p$ do
    if $p$ left end point of a segment $s$ then
        Insert($T, s$)
        if Above($T, s$) $\cap\, s \neq \emptyset \;\vee\;$ Below($T, s$) $\cap\, s \neq \emptyset$ then return true
    if $p$ right end point of a segment $s$ then
        if Above($T, s$) $\cap$ Below($T, s$) $\neq \emptyset$ then return true
        Delete($T, s$)
return false;
```

Illustration



Analysis

Runtime of the algorithm Any-Segments-Intersect

- Sorting $\mathcal{O}(n \log n)$
- $2n$ iterations of the for loop. Each operation on the balanced tree $\mathcal{O}(\log n)$

Overal $\mathcal{O}(n \log n)$

24.4 Closest Point Pair

Closest Point Pair

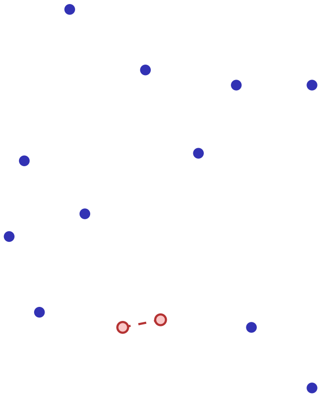
Euclidean Distance $d(s,t)$ of two points s and t:

$$d(s,t) = \|s-t\|_2 = \sqrt{(s_x - t_x)^2 + (s_y - t_y)^2}$$

Problem: Find points p and q from Q for which

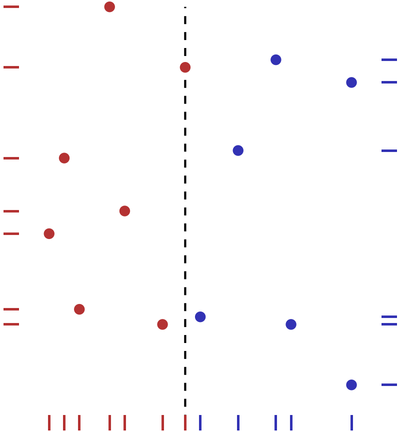
$$d(p,q) \le d(s,t) \; \forall \; s,t \in Q, s \neq t.$$

Naive: all $\binom{n}{2} = \Theta(n^2)$ point pairs.



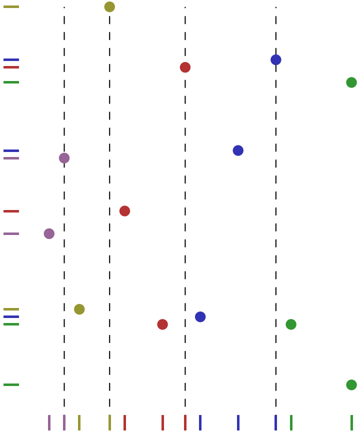
Divide And Conquer

- Set of points P, starting with $P \leftarrow Q$
- Arrays X and Y, containing the elements of P , sorted by x- and y-coordinate, respectively.
- Partition point set into two (approximately) equally sized sets P_L and P_R, separated by a vertical line through a point of P.
- Split arrays X and Y accordingly in X_L, X_R. Y_L and Y_R.



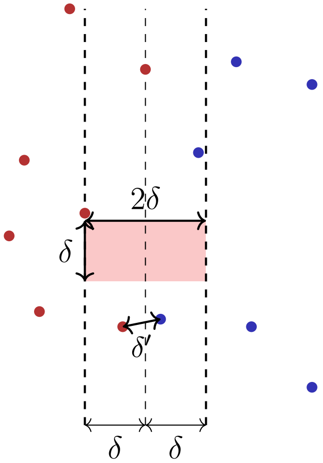
Divide And **Conquer**

- Recursive call with P_L, X_L, Y_L and P_R, X_R, Y_R. Yields minimal distances δ_L, δ_R.
- (If only $k \leq 3$ points: compute the minimal distance directly)
- After recursive call $\delta = \min(\delta_L, \delta_R)$. Combine (next slides) and return best result.

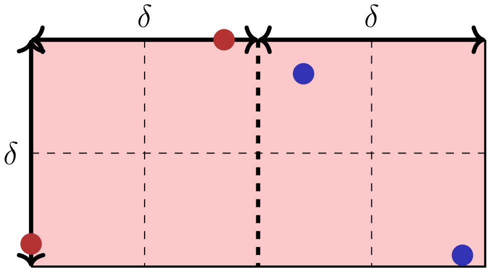


Combine

- Generate an array Y' holding y-sorted points from Y, that are located within a 2δ band around the dividing line
- Consider for each point $p \in Y'$ the maximally seven points after p with y-coordinate distance less than δ. Compute minimal distance δ'.
- If $\delta' < \delta$, then a closer pair in P than in P_L and P_R found. Return minimal distance.



Maximum number of points in the 2δ-rectangle



Two points in the $\delta/2 \times \delta/2$-rectangle have maximum distance $\frac{\sqrt{2}}{2}\delta < \delta$.
$\Rightarrow$ Square with side length $\delta/2$ contains a maximum of one point.
Eight non-overlapping $\delta/2 \times \delta/2$-Rectangles span the $2\delta \times \delta$ rectangle.

Implementation

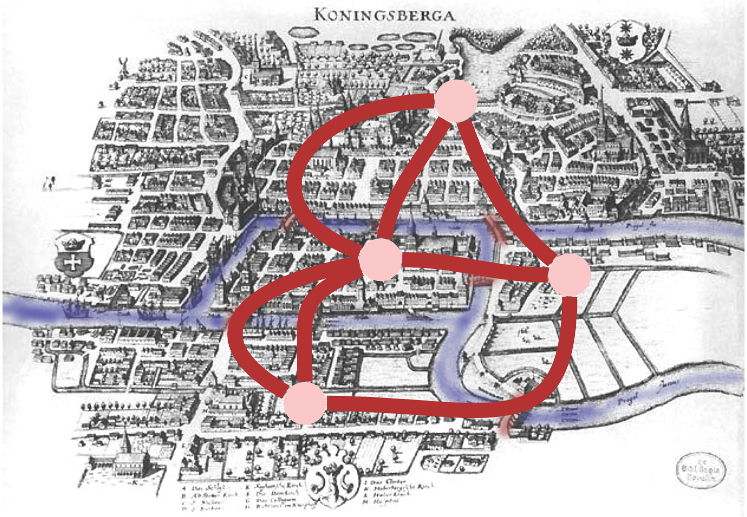
- Goal: recursion equation (runtime) $T(n) = 2 \cdot T(\frac{n}{2}) + \mathcal{O}(n)$.
- Consequence: forbidden to sort in each steps of the recursion.
- Non-trivial: only arrays Y and Y'
- Idea: merge reversed: run through Y that is presorted by y-coordinate. For each element follow the selection criterion of the x-coordinate and append the element either to Y_L or Y_R. Same procedure for Y'. Runtime $\mathcal{O}(|Y|)$.

Overal runtime: $\mathcal{O}(n \log n)$.

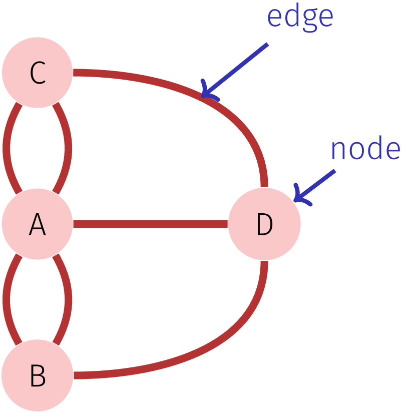
25. Graphs

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



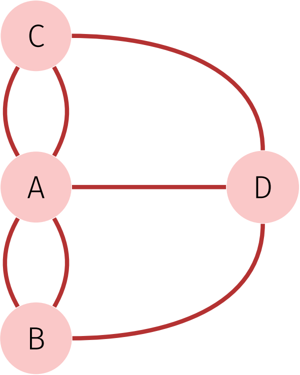
[Multi]Graph



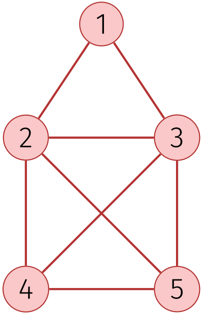
Cycles

- 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 ⇔ each node provides an even number of edges (each node is of an *even degree*).

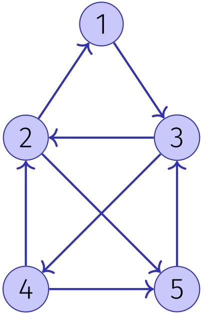
  '⇒" is straightforward, "⇐" ist a bit more difficult but still elementary.



Notation



undirected



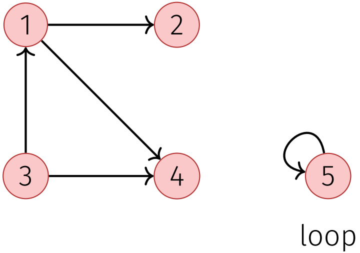
directed

$$V = \{1, 2, 3, 4, 5\}$$
$$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 4\}, \{3, 5\}, \{4, 5\}\}$$

$$V = \{1, 2, 3, 4, 5\}$$
$$E = \{(1, 3), (2, 1), (2, 5), (3, 2), (3, 4), (4, 2), (4, 5), (5, 3)\}$$

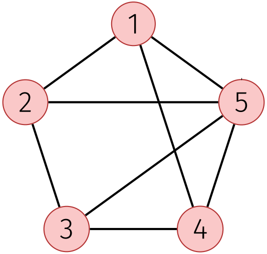
Notation

A **directed graph** consists of a set $V = \{v_1, \ldots, v_n\}$ 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 not be contained more than once.[38]

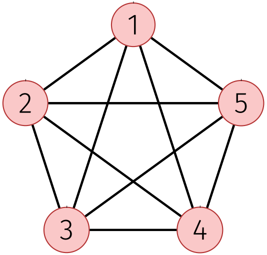


undirected graph

[38] 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**.

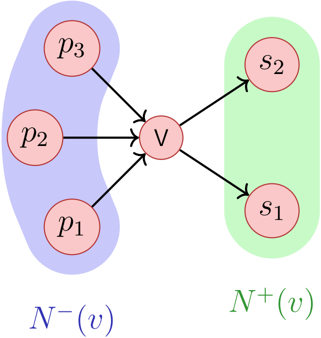


a complete undirected graph

Notation

For directed graphs $G = (V, E)$

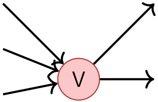
- $w \in V$ is called adjacent to $v \in V$, if $(v, w) \in E$
- **Predecessors** of $v \in V$: $N^-(v) := \{u \in V | (u, v) \in E\}$.
  Successors: $N^+(v) := \{u \in V | (v, u) \in E\}$



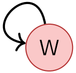
Notation

For directed graphs $G = (V, E)$

- **In-Degree**: $\deg^-(v) = |N^-(v)|$,
  Out-Degree: $\deg^+(v) = |N^+(v)|$



$\deg^-(v) = 3$, $\deg^+(v) = 2$

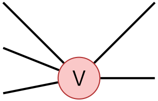


$\deg^-(w) = 1$, $\deg^+(w) = 1$

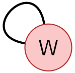
Notation

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.



$\deg(v) = 5$



$\deg(w) = 2$

Node Degrees $\leftrightarrow$ Number of Edges

Handshaking Lemma:

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.
- **Simple path**: path without repeating vertices

Connectedness

- An undirected graph is called **connected**, if for each 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)

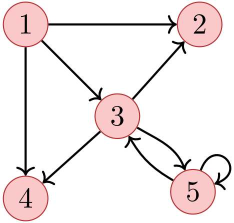
Cycles

- **Cycle**: path $\langle v_1, \dots, v_{k+1} \rangle$ with $v_1 = v_{k+1}$
- **Simple cycle**: Cycle with pairwise different $v_1, \dots, v_k$, 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.

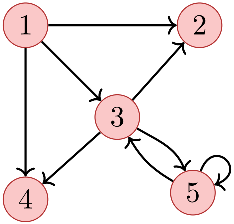


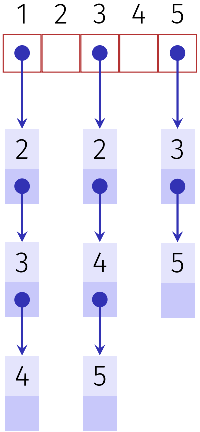
$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

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 $N^+(v_i)$.





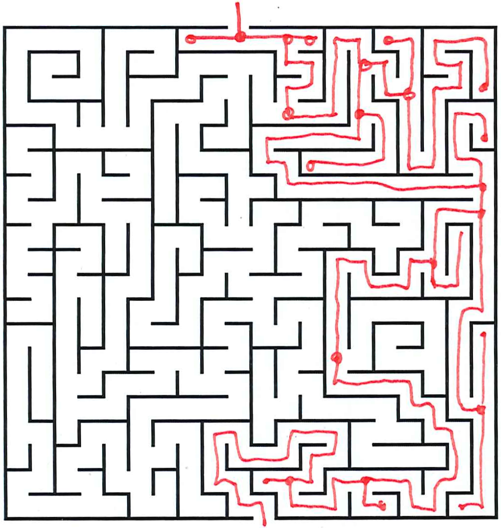
Memory Consumption $\Theta(|V| + |E|)$.

Runtimes of simple Operations

Operation	Matrix	List
Find neighbours/successors of $v \in V$		
find $v \in V$ without neighbour/successor		
$(v, u) \in E$?		
Insert edge		
Delete edge (v, u)		

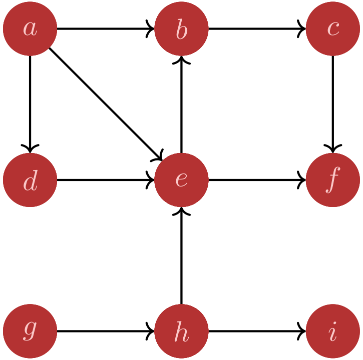
Exercise Class

Depth First Search



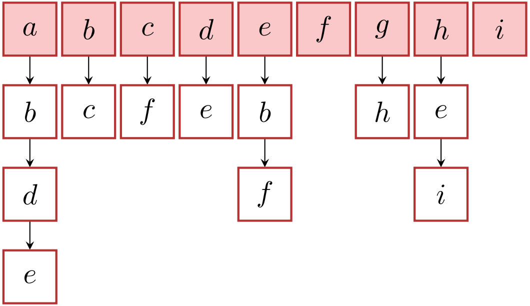
Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.



Order $a, b, c, f, d, e, g, h, i$

adjacency list



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
// visit $v$
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: $\Theta(|V| + \sum_{v \in V}(\deg^+(v) + 1)) = \Theta(|V| + |E|)$.

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: cycle (“back-edge”)
- Black node: forward- / cross edge

[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($v$)
```

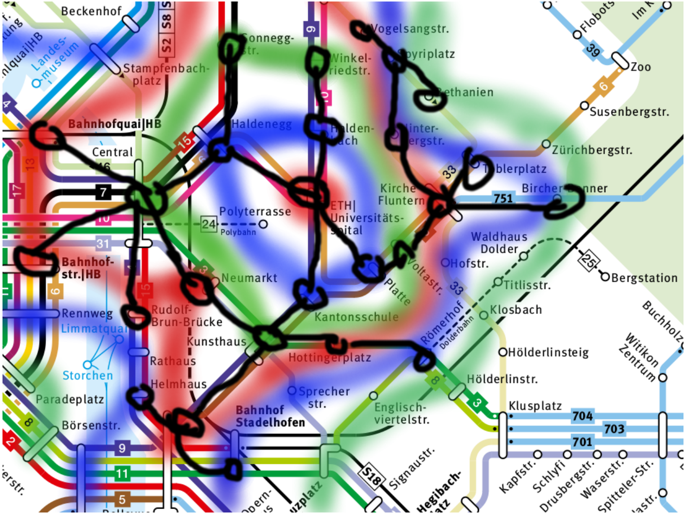
Memory Consumption Stack $\Theta(|V|)$

[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
```

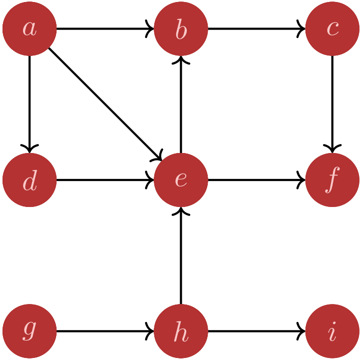
There are simpler variants of iterative depth first search. Howeber, they do not admit the same kind of interpretation of the edges between colored nodes. Moreover, they usually have a worst-case memory consumption of $\Theta(|E|)$

Breadth First Search



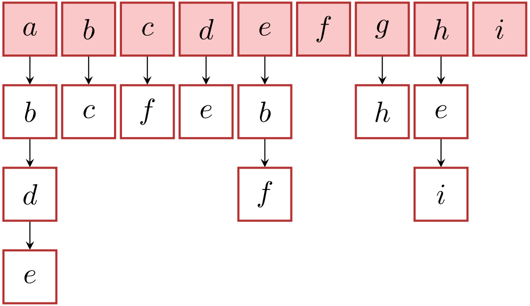
Graph Traversal: Breadth First Search

Follow the path in breadth and only then descend into depth.



Order $a, b, d, e, c, f, g, h, i$

Adjacency List



$a \rightarrow b \rightarrow d \rightarrow e$

$b \rightarrow c$

$c \rightarrow f$

$d \rightarrow e$

$e \rightarrow b \rightarrow f$

f

$g \rightarrow h$

$h \rightarrow e \rightarrow i$

i

(Iterative) BFS-Visit(G, v)

```
**Input:** graph $G = (V, E)$

Queue $Q \leftarrow \emptyset$
enqueue($Q, v$)
$v$.visited $\leftarrow$ true
while $Q \neq \emptyset$ do
    $w \leftarrow$ dequeue($Q$)
    // visit $w$
    foreach $c \in N^+(w)$ do
        if $c$.visited $=$ false then
            $c$.visited $\leftarrow$ true
            enqueue($Q, c$)
```

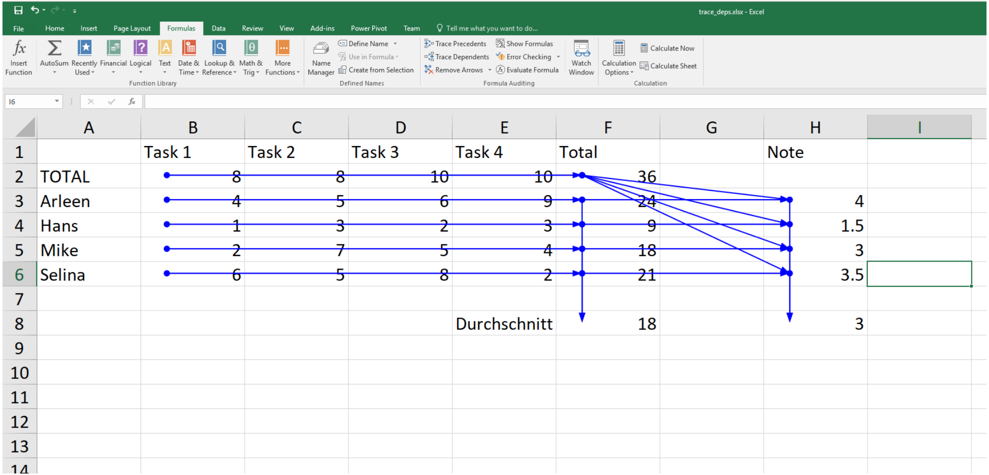
Algorithm requires extra space of $\mathcal{O}(|V|)$.

Main program BFS-Visit(G)

```
Input: graph $G = (V, E)$
foreach $v \in V$ do
    $v$.visited ← false
foreach $v \in V$ do
    if $v$.visited = false 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

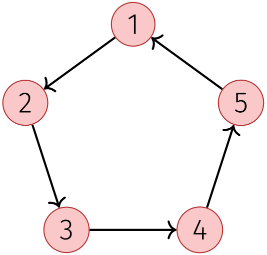
$$\mathrm{ord} : V \to \{1, \ldots, |V|\}$$

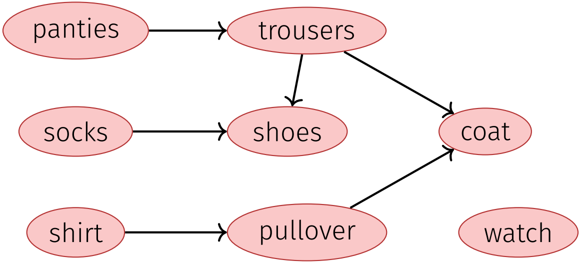
such that

$$\mathrm{ord}(v) < \mathrm{ord}(w) \ \forall \ (v, w) \in E.$$

Identify i with Element $v_i := \mathrm{ord}^1(i)$. Topological sorting $\hat{=} \langle v_1, \ldots, v_{|V|} \rangle$.

(Counter-)Examples





Cyclic graph: cannot be sorted topologically.

A possible toplogical sorting of the graph:
shirt, pullover, panties, watch, trousers, coat, socks, shoes

Observation

Theorem 21

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 $v_{i_1} < \cdots < v_{i_m} < v_{i_1}$.

Proof "$\Leftarrow$"

- 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 ($n \rightarrow n+1$):

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 $\mathrm{ord}(v_q) \leftarrow 1$.

Algorithm Topological-Sort(G)

Input: graph $G = (V, E)$.
Output: Topological sorting 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 22

Let $G = (V, E)$ be a directed acyclic graph. Algorithm **`TopologicalSort`**(G) *computes a topological sorting* ord *for* G *with run-time* $\Theta(|V| + |E|)$.

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 23

Let $G = (V, E)$ be a directed graph containing a cycle. Algorithm TopologicalSort 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 $i \leq V + 1 - k$.

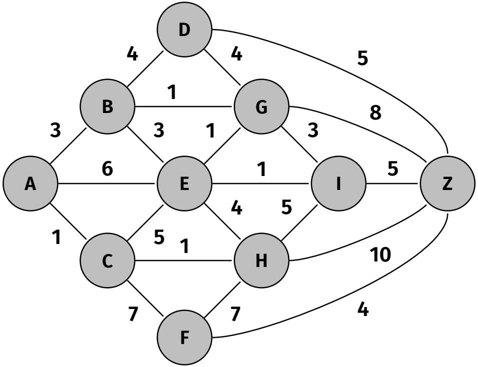
The runtime of the second part of the algorithm can become shorter. But the computation of the in-degree costs already $\Theta(|V| + |E|)$.

26. Shortest Paths

Motivation, Universal Algorithm, Dijkstra’s algorithm on distance graphs, Bellman-Ford Algorithm, Floyd-Warshall Algorithm, Johnson Algorithm [Ottman/Widmayer, Kap. 9.5 Cormen et al, Kap. 24.1-24.3, 25.2-25.3]

Route Finding

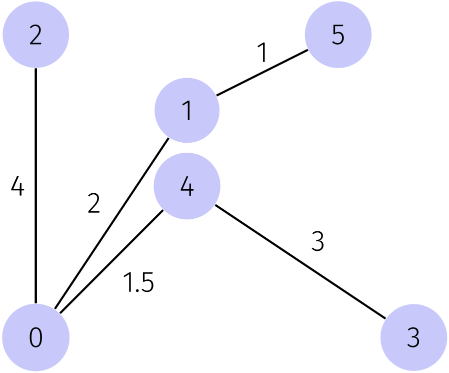
Provided cities A - Z and distances between cities



What is the shortest path from A to Z?

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.

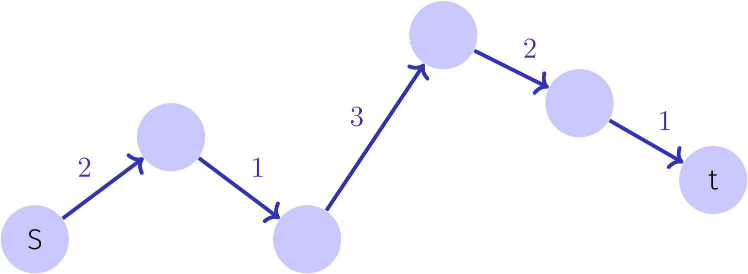


Weighted Paths

Given: $G = (V, E, c)$, $c : E \to \mathbb{R}$, $s, t \in V$.

Path: $p = \langle s = v_0, v_1, \ldots, v_k = t \rangle$, $(v_i, v_{i+1}) \in E$ $(0 \leq i < k)$

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

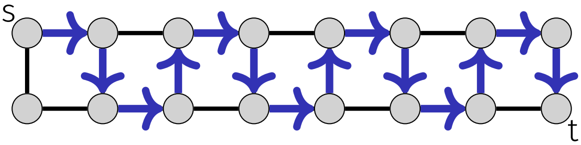
Wanted: $\delta(u, v)$ = minimal weight of a path from u to v:

$$\delta(u,v) = \begin{cases} \infty & \text{no path from } u \text{ to } v \\ \min\{c(p) : u \overset{p}{\rightsquigarrow} v\} & \text{otherwise} \end{cases}$$

In the following we call a path with minimal weight simply a **shortest path**.

Trivial algorithm?

Try out all paths?

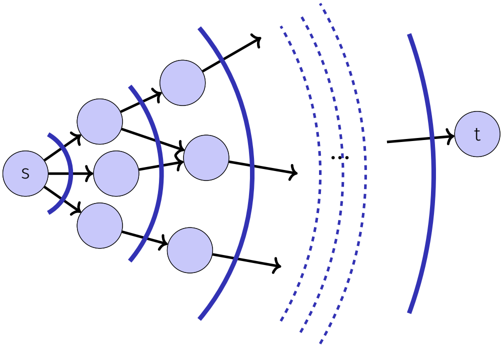


(at least $2^{|V|/2}$ paths from s to t)

$\Rightarrow$ Inefficient. There can be exponentially many paths.

Simplest Case

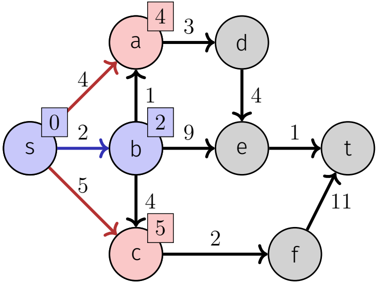
Constant edge weight (every edge has weight 1)

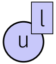


⇒ **Solution:** Breadth First Search $\mathcal{O}(|V| + |E|)$

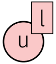
Dijkstra's Algorithm: Observation

important assumption: all weights are positive.





Shortest path $s \rightsquigarrow u$ has length l (exactly).

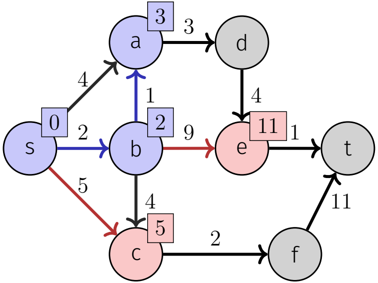


Upper bound:
Shortest path $s \rightsquigarrow u$ has length at most l.

Observation: Shortest outgoing edge (s, u) is the shortest path from s to this node u.

Dijkstra's Algorithm: Observation

important assumption: all weights are positive.



u l Shortest path $s \rightsquigarrow u$ has length l (exactly).

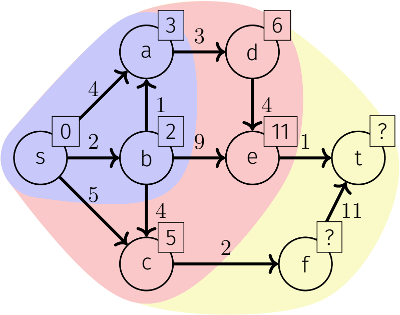
u l **Upper bound:**
Shortest path $s \rightsquigarrow u$ has length at most l.

General Observation: The smallest upper bound of a(n orange) node u constitutes the exact length of the shortest path from s to u.

Dijkstra's Algorithm: Basic Idea (Greedy)

V is split into:

- $\mathbf{K}$: nodes with known shortest path
- $\mathbf{N} = \bigcup_{v \in K} N^+(v) \setminus K$: successors of K
  $\Rightarrow$ an upper bound is known
- $\mathbf{R} = V \setminus (K \cup N)$: remaining nodes
  $\Rightarrow$ nothing is known yet



Greedy:

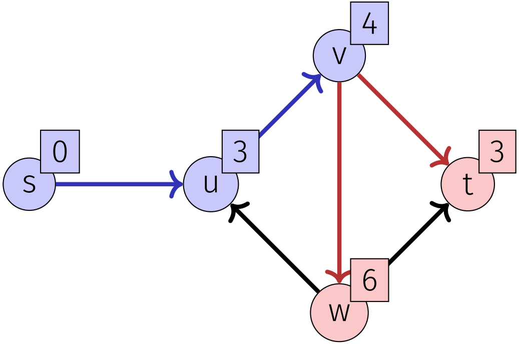
Starting with $\mathbf{N} = \{s\}$, until $\mathbf{N} = \emptyset$: node from $\mathbf{N}$ with smallest upper bound joins $\mathbf{K}$, and its neighbors join $\mathbf{N}$.

Invariants:

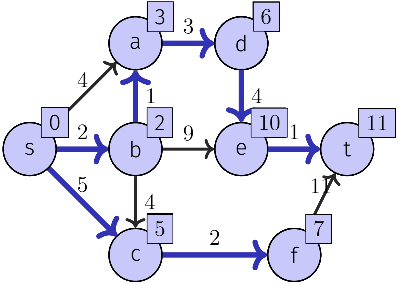
- after i steps: shortest paths to i nodes known ($|K| = i$).
- for all nodes in $\mathbf{v} \in N$: the upper bound is the (exact) length of shortest path $\mathbf{s} \rightsquigarrow \bullet \rightarrow v$ from $\mathbf{s}$ to $\mathbf{v}$ with nodes only from $\mathbf{K} \cup \{\mathbf{v}\}$.

Quiz

Is the following constellation of upper bounds possible?



Example



Known shortest paths from s:

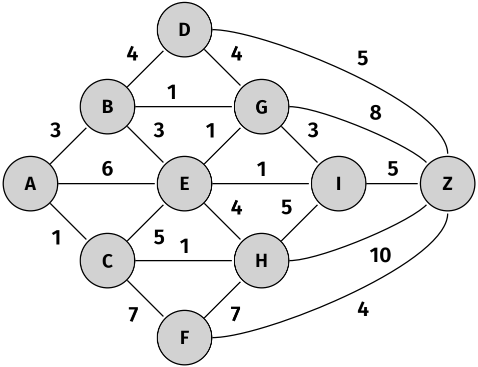
$s \rightsquigarrow s$: 0	$s \rightsquigarrow d$: 6
$s \rightsquigarrow b$: 2	$s \rightsquigarrow f$: 7
$s \rightsquigarrow a$: 3	$s \rightsquigarrow e$: 10
$s \rightsquigarrow c$: 5	$s \rightsquigarrow t$: 11

$\mathbf{K} = \{s, b, a, c, d, f, e, t\}$

$\mathbf{N} = \{\}$

$\mathbf{R} = \{\}$

Quiz



Which nodes are in K (known shortest paths) after six steps of Dijkstra's algorithm with starting node A?

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 $v \in V$

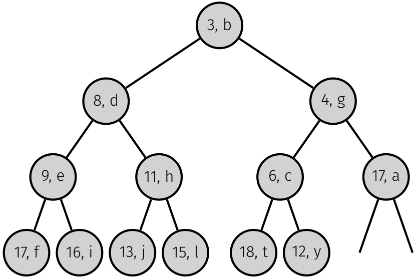
Algorithm: Dijkstra(G, s)

```
Input:  Positively weighted Graph $G = (V, E, c)$,
        starting point $s \in V$,
Output: Length $d_s$ of the shortest paths from $s$ and
        predecessor $\pi_s$ for each node

foreach $u \in V$ do
    $d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow$ null
$d_s[s] \leftarrow 0$; $N \leftarrow \{s\}$
while $N \neq \emptyset$ do
    $u \leftarrow \arg\min_{u \in N} d_s[u]$; $N \leftarrow N \setminus \{u\}$
    foreach $v \in N^+(u)$ do
        if $d_s[u] + c(u, v) < d_s[v]$ then
            $d_s[v] \leftarrow d_s[u] + c(u, v)$
            $\pi_s[v] \leftarrow u$
            $N \leftarrow N \cup \{v\}$
```

Implementation: Data Structure for N?

Required operations:

- **`Insert((p, k))`**: $\mathcal{O}(\log |V|)$
  add key (node) k
  with value (upper bound) p
- **`ExtractMin()`**: $\mathcal{O}(\log |V|)$
  remove element with smallest value
- **`DecreaseKey((p, k))`**: $\mathcal{O}(\log |V|)$
  update the value of key k to p

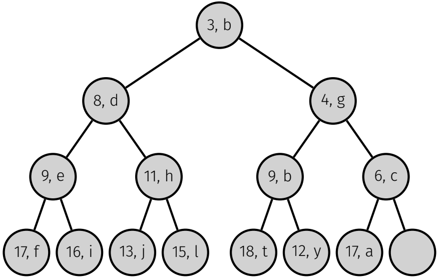


$\Rightarrow$ MinHeap with nodes from N as keys and with upper bounds as value

DecreaseKey

Two possibilities:

- tracking position: store at nodes or external
- or avoid DecreaseKey: with Lazy Deletion



Lazy Deletion:

- Re-insert node with smaller upper bound
- Mark nodes "deleted" once extracted

$\Rightarrow$ Memory consumption of heap can grow to $\Theta(|E|)$ instead of $\Theta(|V|)$

$\Rightarrow$ Because $|E| \leq |V|^2$: Insert and ExtractMin still in $\mathcal{O}(\log |V|^2) = \mathcal{O}(\log |V|)$

Algorithm: Dijkstra(G, s) with Lazy Deletion

```
Input: Positively weighted Graph $G = (V, E, c)$, starting point $s \in V$,
Output: Length $d_s$ of the shortest paths from $s$ and predecessor $\pi_s$ for each node
foreach $u \in V$ do
    $d_s[u] \leftarrow \infty$;  $\pi_s[u] \leftarrow$ null
$K = \{\}$; $d_s[s] \leftarrow 0$; $N \leftarrow \{s\}$
while $N \neq \emptyset$ do
    $d, u \leftarrow$ ExtractMin($N$)
    if $u \notin K$ then
        $K \leftarrow K \cup \{u\}$
        foreach $v \in N^+(u)$ do
            if $d + c(u, v) < d_s[v]$ then
                $d_s[v] \leftarrow d + c(u, v)$; $\pi_s[v] \leftarrow u$
                Insert($(d + c(u, v), v)$)
```

Running time:
Initialization: $\mathcal{O}(|V|)$
$(|V| + |E|)$ times ExtractMin: $\mathcal{O}((|V| + |E|) \cdot \log |V|)$;
$(|E| + 1)$ times Insert: $\mathcal{O}(|E| \cdot \log |V|)$;
⇒ Overall: $\mathcal{O}((|V| + |E|) \cdot \log |V|)$

Runtime of Dijkstra (without Lazy Deletion)

- $|V|\times$ ExtractMin: $\mathcal{O}(|V|\log|V|)$
- $|E|\times$ Insert or DecreaseKey: $\mathcal{O}(|E|\log|V|)$
- $1\times$ Init: $\mathcal{O}(|V|)$
- Overal[39] [40] :

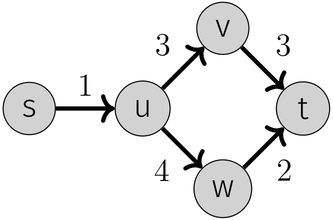
$$\mathcal{O}((|V|+|E|)\log|V|)$$

[39] For connected graphs: $\mathcal{O}(|E|\log|V|)$

[40] Can be improved when a data structure optimized for ExtractMin and DecreaseKey ist used (Fibonacci Heap), then runtime $\mathcal{O}(|E|+|V|\log|V|)$.

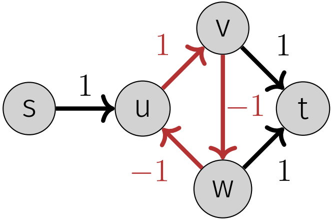
Observations

- Is the shortest path always unique? No!



Dijkstra's algorithm finds one (any) shortest path.

- Is there always at least one shortest path? No! Negative cycles.



26.3 General Algorithm

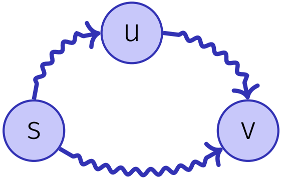
Why Dijkstra is correct and how to generalize.

Observations (1)

Triangle Inequality

For all $s, u, v \in V$:

$$\delta(s,v) \leq \delta(s,u) + \delta(u,v)$$

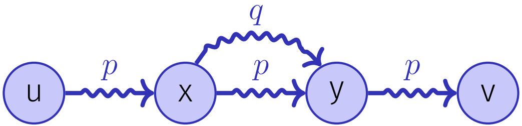


A shortest path from s to v cannot be longer than a shortest path from s to v that has to include u

Observations (2)

Optimal Substructure

Sub-paths of shortest paths are shortest paths. Let $p = \langle v_0, \dots, v_k \rangle$ be a shortest path from v_0 to v_k. Then each of the sub-paths $p_{ij} = \langle v_i, \dots, v_j \rangle$ $(0 \leq i < j \leq k)$ 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 (3)

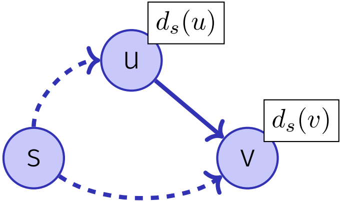
Shortest paths do not contain cycles

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).

General Algorithm

1. Initialise d_s and π_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 $(u, v) \in E$

```
Relax(u, v):
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
```



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:

$$\begin{aligned} \delta(s,v) &\leq \delta(s,u) + \delta(u,v) && \text{[Triangle Inequality].} \\ \delta(s,u) &\leq d_s[u] && \text{[Induction Hypothesis].} \\ \delta(u,v) &\leq c(u,v) && \text{[Minimality of } \delta\text{]} \\ \Rightarrow \quad d_s[u] + c(u,v) &\geq \delta(s,v) \end{aligned}$$

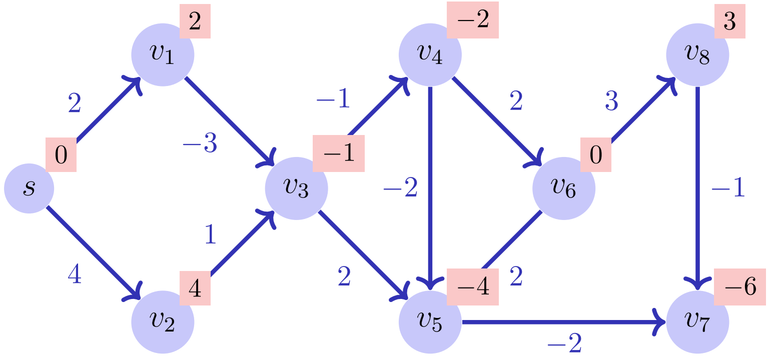
$$\Rightarrow \min\{d_s[v], d_s[u] + c(u,v)\} \geq \delta(s,v)$$

Central Question

How / in which order should edges be chosen in above algorithm?

Special Case: Directed Acyclic Graph (DAG)

DAG ⇒ topological sorting returns optimal visiting order



Top. Sort: ⇒ Order $s, v_1, v_2, v_3, v_4, v_6, v_5, v_8, v_7$.

Other Cases

- Special case: $c \equiv= 1 \Rightarrow$ BFS
- Special Case: Positive Edge Weights $\Rightarrow$ Dijkstra 🙂.
- General Weighted Graphs: cycles with negative weights can shorten the path, a shortest path is not guaranteed to exist.

Dynamic Programming Approach (Bellman)

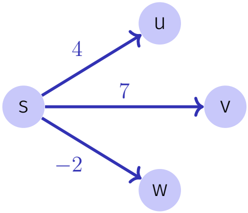
Induction over number of edges $d_s[i, v]$: Shortest path from s to v via maximally i edges.

$$d_s[i, v] = \min\{d_s[i-1, v], \min_{(u,v)\in E}(d_s[i-1, u] + c(u, v))$$

$$d_s[0, s] = 0, d_s[0, v] = \infty \ \forall v \neq s.$$

Dynamic Programming Approach (Bellman)

	s	$\cdots$	v	$\cdots$	w
0	0	∞	∞	∞	∞
1	0	∞	7	∞	-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)

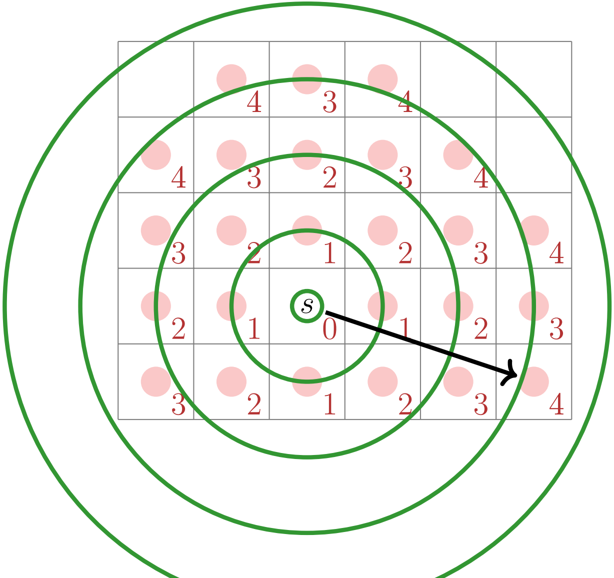
Input: Graph $G = (V, E, c)$, starting point $s \in V$
Output: If return value true, minimal weights d for all shortest paths from s, otherwise no shortest path.

```
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;
```

Runtime $\mathcal{O}(|E| \cdot |V|)$.

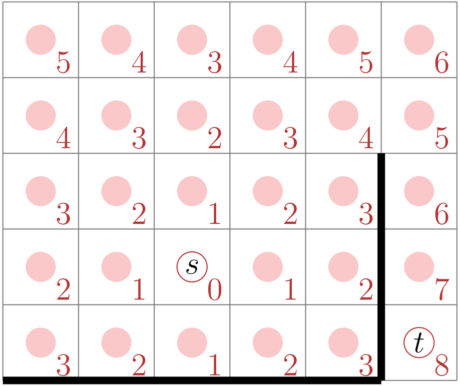
26.5 A*-Algorithm

Motivation A*



- Dijkstra Algorithm searches for all shortest paths, in all directions.

Motivation A*

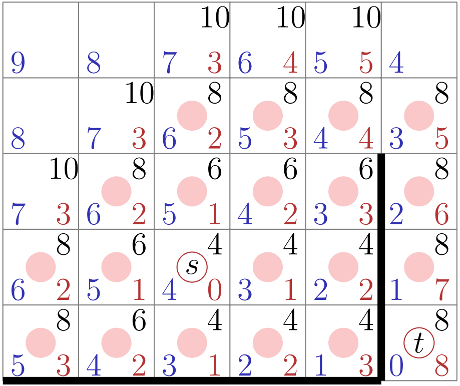


5	4	3	4	5	6
4	3	2	3	4	5
3	2	1	2	3	6
2	1	s 0	1	2	7
3	2	1	2	3	t 8

- Dijkstra Algorithm searches for all shortest paths, in all directions.
- which is correct, because the algorithm does not know about the graph's structure.

A* in Action

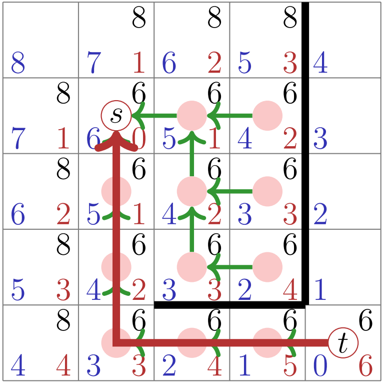
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ ($\hat{h} = \delta_x + \delta_y$ Manhattan-Distance)



- Idea: equip algorithm with a preferred direction by ways of a distance heuristic $\hat{h}$
- The value of this heuristics needs to underestimate the distance to t and is added to the found distance d_s to s

Keep backward path

$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x + \delta_y)$



 8	8 7 1	8 6 2	8 5 3	 4
8 7 1	6 s 6 0	6 5 1	6 4 2	 3
8 6 2	6 5 1	6 4 2	6 3 3	 2
8 5 3	6 4 2	6 3 3	6 2 4	 1
8 4 4	6 3 3	6 2 4	6 1 5	6 t 0 6

- The algorithm works like the Dijkstra-algorithm
- For finding the next candidate of R instead of the value d_s the value of $\hat{f} = \hat{h} + d_s$ is used

A*-Algorithm

Prerequisites

- Positively weighted, finite graph $G = (V, E, c)$
- $s \in V$, $t \in V$
- Distance estimate $\widehat{h}_t(v) \leq h_t(v) := \delta(v, t)\ \forall\ v \in V$.
- Wanted: shortest path $p : s \rightsquigarrow t$

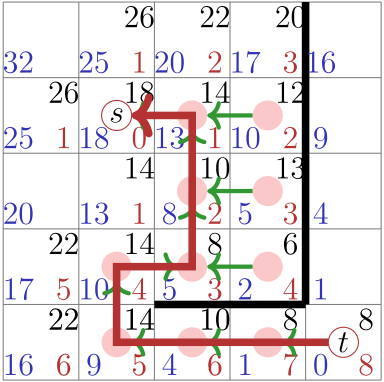
A*-Algorithm($G, s, t, \hat{h}$)

Input: Positively weighted Graph $G = (V, E, c)$, starting point $s \in V$, end point $t \in V$, estimate $\widehat{h}(v) \leq \delta(v, t)$
Output: Existence and value of a shortest path from s to t

```
foreach $u \in V$ do
    $d[u] \leftarrow \infty$; $\widehat{f}[u] \leftarrow \infty$; $\pi[u] \leftarrow$ null
$d[s] \leftarrow 0$; $\widehat{f}[s] \leftarrow \widehat{h}(s)$; $N \leftarrow \{s\}$; $K \leftarrow \{\}$
while $N \neq \emptyset$ do
    $u \leftarrow$ ExtractMin$_{\widehat{f}}(N)$; $K \leftarrow K \cup \{u\}$
    if $u = t$ then return success
    foreach $v \in N^+(u)$ with $d[v] > d[u] + c(u, v)$ do
        $d[v] \leftarrow d[u] + c(u, v)$; $\widehat{f}[v] \leftarrow d[v] + \widehat{h}(v)$; $\pi[v] \leftarrow u$
        $N \leftarrow N \cup \{v\}$; $K \leftarrow K - \{v\}$
return failure
```

What if $\hat{h}$ does not underestimate

$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- Algorithm can terminate with the wrong result when $\hat{h}$ does not under-estimate the distance to t.
- although the heuristics looks reasonable otherwise (it is monotonic, for instance)

Revisiting nodes

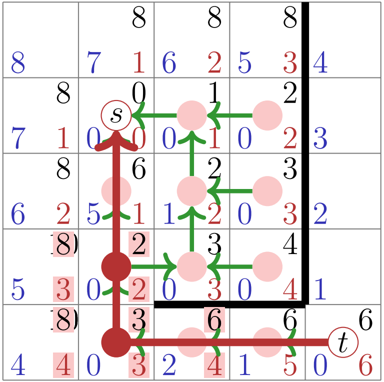
- The A*-algorithm can re-insert nodes that had been extracted from R before.
- This can lead to suboptimal behavior (w.r.t. running time of the algorithm).
- If $\widehat{h}$, in addition to being admissible ($\widehat{h}(v) \leq h(v)$ for all $v \in V$), fulfils monotonicity, i.e. if for all $(u, u') \in E$:

  $$\widehat{h}(u') \leq \widehat{h}(u) + c(u', u)$$

  then the A*-Algorithm is equivalent to the Dijsktra-algorithm with edge weights $\tilde{c}(u, v) = c(u, v) + \widehat{h}(u) - \widehat{h}(v)$, and no node is re-inserted into R.
- It is not always possible to find monotone heuristics.

A crazy $\hat{h}$

$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithm terminates correctly even if the distance heuristic is not monotonic
- It is then possible that nodes are removed and re-inserted into R multiple times.

Conclusion

- The A*-Algorithm is an extension of the Dijkstra algortihm by a distance heuristic $\hat{h}$.
- A* = Dijkstra if $\hat{h} \equiv 0$
- If $\hat{h}$ underestimates the real distance, the algorithm works correctly.
- If $\hat{h}$ is monotone in addition, then the algorithm works efficiently.
- In practical applications (e.g. routing), the choice of $\hat{h}$ is often intuitive and leads to a significant improvement over Dijkstra.

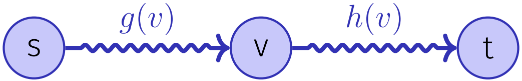
26.6 A*-Algorithm

Proof of correctness Not relevant for the exam

Notation

Let $f(v)$ be the distance of a shortest path from s to t via v, thus

$$f(v) := \underbrace{\delta(s,v)}_{g(v)} + \underbrace{\delta(v,t)}_{h(v)}$$



let p be a shortest path from s to t.
It holds that $f(s) = \delta(s,t)$ and $f(v) = f(s)$ for all $v \in p$.
Let $\widehat{g}(v) := d[v]$ be an estimate of $g(v)$ in the algorithm above. It holds that $\hat{g}(v) \geq g(v)$.
$\widehat{h}(v)$ is an estimate of $h(v)$ with $\widehat{h}(v) \leq h(v)$.

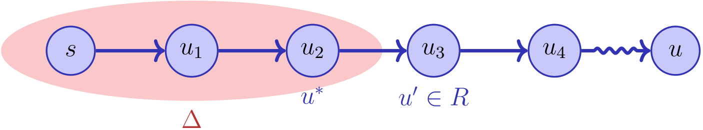
Why the Algorithm Works

Lemma 24

Let $u \in V$ and, at a time during the execution of the algorithm, $u \notin M$. Let p be a shortest path from s to u. Then there is a $u' \in p$ with $\widehat{g}(u') = g(u')$ and $u' \in R$.

The lemma states that there is always a node in the open set R with the minimal distance from s already computed and that belongs to a shortest path (if existing).

Illustration and Proof



Proof: If $s \in R$, then $\widehat{g}(s) = g(s) = 0$. Therefore, let $s \notin R$.

Let $p = \langle s = u_0, u_1, \ldots, u_k = u \rangle$ and $\Delta = \{u_i \in p, u_i \in M, \widehat{g}(u_i) = g(u_i)\}$.
$\Delta \neq \emptyset$, because $s \in \Delta$.

Let $m = \max\{i : u_i \in \Delta\}$, $u^* = u_m$. Then $u^* \neq u$, since $u \notin M$. Let $u' = u_{m+1}$.

1. $\widehat{g}(u') \leq \widehat{g}(u^*) + c(u^*, u')$ because u' has already been relaxed
2. $\widehat{g}(u^*) = g(u^*)$ (because $u^* \in \Delta$)
3. $\hat{g}(u') \geq g(u')$ (construction of $\hat{g}$)
4. $g(u') = g(u^*) + c(u^*, u')$ (because p optimal)

Therefore: $\widehat{g}(u') = g(u')$ and thus also $u' \in R$ because $u' \notin \Delta$. ■

Corollary

Corollary 25

If $\widehat{h}(u) \leq h(u)$ for all $u \in V$ and A- Algorithmus has not yet terminated. The for each shortest path p from s t t there is some node $u' \in p$ with $\hat{f}(u') \leq \delta(s,t) = f(t)$.*

If there is a shortest path p from s to t, then there is always a node in the open set R that underestimates the overal distance and that is on the shortest path.

Proof of the Corollary

Proof:
From the lemma: $\exists u' \in p$ with $\widehat{g}(u') = g(u')$.
Therefore:

$$\begin{aligned} \widehat{f}(u') &= \widehat{g}(u') + \widehat{h}(u') \\ &= g(u') + \widehat{h}(u') \\ &\leq g(u') + h(u') = f(u') \end{aligned}$$

Because p is shortest path: $f(u') = \delta(s,t)$. ■

Admissibility

Theorem 26

If there is a shortest path from s to t and $\hat{h}(u) \leq h(u) \; \forall \; u \in V$ then A terminates with $\hat{g}(t) = \delta(s,t)$*

Proof: If the algorithm terminates, then it termines with t with $f(t) = \widehat{g}(t) + 0 = g(t)$. That is because $\widehat{g}$ overestimates g at most and by the corollary above that algorithm always finds an element $v \in R$ with $f(v) \leq \delta(s,t)$.

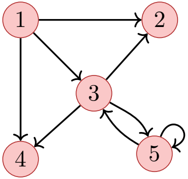
The algorithm terminates in finitely many steps. For finite graphs the maximal number of relaxing steps is bounded.[41] ■

[41] For a δ-graph the maximum number of relaxing steps before R contains only nodes with $\hat{f}(s) > \delta(s,t)$ is limited as well. The exact argument can be found in the seminal article Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths".

27. Transitive Closure, All Pairs Shortest Paths

Reflexive transitive closure [Ottman/Widmayer, Kap. 9.2 Cormen et al, Kap. 25.2] Floyd-Warshall Algorithm [Ottman/Widmayer, Kap. 9.5.3 Cormen et al, Kap. 25.2]

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

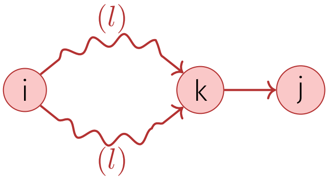
Theorem 27

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 $k = 1$. $a_{i,j} = a_{i,j}^{(1)}$.
Hypothesis: claim is true for all $k \leq l$
Step ($l \rightarrow l + 1$):



$$a_{i,j}^{(l+1)} = \sum_{k=1}^{n} a_{i,k}^{(l)} \cdot a_{k,j}$$

$a_{k,j} = 1$ 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 $l + 1$.

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

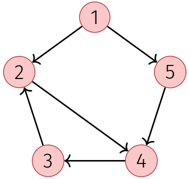
Graph $G = (V, E)$
adjacencies $A_G \mathrel{\widehat{=}}$ Relation $E \subseteq V \times V$ over V

- **reflexive** $\Leftrightarrow a_{i,i} = 1$ for all $i = 1, \dots, n$. (loops)
- **symmetric** $\Leftrightarrow a_{i,j} = a_{j,i}$ for all $i, j = 1, \dots, n$ (undirected)
- **transitive** $\Leftrightarrow$ $(u, v) \in E$, $(v, w) \in E \Rightarrow (u, w) \in E$. (reachability)

Reflexive Transitive Closure

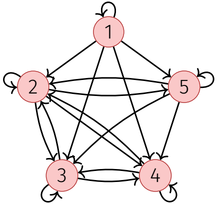
Reflexive transitive closure of G $\Leftrightarrow$ **Reachability relation** E^*: $(v, w) \in E^*$ iff ∃ path from node v to w.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$



$G = (V, E)$

$\Rightarrow$



$G^* = (V, E^*)$

Algorithm $A \cdot A$

Input: (Adjacency-)Matrix $A = (a_{ij})_{i,j=1\dots n}$
Output: Matrix Product $B = (b_{ij})_{i,j=1\dots n} = A \cdot A$

```
$B \leftarrow 0$
for $r \leftarrow 1$ to $n$ do
    for $c \leftarrow 1$ to $n$ do
        for $k \leftarrow 1$ to $n$ do
            $b_{rc} \leftarrow b_{rc} + a_{rk} \cdot a_{kc}$        // Number of Paths
return $B$
```

Counts number of paths of length 2

Algorithm $A \otimes A$

Input: Adjacency-Matrix $A = (a_{ij})_{i,j=1\dots n}$
Output: Modified Matrix Product $B = (b_{ij})_{i,j=1\dots n} = A \otimes A$

```
$B \leftarrow A$                                              // Keep paths
for $r \leftarrow 1$ to $n$ do
    for $c \leftarrow 1$ to $n$ do
        for $k \leftarrow 1$ to $n$ do
            $b_{rc} \leftarrow \max\{b_{rc}, a_{rk} \cdot a_{kc}\}$    // Path: yes/no
return $B$
```

Computes which paths of length 1 and 2 exist

Computation of the Reflexive Transitive Closure

Goal: computation of $B = (b_{ij})_{1 \leq i,j \leq n}$ with $b_{ij} = 1 \Leftrightarrow (v_i, v_j) \in E^*$ First idea:

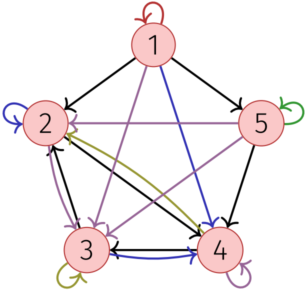
- Start with $B \leftarrow A$ and set $b_{ii} = 1$ for each i (Reflexivity.).
- Compute

$$B_n = \bigotimes_{i=1}^{n} B$$

with powers of 2 $B_2 := B \otimes B$, $B_4 := B_2 \otimes B_2$, $B_8 = B_4 \otimes B_4$ ...
$\Rightarrow$ 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 $A_G = (a_{ij})_{i,j=1\ldots n}$
Output: Reflexive transitive closure $B = (b_{ij})_{i,j=1\ldots n}$ of G

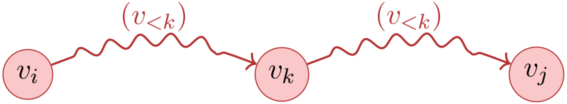
```
$B \leftarrow A_G$
for $k \leftarrow 1$ to $n$ do
    $b_{kk} \leftarrow 1$                                                    // Reflexivity
    for $r \leftarrow 1$ to $n$ do
        for $c \leftarrow 1$ to $n$ do
            $b_{rc} \leftarrow \max\{b_{rc}, b_{rk} \cdot b_{kc}\}$            // 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: $b_{ij} \leftarrow 1$.



All shortest Paths

Compute the weight of a shortest path for each pair of nodes.

- $|V|\times$ Application of Dijkstra's Shortest Path algorithm $\mathcal{O}(|V|\cdot(|E|+|V|)\cdot\log|V|)$ (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

Consider weights of all shortest paths S^k with intermediate nodes in[42] $V^k := \{v_1, \dots, v_k\}$, 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}.

[42] like for the algorithm of the reflexive transitive closure of Warshall

Induction via node number

$d^k(u,v)$ = Minimal weight of a path $u \rightsquigarrow v$ with intermediate nodes in V^k

Induktion

$$\begin{aligned} d^k(u,v) &= \min\{d^{k-1}(u,v), d^{k-1}(u,k) + d^{k-1}(k,v)\} (k \geq 1) \\ d^0(u,v) &= c(u,v) \end{aligned}$$

Algorithm Floyd-Warshall(G)

```
Input: Graph $G = (V, E, c)$ without negative weight cycles.
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
            $d^k(v_i, v_j) = \min\{d^{k-1}(v_i, v_j), d^{k-1}(v_i, v_k) + d^{k-1}(v_k, v_j)\}$
```

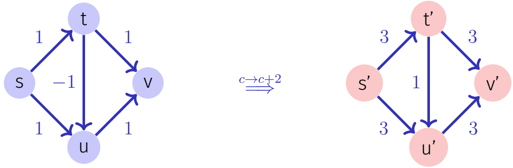
Runtime: $\Theta(|V|^3)$

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: $\tilde{c}(v_0, \dots, v_k = v_0) = c(v_0, \dots, v_k = v_0)$

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: $\mathcal{O}(|V|^2 \log |V| + |V| \cdot |E|)$)

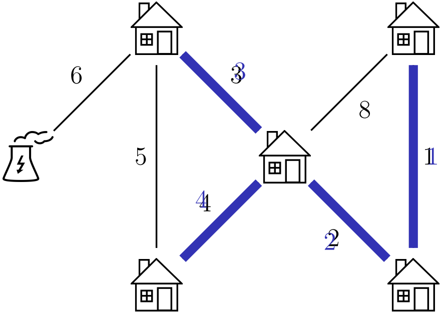
Overal $\mathcal{O}(|V| \cdot |E| \cdot \log |V|)$
($\mathcal{O}(|V|^2 \log |V| + |V| \cdot |E|)$)

28. Minimum Spanning Trees

Motivation, Greedy, Algorithm Kruskal, General Rules, ADT Union-Find, Algorithm Jarnik, Prim, Dijkstra , Fibonacci Heaps [Ottman/Widmayer, Kap. 9.6, 6.2, 6.1, Cormen et al, Kap. 23, 19]

Cheapest Electricity Grid

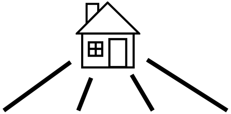
Given: Houses and costs to connect the houses with electricity.



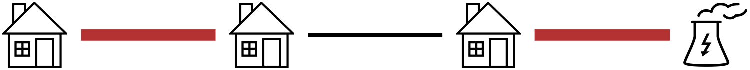
Wanted: Cheapest electricity grid that reaches every house.

Requirements for the power grid

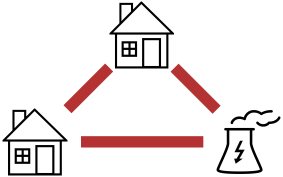
- Every house must have at least one power line.



- The power grid needs to be connected (just one grid).

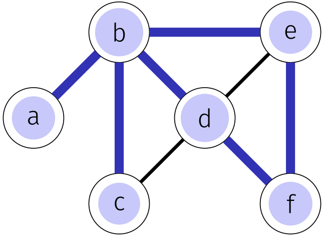


- The power grid should not have cycles.



Spanning Tree

Given: undirected, connected graph $G = (V, E)$



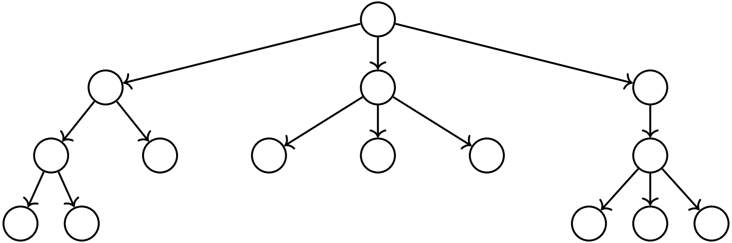
Spanning Tree of G: Subgraph $T = (V', E')$ with $V' \subseteq V, E' \subseteq E$ such that

- Spanning: $V' = V$ (spans all nodes)
- Tree: connected and cycle-free

$\Rightarrow$ for each pair of nodes: exactly one connecting path

$\Rightarrow$ spanning tree has exactly $|V| - 1$ edges $(|E'| = |V| - 1)$

Trees

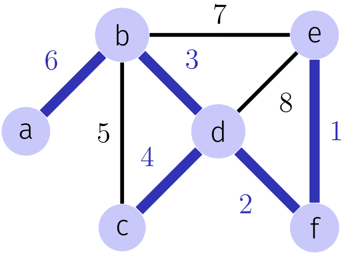


Up to this point trees were directed trees!

- connected
- cycle-free
- directed from parents to children

Minimum Spanning Tree (MST)

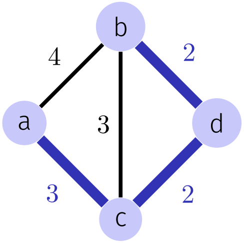
Given: undirected, weighted, connected graph $G = (V, E, c)$ with edge weights $c\colon E \to \mathbb{R}$



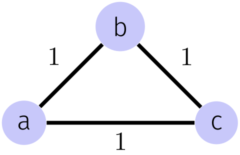
Wanted: Spanning tree $T = (V, E')$ of G with minimum weight $\sum_{e \in E'} c(e)$

Observations

- Is that the same as shortest paths? No!

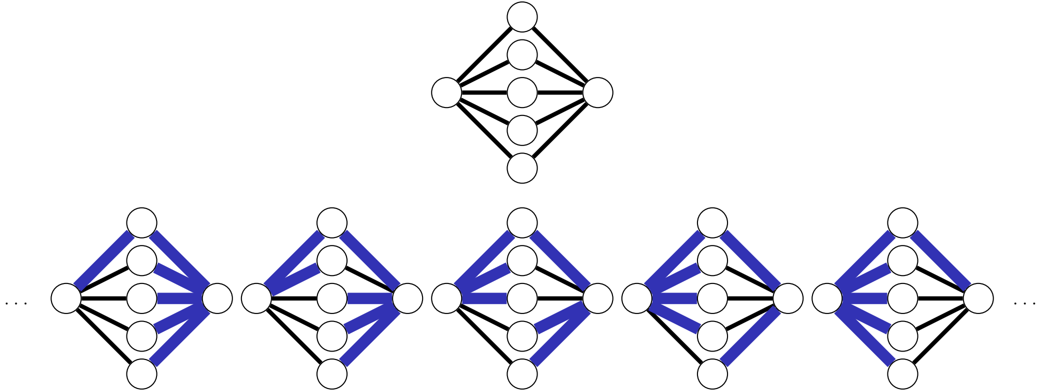


- Is the minimum spanning tree unique? Not always.



Trivial brute force algorithm?

Try out all spanning trees?

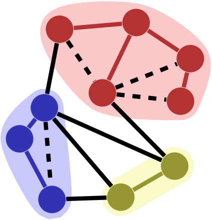


⇒ Inefficient: There are graphs with exponentially many spanning trees.

28.2 Algorithm of Kruskal

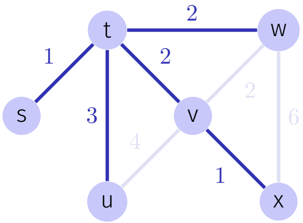
Kruskal’s Algorithm

Idea: add lightest edge if it does not lead to a cycle
Invariant: After i steps, i edges of the MST and the corresponding components are known



Beispiel

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 $(V, A, c)$
```

(Corrrectness proof in handout.)

[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

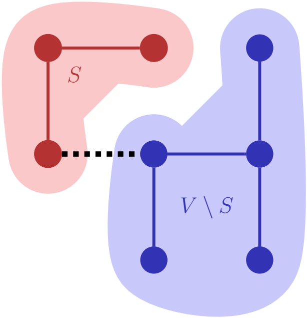
Notation (snapshot of the state in the running algorithm)

- 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 28

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.[43] 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 $c(e) = c(e')$).

[43] 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.

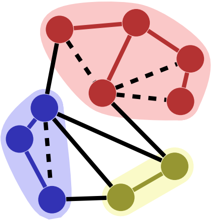
Choose a cycle without rejected edges. Of all undecided edges of the cycle, reject an edge e with maximal weight.

- 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 $i \equiv V_i \subset V$.

To identify 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.

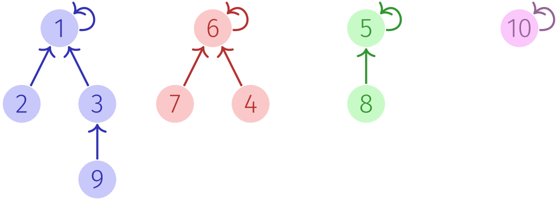
Union-Find Algorithm MST-Kruskal(G)

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 $(V, A, c)$
```

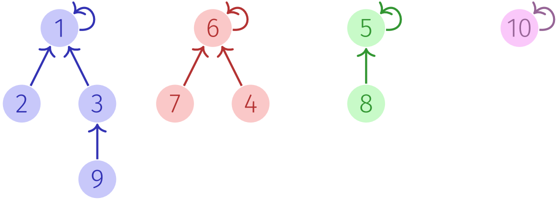
Implementation Union-Find

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	6	5	3	10

Implementation Union-Find

Index	**1**	2	3	4	**5**	**6**	7	8	9	**10**
Parent	1	1	1	6	5	6	6	5	3	10

Make-Set(i)	$p[i] \leftarrow i$; **return** i

Find(i)	**while** $(p[i] \neq i)$ **do** $i \leftarrow p[i]$
	return i

Union(i, j) [44]	$p[j] \leftarrow i$;

[44] 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-ase running time for Find) in $\Theta(\log n)$

[Observation]

Theorem 29

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: $h_2 \leq h_1$

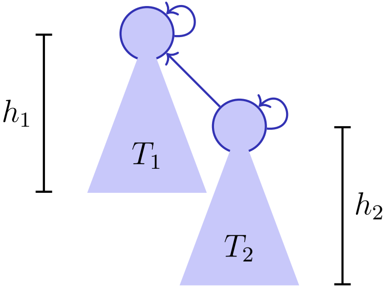
- $h_2 < h_1$:

$$h(T_1 \oplus T_2) = h_1 \Rightarrow g(T_1 \oplus T_2) \geq 2^h$$

- $h_2 = h_1$:

$$g(T_1) \geq g(T_2) \geq 2^{h_2}$$

$$\Rightarrow g(T_1 \oplus T_2) = g(T_1) + g(T_2) \geq 2 \cdot 2^{h_2} = 2^{h(T_1 \oplus T_2)}$$



Alterantive 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).[45]

[45] When combined with union by size, we do not go into any details here. Cf. Cormen et al, Kap. 21.4

Running time of Kruskal's Algorithm

- Sorting of the edges: $\Theta(|E| \log |E|) = \Theta(|E| \log |V|)$. [46]
- 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 $\Theta(|E| \log |V|)$.

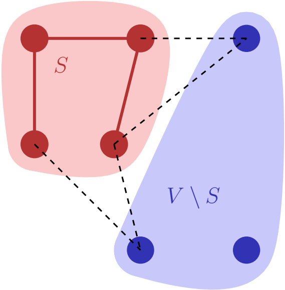
[46] because G is connected: $|V| \leq |E| \leq |V|^2$

28.5 Algorithm Jarnik, Prim, Dijkstra

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.

Implementation and Running time

Implementation like with Dijkstra's ShortestPath. Only difference:

Shortest Paths

```
Relax $(u, v)$:
    if $d_s[v] > d[u] + c(u, v)$ then
        $d_s[v] \leftarrow d_s[u] + c(u, v)$
        $\pi_s[v] \leftarrow u$
```

$\Rightarrow$

Minimum Spanning Tree

```
Relax $(u, v)$:
    if $d_s[v] > c(u, v)$ then
        $d_s[v] \leftarrow c(u, v)$
        $\pi_s[v] \leftarrow u$
```

- With Min-Heap: costs $\mathcal{O}(|E| \cdot \log |V|)$:
  - Initialization (node coloring) $\mathcal{O}(|V|)$
  - $|V| \times$ ExtractMin = $\mathcal{O}(|V| \log |V|)$,
  - $|E| \times$ Insert or DecreaseKey: $\mathcal{O}(|E| \log |V|)$,
- With a Fibonacci-Heap: $\mathcal{O}(|E| + |V| \cdot \log |V|)$.

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.

28.7 Fibonacci Heaps

Fibonacci Heaps

Data structure for elements with key with operations

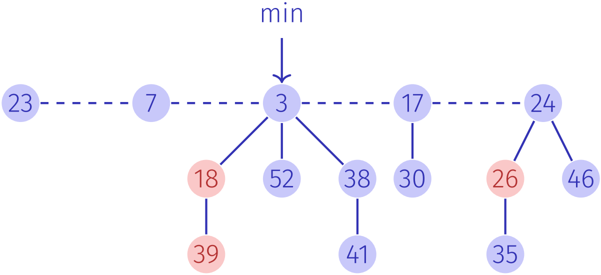
- MakeHeap(): Return new heap without elements
- Insert(H, x): Add x to H
- Minimum(H): return a pointer to element m with minimal key
- ExtractMin(H): return and remove (from H) pointer to the element m
- Union(H_1, H_2): return a heap merged from H_1 and H_2
- DecreaseKey(H, x, k): decrease the key 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	$\Theta(\log n)$	$\Theta(\log n)$

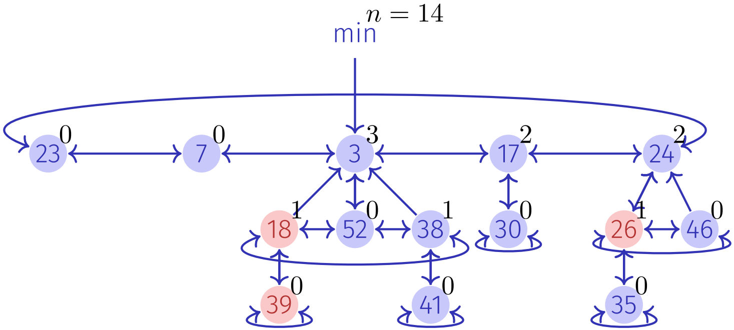
Structure

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 $p \leftarrow pp$.

A sketch of the amoritized analysis is in the handout.

[Estimation of the degree]

Theorem 30

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 ith child node has a degree of at least $i-2$.

Proof: p may have had more children and lost by cutting. When the ith 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 31

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:

$$\mathcal{O}(c) + (t(H) + c) + 2 \cdot (m(H) - c + 2)) - (t(H) + 2m(H)) = \mathcal{O}(1)$$

29. Flow in Networks

Flow Network, Flow, Maximum Flow
Residual Capacity, Remainder Network, Augmenting path

Ford-Fulkerson Algorithm
Edmonds-Karp Algorithm

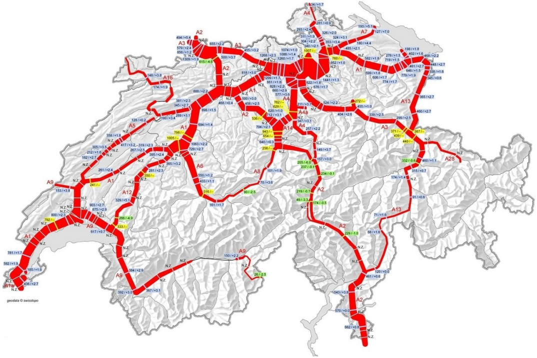
Cuts, Max-Flow Min-Cut Theorem

[Ottman/Widmayer, Kap. 9.7, 9.8.1], [Cormen et al, Kap. 26.1-26.3]

Slides redesigned by Manuela Fischer – thank you!

Maximum Traffic Flow

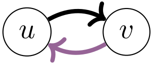
Given: Road Network with capacities

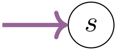


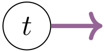
Wanted: Maximum traffic flow between Zurich and Geneva

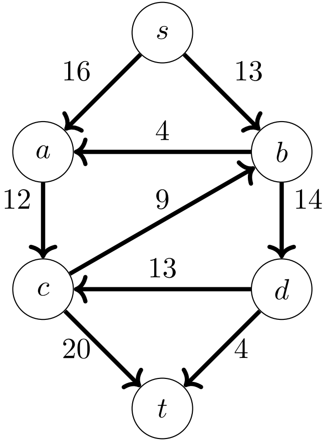
Flow Network

directed, weighted graph $G = (V, E, c)$ with capacities $c\colon E \to \mathbb{R}^{>0}$

- without antiparallel edges:
  $(u, v) \in E \;\Rightarrow\; (v, u) \notin E$
  

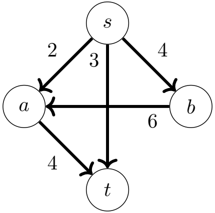
- source $s \in V$ without ingoing edges:
  $\forall v \in V : (v, s) \notin E$
  

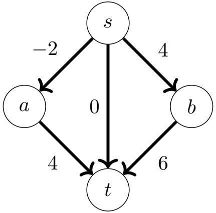
- sink $t \in V$ without outgoing edges:
  $\forall v \in V : (t, v) \notin E$
  

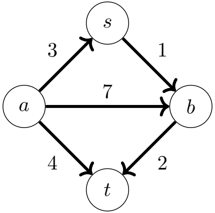


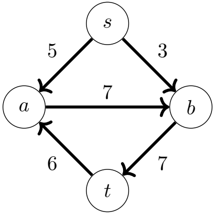
Quiz Flow Network

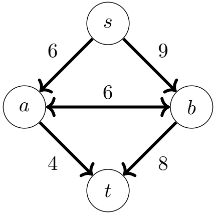
Which of the following graphs are flow networks?

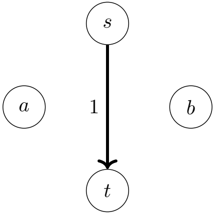










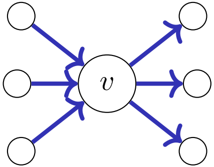


Flow in Flow Network

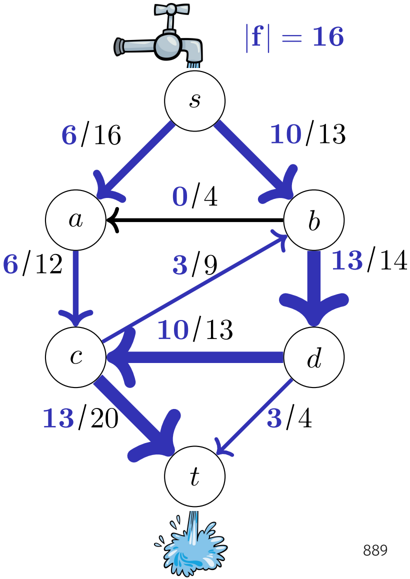
Flow is function $f\colon E \to \mathbb{R}^{\geq 0}$ such that

- **Bounded Capacity**: $\forall e \in E\colon\ f(e) \leq c(e)$
- **Conservation of flow**: $\forall v \in V \setminus \{s,t\}\colon$

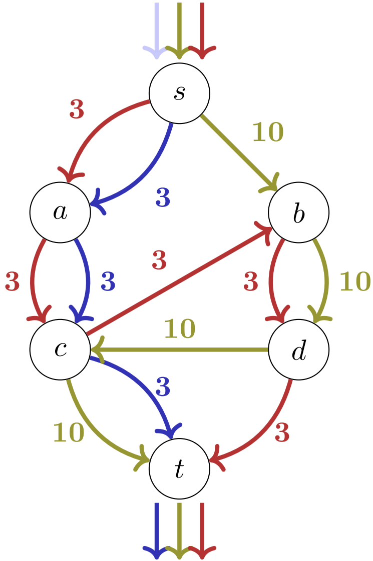
$$\underbrace{\sum_{e \in E^-(v)} f(e)}_{=:f^-(v)} = \underbrace{\sum_{e \in E^+(v)} f(e)}_{=:f^+(v)}$$

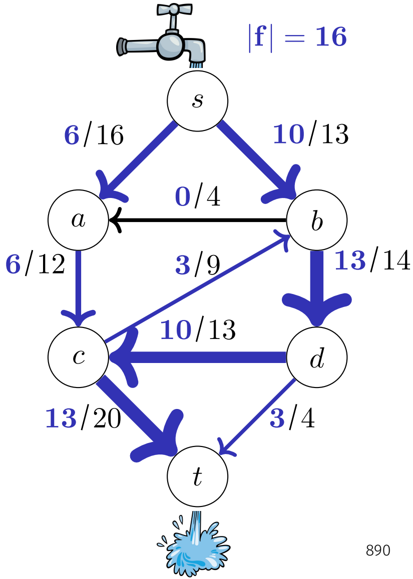


Size of flow: $|f| := f^+(s) = f^-(t)$



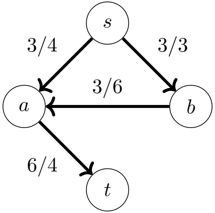
Intuition: Flow as set of paths $s \rightsquigarrow t$

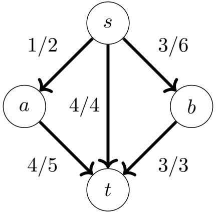


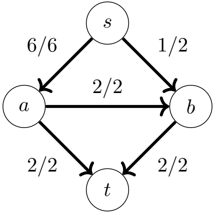


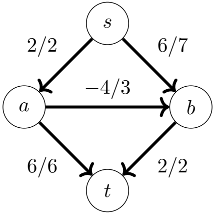
Quiz Flow

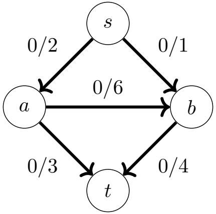
Which of the following are flows?

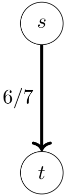






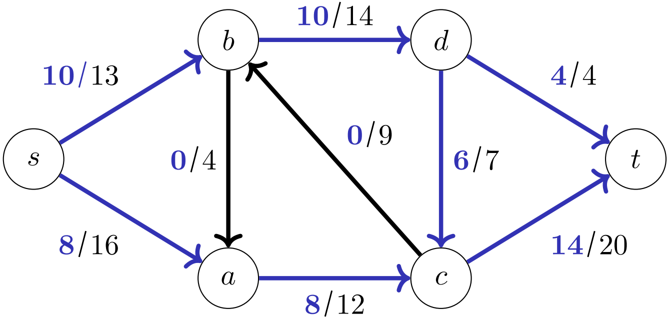






Maximal Flow

Given: Flow network: $G = (V, E, c)$, directed, positively weighted, without antiparallel edges, with source s and sink t

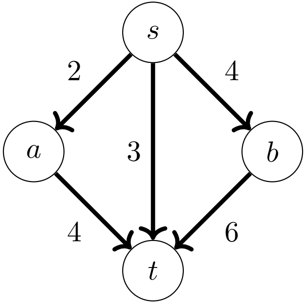


$\mathbf{18} = |\mathbf{f}| \leq |\mathbf{f}_{\max}| = \mathbf{23}$

Wanted: Size $|f_{\max}|$ of the maximum flow in G

Quiz Maximum Flow

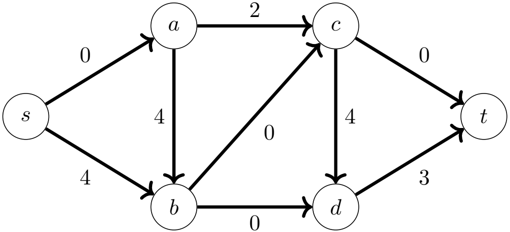
What is the maximum flow in the following flow network?



Greedy Algorithm?

Residual capacity of an edge e**:** $r(e) := c(e) - f(e)$
Residual capacity of a path P**:** $\min_{e \in P} r(e)$

Greedy: Starting with $f(e) = 0$ for all $e \in E$, as long as there exists a path $s \rightsquigarrow t$ with remaining capacity $d > 0$, increase flow along this path by d.



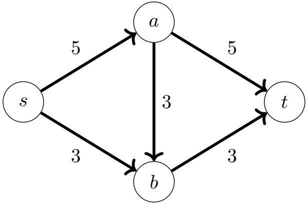
$G_f^+ := (V, E, r := c - f)$

$|f| = 8$

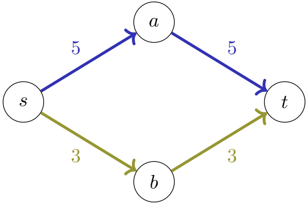
$s \rightarrow a \rightarrow b \rightarrow d \rightarrow t$: 3
$s \rightarrow a \rightarrow c \rightarrow t$: 2
$s \rightarrow b \rightarrow c \rightarrow t$: 3

but $|f_{\max}| = 10$

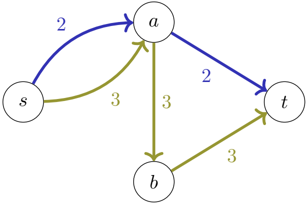
Problem with Greedy



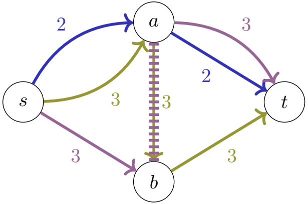
$G = (V, E, c)$



$|f_{\max}| = 8$



Greedy: $|f| = 5$



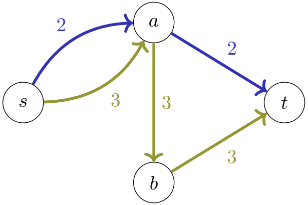
Redirection

29.1 Flow Algorithms

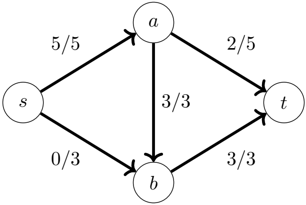
Ford-Fulkerson Algorithm

Edmonds-Karp Algorithm

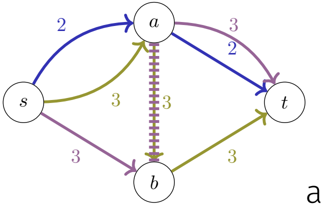
Redirection using flow decrement



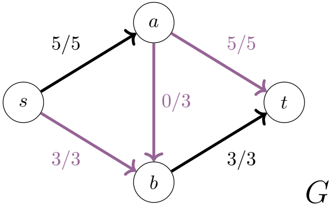
before



$G = (V, E, f/c)$



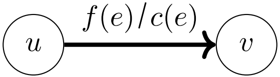
after



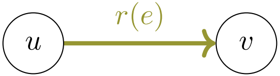
$G = (V, E, f'/c)$

⇒ Umleitung entspricht Verringerung des Flusses durch Kante

Idea: Flow increments and decrements



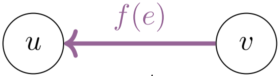
- **Increment:**
  flow through e can be increased by at most $r(e) := c(e) - f(e)$



$G_f^+ := (V, E, r := c - f)$

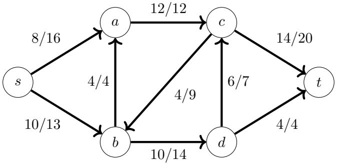
- **Decrement:**
  flow through e can be decreased by at most $f(e)$

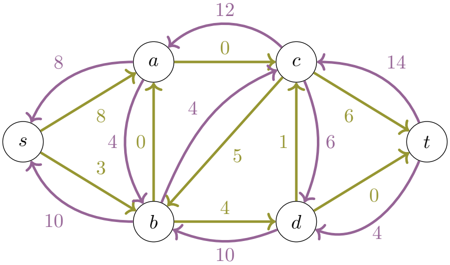
  $\Rightarrow$ flow through $\overleftarrow{e}$ can be increased by at most $f(e)$



$G_f^- := (V, \overleftarrow{E}, f)$

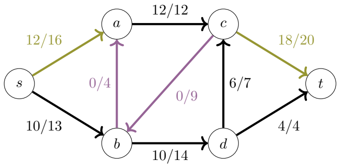
Residual Network

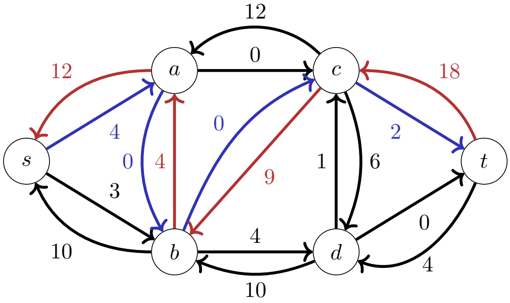




Residual network: $G_f := \mathbf{G_f^+} \cup \mathbf{G_f^-} = (V, E_f, c_f)$

Ford-Fulkerson: Flow augmentation





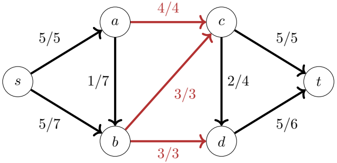
- **Augmenting Path:** Find a path $\mathbf{P : s \to t}$ with residual capacity $d > 0$ in G_f
- augment flow along this path for all $e \in P$ by d:
  - decrease residual capacity $\mathbf{c_f(e)}$ in G_f by d; increase $\mathbf{c_f}(\overleftarrow{e})$ by d
  - increase flow through $\mathbf{e} \in \mathbf{E}$ by d; decrease through $\overleftarrow{e} \in \mathbf{E}$

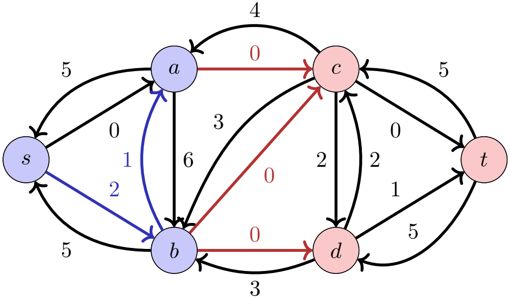
- total flow $|f|$ in G is increased

Algorithm Ford-Fulkerson(G, s, t)

```
Input: Flow network $G = (V, E, c)$, source $s$, sink $t$
Output: Maximal flow $f$

for $e \in E$ do
    $f(e) \leftarrow 0$
while exists positive path $P\colon s \rightsquigarrow t$ in residual network $G_f = (V, E_f, c_f)$ do
    $d \leftarrow \min_{e \in P} c_f(e)$
    foreach $e \in P$ do
        if $e \in E$ then
            $f(e) \leftarrow f(e) + d$
        else
            $f(\overleftarrow{e}) \leftarrow f(\overleftarrow{e}) - d$
```

Example Ford-Fulkerson





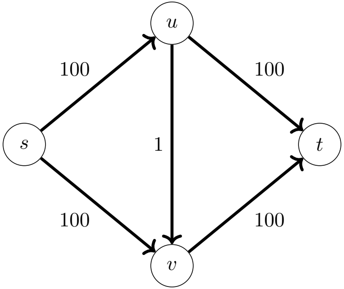
nodes reachable from s

nodes not reachable from s

all outgoing edges have residual capacity 0 in G_f

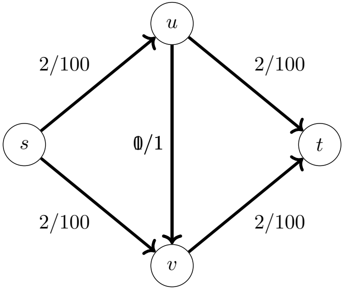
$\Rightarrow$ flow fully exhausts capacity on these edges!

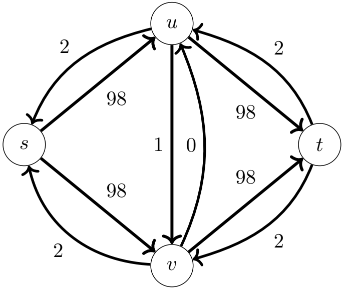
Quiz Ford-Fulkerson



How many iterations does Ford-Fulkerson need in the worst case?

Solution





After i iterations: $|f| = i$

$\Rightarrow$ in total $|f_{\max}| = 200$ iterations

Running Time Analysis of Ford-Fulkerson

Running time of each iteration: search of an augmenting path $s \rightsquigarrow t$
$\Rightarrow$ BFS or DFS: $\mathcal{O}(|V| + |E|) = \mathcal{O}(|E|)$

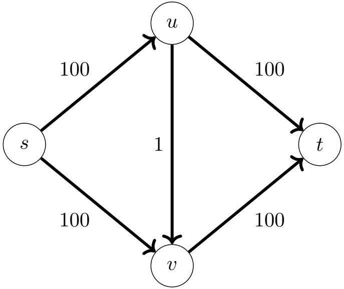
($|V| \leq |E|$, because all non-reachable nodes can be ignored.)

Number of iterations:
In every step, the size of the flow increases by $d > 0$.
integer capacities $\Rightarrow$ increment by $\geq 1 \Rightarrow$ at most $|f_{\max}|$ iterations

$\Rightarrow \mathcal{O}(|f_{\max}| \cdot |E|)$ for flow networks $G = (V, E, c)$ with $c\colon E \to \mathbb{N}^{\geq 1}$

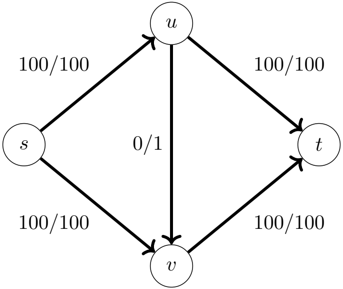
Edmonds-Karp Algorithm: (Variant of Ford-Fulkerson)
shortest augmenting path (number of edges) $\Rightarrow \mathcal{O}(|V| \cdot |E|^2)$ (without explanation)

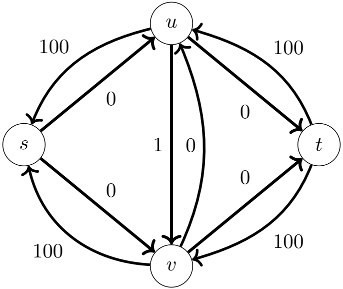
Quiz Edmonds-Karp



How many iterations does Edmonds-Karp need in the worst case?

Solution





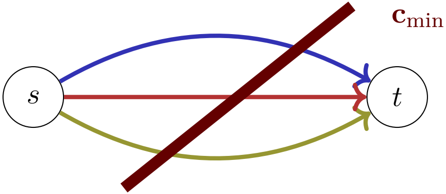
Termination after 2 iterations!

29.2 Max-Flow Min-Cut

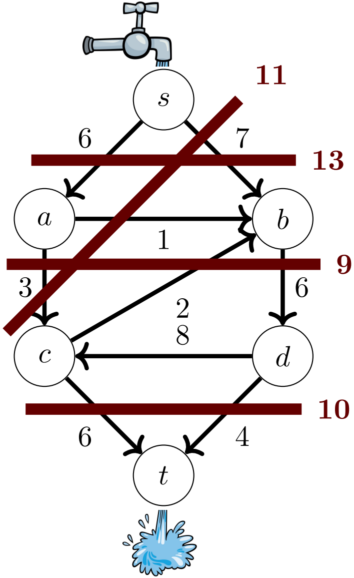
Flows and Cuts: Bottleneck Intuition

Upper bounds on size of flow:

- what can flow out of s: $c^+(s)$
- what can flow into t: $c^-(t)$
- what can flow through arbitrary cut
- what can flow through bottleneck: $c_{\min}$



$\Rightarrow$ flow $|f| \leq$ bottleneck
$\Rightarrow$ maximum flow $\leq$ bottleneck



Cut

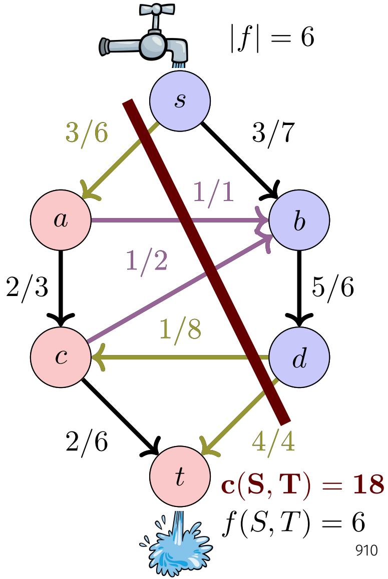
(s,t)**-Cut** of graph $G = (V, E, c)$: Partition $(\mathbf{S}, \mathbf{T})$ of V such that $s \in S, t \in T$

Size of cut:
$c(S,T) := \sum_{\mathbf{e}:\, \mathbf{S} \to \mathbf{T}} \mathbf{c}(\mathbf{e})$

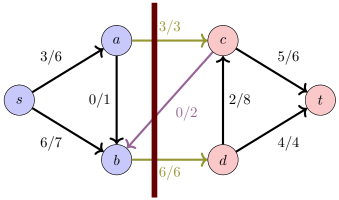
Flow through cut of flow network:
$f(S,T) := \sum_{\mathbf{e}:\, \mathbf{S} \to \mathbf{T}} \mathbf{f}(\mathbf{e}) - \sum_{\mathbf{e}:\, \mathbf{T} \to \mathbf{S}} \mathbf{f}(\mathbf{e})$

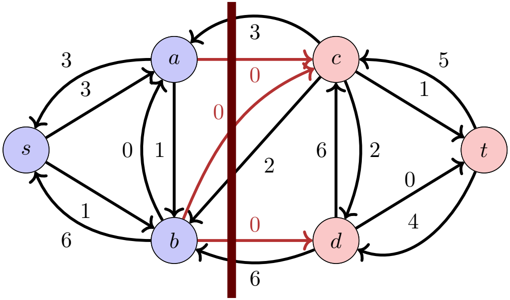
Observation:
$\forall f, S, T\colon |f| = f(S,T) \le c(S,T)$

$\Rightarrow |f_{\max}| \le c_{\min}$



Maximum Flow and Minimum Cut





after termination of Ford-Fulkerson/Edmonds-Karp:

- reachable from s, $\mathbf{T} \subseteq \mathbf{V}$ nodes not reachable from $s \Rightarrow$ **Cut** $(\mathbf{S}, \mathbf{T})$
- all outgoing edges e have remaining capacity 0 in G_f
- $f(S,T) = \sum_{\mathbf{e:\,S\to T}} \mathbf{f(e)} - \sum_{\mathbf{e:\,T\to S}} \mathbf{f(e)} = \sum_{\mathbf{e:\,S\to T}} \mathbf{c(e)} = \mathbf{c(S,T)}$
  $\Rightarrow |f_{\max}| = c_{\min}$

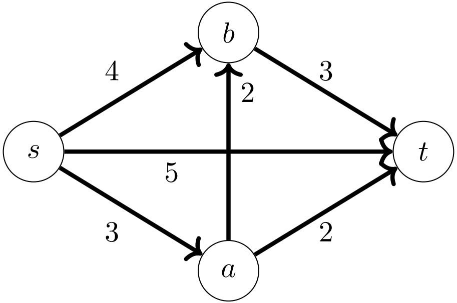
Max-Flow Min-Cut Theorem

Max-Flow Min-Cut Theorem

For a flow f in a flow network $G = (V, E, c)$ with source s and sink t, the following statements are equivalent:

1. f is a maximum flow in G
2. The residual network G_f does not provide any augmenting paths
3. $|f| = c(S, T)$ for a cut (S, T) of G.

Quiz



What is the minimum cut?
What is the maximum flow?

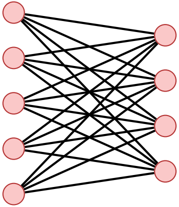
Application Examples

- Maximum Rate:
  - water in sewage system
  - cars in traffic
  - current in electrical networks
  - components on conveyors
  - information flow in communication networks
- Scheduling
- Bipartite Matching
- Image Segmentation

29.4 Maximales Bipartites Matching

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**.

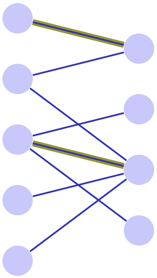


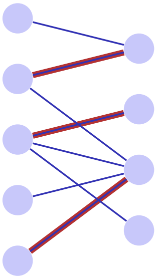
Application: maximal bipartite matching

Given: bipartite undirected graph $G = (V, E)$.

Matching M: $M \subseteq E$ such that $|\{m \in M : v \in m\}| \leq 1$ for all $v \in V$.

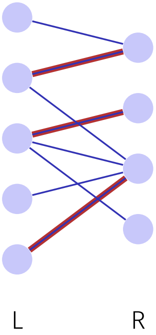
Maximal Matching M: Matching M, such that $|M| \geq |M'|$ for each matching M'.

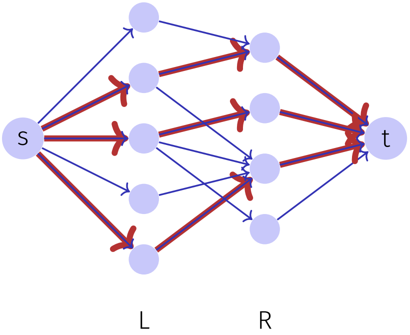




Corresponding flow network

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.





Summary

- Definitions: flow networks, flow, cut
- Concepts: Redirection, remainder network, augmenting path
- Algorithms
  - Greedy: incorrect!
  - Ford-Fulkerson: $\mathcal{O}(|f_{\max}| \cdot |E|)$
    Greedy augmenting paths in remainder network
  - Edmonds-Karp: $\mathcal{O}(|V| \cdot |E|^2)$
    Ford-Fulkerson with shortest augmenting paths (number of edges)
- Max Flow = Min Cut

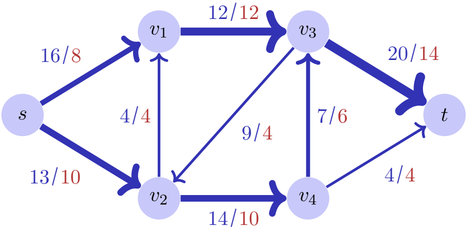
29.5 Appendix: Some Formal Things

Flow: Formulation with Skew Symmetry

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 $|f| = 18$.

Cuts

- **Capacity** of an (s,t)—-cut: $c(S,T) = \sum_{v\in S, v'\in T} c(v,v')$
- **Minimal cut**: cut with minimal capacity.
- **Flow over the cut**: $f(S,T) = \sum_{v\in S, v'\in T} f(v,v')$

Generally: Let $U, U' \subseteq V$

$$f(U,U') := \sum_{\substack{u\in U\\ u'\in U'}} f(u,u'), \qquad f(u,U') := f(\{u\},U')$$

Then

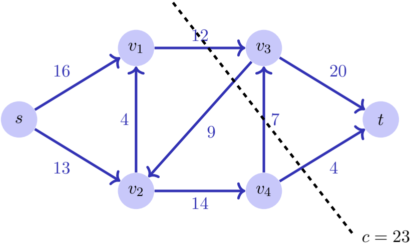
- $|f| = f(s,V)$
- $f(U,U) = 0$
- $f(U,U') = -f(U',U)$
- $f(X\cup Y, Z) = f(X,Z) + f(Y,Z)$, if $X \cap Y = \emptyset$.
- $f(R,V) = 0$ if $R \cap \{s,t\} = \emptyset$. [flow conversation!]

How large can a flow possibly be?

$$f(S,T) = f(S,V) - \underbrace{f(S,S)}_{0} = f(S,V)$$

$$= f(s,V) + f(\underbrace{S - \{s\}}_{\not\ni t, \not\ni s}, V) = |f|.$$

$$\Rightarrow |f| \le \sum_{v \in S, v' \in T} c(v,v') = c(S,T)$$



Rest Network

Rest network G_f provided by the edges with positive rest capacity:

$$
\begin{aligned}
G_f &:= (V, E_f, c_f) \\
c_f(u,v) &:= c(u,v) - f(u,v) \quad \forall u,v \in V \\
E_f &:= \{(u,v) \in V \times V | c_f(u,v) > 0\}
\end{aligned}
$$

- 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 $c_f(v,u) = f(u,v) > 0$.

The increased flow is a flow

Theorem 32

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 $p\}$

Theorem 33

The mapping $f_p : V \times V \to \mathbb{R}$,

$$f_p(u,v) = \begin{cases} c_f(p) & \text{if } (u,v) \text{ edge in } p \\ -c_f(p) & \text{if } (v,u) \text{ edge in } p \\ 0 & \text{otherwise} \end{cases}$$

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 $|f_p| = \sum_{v \in V} f_p(s,v) = f_p(s,u) = c_f(p)$.

Max-Flow Min-Cut Theorem

Theorem 34

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

$$|f| = f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) = \sum_{u \in S} \sum_{v \in T} c(u,v) = C(S,T).$$

$\blacksquare$

Edmonds-Karp Algorithm

Theorem 35

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]

Edmonds-Karp Algorithmus

Theorem 36

Wenn der Edmonds-Karp Algorithmus auf Flussnetzwerk $G = (V, E)$ mit Quelle s und Senke t angewendet wird, dann wächst für jeden Knoten $v \in V \setminus \{s, t\}$ die Distanz $\delta_f(s, v)$ des kürzesten Pfades von s nach v im Restnetzwerk G_f monoton mit jeder Flusserhöhung.

Beweis

Annahme: Distanz $\delta_f(s,v)$ wird bei Flusserhöhung $f \to f'$ kleiner für ein v: $\delta_f(s,v) < \delta_{f'}(s,v)$
Sei $p = s \rightsquigarrow u \to v$ kürzester Pfad von s nach v in $G_{f'}$, so dass $(u,v) \in E_{f'}$ und $\delta_{f'}(s,u) = \delta_{f'}(s,v) - 1$. Es gilt $\delta_{f'}(s,u) \geq \delta_f(s,u)$.
Wenn $(u,v) \in E_f$: $\delta_f s, v \leq \delta_f(s,u) + 1 \leq \delta_{f'}(s,u) + 1 = \delta_{f'}(s,v)$ Widerspruch.
Also $(u,v) \notin E_f$.

Integer number theorem

Theorem 37

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]

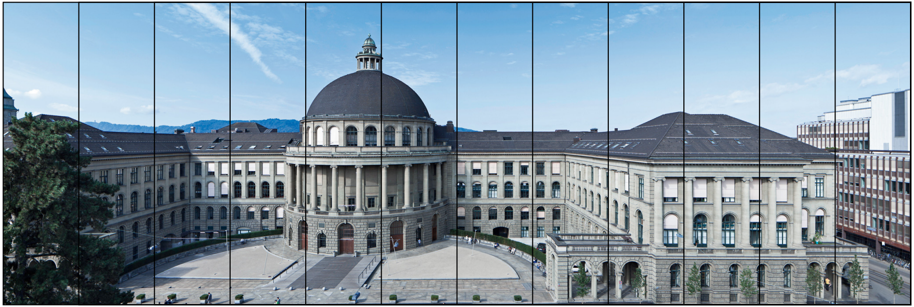
Consequence: Ford-Fulkerson generates for a flow network that corresponds to a bipartite graph a maximal matching $M = \{(u,v) : f(u,v) = 1\}$.

30. Parallel Programming I

Moore’s Law, Hardware Architectures, Parallel Execution , Multi-Threading, Parallelism and Concurrency, C++ Threads, Scalability: Amdahl and Gustafson , Scheduling

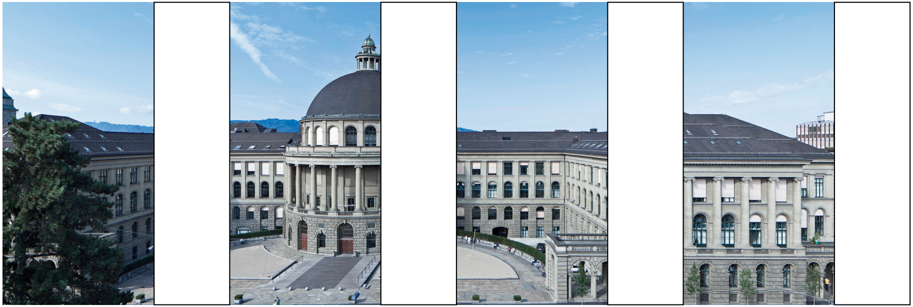
[Task-Scheduling: Cormen et al, Kap. 27] [Concurrency, Scheduling: Williams, Kap. 1.1 – 1.2]

Motivation: Paint a Picture (1 Artist)



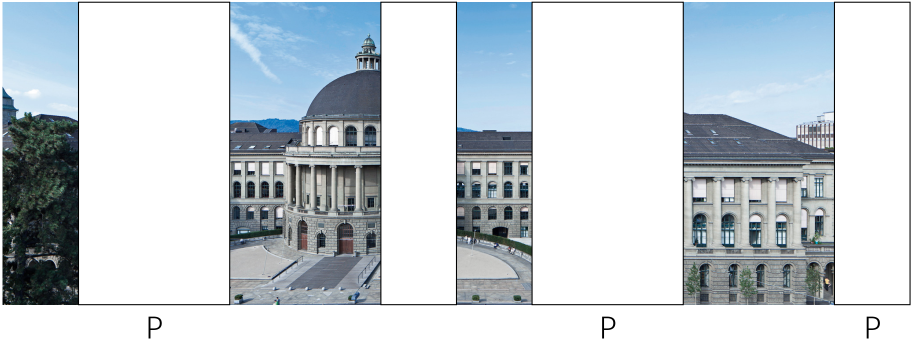
Model: sequential execution

Motivation: Paint a Picture (4 Artists)



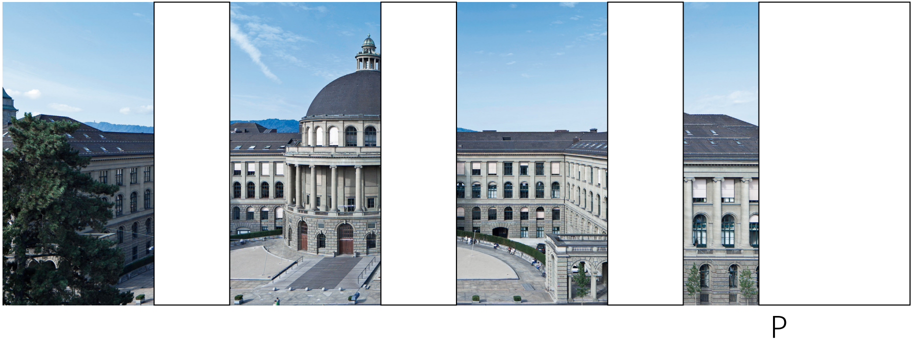
Model: parallel execution

Motivation: Paint a Picture (4 Artists, 3 Brushes)



Model: concurrent execution

Motivation: Paint a Picture (4 Artists, 1 Brush)



Model: concurrent execution

Models

- **Sequential:** the program is executed step-by-step in the prescribed order
- **Parallel:** tasks are executed in parallel. No synchronisation necessary, enough resources available.
- **Concurrent:** Tasks are executed in parallel. But there is a need for synchronisation: tasks have to be interrupted.

Why Parallelism and Concurrency?

- **Reactive / Interactive / Multi-User- Systems** particularly graphical user interfaces ⇒ Concurrency
- Computation-heavy tasks, like data processing and analysis, where **performance** is important ⇒ Parallelism
- **Natural parallelism / concurrency**: distributed systems, device drivers.

A Bit of Technical Background: CPUs

Today's typical computers (desktops, phones, ...) offer (at least)

- CPU: central processing unit, general-purpose computation device
- GPU: graphics processing unit, incredibly efficient for linear- algebra computations (games, graphics; machine learning).

Today's CPUs are typically multi-core:

- Each core is essentially a dedicated CPU that can execute code
- Examples:
  - Intel i7-8700K has 6 cores (from 2017)
  - Intel i9-12900KF has 16 cores (from 2021)
  - AMD Ryzen 9 5950X has 16 cores (from 2020)
  - Apple M1 Max has 10 cores (from 2021)

Intel i7 (2017)

Moore’s Law

Observation by Gordon E. Moore:

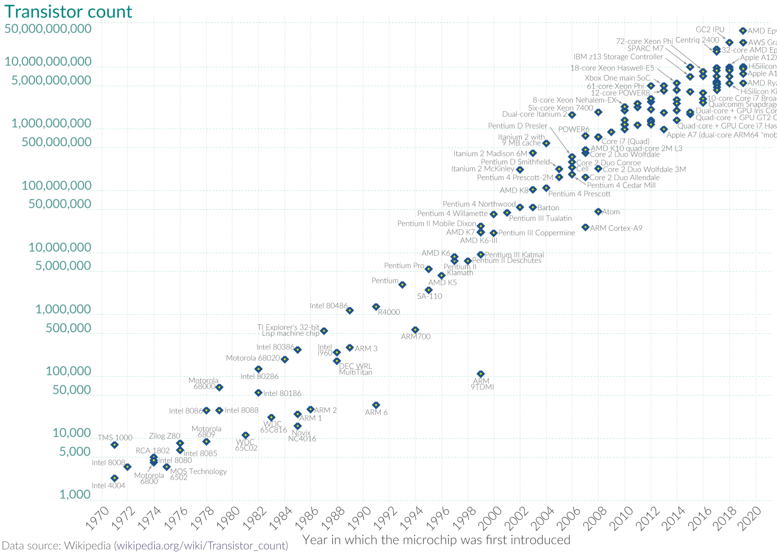
The number of transistors on integrated circuits doubles approximately every two years.

Gordon E. Moore (1929)

Moore's Law: The number of transistors on microchips doubles every two years

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years.
This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.



Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)
OurWorldinData.org – Research and data to make progress against the world's largest problems.

Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

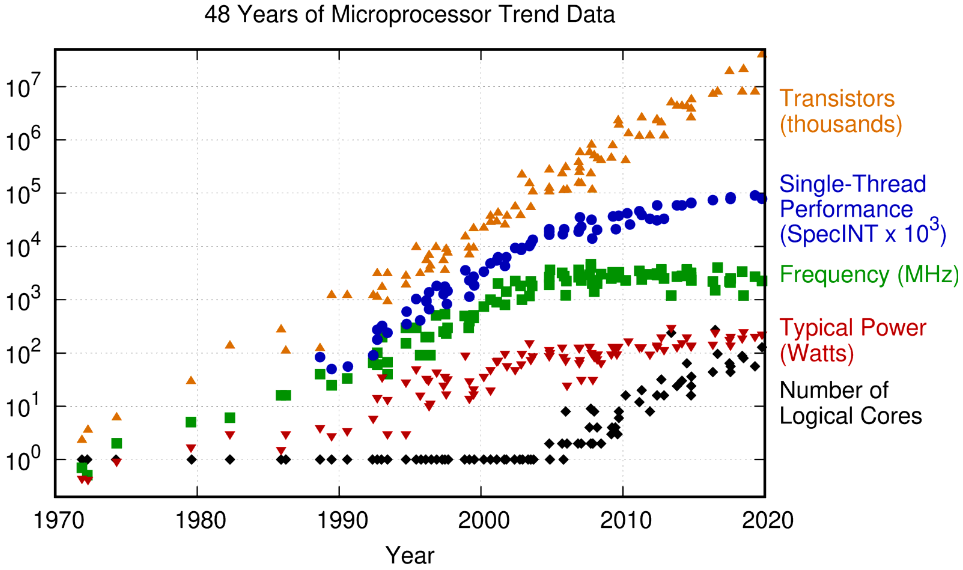
ourworldindata.org, https://en.wikipedia.org/wiki/Transistor_count

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 to improve the performance of their programs.

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)



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

https://github.com/karlrupp/microprocessor-trend-data

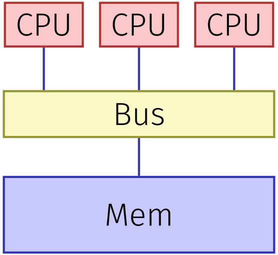
Multicore

- Use transistors for more compute cores
- Parallelism in the software

⇒ programmers have to write **parallel programs** to benefit from new hardware

(Simplified) Assumption: Computing Model

Independent Computing Cores



Shared Memory

30.1 Multi-Threading, Parallelism and Concurrency

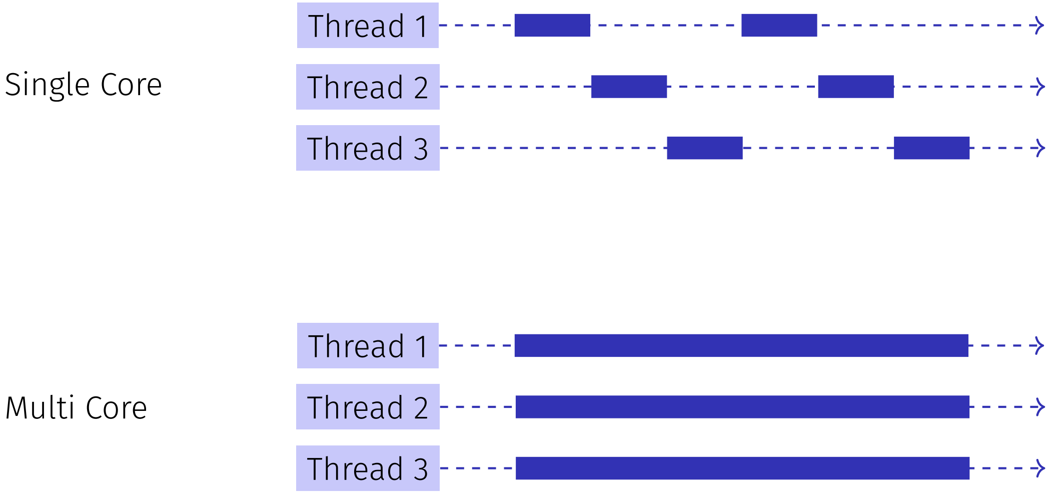
Processes and Threads

- **Process**: instance of a program
  - each process has a separate context, even a separate address space (”can only see its own memory“)
  - OS manages processes (resource control, scheduling, synchronisation)
- **Thread**: thread of execution of a program
  - Threads share the address space
  - fast context switch between threads

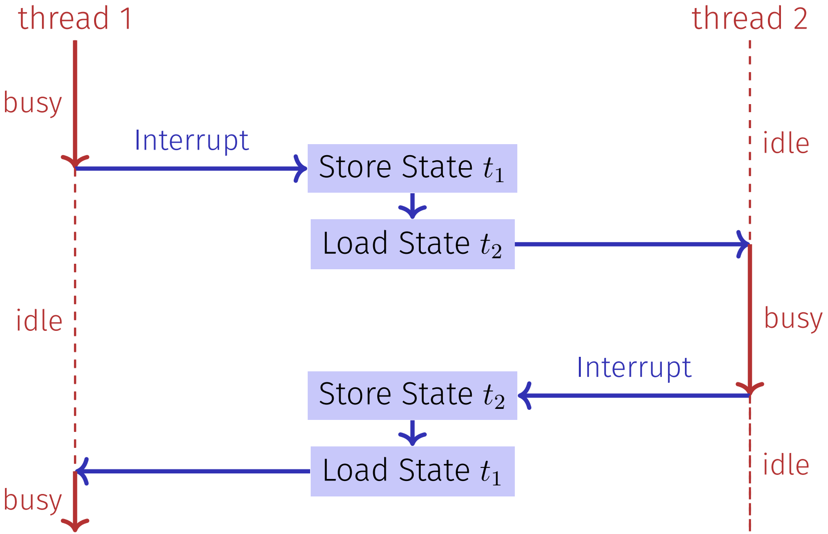
Why Multithreading?

- Avoid “polling” resources (files, network, keyboard)
- Interactivity (e.g. responsivity of GUI programs)
- Several applications / clients in parallel
- Parallelism (performance!)

Multithreading conceptually

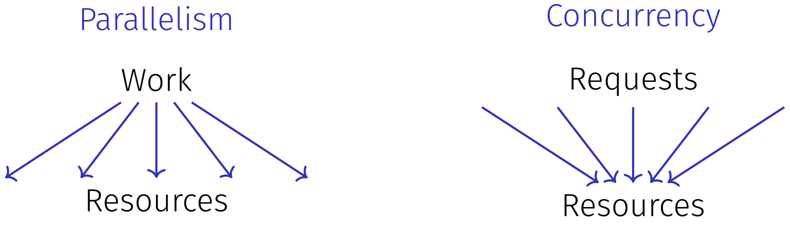


Thread switch on one core (Preemption)



Parallelism vs. Concurrency

- **Parallelism:** Use extra resources to solve a problem faster
- **Concurrency:** Correctly and efficiently manage access to shared resources
- The notions overlap. With parallel computations there is nearly always a need to synchronise.



Parallelism

Work

Resources

Concurrency

Requests

Resources

Thread Safety

Thread Safety means that in a concurrent application of a program this always yields the desired results.

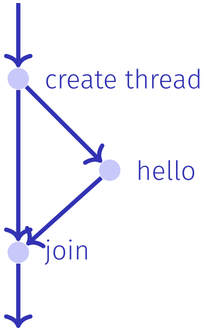
Many optimisations (Hardware, Compiler) target towards the correct execution of a *sequential* program.

Concurrent programs need an annotation that switches off certain optimisations selectively.

30.2 C++ Threads

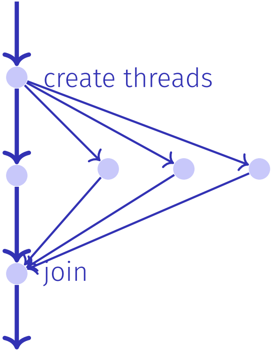
C++11 Threads

```
#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.

What can (not) be Parallelized

```
int pow8(int b){
  int b2 = b * b;
  int b4 = b2 * b2;
  return b4 * b4;
}

int main(){
  int x;
  std::cin >> x;
  x = pow8(x);
  std::cout << x;
}
```

- Program computes x^8
- All parts of the program must be executed in fixed order
  - Input → Computation → Output
  - b2 → b4 → b8
- No two computations are independent
- and must all be executed sequentially

What can (not) be Parallelized

```
int pow8(int b){
  int b2 = b * b;
  int b4 = b2 * b2;
  return b4 * b4;
}

int main(){
  std::vector<int> v = ....;
  for (int& x : v)
    x = pow8(x);
  ...
}
```

- Program computes x^8 for each $x \in v$
- The computation of x_i^8 does not depend on the computation of x_j^8.
- We can parallelise the computation of all x^8
- Parallelisation *can* reduce runtime if
  - the computation to be parallelised runs long enough)
  - sufficient CPUs are available

What can (not) be Parallelized

```
int main(){
  std::vector<int> v = ....;
  for (int& x : v)
    x = pow8(x);
  ...
}
```

- Example is obviously parallelisable
- Solche Probleme nennt man **emberassingly parallel**

- Many problems and algorithms are different:
  - can be parallelized but that requires a deeper analyis
  - need preprocessing or postprocessing in order to decompose the problem into parallelisable subproblems or to combine the partial results, respectively.
- Example: Matrix-Multiplication, Mergesort

30.3 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)

T_1: Sequential execution time

T_p: 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:

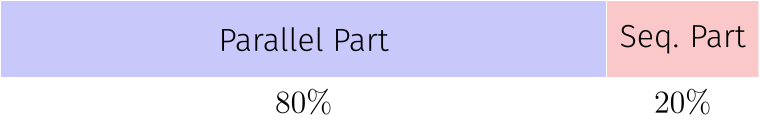
$$S_p = \frac{W/T_p}{W/T_1} = \frac{T_1}{T_p}.$$

- 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



$$
\begin{aligned}
T_1 &= 10 \\
T_8 &= \frac{10 \cdot 0.8}{8} + 10 \cdot 0.2 = 1 + 2 = 3 \\
S_8 &= \frac{T_1}{T_8} = \frac{10}{3} \approx 3.3 < 8 \quad (!)
\end{aligned}
$$

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

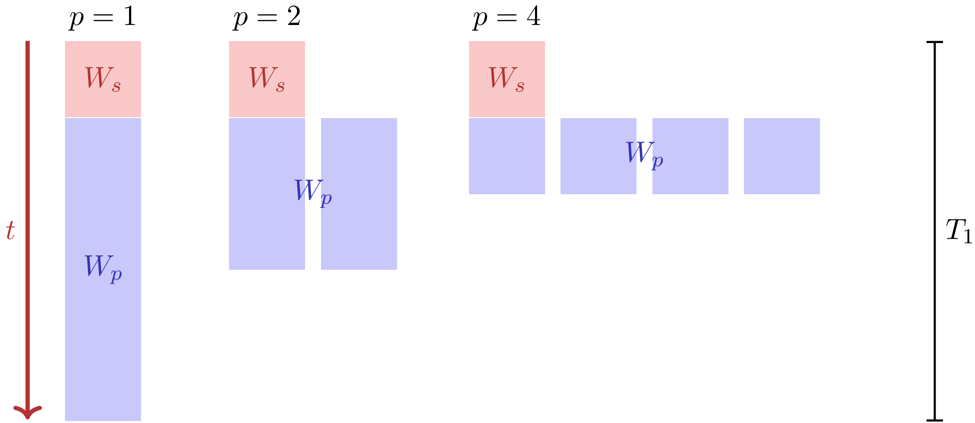
With sequential, not parallelizable fraction λ: $W_s = \lambda W$, $W_p = (1-\lambda)W$:

$$S_p \le \frac{1}{\lambda + \frac{1-\lambda}{p}}$$

Thus

$$S_\infty \le \frac{1}{\lambda}$$

Illustration Amdahl's Law



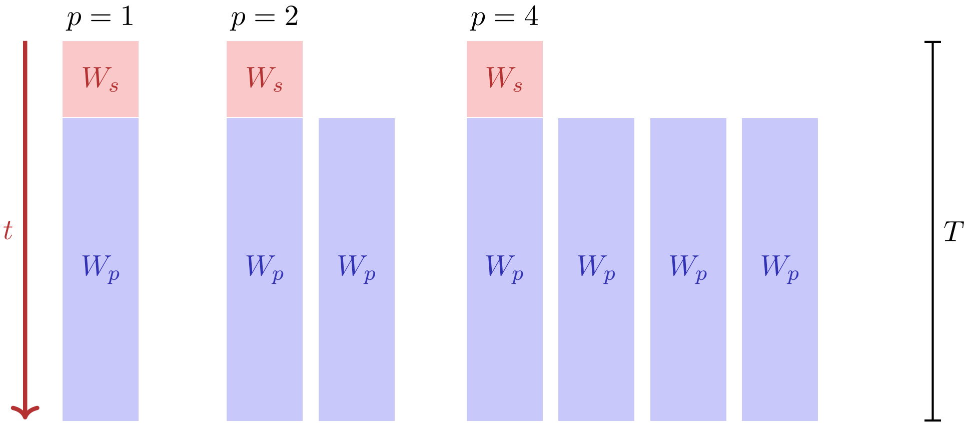
Amdahl’s Law is bad news

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:

$$W_s + W_p = T$$

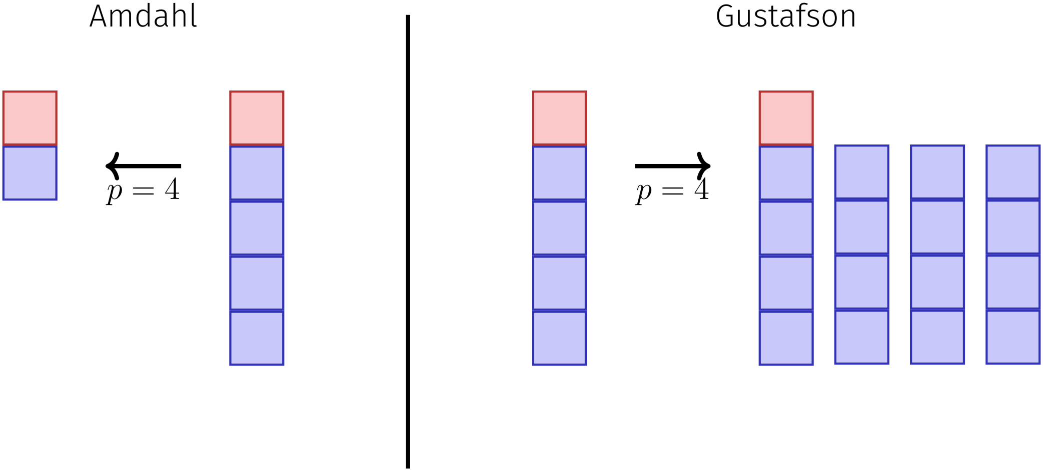
Work that can be executed by p processors in time T:

$$W_s + p \cdot W_p = \lambda \cdot T + p \cdot (1 - \lambda) \cdot T$$

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).

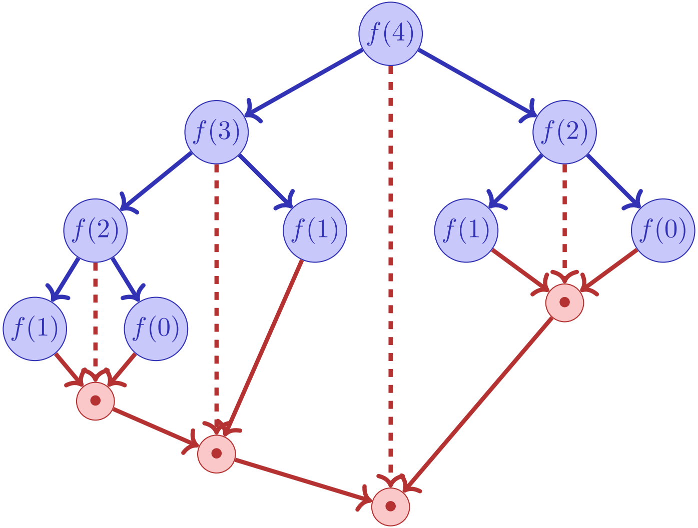
The two models do not contradict each other but describe the runtime speedup of different problems and algorithms.

30.4 Scheduling

Example: Fibonacci

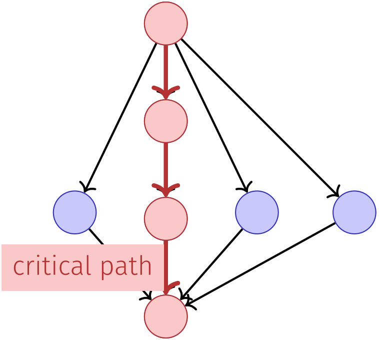
```
int fib_task(int x){
    if (x < 2) {
     return x;
  } else {
     auto f1 = std::async(fib_task, x-1);
     auto f2 = std::async(fib_task, x-2);
     return f1.get() + f2.get();
    }
}
```

Task-Graph



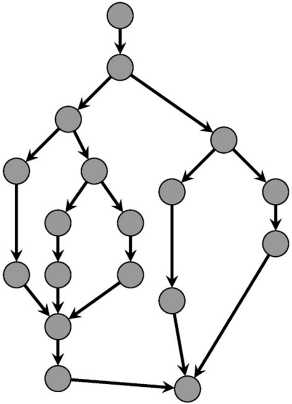
Question

- Each Node (task) takes 1 time unit.
- Arrows depict dependencies.
- Minimal execution time when number of processors = ∞?



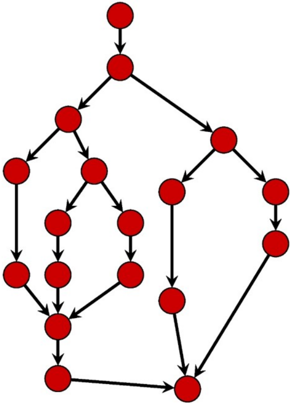
Performance Model

- p processors
- Dynamic scheduling
- T_p: Execution time on p processors



Performance Model

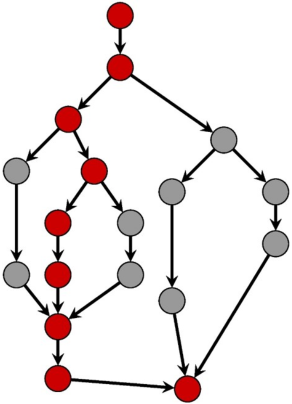
- T_p: Execution time on p processors
- T_1: **Work**: time for executing total work on one processor
- T_1/T_p: Speedup



Performance Model

- T_∞: **Span**: critical path, execution time on ∞ processors. Longest path from root to sink.
- T_1/T_∞: **Parallelism:** wider is better
- Lower bounds:

$$T_p \geq T_1/p \quad \text{Work law}$$
$$T_p \geq T_\infty \quad \text{Span law}$$



Greedy Scheduler

Greedy scheduler: at each time it schedules as many as availbale tasks.

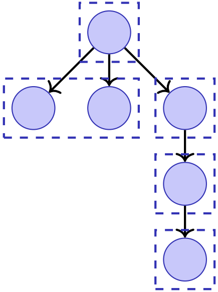
Theorem 38

On an ideal parallel computer with p processors, a greedy scheduler executes a multi-threaded computation with work T_1 and span T_∞ in time

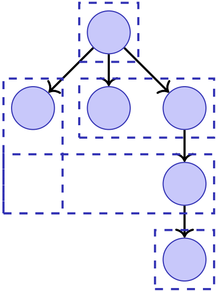
$$T_p \le T_1/p + T_\infty$$

Example

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_∞.

Consequence

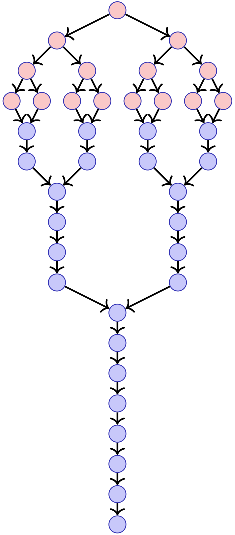
if $p \ll T_1/T_\infty$, i.e. $T_\infty \ll T_1/p$, then

$$T_p \leq T_1/p + T_\infty \quad \Rightarrow \quad T_p \lesssim T_1/p$$

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.

Example: Parallelism of Mergesort



split

merge

- 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)

31. Parallel Programming II

Shared Memory, Concurrency , 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]

31.1 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 common 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 ⇒ **Need for synchronisation**

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

t

```
Thread 1                      Thread 2

int b = getBalance();

                              int b = getBalance();

                              setBalance(b-amount);

setBalance(b-amount);
```

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!

t

```
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);
```

How ist this correctly implemented?

- We use **locks** (mutexes) from libraries
- They use hardware primitives, so called **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.

31.2 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
```

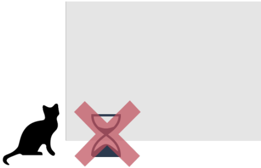
Required Properties of Mutual Exclusion

Correctness (Safety)

- At most one thread executes the critical section code

Liveness

- Acquiring the mutex must terminate in finite time when no process executes in the critical section



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();
  }
};
```

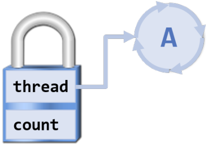
What if an exception occurs?

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);
  }
};
```

31.3 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!

Stack s shared between threads 1 and 2. Both threads call peek()

```
Thread 1                      Thread 2

int value = s.pop();
                              int value = s.pop();
s.push(value);
                              s.push(value);
                              return value;

return value;
```

Elements get swapped: the LIFO-invariant does not hold.

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(){
    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 A 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

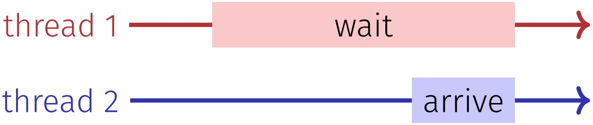
Modern compilers do not give guarantees that a global ordering of memory accesses is provided as in the sourcecode:

- 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
```

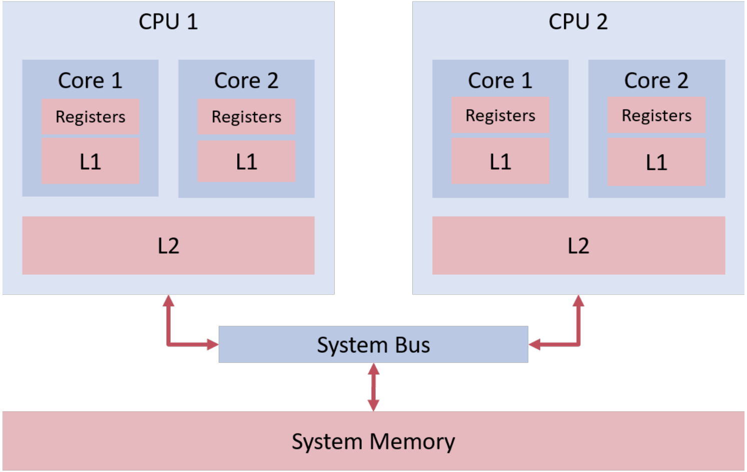
always

Hardware Perspective

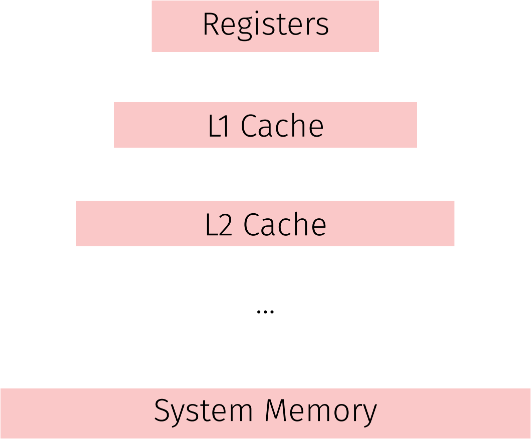
Modern multiprocessors do not enforce global ordering of all instructions for performance reasons:

- 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 Hierarchies



Memory Hierarchies



Registers

L1 Cache

L2 Cache

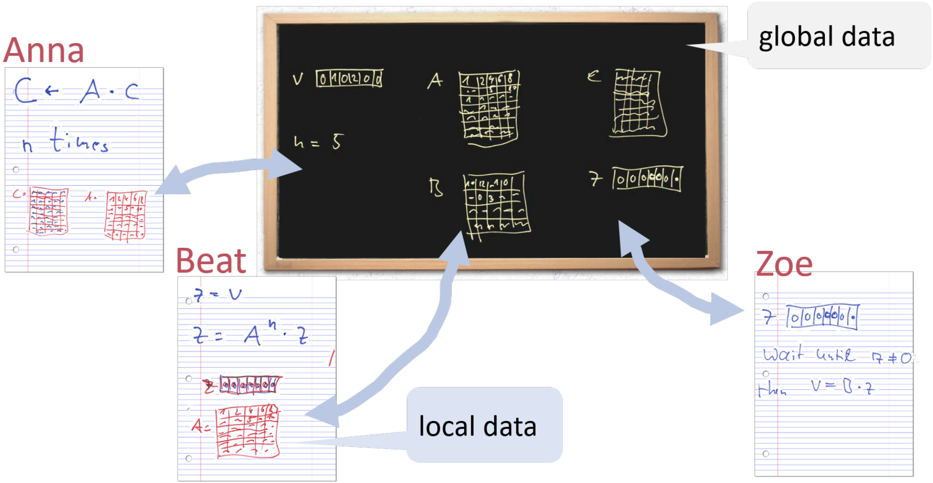
...

System Memory

fast,low latency, high cost, low capacity

slow,high latency,low cost,high capacity

An Analogy



Anna

Beat

Zoe

global data

local data

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
  }
};
```

32. 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]

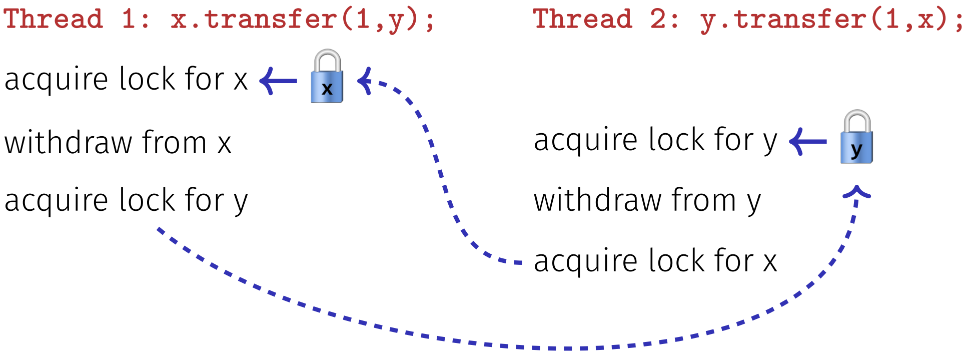
Deadlock Motivation

```
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

withdraw from x

acquire lock for y

Thread 2: y.transfer(1,x);

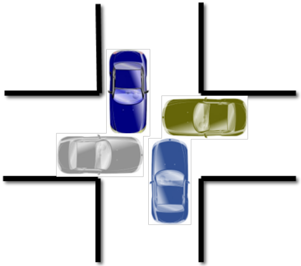
acquire lock for y

withdraw from y

acquire lock for x

Deadlock

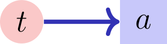
Deadlock: two or more processes are mutually blocked because each process waits for another of these processes to proceed.

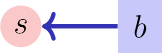


Threads and Resources

- Grafically

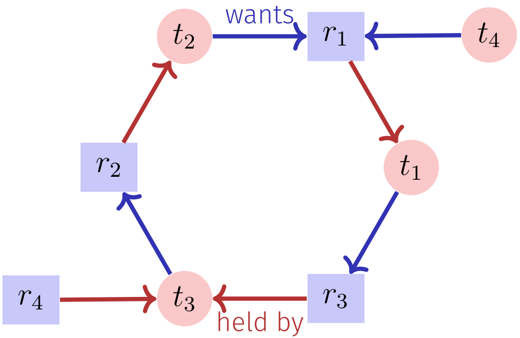
and Resources (Locks)

- Thread t attempts to acquire resource a: 

- Resource b is held by thread q: 

Deadlock – Detection

A deadlock for threads $t_1, \ldots, t_n$ occurs when the graph describing the relation of the n threads and resources $r_1, \ldots, r_m$ 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);
  }
};
```

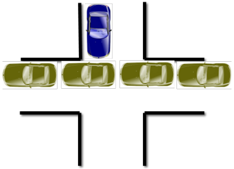
By the way...

```
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);
  }
};
```

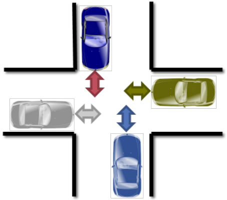
This would have worked here also. But then for a very short amount of time, money disappears, which does not seem acceptable (transient inconsistency!)

Starvation und Livelock

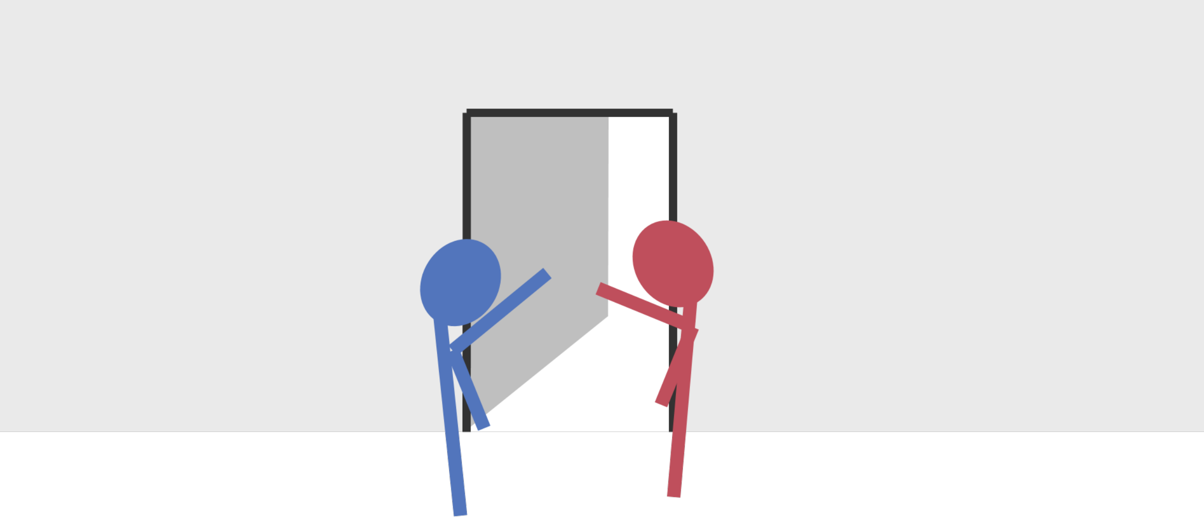
Starvation: the repeated but unsuccessful attempt to acquire a resource that was recently (transiently) free.



Livelock: competing processes are able to detect a potential deadlock but make no progress while trying to resolve it.



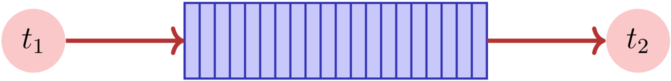
Politelock



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.

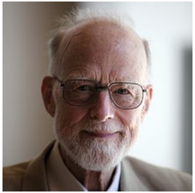
Moral

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.

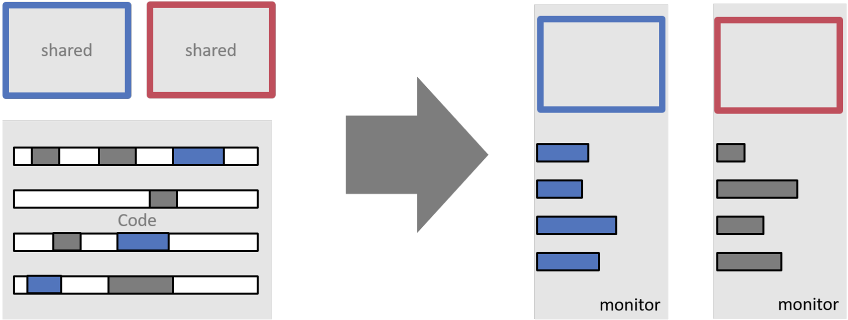
Invented by C.A.R. Hoare and Per Brinch Hansen (cf. Monitors – An Operating System Structuring Concept, C.A.R. Hoare 1974)



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 ();
}
```

33. 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

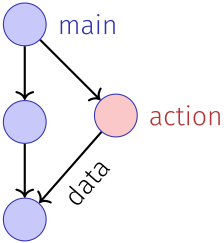
Up to this point, threads have been functions without a result:

```
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

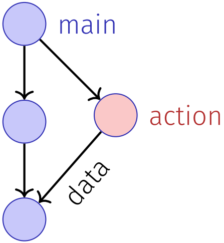
```
template <typename T>
class Buffer {
  std::queue<T> buf;
  std::mutex m;
  std::condition_variable cond;
public:
  void put(T x){ std::unique_lock<std::mutex> g(m);
    buf.push(x);
    cond.notify_one();
  }
  T get(){ std::unique_lock<std::mutex> g(m);
    cond.wait(g, [&]{return (!buf.empty());});
    T x = buf.front(); buf.pop(); return x;
  }
};
```

Simpler: only one value

```
template <typename T>
class Buffer {
  T value; bool received = false;
  std::mutex m;
  std::condition_variable cond;
public:
  void put(T x){ std::unique_lock<std::mutex> g(m);
    value = x; received = true;
    cond.notify_one();
  }
  T get(){ std::unique_lock<std::mutex> g(m);
    cond.wait(g, [&]{return received;});
    return value;
  }
};
```

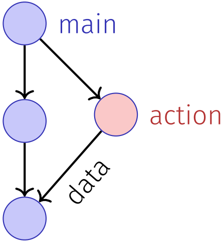
Application

```
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;
}
```

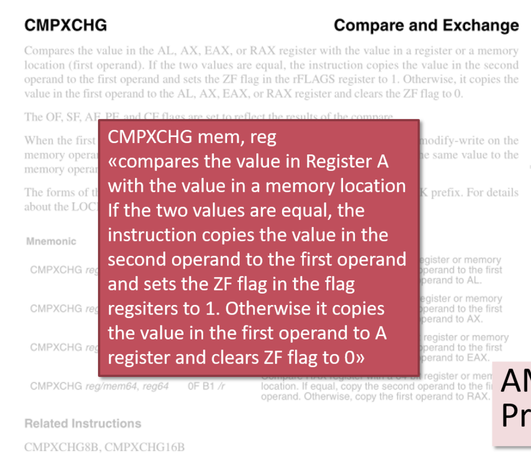


Disclaimer

The explanations above are simplified. The real implementation of a Future can deal with timeouts, exceptions, memory allocators and is generally written more closely to the unerlying operating system.

33.2 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.

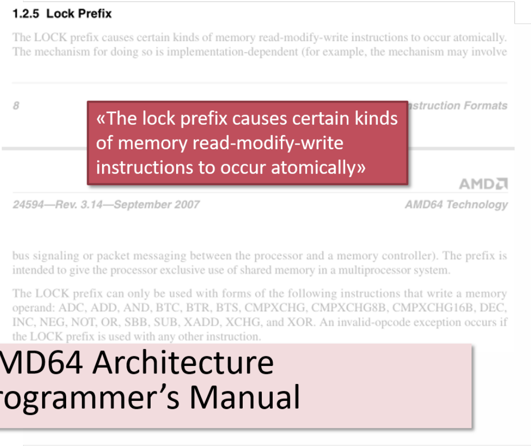
CMPXCHG *reg* … register or memory … perand to the first … perand to EAX.

CMPXCHG *reg/mem64, reg64* 0F B1 /r … 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 *…nstruction 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));
  }
};
```

33.3 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 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

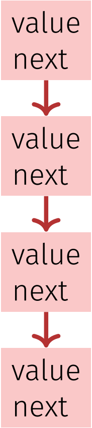
Simplified variant of a stack in the following

- pop does not check for an empty stack
- pop does not return a value

(Node)

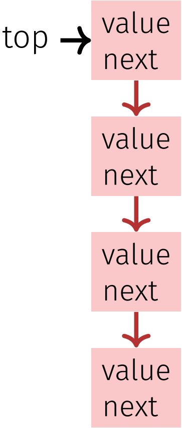
Nodes:

```
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;
    }
};
```



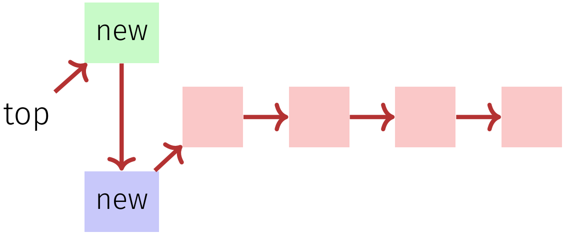
Lock-Free

```
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));
}
```

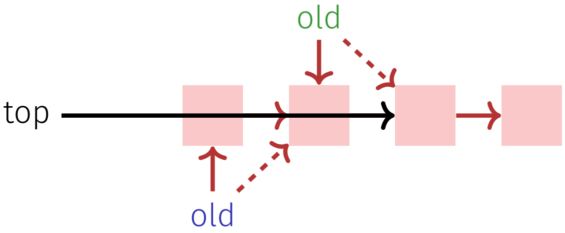
2 Threads:



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.