

Computer Science II

Course at D-BAUG, ETH Zurich

Felix Friedrich & Hermann Lehner

SS 2019

Welcome!

Course homepage

<http://lec.inf.ethz.ch/baug/informatik2/2019/>

The team:

Lecturers

Felix Friedrich

Hermann Lehner

Assistants

Patrick Gruntz

Aristeidis Mastoras

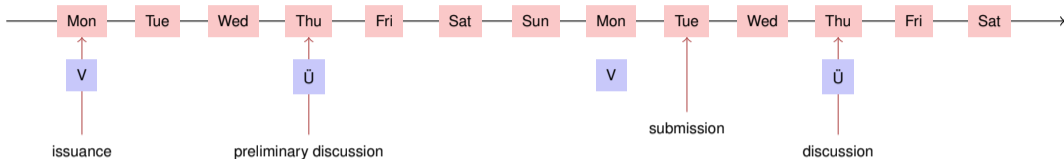
Chris Wendler

Manuel Winkler

Back-office

Katja Wolff

Exercises



- Exercises available at lectures.
- Preliminary discussion in the following recitation session
- Solution of the exercise until two days before the next recitation session.
- Discussion of the exercise in the next recitation session.

Exercises

- The solution of the weekly exercises is voluntary but *strongly* 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

Algorithmen Kapiere, *Aditya Y. Bhargava*, MITP, 2019.

Exams

The exam will cover

- Lectures content (lectures, handouts)
- Exercise content (recitation hours, exercise tasks).

Written exam.

We will test your practical skills (algorithmic and programming skills) and theoretical knowledge (background knowledge, systematics).

Offer

- Doing the weekly exercise series → bonus of maximally 0.25 of a grade points for the exam.
- The bonus is proportional to the achieved points of **specially marked bonus-task**. The full number of points corresponds to a bonus of 0.25 of a grade point.
- The **admission** to the specially marked bonus tasks can depend on the successful 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)

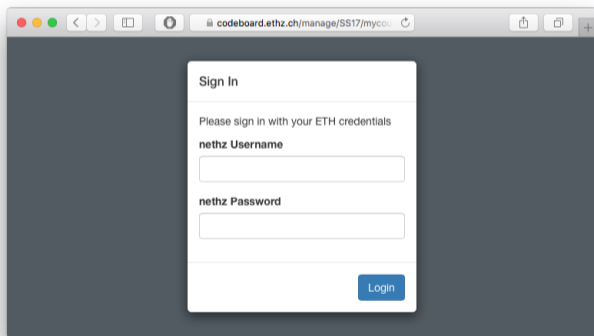
Academic integrity

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

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

Exercise group registration I

- Visit `http://expert.ethz.ch/enroll/SS19/ifbaug2`
- Log in with your nethz account.

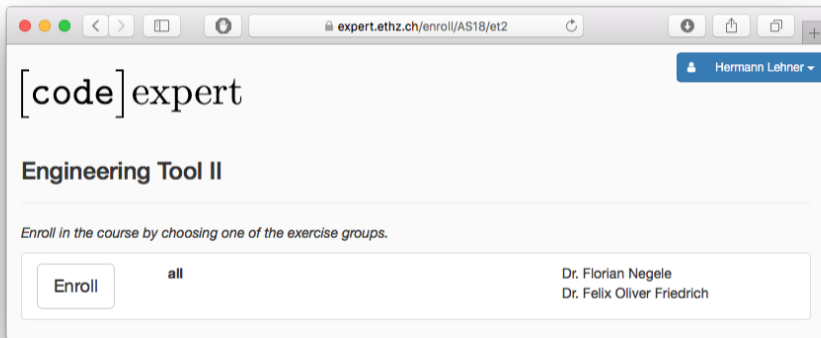


The image shows a browser window with the address bar containing `codeboard.ethz.ch/manage/SS17/myco...`. The main content area features a white 'Sign In' form centered on a dark grey background. The form includes the following elements:

- Sign In** (Section Header)
- Please sign in with your ETH credentials
- nethz Username** (Label) followed by an empty text input field.
- nethz Password** (Label) followed by an empty password input field.
- Login** (Button)

Exercise group registration II

Register with the subsequent dialog for an exercise group.



Overview

[code]expert

Felix Oliver Friedrich

Autum 2017

Enrolled Courses

My Exercise Groups

My Courses

Demo Course Demo Group - Dr. Hermann Lehner [change...](#)

Coding Demo Exercise

Earned XP

Submissions

Handout Date

Due Date



1,000 / 1,000

9. Sep. 2017 00:00

31. Dez. 2027 00:00

[Quadratic Equations In C++](#)

1,000 ✓

100%

Hand in now

Markdown Editor Manual

Submissions

Handout Date

Due Date



1. Aug. 2017 00:00

1. Aug. 2017 00:01

[Basic Markdown Syntax](#)

[Code Blocks and Inline Code](#)

Programming Exercise

The image shows a programming IDE interface. On the left, a code editor displays a C++ program for finding the minimum and maximum of ten integers. The code is as follows:

```
1 #include <iostream>
2
3 int main () {
4     int min; int max;
5     std::cin >> min;
6     max = min;
7     for (int i = 0; i < 8; ++i){ // (there is
8         int v;
9         std::cin >> v;
10        if (v<min) min = v;
11        if (v>max) max = v;
12    }
13    std::cout << min << "/" << max << std::endl;
14 }
```

On the right, a task description panel is visible. It includes the user name "Felix Oliver Friedrich", a "Status Not submitted yet" indicator, and a "Task" label. Below this, the task description reads: "minimum and maximum of a series of ten integers." It lists the input format as "10 consecutive integers number:int, example:" and shows an input example: "0 100250 45 0 0 1 -1000001 45 -25065 1". The expected output format is "minimum:int \"/>

At the bottom of the IDE, a control bar contains three icons: a gear (A), a play button (B), and a flask (C). A legend box explains these icons: "A: compile", "B: run", and "C: test".

Test and Submit

The screenshot displays a code editor interface with a dark theme. On the left, a sidebar shows 'Project Files' with 'main.cpp' selected. The main editor area is titled 'Minimax - Student Attempt' and contains the following C++ code:

```
1 #include <iostream>
2
3 int main () {
4     int min; int max;
5     std::cin >> min; std::cin >> max;
6     max = min-1;
7     for (int i = 0; i < 8; ++i){
8         int v;
9         std::cin >> v;
10        if (v<min) min = v;
11        if (v>max) max = v;
12    }
13    std::cout << min << "/" << max << std::endl;
14 }
```

Below the code, a console window shows the output of running tests:

```
Running tests.....
min_first passed
min_last passed
min_middle passed
max_first failed
input:
100251 -25065 45 -1000001 1 0 0 45 100250 0
expected output:
-1000001/100251
actual output:
-1000001/100250
-----
max_last passed
max_middle passed
unique passed

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

On the right side of the editor, a sidebar shows the user 'Felix Oliver Friedrich' and the status 'Status Not submitted yet'. A green button labeled 'Create new Submission' is visible, with a red arrow pointing to it from a pink box labeled 'Submission'. Below this, there are sections for 'Filter Snapshots' with a 'Create Snapshot' button, and 'First Working Version' (2 minutes ago) and 'Initial Snapshot' (13 minutes ago), each with a refresh and download icon. At the bottom of the console, a green progress bar is shown next to the score '86%'.

Where is the Save Button?

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

Snapshots

The image shows a code editor interface with a C++ program and its test results. The code is as follows:

```
1 #include <iostream>
2
3 int main () {
4     int min; int max;
5     std::cin >> min; std::cin >> max;
6     max = min;
7     for (int i = 0; i < 8; ++i){ // (there is a bug here)
8         int v;
9         std::cin >> v;
10        if (v<min) min = v;
11        if (v>max) max = v;
12    }
13    std::cout << min << "/" << M
14 }
```

The test results show that all tests passed:

```
Running tests.....
min_first passed
min_last passed
min_middle passed
max_first passed
max_last passed
max_middle passed
unique passed

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

The submission history sidebar on the right shows the following snapshots:

- Really Working Version (2 minutes ago)
- First Working Version (6 minutes ago)
- Initial Snapshot (16 minutes ago)

Annotations in the image include:

- A red box labeled "Look at snapshot" with an arrow pointing to the "First Working Version" entry.
- A red box labeled "Submission" with an arrow pointing to the "Initial Snapshot" entry.
- A red box labeled "Go Back" with an arrow pointing to the "Initial Snapshot" entry.

Should there be any Problems ...

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

1. Introduction

Algorithms and Data Structures, Correctness, a First Example

Goals of the course

- Understand the design and analysis of fundamental algorithms and data structures.
- Understand how an algorithmic problem is mapped to a sufficiently efficient computer program.

Contents

data structures / algorithms

The notion invariant, cost model, Landau notation

algorithms design, induction

searching, selection and sorting

dictionaries: hashing and search trees, balanced trees

dynamic programming

fundamental graph algorithms, shortest paths, maximum flow

Software Engineering

Python Introduction

Python Datastructures

1.1 Algorithms

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

Algorithm

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

example problem

Input: A sequence of n numbers (a_1, a_2, \dots, a_n)

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

Possible input

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

Every example represents a *problem instance*

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

Examples for algorithmic problems

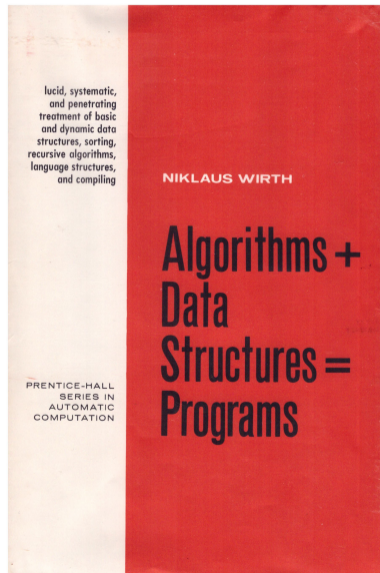
- Tables and statistics: sorting, selection and searching
- routing: shortest path algorithm, heap data structure
- DNA matching: Dynamic Programming
- evaluation order: Topological Sorting
- autocompletion and spell-checking: Dictionaries / Trees
- Fast Lookup : Hash-Tables
- The travelling Salesman: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

Characteristics

- Extremely large number of potential solutions
- Practical applicability

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.



Efficiency

Illusion:

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

Reality: resources are bounded and not free:

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

Actually, this course is nearly only about efficiency.

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]

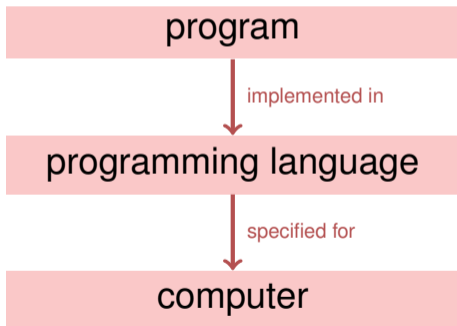
Efficiency of Algorithms

Goals

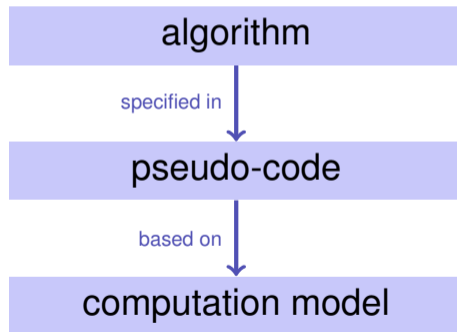
- Quantify the runtime behavior of an algorithm independent of the machine.
- Compare efficiency of algorithms.
- Understand dependence on the input size.

Programs and Algorithms

Technology



Abstraction



Technology Model

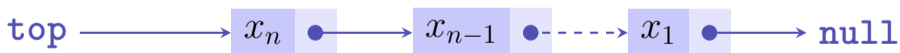
Random Access Machine (RAM)

- Execution model: instructions are executed one after the other (on one processor core).
- Memory model: constant access time (big array)
- Fundamental operations: computations (+, -, ·, ...) 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.

Pointer Machine Model

We assume

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



Asymptotic behavior

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

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

Example

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

Algorithms, Programs and Execution Time

Program: concrete implementation of an algorithm.

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

Beispiel

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

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

2.2 Function growth

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

Superficially

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

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

More precise: asymptotic upper bound

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

Definition:¹

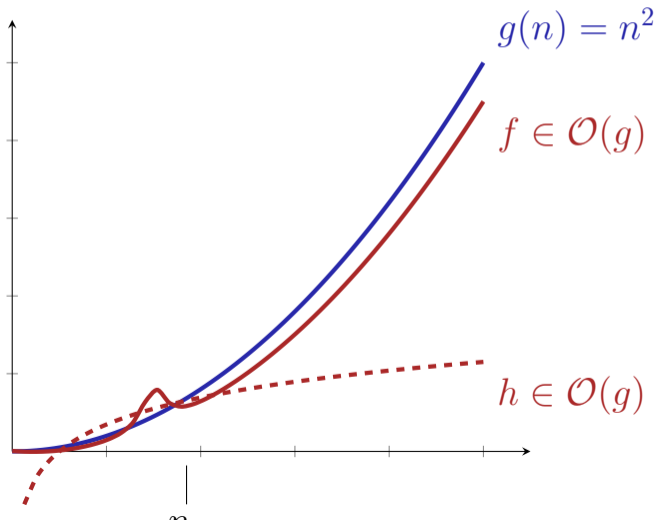
$$\begin{aligned} \mathcal{O}(g) = \{ & f : \mathbb{N} \rightarrow \mathbb{R} \mid \\ & \exists c > 0, \exists n_0 \in \mathbb{N} : \\ & \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n) \} \end{aligned}$$

Notation:

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

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

Graphic



Examples

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

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

Property

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

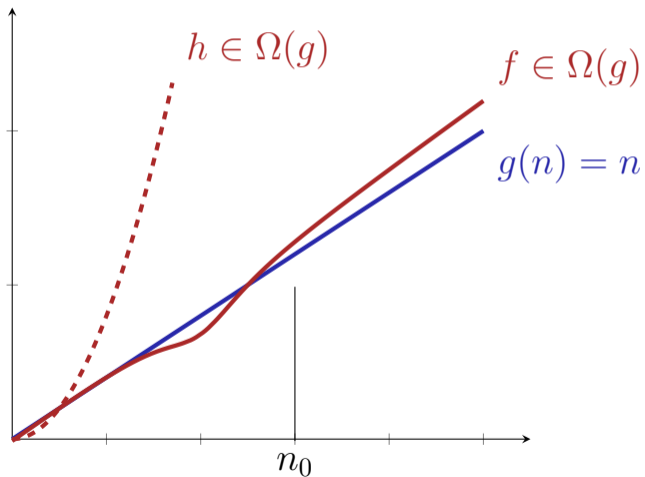
Converse: asymptotic lower bound

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

Definition:

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

Example



Asymptotic tight bound

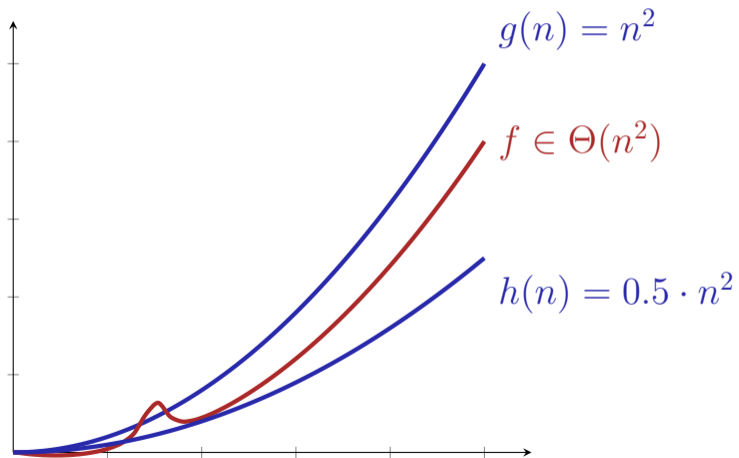
Given: function $g : \mathbb{N} \rightarrow \mathbb{R}$.

Definition:

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

Simple, closed form: exercise.

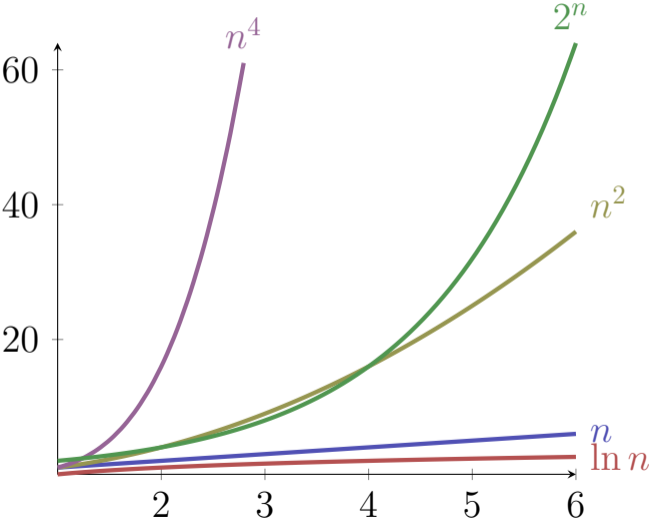
Example



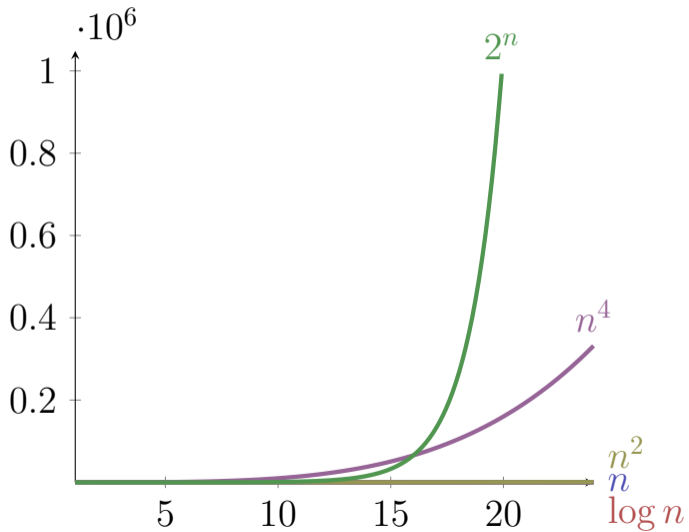
Notions of Growth

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

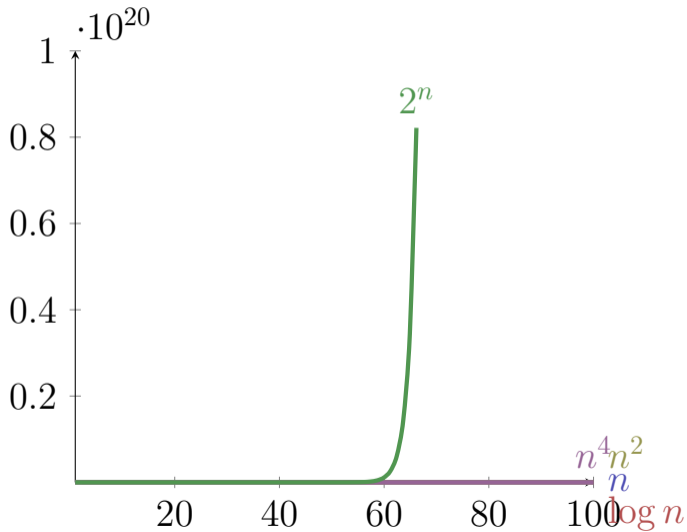
Small n



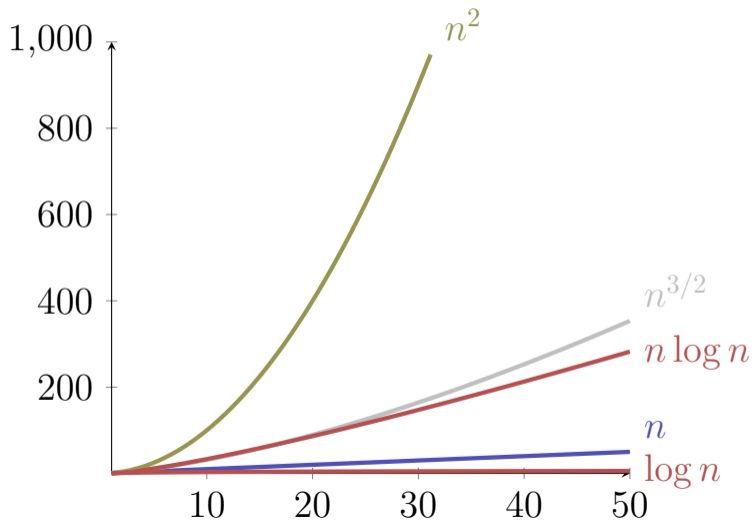
Larger n



“Large” n



Logarithms



Time Consumption

Assumption 1 Operation = $1\mu s$.

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

Useful Tool

Theorem

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

1 $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in \mathcal{O}(g), \mathcal{O}(f) \subsetneq \mathcal{O}(g).$

2 $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = C > 0$ (C constant) $\Rightarrow f \in \Theta(g).$

3 $\frac{f(n)}{g(n)} \xrightarrow{n \rightarrow \infty} \infty \Rightarrow g \in \mathcal{O}(f), \mathcal{O}(g) \subsetneq \mathcal{O}(f).$

About the Notation

Common casual notation

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

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

Clearly it holds that

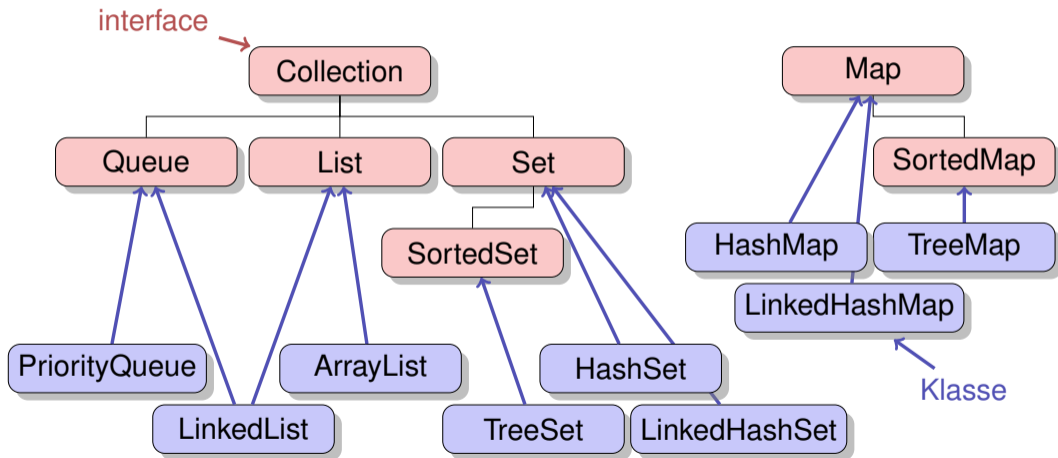
$$f_1 = \mathcal{O}(g), f_2 = \mathcal{O}(g) \not\Rightarrow f_1 = f_2!$$

Beispiel

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

We avoid this notation where it could lead to ambiguities.

Reminder: Java Collections / Maps

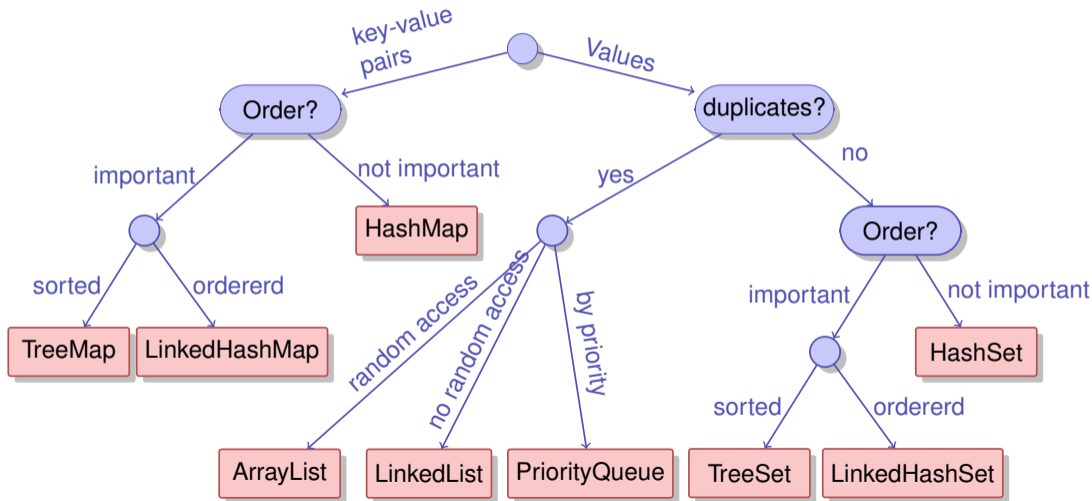


ArrayList versus LinkedList

run time measurements for 10000 operations (on [code] expert)

ArrayList	LinkedList
469 μ s	1787 μ s
37900 μ s	761 μ s
1840 μ s	2050 μ s
426 μ s	110600 μ s
31 ms	301ms
38ms	141ms
228ms	1080ms
648 μ s	757 μ s
58075 μ s	609 μ s

Reminder: Decision



Asymptotic Runtimes

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

Asymptotic running times (Anticipation!)

Data structure	Random Access	Insert	Next	Insert After Element	Search
ArrayList	$\Theta(1)$	$\Theta(1) A$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
LinkedList	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
TreeSet	–	$\Theta(\log n)$	$\Theta(\log n)$	–	$\Theta(\log n)$
HashSet	–	$\Theta(1) P$	–	–	$\Theta(1) P$

A = amortized, P =expected, otherwise worst case

3. Searching

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

The Search Problem

Provided

- A set of data sets

examples

telephone book, dictionary, symbol table

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

Task: find data set by key k .

Search in Array

Provided

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

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

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

Linear Search

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

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

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

Search in a Sorted Array

Provided

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

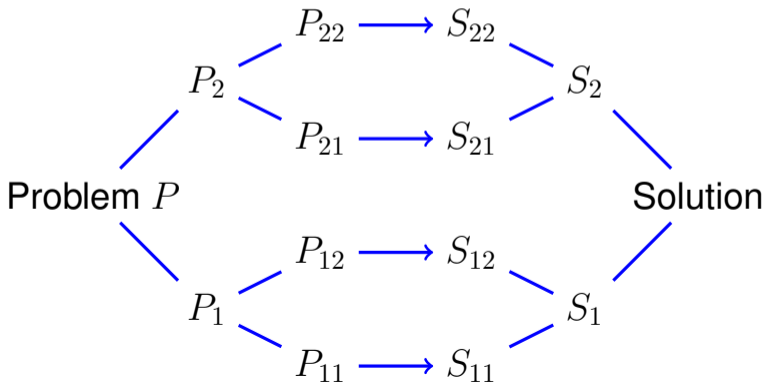
Wanted: index k , $1 \leq k \leq n$ with $A[k] = b$ or "not found".

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

divide et impera

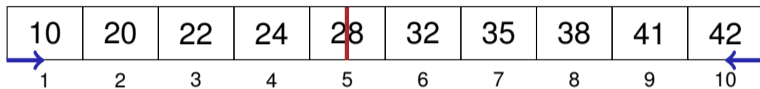
Divide and Conquer

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



Divide and Conquer!

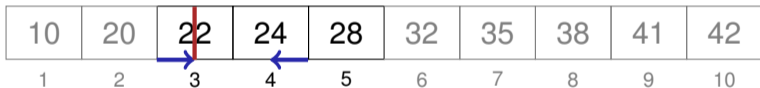
Search $b = 23$.



$b < 28$



$b > 20$



$b > 22$



$b < 24$



erfolglos

Binary Search Algorithm

BSearch($A[l..r], b$)

Input: Sorted array A of n keys. Key b . Bounds $1 \leq l \leq r \leq n$ or $l > r$ beliebig.

Output: Index of the found element. 0, if not found.

$m \leftarrow \lfloor (l + r) / 2 \rfloor$

if $l > r$ **then** // Unsuccessful search

return *NotFound*

else if $b = A[m]$ **then** // found

return m

else if $b < A[m]$ **then** // element to the left

return BSearch($A[l..m - 1], b$)

else // $b > A[m]$: element to the right

return BSearch($A[m + 1..r], b$)

Analysis (worst case)

Recurrence ($n = 2^k$)

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

Compute: ²

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

²Try to find a closed form of T by applying the recurrence repeatedly (starting with $T(n)$).

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: $(n/2 \rightarrow n)$

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

Result

Theorem

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

Iterative Binary Search Algorithm

Input: Sorted array A of n keys. Key b .

Output: Index of the found element. 0, if unsuccessful.

$l \leftarrow 1; r \leftarrow n$

while $l \leq r$ **do**

$m \leftarrow \lfloor (l + r)/2 \rfloor$

if $A[m] = b$ **then**

return m

else if $A[m] < b$ **then**

$l \leftarrow m + 1$

else

$r \leftarrow m - 1$

return *NotFound*;

4. Sorting

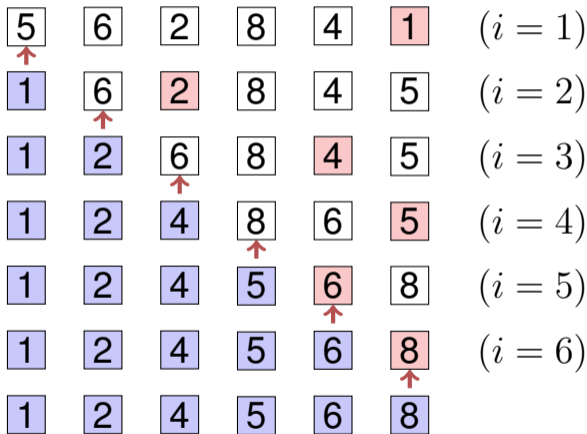
Simple Sorting, Quicksort, Mergesort

Problem

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

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

Selection Sort



- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element ($i \rightarrow i + 1$). 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: $n - 1 = \Theta(n)$

4.1 Mergesort

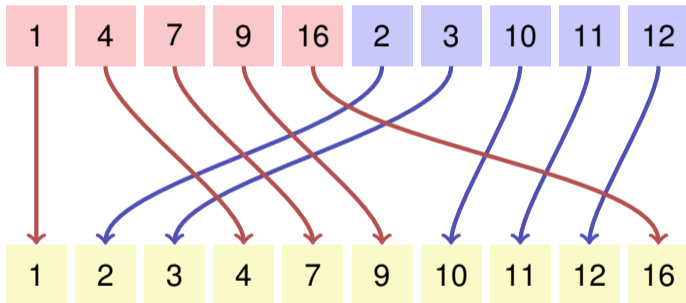
[Ottman/Widmayer, Kap. 2.4, Cormen et al, Kap. 2.3],

Mergesort

Divide and Conquer!

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

Merge



Algorithm Merge(A, l, m, r)

Input: Array A with length n , indexes $1 \leq l \leq m \leq r \leq n$.
 $A[l, \dots, m]$, $A[m + 1, \dots, r]$ sorted

Output: $A[l, \dots, r]$ sorted

1 $B \leftarrow$ new Array($r - l + 1$)

2 $i \leftarrow l$; $j \leftarrow m + 1$; $k \leftarrow 1$

3 **while** $i \leq m$ and $j \leq r$ **do**

4 **if** $A[i] \leq A[j]$ **then** $B[k] \leftarrow A[i]$; $i \leftarrow i + 1$

5 **else** $B[k] \leftarrow A[j]$; $j \leftarrow j + 1$

6 $k \leftarrow k + 1$;

7 **while** $i \leq m$ **do** $B[k] \leftarrow A[i]$; $i \leftarrow i + 1$; $k \leftarrow k + 1$

8 **while** $j \leq r$ **do** $B[k] \leftarrow A[j]$; $j \leftarrow j + 1$; $k \leftarrow k + 1$

9 **for** $k \leftarrow l$ **to** r **do** $A[k] \leftarrow B[k - l + 1]$

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 . $1 \leq l \leq r \leq n$

Output: Array $A[l, \dots, r]$ sorted.

if $l < r$ **then**

$m \leftarrow \lfloor (l + r) / 2 \rfloor$ // middle position

 Mergesort(A, l, m) // sort lower half

 Mergesort($A, m + 1, r$) // sort higher half

 Merge(A, l, m, r) // Merge subsequences

Analysis

Recursion equation for the number of comparisons and key movements:

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

Derivation for $n = 2^k$

Let $n = 2^k$, $k > 0$. Recurrence

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

Apply recursively

$$\begin{aligned} T(n) &= 2T(n/2) + cn = 2(2T(n/4) + cn/2) + cn \\ &= 2(2(T(n/8) + cn/4) + cn/2) + cn = \dots \\ &= 2(2(\dots(2(2T(n/2^k) + cn/2^{k-1})\dots) + cn/2^2) + cn/2^1) + cn \\ &= 2^k T(1) + \underbrace{2^{k-1}cn/2^{k-1} + 2^{k-2}cn/2^{k-2} + \dots + 2^{k-k}cn/2^{k-k}}_{k \text{ terms}} \\ &= nd + cnk = nd + cn \log_2 n \in \Theta(n \log n). \end{aligned}$$

4.2 Quicksort

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

Quicksort

② What is the disadvantage of Mergesort?

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

② How could we reduce the merge costs?

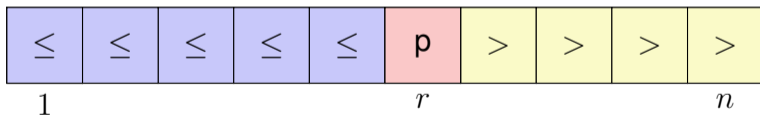
⚠ Make sure that the left part contains only smaller elements than the right part.

② How?

⚠ Pivot and Partition!

Use a pivot

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



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

Input: Array A , that contains the pivot p in the interval $[l, r]$ at least once.

Output: Array A partitioned in $[l..r]$ around p . Returns position of p .

while $l \leq r$ **do**

while $A[l] < p$ **do**

$l \leftarrow l + 1$

while $A[r] > p$ **do**

$r \leftarrow r - 1$

 swap($A[l], A[r]$)

if $A[l] = A[r]$ **then**

$l \leftarrow l + 1$

return $l-1$

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

Input: Array A with length n . $1 \leq l \leq r \leq n$.

Output: Array A , sorted between l and r .

if $l < r$ **then**

 Choose pivot $p \in A[l, \dots, r]$

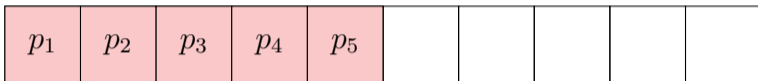
$k \leftarrow \text{Partition}(A[l, \dots, r], p)$

 Quicksort($A[l, \dots, k - 1]$)

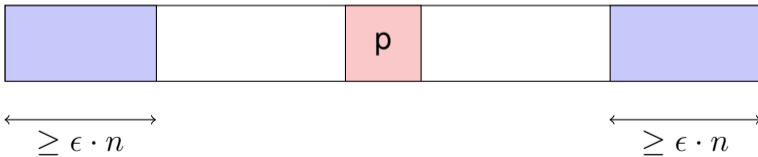
 Quicksort($A[k + 1, \dots, r]$)

Choice of the pivot.

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

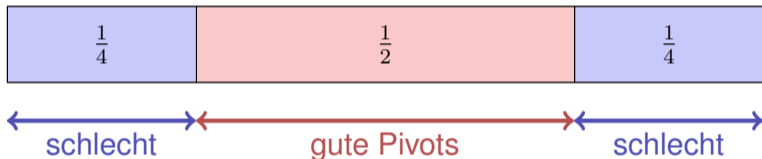


A good pivot has a linear number of elements on both sides.



Choice of the Pivot?

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 trials³: $1/\rho = 2$

³Expected value of the geometric distribution:

Quicksort (arbitrary pivot)

2 4 5 6 8 3 7 9 1

2 1 3 6 8 5 7 9 4

1 2 3 4 5 8 7 9 6

1 2 3 4 5 6 7 9 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:

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

Analysis (randomized quicksort)

Theorem

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

(without proof.)

Practical Considerations.

- Practically the pivot is often the median of three elements. For example: $\text{Median3}(A[l], A[r], A[\lfloor l + r/2 \rfloor])$.

5. From Java to Python

First Python Program, Transfer Java → Python, Dynamic Data Structures in Python

Learning Objectives

- see a new programming language (Python) and learn how to transfer from one programming language to another
- learn the most important differences between Java and Python, both from a syntactical and semantical point of view
- learn about the basic data types of Python (list, set, dict, tuple) and operations leveraging the use of such data types
- get used to the new programming language and environment (Python) by re-implementing known algorithms

First Java Program

```
public class Hello {  
    public static void main (String[] args) {  
        System.out.print("Hello World!");  
    }  
}
```


First Python Program

```
print("Hello World!")
```

Comments

Comments are preceded by a #

```
# prints 'Hello World!' to the console  
print("Hello World!")
```

Formatting Matters: Statements

- Whitespace is relevant
- Each line represents a statement
- So, *exactly one* Statement per line
- Comments start with #

Example program with two statements:

```
# two print-statements
print("Hurray, finally ...")
print("... no Semicolons!")
```

Formatting Matters: Blocks

- Blocks must be indented.
- All indented statements are part of a block. The block ends as soon as the indentation ends.
- Start of a Block is marked by a colon “:”

```
# in Python
while i > 0:
    x = x + 1 / i
    i = i - 1

print(x)
```

```
// in Java
while (i > 0) {
    x = x + 1.0 / i;
    i = i - 1;
}
System.out.print(x)
```

Literals: Numbers

- **integer**: 42, -5, 0x1b, 0o33, 7729684762313578932578932
Arbitrary precise integer numbers
- **float**: -0.1, 34.567e-4
Like **double** in Java, but precision depends on platform (CPU/
operating system)
- **complex**: 2 + 3j, (0.21 - 1.2j)
Complex numbers in the form $a+bj$. Optional round parentheses.

Literals: Booleans

- True
- False

Literals: Strings

- 'a single quoted string\nand a second line'
- "a double quoted string\nand a second line"
- Multi-line strings (triple double quotes):

```
"""a multiline string  
and a second line"""
```

Literals: Sequences

- *arrays*: There are no primitive arrays in Python
- **lists**: [17, True, "abc"] , []
Mutable ordered sequence of 0 or more Values of arbitrary types.
- **tuples**: (17, True, "abc") , (42,)
Immutable ordered sequence of 1 or more Values of arbitrary types.

Literals: Collections

- **dicts:** { "a": 42, "b": 27, False: 0 }, {}
Mutable Key-Value store. Keys and values may have arbitrary types.
- **sets:** {17, True, "abc"}, {42}
Mutable unordered sequence of 0 or more Values of arbitrary types. No duplicates.

Variables

- Variables are automatically created upon the first assignment
- The type of a variable is not checked upon assignment. That is, values of different types can be assigned to a variable over time.
- Assignment of values with the assignment operator: =
- Assignment to multiple variables with tuples

```
a = "Ein Text"
print(a) # prints: Ein Text
a = 42
print(a) # prints: 42
```

```
x, y = 4, 5
print(x) # prints: 4
print(y) # prints: 5
```

Variables

- Variables must always be assigned first before it's possible to read their value

Assume b never got a value assigned:

```
a = b
```

Results in the following error

NameError: name 'b' is not defined

Numeric and Boolean Operators

- Numeric operators as in Java: `+`, `-`, `*`, `/`, `%`, `**`, `//`
- Caution: “`/`” always results in a floating-point number
- `**`: Power function, $a^{**}b = a^b$.
- `//`: Integer division, `5//2` results in `2.5`.
- Comparison operators as in Java: `==`, `>=`, `<=`, `>`, `<`, `!=`
- Logical Operators: `and`, `or`, `not`
- Membership Operator: “`in`” Determines if a value is in a list, set or string.
- Identity Operator: “`is`” Checks if two variables point to the same object.

Input/Output

- Reading of inputs using `input()`
- A prompt can be provided.
- Output using `print(...)`
- `print` accepts one or more arguments and prints them separated with a space

```
name = input("What is your name: ")  
print("Hello", name)
```

Input/Output

- Input is always read as string
- To read a number, the input must be converted to a number first
- No implicit conversion happens
- Explicit conversion using:
`int()`, `float()`, `complex()`, `list()`, ...

```
i = int(input("Enter a number: "))  
print("The", i, "th power of two is", 2**i)
```

Conditions

- No parentheses required around the test
- `elif` to test another case
- Mind the indentation!

```
a = int(input("Enter a number: "))
if a == 42:
    print("Naturally, the answer")
elif a == 28:
    print("A perfect number, good choice")
else:
    print(a, "is just some boring number")
```

While-Loops

The well-known Collatz-Folge

```
a = int(input("Enter a number: "))
while a != 1:
    if a % 2 == 0:
        a = a // 2
    else:
        a = a * 3 + 1
print(a, end=' ')
```


For-Loops

- For-Loops work differently than in Java
- Iterates over the elements of the given set

```
some_list = [14, 'lala', 22, True, 6]
total = 0;
for item in some_list:
    if type(item) == int:
        total += item
print("Total of the numbers is", total)
```

For-Loops over a value range

- The function `range(start, end, step)` creates a list of values, starting with `start` until `end` - exclusive. Stepsize is `step`.
- Step size is 1 if the third argument is omitted.

```
# the following loop prints "1 2 3 4"  
for i in range(1,5):  
    print(i, end=' ')
```

```
# the following loop prints "10 8 6 4 2"  
for i in range(10, 0, -2):  
    print(i, end=' ')
```

Methods

■ The Cookie Calculator revisited

```
def readInt(prompt, atleast = 1):  
    """Prompt for a number greater 0 (or min, if specified)"""  
    number = 0;  
    while number < atleast:  
        number = int(input(prompt))  
        if (number < atleast):  
            print("Too small, pick a number larger than", atleast)  
    return number
```

```
kids = readInt("Kids: ")  
cookies = readInt("Cookies: ", atleast=kids)  
print("Each Kid gets", cookies // kids, "cookies.")  
print("Papa gets", cookies % kids, "cookies.")
```

Lists: Basic Operations

- Element-Access (0-based): `a[2]` points to the third element.
- Negative indices count from the last element!

```
a = [ 3, 7, 4]
print(a[-1]) # prints '4'
```

- Add value to the tail: `a.append(12)`
- Test if an element is in a collection:

```
if 12 in a:
    print('12 is in the list, we just added it before')
```

- Anzahl Elemente in einer Collection: `len(a)`

Lists: Slicing

- Slicing: address partition: `a[start:end]`
- `a` and/or `b` are positive or negative indices.
- `end` is not inclusive

```
a = [ 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(a[2:4])      # [3, 4]
print(a[3:-3])    # [4, 5, 6]
print(a[-3:-1])   # [7, 8]
print(a[5:])      # [6, 7, 8, 9]
print(a[:3])      # [1, 2, 3]
```

Dictionaries

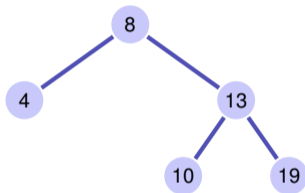
Dictionaries are very important primitive data structures in Python

- Easy and efficient possibility to name and group several fields of data
- Build hierarchical data structures by nesting
- Accessing elements using [] Operator

```
record = { 'Firstname': 'Hermann', 'Lastname': 'Lehner',  
          'Salary': 420000, 'Mac User': True }  
record['Salary'] = 450000  
if record['Mac User']:  
    print('... one more thing!')
```

Dynamic Data Structures with Dicts

```
tree = {  
    'key': 8,  
    'left' : {  
        'key': 4, 'left' : None, 'right': None  
    },  
    'right': {  
        'key': 13,  
        'left' : {  
            'key': 10, 'left' : None, 'right': None  
        },  
        'right': {  
            'key': 19, 'left' : None, 'right': None  
        }  
    }  
}
```



Dynamic Data Structures with Dicts

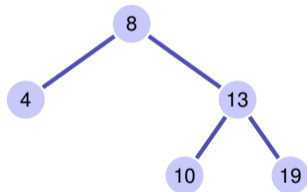
■ Working with Dicts (Examples)

```
l = tree['left'] # assign left subtree to variable l
l['key'] = 6     # changes key from 4 to 6

if l['left'] is None: # proper way to test against None
    print("There is no left child here...")
else:
    print("Value of left subtree is", l['left']['key'])
```


Dynamic Data Structures with Classes

```
class Node:  
    def __init__(self, k, l=None, r=None):  
        self.key, self.left, self.right = k, l, r
```



```
# create the tree depicted on the right  
rightSubtree = Node(13, l=Node(10), r=Node(19))  
tree = Node(8, l=Node(4), r=rightSubtree)
```

```
# an example query  
print(tree.right.right.key) # prints: 19
```

Modules

Python has a vast amount of libraries in form of modules that can be imported.

■ Importing a whole module:

```
import math  
x = math.sqrt(4)
```

```
from math import *  
x = sqrt(4)
```

■ Importing parts of a module:

```
from datetime import date  
t = date.today()
```

6. Advanced Python Concepts

Built-in Functions, Conditional Expressions, List and Dict Comprehension, File IO, Exception-Handling

Built-In Functions: Enumerate with Indices

Sometimes, one wants to iterate through a list, including the index of each element. This works with `enumerate(...)`

```
data = [ 'Spam', 'Eggs', 'Ham' ]  
  
for index, value in enumerate(data):  
    print(index, ":", value)
```

Output:

```
0 : Spam  
1 : Eggs  
2 : Ham
```

Built-In Functions: Combining lists

There is a simple possibility to combine lists element-wise (like a zipper!): `zip(...)`

```
places = [ 'Zurich', 'Basel', 'Bern' ]  
plz = [ 8000, 4000, 3000, ]
```

```
list(zip(places, plz))  
# [('Zurich', 8000), ('Basel', 4000), ('Bern', 3000)]
```

```
dict(zip(places, plz))  
# {'Zurich': 8000, 'Basel': 4000, 'Bern': 3000}
```

Conditional Expressions

In Python, the value of an expression can depend on a condition (as part of the expression!)

Example: Collaz Sequence

```
while a != 1:  
    a = a // 2 if a % 2 == 0 else a * 3 + 1
```

Example: Text formatting

```
print('I see', n, 'mouse' if n == 1 else 'mice')
```

List Comprehension

- Python provides a convenient way of creating lists declaratively
- Similar technique to 'map' and 'filter' in functional languages

Example: Read-in a sequence of numbers

```
line = input('Enter some numbers: ')
s_list = line.split()
n_list = [ int(x) for x in s_list ]
```

The same combined in one expression

```
n_list = [ int(x) for x in input('Enter some numbers: ').split() ]
```

List Comprehension

Example: Eliminate whitespace in front and at the back

```
line = [ ' some eggs ', ' slice of ham ', ' a lot of spam ' ]  
cleaned = [ item.strip() for item in line ]  
  
# cleaned == [ 'some eggs', 'slice of ham', 'a lot of spam' ]
```


Dict Comprehension

- Like with lists, but with key/value pairs

Example: extract data from a dict

```
data = {  
    'Spam' : { 'Amount' : 12, 'Price': 0.45 },  
    'Eggs' : { 'Price': 0.8 },  
    'Ham'   : { 'Amount': 5, 'Price': 1.20 }  
}
```

```
total_prices = { item : record['Amount'] * record['Price']  
    for item, record in data.items()  
    if 'Amount' in record }
```

```
# total_prices == {'Spam': 5.4, 'Ham': 6.0}
```

File IO

- Files can be opened with the command `open`
- To automatically close files afterwards, this must happen in a `with` block

Example: Read CSV file

```
import csv

with open('times.csv', mode='r') as csv_file:
    csv_lines = csv.reader(csv_file)
    for line in csv_lines:
        # do something for each record
```

Writing works similarly. See Python documentation.

Exception Handling

Given the following code:

```
x = int(input('A number please: '))
```

If no number is entered, the program crashes:

```
Traceback (most recent call last):
```

```
  File "main.py", line 1, in <module>
```

```
    x = int(input('A number please: '))
```

```
ValueError: invalid literal for int() with base 10: 'a'
```

We can catch this error and react accordingly.

Exception Handling

```
try:
    x = int(input('A number please: '))
except ValueError:
    print('Oh boy, that was no number...')
    x = 0
print('x:', x)
```

Output, if spam is entered instead of a number:

```
Oh boy, that was no number...
x: 0
```

7. Java Input/Output

User Input/Console Output, File Input and Output (I/O)

User Input (half the truth)

- e.g. reading a number: `int i = In.readInt();`
- Our class `In` provides various such methods.
- Some of those methods have to deal with wrong inputs: What happens with `readInt()` for the following input?

"spam"

User Input (half the truth)

```
public class Main {  
    public static void main(String[] args) {  
        Out.print("Number: ");  
        int i = In.readInt();  
        Out.print("Your number: " + i);  
    }  
}
```

It seems not much happens!

Number: spam

Your number: 0

User Input (the whole truth)

- e.g. reading a number using the class `Scanner`

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Out.print("Number: ");
        Scanner input = new Scanner(System.in);
        int i = input.nextInt();
        Out.print("Your number: " + i);
    }
}
```

What happens for the following input?

"spam"

User Input (the whole truth)

Number: spam

```
Exception in thread "main" java.util.InputMismatchException
    at java.base/java.util.Scanner.throwFor(Scanner.java:939)
    at java.base/java.util.Scanner.next(Scanner.java:1594)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
    at Main.main(Main.java:7)
    at TestRunner.main(TestRunner.java:330)
```

Oh, we come back to this in the next chapter...

Console Output

- Until now, you knew: `Out.print("Hi")` oder `Out.println("Hi")`
- Without our `Out` class:

```
System.out.print("The answer is: ");  
System.out.println(42);  
System.out.println("What was the question?!");
```

This leads to the following output:

```
The answer is: 42  
What was the question?!
```

So: User Input/Console Output

- Reading of input via the input stream `System.in`
- Writing of output via output stream `System.out`

Reading/Writing Files (line by line)

- Files can be read byte by byte using the class `java.io.FileReader`
- To read entire lines, we use in addition a `java.io.BufferedReader`
- Files can be written byte by byte using the class `java.io.FileWriter`
- To read entire lines, we use in addition a `java.io.BufferedWriter`

Reading Files (line by line)

```
import java.io.FileReader;
import java.io.BufferedReader;

public class Main {
    public static void main(String[] args) {
        FileReader fr = new FileReader("gedicht.txt");
        BufferedReader bufr = new BufferedReader(fr);
        String line;
        while ((line = bufr.readLine()) != null){
            System.out.println(line);
        }
    }
}
```

Reading Files (line by line)

We get the following compilation error:

```
./Main.java:6: error: unreported exception FileNotFoundException;  
must be caught or declared to be thrown
```

```
    FileReader fr = new FileReader("gedicht.txt");  
                    ^
```

```
./Main.java:9: error: unreported exception IOException; must be  
caught or declared to be thrown
```

```
    while ((line = bufr.readLine()) != null){  
            ^
```

2 errors

It seems we need to understand more about the topic “Exceptions”

... therefore ...

8. Errors and Exceptions

Errors, runtime-exceptions, checked-exceptions, exception handling, special case: resources

Errors and Exceptions in Java

Errors and exceptions interrupt the normal execution of the program abruptly and represent an *unplanned event*.



Exceptions are bad, or not?

- Java allows to catch such events and deal with it (as opposed to crashing the entire program)
- Unhandled errors and exceptions are passed up through the call stack.

Errors



This glass is broken for good

Errors happen in the virtual machine of Java and are *not repairable*.

Examples

- No more memory available
- Too high call stack (→ recursion)
- Missing libraries
- Bug in the virtual machine
- Hardware error

Exceptions

Exceptions are triggered by the virtual machine or the program itself and can typically be handled in order to *re-establish the normal situation*



Clean-up and pour in a new glass

Examples

- De-reference `null`
- Division by zero
- Read/write errors (on files)
- Errors in business logic

Exception Types

Runtime Exceptions

- Can happen anywhere
- *Can* be handled
- Cause: bug in the code

Checked Exceptions

- Must be declared
- *Must* be handled
- Cause: Unlikely but not impossible event

Example of a Runtime Exception

```
1 import java.util.Scanner;
2 class ReadTest {
3     public static void main(String[] args){
4         int i = readInt("Number");
5     }
6     private static int readInt(String prompt){
7         System.out.print(prompt + ": ");
8         Scanner input = new Scanner(System.in);
9         return input.nextInt();
10    }
11 }
```

Input: Number: asdf

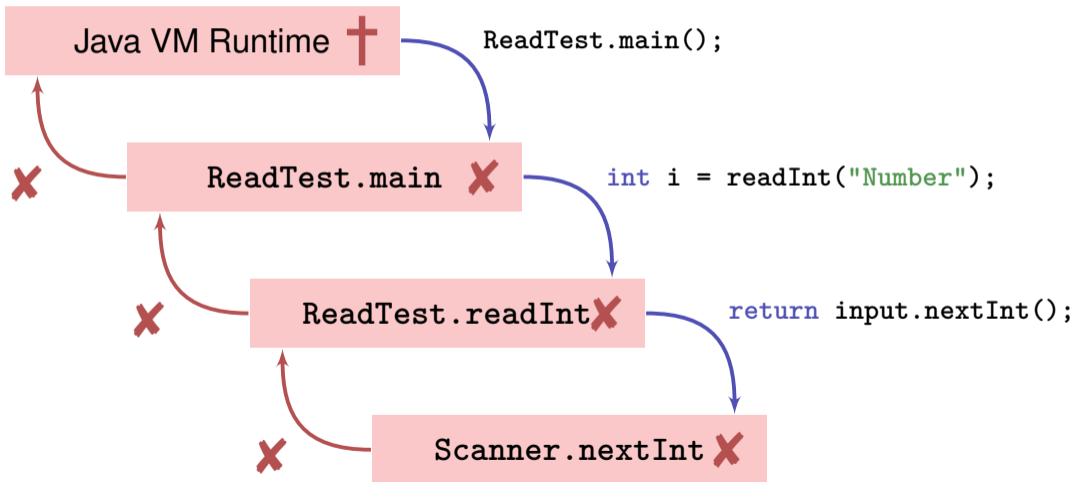
Unhandled Errors and Exceptions

The program crashes and leaves behind a *stack trace*. In there, we can see the where the program got interrupted.

```
Exception in thread "main" java.util.InputMismatchException
    [...]
    at java.util.Scanner.nextInt(Scanner.java:2076)
    at ReadTest.readInt(ReadTest.java:9)
    at ReadTest.main(ReadTest.java:4)
```

⇒ Forensic investigation based on this information.

Exception gets Propagated through Call Stack



Unstanding Stack Traces

Output:

An unsuited input ...

```
Exception in thread "main" java.util.InputMismatchException
at java.util.Scanner.throwFor(Scanner.java:864)
at java.util.Scanner.next(Scanner.java:1485)
at java.util.Scanner.nextInt(Scanner.java:2117)
at java.util.Scanner.nextInt(Scanner.java:2076)
at ReadTest.readInt(ReadTest.java:9)
at ReadTest.main(ReadTest.java:4)
```

... in method `readInt` on line 9 ...

called by method `main` on line 4

Unstanding Stack Traces

```
1 import java.util.Scanner;
2 class ReadTest {
3     public static void main(String[] args){
4         int i = readInt("Number");
5     }
6     private static int readInt(String prompt){
7         System.out.print(prompt + ": ");
8         Scanner input = new Scanner(System.in);
9         return input.nextInt();
10    }
11 }
```

at ReadTest.readInt(ReadTest.java:9)
at ReadTest.main(ReadTest.java:4)

Runtime Exception: Bug in the Code?!

Where is the bug?

```
private static int readInt(String prompt){  
    System.out.print(prompt + ": ");  
    Scanner input = new Scanner(System.in);  
    return input.nextInt();  
}
```



Not guaranteed that the next input is an int

⇒ The scanner class provides a test for this

Runtime Exception: Bug Fix!

Check first!

```
private static int readInt(String prompt){
    System.out.print(prompt + ": ");
    Scanner input = new Scanner(System.in);
    if (input.hasNextInt()){
        return input.nextInt();
    } else {
        return 0; // or do something else ...?!
    }
}
```

First Finding: often no Exceptional Situation

Often, those “exceptional” cases aren’t that unusual, but pretty foreseeable. In those cases *no* exceptions should be used!



Kids are tipping over cups. You get used to it.

Examples

- Wrong credentials when logging in
- Empty required fields in forms
- Unavailable internet resources
- Timeouts

Second Finding: Avoid Exceptions



Problem solved.

Instead of letting a runtime exception happen, *actively prevent* such a situation to arise.

Examples

- Check user inputs early
- Use optional types
- Predict timeout situations
- Plan B for unavailable resources

Exception Types

Runtime Exceptions

- Can happen anywhere
- *Can* be handled
- Cause: bug in the code

Checked Exceptions

- Must be declared
- *Must* be handled
- Cause: Unlikely but not impossible event

Example of a Checked Exception

```
private static String[] readFile(String filename){
    FileReader fr = new FileReader(filename);
    BufferedReader bufr = new BufferedReader(fr);
    ...
    line = bufr.readLine();
    ...
}
```

Compiler Error:

```
./Root/Main.java:9: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
    FileReader fr = new FileReader(filename);
                    ^
```

```
./Root/Main.java:11: error: unreported exception IOException; must be caught or declared to be thrown
    String line = bufr.readLine();
                    ^
```

Quick Look into Javadoc

`readLine`

```
public String readLine()  
            throws IOException
```

Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

Returns:

A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached

Throws:

`IOException` - If an I/O error occurs

See Also:

`Files.readAllLines(java.nio.file.Path, java.nio.charset.Charset)`

Why use Checked Exceptions?

The following situations justify checked exception:

- Fault is *unprobable but not impossible* – and can be fixed by taking suitable measures at runtime.

The caller of a method with a declared checked exception is forced to deal with it – catch it or pass it up.

Handling Exceptions

```
private static String[] readFile(String filename){
```

```
    try{
```

```
        FileReader fr = new FileReader(filename);  
        BufferedReader bufr = new BufferedReader(fr);  
        ...  
        line = bufr.readLine();  
        ...
```

*Protected
scope*

```
    } catch (IOException e){
```

```
        // do some recovery handling
```

*Measures to re-establish the
normal situation*

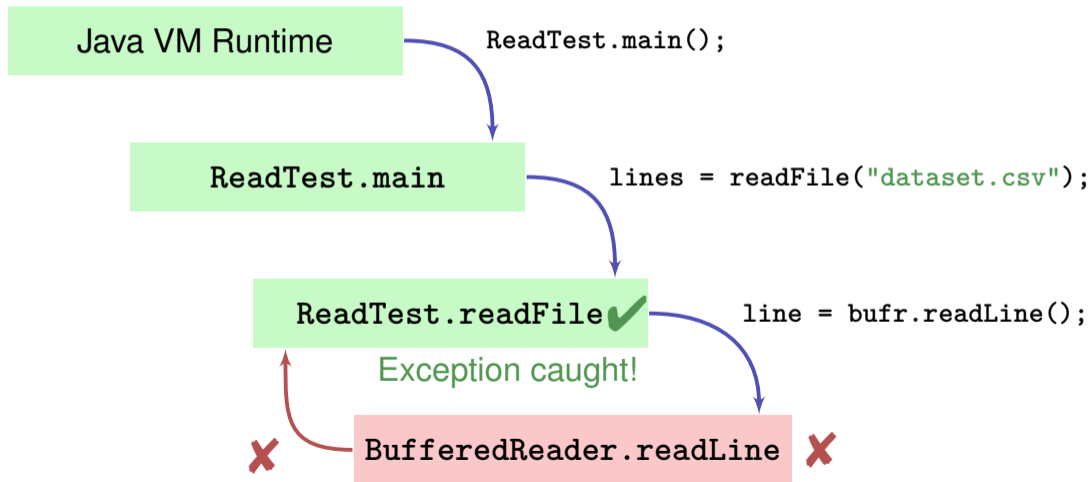
```
    } finally {
```

```
        // close resources
```

*Gets executed in any case, at
the end, always!*

```
    }  
}
```

Handling Exceptions: Stop Propagation!



Finally: Closing Resources

In Java, *resources* must be closed after use at all costs. Otherwise, memory won't get freed.

Resources:

- Files
- Data streams
- UI elements
- ...



Try-With-Resources Statement

Specific syntax to close resources *automatically*:

```
private static String[] readFile(String filename){  
    try (  
        FileReader fr = new FileReader(filename);  
        BufferedReader bufr = new BufferedReader(fr)) {  
        ...  
        line = bufr.readLine();  
        ...  
    } catch (IOException e){  
        // do some recovery handling  
    }  
}
```

Resources get opened here

Resources get closed automatically here

9. Hashing

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

Motivating Example

Goal: Efficient management of a table of all n ETH-students of

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

Dictionary

Abstract Data Type (ADT) D to manage items⁴ i with keys $k \in \mathcal{K}$ with operations

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

⁴Key-value pairs (k, v) , in the following we consider mainly the keys

Dictionaries in Python

dictionary →

```
fruits = {  
    "banana": 2.95, "kiwi": 0.70,  
    "pear": 4.20, "apple": 3.95  
}
```

insert →

```
fruits["melon"] = 3.95
```

update →

```
fruits["banana"] = 1.90
```

find →

```
print("banana", fruits["banana"])  
print("melon in fruits", "melon" in  
fruits)print("onion in fruits"  
 , "onion" in fruits)
```

remove →

```
del fruits["strawberry"]
```

iterate →

```
for name,price in fruits.items():  
    print(name,"->",price)
```

Dictionaries in Java

dictionary → `Map<String,Double> fruits =
new HashMap<String,Double>();`

insert → `fruits.put("banana", 2.95);
fruits.put("kiwi", 0.70);
fruits.put("strawberry", 9.95);
fruits.put("pear", 4.20);
fruits.put("apple", 3.95);`

update → `fruits.put("banana", 2.90);`

find → `Out.println("banana " + fruits.get("banana"));`

remove → `fruits.remove("banana");`

iterate → `for (String s: fruits.keySet())
Out.println(s+ " " + fruits.get(s));`

Motivation / Use

Perhaps *the* most popular data structure.

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

1. Idea: Direct Access Table (Array)

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

Problems

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

Solution to the first problem: Pre-hashing

Prehashing: Map keys to positive integers using a function

$$ph : \mathcal{K} \rightarrow \mathbb{N}$$

- Theoretically always possible because each key is stored as a bit-sequence in the computer
- Theoretically also: $x = y \Leftrightarrow ph(x) = ph(y)$
- Practically: APIs offer functions for pre-hashing. (Java: `object.hashCode()`, C++: `std::hash<>`, Python: `hash(object)`)
- APIs map the key from the key set to an integer with a restricted size.⁵

⁵Therefore the implication $ph(x) = ph(y) \Rightarrow x = y$ does **not** hold any more for all x, y .

Prehashing Example : String

Mapping Name $s = s_1s_2 \dots s_{l_s}$ to key

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

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

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

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

Anna \mapsto 2045632

Jacqueline \mapsto 2042089953442505 $\bmod 2^{32} = 507919049$

Implementation Prehashing (String) in Java

$$ph_{b,m}(s) = \left(\sum_{i=0}^{l-1} s_{l-i+1} \cdot b^i \right) \text{ mod } m$$

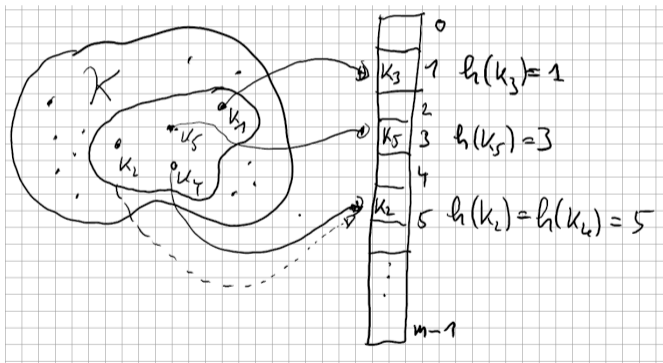
With $b = 31$ and $m = 2^{32}$ we get in Java⁶

```
int prehash(String s){
    int h = 0;
    for (int k = 0; k < s.length(); ++k){
        h = h * b + s.charAt(k);
    }
    return h;
}
```

⁶Try to understand why this works

Lösung zum zweiten Problem: Hashing

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

$$h : \mathcal{K} \rightarrow \{0, 1, \dots, m - 1\}.$$

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

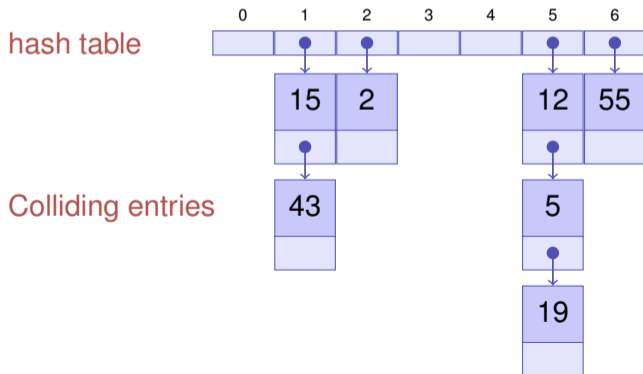
A hash function should map the set of keys as uniformly as possible to the hash table.

Resolving Collisions: Chaining

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

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

Direct Chaining of the Colliding entries



Algorithm for Hashing with Chaining

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

Worst-case Analysis

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

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

Simple Uniform Hashing

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

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

Simple Uniform Hashing

Under the assumption of simple uniform hashing:

Expected length of a chain when n elements are inserted into a hash table with m elements

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

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

Simple Uniform Hashing

Theorem

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

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

Advantages and Disadvantages of Chaining

Advantages

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

Disadvantages

- Memory consumption of the chains-

An Example of a popular Hash Function

Division method

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

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

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

Other method: multiplication method (cf. Cormen et al, Kap. 11.3).

Table size increase

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

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

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

How to choose m' ?

Table size increase

- 1. Idea $n = m \Rightarrow m' \leftarrow m + 1$

Increase for each insertion: Costs $\Theta(1 + 2 + 3 + \dots + n) = \Theta(n^2)$



- 2. Idea $n = m \Rightarrow m' \leftarrow 2m$ Increase only if $m = 2^i$:

$\Theta(1 + 2 + 4 + 8 + \dots + n) = \Theta(n)$

Few insertions cost linear time but on average we have $\Theta(1)$ 😊

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

(\Rightarrow Amortized Analysis)

Amortisierte Analyse

General procedure for dynamic arrays (e.g. Java: `ArrayList`, Python: `List`)

- The data structure provides, besides the data array, two numbers: size of the array (capacity m) and the number of used entries (size n)
- Double the size and copy entries when the list is full $n = m \Rightarrow m \leftarrow 2n$. Kosten $\Theta(m)$.
- Runtime costs for $n = 2^k$ insertion operations:
 $\Theta(1 + 2 + 4 + 8 + \dots + 2^k) = \Theta(2^{k+1} - 1) = \Theta(n)$.

Costs per operation *averaged over all operations* = *amortized costs*
= $\Theta(1)$ per insertion operation

Open Addressing⁷

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

Key table position along a *probing sequence*

$$S(k) := (s(k, 0), s(k, 1), \dots, s(k, m - 1)) \quad \text{mod } m$$

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

⁷Notational clarification: this method uses *open addressing* (meaning that the positions in the hashtable are not fixed) but it is a *closed hashing* procedure (because the entries stay in the hashtable)

Algorithms for open addressing

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

Linear Probing

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

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

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

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

Discussion

Example $\alpha = 0.95$

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

❓ Disadvantage of the method?

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

Quadratic Probing

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

$$S(k) = (h(k), h(k) + 1, h(k) - 1, h(k) + 4, h(k) - 4, \dots) \pmod{m}$$

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

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

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

Discussion

Example $\alpha = 0.95$

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

❓ Problems of this method?

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

Double Hashing

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

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

Example:

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

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

Uniform Hashing

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

(Double hashing is reasonably close)

Analysis of Uniform Hashing with Open Addressing

Theorem

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

Without Proof, cf. e.g. Cormen et al, Kap. 11.4

10. Repetition Binary Search Trees and Heaps

[Ottman/Widmayer, Kap. 2.3, 5.1, Cormen et al, Kap. 6, 12.1 - 12.3]

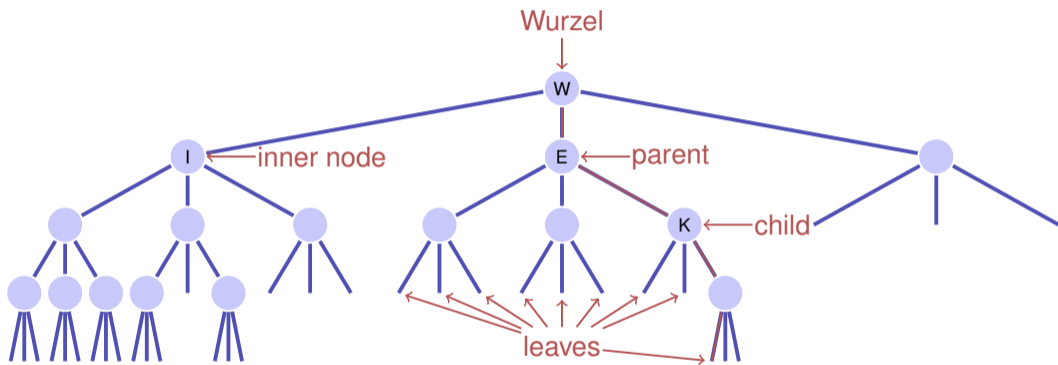
Dictionary implementation

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

Nomenclature



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

Binary Trees

A binary tree is either

- a leaf, i.e. an empty tree, or
- an inner leaf with two trees T_l (left subtree) and T_r (right subtree) as left and right successor.

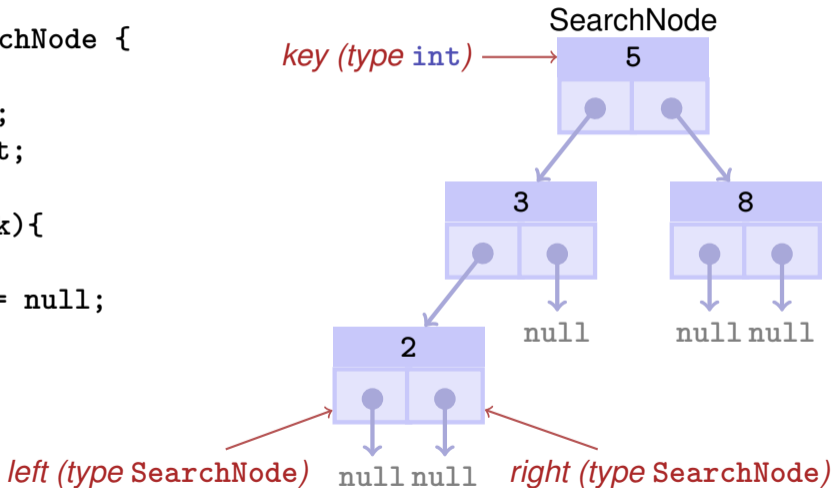
In each node v we store



- a key $v.key$ and
- two nodes $v.left$ and $v.right$ to the roots of the left and right subtree.
- a leaf is represented by the **null**-pointer

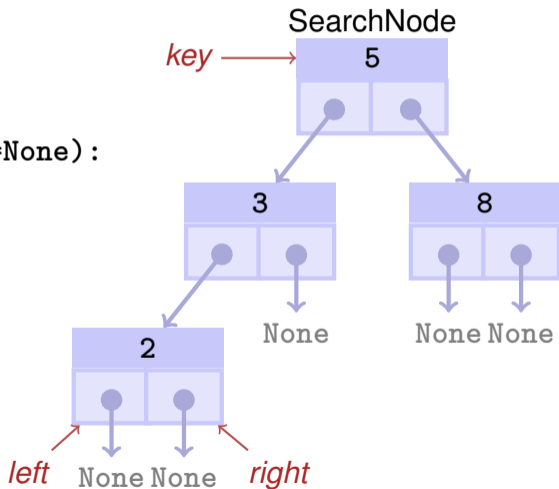
Baumknoten in Java

```
public class SearchNode {  
    int key;  
    SearchNode left;  
    SearchNode right;  
  
    SearchNode(int k){  
        key = k;  
        left = right = null;  
    }  
}
```



Baumknoten in Python

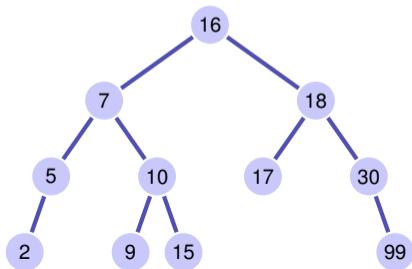
```
class SearchNode:  
    def __init__(self, k, l=None, r=None):  
        self.key = k  
        self.left, self.right = l, r  
        self.flagged = False
```



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 $v.\text{left}$ are smaller than $v.\text{key}$
- Keys in right subtree $v.\text{right}$ are greater than $v.\text{key}$



Searching

Input: Binary search tree with root r , key k

Output: Node v with $v.key = k$ or **null**

$v \leftarrow r$

while $v \neq \text{null}$ **do**

if $k = v.key$ **then**

 | **return** v

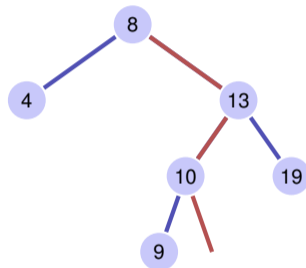
else if $k < v.key$ **then**

 | $v \leftarrow v.left$

else

 | $v \leftarrow v.right$

return null

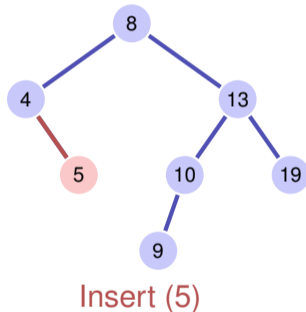


Search (12) \rightarrow **null**

Insertion of a key

Insertion of the key k

- Search for k
- If successful search: output error
- No success: replace the reached leaf by a new node with key

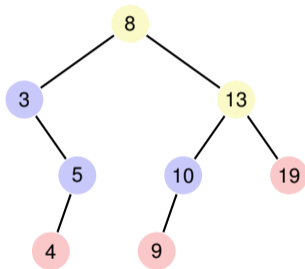


Remove node

Three cases possible:

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

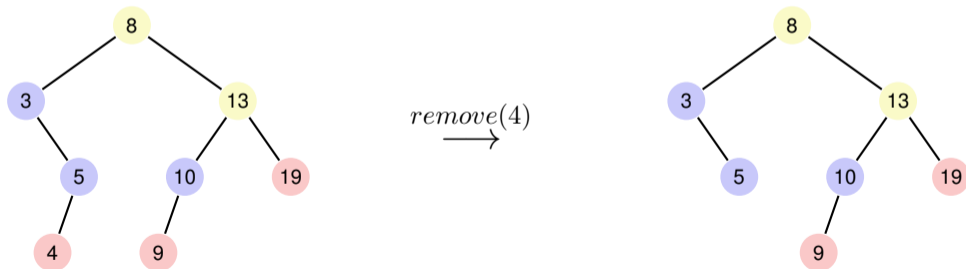
[Leaves do not count here]



Remove node

Node has no children

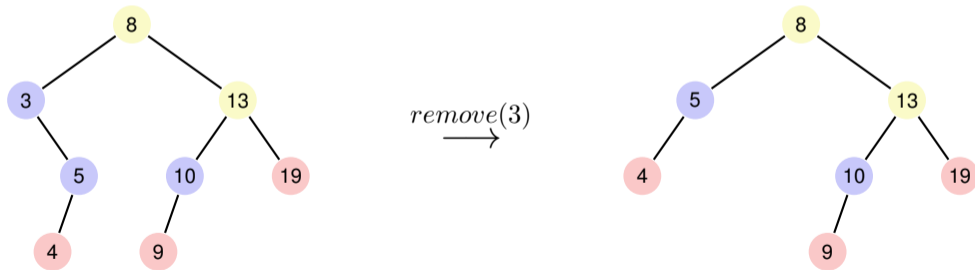
Simple case: replace node by leaf.



Remove node

Node has one child

Also simple: replace node by single child.



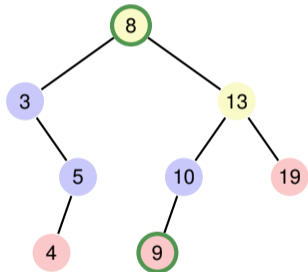
Remove node

Node has two children

The following observation helps: the smallest key in the right subtree $v.\text{right}$ (the *symmetric successor* of v)

- is smaller than all keys in $v.\text{right}$
- is greater than all keys in $v.\text{left}$
- and cannot have a left child.

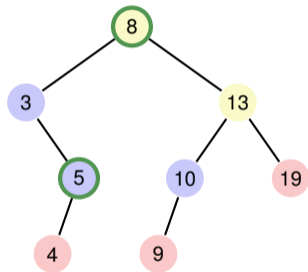
Solution: replace v by its symmetric successor.



By symmetry...

Node 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

$w \leftarrow v.\text{right}$

$x \leftarrow w.\text{left}$

while $x \neq \text{null}$ **do**

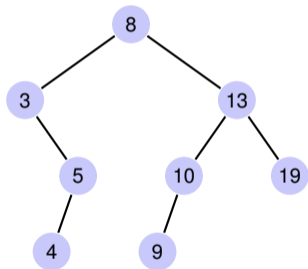
$w \leftarrow x$

$x \leftarrow x.\text{left}$

return w

Traversal possibilities

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



Height of a tree

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

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

The worst case run time of the search is thus $\mathcal{O}(h(T))$

Analysis

Search, Insertion and Deletion of an element v from a tree T requires $\mathcal{O}(h(T))$ fundamental steps in the worst case.

Possible Heights

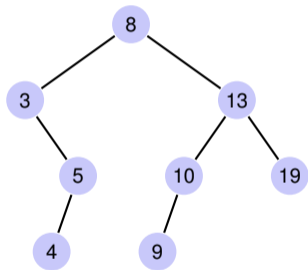
- 1 The maximal height h_n of a tree with n inner nodes is given with $h_1 = 1$ and $h_{n+1} \leq 1 + h_n$ by $h_n \geq n$.
- 2 The minimal height h_n of an (ideally balanced) tree with n inner nodes fulfils $n \leq \sum_{i=0}^{h-1} 2^i = 2^h - 1$.

Thus

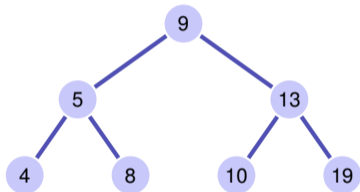
$$\lceil \log_2(n + 1) \rceil \leq h \leq n$$

Further supported operations

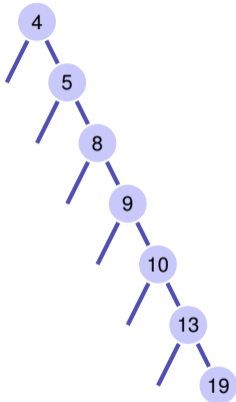
- $\text{Min}(T)$: Read-out minimal value in $\mathcal{O}(h)$
- $\text{ExtractMin}(T)$: Read-out and remove minimal value in $\mathcal{O}(h)$
- $\text{List}(T)$: Output the sorted list of elements
- $\text{Join}(T_1, T_2)$: Merge two trees with $\max(T_1) < \min(T_2)$ in $\mathcal{O}(n)$.



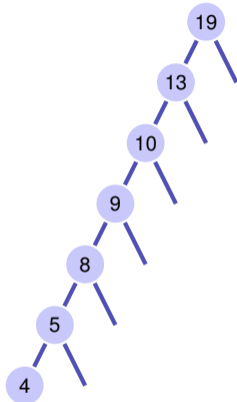
Degenerated search trees



Insert 9,5,13,4,8,10,19
ideally balanced



Insert 4,5,8,9,10,13,19
linear list



Insert 19,13,10,9,8,5,4
linear list

[Probabilistically]

A search tree constructed from a random sequence of numbers provides an an expected path length of $\mathcal{O}(\log n)$.

Attention: this only holds for insertions. If the tree is constructed by random insertions and deletions, the expected path length is $\mathcal{O}(\sqrt{n})$.

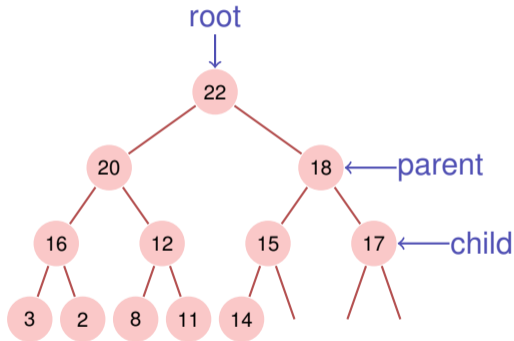
Balanced trees make sure (e.g. with *rotations*) during insertion or deletion that the tree stays balanced and provide a $\mathcal{O}(\log n)$

Worst-case guarantee.

[Max-]Heap⁸

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

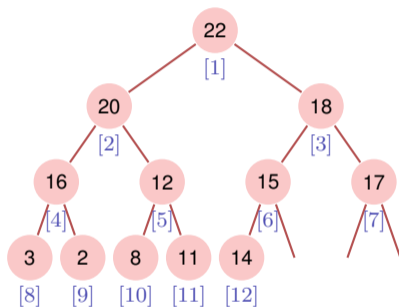
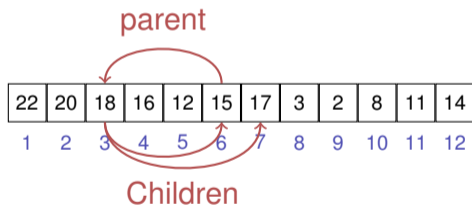


⁸Heap(data structure), not: as in “heap and stack” (memory allocation)

Heap and Array

Tree \rightarrow Array:

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



Depends on the starting index⁹

⁹For array 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

A complete binary tree with height¹⁰ h provides

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

nodes. Thus for a heap with height h :

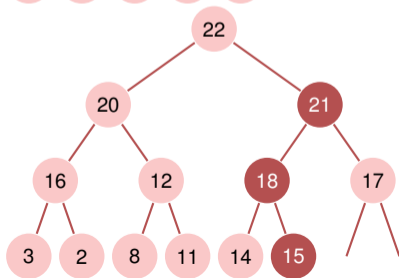
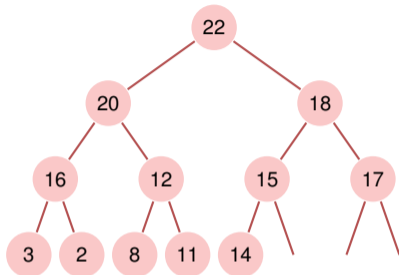
$$\begin{aligned} 2^{h-1} - 1 < n \leq 2^h - 1 \\ \Leftrightarrow 2^{h-1} < n + 1 \leq 2^h \end{aligned}$$

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

¹⁰here: number of edges from the root to a leaf

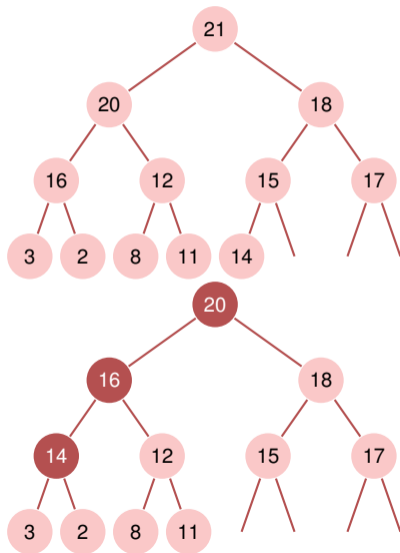
Insert

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



Remove the maximum

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



Algorithm SiftDown(A, i, m)

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

Output: Array A with heap structure for i with last element m .

while $2i \leq m$ **do**

$j \leftarrow 2i$; // j left child

if $j < m$ and $A[j] < A[j + 1]$ **then**

$j \leftarrow j + 1$; // j right child with greater key

if $A[i] < A[j]$ **then**

 swap($A[i], A[j]$)

$i \leftarrow j$; // keep sinking down

else

$i \leftarrow m$; // sift down finished

Sort heap

$A[1, \dots, n]$ is a Heap.

While $n > 1$

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

swap \Rightarrow

siftDown \Rightarrow

swap \Rightarrow

siftDown \Rightarrow

swap \Rightarrow

siftDown \Rightarrow

swap \Rightarrow

siftDown \Rightarrow

swap \Rightarrow



Heap creation

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

Consequence: Induction from below!

Algorithm HeapSort(A, n)

Input: Array A with length n .

Output: A sorted.

// Build the heap.

for $i \leftarrow n/2$ **downto** 1 **do**

└ SiftDown(A, i, n);

// Now A is a heap.

for $i \leftarrow n$ **downto** 2 **do**

└ swap($A[1], A[i]$)

└ SiftDown($A, 1, i - 1$)

// Now A is sorted.

Analysis: sorting a heap

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

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

[Analysis: creating a heap]

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

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

$$\begin{aligned} v(n) &= \sum_{l=0}^{\lfloor \log n \rfloor} \underbrace{2^l}_{\text{number heaps on level } l} \cdot \underbrace{(\lfloor \log n \rfloor - l)}_{\text{height heaps on level } l} = \sum_{k=0}^{\lfloor \log n \rfloor} 2^{\lfloor \log n \rfloor - k} \cdot k \\ &\leq \sum_{k=0}^{\lfloor \log n \rfloor} \frac{n}{2^k} \cdot k = n \cdot \sum_{k=0}^{\lfloor \log n \rfloor} \frac{k}{2^k} \in \mathcal{O}(n) \end{aligned}$$

with $s(x) := \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ ($0 < x < 1$)¹¹ and $s(\frac{1}{2}) = 2$

¹¹ $f(x) = \frac{1}{1-x} = 1 + x + x^2 \dots \Rightarrow f'(x) = \frac{1}{(1-x)^2} = 1 + 2x + \dots$

11. AVL Trees

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

Objective

Searching, insertion and removal of a key in a tree generated from n keys inserted in random order takes expected number of steps $\mathcal{O}(\log_2 n)$.

But worst case $\Theta(n)$ (degenerated tree).

Goal: avoidance of degeneration. Artificial balancing of the tree for each update-operation of a tree.

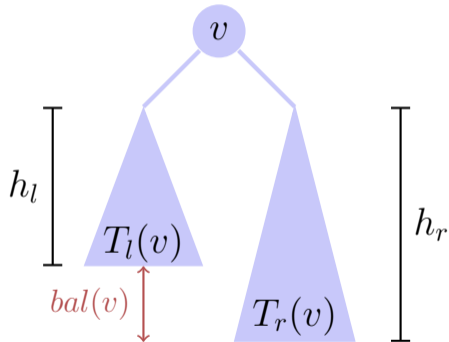
Balancing: guarantee that a tree with n nodes always has a height of $\mathcal{O}(\log n)$.

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

Balance of a node

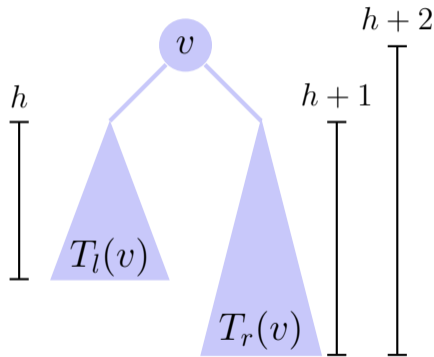
The height *balance* of a node v is defined as the height difference of its sub-trees $T_l(v)$ and $T_r(v)$

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

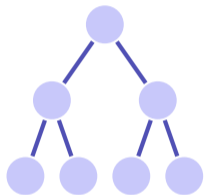


AVL Condition

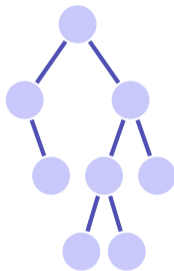
AVL Condition: for each node v of a tree $\text{bal}(v) \in \{-1, 0, 1\}$



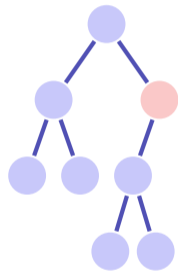
(Counter-)Examples



AVL tree with height
2



AVL tree with height
3

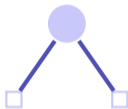


No AVL tree

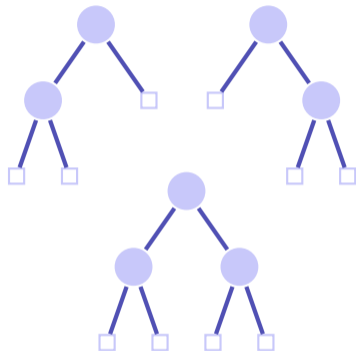
Number of Leaves

- 1. observation: a binary search tree with n keys provides exactly $n + 1$ leaves. Simple induction argument.
 - The binary search tree with $n = 0$ keys has $m = 1$ leaves
 - When a key is added ($n \rightarrow n + 1$), then it replaces a leaf and adds two new leaves ($m \rightarrow m - 1 + 2 = m + 1$).
- 2. observation: a lower bound of the number of leaves in a search tree with given height implies an upper bound of the height of a search tree with given number of keys.

Lower bound of the leaves



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



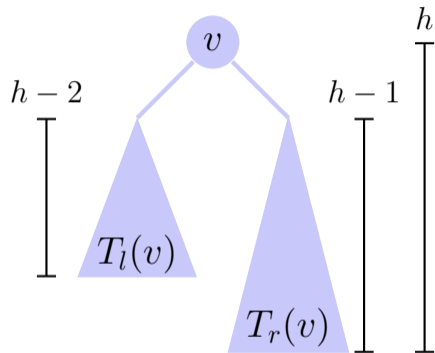
AVL tree with height 2 has
at least $N(2) := 3$ leaves.

Lower bound of the leaves for $h > 2$

- Height of one subtree $\geq h - 1$.
- Height of the other subtree $\geq h - 2$.

Minimal number of leaves $N(h)$ is

$$N(h) = N(h - 1) + N(h - 2)$$



Overall 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

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

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

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

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

[Fibonacci Numbers, Inductive Proof]

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

- 1 Immediate for $i = 0, i = 1$.
- 2 Let $i > 2$ and claim $[*]$ true for all $F_j, j < i$.

$$\begin{aligned} F_i &\stackrel{\text{def}}{=} F_{i-1} + F_{i-2} \stackrel{[*]}{=} \frac{1}{\sqrt{5}}(\phi^{i-1} - \hat{\phi}^{i-1}) + \frac{1}{\sqrt{5}}(\phi^{i-2} - \hat{\phi}^{i-2}) \\ &= \frac{1}{\sqrt{5}}(\phi^{i-1} + \phi^{i-2}) - \frac{1}{\sqrt{5}}(\hat{\phi}^{i-1} + \hat{\phi}^{i-2}) = \frac{1}{\sqrt{5}}\phi^{i-2}(\phi + 1) - \frac{1}{\sqrt{5}}\hat{\phi}^{i-2}(\hat{\phi} + 1) \end{aligned}$$

$(\phi, \hat{\phi} \text{ fulfil } x + 1 = x^2)$

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

Tree Height

Because $|\hat{\phi}| < 1$, overall we have

$$N(h) \in \Theta \left(\left(\frac{1 + \sqrt{5}}{2} \right)^h \right) \subseteq \Omega(1.618^h)$$

and thus

$$\begin{aligned} N(h) &\geq c \cdot 1.618^h \\ \Rightarrow h &\leq 1.44 \log_2 n + c'. \end{aligned}$$

An AVL tree is asymptotically not more than 44% higher than a perfectly balanced tree.¹²

¹²The perfectly balanced tree has a height of $\lceil \log_2 n + 1 \rceil$

Insertion

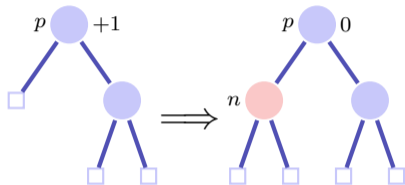
Balance

- Keep the balance stored in each node
- Re-balance the tree in each update-operation

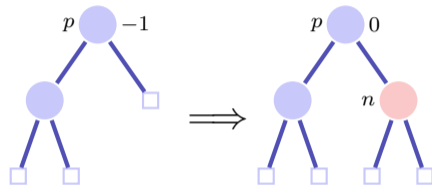
New node n is inserted:

- Insert the node as for a search tree.
- Check the balance condition increasing from n to the root.

Balance at Insertion Point



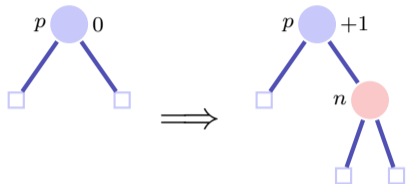
case 1: $\text{bal}(p) = +1$



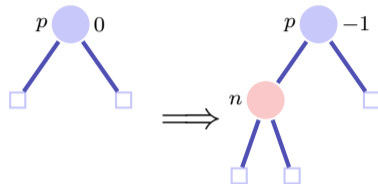
case 2: $\text{bal}(p) = -1$

Finished in both cases because the subtree height did not change

Balance at Insertion Point



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



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

Not finished in both case. Call of `upin(p)`

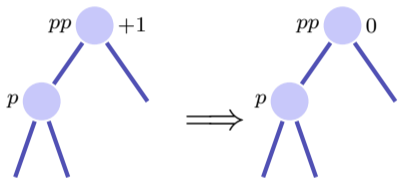
upin(p) - invariant

When `upin(p)` is called it holds that

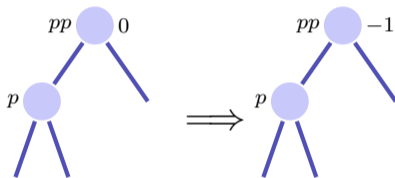
- the subtree from p is grown and
- $\text{bal}(p) \in \{-1, +1\}$

upin(p)

Assumption: p is left son of pp ¹³



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



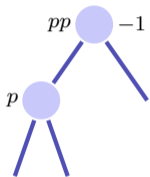
case 2: $\text{bal}(pp) = 0$, **upin(pp)**

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

¹³If p is a right son: symmetric cases with exchange of $+1$ and -1

upin(p)

Assumption: p is left son of pp



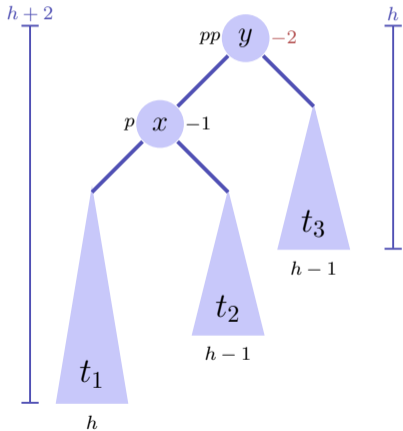
case 3: $\text{bal}(pp) = -1,$

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

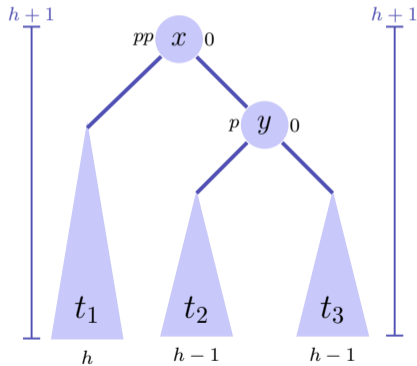
Two cases $\text{bal}(p) = -1, \text{bal}(p) = +1$

Rotations

case 1.1 $\text{bal}(p) = -1$.¹⁴



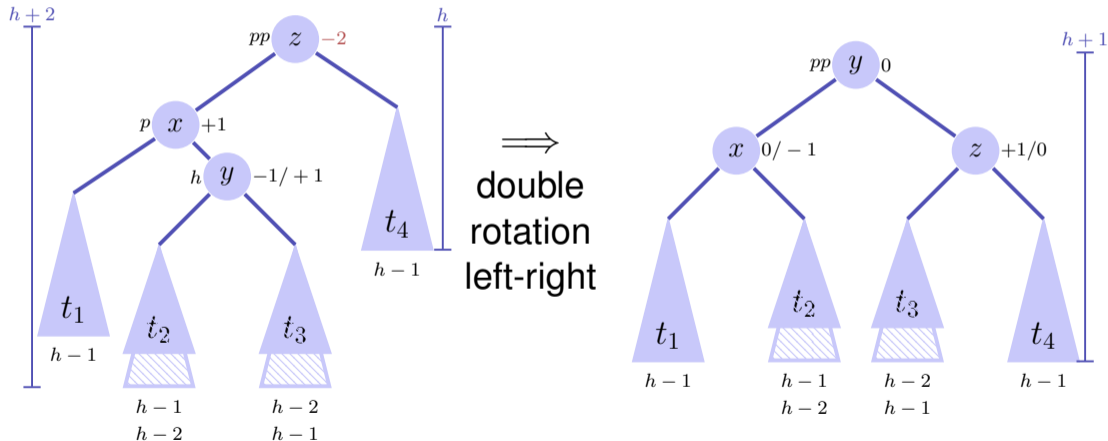
\Rightarrow
rotation
right



¹⁴ p right son: $\Rightarrow \text{bal}(pp) = \text{bal}(p) = +1$, left rotation

Rotations

case 1.1 $\text{bal}(p) = -1$.¹⁵



¹⁵ p right son $\implies \text{bal}(pp) = +1, \text{bal}(p) = -1$, double rotation right left

Analysis

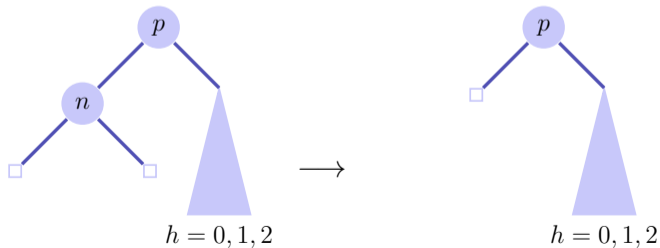
- Tree height: $\mathcal{O}(\log n)$.
- Insertion like in binary search tree.
- Balancing via recursion from node to the root. Maximal path length $\mathcal{O}(\log n)$.

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 .

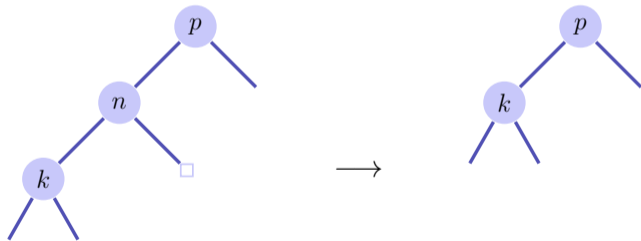
- $h' = 1$: Adapt $\text{bal}(p)$.
- $h' = 0$: Adapt $\text{bal}(p)$. Call $\text{upout}(p)$.
- $h' = 2$: Rebalanciere des Teilbaumes. Call $\text{upout}(p)$.



Deletion

Case 2: one child k of node n is an inner node

- Replace n by k . `upout(k)`



Deletion

Case 3: both children of node n are inner nodes

- Replace n by symmetric successor. `upout(k)`
- Deletion of the symmetric successor is as in case 1 or 2.

upout (p)

Let pp be the parent node of p .

(a) p left child of pp

1 $\text{bal}(pp) = -1 \Rightarrow \text{bal}(pp) \leftarrow 0$. **upout (pp)**

2 $\text{bal}(pp) = 0 \Rightarrow \text{bal}(pp) \leftarrow +1$.

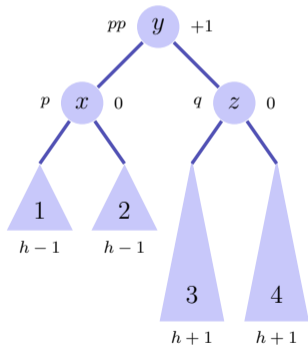
3 $\text{bal}(pp) = +1 \Rightarrow$ next slides.

(b) p right child of pp : Symmetric cases exchanging $+1$ and -1 .

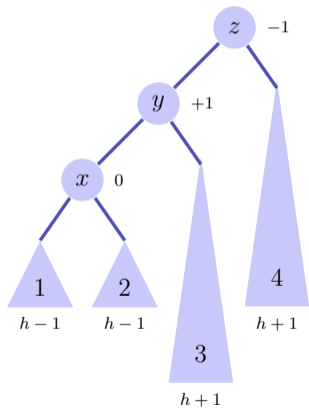
upout (p)

Case (a).3: $\text{bal}(pp) = +1$. Let q be brother of p

(a).3.1: $\text{bal}(q) = 0$.¹⁶



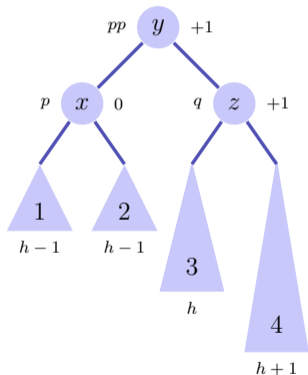
\implies
Left Rotate(y)



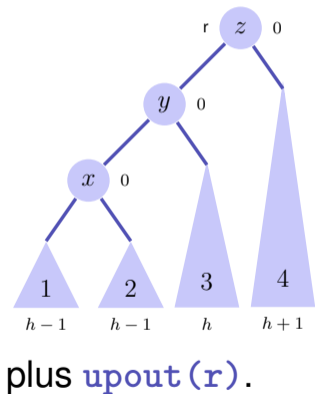
¹⁶(b).3.1: $\text{bal}(pp) = -1$, $\text{bal}(q) = -1$, Right rotation

upout (p)

Case (a).3: $\text{bal}(pp) = +1$. (a).3.2: $\text{bal}(q) = +1$.¹⁷



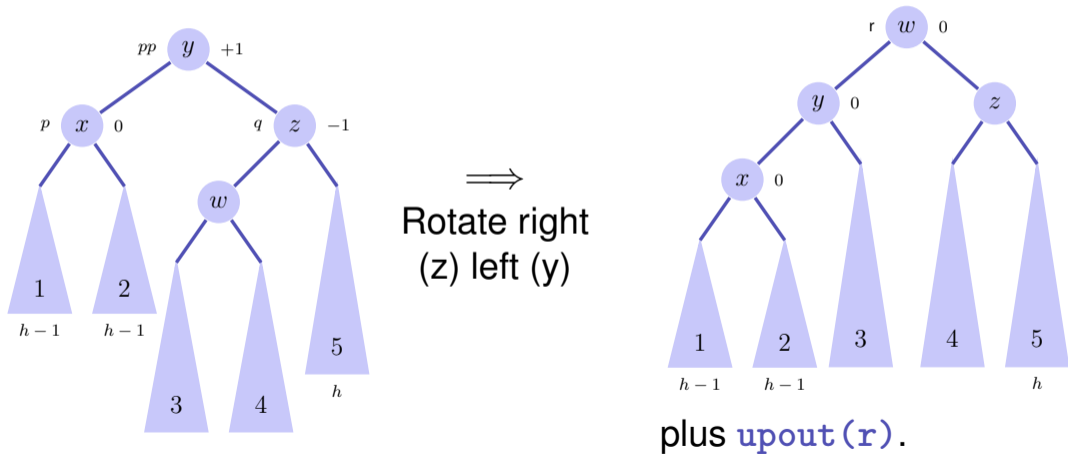
\implies
Left Rotate(y)



¹⁷(b).3.2: $\text{bal}(pp) = -1$, $\text{bal}(q) = +1$, Right rotation+upout

upout (p)

Case (a).3: $\text{bal}(pp) = +1$. (a).3.3: $\text{bal}(q) = -1$.¹⁸



¹⁸(b).3.3: $\text{bal}(pp) = -1$, $\text{bal}(q) = -1$, left-right rotation + upout

Conclusion

- AVL trees have worst-case asymptotic runtimes of $\mathcal{O}(\log n)$ for searching, insertion and deletion of keys.
- Insertion and deletion is relatively involved and an overkill for really small problems.

12. Dynamic Programming

Memoization, Optimal Substructure, Overlapping Sub-Problems, Dependencies, General Procedure. Examples: Rod Cutting, Rabbits, Edit Distance

[Ottman/Widmayer, Kap. 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 is 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

$f \leftarrow \text{FibonacciRecursive}(n - 1) + \text{FibonacciRecursive}(n - 2)$

return f

Analysis

$T(n)$: Number executed operations.

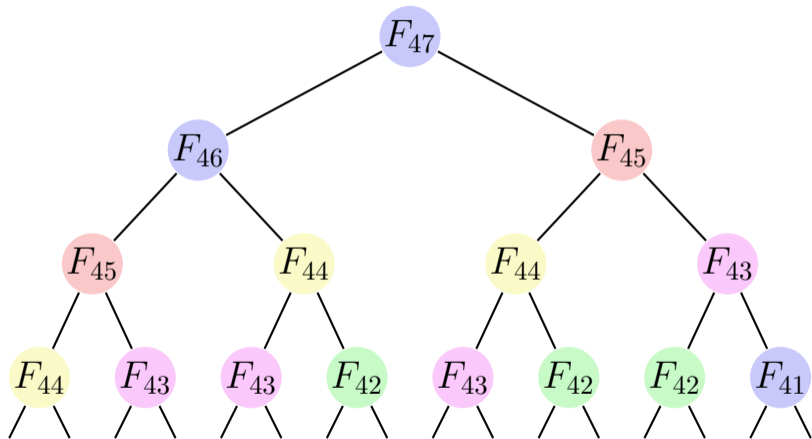
■ $n = 0, 1: T(n) = \Theta(1)$

■ $n \geq 2: T(n) = T(n - 2) + T(n - 1) + c.$

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

Algorithm is *exponential* in n .

Reason (visual)



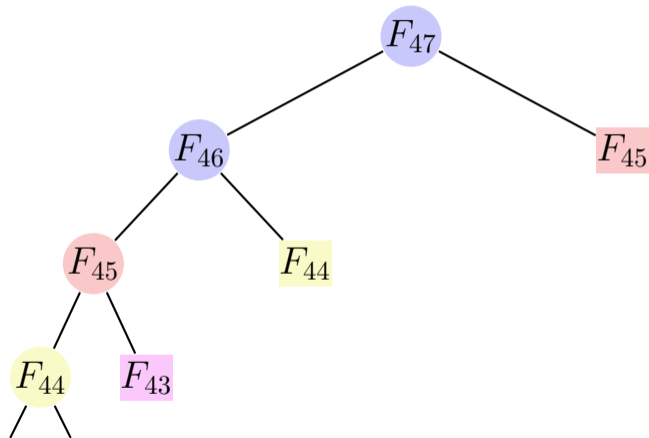
Nodes with same values are evaluated (too) often.

Memoization

Memoization (sic) saving intermediate results.

- Before a subproblem is solved, the existence of the corresponding intermediate result is checked.
- If an intermediate result exists then it is used.
- Otherwise the algorithm is executed and the result is saved accordingly.

Memoization with Fibonacci



Rechteckige Knoten wurden bereits ausgewertet.

Algorithm FibonacciMemoization(n)

Input: $n \geq 0$

Output: n -th Fibonacci number

if $n \leq 2$ **then**

| $f \leftarrow 1$

else if $\exists \text{memo}[n]$ **then**

| $f \leftarrow \text{memo}[n]$

else

| $f \leftarrow \text{FibonacciMemoization}(n - 1) + \text{FibonacciMemoization}(n - 2)$

| $\text{memo}[n] \leftarrow f$

return f

Analysis

Computational complexity:

$$T(n) = T(n - 1) + c = \dots = \mathcal{O}(n).$$

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.¹⁹

¹⁹But the naive recursive algorithm also requires $\Theta(n)$ memory implicitly.

Looking closer ...

... the algorithm computes the values of F_1, F_2, F_3, \dots 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

⇒ Results are saved

Arbeitsspeicher



192.-

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

★★★★★ 16

We trade speed against
memory consumption

Dynamic Programming: Description

- 1 Use a *DP-table* with information to the subproblems.
Dimension of the entries? Semantics of the entries?
- 2 Computation of the *base cases*
Which entries do not depend on others?
- 3 Determine *computation order*.
In which order can the entries be computed such that dependencies are fulfilled?
- 4 Read-out the *solution*
How can the solution be read out from the table?

Runtime (typical) = number entries of the table times required operations per entry.

Dynamic Programming: Description with the example

- 1 Dimension of the table? Semantics of the entries?
 $n \times 1$ table. n th entry contains n th Fibonacci number.
- 2 Which entries do not depend on other entries?
Values F_1 and F_2 can be computed easily and independently.
- 3 What is the execution order such that required entries are always available?
 F_i with increasing i .
- 4 Wie kann sich Lösung aus der Tabelle konstruieren lassen?
 F_n ist die n -te Fibonacci-Zahl.

Dynamic Programming = Divide-And-Conquer ?

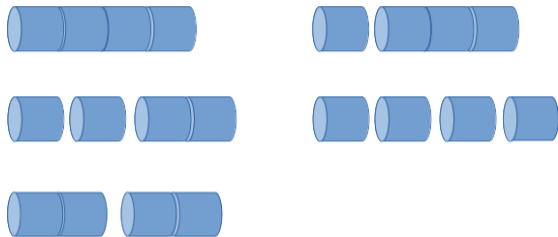
- In both cases the original problem can be solved (more easily) by utilizing the solutions of sub-problems. The problem provides *optimal substructure*.
- Divide-And-Conquer algorithms (such as Mergesort): sub-problems are independent; their solutions are required only once in the algorithm.
- DP: sub-problems are dependent. The problem is said to have *overlapping sub-problems* that are required multiple-times in the algorithm.
- In order to avoid redundant computations, results are tabulated. For *sub-problems there must not be any circular dependencies*.

Rod Cutting

- Rods (metal sticks) are cut and sold.
- Rods of length $n \in \mathbb{N}$ are available. A cut does not provide any costs.
- For each length $l \in \mathbb{N}$, $l \leq n$ known is the value $v_l \in \mathbb{R}^+$
- Goal: cut the rods such (into $k \in \mathbb{N}$ pieces) that

$$\sum_{i=1}^k v_{l_i} \text{ is maximized subject to } \sum_{i=1}^k l_i = n.$$

Rod Cutting: Example



Possibilities to cut a rod of length 4 (without permutations)

Length	0	1	2	3	4
Price	0	2	3	8	9

\Rightarrow Best cut: 3 + 1 with value 10.

Wie findet man den DP Algorithms

- 0 Exact formulation of the wanted solution
- 1 Define sub-problems (and compute the cardinality)
- 2 Guess / Enumerate (and determine the running time for guessing)
- 3 Recursion: relate sub-problems
- 4 Memoize / Tabularize. Determine the dependencies of the sub-problems
- 5 Solve the problem
Running time = #sub-problems \times time/sub-problem

Structure of the problem

- 0 **Wanted:** r_n = maximal value of rod (cut or as a whole) with length n .
- 1 **sub-problems:** maximal value r_k for each $0 \leq k < n$
- 2 **Guess** the length of the first piece
- 3 **Recursion**

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

- 4 **Dependency:** r_k depends (only) on values v_i , $1 \leq i \leq k$ and the optimal cuts r_i , $i < k$
- 5 **Solution** in r_n

Algorithm RodCut(v, n)

Input: $n \geq 0$, Prices v

Output: best value

$q \leftarrow 0$

if $n > 0$ **then**

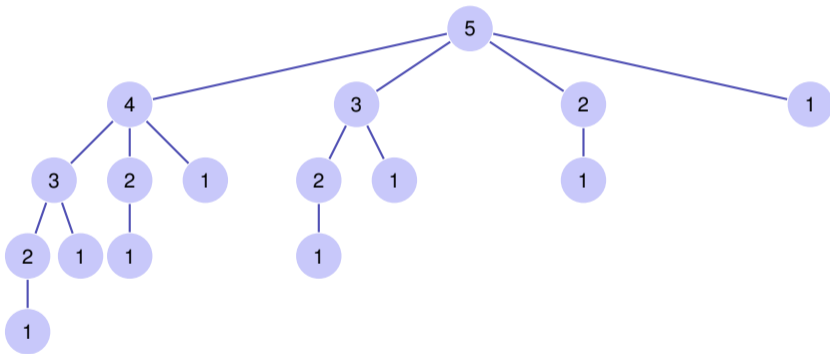
for $i \leftarrow 1, \dots, n$ **do**
 $q \leftarrow \max\{q, v_i + \text{RodCut}(v, n - i)\};$

return q

Running time $T(n) = \sum_{i=0}^{n-1} T(i) + c \Rightarrow^{20} T(n) \in \Theta(2^n)$

²⁰ $T(n) = T(n-1) + \sum_{i=0}^{n-2} T(i) + c = T(n-1) + (T(n-1) - c) + c = 2T(n-1) \quad (n > 0)$

Recursion Tree



Algorithm RodCutMemoized(m, v, n)

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

Output: best value

$q \leftarrow 0$

if $n > 0$ **then**

if $\exists m[n]$ **then**

$q \leftarrow m[n]$

else

for $i \leftarrow 1, \dots, n$ **do**

$q \leftarrow \max\{q, v_i + \text{RodCutMemoized}(m, v, n - i)\};$

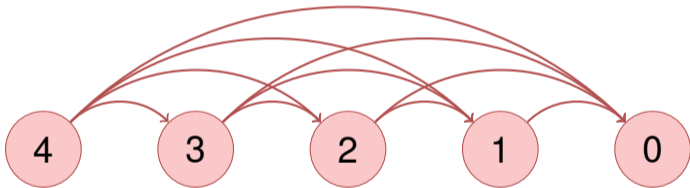
$m[n] \leftarrow q$

return q

Running time $\sum_{i=1}^n i = \Theta(n^2)$

Subproblem-Graph

Describes the mutual dependencies of the subproblems



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 length of the first rod in a separate table of length n

Bottom-up Description with the example

1 Dimension of the table? Semantics of the entries?

$n \times 1$ table. n th entry contains the best value of a rod of length n .

2 Which entries do not depend on other entries?

Value r_0 is 0

3 What is the execution order such that required entries are always available?

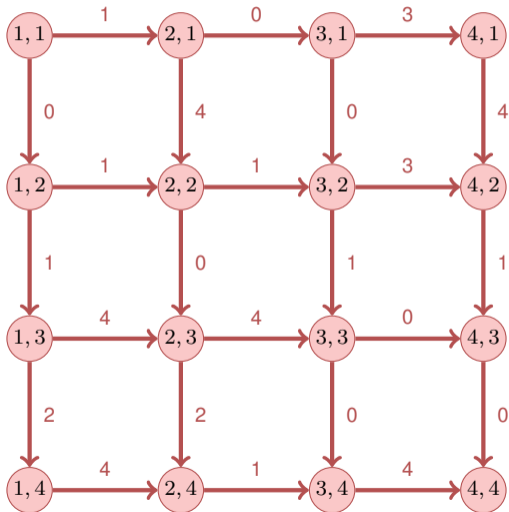
$r_i, i = 1, \dots, n$.

4 Wie kann sich Lösung aus der Tabelle konstruieren lassen?

r_n is the best value for the rod of length n .

Rabbit!

A rabbit sits on cite $(1, 1)$ of an $n \times n$ grid. It can only move to east or south. On each pathway there is a number of carrots. How many carrots does the rabbit collect maximally?



Rabbit!

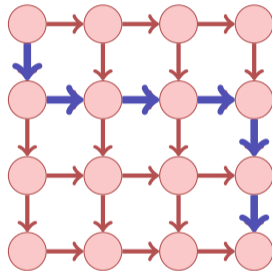
Number of possible paths?

- Choice of $n - 1$ ways to south out of $2n - 2$ ways overall.



$$\binom{2n - 2}{n - 1} \in \Omega(2^n)$$

⇒ No chance for a naive algorithm



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

Recursion

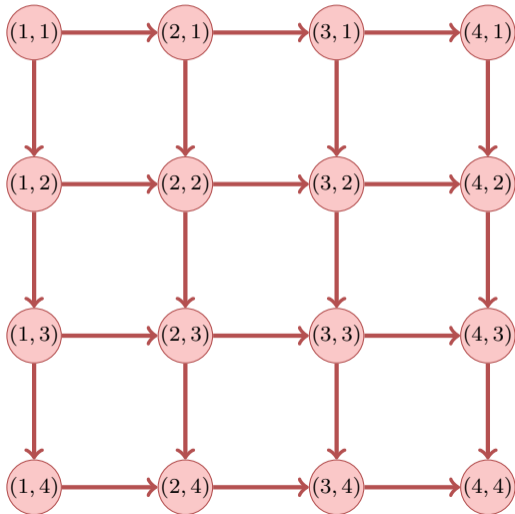
Wanted: $T_{0,0} = \text{maximal number carrots from } (0,0) \text{ to } (n,n)$.

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

Dimension of the table? Semantics of the entries?

- 1 Table T with size $n \times n$. Entry at i, j provides the maximal number of carrots from (i, j) to (n, n) .

Which entries do not depend on other entries?

- 2 Value $T_{n,n}$ is 0

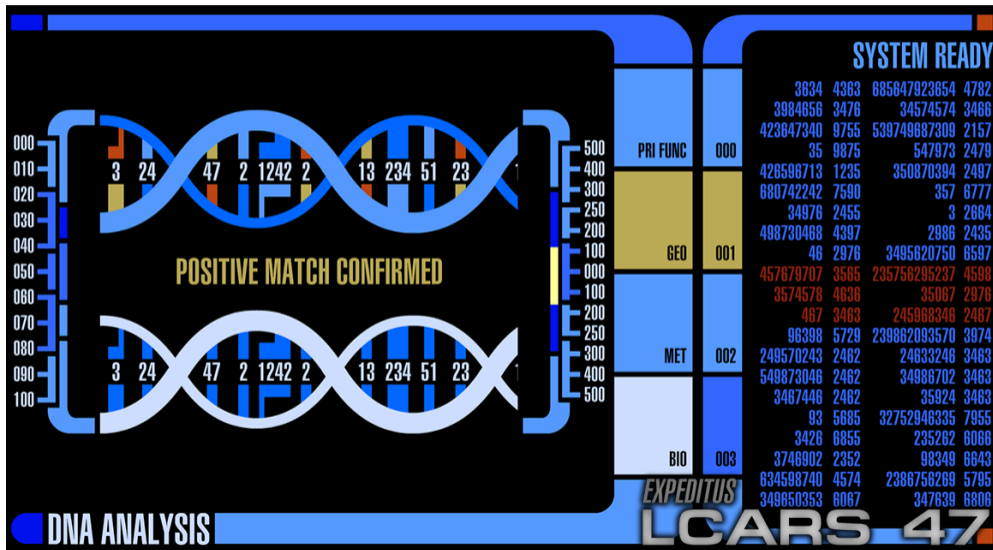
What is the execution order such that required entries are always available?

- 3 $T_{i,j}$ with $i = n \searrow 1$ and for each $i: j = n \searrow 1$, (or vice-versa: $j = n \searrow 1$ and for each $j: i = n \searrow 1$).

Wie kann sich Lösung aus der Tabelle konstruieren lassen?

- 4 $T_{1,1}$ provides the maximal number of carrots.

DNA - Comparison (Star Trek)



DNA - Comparison

- DNA consists of sequences of four different nucleotides **A**denine **G**uanine **T**hymine **C**ytosine
- DNA sequences (genes) thus can be described with strings of A, G, T and C.
- Possible comparison of two genes: determine the **longest common subsequence**

The longest common subsequence problem is a special case of the minimal edit distance problem. The following slides are therefore not presented in the lectures.

[Longest common subsequence]

Subsequences of a string:

Subsequences(*KUH*): $()$, (K) , (U) , (H) , (KU) , (KH) , (UH) ,
 (KUH)

Problem:

- **Input:** two strings $A = (a_1, \dots, a_m)$, $B = (b_1, \dots, b_n)$ with lengths $m > 0$ and $n > 0$.
- **Wanted:** Longest common subsequences (LCS) of A and B .

[Longest Common Subsequence]

Examples:

$LGT(IGEL, KATZE) = E$, $LGT(TIGER, ZIEGE) = IGE$

Ideas to solve?

T	I		G	E	R
Z	I	E	G	E	

[Recursive Procedure]

Assumption: solutions $L(i, j)$ known for $A[1, \dots, i]$ and $B[1, \dots, j]$ for all $1 \leq i \leq m$ and $1 \leq j \leq n$, but not for $i = m$ and $j = n$.

T I G E R
Z I E G E

Consider characters a_m, b_n . Three possibilities:

- 1 A is enlarged by one whitespace. $L(m, n) = L(m, n - 1)$
- 2 B is enlarged by one whitespace. $L(m, n) = L(m - 1, n)$
- 3 $L(m, n) = L(m - 1, n - 1) + \delta_{mn}$ with $\delta_{mn} = 1$ if $a_m = b_n$ and $\delta_{mn} = 0$ otherwise

[Recursion]

$$L(m, n) \leftarrow \max \{L(m - 1, n - 1) + \delta_{mn}, L(m, n - 1), L(m - 1, n)\}$$

for $m, n > 0$ and base cases $L(\cdot, 0) = 0, L(0, \cdot) = 0$.

	\emptyset	Z	I	E	G	E
\emptyset	0	0	0	0	0	0
T	0	0	0	0	0	0
I	0	0	1	1	1	1
G	0	0	1	1	2	2
E	0	0	1	2	2	3
R	0	0	1	2	2	3

[Dynamic Programming algorithm LCS]

Dimension of the table? Semantics?

- 1 Table $L[0, \dots, m][0, \dots, n]$. $L[i, j]$: length of a LCS of the strings (a_1, \dots, a_i) and (b_1, \dots, b_j)

Computation of an entry

- 2 $L[0, i] \leftarrow 0 \forall 0 \leq i \leq m$, $L[j, 0] \leftarrow 0 \forall 0 \leq j \leq n$. Computation of $L[i, j]$ otherwise via $L[i, j] = \max(L[i-1, j-1] + \delta_{ij}, L[i, j-1], L[i-1, j])$.

[Dynamic Programming algorithm LCS]

Computation order

- 3 Rows increasing and within columns increasing (or the other way round).

Reconstruct solution?

- 4 Start with $j = m, i = n$. If $a_i = b_j$ then output a_i and continue with $(j, i) \leftarrow (j - 1, i - 1)$; otherwise, if $L[i, j] = L[i, j - 1]$ continue with $j \leftarrow j - 1$ otherwise, if $L[i, j] = L[i - 1, j]$ continue with $i \leftarrow i - 1$.
Terminate for $i = 0$ or $j = 0$.

[Analysis LCS]

- Number table entries: $(m + 1) \cdot (n + 1)$.
- Constant number of assignments and comparisons each. Number steps: $\mathcal{O}(mn)$
- Determination of solution: decrease i or j . Maximally $\mathcal{O}(n + m)$ steps.

Runtime overall:

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

Minimal Editing Distance

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

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 $A_n \rightarrow B_m$ with costs

operation	Levenshtein	LCS ²¹	general
Insert c	1	1	ins(c)
Delete c	1	1	del(c)
Replace $c \rightarrow c'$	$\mathbb{1}(c \neq c')$	$\infty \cdot \mathbb{1}(c \neq c')$	repl(c, c')

Beispiel

T I G E R
Z I E G E

T I _ G E R
Z I E G E _

T \rightarrow Z +E -R
Z \rightarrow T -E +R

²¹Longest common subsequence – A special case of an editing problem

DP

0 $E(n, m)$ = minimum number edit operations (ED cost)

$$a_{1\dots n} \rightarrow b_{1\dots m}$$

1 Subproblems $E(i, j)$ = ED von $a_{1\dots i}$ $b_{1\dots j}$.

$$\#SP = n \cdot m$$

2 Guess

$$\text{Costs } \Theta(1)$$

■ $a_{1\dots i} \rightarrow a_{1\dots i-1}$ (delete)

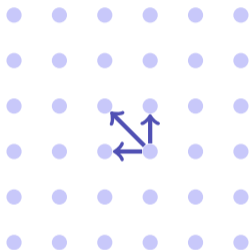
■ $a_{1\dots i} \rightarrow a_{1\dots i}b_j$ (insert)

■ $a_{1\dots i} \rightarrow a_{1\dots i_1}b_j$ (replace)

3 Rekursion

$$E(i, j) = \min \begin{cases} \text{del}(a_i) + E(i-1, j), \\ \text{ins}(b_j) + E(i, j-1), \\ \text{repl}(a_i, b_j) + E(i-1, j-1) \end{cases}$$

4 Dependencies



⇒ Computation from left top to bottom right. Row- or column-wise.

5 Solution in $E(n, m)$

Example (Levenshtein Distance)

$$E[i, j] \leftarrow \min \{ E[i-1, j] + 1, E[i, j-1] + 1, E[i-1, j-1] + \mathbb{1}(a_i \neq b_j) \}$$

	\emptyset	Z	I	E	G	E
\emptyset	0	1	2	3	4	5
T	1	1	2	3	4	5
I	2	2	1	2	3	4
G	3	3	2	2	2	3
E	4	4	3	2	3	2
R	5	5	4	3	3	3

Editing steps: from bottom right to top left, following the recursion.
Bottom-Up description of the algorithm: exercise

Bottom-Up DP algorithm ED]

Dimension of the table? Semantics?

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

Computation of an entry

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

Bottom-Up DP algorithm ED

Computation order

- 3 Rows increasing and within columns increasing (or the other way round).

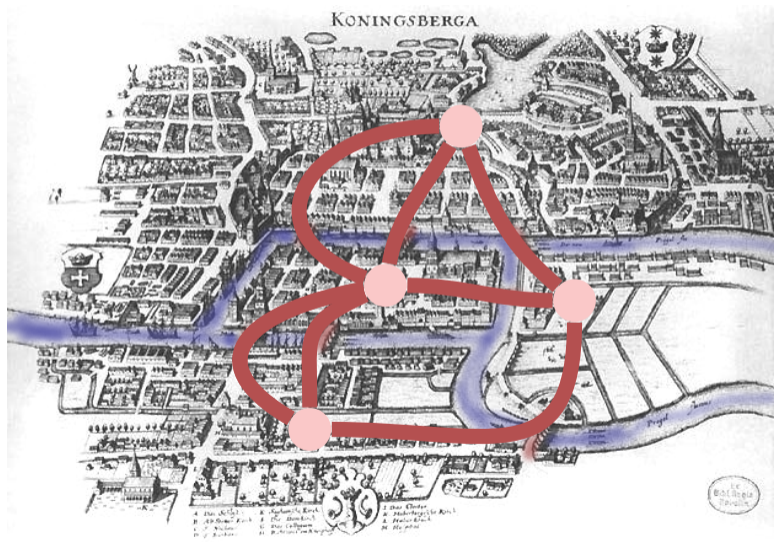
Reconstruct solution?

- 4 Start with $j = m, i = n$. If $E[i, j] = \text{repl}(a_i, b_j) + E(i - 1, j - 1)$ then output $a_i \rightarrow b_j$ and continue with $(j, i) \leftarrow (j - 1, i - 1)$; otherwise, if $E[i, j] = \text{del}(a_i) + E(i - 1, j)$ output $\text{del}(a_i)$ and continue with $j \leftarrow j - 1$ otherwise, if $E[i, j] = \text{ins}(b_j) + E(i, j - 1)$, continue with $i \leftarrow i - 1$.
Terminate for $i = 0$ and $j = 0$.

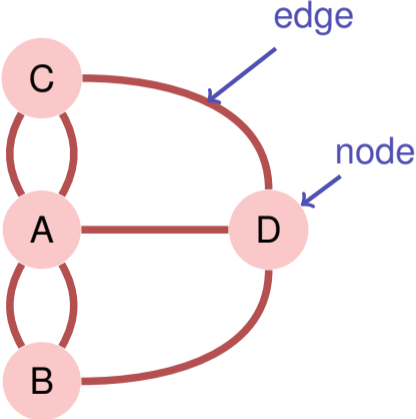
13. Graphs

Notation, Representation, Graph Traversal (DFS, BFS), Topological Sorting [Ottman/Widmayer, Kap. 9.1 - 9.4, Cormen et al, Kap. 22]

Königsberg 1736



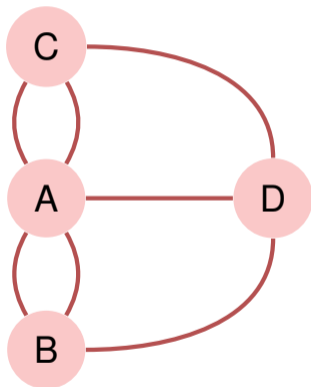
[Multi]Graph



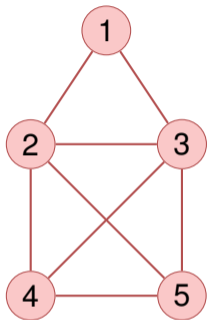
Cycles

- Is there a cycle through the town (the graph) that uses each bridge (each edge) exactly once?
- Euler (1736): no.
- Such a *cycle* is called *Eulerian path*.
- Eulerian path \Leftrightarrow each node provides an even number of edges (each node is of an *even degree*).

' \Rightarrow ' is straightforward, " \Leftarrow " is a bit more difficult but still elementary.

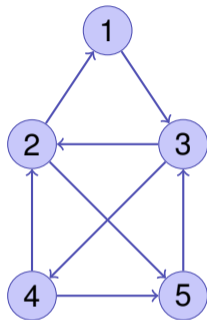


Notation



undirected

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

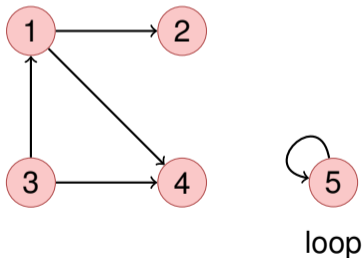


directed

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

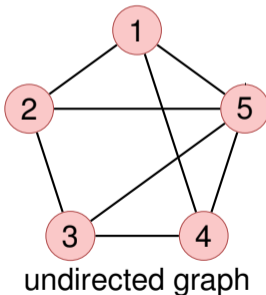
Notation

A *directed graph* consists of a set $V = \{v_1, \dots, v_n\}$ of nodes (*Vertices*) and a set $E \subseteq V \times V$ of Edges. The same edges may not be contained more than once.



Notation

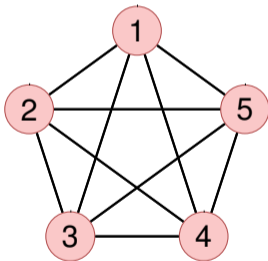
An *undirected graph* consists of a set $V = \{v_1, \dots, v_n\}$ of nodes and a set $E \subseteq \{\{u, v\} \mid u, v \in V\}$ of edges. Edges may not be contained more than once.²²



²²As opposed to the introductory example – it is then called multi-graph.

Notation

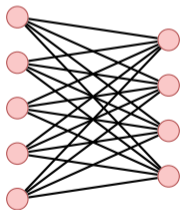
An undirected graph $G = (V, E)$ without loops where E comprises all edges between pairwise different nodes is called *complete*.



a complete undirected graph

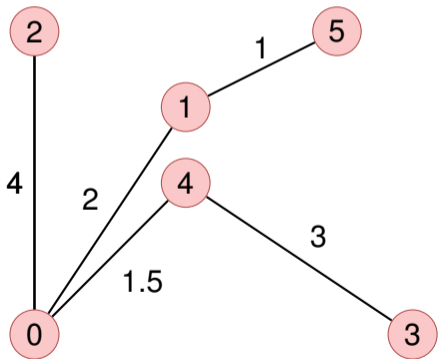
Notation

A graph where V can be partitioned into disjoint sets U and W such that each $e \in E$ provides a node in U and a node in W is called *bipartite*.



Notation

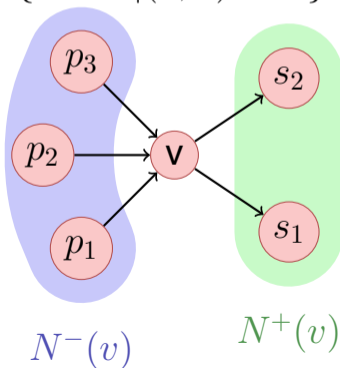
A *weighted graph* $G = (V, E, c)$ is a graph $G = (V, E)$ with an *edge weight function* $c : E \rightarrow \mathbb{R}$. $c(e)$ is called *weight* of the edge e .



Notation

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

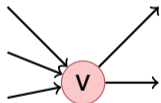
- $w \in V$ is called adjacent to $v \in V$, if $(v, w) \in E$
- **Predecessors** of $v \in V$: $N^-(v) := \{u \in V \mid (u, v) \in E\}$.
Successors: $N^+(v) := \{u \in V \mid (v, u) \in E\}$



Notation

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

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



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

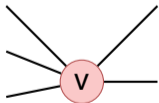


$$\deg^-(w) = 1, \deg^+(w) = 1$$

Notation

For undirected graphs $G = (V, E)$:

- $w \in V$ is called *adjacent* to $v \in V$, if $\{v, w\} \in E$
- *Neighbourhood* of $v \in V$: $N(v) = \{w \in V | \{v, w\} \in E\}$
- *Degree* of v : $\deg(v) = |N(v)|$ with a special case for the loops: increase the degree by 2.



$$\deg(v) = 5$$



$$\deg(w) = 2$$

Relationship between node degrees and number of edges

For each graph $G = (V, E)$ it holds

- 1 $\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$, for G directed
- 2 $\sum_{v \in V} \deg(v) = 2|E|$, for G undirected.

Paths

- **Path**: a sequence of nodes $\langle v_1, \dots, v_{k+1} \rangle$ such that for each $i \in \{1 \dots k\}$ there is an edge from v_i to v_{i+1} .
- **Length** of a path: number of contained edges k .
- **Weight** of a path (in weighted graphs): $\sum_{i=1}^k c((v_i, v_{i+1}))$ (bzw. $\sum_{i=1}^k c(\{v_i, v_{i+1}\})$)
- **Simple path**: path without repeating vertices

Connectedness

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

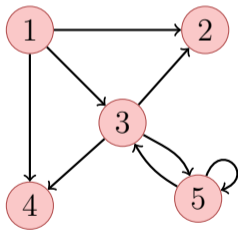
Cycles

- **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 $v_1 \dots, v_n$ stored as *adjacency matrix* $A_G = (a_{ij})_{1 \leq i, j \leq n}$ with entries from $\{0, 1\}$. $a_{ij} = 1$ if and only if edge from v_i to v_j .

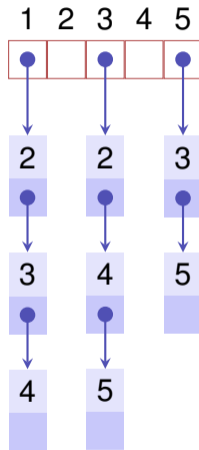
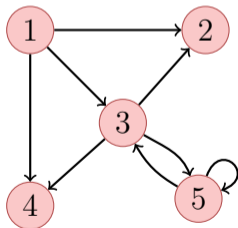


$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Memory consumption $\Theta(|V|^2)$. A_G is symmetric, if G undirected.

Representation with a List

Many graphs $G = (V, E)$ with nodes v_1, \dots, v_n provide much less than n^2 edges. Representation with *adjacency list*: Array $A[1], \dots, A[n]$, A_i comprises a linked list of nodes in $N^+(v_i)$.



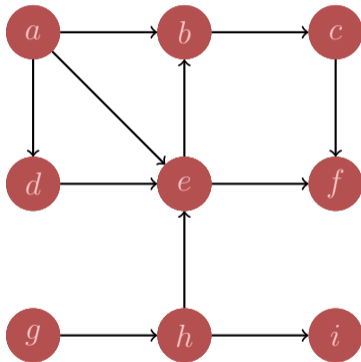
Memory Consumption $\Theta(|V| + |E|)$.

Runtimes of simple Operations

Operation	Matrix	List
Find neighbours/successors of $v \in V$	$\Theta(n)$	$\Theta(\deg^+ v)$
find $v \in V$ without neighbour/successor	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Insert edge	$\Theta(1)$	$\Theta(1)$
Delete edge	$\Theta(1)$	$\Theta(\deg^+ v)$

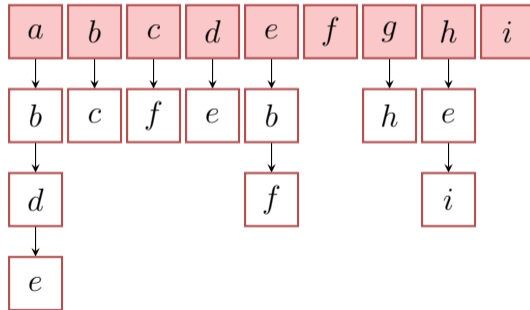
Graph Traversal: Depth First Search

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



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

Adjazenzliste



Colors

Conceptual coloring of nodes

- **white:** node has not been discovered yet.
- **grey:** node has been discovered and is marked for traversal / being processed.
- **black:** node was discovered and entirely processed.

Algorithm Depth First visit DFS-Visit(G, v)

Input: graph $G = (V, E)$, Knoten v .

$v.color \leftarrow \text{grey}$

foreach $w \in N^+(v)$ **do**

if $w.color = \text{white}$ **then**
 └ DFS-Visit(G, w)

$v.color \leftarrow \text{black}$

Depth First Search starting from node v . Running time (without recursion): $\Theta(\text{deg}^+ v)$

Algorithm Depth First visit DFS-Visit(G)

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

foreach $v \in V$ **do**

└ $v.color \leftarrow \text{white}$

foreach $v \in V$ **do**

└ **if** $v.color = \text{white}$ **then**
└ DFS-Visit(G, v)

Depth First Search for all nodes of a graph. Running time:

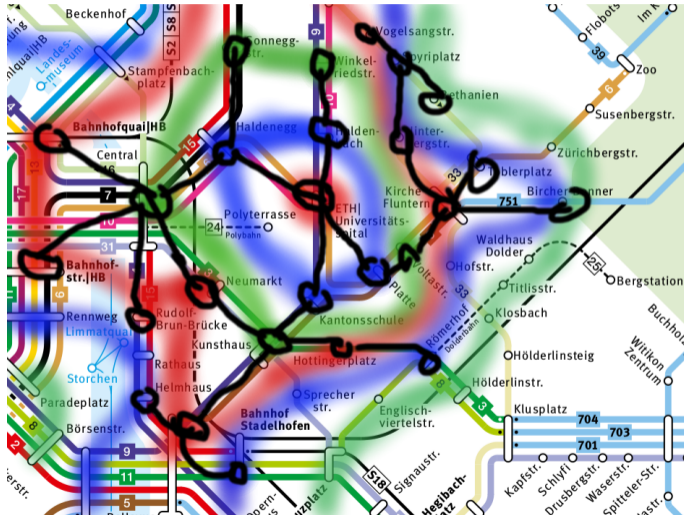
$$\Theta(|V| + \sum_{v \in V} (\deg^+(v) + 1)) = \Theta(|V| + |E|).$$

Interpretation of the Colors

When traversing the graph, a tree (or Forest) is built. When nodes are discovered there are three cases

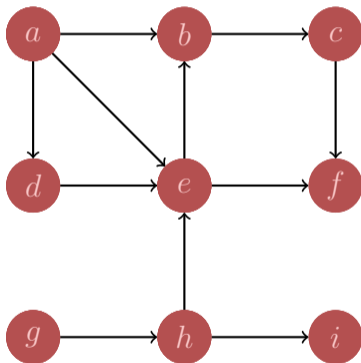
- White node: new tree edge
- Grey node: Zyklus (“back-edge”)
- Black node: forward- / cross edge

Breadth First Search



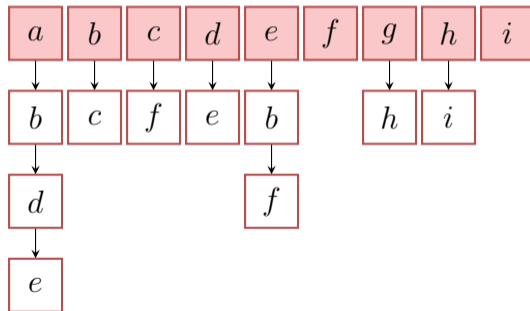
Graph Traversal: Breadth First Search

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



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

Adjazenzliste



(Iterative) BFS-Visit(G, v)

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

Queue $Q \leftarrow \emptyset$

$v.color \leftarrow \text{grey}$

enqueue(Q, v)

while $Q \neq \emptyset$ **do**

$w \leftarrow \text{dequeue}(Q)$

foreach $c \in N^+(w)$ **do**

if $c.color = \text{white}$ **then**

$c.color \leftarrow \text{grey}$

 enqueue(Q, c)

$w.color \leftarrow \text{black}$

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

Main program BFS-Visit(G)

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

foreach $v \in V$ **do**

└ $v.color \leftarrow \text{white}$

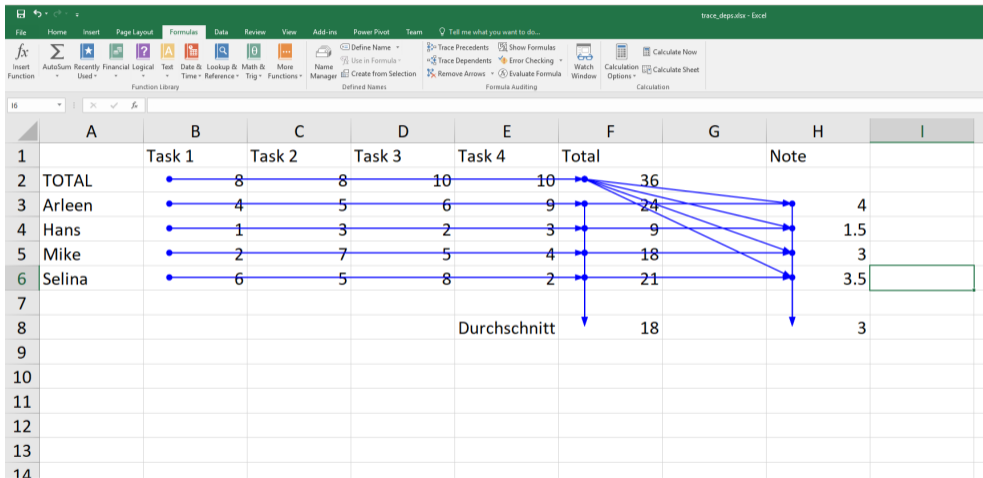
foreach $v \in V$ **do**

└ **if** $v.color = \text{white}$ **then**
└ BFS-Visit(G, v)

Breadth First Search for all nodes of a graph. Running time:

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

Topological Sorting



Evaluation Order?

Topological Sorting

Topological Sorting of an acyclic directed graph $G = (V, E)$:

Bijjective mapping

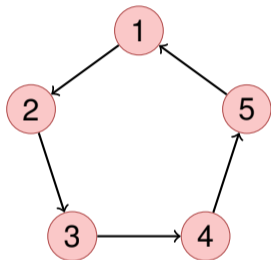
$$\text{ord} : V \rightarrow \{1, \dots, |V|\}$$

such that

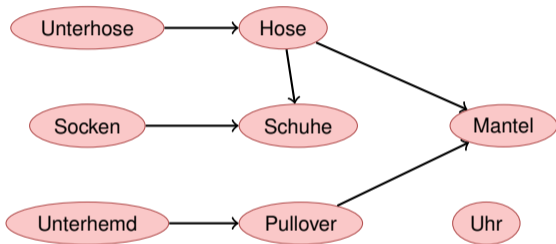
$$\text{ord}(v) < \text{ord}(w) \quad \forall (v, w) \in E.$$

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

(Counter-)Examples



Cyclic graph: cannot be sorted topologically.



A possible topological sorting of the graph:
Unterhemd,Pullover,Unterhose,Uhr,Hose,Mantel,Socken,Schuhe

Observation

Theorem

A directed graph $G = (V, E)$ permits a topological sorting if and only if it is acyclic.

Proof “ \Rightarrow ”: If G contains a cycle it cannot permit a topological sorting, because in a cycle $\langle v_{i_1}, \dots, v_{i_m} \rangle$ it would hold that $v_{i_1} < \dots < v_{i_m} < v_{i_1}$.

Inductive Proof Opposite Direction

- Base case ($n = 1$): Graph with a single node without loop can be sorted topologically, set $\text{ord}(v_1) = 1$.
- Hypothesis: Graph with n nodes can be sorted topologically
- Step ($n \rightarrow n + 1$):
 - 1 G contains a node v_q with in-degree $\text{deg}^-(v_q) = 0$. 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 $\text{ord}(v_i) \leftarrow \text{ord}(v_i) + 1$ for all $i \neq q$ and set $\text{ord}(v_q) \leftarrow 1$.

Preliminary Sketch of an Algorithm

Graph $G = (V, E)$. $d \leftarrow 1$

- 1 Traverse backwards starting from any node until a node v_q with in-degree 0 is found.
- 2 If no node with in-degree 0 found after n steps, then the graph has a cycle.
- 3 Set $\text{ord}(v_q) \leftarrow d$.
- 4 Remove v_q and his edges from G .
- 5 If $V \neq \emptyset$, then $d \leftarrow d + 1$, go to step 1.

Worst case runtime: $\Theta(|V|^2)$.

Improvement

Idea?

Compute the in-degree of all nodes in advance and traverse the nodes with in-degree 0 while correcting the in-degrees of following nodes.

Algorithm Topological-Sort(G)

Input: graph $G = (V, E)$.

Output: Topological sorting 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** $\text{push}(S, v)$ // Memorize nodes with in-degree 0

$i \leftarrow 1$

while $S \neq \emptyset$ **do**

$v \leftarrow \text{pop}(S)$; $\text{ord}[v] \leftarrow i$; $i \leftarrow i + 1$ // 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** $\text{push}(S, w)$

if $i = |V| + 1$ **then return** ord **else return** "Cycle Detected"

Algorithm Correctness

Theorem

Let $G = (V, E)$ be a directed acyclic graph. Algorithm $\text{TopologicalSort}(G)$ computes a topological sorting ord for G with runtime $\Theta(|V| + |E|)$.

Proof: follows from previous theorem:

- 1 Decreasing the in-degree corresponds with node removal.
- 2 In the algorithm it holds for each node v with $A[v] = 0$ that either the node has in-degree 0 or that previously all predecessors have been assigned a value $\text{ord}[u] \leftarrow i$ and thus $\text{ord}[v] > \text{ord}[u]$ for all predecessors u of v . Nodes are put to the stack only once.
- 3 Runtime: inspection of the algorithm (with some arguments like with graph traversal)

Algorithm Correctness

Theorem

Let $G = (V, E)$ be a directed graph containing a cycle. Algorithm $\text{TopologicalSort}(G)$ terminates within $\Theta(|V| + |E|)$ steps and detects a cycle.

Proof: let $\langle v_{i_1}, \dots, v_{i_k} \rangle$ be a cycle in G . In each step of the algorithm remains $A[v_{i_j}] \geq 1$ for all $j = 1, \dots, k$. Thus k nodes are never pushed on the stack and 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 $\Theta(|V| + |E|)$.

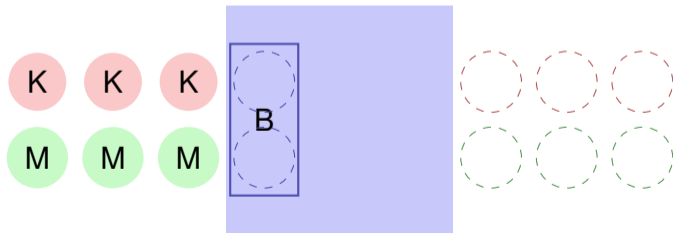
14. Shortest Paths

Motivation, Dijkstra's algorithm on distance graphs, Bellman-Ford Algorithm

[Ottman/Widmayer, Kap. 9.5.1-9.5.2 Cormen et al, Kap. 24.1-24.3]

River Crossing (Missionaries and Cannibals)

Problem: Three cannibals and three missionaries are standing at a river bank. The available boat can carry two people. At no time may at any place (banks or boat) be more cannibals than missionaries. How can the missionaries and cannibals cross the river as fast as possible? ²³



²³There are slight variations of this problem. It is equivalent to the jealous husbands problem.

Problem as Graph

Enumerate permitted configurations as nodes and connect them with an edge, when a crossing is allowed. The problem then becomes a shortest path problem.

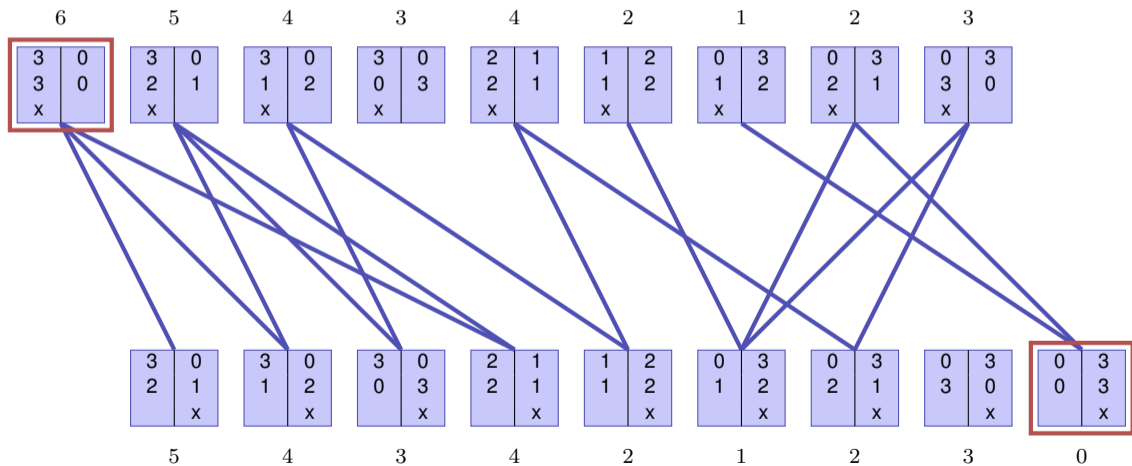
Example

	links	rechts		links	rechts	
Missionare	3	0	Überfahrt möglich	Missionare	2	1
Kannibalen	3	0		Kannibalen	2	1
Boot	x			Boot		x

6 Personen am linken Ufer

4 Personen am linken Ufer

The whole problem as a graph

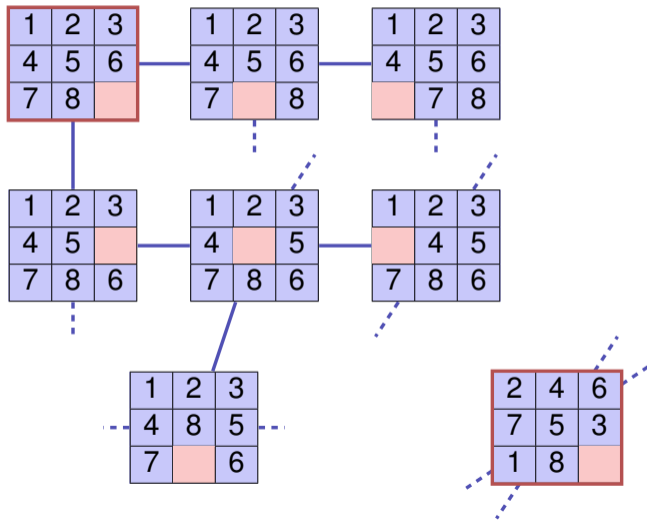


Another Example: Mystic Square

Want to find the fastest solution for

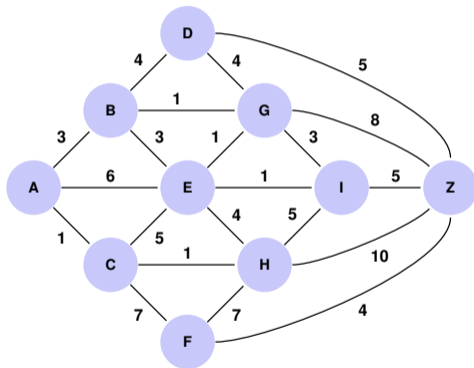


Problem as Graph



Route Finding

Provided cities A - Z and Distances between cities.

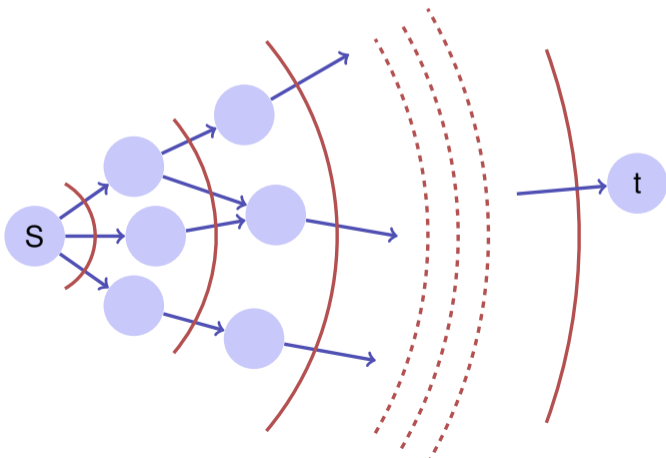


What is the shortest path from A to Z?

Simplest Case

Constant edge weight 1 (wlog)

Solution: Breadth First Search



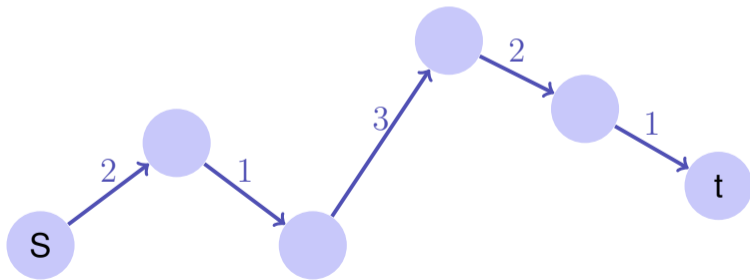
Weighted Graphs

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

Wanted: Length (weight) of a shortest path from s to t .

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

Weight: $c(p) := \sum_{i=0}^{k-1} c((v_i, v_{i+1}))$.



Path with weight 9

Shortest Paths

Notation: we write

$$u \overset{p}{\rightsquigarrow} v \quad \text{oder} \quad p : u \rightsquigarrow v$$

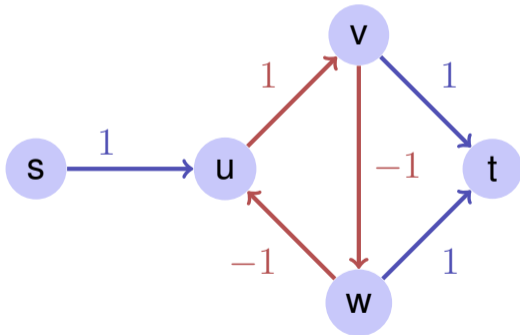
and mean a path p from u to v

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

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

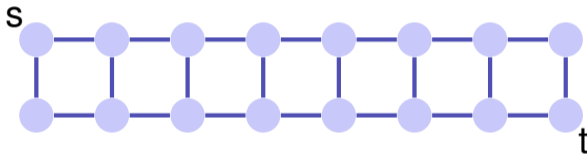
Observations (1)

It may happen that a shortest paths does not exist: negative cycles can occur.



Observations (2)

There can be exponentially many paths.



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

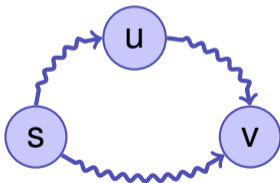
⇒ To try all paths is too inefficient

Observations (3)

Triangle Inequality

For all $s, u, v \in V$:

$$\delta(s, v) \leq \delta(s, u) + \delta(u, v)$$

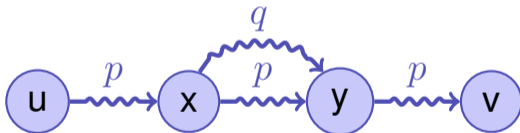


A shortest path from s to v cannot be longer than a shortest path from s to v that has to include u

Observations (4)

Optimal Substructure

Sub-paths of shortest paths are shortest paths. Let $p = \langle v_0, \dots, v_k \rangle$ 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 (5)

Shortest paths do not contain cycles

- 1 Shortest path contains a negative cycle: there is no shortest path, contradiction
- 2 Path contains a positive cycle: removing the cycle from the path will reduce the weight. Contradiction.
- 3 Path contains a cycle with weight 0: removing the cycle from the path will not change the weight. Remove the cycle (convention).

Ingredients of an Algorithm

Wanted: shortest paths from a starting node s .

- Weight of the shortest path found so far

$$d_s : V \rightarrow \mathbb{R}$$

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

$$\pi_s : V \rightarrow V$$

Initially $\pi_s[v]$ undefined for each node $v \in V$

General Algorithm

- 1 Initialise d_s and π_s : $d_s[v] = \infty$, $\pi_s[v] = \text{null}$ for each $v \in V$
- 2 Set $d_s[s] \leftarrow 0$
- 3 Choose an edge $(u, v) \in E$

Relaxiere (u, v) :

if $d_s[v] > d_s[u] + c(u, v)$ then

$d_s[v] \leftarrow d_s[u] + c(u, v)$

$\pi_s[v] \leftarrow u$

- 4 Repeat 3 until nothing can be relaxed any more.
(until $d_s[v] \leq d_s[u] + c(u, v) \quad \forall (u, v) \in E$)

It is Safe to Relax

At any time in the algorithm above it holds

$$d_s[v] \geq \delta(s, v) \quad \forall v \in V$$

In the relaxation step:

$$\delta(s, v) \leq \delta(s, u) + \delta(u, v) \quad \text{[Triangle Inequality].}$$

$$\delta(s, u) \leq d_s[u] \quad \text{[Induction Hypothesis].}$$

$$\delta(u, v) \leq c(u, v) \quad \text{[Minimality of } \delta \text{]}$$

$$\Rightarrow d_s[u] + c(u, v) \geq \delta(s, v)$$

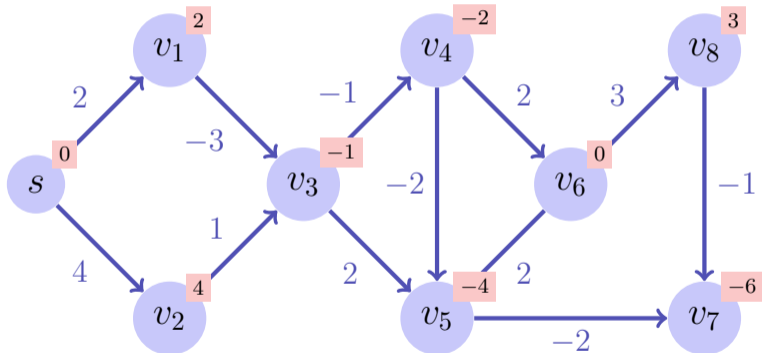
$$\Rightarrow \min\{d_s[v], d_s[u] + c(u, v)\} \geq \delta(s, v)$$

Central Question

How / in which order should edges be chosen in above algorithm?

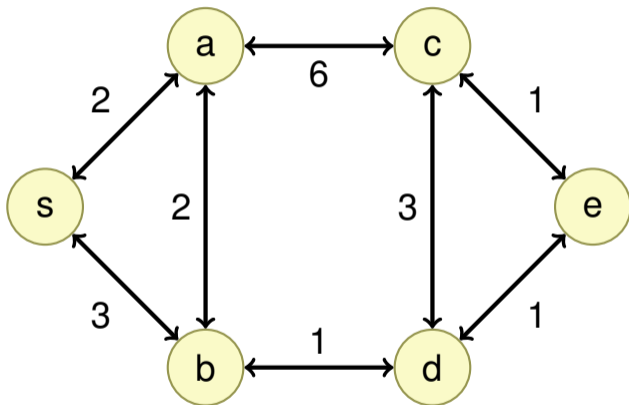
Special Case: Directed Acyclic Graph (DAG)

DAG \Rightarrow topological sorting returns optimal visiting order



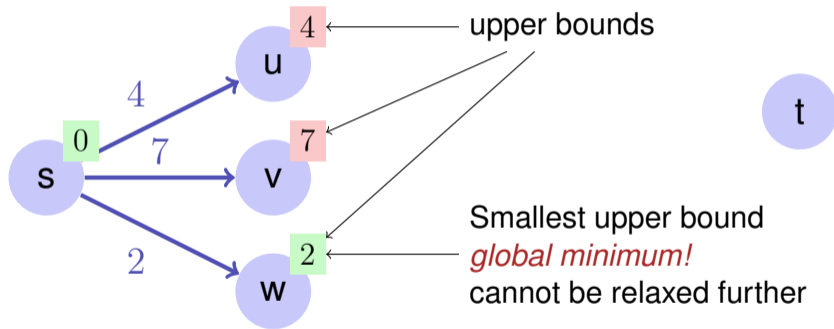
Top. Sort: \Rightarrow Order $s, v_1, v_2, v_3, v_4, v_6, v_5, v_8, v_7$.

Assumption (preliminary)



All weights of G are *positive*.

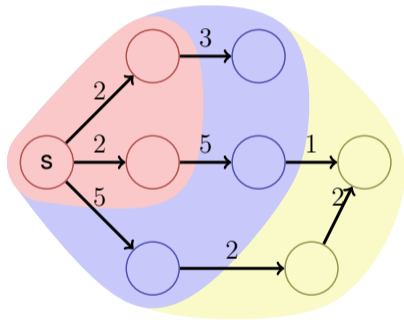
Observation (Dijkstra)



Basic Idea

Set V of nodes is partitioned into

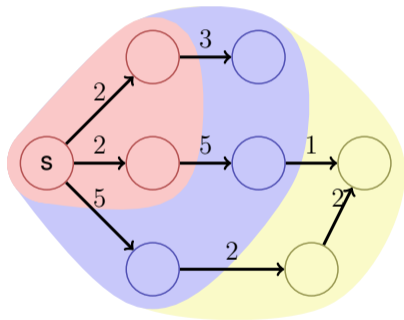
- the set M of nodes for which a shortest path from s is already known,
- the set $R = \bigcup_{v \in M} N^+(v) \setminus M$ of nodes where a shortest path is not yet known but that are accessible directly from M ,
- the set $U = V \setminus (M \cup R)$ of nodes that have not yet been considered.



Induction

Induction over $|M|$: choose nodes from R with smallest upper bound. Add r to M and update R and U accordingly.

Correctness: if within the “wavefront” a node with minimal weight w has been found then no path over later nodes (providing weight $\geq d$) can provide any improvement.



Algorithm Dijkstra(G, s)

Input: Positively weighted Graph $G = (V, E, c)$, starting point $s \in V$,

Output: Minimal weights d of the shortest paths and corresponding predecessor node for each node.

foreach $u \in V$ **do**

$d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow \mathbf{null}$

$d_s[s] \leftarrow 0$; $R \leftarrow \{s\}$

while $R \neq \emptyset$ **do**

$u \leftarrow \text{ExtractMin}(R)$

foreach $v \in N^+(u)$ **do**

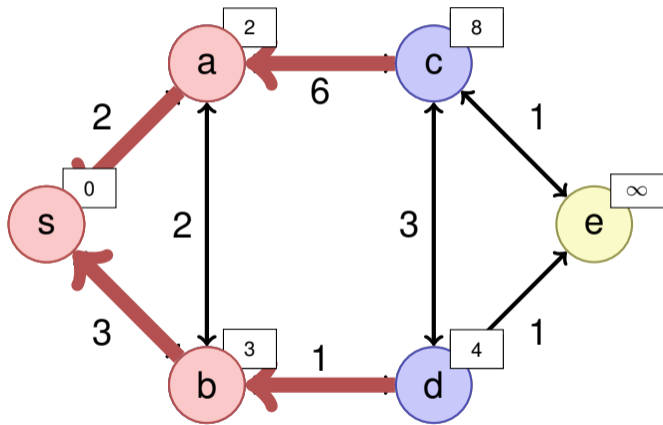
if $d_s[u] + c(u, v) < d_s[v]$ **then**

$d_s[v] \leftarrow d_s[u] + c(u, v)$

$\pi_s[v] \leftarrow u$

$R \leftarrow R \cup \{v\}$

Example



$$M = \{s, a, b\}$$

$$R = \{c, d\}$$

$$U = \{e\}$$

Implementation: Data Structure for R ?

Required operations:

- Insert (add to R)
- ExtractMin (over R) and DecreaseKey (Update in R)

```
foreach  $v \in N^+(u)$  do
  if  $d_s[u] + c(u, v) < d_s[v]$  then
     $d_s[v] \leftarrow d_s[u] + c(u, v)$ 
     $\pi_s[v] \leftarrow u$ 
    if  $v \in R$  then
      DecreaseKey( $R, v$ )           // Update of a  $d(v)$  in the heap of  $R$ 
    else
       $R \leftarrow R \cup \{v\}$     // Update of  $d(v)$  in the heap of  $R$ 
```

MinHeap!

DecreaseKey

- DecreaseKey: climbing in MinHeap in $\mathcal{O}(\log |V|)$
- Position in the heap?
 - alternative (a): Store position at the nodes
 - alternative (b): Hashtable of the nodes
 - alternative (c): re-insert node after successful relax operation and mark it "deleted" once extracted (Lazy Deletion).²⁴

²⁴For lazy deletion a pair of edge (or target node) and distance is required.

Runtime

- $|V| \times$ ExtractMin: $\mathcal{O}(|V| \log |V|)$
- $|E| \times$ Insert or DecreaseKey: $\mathcal{O}(|E| \log |V|)$
- $1 \times$ Init: $\mathcal{O}(|V|)$
- Overall: $\mathcal{O}(|E| \log |V|)$.

General Weighted Graphs

Relaxing Step as before but with a return value:

Relax(u, v) ($u, v \in V, (u, v) \in E$)

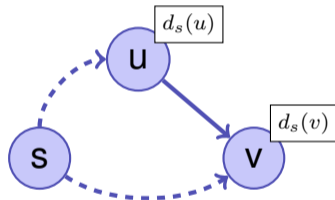
if $d_s[u] + c(u, v) < d_s[v]$ **then**

$d_s[v] \leftarrow d_s[u] + c(u, v)$

$\pi_s[v] \leftarrow u$

return true

return false



Problem: cycles with negative weights can shorten the path, a shortest path is not guaranteed to exist.

Dynamic Programming Approach (Bellman)

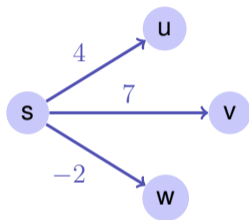
Induction over number of edges $d_s[i, v]$: Shortest path from s to v via maximally i edges.

$$d_s[i, v] = \min\{d_s[i - 1, v], \min_{(u,v) \in E} (d_s[i - 1, u] + c(u, v))\}$$

$$d_s[0, s] = 0, d_s[0, v] = \infty \quad \forall v \neq s.$$

Dynamic Programming Approach (Bellman)

	s	\dots	v	\dots	w
0	0	∞	∞	∞	∞
1	0	∞	7	∞	-2
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
$n - 1$	0	\dots	\dots	\dots	\dots



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**

$d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow \mathbf{null}$

$d_s[s] \leftarrow 0$;

for $i \leftarrow 1$ **to** $|V|$ **do**

$f \leftarrow \mathbf{false}$

foreach $(u, v) \in E$ **do**

$f \leftarrow f \vee \text{Relax}(u, v)$

if $f = \mathbf{false}$ **then return** true

return false;

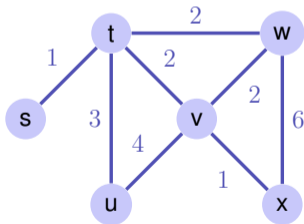
15. Minimum Spanning Trees

Motivation, Greedy, Algorithm Kruskal, General Rules, ADT Union-Find, Algorithm Jarnik, Prim, Dijkstra, [Ottman/Widmayer, Kap. 9.6, 6.2, 6.1, Cormen et al, Kap. 23, 19]

Problem

Given: Undirected, weighted, connected graph $G = (V, E, c)$.

Wanted: Minimum Spanning Tree $T = (V, E')$: connected, cycle-free subgraph $E' \subset E$, such that $\sum_{e \in E'} c(e)$ minimal.



Application Examples

- Network-Design: find the cheapest / shortest network that connects all nodes.
- Approximation of a solution of the travelling salesman problem: find a round-trip, as short as possible, that visits each node once.

25

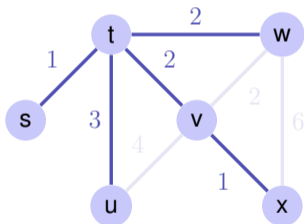
²⁵The best known algorithm to solve the TS problem exactly has exponential running time.

Greedy Procedure

- Greedy algorithms compute the solution stepwise choosing locally optimal solutions.
- Most problems cannot be solved with a greedy algorithm.
- The Minimum Spanning Tree problem can be solved with a greedy strategy.

Greedy Idea (Kruskal, 1956)

Construct T by adding the cheapest edge that does not generate a cycle.



(Solution is not unique.)

Algorithm MST-Kruskal(G)

Input: Weighted Graph $G = (V, E, c)$

Output: Minimum spanning tree with edges A .

Sort edges by weight $c(e_1) \leq \dots \leq c(e_m)$

$A \leftarrow \emptyset$

for $k = 1$ **to** $|E|$ **do**

if $(V, A \cup \{e_k\})$ acyclic **then**
 $A \leftarrow A \cup \{e_k\}$

return (V, A, c)

[Correctness]

At each point in the algorithm (V, A) is a forest, a set of trees.

MST-Kruskal considers each edge e_k exactly once and either chooses or rejects e_k

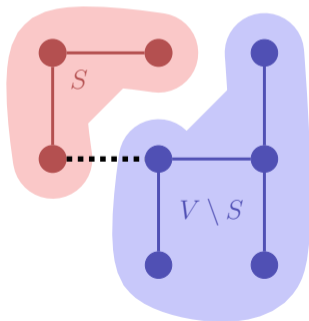
Notation (snapshot of the state in the running algorithm)

- A : Set of selected edges
- R : Set of rejected edges
- U : Set of yet undecided edges

[Cut]

A cut of G is a partition $S, V - S$ of V . ($S \subseteq V$).

An edge crosses a cut when one of its endpoints is in S and the other is in $V \setminus S$.



[Rules]

- 1 Selection rule: choose a cut that is not crossed by a selected edge. Of all undecided edges that cross the cut, select the one with minimal weight.
- 2 Rejection rule: choose a circle without rejected edges. Of all undecided edges of the circle, reject those with minimal weight.

[Rules]

Kruskal applies both rules:

- 1 A selected e_k connects two connection components, otherwise it would generate a circle. 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 circle. Within the circle e_k has minimal weight.

[Correctness]

Theorem

Every algorithm that applies the rules above in a step-wise manner until $U = \emptyset$ is correct.

Consequence: MST-Kruskal is correct.

[Selection invariant]

Invariant: At each step there is a minimal spanning tree that contains all selected and none of the rejected edges.

If both rules satisfy the invariant, then the algorithm is correct.

Induction:

- At beginning: $U = E$, $R = A = \emptyset$. Invariant obviously holds.
- Invariant is preserved at each step of the algorithm.
- At the end: $U = \emptyset$, $R \cup A = E \Rightarrow (V, A)$ is a spanning tree.

Proof of the theorem: show that both rules preserve the invariant.

[Selection rule preserves the invariant]

At each step there is a minimal spanning tree T that contains all selected and none of the rejected edges.

Choose a cut that is not crossed by a selected edge. Of all undecided edges that cross the cut, select the edge e with minimal weight.

- Case 1: $e \in T$ (done)
- Case 2: $e \notin T$. Then $T \cup \{e\}$ contains a circle that contains e . Circle must have a second edge e' that also crosses the cut.²⁶ 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 $c(e) = c(e')$).

²⁶Such a circle contains at least one node in S and one node in $V \setminus S$ and therefore at least one edge between S and $V \setminus S$.

[Rejection rule preserves the invariant]

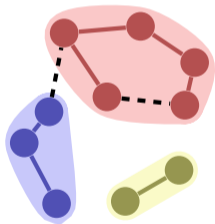
At each step there is a minimal spanning tree T that contains all selected and none of the rejected edges.

Choose a circle without rejected edges. Of all undecided edges of the circle, reject an edge e with minimal 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 circle. Because $c(e') \leq c(e)$, $T' = T \setminus \{e\} \cup \{e'\}$ is also minimal (and $c(e) = c(e')$).

Implementation Issues

Consider a set of sets $i \equiv A_i \subset V$. To identify cuts and circles: 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

- $\text{Make-Set}(i)$: create a new set represented by i .
- $\text{Find}(e)$: name of the set i that contains e .
- $\text{Union}(i, j)$: 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 $c(e_1) \leq \dots \leq c(e_m)$

$A \leftarrow \emptyset$

for $k = 1$ **to** $|V|$ **do**

\lfloor MakeSet(k)

for $k = 1$ **to** m **do**

$(u, v) \leftarrow e_k$

if Find(u) \neq Find(v) **then**

 Union(Find(u), Find(v))

$A \leftarrow A \cup e_k$

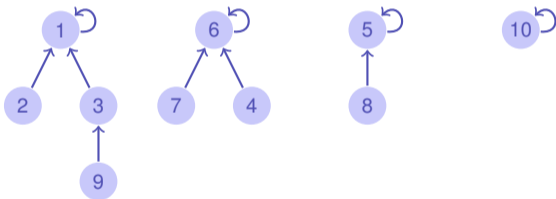
else

// conceptual: $R \leftarrow R \cup e_k$

return (V, A, c)

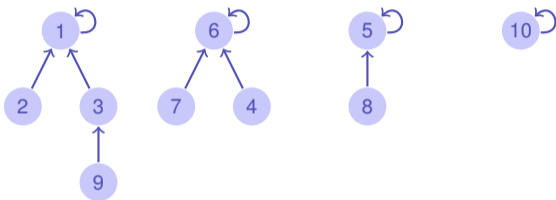
Implementation Union-Find

Idea: tree for each subset in the partition, e.g.
 $\{\{1, 2, 3, 9\}, \{7, 6, 4\}, \{5, 8\}, \{10\}\}$



roots = names (representatives) of the sets,
trees = elements of the sets

Implementation Union-Find



Representation as array:

Index	1	2	3	4	5	6	7	8	9	10
Parent	1	1	1	6	5	6	5	5	3	10

Implementation Union-Find

Index	1	2	3	4	5	6	7	8	9	10
Parent	1	1	1	6	5	6	5	5	3	10

Make-Set(i) $p[i] \leftarrow i$; **return** i

Find(i) **while** ($p[i] \neq i$) **do** $i \leftarrow p[i]$
 return i

Union(i, j)²⁷ $p[j] \leftarrow i$;

²⁷ 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) $p[i] \leftarrow i; g[i] \leftarrow 1; \text{ return } i$

Union(i, j) **if** $g[j] > g[i]$ **then** $\text{swap}(i, j)$
 $p[j] \leftarrow i$
 if $g[i] = g[j]$ **then** $g[i] \leftarrow g[i] + 1$

\Rightarrow Tree depth (and worst-case running time for Find) in $\Theta(\log n)$

[Observation]

Theorem

The method above (union by size) preserves the following property of the trees: a tree of height h has at least 2^h nodes.

Immediate consequence: runtime Find = $\mathcal{O}(\log n)$.

[Proof]

Induction: by assumption, sub-trees have at least 2^{h_i} nodes. WLOG: $h_2 \leq h_1$

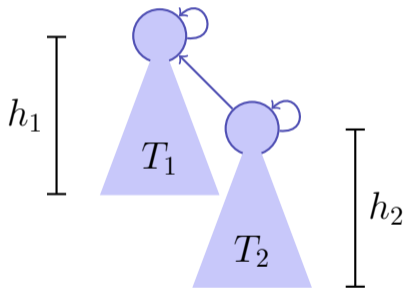
■ $h_2 < h_1$:

$$h(T_1 \oplus T_2) = h_1 \Rightarrow g(T_1 \oplus T_2) \geq 2^{h_1}$$

■ $h_2 = h_1$:

$$g(T_1) \geq g(T_2) \geq 2^{h_2}$$

$$\Rightarrow g(T_1 \oplus T_2) = g(T_1) + g(T_2) \geq 2 \cdot 2^{h_2} = 2^{h(T_1 \oplus T_2)}$$



Further improvement

Link all nodes to the root when Find is called.

Find(i):

$j \leftarrow i$

while ($p[i] \neq i$) **do** $i \leftarrow p[i]$

while ($j \neq i$) **do**

$t \leftarrow j$
 $j \leftarrow p[j]$
 $p[t] \leftarrow i$

return i

Cost: amortised *nearly* constant (inverse of the Ackermann-function).²⁸

²⁸We do not go into details here.

Running time of Kruskal's Algorithm

- Sorting of the edges: $\Theta(|E| \log |E|) = \Theta(|E| \log |V|)$.²⁹
- Initialisation of the Union-Find data structure $\Theta(|V|)$
- $|E| \times \text{Union}(\text{Find}(x), \text{Find}(y))$: $\mathcal{O}(|E| \log |E|) = \mathcal{O}(|E| \log |V|)$.

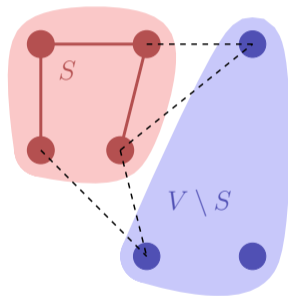
Overall $\Theta(|E| \log |V|)$.

²⁹because G is connected: $|V| \leq |E| \leq |V|^2$

Algorithm of Jarnik (1930), Prim, Dijkstra (1959)

Idea: start with some $v \in V$ and grow the spanning tree from here by the acceptance rule.

```
A ← ∅  
S ← {v0}  
for i ← 1 to |V| do  
  Choose cheapest (u, v) mit u ∈ S, v ∉ S  
  A ← A ∪ {(u, v)}  
  S ← S ∪ {v} // (Coloring)
```



Remark: a union-Find data structure is not required. It suffices to color nodes when they are added to S .

Running time

Trivially $\mathcal{O}(|V| \cdot |E|)$.

Improvement (like with Dijkstra's ShortestPath)

- With Min-Heap: costs

- Initialization (node coloring) $\mathcal{O}(|V|)$
- $|V| \times \text{ExtractMin} = \mathcal{O}(|V| \log |V|)$,
- $|E| \times \text{Insert or DecreaseKey} = \mathcal{O}(|E| \log |V|)$,

$$\mathcal{O}(|E| \cdot \log |V|)$$

16. Flow in Networks

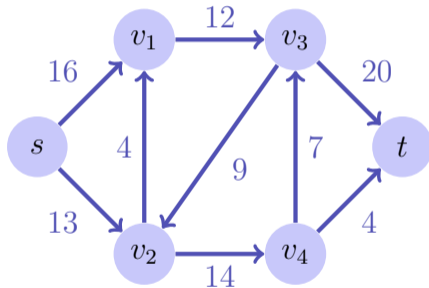
Flow Network, Maximal Flow, Cut, Rest Network, Max-flow Min-cut Theorem, Ford-Fulkerson Method, Edmonds-Karp Algorithm, Maximal Bipartite Matching [Ottman/Widmayer, Kap. 9.7, 9.8.1], [Cormen et al, Kap. 26.1-26.3]

Motivation

- Modelling flow of fluents, components on conveyors, current in electrical networks or information flow in communication networks.
- Connectivity of Communication Networks, Bipartite Matching, Circulation, Scheduling, Image Segmentation, Baseball Elimination...

Flow Network

- **Flow network** $G = (V, E, c)$: directed graph with **capacities**
- Antiparallel edges forbidden:
 $(u, v) \in E \Rightarrow (v, u) \notin E$.
- Model a missing edge (u, v) by
 $c(u, v) = 0$.
- **Source** s and **sink** t : special nodes.
Every node v is on a path between s
and t : $s \rightsquigarrow v \rightsquigarrow t$



Flow

A *Flow* $f : V \times V \rightarrow \mathbb{R}$ fulfills the following conditions:

- *Bounded Capacity:*

For all $u, v \in V$: $f(u, v) \leq c(u, v)$.

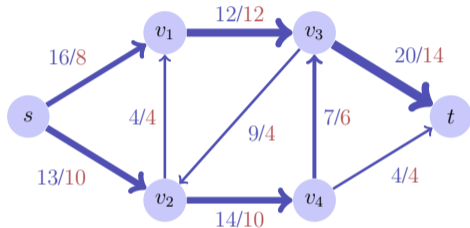
- *Skew Symmetry:*

For all $u, v \in V$: $f(u, v) = -f(v, u)$.

- *Conservation of flow:*

For all $u \in V \setminus \{s, t\}$:

$$\sum_{v \in V} f(u, v) = 0.$$



Value of the flow:

$$|f| = \sum_{v \in V} f(s, v).$$

Here $|f| = 18$.

How large can a flow possibly be?

Limiting factors: cuts

- *cut separating s from t* : Partition of V into S and T with $s \in S$, $t \in T$.
- *Capacity* of a cut: $c(S, T) = \sum_{v \in S, v' \in T} c(v, v')$
- *Minimal cut*: cut with minimal capacity.
- *Flow over the cut*: $f(S, T) = \sum_{v \in S, v' \in T} f(v, v')$

Implicit Summation

Notation: Let $U, U' \subseteq V$

$$f(U, U') := \sum_{\substack{u \in U \\ u' \in U'}} f(u, u'), \quad f(u, U') := f(\{u\}, U')$$

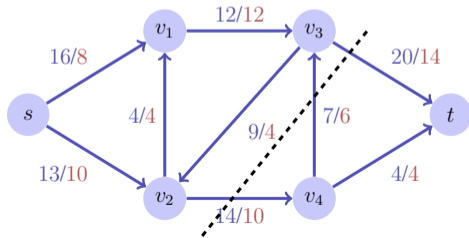
Thus

- $|f| = f(s, V)$
- $f(U, U) = 0$
- $f(U, U') = -f(U', U)$
- $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$, if $X \cap Y = \emptyset$.
- $f(R, V) = 0$ if $R \cap \{s, t\} = \emptyset$. [flow conversation!]

How large can a flow possibly be?

For each flow and each cut it holds that $f(S, T) = |f|$:

$$\begin{aligned} f(S, T) &= f(S, V) - \underbrace{f(S, S)}_0 = f(S, V) \\ &= f(s, V) + \underbrace{f(S - \{s\}, V)}_{\not\rightarrow t, \not\rightarrow s} = |f|. \end{aligned}$$

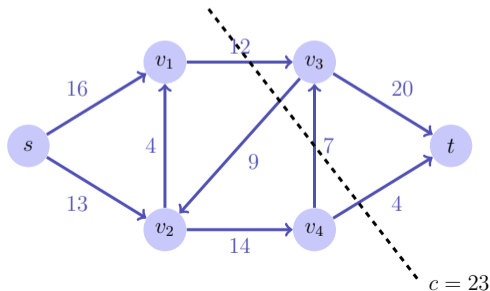


Maximal Flow ?

In particular, for each cut (S, T) of V .

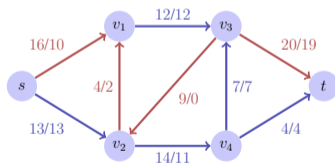
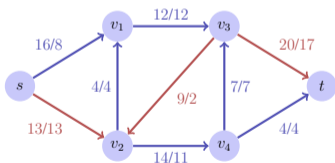
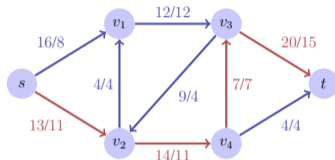
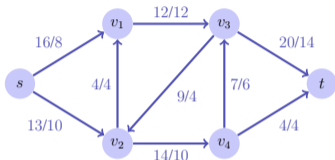
$$|f| \leq \sum_{v \in S, v' \in T} c(v, v') = c(S, T)$$

Will discover that equality holds for $\min_{S, T} c(S, T)$.



Maximal Flow ?

Naive Procedure



Conclusion: greedy increase of flow does not solve the problem.

The Method of Ford-Fulkerson

- Start with $f(u, v) = 0$ for all $u, v \in V$
- Determine rest network* G_f and expansion path in G_f
- Increase flow via expansion path*
- Repeat until no expansion path available.

$$G_f := (V, E_f, c_f)$$
$$c_f(u, v) := c(u, v) - f(u, v) \quad \forall u, v \in V$$
$$E_f := \{(u, v) \in V \times V \mid c_f(u, v) > 0\}$$

*Will now be explained

Increase of flow, negative!

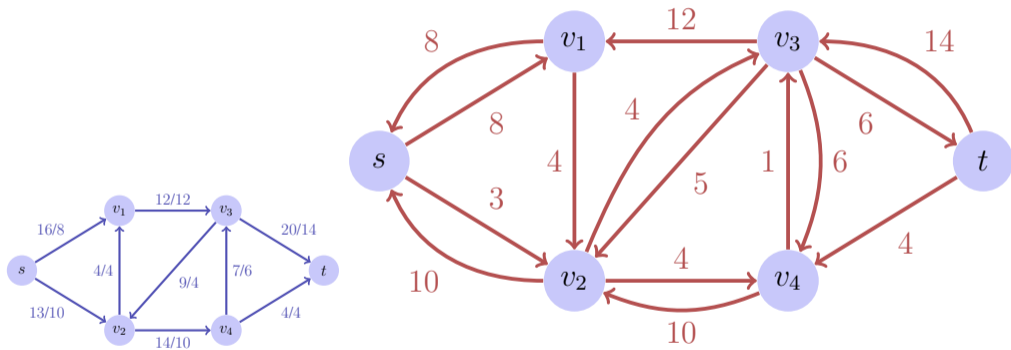
Let some flow f in the network be given.

Finding:

- Increase of the flow along some edge possible, when flow can be increased along the edge, i.e. if $f(u, v) < c(u, v)$.
Rest capacity $c_f(u, v) = c(u, v) - f(u, v) > 0$.
- Increase of flow *against the direction* of the edge possible, if flow can be reduced along the edge, i.e. if $f(u, v) > 0$.
Rest capacity $c_f(v, u) = f(u, v) > 0$.

Rest Network

Rest network G_f provided by the edges with positive rest capacity:



Rest networks provide the same kind of properties as flow networks with the exception of permitting antiparallel capacity-edges

Observation

Theorem

Let $G = (V, E, c)$ be a flow network with source s and sink t and f a flow in G . Let G_f be the corresponding rest networks and let f' be a flow in G_f . Then $f \oplus f'$ with

$$(f \oplus f')(u, v) = f(u, v) + f'(u, v)$$

defines a flow in G with value $|f| + |f'|$.

Proof

$f \oplus f'$ defines a flow in G :

■ capacity limit

$$(f \oplus f')(u, v) = f(u, v) + \underbrace{f'(u, v)}_{\leq c(u, v) - f(u, v)} \leq c(u, v)$$

■ skew symmetry

$$(f \oplus f')(u, v) = -f(v, u) + -f'(v, u) = -(f \oplus f')(v, u)$$

■ flow conservation $u \in V - \{s, t\}$:

$$\sum_{v \in V} (f \oplus f')(u, v) = \sum_{v \in V} f(u, v) + \sum_{v \in V} f'(u, v) = 0$$

Proof

Value of $f \oplus f'$

$$\begin{aligned} |f \oplus f'| &= (f \oplus f')(s, V) \\ &= \sum_{u \in V} f(s, u) + f'(s, u) \\ &= f(s, V) + f'(s, V) \\ &= |f| + |f'| \end{aligned}$$



Augmenting Paths

expansion path p : simple path from s to t in the rest network G_f .

Rest capacity $c_f(p) = \min\{c_f(u, v) : (u, v) \text{ edge in } p\}$

Flow in G_f

Theorem

The mapping $f_p : V \times V \rightarrow \mathbb{R}$,

$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \text{ edge in } p \\ -c_f(p) & \text{if } (v, u) \text{ edge in } p \\ 0 & \text{otherwise} \end{cases}$$

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 $|f_p| = \sum_{v \in V} f_p(s, v) = f_p(s, u) = c_f(p)$.

Consequence

Strategy for an algorithm:

With an expansion path p in G_f the flow $f \oplus f_p$ defines a new flow with value $|f \oplus f_p| = |f| + |f_p| > |f|$.

Max-Flow Min-Cut Theorem

Theorem

Let f be a flow in a flow network $G = (V, E, c)$ with source s and sink t . The following statements are equivalent:

- 1 f is a maximal flow in G*
- 2 The rest network G_f does not provide any expansion paths*
- 3 It holds that $|f| = c(S, T)$ for a cut (S, T) of G .*

[Proof]

- (3) \Rightarrow (1):

It holds that $|f| \leq c(S, T)$ for all cuts S, T . From $|f| = c(S, T)$ it follows that $|f|$ is maximal.

- (1) \Rightarrow (2):

f maximal Flow in G . Assumption: G_f has some expansion path $|f \oplus f_p| = |f| + |f_p| > |f|$. Contradiction.

[Proof (2) \Rightarrow (3)]

Assumption: G_f has no expansion path

Define $S = \{v \in V : \text{there is a path } s \rightsquigarrow v \text{ in } G_f\}$.

$(S, T) := (S, V \setminus S)$ is a cut: $s \in S, t \in T$.

Let $u \in S$ and $v \in T$. Then $c_f(u, v) = 0$, also

$c_f(u, v) = c(u, v) - f(u, v) = 0$. **Somit** $f(u, v) = c(u, v)$.

Thus

$$|f| = f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) = \sum_{u \in S} \sum_{v \in T} c(u, v) = C(S, T).$$



Algorithm Ford-Fulkerson(G, s, t)

Input: Flow network $G = (V, E, c)$

Output: Maximal flow f .

for $(u, v) \in E$ **do**

└ $f(u, v) \leftarrow 0$

while Exists path $p : s \rightsquigarrow t$ in rest network G_f **do**

└ $c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \in p\}$

└ **foreach** $(u, v) \in p$ **do**

└└ $f(u, v) \leftarrow f(u, v) + c_f(p)$

└└ $f(v, u) \leftarrow f(v, u) - c_f(p)$

Practical Consideration

In an implementation of the Ford-Fulkerson algorithm the negative flow edges are usually not stored because their value always equals the negated value of the antiparallel edge.

$$f(u, v) \leftarrow f(u, v) + c_f(p)$$

$$f(v, u) \leftarrow f(v, u) - c_f(p)$$

is then transformed to

if $(u, v) \in E$ **then**

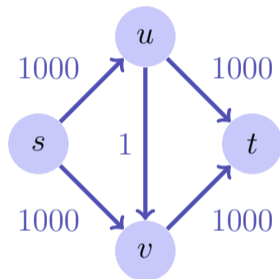
$$\quad | \quad f(u, v) \leftarrow f(u, v) + c_f(p)$$

else

$$\quad | \quad f(v, u) \leftarrow f(v, u) - c_f(p)$$

Analysis

- The Ford-Fulkerson algorithm does not necessarily have to converge for irrational capacities. For integers or rational numbers it terminates.
- For an integer flow, the algorithm requires maximally $|f_{\max}|$ iterations of the while loop (because the flow increases minimally by 1). Search a single increasing path (e.g. with DFS or BFS) $\mathcal{O}(|E|)$ Therefore $\mathcal{O}(f_{\max}|E|)$.



With an unlucky choice the algorithm may require up to 2000 iterations here.

Edmonds-Karp Algorithm

Choose in the Ford-Fulkerson-Method for finding a path in G_f the expansion path of shortest possible length (e.g. with BFS)

Edmonds-Karp Algorithm

Theorem

When the Edmonds-Karp algorithm is applied to some integer valued flow network $G = (V, E)$ with source s and sink t then the number of flow increases applied by the algorithm is in $\mathcal{O}(|V| \cdot |E|)$.

\Rightarrow Overall asymptotic runtime: $\mathcal{O}(|V| \cdot |E|^2)$

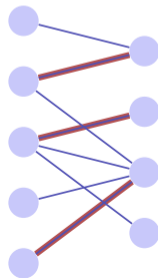
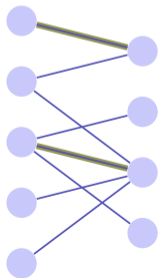
[Without proof]

Application: maximal bipartite matching

Given: bipartite undirected graph $G = (V, E)$.

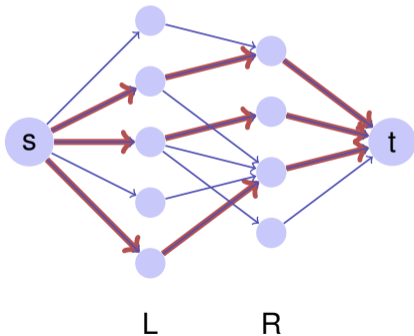
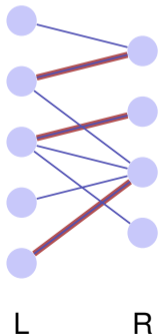
Matching M : $M \subseteq E$ such that $|\{m \in M : v \in m\}| \leq 1$ for all $v \in V$.

Maximal Matching M : Matching M , such that $|M| \geq |M'|$ for each matching M' .



Corresponding flow network

Construct a flow network that corresponds to the partition L, R of a bipartite graph with source s and sink t , with directed edges from s to L , from L to R and from R to t . Each edge has capacity 1.



[Integer number theorem]

Theorem

If the capacities of a flow network are integers, then the maximal flow generated by the Ford-Fulkerson method provides integer numbers for each $f(u, v)$, $u, v \in V$.

[without proof]

Consequence: Ford-Fulkerson generates for a flow network that corresponds to a bipartite graph a maximal matching

$$M = \{(u, v) : f(u, v) = 1\}.$$