Felix Friedrich & Hermann Lehner

Computer Science II

Course at D-BAUG, ETH Zurich

Spring 2020

Welcome!

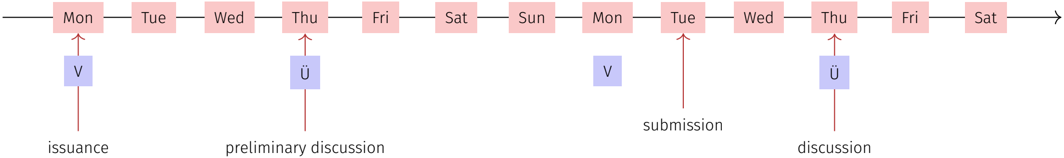
Course homepage

`http://lec.inf.ethz.ch/baug/informatik2`

The team:

Lecturers	Felix Friedrich
	Hermann Lehner
Assistants	Prashanth Chandran
	Sverrir Thorgeirsson
	Vu Nguyen
	Jan Osusky
	Michael Seeber
Back-office	Katja Wolff

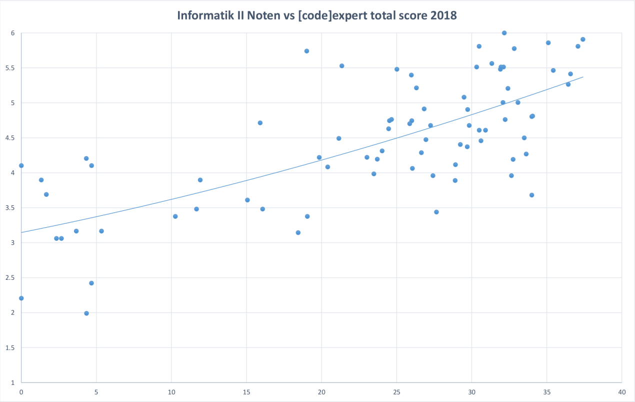
Exercises



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

Exercises

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



It is so simple!

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

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

Literature

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

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

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

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

Exams

The exam will cover

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

Exams

Written exam.

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

Offer

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

Offer (concretely)

- 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.

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

Objectives of this Course

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

Software Engineering

Java To Python Introduction
Python Datastructures

data structures / algorithms

The notion invariant, cost model, Landau notation
algorithms design, induction, divide & conquer
searching and sorting
dictionaries: hashing and search trees, balanced trees
dynamic programming
fundamental graph algorithms
Shortest paths, maximum flow

2. From Java to Python

First Python Program, Transfer Java $\rightarrow$ 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

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

First Python Program

```python
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 “:”

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

print(x)
```

```java
// 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+b`j`. Optional round parentheses.

Literals: Booleans

- `True`
- `False`

Literals: Strings

- **`'a single quoted string\nand a second line'`**
- **`"a doube quoted string\nand a second line"`**
- Multi-line strings (tripple 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`**.
- 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 Collaz-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

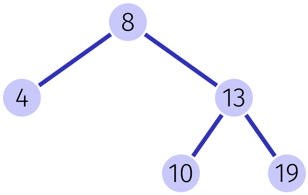
Dictionaries are very important primitive data structures in Python

- 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

```python
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
        }
    }
}
```



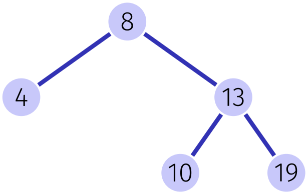
Dynamic Data Structures with Dicts

- 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()
```

3. 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
```

4. Algorithmen und Datenstrukturen

Algorithms and Data Structures, Overview
[Cormen et al, Kap. 1; Ottman/Widmayer, Kap. 1.1]

Algorithm

Algorithm

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

Example Problem: Sorting

Input: A sequence of n numbers (comparable objects) $(a_1, a_2, \ldots, a_n)$

Example Problem: Sorting

Input: A sequence of n numbers (comparable objects) $(a_1, a_2, \ldots, a_n)$

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

Example Problem: Sorting

Input: A sequence of n numbers (comparable objects) $(a_1, a_2, \ldots, a_n)$

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

Possible input

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

Example Problem: Sorting

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

Possible input

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

Every example represents a **problem instance**

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

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

Examples for algorithmic problems

- Tables and statistis: sorting, selection and searching

Examples for algorithmic problems

- Tables and statistis: sorting, selection and searching
- routing: shortest path algorithm, heap data structure

Examples for algorithmic problems

- Tables and statistis: sorting, selection and searching
- routing: shortest path algorithm, heap data structure
- DNA matching: Dynamic Programming

Examples for algorithmic problems

- Tables and statistis: sorting, selection and searching
- routing: shortest path algorithm, heap data structure
- DNA matching: Dynamic Programming
- evaluation order: Topological Sorting

Examples for algorithmic problems

- Tables and statistis: sorting, selection and searching
- routing: shortest path algorithm, heap data structure
- DNA matching: Dynamic Programming
- evaluation order: Topological Sorting
- autocomletion and spell-checking: Dictionaries / Trees

Examples for algorithmic problems

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

Examples for algorithmic problems

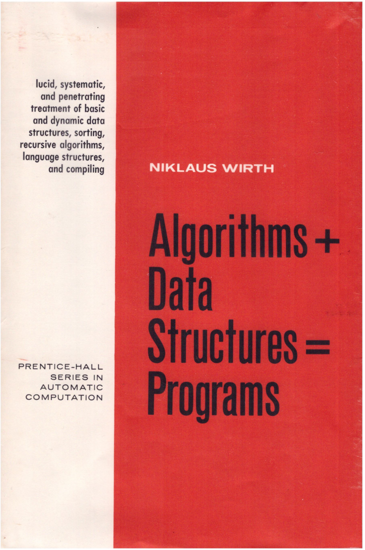
- Tables and statistis: sorting, selection and searching
- routing: shortest path algorithm, heap data structure
- DNA matching: Dynamic Programming
- evaluation order: Topological Sorting
- autocomletion and spell-checking: Dictionaries / Trees
- Fast Lookup : Hash-Tables
- The travelling Salesman: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

Characteristics

- Extremely large number of potential solutions
- Practical applicability

Data Structures

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



Efficiency

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

Efficiency

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

Reality: resources are bounded and not free:

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

Actually, this course is nearly only about efficiency.

Hard problems.

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

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

5. Efficiency of algorithms

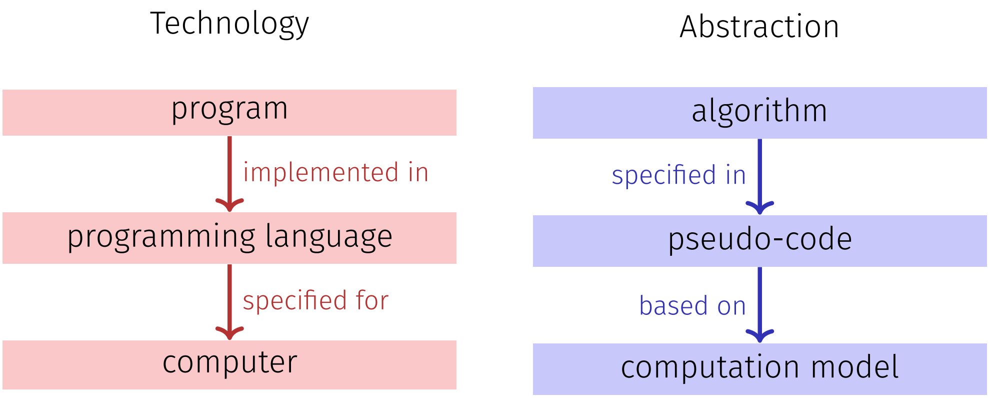
Efficiency of Algorithms, Random Access Machine Model, Function Growth, Asymptotics [Cormen et al, Kap. 2.2,3,4.2-4.4 | Ottman/Widmayer, Kap. 1.1]

Efficiency of Algorithms

Goals

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

Programs and Algorithms



Technology Model

Random Access Machine (RAM) Model

- Execution model: instructions are executed one after the other (on one processor core).

Technology Model

Random Access Machine (RAM) Model

- Execution model: instructions are executed one after the other (on one processor core).
- Memory model: constant access time (big array)

Technology Model

Random Access Machine (RAM) Model

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

Technology Model

Random Access Machine (RAM) Model

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

Technology Model

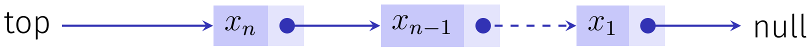
Random Access Machine (RAM) Model

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

For Dynamic Data Strcutures

Pointer Machine Model

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



Asymptotic behavior

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

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

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

5.2 Function growth

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

Superficially

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

More precise: asymptotic upper bound

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

Definition:[1]

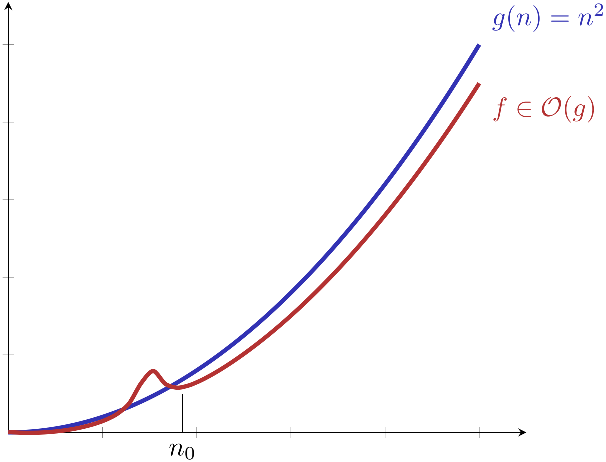
$$\begin{aligned}\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R}|\\ \exists\, c > 0, \exists n_0 \in \mathbb{N} :\\ \forall\, n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}\end{aligned}$$

Notation:

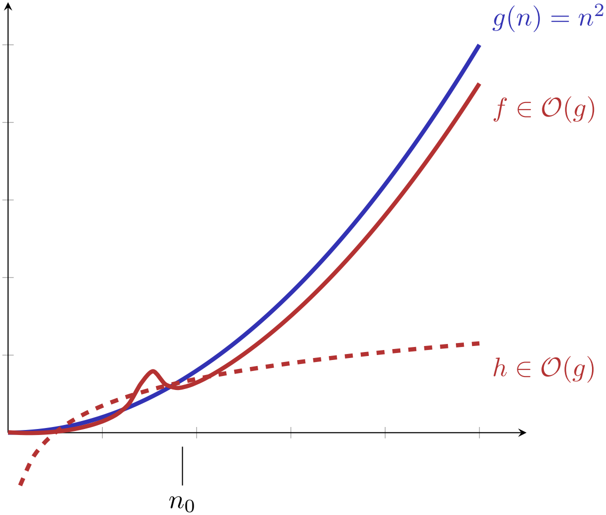
$$\mathcal{O}(g(n)) := \mathcal{O}(g(\cdot)) = \mathcal{O}(g).$$

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

Graphic



Graphic



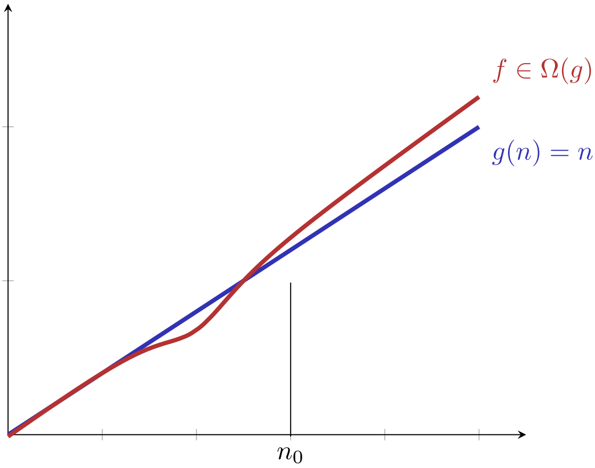
Converse: asymptotic lower bound

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

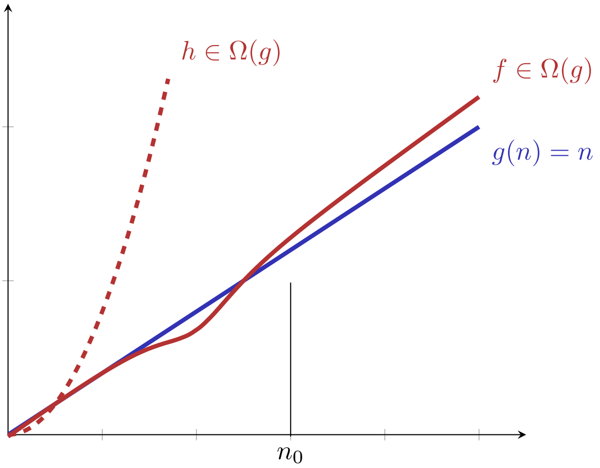
Definition:

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

Example



Example



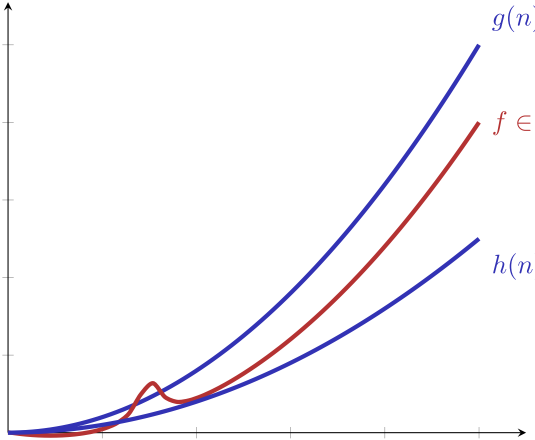
Asymptotic tight bound

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

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

Simple, closed form: exercise.

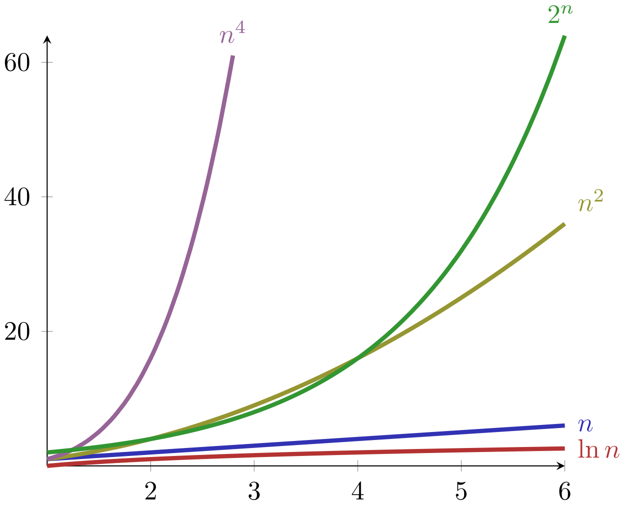
Example



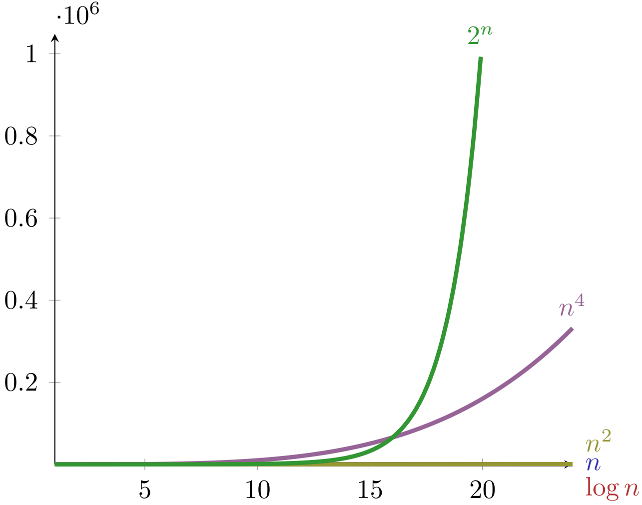
Notions of Growth

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

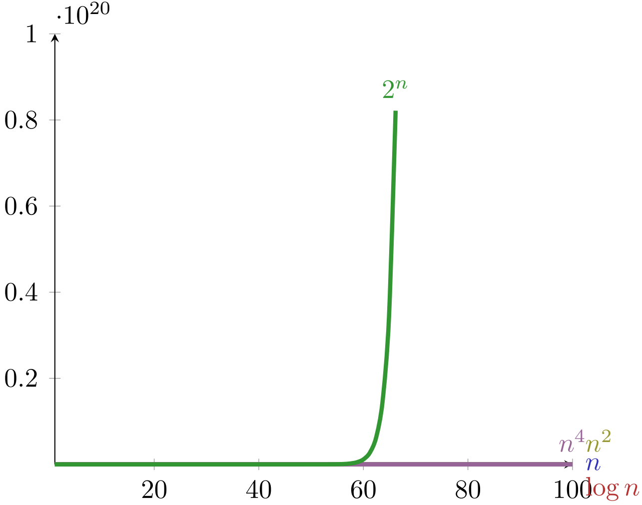
Small n



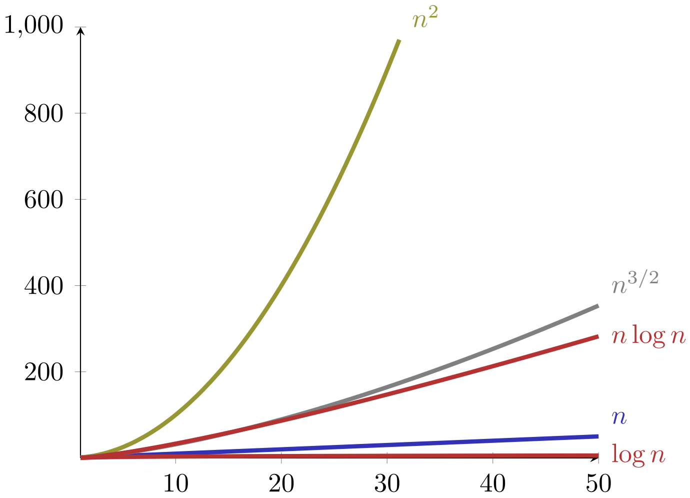
Larger n



“Large” n



Logarithms



Time Consumption

Assumption 1 Operation = $1\mu s$.

problem size	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$				
n	$1\mu s$				
$n \log_2 n$	$1\mu s$				
n^2	$1\mu s$				
2^n	$1\mu s$				

Time Consumption

Assumption 1 Operation = $1\mu s$.

problem size	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$				
n	$1\mu s$	$100\mu s$	$1/100s$	$1s$	17 minutes
$n \log_2 n$	$1\mu s$				
n^2	$1\mu s$				
2^n	$1\mu s$				

Time Consumption

Assumption 1 Operation = $1\mu s$.

problem size	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$				
n	$1\mu s$	$100\mu s$	$1/100s$	$1s$	17 minutes
$n \log_2 n$	$1\mu s$				
n^2	$1\mu s$	$1/100s$	1.7 minutes	11.5 days	317 centuries
2^n	$1\mu s$				

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
$n \log_2 n$	$1\mu s$				
n^2	$1\mu s$	$1/100s$	1.7 minutes	11.5 days	317 centuries
2^n	$1\mu s$				

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	$1\mu s$				

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	$\approx \infty$	$\approx \infty$	$\approx \infty$

About the Notation

Common casual notation

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

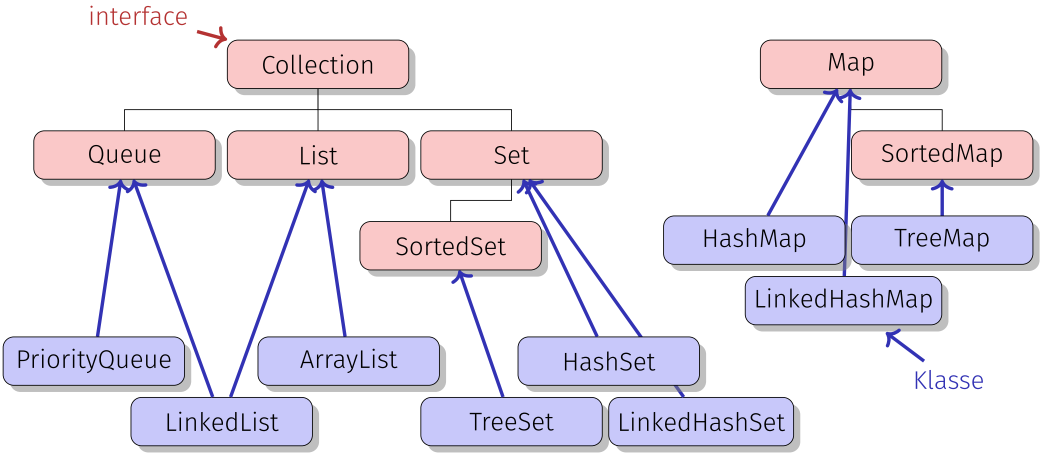
should be read as $f \in \mathcal{O}(g)$.
Clearly it holds that

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

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

We avoid this notation where it could lead to ambiguities.

Reminder: Java Collections / Maps

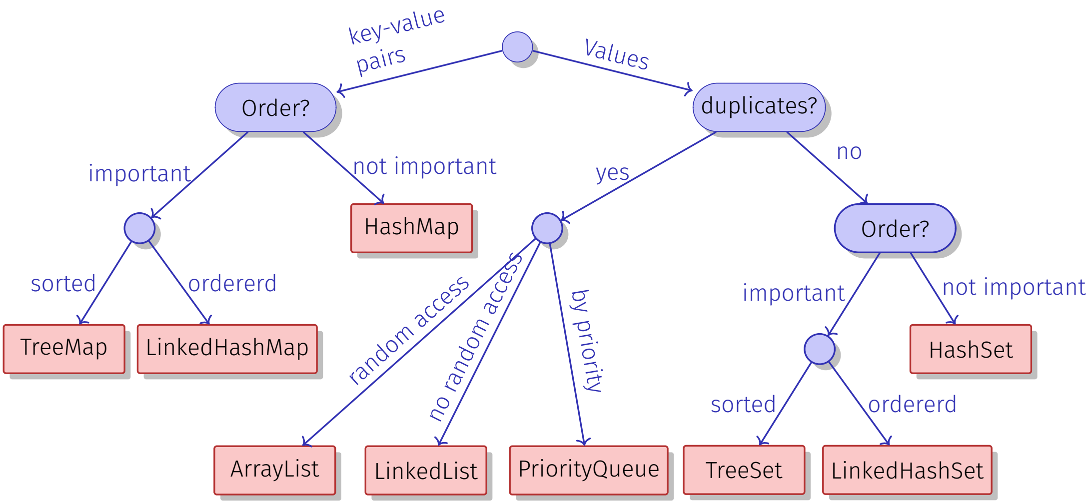


ArrayList versus LinkedList

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

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

Reminder: Decision



Asymptotic Runtimes (Java)

With our new language ($\Omega, \mathcal{O}, \Theta$), 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

Asymptotic Runtimes (Python)

Asymptotic running times

Data structure	Random Access	Insert	Iteration	Insert After Element	Search `x in S`
`list`	$\Theta(1)$	$\Theta(1)$ A	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
`set`	–	$\Theta(1)$ P	$\Theta(n)$	–	$\Theta(1)$ P
`dict`	–	$\Theta(1)$ P	$\Theta(n)$	–	$\Theta(1)$ P

A = amortized, P=expected, otherwise worst case

6. 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

  telephone book, dictionary, symbol table

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

Task: find data set by key k.

Search in Array

Provided

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

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

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

Linear Search

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

Linear Search

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

- **Best case:** 1 comparison.

Linear Search

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

- **Best case:** 1 comparison.
- **Worst case:** n comparisons.

Search in a Sorted Array

Provided

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

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

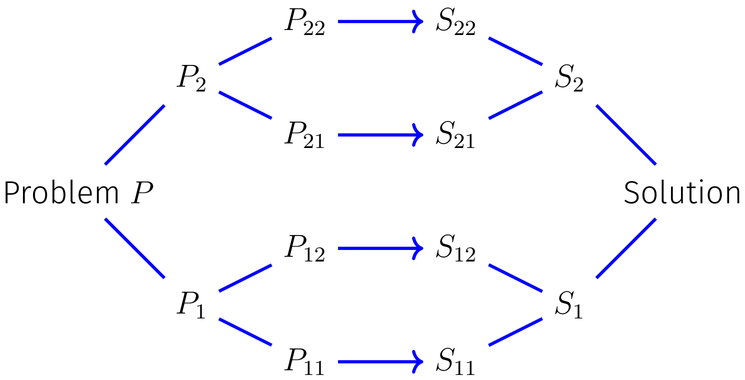
10	20	22	24	28	32	35	38	41	42
1	2	3	4	5	6	7	8	9	10

divide et impera

Divide and Conquer

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

divide et impera



Divide and Conquer!

Search $b = 23$.

Divide and Conquer!

Search $b = 23$.

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

Divide and Conquer!

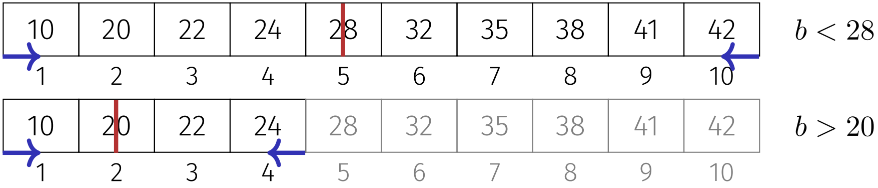
Search $b = 23$.

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

$b < 28$

Divide and Conquer!

Search $b = 23$.



Divide and Conquer!

Search $b = 23$.

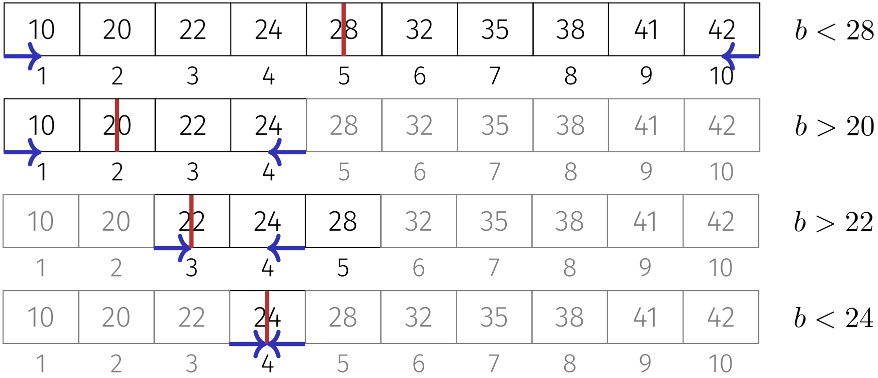
10	20	22	24	28	32	35	38	41	42	$b < 28$
1	2	3	4	5	6	7	8	9	10	

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

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

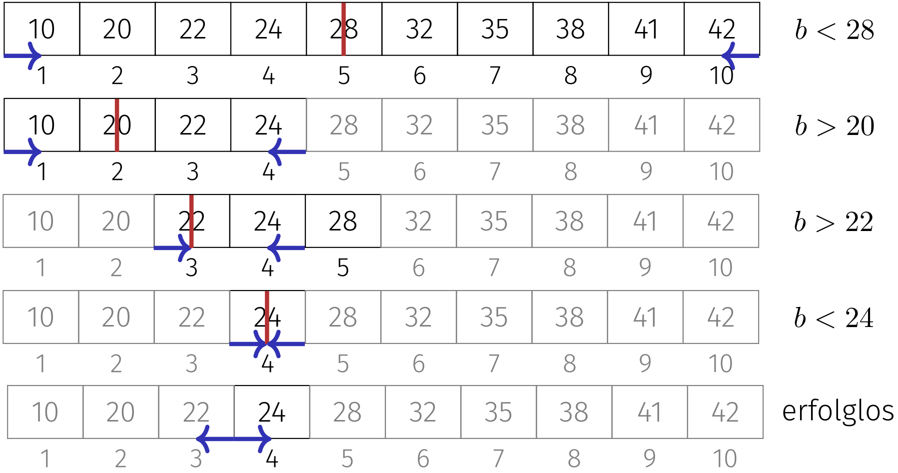
Divide and Conquer!

Search $b = 23$.



Divide and Conquer!

Search $b = 23$.



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

Input: Sorted array A of n keys. Key b. Bounds $1 \leq l, r \leq n$ mit $l \leq r$ or $l = r + 1$.

Output: Index $m \in [l, \ldots, r + 1]$, such that $A[i] \leq b$ for all $l \leq i < m$ and $A[i] \geq b$ for all $m < i \leq r$.

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

Analysis (worst case)

Recurrence ($n = 2^k$)

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

Compute: [2]

$$T(n) = T\left(\frac{n}{2}\right) + c$$

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

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]

$$T(n) = T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c$$

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

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]

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

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

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]

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

---

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

Result

Theorem 3

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

7. Sorting

Simple Sorting, Quicksort, Mergesort

Problem

Input: An array $A = (A[1], ..., A[n])$ with length n.
Output: a permutation A' of A, that is sorted: $A'[i] \leq A'[j]$ for all $1 \leq i \leq j \leq n$.

Selection Sort

5	6	2	8	4	1	$(i = 1)$
↑						

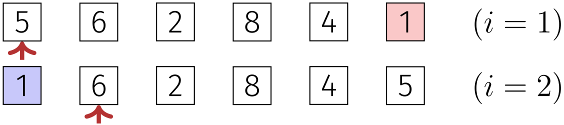
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.

Selection Sort

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

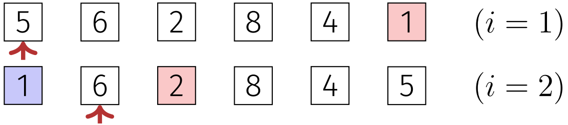
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.

Selection Sort



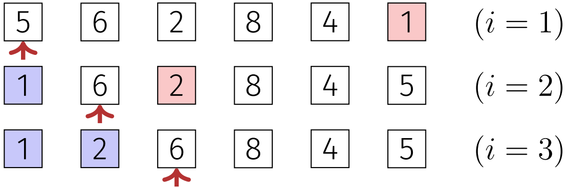
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element ($i \to i + 1$). Repeat until all is sorted. ($i = n$)

Selection Sort



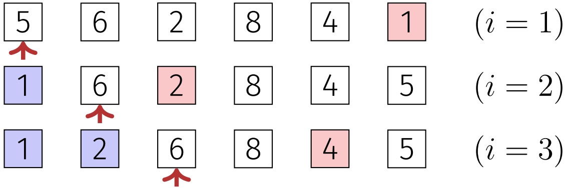
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element $(i \to i + 1)$. Repeat until all is sorted. $(i = n)$

Selection Sort



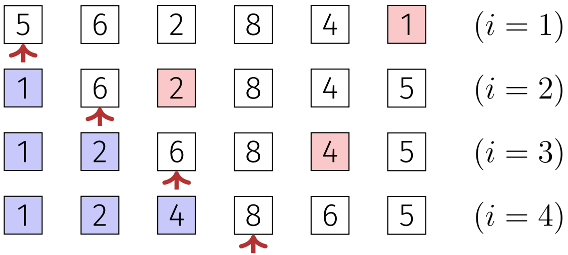
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element ($i \rightarrow i + 1$). Repeat until all is sorted. ($i = n$)

Selection Sort



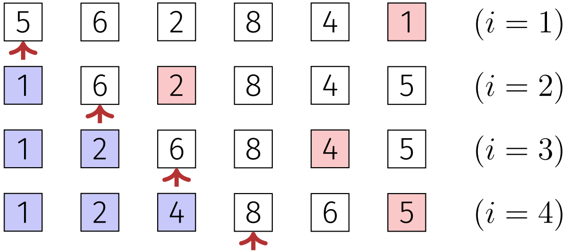
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element ($i \to i+1$). Repeat until all is sorted. ($i = n$)

Selection Sort



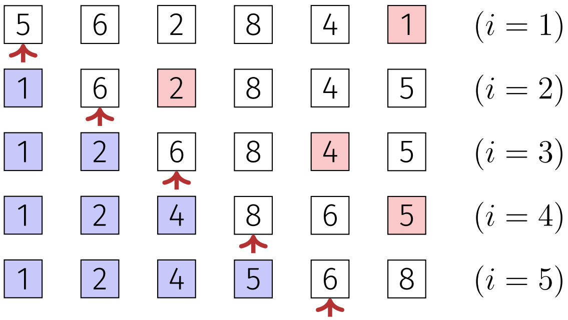
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element $(i \rightarrow i + 1)$. Repeat until all is sorted. $(i = n)$

Selection Sort



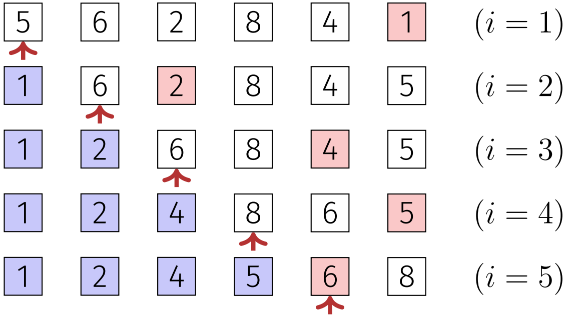
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element $(i \to i+1)$. Repeat until all is sorted. $(i = n)$

Selection Sort



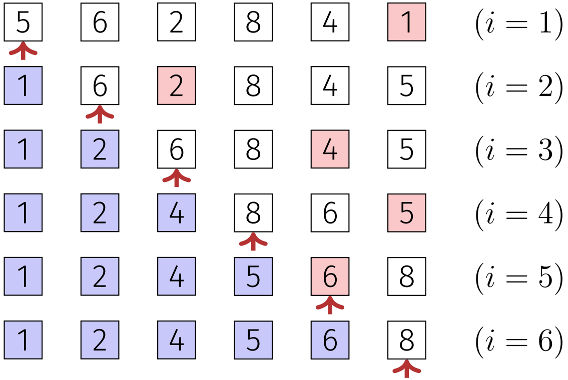
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element ($i \to i+1$). Repeat until all is sorted. ($i = n$)

Selection Sort



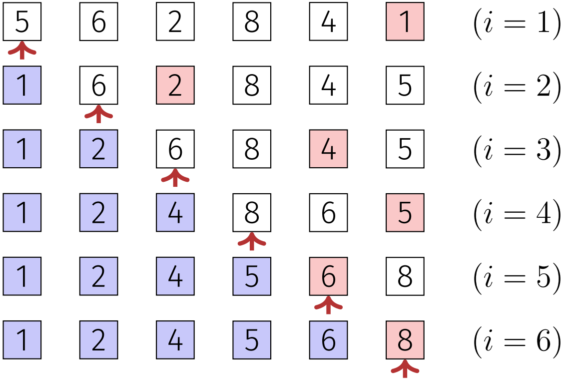
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element $(i \rightarrow i + 1)$. Repeat until all is sorted. $(i = n)$

Selection Sort



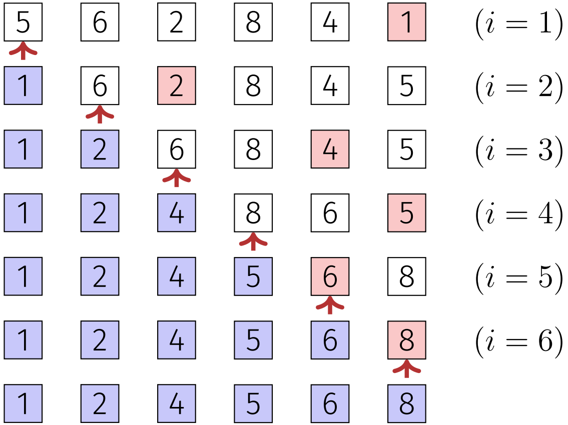
- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element $(i \to i+1)$. Repeat until all is sorted. $(i = n)$

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. ($i = n$)

Selection Sort



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

Algorithm: Selection Sort

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

```
for i ← 1 to n − 1 do
    p ← i
    for j ← i + 1 to n do
        if A[j] < A[p] then
            p ← j;
    swap(A[i], A[p])
```

Analysis

Number comparisons in worst case:

Analysis

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

Analysis

Number comparisons in worst case: $\Theta(n^2)$.
Number swaps in the worst case: $n - 1 = \Theta(n)$

Insertion Sort

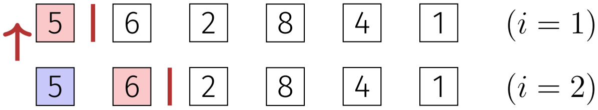
5 | 6 2 8 4 1 $(i = 1)$

Insertion Sort

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

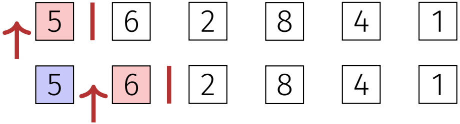
- Iterative procedure: $i = 1...n$

Insertion Sort



- Iterative procedure: $i = 1...n$
- Determine insertion position for element i.

Insertion Sort

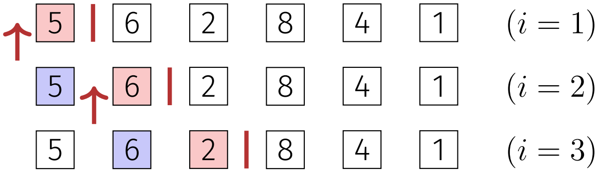


$(i = 1)$

$(i = 2)$

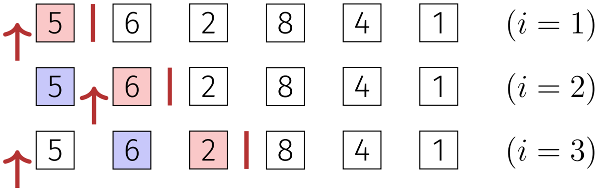
- Iterative procedure: $i = 1...n$
- Determine insertion position for element i.
- Insert element i

Insertion Sort



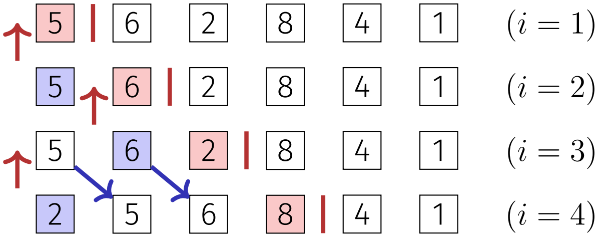
- Iterative procedure: $i = 1...n$
- Determine insertion position for element i.
- Insert element i

Insertion Sort



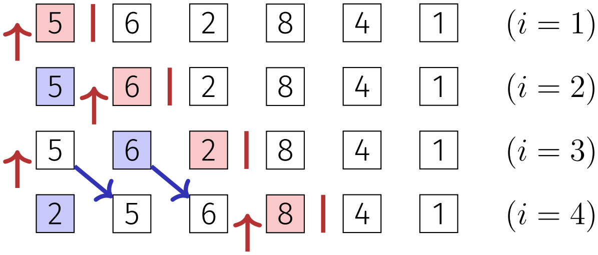
- Iterative procedure: $i = 1...n$
- Determine insertion position for element i.
- Insert element i

Insertion Sort



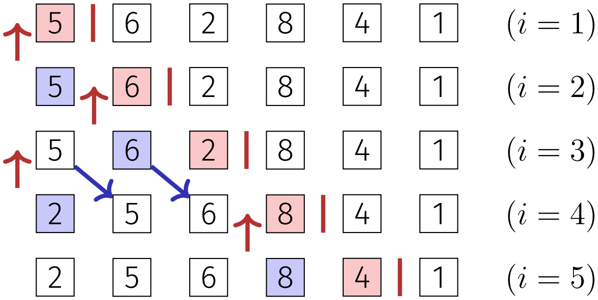
- Iterative procedure: $i = 1...n$
- Determine insertion position for element i.
- Insert element i array block movement potentially required

Insertion Sort



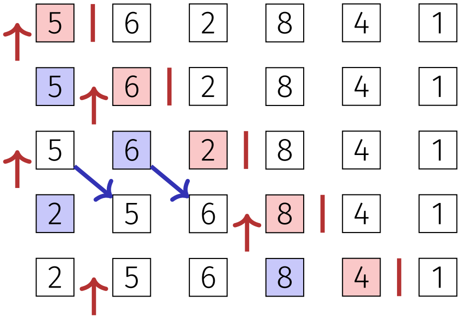
- Iterative procedure: $i = 1...n$
- Determine insertion position for element i.
- Insert element i array block movement potentially required

Insertion Sort



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

Insertion Sort



$(i = 1)$

$(i = 2)$

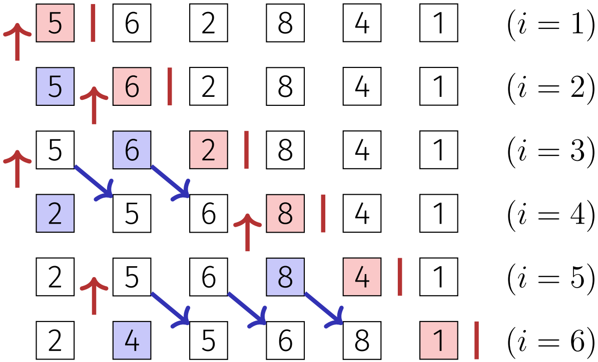
$(i = 3)$

$(i = 4)$

$(i = 5)$

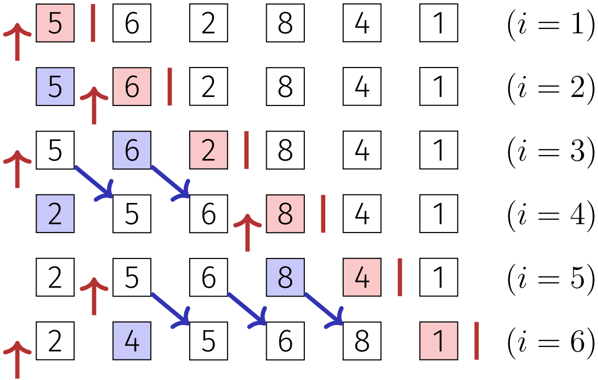
- Iterative procedure: $i = 1...n$
- Determine insertion position for element i.
- Insert element i array block movement potentially required

Insertion Sort



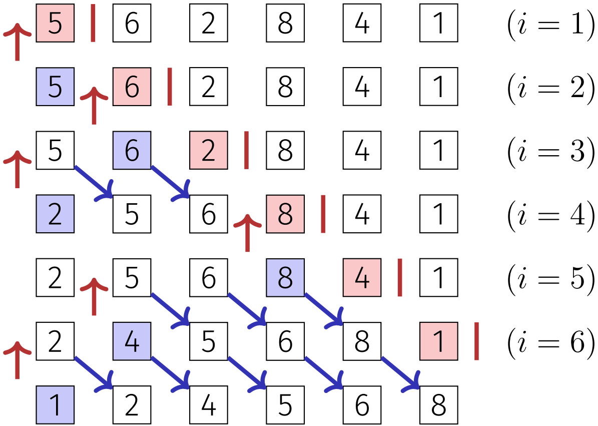
- Iterative procedure: $i = 1...n$
- Determine insertion position for element i.
- Insert element i array block movement potentially required

Insertion Sort



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

Insertion Sort



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

Insertion Sort

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

Insertion Sort

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

Many element movements in the worst case.

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

Insertion Sort

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

Many element movements in the worst case.

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

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

Algorithm: Insertion Sort

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

```
for i ← 2 to n do
    x ← A[i]
    p ← BinarySearch(A, 1, i − 1, x); // Smallest p ∈ [1, i] with A[p] ≥ x
    for j ← i − 1 downto p do
        A[j + 1] ← A[j]
    A[p] ← x
```

7.1 Mergesort

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

Mergesort

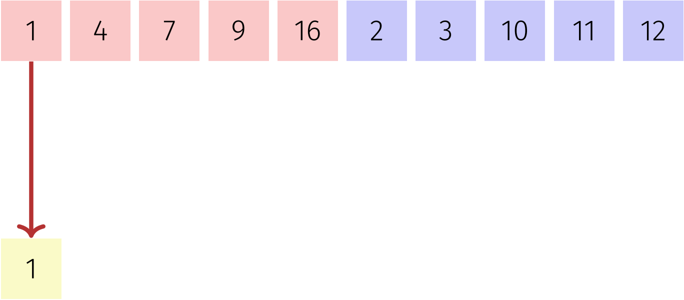
Divide and Conquer!

- 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 $\mathcal{O}(n)$.

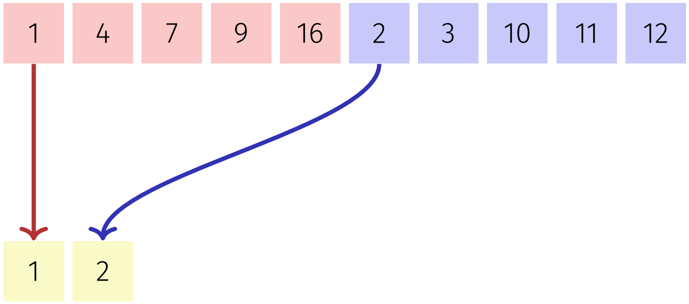
Merge

1	4	7	9	16	2	3	10	11	12

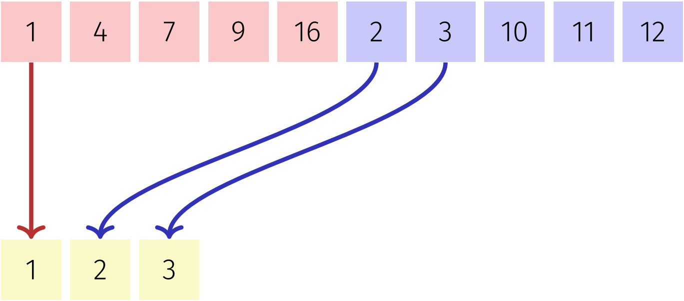
Merge



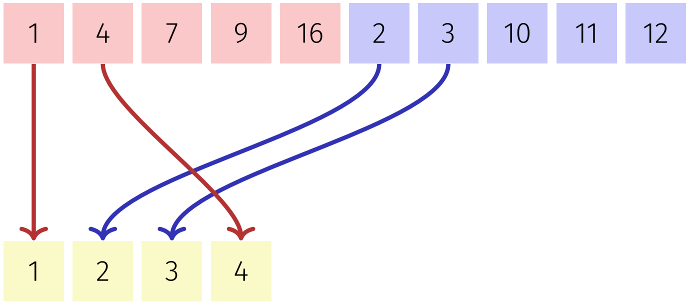
Merge



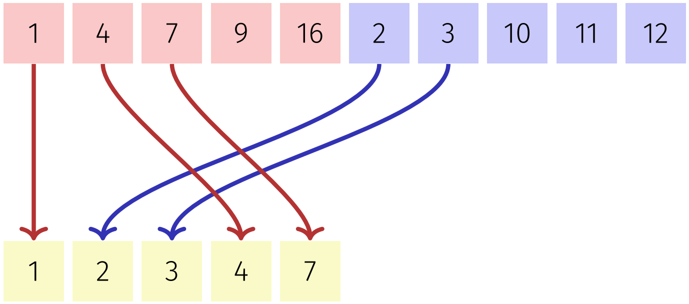
Merge



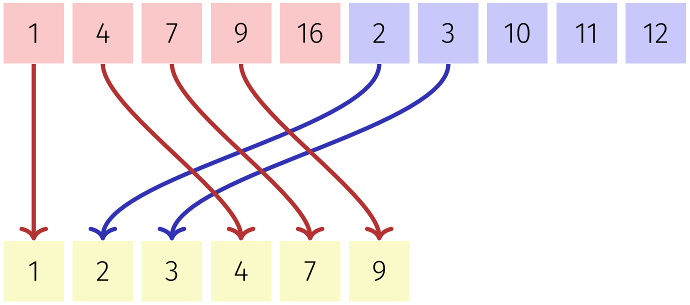
Merge



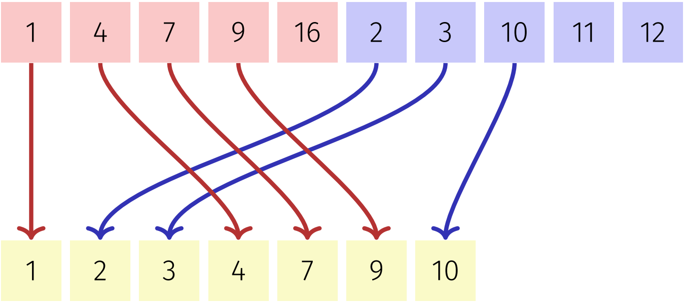
Merge



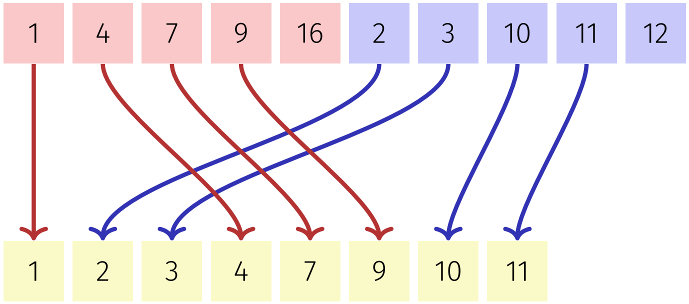
Merge



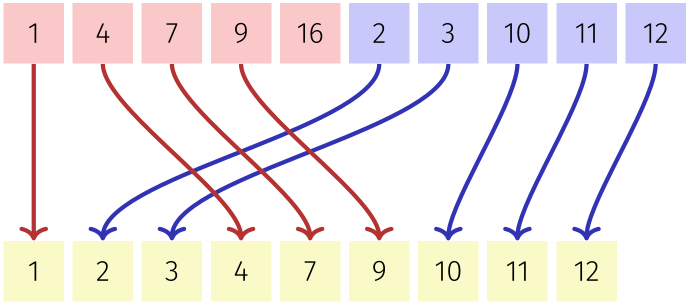
Merge



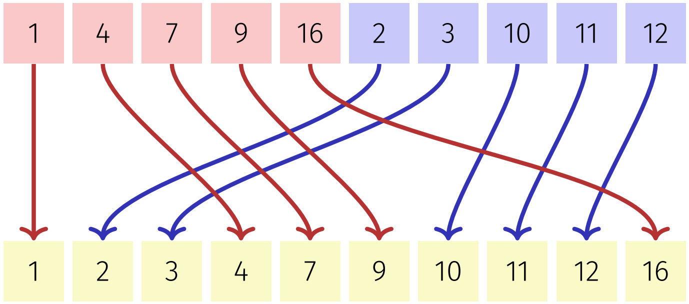
Merge



Merge



Merge



Algorithm **Merge**(A, l, m, r)

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

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

$B \leftarrow$ new Array$(r - l + 1)$
$i \leftarrow l$; $j \leftarrow m + 1$; $k \leftarrow 1$
while $i \leq m$ and $j \leq r$ **do**
    if $A[i] \leq A[j]$ **then** $B[k] \leftarrow A[i]$; $i \leftarrow i + 1$
    else $B[k] \leftarrow A[j]$; $j \leftarrow j + 1$
    $k \leftarrow k + 1$;
while $i \leq m$ **do** $B[k] \leftarrow A[i]$; $i \leftarrow i + 1$; $k \leftarrow k + 1$
while $j \leq r$ **do** $B[k] \leftarrow A[j]$; $j \leftarrow j + 1$; $k \leftarrow k + 1$
for $k \leftarrow l$ **to** r **do** $A[k] \leftarrow B[k - l + 1]$

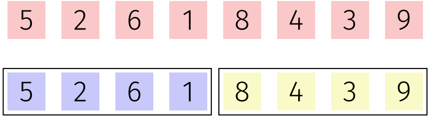
Mergesort

5	2	6	1	8	4	3	9

Mergesort

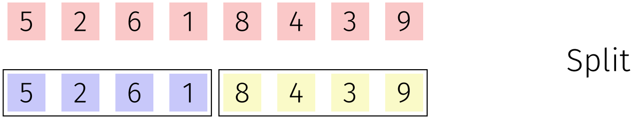
Split

Mergesort

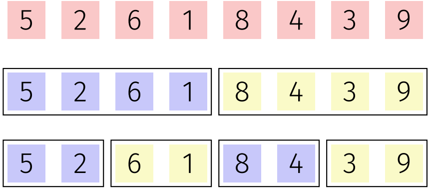


Split

Mergesort



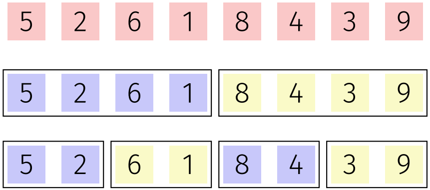
Mergesort



Split

Split

Mergesort

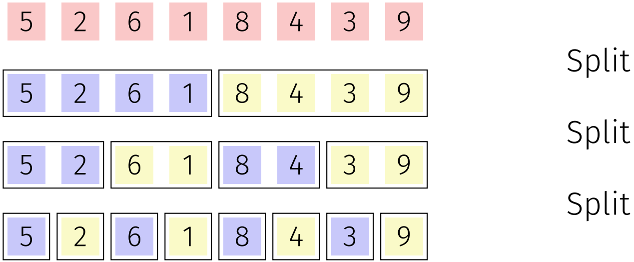


Split

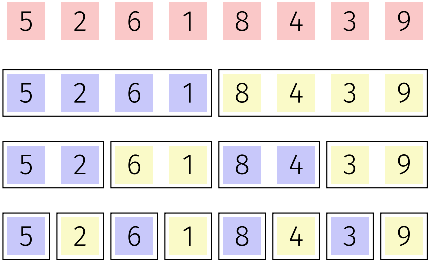
Split

Split

Mergesort



Mergesort



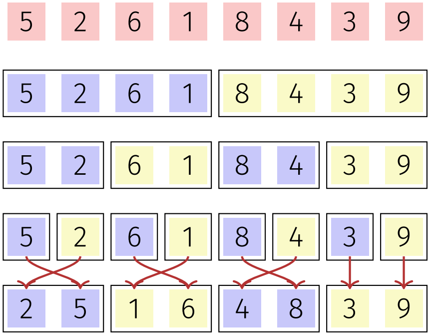
Split

Split

Split

Merge

Mergesort



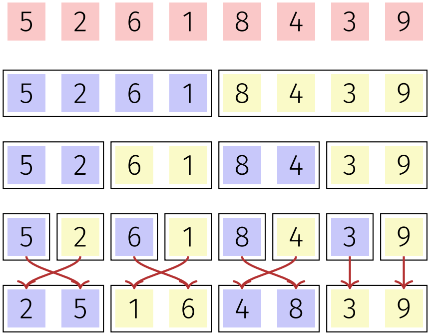
Split

Split

Split

Merge

Mergesort



Split

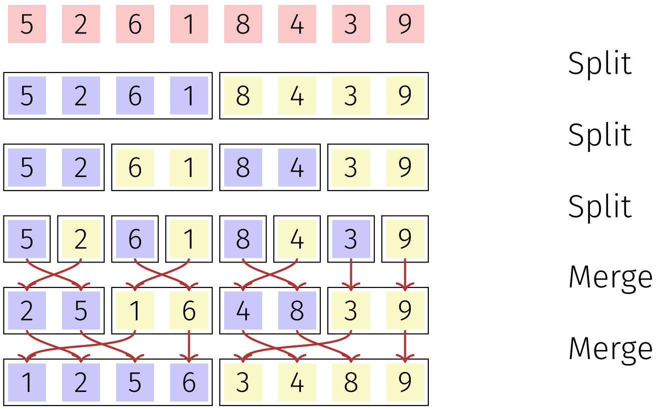
Split

Split

Merge

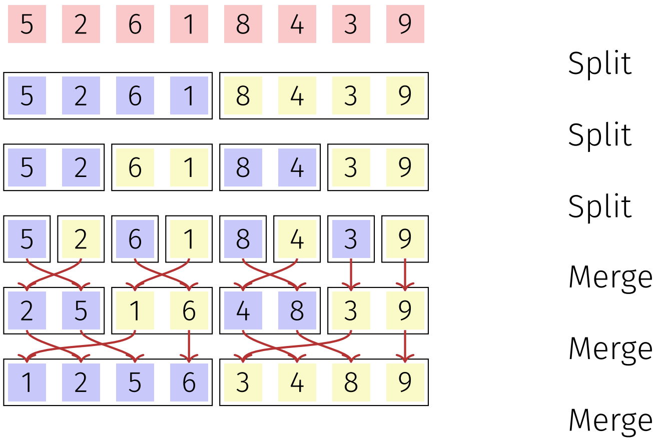
Merge

Mergesort

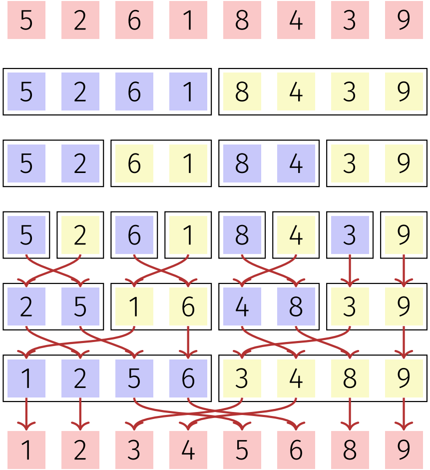


5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
2 5 1 6 4 8 3 9
1 2 5 6 3 4 8 9
Split
Split
Split
Merge
Merge

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
Split
Split
Split
Merge
Merge
Merge

Mergesort



Split

Split

Split

Merge

Merge

Merge

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

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

```
if l < r then
    m ← ⌊(l + r)/2⌋              // middle position
    Mergesort(A, l, m)           // sort lower half
    Mergesort(A, m + 1, r)       // sort higher half
    Merge(A, l, m, r)            // Merge subsequences
```

Analysis

Recursion equation for the number of comparisons and key movements:

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

Analysis

Recursion equation for the number of comparisons and key movements:

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

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

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

7.2 Quicksort

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

Quicksort

What is the disadvantage of Mergesort?

Quicksort

What is the disadvantage of Mergesort?

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

Quicksort

What is the disadvantage of Mergesort?

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

How could we reduce the merge costs?

Quicksort

What is the disadvantage of Mergesort?

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

How could we reduce the merge costs?

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

How?

Quicksort

What is the disadvantage of Mergesort?

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

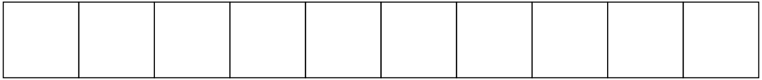
How could we reduce the merge costs?

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

How?

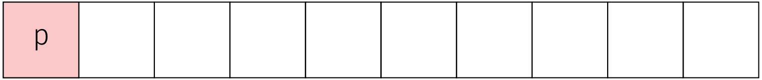
Pivot and Partition!

Use a pivot



Use a pivot

1. Choose a (an arbitrary) **pivot** p



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 $A[i] > p$

p	$>$	$\leq$	$\leq$	$>$	$>$	$\leq$	$\leq$	$>$	$\leq$

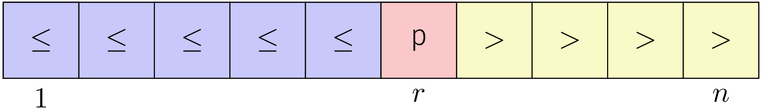
Use a pivot

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

p	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$>$	$>$	$>$	$>$

Use a pivot

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



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

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

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

return l-1

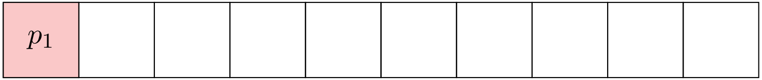
Algorithm `Quicksort`(A, l, r)

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

if $l < r$ **then**
  Choose pivot $p \in A[l, \dots, r]$
  $k \leftarrow$ `Partition`(A, l, r, p)
  `Quicksort`$(A, l, k-1)$
  `Quicksort`$(A, k+1, r)$

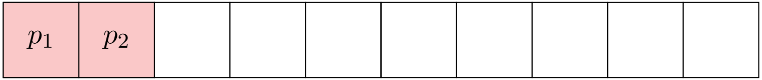
Choice of the pivot.

The minimum is a bad pivot: worst case $\Theta(?)$



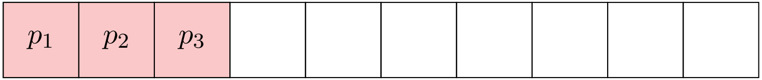
Choice of the pivot.

The minimum is a bad pivot: worst case $\Theta(?)$



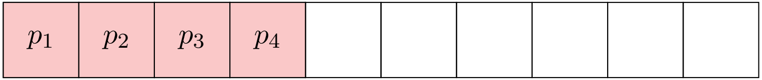
Choice of the pivot.

The minimum is a bad pivot: worst case $\Theta(?)$



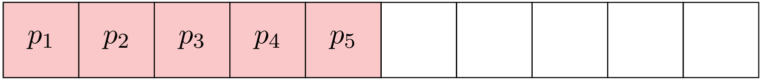
Choice of the pivot.

The minimum is a bad pivot: worst case $\Theta(?)$



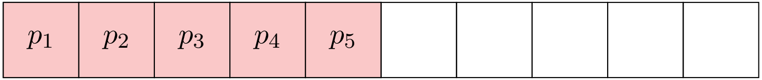
Choice of the pivot.

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

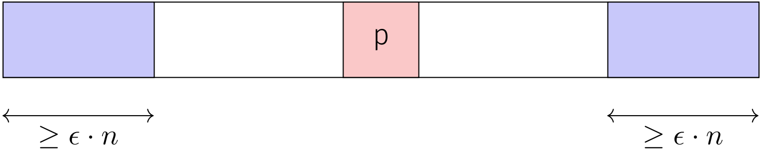


Choice of the pivot.

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

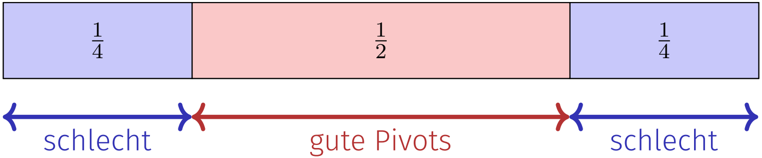


A good pivot has a linear number of elements on both sides.



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 trials[3]: $1/\rho = 2$

[3] Expected value of the geometric distribution:

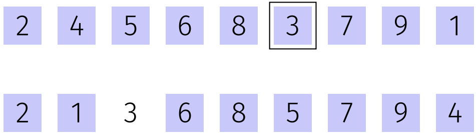
Quicksort (arbitrary pivot)

2	4	5	6	8	3	7	9	1

Quicksort (arbitrary pivot)

2	4	5	6	8	3	7	9	1

Quicksort (arbitrary pivot)

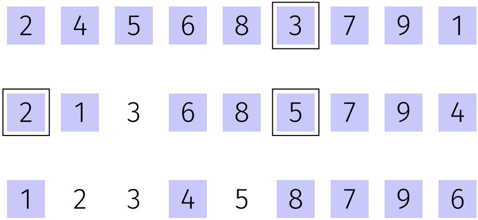


Quicksort (arbitrary pivot)

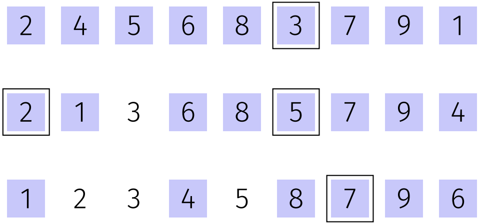
2 4 5 6 8 3 7 9 1

2 1 3 6 8 5 7 9 4

Quicksort (arbitrary pivot)



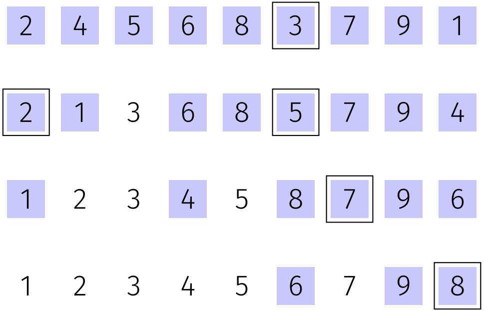
Quicksort (arbitrary pivot)



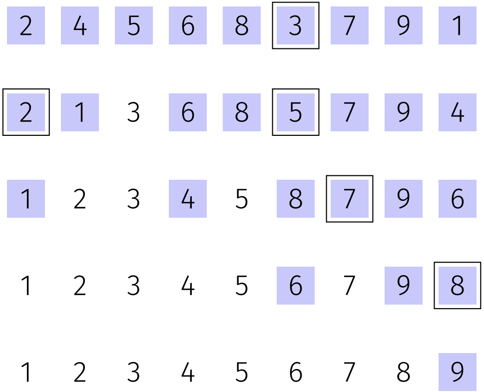
Quicksort (arbitrary pivot)

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

Quicksort (arbitrary pivot)



Quicksort (arbitrary pivot)



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.

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 4

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: Median3($A[l], A[r], A[\lfloor l + r/2 \rfloor]$).

8. Binary Search Trees

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

Trees

Trees are

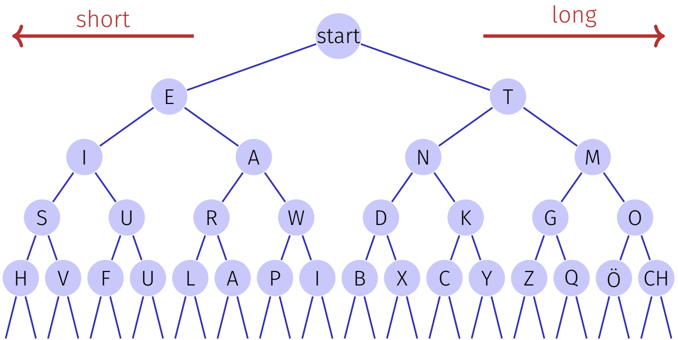
- Generalized lists: nodes can have more than one successor
- Special graphs: graphs consist of nodes and edges. A tree is a fully connected, directed, acyclic graph.

Trees

Use

- Decision trees: hierarchic representation of decision rules
- syntax trees: parsing and traversing of expressions, e.g. in a compiler
- Code tress: representation of a code, e.g. morse alphabet, huffman code
- Search trees: allow efficient searching for an element by value

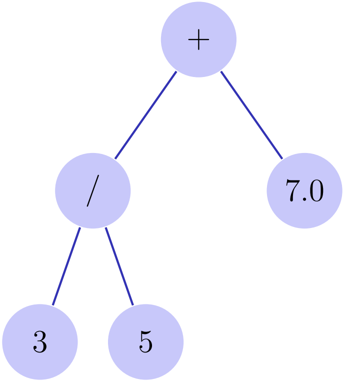
Examples



Morsealphabet

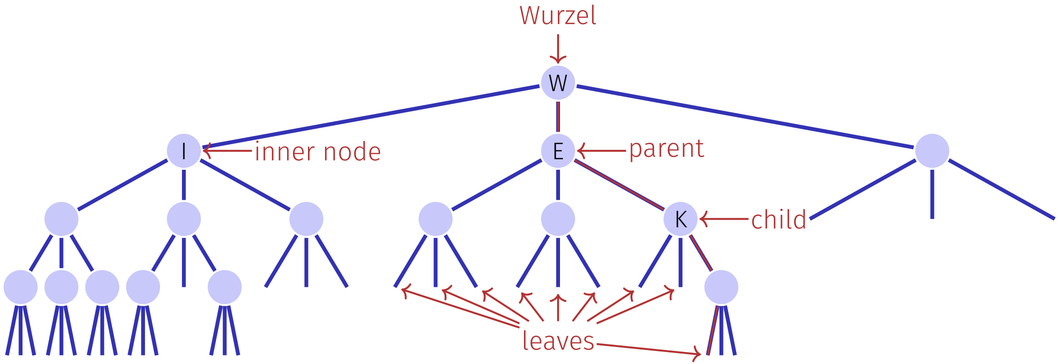
Examples

$$3/5 + 7.0$$



Expression tree

Nomenclature



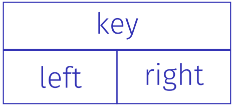
- Order of the tree: maximum number of child nodes, here: 3
- Height of the tree: maximum path length root – leaf (here: 4)

Binary Trees

A binary tree is

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

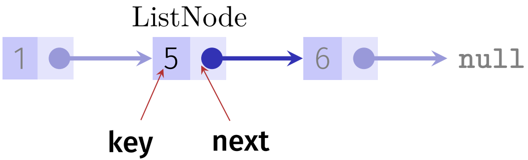
In each inner node **`v`** we store



- a key **`v.key`** and
- two nodes **`v.left`** and **`v.right`** to the roots of the left and right subtree.

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

Linked List Node in Python



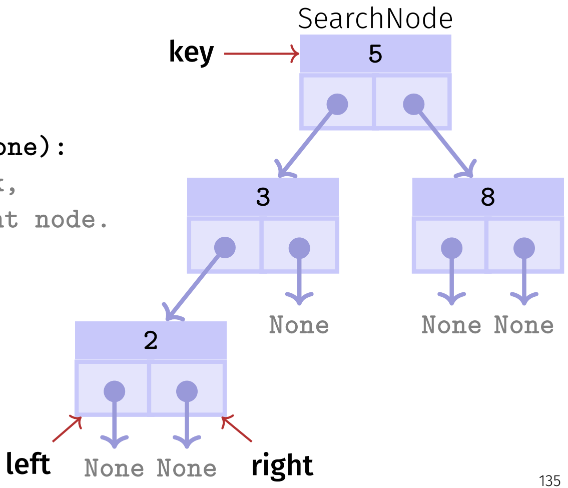

```
class ListNode:
   # entries key, next implicit via constructor

  def __init__(self, key , next = None):
    """Constructor that takes a key and, optionally, next."""
    self.key = key
    self.next = next
}
```

Now: tree nodes in Python

```
class SearchNode:
# implicit entries key, left, right

  def __init__(self, k, l=None, r=None):
    # Constructor that takes a key k,
    # and optionally a left and right node.
    self.key = k
    self.left, self.right = l, r
```

Now: tree nodes in Python

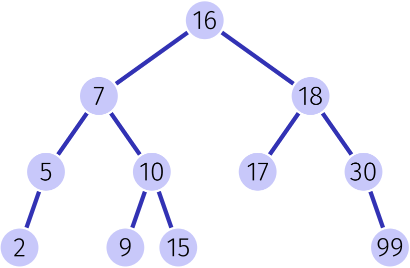
```
class SearchNode:
# implicit entries key, left, right

  def __init__(self, k, l=None, r=None):
    # Constructor that takes a key k,
    # and optionally a left and right node.
    self.key = k
    self.left, self.right = l, r
```



Binary search tree

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

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

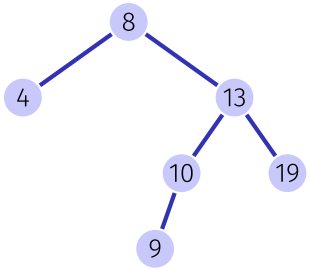


Searching

Input: Binary search tree with root r, key k
Output: Node v with $v.\text{key} = k$ or **null**

$v \leftarrow r$
while $v \neq$ **null do**
  if $k = v.\text{key}$ **then**
    return v
  else if $k < v.\text{key}$ **then**
    $v \leftarrow v.\text{left}$
  else
    $v \leftarrow v.\text{right}$

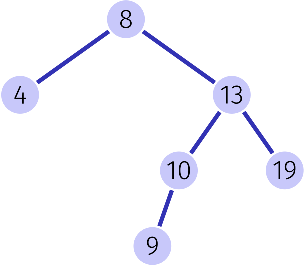
return null



Searching

Input: Binary search tree with root r, key k
Output: Node v with $v.\text{key} = k$ or **null**

$v \leftarrow r$
while $v \neq$ **null do**
  if $k = v.\text{key}$ **then**
    return v
  else if $k < v.\text{key}$ **then**
    $v \leftarrow v.\text{left}$
  else
    $v \leftarrow v.\text{right}$
return null



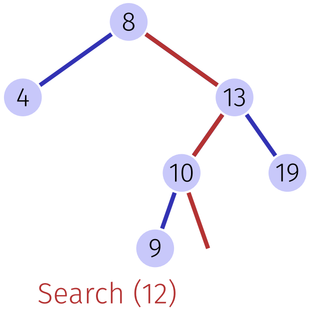
Search (12)

Searching

Input: Binary search tree with root r, key k
Output: Node v with $v.\text{key} = k$ or **null**

$v \leftarrow r$
while $v \neq$ **null do**
  if $k = v.\text{key}$ **then**
    return v
  else if $k < v.\text{key}$ **then**
    $v \leftarrow v.\text{left}$
  else
    $v \leftarrow v.\text{right}$

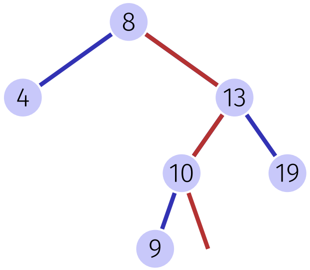
return null



Searching

Input: Binary search tree with root r, key k
Output: Node v with $v.\mathrm{key} = k$ or **null**

$v \leftarrow r$
while $v \neq$ **null do**
  if $k = v.\mathrm{key}$ **then**
    return v
  else if $k < v.\mathrm{key}$ **then**
    $v \leftarrow v.\mathrm{left}$
  else
    $v \leftarrow v.\mathrm{right}$
return null



Search (12) $\to$ **null**

Searching in Python

```python
def findNode(root, key):
    n = root
    while n != None and n.key != key:
        if key < n.key:
            n = n.left
        else:
            n = n.right
    return n
```

Height of a tree

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

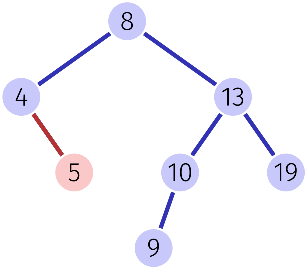
$$h(r) = \begin{cases} 0 & \text{if } r = \textbf{null} \\ 1 + \max\{h(r.\text{left}), h(r.\text{right})\} & \text{otherwise.} \end{cases}$$

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

Insertion of a key

Insertion of the key k

- Search for k
- If successful search: e.g. output error
- Of no success: insert the key at the leaf reached



Insert (5)

Insert Nodes in Python

```python
def addNode(root, key):
  n = root
  if n == None:
    root = Node(key)
  while n.key != key:
    if key < n.key:
      if n.left == None:
        n.left = Node(key)
      n = n.left
    else:
      if n.right == None:
        n.right = Node(key)
      n = n.right
  return root
```

Tree in Python

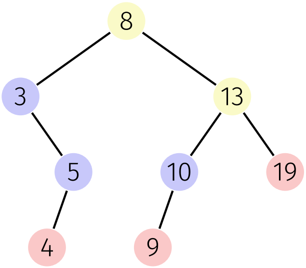
```python
class Tree:
    def __init__(self):
        self.root = None

    def find(self,key):
        return findNode(self.root, key)

    def has(self,key):
        return self.find(key) != None

    def add(self,key):
        self.root = addNode(self.root, key)

    # ....
```

Remove node

Three cases possible:

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

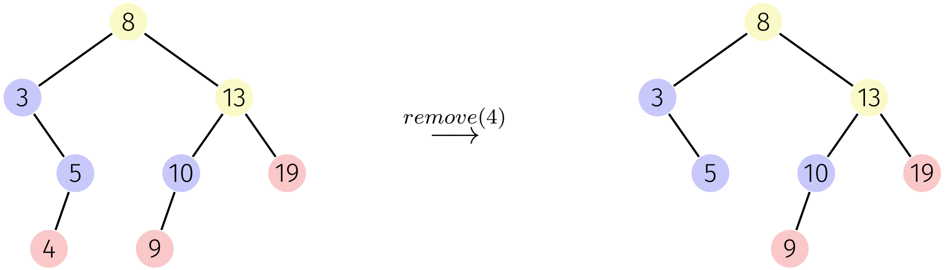
[Leaves do not count here]



Remove node

Node has no children

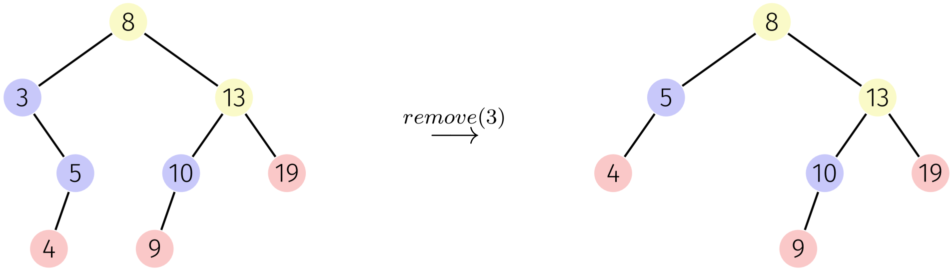
Simple case: replace node by leaf.



Remove node

Node has one child

Also simple: replace node by single child.



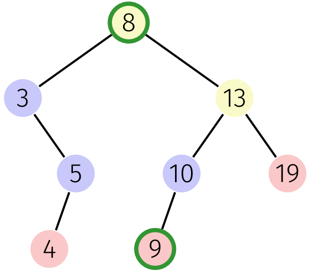
Remove node

Node `v` has two children

The following observation helps: the smallest key in the right subtree `v.right` (the **symmetric successor** of `v`)

- is smaller than all keys in `v.right`
- is greater than all keys in `v.left`
- and cannot have a left child.

Solution: replace `v` by its symmetric successor.

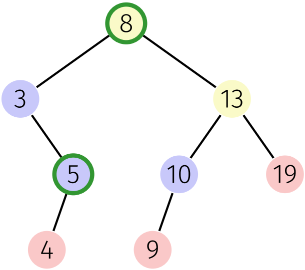


By symmetry...

Node **v** has two children

Also possible: replace **v** by its symmetric predecessor.

Implementation: devil is in the detail!



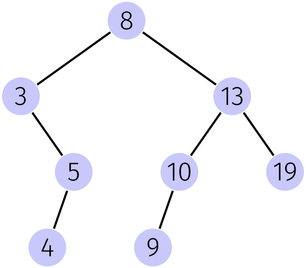
Algorithm SymmetricSuccessor(v)

Input: Node v of a binary search tree.
Output: Symmetric successor of v

$w \leftarrow v.\text{right}$
$x \leftarrow w.\text{left}$
while $x \neq$ **null do**
    $w \leftarrow x$
    $x \leftarrow x.\text{left}$
return w

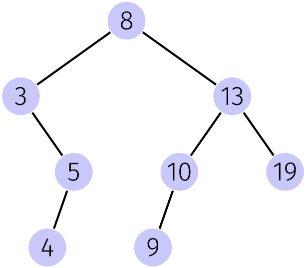
Traversal possibilities

- preorder: v, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.



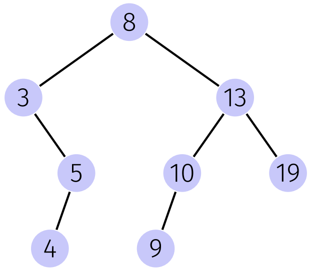
Traversal possibilities

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



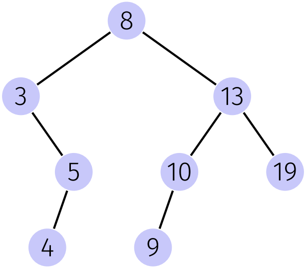
Traversal possibilities

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



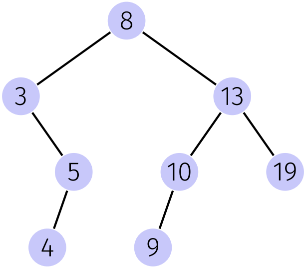
Traversal possibilities

- 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



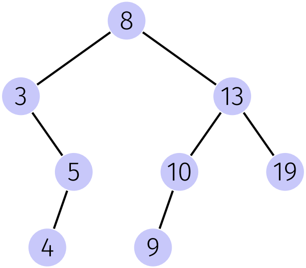
Traversal possibilities

- 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 $T_{\text{right}}(v)$.

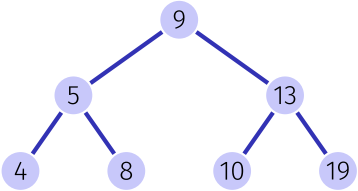


Traversal possibilities

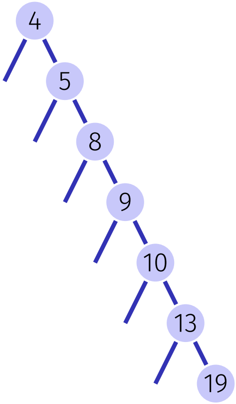
- preorder: v, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- postorder: $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$, then v.
  4, 5, 3, 9, 10, 19, 13, 8
- inorder: $T_{\text{left}}(v)$, then v, then $T_{\text{right}}(v)$.
  3, 4, 5, 8, 9, 10, 13, 19



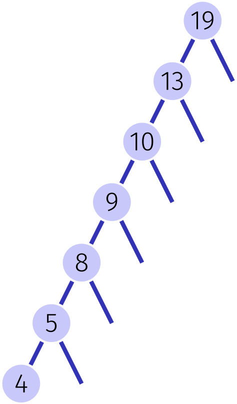
Degenerated search trees



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 $\mathcal{O}(\sqrt{n})$.

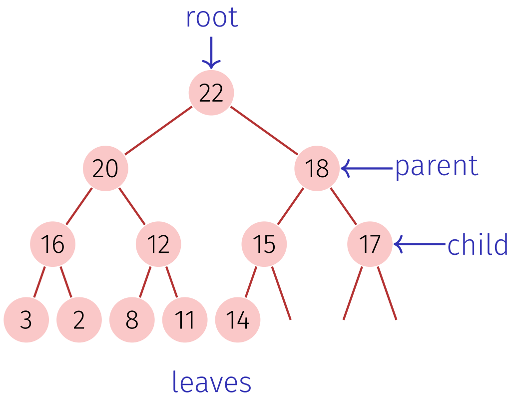
Balanced trees make sure (e.g. with *rotations*) during insertion or deletion that the tree stays balanced and provide a $\mathcal{O}(\log n)$ Worst-case guarantee.

9. Heaps

Datenstruktur optimiert zum schnellen Extrahieren von Minimum oder Maximum und Sortieren. [Ottman/Widmayer, Kap. 2.3, Cormen et al, Kap. 6]

[Max-]Heap*

Binary tree with the following properties

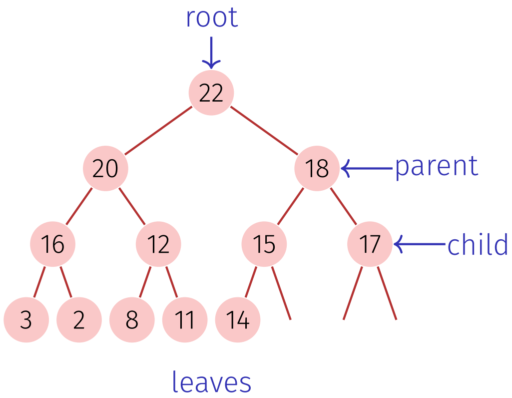


*Heap(data structure), not: as in “heap and stack” (memory allocation)

[Max-]Heap*

Binary tree with the following properties

1. complete up to the lowest level

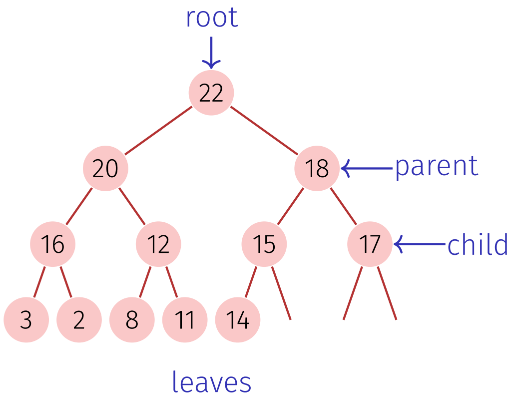


*Heap(data structure), not: as in “heap and stack” (memory allocation)

[Max-]Heap*

Binary tree with the following properties

1. complete up to the lowest level
2. Gaps (if any) of the tree in the last level to the right

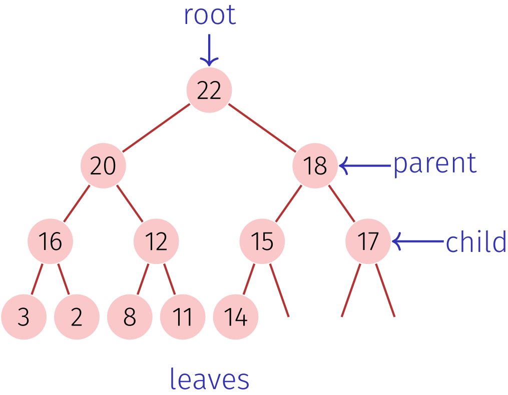


*Heap(data structure), not: as in “heap and stack” (memory allocation)

[Max-]Heap*

Binary tree with the following properties

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

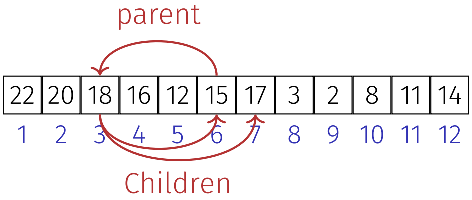


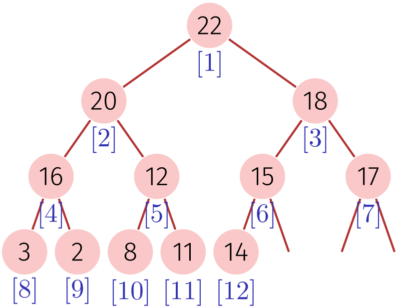
*Heap(data structure), not: as in "heap and stack" (memory allocation)

Heap as Array

Tree → Array:

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





Depends on the starting index[4]

[4]For 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

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

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

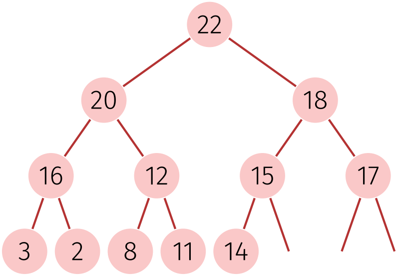
with $\sum_{i=0}^{h-1} 2^i = 2^h - 1$:

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

thus

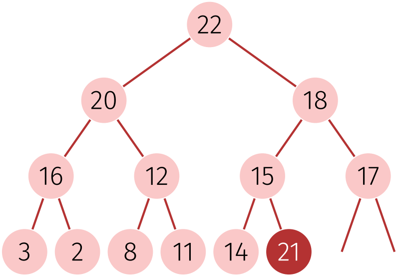
$$H(n) = \lceil \log_2(n + 1) \rceil.$$

Insert



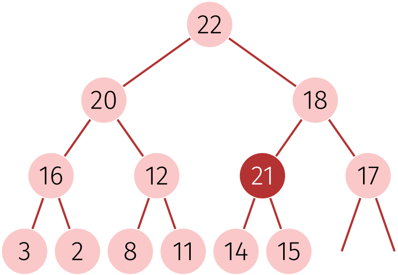
Insert

- Insert new element at the first free position. Potentially violates the heap property.



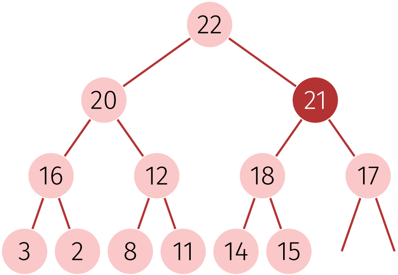
Insert

- Insert new element at the first free position. Potentially violates the heap property.
- Reestablish heap property: climb successively



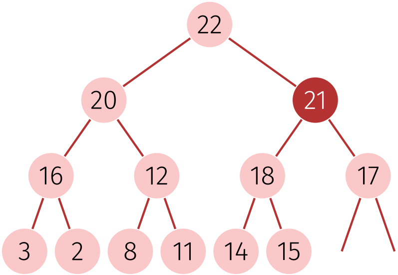
Insert

- Insert new element at the first free position. Potentially violates the heap property.
- Reestablish heap property: climb successively



Insert

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



Algorithm Sift-Up(A, m)

Input: Array A with at least m elements and Max-Heap-Structure on $A[1, \dots, m-1]$

Output: Array A with Max-Heap-Structure on $A[1, \dots, m]$.

$v \leftarrow A[m]$ // value

$c \leftarrow m$ // current position (child)

$p \leftarrow \lfloor c/2 \rfloor$ // parent node

while $c > 1$ and $v > A[p]$ **do**

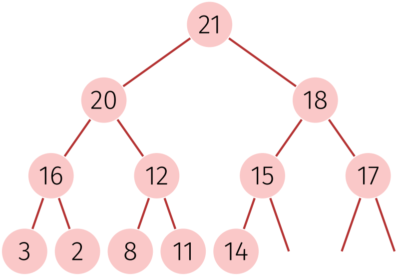
  $A[c] \leftarrow A[p]$ // Value parent node $\rightarrow$ current node

  $c \leftarrow p$ // parent node $\rightarrow$ current node

  $p \leftarrow \lfloor c/2 \rfloor$

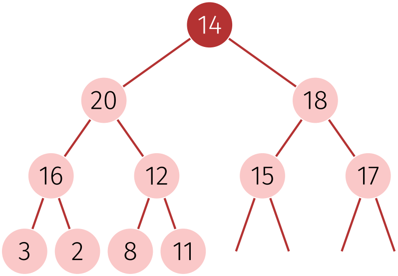
$A[c] \leftarrow v$ // value $\rightarrow$ root of the (sub)tree

Remove the maximum



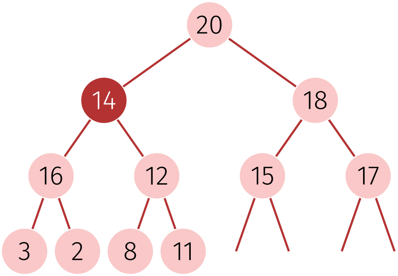
Remove the maximum

- Replace the maximum by the lower right element



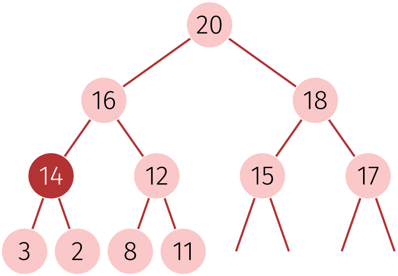
Remove the maximum

- Replace the maximum by the lower right element
- Reestablish heap property: sink successively (in the direction of the greater child)



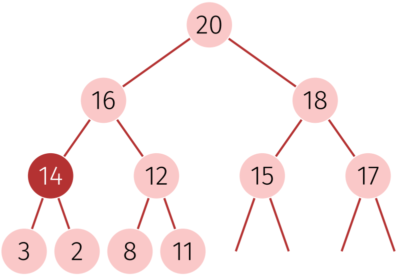
Remove the maximum

- Replace the maximum by the lower right element
- Reestablish heap property: sink successively (in the direction of the greater child)



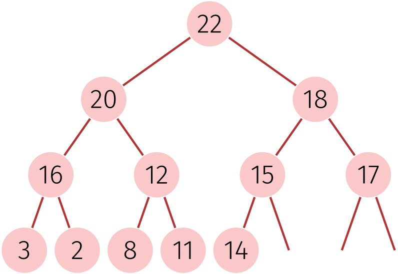
Remove the maximum

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



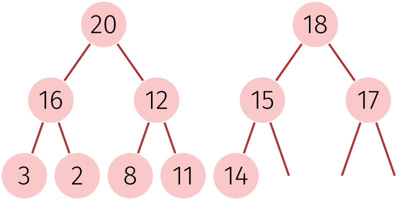
Why this is correct: Recursive heap structure

A heap consists of two heaps:



Why this is correct: Recursive heap structure

A heap consists of two heaps:

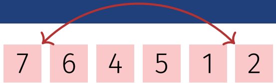


Algorithm SiftDown(A, i, m)

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

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

Sort heap



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

While $n > 1$

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

Sort heap

	7	6	4	5	1	2
swap $\Rightarrow$	2	6	4	5	1	7

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

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

Sort heap

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

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

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

Sort heap

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

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

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

Sort heap

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

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

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

Heap creation

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

Consequence:

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 ← n/2 downto 1 do
    SiftDown(A, i, n);
// Now A is a heap.
for i ← 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:

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

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

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

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

10. 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.

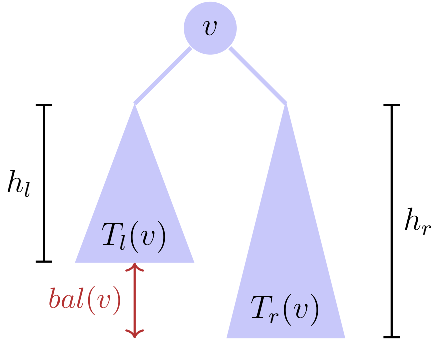
Balancing: guarantee that a tree with n nodes always has a height of $\mathcal{O}(\log n)$.

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

Balance of a node

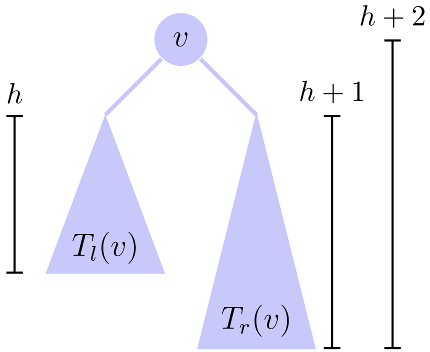
The height **balance** of a node v is defined as the height difference of its sub-trees $T_l(v)$ and $T_r(v)$

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

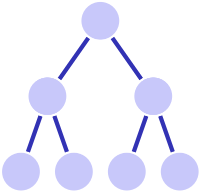


AVL Condition

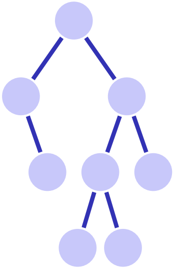
AVL Condition: for eacn node v of a tree
$\mathrm{bal}(v) \in \{-1, 0, 1\}$



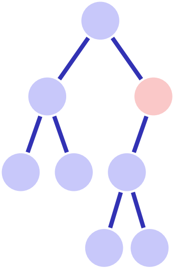
(Counter-)Examples



AVL tree with height 2



AVL tree with height 3

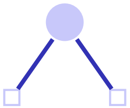


No AVL tree

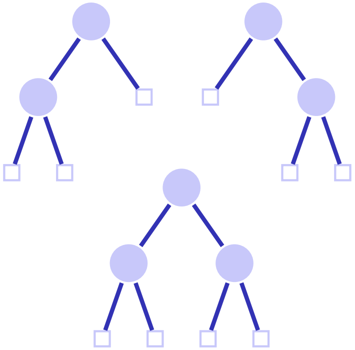
Number of Leaves

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

Lower bound of the leaves



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



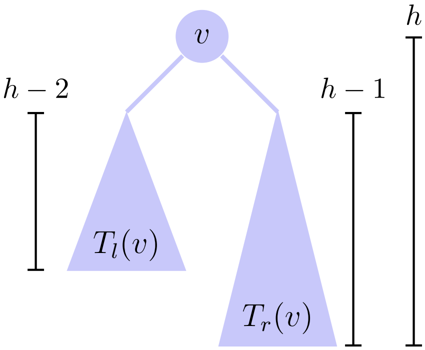
AVL tree with height 2 has at least $N(2) := 3$ leaves.

Lower bound of the leaves for $h > 2$

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

Minimal number of leaves $N(h)$ is

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



Overal we have $N(h) = F_{h+2}$ with **Fibonacci-numbers** $F_0 := 0$, $F_1 := 1$, $F_n := F_{n-1} + F_{n-2}$ for $n > 1$.

Fibonacci Numbers, closed Form

It holds that

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

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

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

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

$$F_i \overset{def}{=} F_{i-1} + F_{i-2} \overset{[*]}{=} \frac{1}{\sqrt{5}}(\phi^{i-1} - \hat{\phi}^{i-1}) + \frac{1}{\sqrt{5}}(\phi^{i-2} - \hat{\phi}^{i-2})$$

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

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

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

Tree Height

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

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

and thus

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

An AVL tree is asymptotically not more than 44% higher than a perfectly balanced tree.[5]

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

Insertion

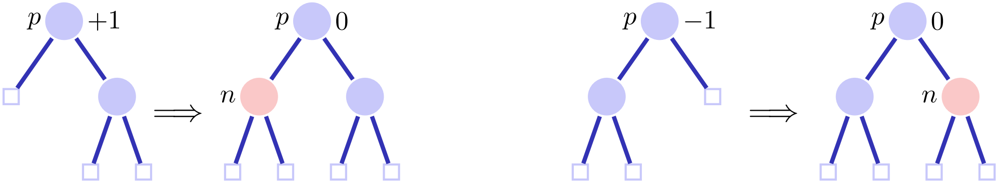
Balance

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

New node n is inserted:

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

Balance at Insertion Point

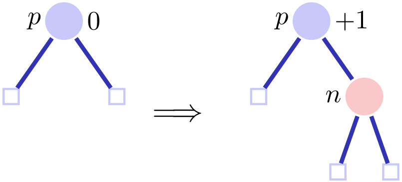


case 1: $\mathrm{bal}(p) = +1$

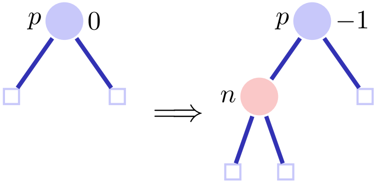
case 2: $\mathrm{bal}(p) = -1$

Finished in both cases because the subtree height did not change

Balance at Insertion Point



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



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

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

upin(p) - invariant

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

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

upin(p)

Assumption: p is left son of pp[6]

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

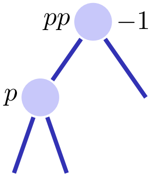
case 2: $\text{bal}(pp) = 0$, **`upin(pp)`**

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

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

upin(p)

Assumption: p is left son of pp



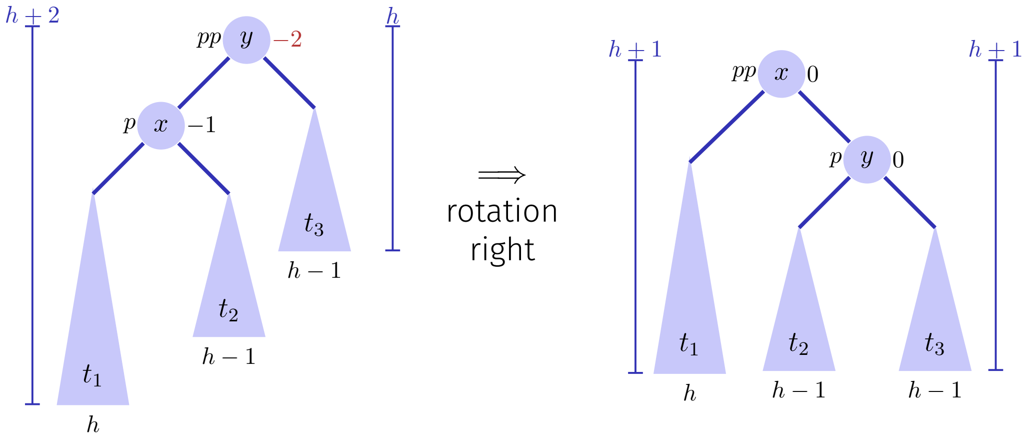
case 3: $\text{bal}(pp) = -1$,

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

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

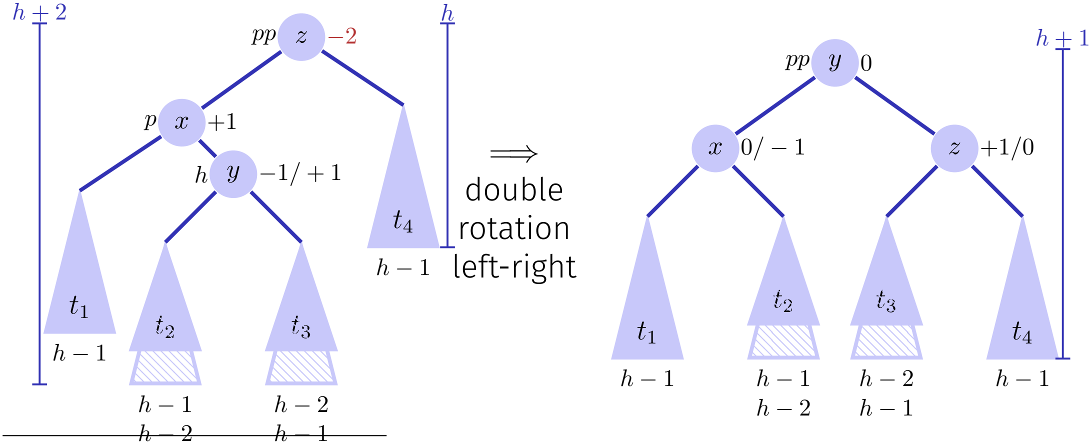
Rotations

case 1.1 $\mathrm{bal}(p) = -1$.[7]



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

case 1.1 $\text{bal}(p) = -1$. [8]



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

Analysis

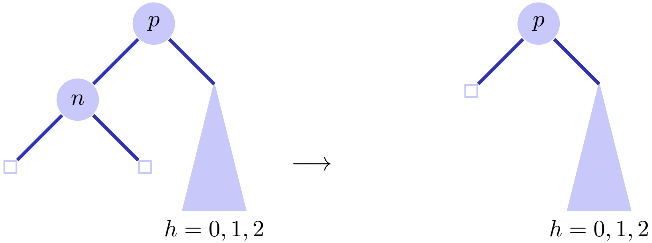
- Tree height: $\mathcal{O}(\log n)$.
- Insertion like in binary search tree.
- Balancing via recursion from node to the root. Maximal path lenght $\mathcal{O}(\log n)$.

Insertion in an AVL-tree provides run time costs of $\mathcal{O}(\log n)$.

Deletion

Case 1: Children of node n are both leaves Let p be parent node of n. $\Rightarrow$ Other subtree has height $h' = 0$, 1 or 2.

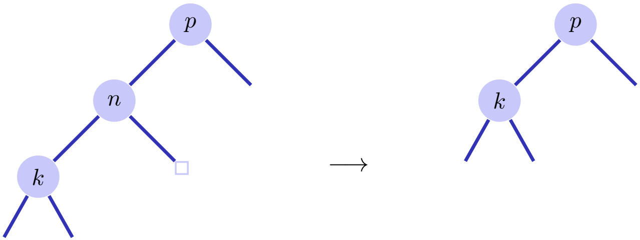
- $h' = 1$: Adapt $\text{bal}(p)$.
- $h' = 0$: Adapt $\text{bal}(p)$. Call **`upout(p)`**.
- $h' = 2$: Rebalanciere des Teilbaumes. Call **`upout(p)`**.



Deletion

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

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



Deletion

Case 3: both children of node n are inner nodes

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

upout(p)

Let pp be the parent node of p.

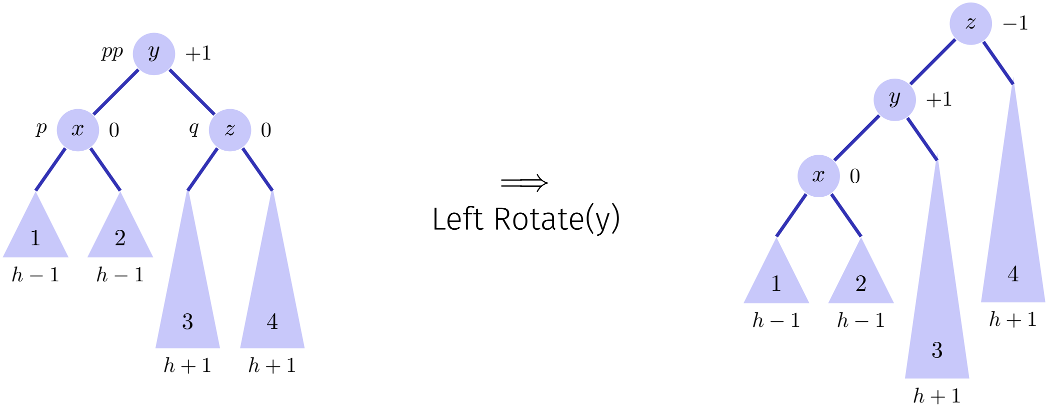
(a) p left child of pp

1. $\mathrm{bal}(pp) = -1 \;\Rightarrow\; \mathrm{bal}(pp) \leftarrow 0$. **`upout(pp)`**
2. $\mathrm{bal}(pp) = 0 \;\Rightarrow\; \mathrm{bal}(pp) \leftarrow +1$.
3. $\mathrm{bal}(pp) = +1 \Rightarrow$ next slides.

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

upout(p)

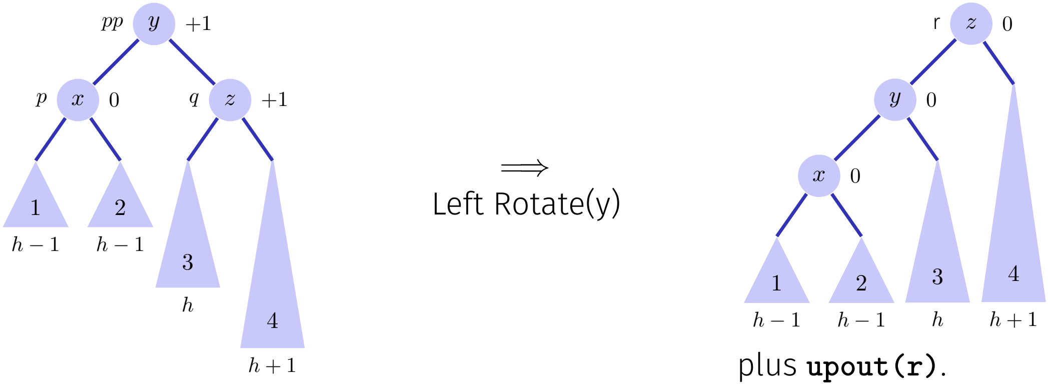
Case (a).3: $\text{bal}(pp) = +1$. Let q be brother of p
(a).3.1: $\text{bal}(q) = 0$.[9]



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

upout(p)

Case (a).3: $\mathrm{bal}(pp) = +1$. (a).3.2: $\mathrm{bal}(q) = +1$.[10]

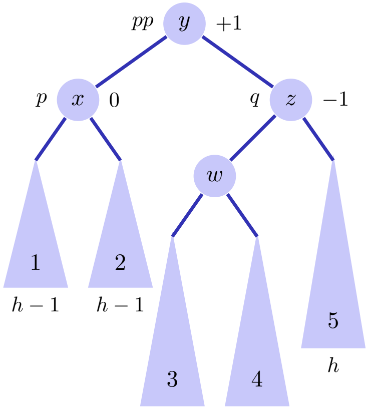


plus **upout(r)**.

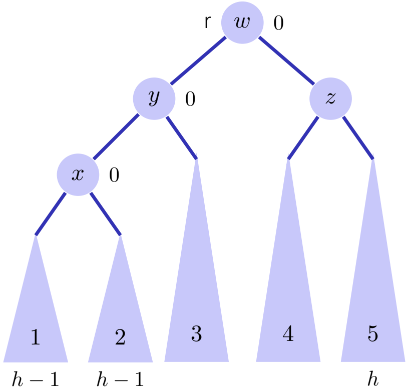
[10](b).3.2: $\mathrm{bal}(pp) = -1$, $\mathrm{bal}(q) = +1$, Right rotation+upout

upout(p)

Case (a).3: $\text{bal}(pp) = +1$. (a).3.3: $\text{bal}(q) = -1$.[11]



$\Longrightarrow$
Rotate right (z)
left (y)



plus **upout(r)**.

[11] **(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.

11. 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

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

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

Dictionary

Abstract Data Type (ADT) D to manage items[12] i with keys $k \in \mathcal{K}$ with operations

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

[12] Key-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. 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

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

- Theoretically always possible because each key is stored as a bit-sequence in the computer
- Theoretically also: $x = y \Leftrightarrow ph(x) = ph(y)$
- Practically: APIs offer functions for pre-hashing. (Java: **`object.hashCode()`**, C++: **`std::hash<>`**, Python: **`hash(object)`**)
- APIs map the key from the key set to an integer with a restricted size.[13]

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

Prehashing Example : String

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

$$ph(s) = \left(\sum_{i=0}^{l_s - 1} s_{l_s - i} \cdot b^i \right) \bmod 2^w$$

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

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

Example (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

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

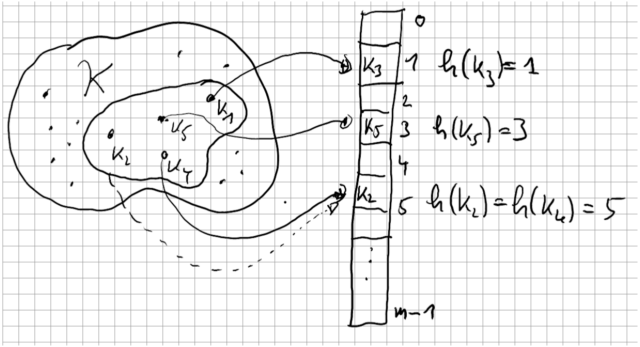
With $b = 31$ and $m = 2^{32}$ we get in Java[14]

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

[14]Try to understand why this works

Lösung zum zweiten Problem: Hashing

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



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

Nomenclature

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

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

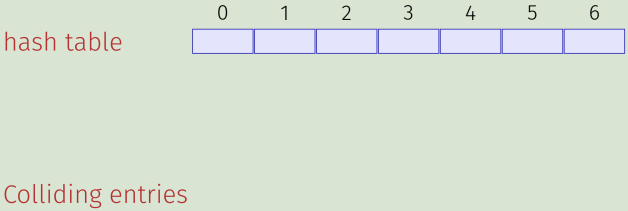
Normally $|\mathcal{K}| \gg m$. There are $k_1, k_2 \in \mathcal{K}$ with $h(k_1) = h(k_2)$ (**collision**).
A hash function should map the set of keys as uniformly as possible to the hash table.

Resolving Collisions: Chaining

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

Keys 12

Direct Chaining of the Colliding entries



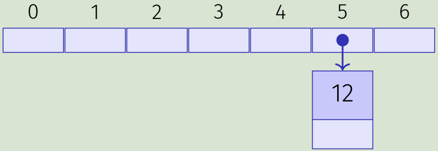
Resolving Collisions: Chaining

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

Keys 12 , 55

Direct Chaining of the Colliding entries

hash table



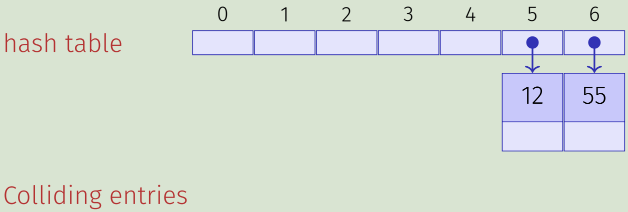
Colliding entries

Resolving Collisions: Chaining

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

Keys 12 , 55 , 5

Direct Chaining of the Colliding entries

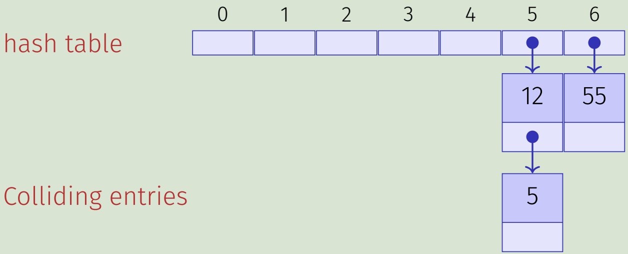


Resolving Collisions: Chaining

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

Keys 12 , 55 , 5 , 15

Direct Chaining of the Colliding entries

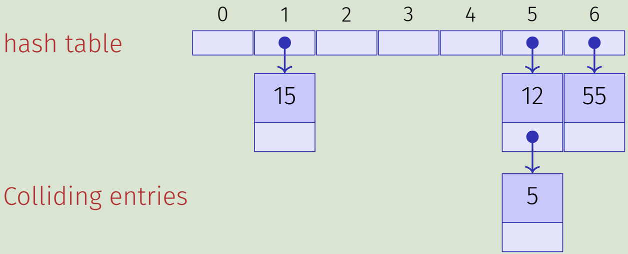


Resolving Collisions: Chaining

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

Keys 12 , 55 , 5 , 15 , 2

Direct Chaining of the Colliding entries

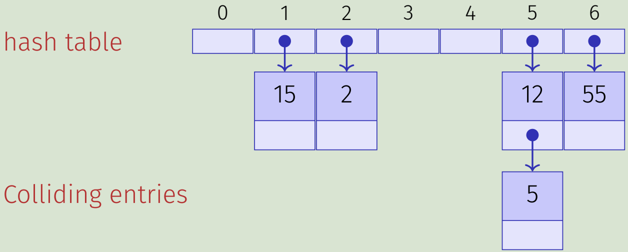


Resolving Collisions: Chaining

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

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

Direct Chaining of the Colliding entries

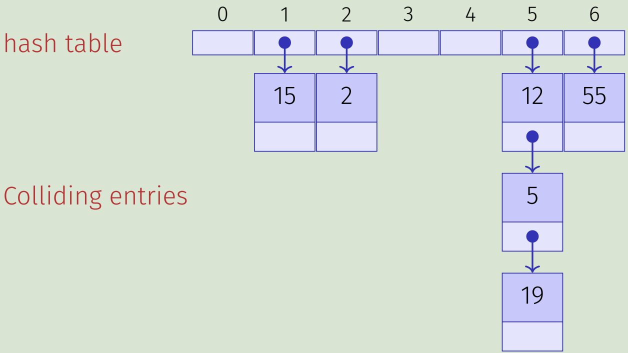


Resolving Collisions: Chaining

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

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

Direct Chaining of the Colliding entries

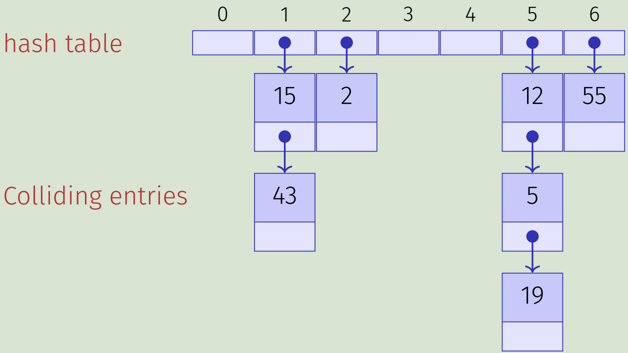


Resolving Collisions: Chaining

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

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

Direct Chaining of the Colliding entries



Algorithm for Hashing with Chaining

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

Worst-case Analysis

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

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

Simple Uniform Hashing

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

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

Simple Uniform Hashing

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

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

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

Simple Uniform Hashing

Theorem 5

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

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

($\Rightarrow$ Amortized Analysis)

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: $\Theta(1 + 2 + 4 + 8 + \cdots + 2^k) = \Theta(2^{k+1} - 1) = \Theta(n)$.

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

Open Addressing

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

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

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

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

Algorithms for open addressing

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

Linear Probing

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

Linear Probing

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

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

Key 12

0	1	2	3	4	5	6

Linear Probing

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

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

Key 12 , 55

0	1	2	3	4	5	6
					12	

Linear Probing

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

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

Key 12 , 55 , 5

0	1	2	3	4	5	6
					12	55

Linear Probing

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

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

Key 12 , 55 , 5 , 15

0	1	2	3	4	5	6
5					12	55

Linear Probing

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

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

Key 12 , 55 , 5 , 15 , 2

0	1	2	3	4	5	6
5	15				12	55

Linear Probing

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

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

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

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

Linear Probing

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

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

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

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

Discussion

Discussion

Example $\alpha = 0.95$

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

Discussion

Example $\alpha = 0.95$

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

Disadvantage of the method?

Discussion

Example $\alpha = 0.95$

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

Disadvantage of the method?

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

Quadratic Probing

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

Quadratic Probing

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

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

Keys 12

0	1	2	3	4	5	6

Quadratic Probing

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

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

Keys 12 , 55

0	1	2	3	4	5	6
					12	

Quadratic Probing

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

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

Keys 12 , 55 , 5

0	1	2	3	4	5	6
					12	55

Quadratic Probing

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

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

Keys 12 , 55 , 5 , 15

0	1	2	3	4	5	6
				5	12	55

Quadratic Probing

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

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

Keys 12 , 55 , 5 , 15 , 2

0	1	2	3	4	5	6
	15			5	12	55

Quadratic Probing

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

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

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

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

Quadratic Probing

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

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

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

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

Discussion

Example $\alpha = 0.95$

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

Discussion

Example $\alpha = 0.95$

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

Problems of this method?

Discussion

Example $\alpha = 0.95$

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

Problems of this method?

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

Double Hashing

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

Double Hashing

Two hash functions $h(k)$ and $h'(k)$. $s(k, j) = h(k) + j \cdot h'(k)$.
$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \ldots, h(k) + (m-1)h'(k)) \mod m$

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

Keys 12

0	1	2	3	4	5	6

Double Hashing

Two hash functions $h(k)$ and $h'(k)$. $s(k,j) = h(k) + j \cdot h'(k)$.
$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \ldots, h(k) + (m-1)h'(k)) \mod m$

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

Keys 12 , 55

0	1	2	3	4	5	6
					12	

Double Hashing

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

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

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

Keys 12 , 55 , 5

0	1	2	3	4	5	6
					12	55

Double Hashing

Two hash functions $h(k)$ and $h'(k)$. $s(k, j) = h(k) + j \cdot h'(k)$.
$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \ldots, h(k) + (m-1)h'(k)) \mod m$

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

Keys 12 , 55 , 5 , 15

0	1	2	3	4	5	6
5					12	55

Double Hashing

Two hash functions $h(k)$ and $h'(k)$. $s(k,j) = h(k) + j \cdot h'(k)$.
$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \ldots, h(k) + (m-1)h'(k)) \mod m$

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

Keys 12 , 55 , 5 , 15 , 2

0	1	2	3	4	5	6
5	15				12	55

Double Hashing

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

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

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

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

Double Hashing

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

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

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

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

Double Hashing

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

Independence 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 equaly likely to be any of the $m!$ permutations of $\{0, 1, \ldots, m-1\}$

(Double hashing is reasonably close)

Analysis of Uniform Hashing with Open Addressing

Theorem 6

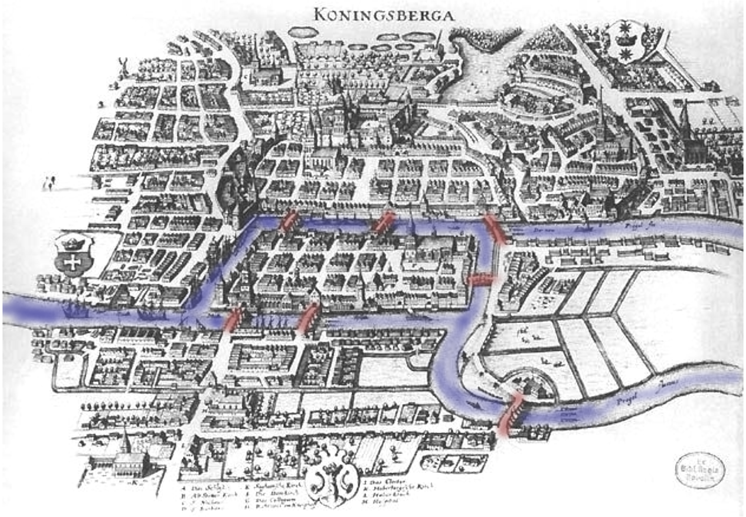
Let an open-addressing hash table be filled with load-factor $\alpha = \frac{n}{m} < 1$. *Under the assumption of uniform hashing, the next operation has expected costs of* $\leq \frac{1}{1-\alpha}$.

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

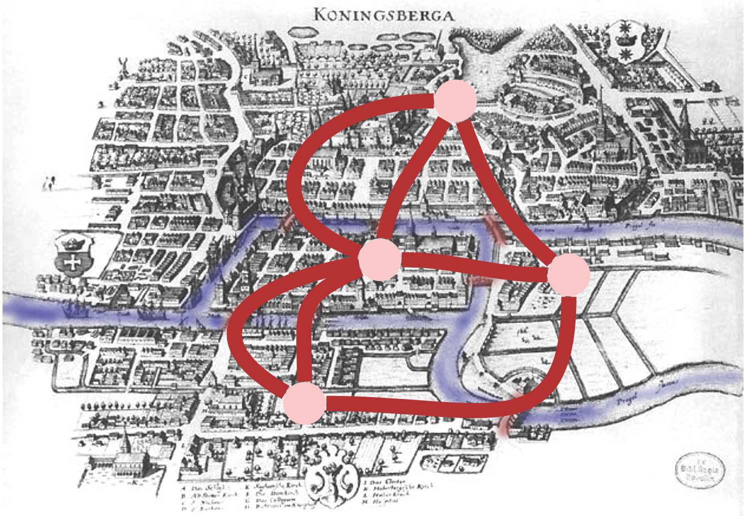
12. Graphs

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

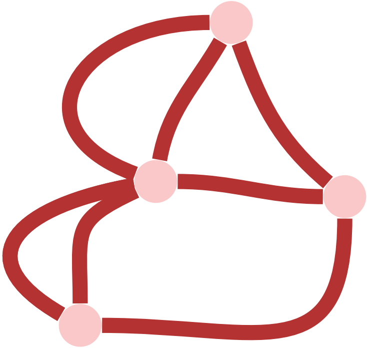
Königsberg 1736



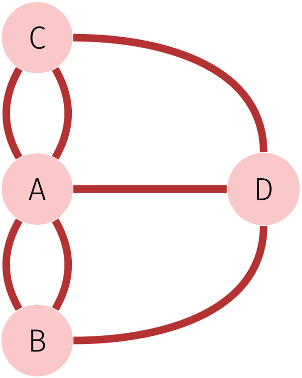
Königsberg 1736



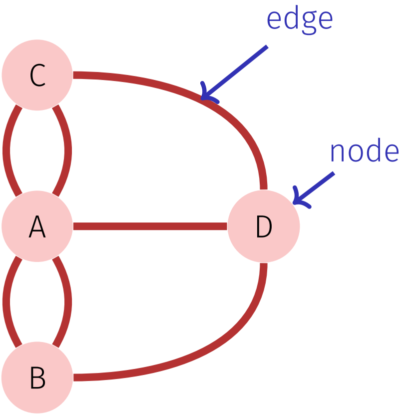
Königsberg 1736



[Multi]Graph

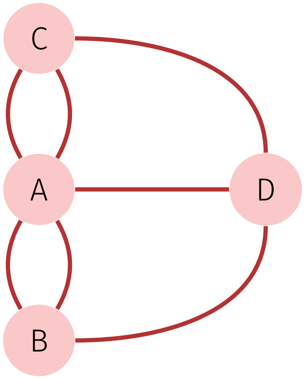


[Multi]Graph



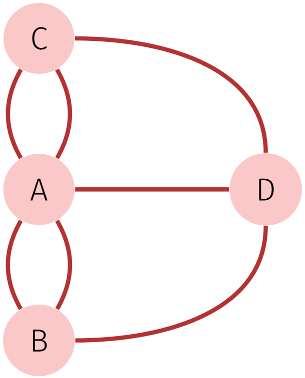
Cycles

- Is there a cycle through the town (the graph) that uses each bridge (each edge) exactly once?



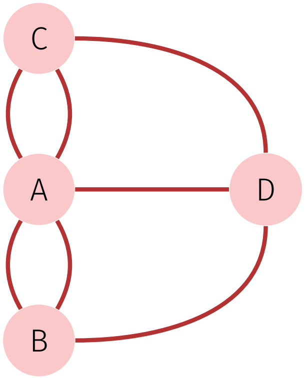
Cycles

- Is there a cycle through the town (the graph) that uses each bridge (each edge) exactly once?
- Euler (1736): no.



Cycles

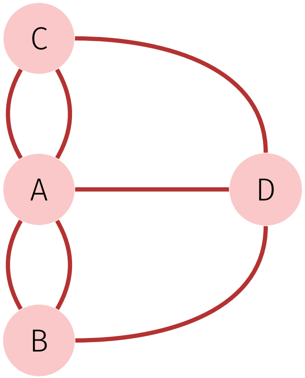
- Is there a cycle through the town (the graph) that uses each bridge (each edge) exactly once?
- Euler (1736): no.
- Such a *cycle* is called *Eulerian path*.



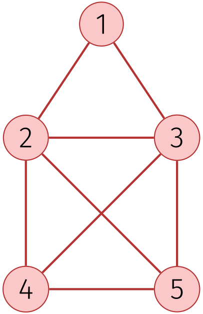
Cycles

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

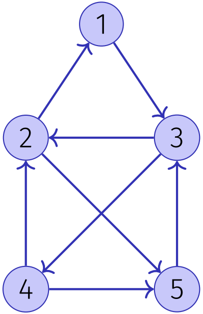
  '$\Rightarrow$" is straightforward, "$\Leftarrow$" ist a bit more difficult but still elementary.



Notation



undirected



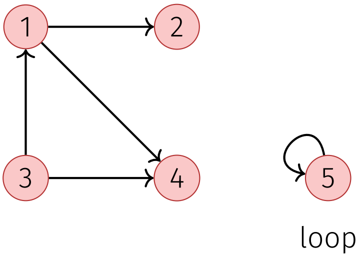
directed

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

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

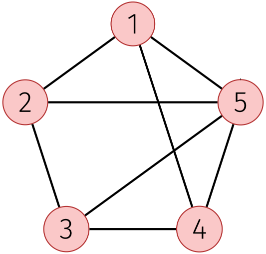
Notation

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



Notation

An **undirected graph** consists of a set $V = \{v_1, \ldots, v_n\}$ of nodes a and a set $E \subseteq \{\{u,v\} | u,v \in V\}$ of edges. Edges may bot be contained more than once.[15]

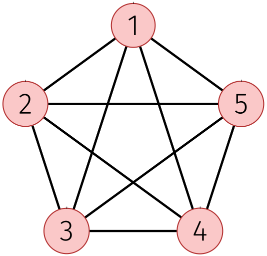


undirected graph

[15]As opposed to the introductory example – it is then called multi-graph.

Notation

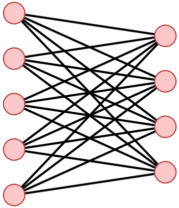
An undirected graph $G = (V, E)$ without loops where E comprises all edges between pairwise different nodes is called **complete**.



a complete undirected graph

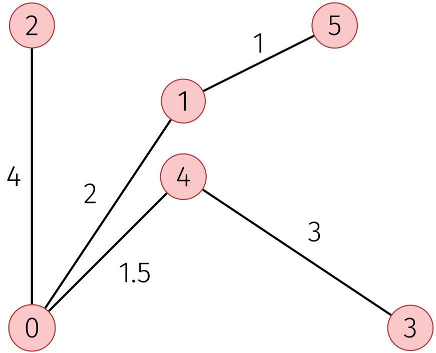
Notation

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



Notation

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



Notation

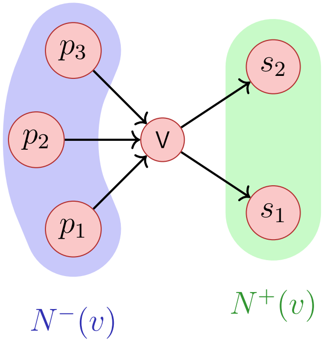
For directed graphs $G = (V, E)$

- $w \in V$ is called adjacent to $v \in V$, if $(v, w) \in E$

Notation

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

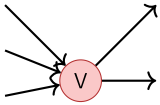
- $w \in V$ is called adjacent to $v \in V$, if $(v, w) \in E$
- **Predecessors** of $v \in V$: $N^-(v) := \{u \in V | (u, v) \in E\}$.
  Successors: $N^+(v) := \{u \in V | (v, u) \in E\}$



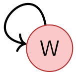
Notation

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

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



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



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

Notation

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

- $w \in V$ is called **adjacent** to $v \in V$, if $\{v, w\} \in E$

Notation

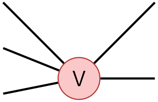
For undirected graphs $G = (V, E)$:

- $w \in V$ is called **adjacent** to $v \in V$, if $\{v, w\} \in E$
- **Neighbourhood** of $v \in V$: $N(v) = \{w \in V | \{v, w\} \in E\}$

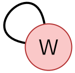
Notation

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 $\langle v_1, \ldots, v_{k+1} \rangle$ such that for each $i \in \{1 \ldots k\}$ there is an edge from v_i to v_{i+1} .

Paths

- **Path**: a sequence of nodes $\langle v_1, \ldots, v_{k+1} \rangle$ such that for each $i \in \{1 \ldots k\}$ there is an edge from v_i to v_{i+1} .
- **Length** of a path: number of contained edges k.

Paths

- **Path**: a sequence of nodes $\langle v_1, \ldots, v_{k+1} \rangle$ such that for each $i \in \{1 \ldots k\}$ 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. $\sum_{i=1}^{k} c(\{v_i, v_{i+1}\})$)

Paths

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

Connectedness

- An undirected graph is called **connected**, if for eacheach pair $v, w \in V$ there is a connecting path.
- A directed graph is called **strongly connected**, if for each pair $v, w \in V$ there is a connecting path.
- A directed graph is called **weakly connected**, if the corresponding undirected graph is connected.

Simple Observations

- generally: $0 \leq |E| \in \mathcal{O}(|V|^2)$
- connected graph: $|E| \in \Omega(|V|)$
- complete graph: $|E| = \frac{|V| \cdot (|V|-1)}{2}$ (undirected)
- Maximally $|E| = |V|^2$ (directed),$|E| = \frac{|V| \cdot (|V|+1)}{2}$ (undirected)

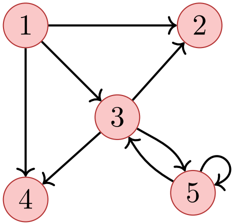
Cycles

- **Cycle**: path $\langle v_1, \ldots, v_{k+1} \rangle$ with $v_1 = v_{k+1}$
- **Simple cycle**: Cycle with pairwise different $v_1, \ldots, v_k$, that does not use an edge more than once.
- **Acyclic**: graph without any cycles.

Conclusion: undirected graphs cannot contain cycles with length 2 (loops have length 1)

Representation using a Matrix

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

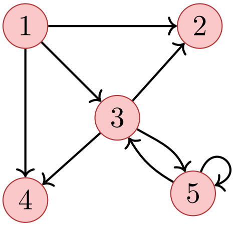


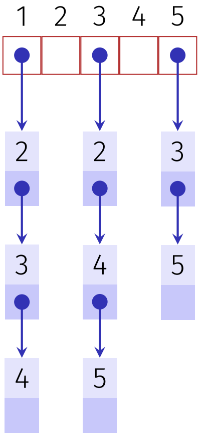
$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

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

Representation with a List

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





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

Runtimes of simple Operations

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

Runtimes of simple Operations

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

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		
$(u, v) \in E$?		
Insert edge		
Delete edge		

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	$\Theta(n^2)$	
$(u, v) \in E$?		
Insert edge		
Delete edge		

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	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?		
Insert edge		
Delete edge		

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	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?	$\Theta(1)$	
Insert edge		
Delete edge		

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		
Delete edge		

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	$\Theta(1)$	
Delete edge		

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		

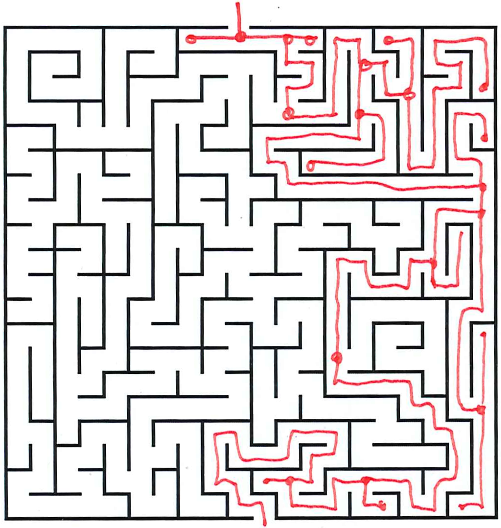
Runtimes of simple Operations

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	$\Theta(1)$	

Runtimes of simple Operations

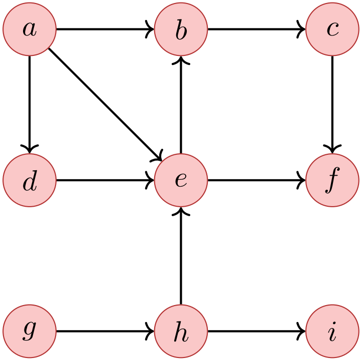
Operation	Matrix	List
Find neighbours/successors of $v \in V$	$\Theta(n)$	$\Theta(\deg^+ v)$
find $v \in V$ without neighbour/successor	$\Theta(n^2)$	$\Theta(n)$
$(u, v) \in E$?	$\Theta(1)$	$\Theta(\deg^+ v)$
Insert edge	$\Theta(1)$	$\Theta(1)$
Delete edge	$\Theta(1)$	$\Theta(\deg^+ v)$

Depth First Search



Graph Traversal: Depth First Search

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

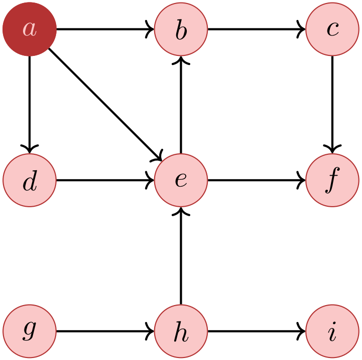


Adjazenzliste

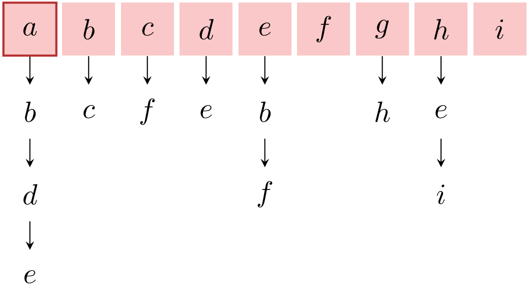
- $a \to b \to d \to e$
- $b \to c$
- $c \to f$
- $d \to e$
- $e \to b \to f$
- f
- $g \to h$
- $h \to e \to i$
- i

Graph Traversal: Depth First Search

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

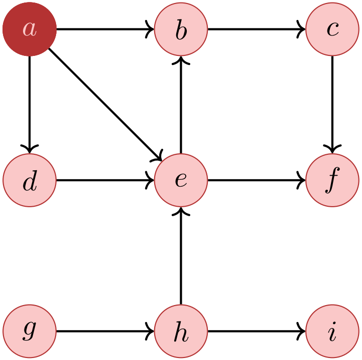


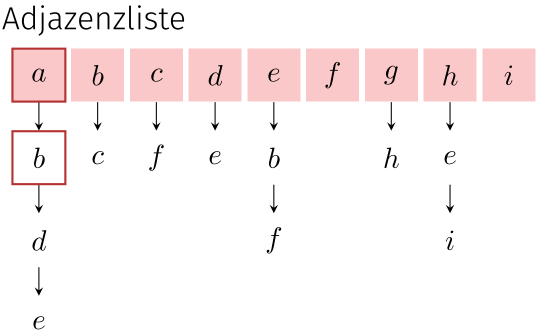
Adjazenzliste



Graph Traversal: Depth First Search

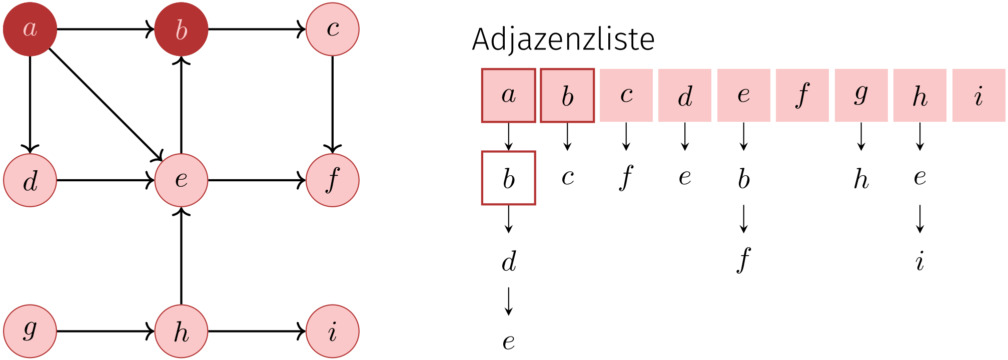
Follow the path into its depth until nothing is left to visit.





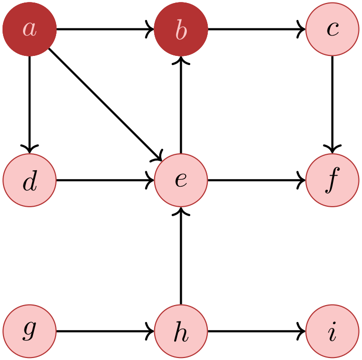
Graph Traversal: Depth First Search

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

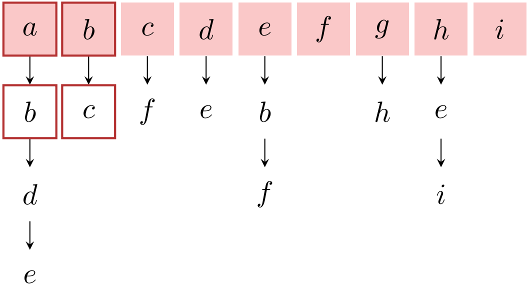


Graph Traversal: Depth First Search

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

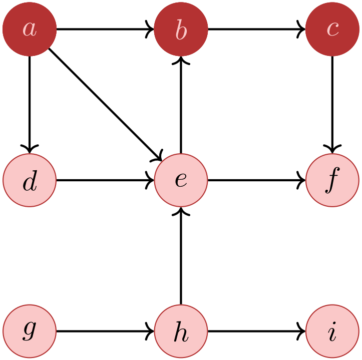


Adjazenzliste

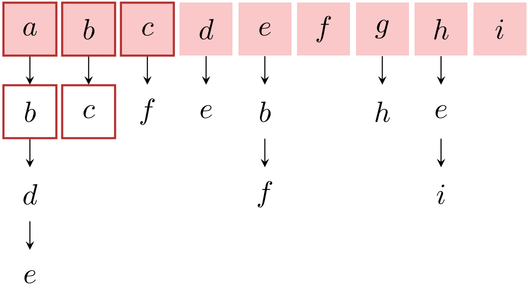


Graph Traversal: Depth First Search

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

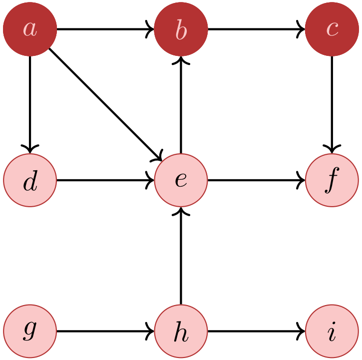


Adjazenzliste

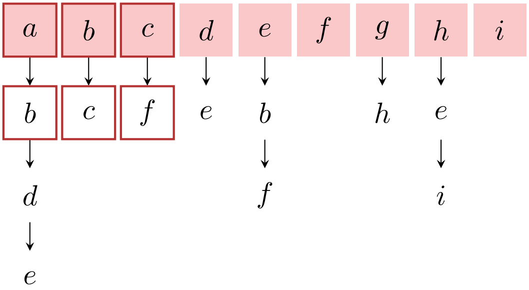


Graph Traversal: Depth First Search

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

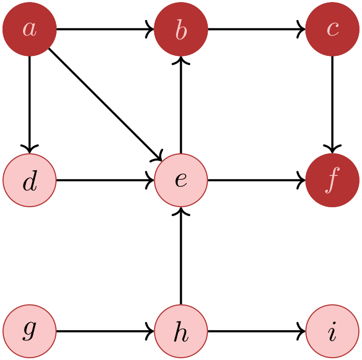


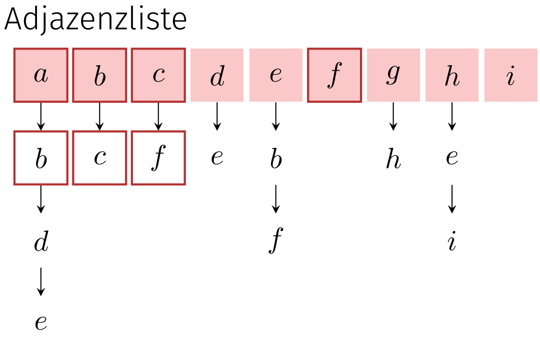
Adjazenzliste



Graph Traversal: Depth First Search

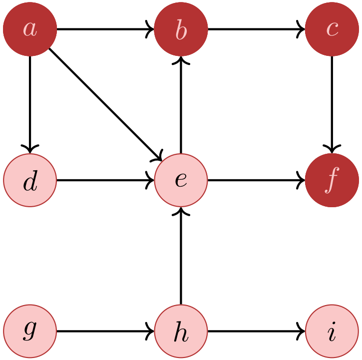
Follow the path into its depth until nothing is left to visit.

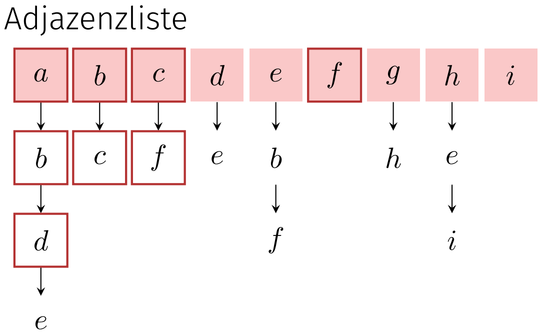




Graph Traversal: Depth First Search

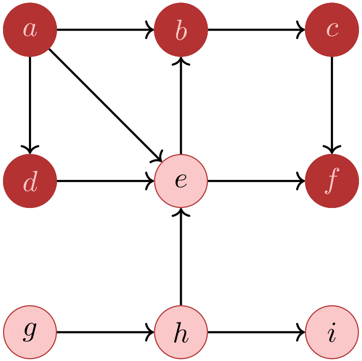
Follow the path into its depth until nothing is left to visit.



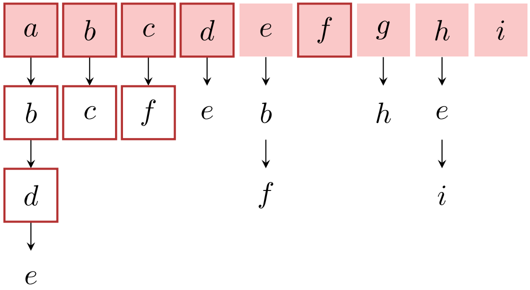


Graph Traversal: Depth First Search

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

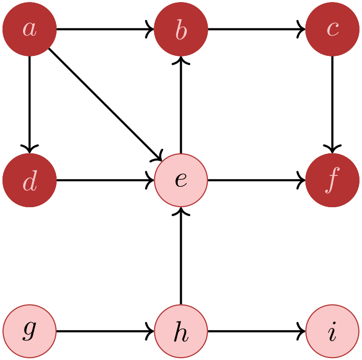


Adjazenzliste

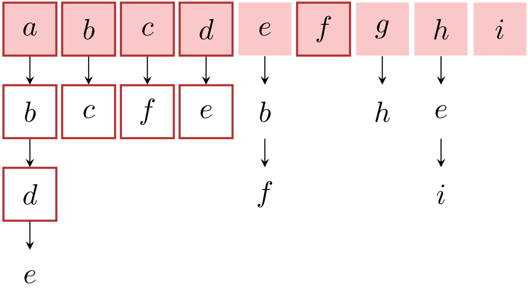


Graph Traversal: Depth First Search

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

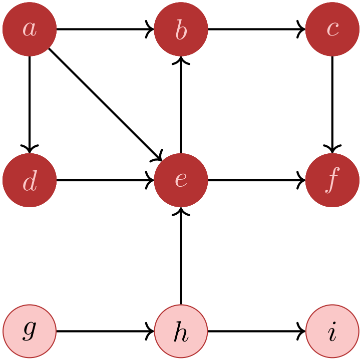


Adjazenzliste

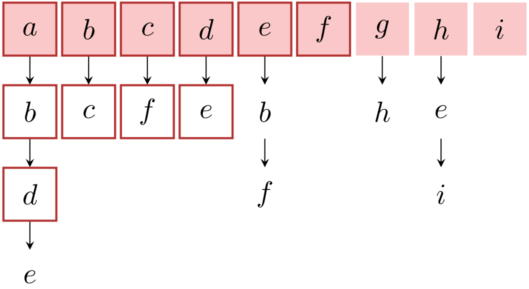


Graph Traversal: Depth First Search

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

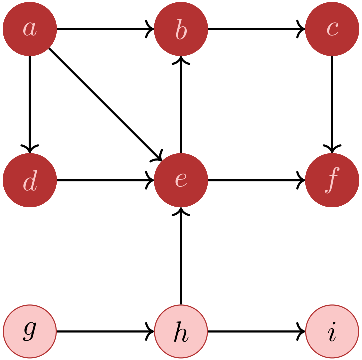


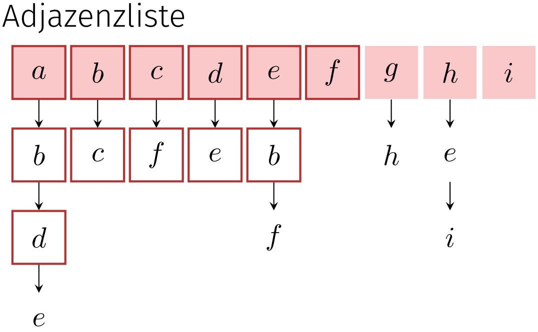
Adjazenzliste



Graph Traversal: Depth First Search

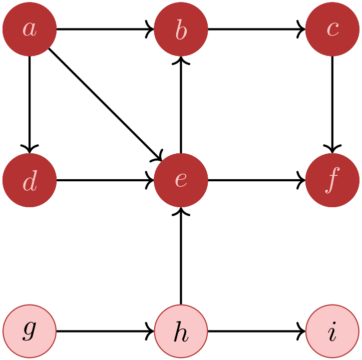
Follow the path into its depth until nothing is left to visit.

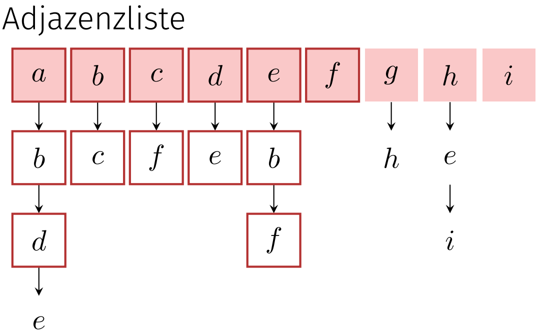




Graph Traversal: Depth First Search

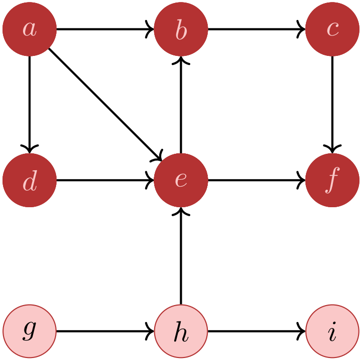
Follow the path into its depth until nothing is left to visit.

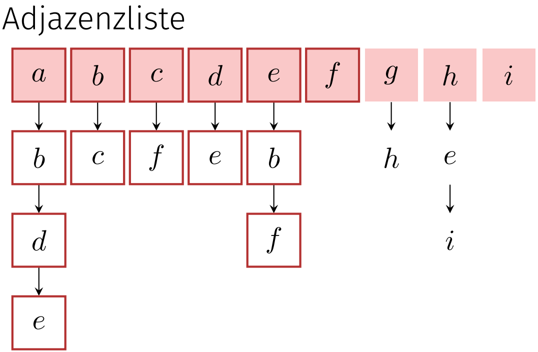




Graph Traversal: Depth First Search

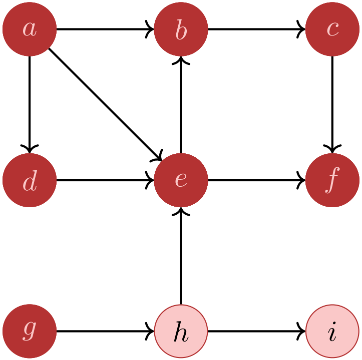
Follow the path into its depth until nothing is left to visit.

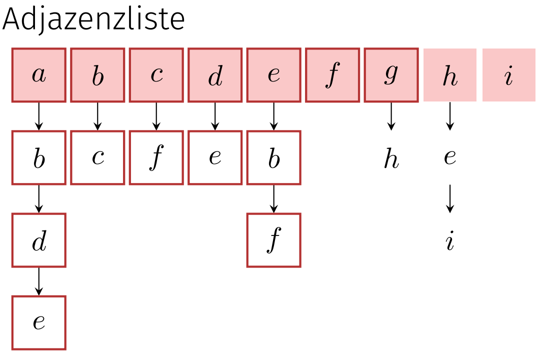




Graph Traversal: Depth First Search

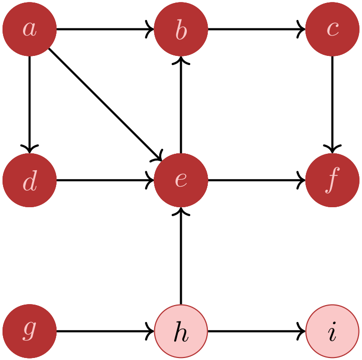
Follow the path into its depth until nothing is left to visit.



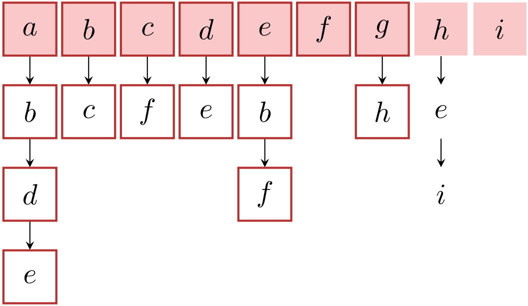


Graph Traversal: Depth First Search

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

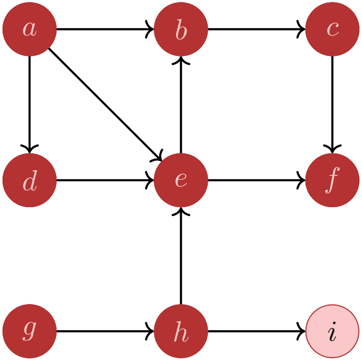


Adjazenzliste

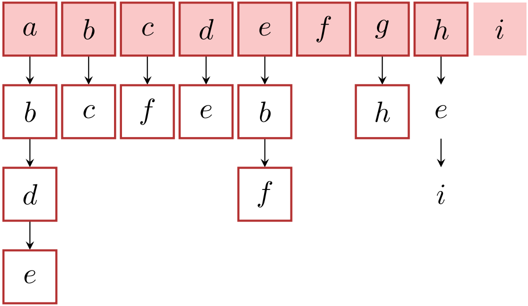


Graph Traversal: Depth First Search

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

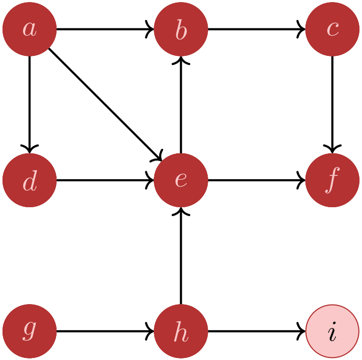


Adjazenzliste

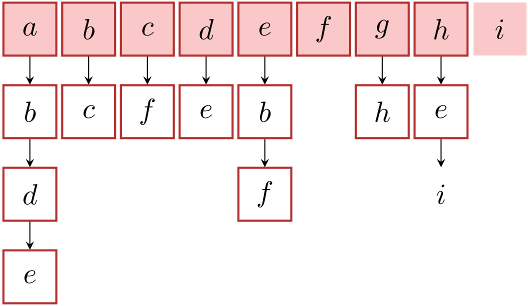


Graph Traversal: Depth First Search

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

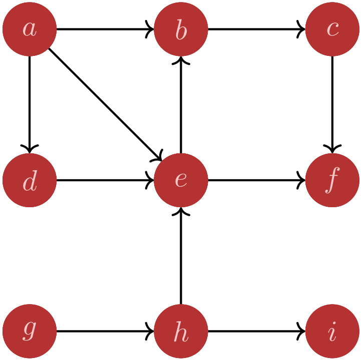


Adjazenzliste



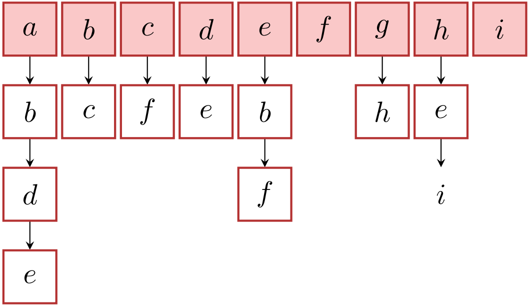
Graph Traversal: Depth First Search

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



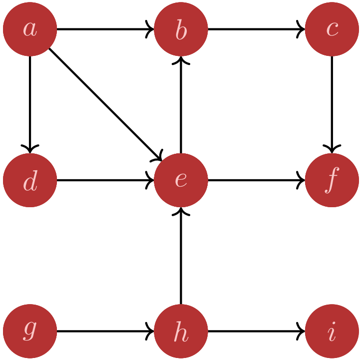
Order $a, b, c, f, d, e, g, h, i$

Adjazenzliste



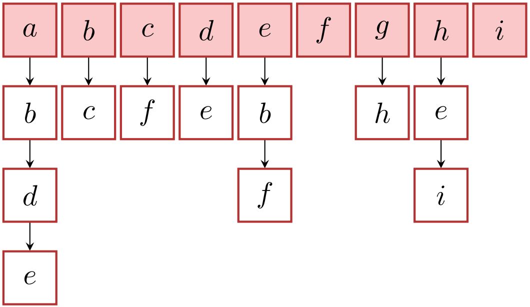
Graph Traversal: Depth First Search

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



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

Adjazenzliste



Colors

Conceptual coloring of nodes

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

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

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

$v.color \leftarrow$ grey
foreach $w \in N^+(v)$ **do**
  if $w.color =$ white **then**
    DFS-Visit(G, w)

$v.color \leftarrow$ black

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

Algorithm Depth First visit DFS-Visit(G)

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

foreach $v \in V$ **do**
    $v.color \leftarrow$ white

foreach $v \in V$ **do**
    if $v.color =$ white **then**
        DFS-Visit(G,v)

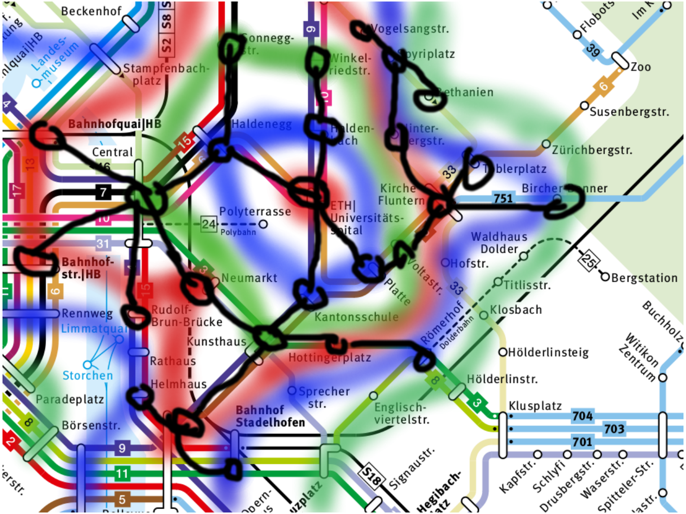
Depth First Search for all nodes of a graph. Running time:
$\Theta(|V| + \sum_{v \in V}(\deg^+(v) + 1)) = \Theta(|V| + |E|)$.

Interpretation of the Colors

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

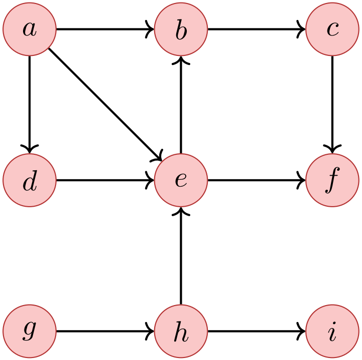
- White node: new tree edge
- Grey node: Zyklus (“back-egde”)
- Black node: forward- / cross edge

Breadth First Search

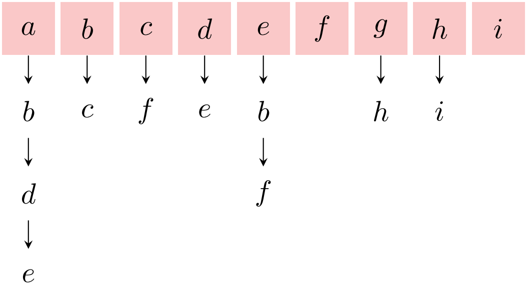


Graph Traversal: Breadth First Search

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

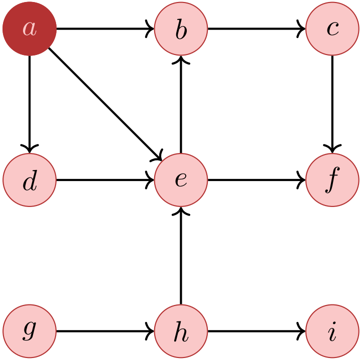


Adjazenzliste

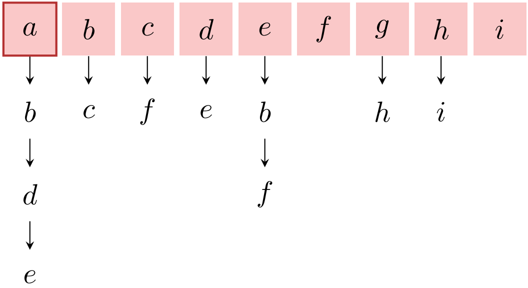


Graph Traversal: Breadth First Search

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

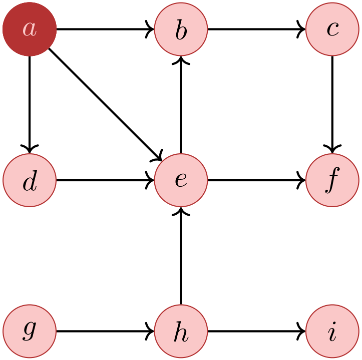


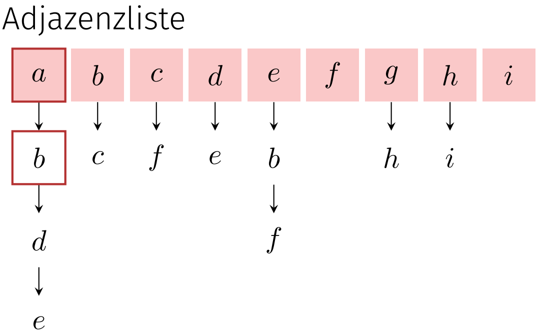
Adjazenzliste



Graph Traversal: Breadth First Search

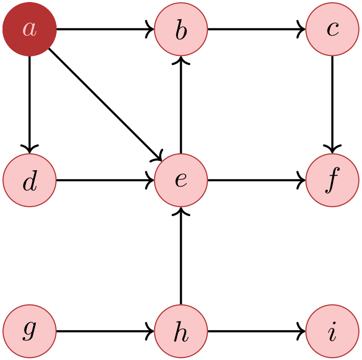
Follow the path in breadth and only then descend into depth.

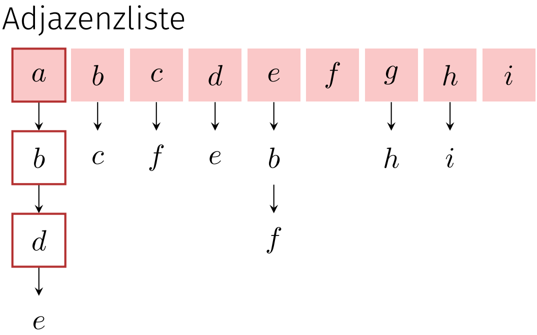




Graph Traversal: Breadth First Search

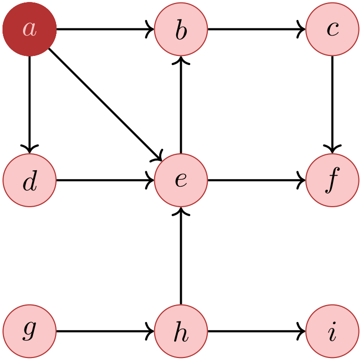
Follow the path in breadth and only then descend into depth.



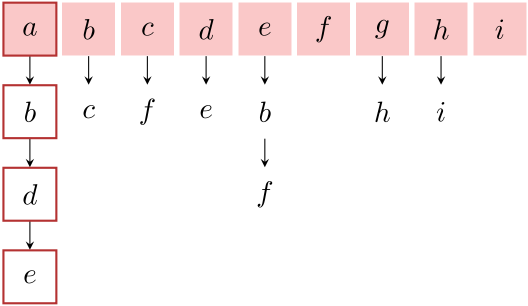


Graph Traversal: Breadth First Search

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

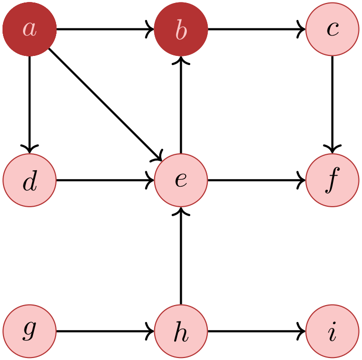


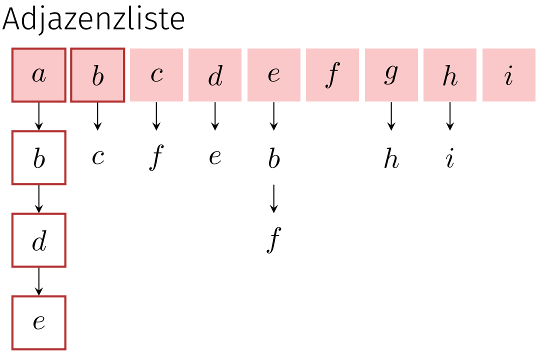
Adjazenzliste



Graph Traversal: Breadth First Search

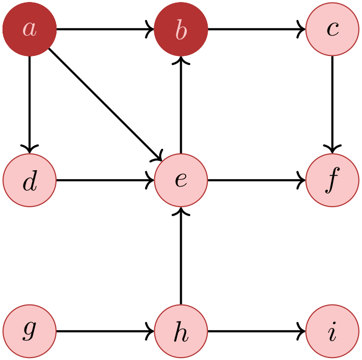
Follow the path in breadth and only then descend into depth.

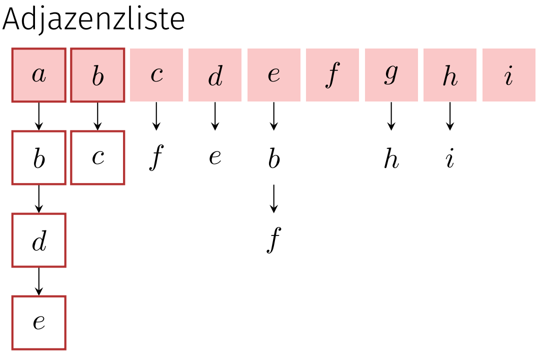




Graph Traversal: Breadth First Search

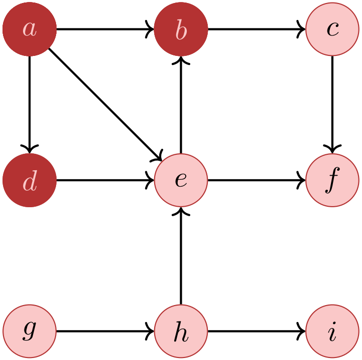
Follow the path in breadth and only then descend into depth.



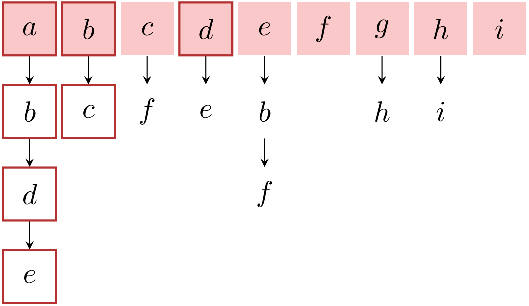


Graph Traversal: Breadth First Search

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

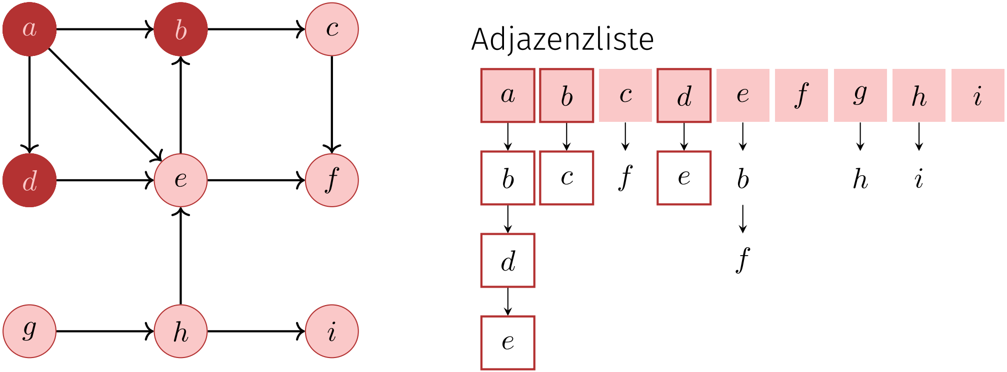


Adjazenzliste



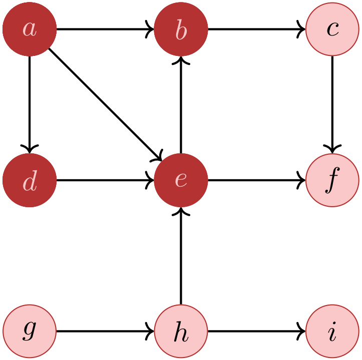
Graph Traversal: Breadth First Search

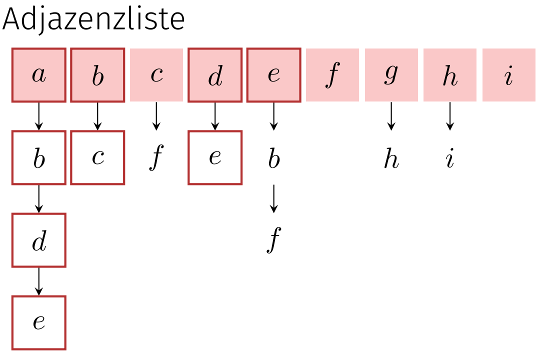
Follow the path in breadth and only then descend into depth.



Graph Traversal: Breadth First Search

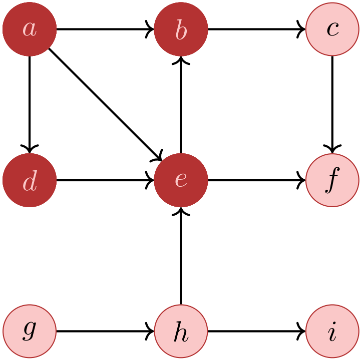
Follow the path in breadth and only then descend into depth.

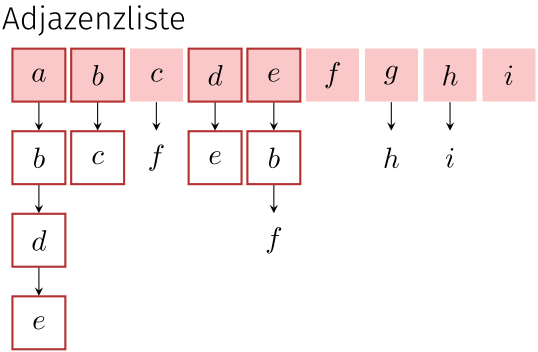




Graph Traversal: Breadth First Search

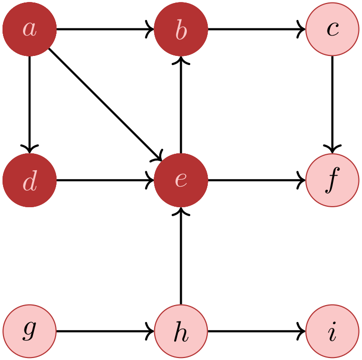
Follow the path in breadth and only then descend into depth.

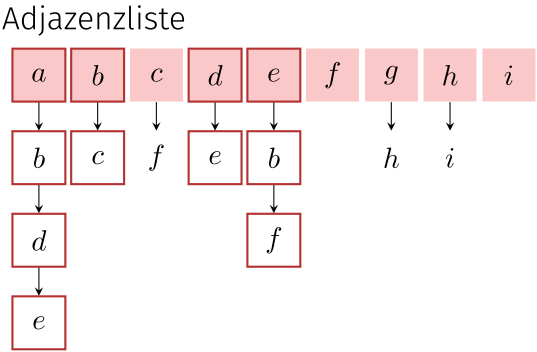




Graph Traversal: Breadth First Search

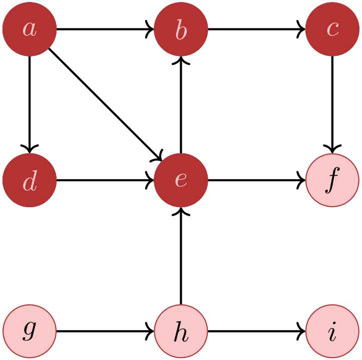
Follow the path in breadth and only then descend into depth.

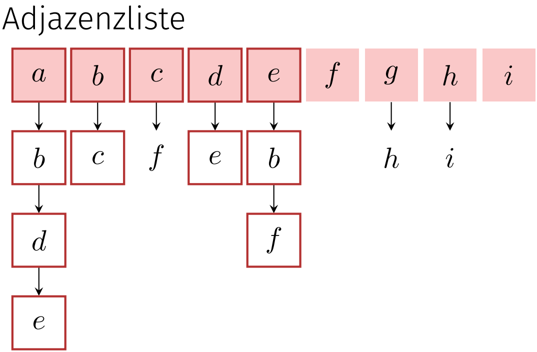




Graph Traversal: Breadth First Search

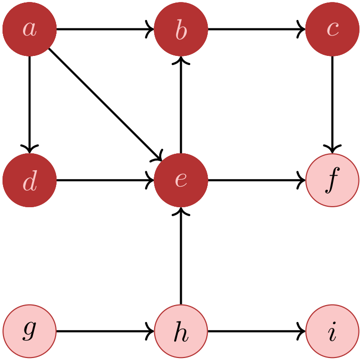
Follow the path in breadth and only then descend into depth.

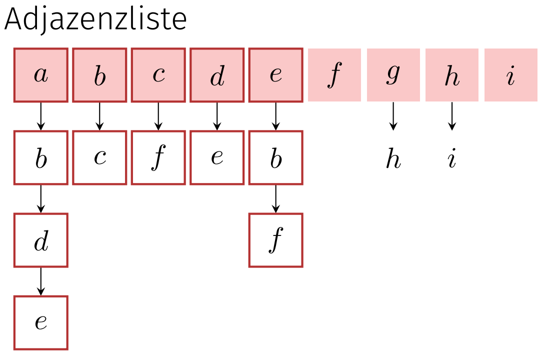




Graph Traversal: Breadth First Search

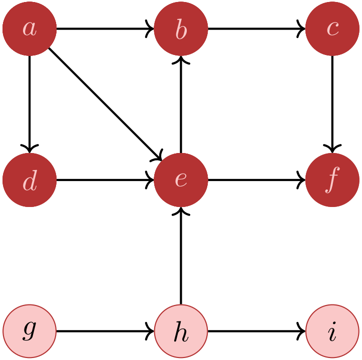
Follow the path in breadth and only then descend into depth.



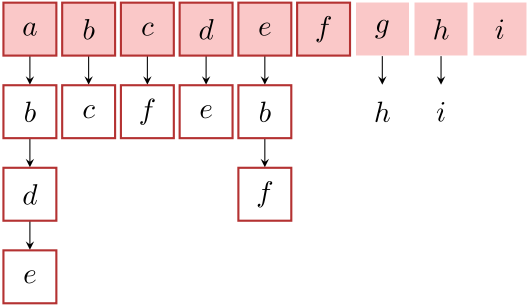


Graph Traversal: Breadth First Search

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

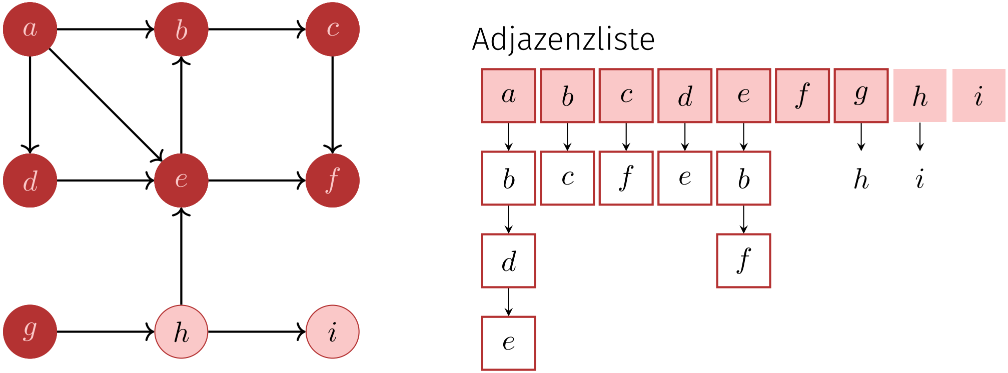


Adjazenzliste



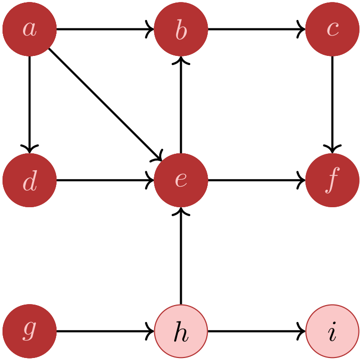
Graph Traversal: Breadth First Search

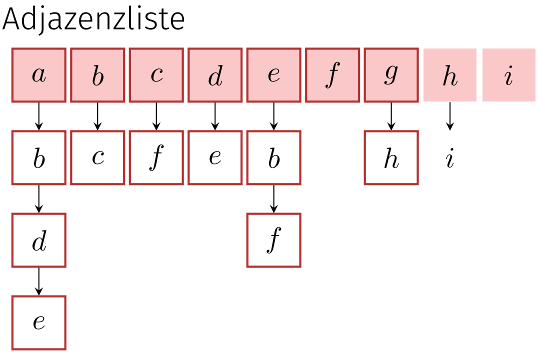
Follow the path in breadth and only then descend into depth.



Graph Traversal: Breadth First Search

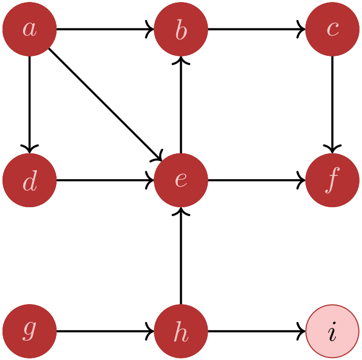
Follow the path in breadth and only then descend into depth.

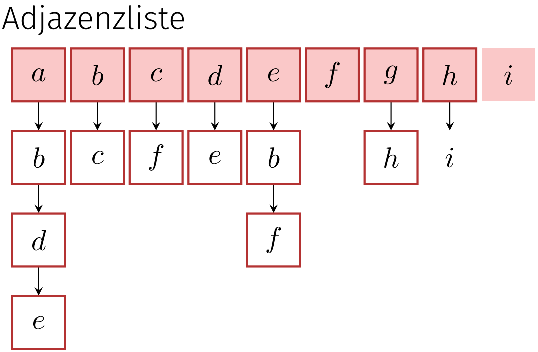




Graph Traversal: Breadth First Search

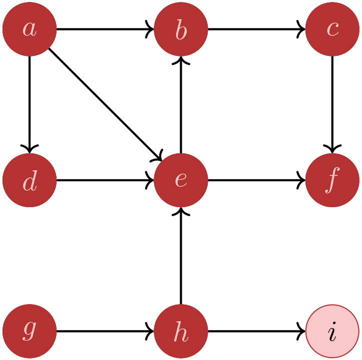
Follow the path in breadth and only then descend into depth.



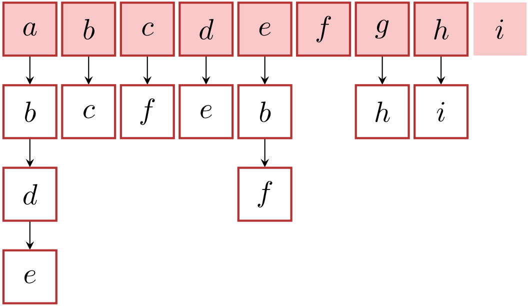


Graph Traversal: Breadth First Search

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

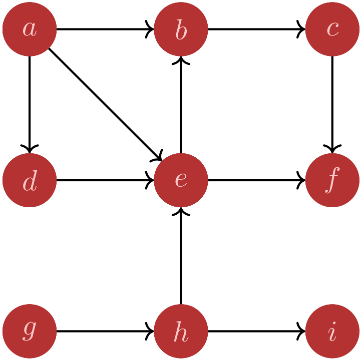


Adjazenzliste



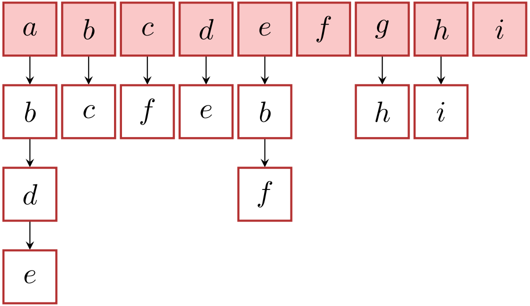
Graph Traversal: Breadth First Search

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



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

Adjazenzliste



(Iterative) BFS-Visit(G, v)

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

Queue $Q \leftarrow \emptyset$
$v.color \leftarrow$ grey
enqueue(Q, v)
while $Q \neq \emptyset$ **do**
    $w \leftarrow$ dequeue(Q)
    foreach $c \in N^+(w)$ **do**
        if $c.color =$ white **then**
            $c.color \leftarrow$ grey
            enqueue(Q, c)
    $w.color \leftarrow$ black

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

Main program BFS-Visit(G)

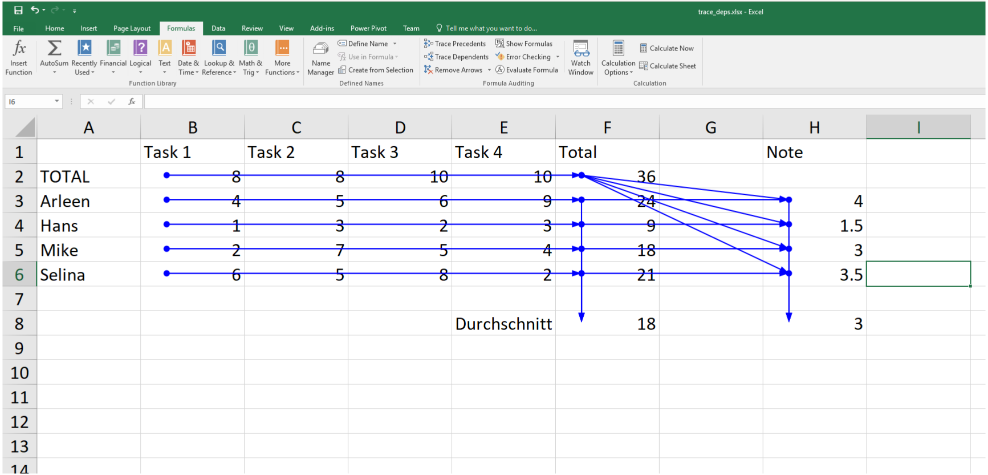
Input: graph $G = (V, E)$

foreach $v \in V$ **do**
    $v.color \leftarrow$ white

foreach $v \in V$ **do**
    if $v.color =$ white **then**
        BFS-Visit(G,v)

Breadth First Search for all nodes of a graph. Running time: $\Theta(|V| + |E|)$.

Topological Sorting



Evaluation Order?

Topological Sorting

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

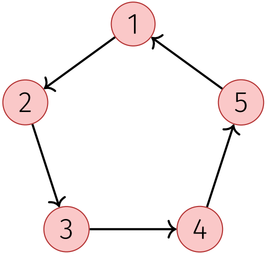
$$\mathrm{ord} : V \to \{1, \dots, |V|\}$$

such that

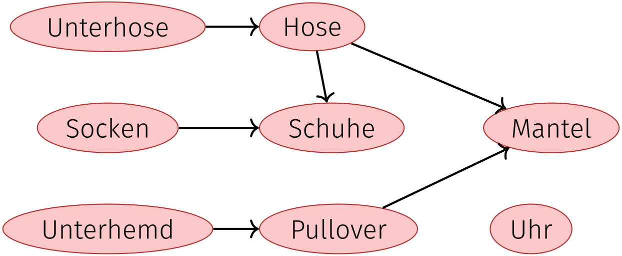
$$\mathrm{ord}(v) < \mathrm{ord}(w) \; \forall \; (v, w) \in E.$$

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

(Counter-)Examples



Cyclic graph: cannot be sorted topologically.



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

Observation

Theorem 7

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

Observation

Theorem 7

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

Inductive Proof Opposite Direction

- Base case $(n = 1)$: Graph with a single node without loop can be sorted topologically, set$\mathrm{ord}(v_1) = 1$.

Inductive Proof Opposite Direction

- Base case ($n = 1$): Graph with a single node without loop can be sorted topologically, set$\mathrm{ord}(v_1) = 1$.
- Hypothesis: Graph with n nodes can be sorted topologically

Inductive Proof Opposite Direction

- Base case ($n = 1$): Graph with a single node without loop can be sorted topologically, set$\mathrm{ord}(v_1) = 1$.
- Hypothesis: Graph with n nodes can be sorted topologically
- Step ($n \to n + 1$):

Inductive Proof Opposite Direction

- Base case ($n = 1$): Graph with a single node without loop can be sorted topologically, setord$(v_1) = 1$.
- Hypothesis: Graph with n nodes can be sorted topologically
- Step ($n \to n + 1$):

  1. G contains a node v_q with in-degree $\deg^-(v_q) = 0$. Otherwise iteratively follow edges backwards – after at most $n + 1$ iterations a node would be revisited. Contradiction to the cycle-freeness.

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 ($n \to n + 1$):

  1. G contains a node v_q with in-degree $\deg^-(v_q) = 0$. Otherwise iteratively follow edges backwards – after at most $n + 1$ iterations a node would be revisited. Contradiction to the cycle-freeness.
  2. Graph without node v_q and without its edges can be topologically sorted by the hypothesis. Now use this sorting and set $\text{ord}(v_i) \leftarrow \text{ord}(v_i) + 1$ 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.

Worst case runtime:

Preliminary Sketch of an Algorithm

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

1. Traverse backwards starting from any node until a node v_q with in-degree 0 is found.
2. If no node with in-degree 0 found after n stepsm, then the graph has a cycle.

Worst case runtime:

Preliminary Sketch of an Algorithm

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

1. Traverse backwards starting from any node until a node v_q with in-degree 0 is found.
2. If no node with in-degree 0 found after n stepsm, then the graph has a cycle.
3. Set $\mathrm{ord}(v_q) \leftarrow d$.

Worst case runtime:

Preliminary Sketch of an Algorithm

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

1. Traverse backwards starting from any node until a node v_q with in-degree 0 is found.
2. If no node with in-degree 0 found after n stepsm, then the graph has a cycle.
3. Set $\operatorname{ord}(v_q) \leftarrow d$.
4. Remove v_q and his edges from G.

Worst case runtime:

Preliminary Sketch of an Algorithm

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

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

Worst case runtime:

Preliminary Sketch of an Algorithm

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

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

Worst case runtime:

Preliminary Sketch of an Algorithm

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

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

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

Improvement

Idea?

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 ← ∅
foreach v ∈ V do A[v] ← 0
foreach (v, w) ∈ E do A[w] ← A[w] + 1 // Compute in-degrees
foreach v ∈ V with A[v] = 0 do push(S, v) // Memorize nodes with in-degree 0
i ← 1
while S ≠ ∅ do
    v ← pop(S); ord[v] ← i; i ← i + 1 // Choose node with in-degree 0
    foreach (v, w) ∈ E do // Decrease in-degree of successors
        A[w] ← A[w] − 1
        if A[w] = 0 then push(S, w)
if i = |V| + 1 then return ord else return "Cycle Detected"
```

Algorithm Correctness

Theorem 8

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

Algorithm Correctness

Theorem 8

Let $G = (V, E)$ be a directed acyclic graph. Algorithm 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 $\mathrm{ord}[u] \leftarrow i$ and thus $\mathrm{ord}[v] > \mathrm{ord}[u]$ for all predecessors u of v. Nodes are put to the stack only once.
3. Runtime: inspection of the algorithm (with some arguments like with graph traversal)

Algorithm Correctness

Theorem 9

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

Algorithm Correctness

Theorem 9

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

Proof: let $\langle v_{i_1}, \ldots, v_{i_k} \rangle$ be a cycle in G. In each step of the algorithm remains $A[v_{i_j}] \geq 1$ for all $j = 1, \ldots, k$. Thus k nodes are never pushed on the stack und therefore at the end it holds that $i \leq V + 1 - k$.

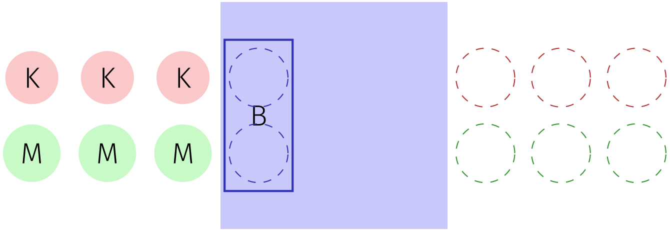
The runtime of the second part of the algorithm can become shorter. But the computation of the in-degree costs already $\Theta(|V| + |E|)$.

13. Shortest Paths

Motivation, Universal Algorithm, Dijkstra's algorithm on distance graphs, [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? [16]



[16] There are slight variations of this problem. It is equivalent to the jealous husbands problem.

Problem as Graph

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

Example

	links	rechts
Missionare	3	0
Kannibalen	3	0
Boot	x	

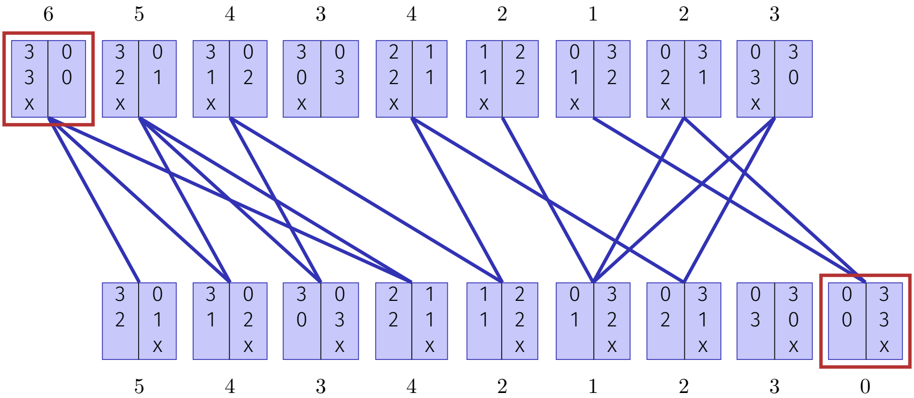
Überfahrt möglich

	links	rechts
Missionare	2	1
Kannibalen	2	1
Boot		x

6 Personen am linken Ufer

4 Personen am linken Ufer

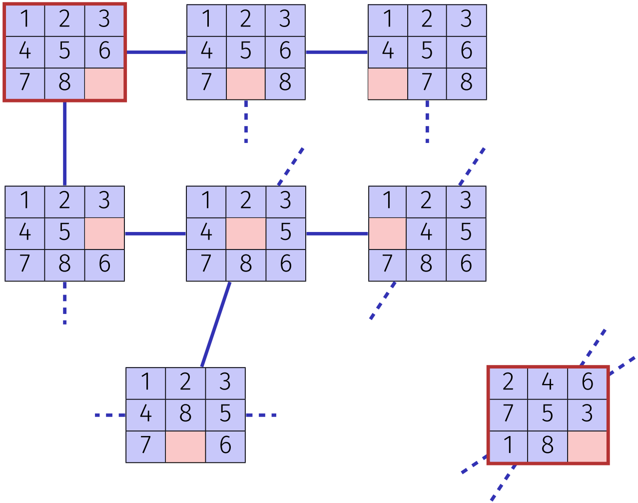
The whole problem as a graph



Another Example: Mystic Square

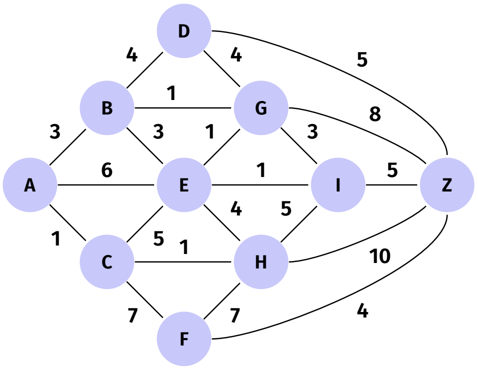
Want to find the fastest solution for

Problem as Graph



Route Finding

Provided cities A - Z and Distances between cities.

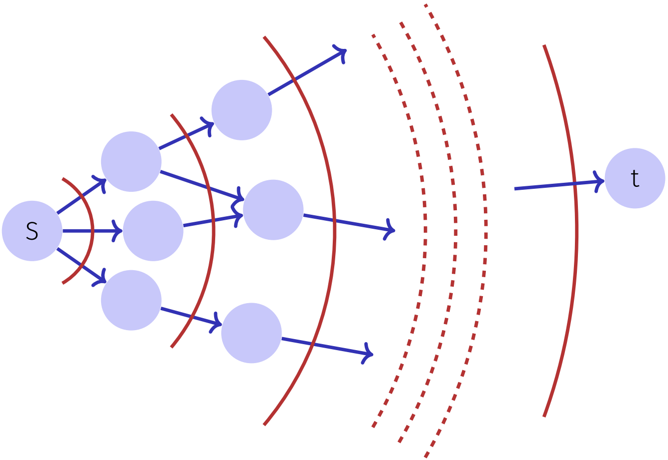


What is the shortest path from A to Z?

Simplest Case

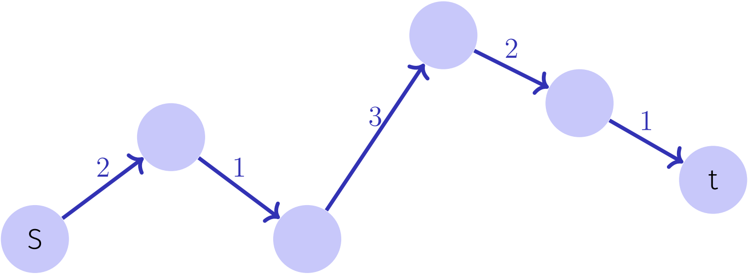
Constant edge weight 1 (wlog)

Solution: Breadth First Search



Weighted Graphs

Given: $G = (V, E, c)$, $c : E \to \mathbb{R}$, $s, t \in V$.
Wanted: Length (weight) of a shortest path from s to t.
Path: $p = \langle s = v_0, v_1, \ldots, v_k = t \rangle$, $(v_i, v_{i+1}) \in E$ $(0 \leq i < k)$
Weight: $c(p) := \sum_{i=0}^{k-1} c((v_i, v_{i+1}))$.



Path with weight 9

Shortest Paths

Notation: we write

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

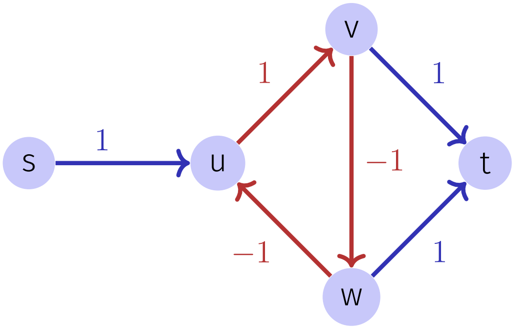
and mean a path p from u to v

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

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

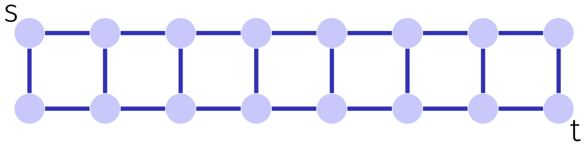
Observations (1)

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



Observations (2)

There can be exponentially many paths.



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

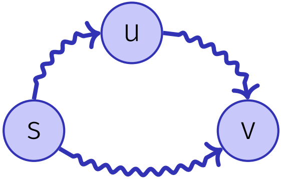
$\Rightarrow$ To try all paths is too inefficient

Observations (3)

Triangle Inequality

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

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

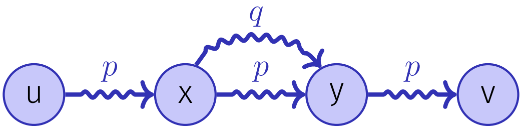


A shortest path from s to v cannot be longer than a shortest path from s to v that has to include u

Observations (4)

Optimal Substructure

Sub-paths of shortest paths are shortest paths. Let $p = \langle v_0, \ldots, v_k \rangle$ be a shortest path from v_0 to v_k. Then each of the sub-paths $p_{ij} = \langle v_i, \ldots, v_j \rangle$ $(0 \leq i < j \leq k)$ 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

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

At the beginning: $d_s[v] = \infty$ for all $v \in V$.
Goal: $d_s[v] = \delta(s, v)$ for all $v \in V$.

- Predecessor of a node

$$\pi_s : V \to V$$

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

General Algorithm

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

   Relaxiere (u, v):

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

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

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

It is Safe to Relax

At any time in the algorithm above it holds

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

It is Safe to Relax

At any time in the algorithm above it holds

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

In the relaxation step:

$$\begin{aligned} \delta(s,v) &\leq \delta(s,u) + \delta(u,v) && \text{[Triangle Inequality].} \\ \delta(s,u) &\leq d_s[u] && \text{[Induction Hypothesis].} \\ \delta(u,v) &\leq c(u,v) && \text{[Minimality of } \delta\text{]} \\ \Rightarrow \quad d_s[u] + c(u,v) &\geq \delta(s,v) \end{aligned}$$

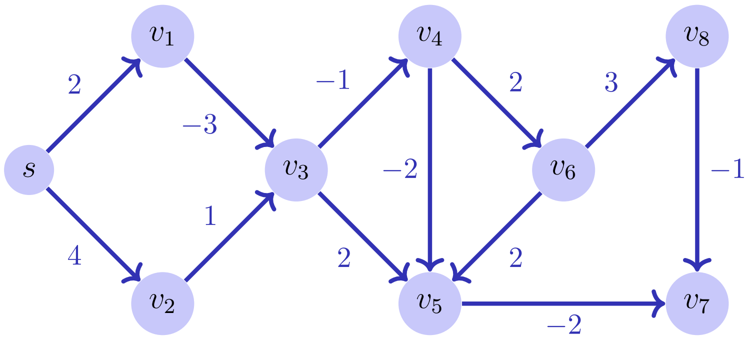
$$\Rightarrow \min\{d_s[v], d_s[u] + c(u,v)\} \geq \delta(s,v)$$

Central Question

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

Special Case: Directed Acyclic Graph (DAG)

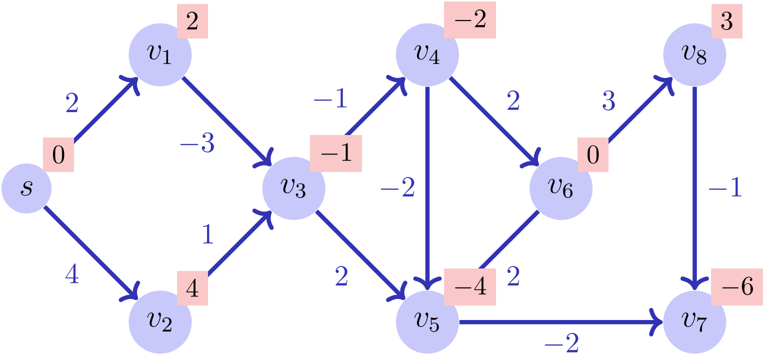
DAG $\Rightarrow$ topological sorting returns optimal visiting order



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

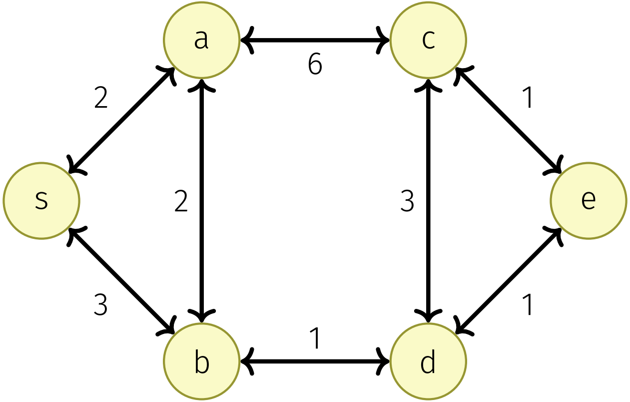
Special Case: Directed Acyclic Graph (DAG)

DAG ⇒ topological sorting returns optimal visiting order



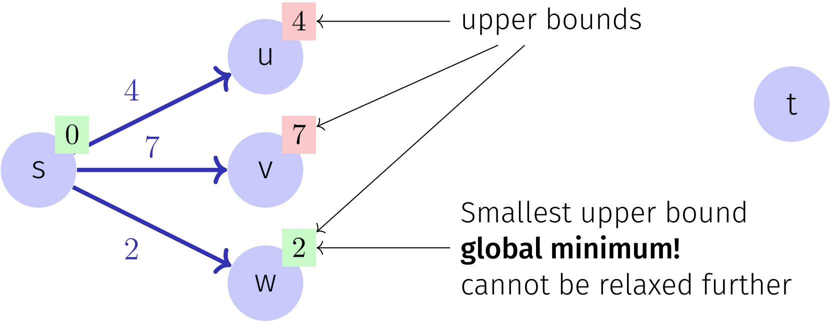
Top. Sort: ⇒ Order $s, v_1, v_2, v_3, v_4, v_6, v_5, v_8, v_7$.

Assumption (preliminary)



All weights of G are **positive**.

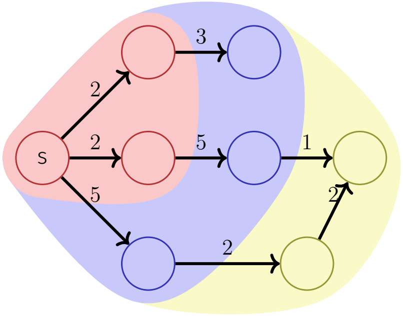
Observation (Dijkstra)



Basic Idea

Set V of nodes is partitioned into

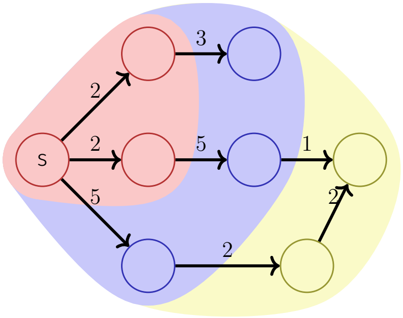
- the set M of nodes for which a shortest path from s is already known,
- the set $R = \bigcup_{v \in M} N^+(v) \setminus M$ of nodes where a shortest path is not yet known but that are accessible directly from M,
- the set $U = V \setminus (M \cup R)$ of nodes that have not yet been considered.



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.



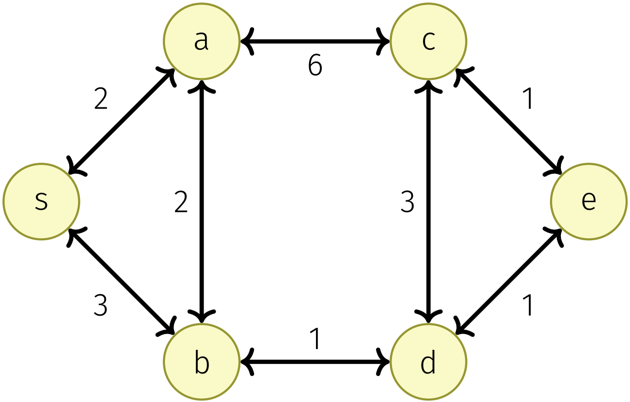
Algorithm Dijkstra(G, s)

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.

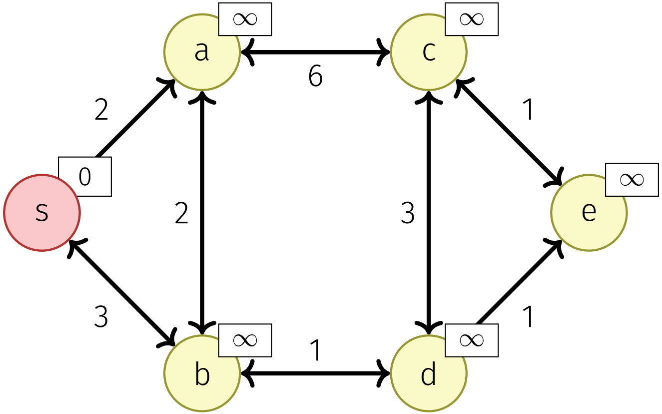
foreach $u \in V$ **do**
  $d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow$ **null**

$d_s[s] \leftarrow 0$; $R \leftarrow \{s\}$
while $R \neq \emptyset$ **do**
  $u \leftarrow$ ExtractMin(R)
  foreach $v \in N^+(u)$ **do**
    if $d_s[u] + c(u, v) < d_s[v]$ **then**
      $d_s[v] \leftarrow d_s[u] + c(u, v)$
      $\pi_s[v] \leftarrow u$
      $R \leftarrow R \cup \{v\}$

Example



Example

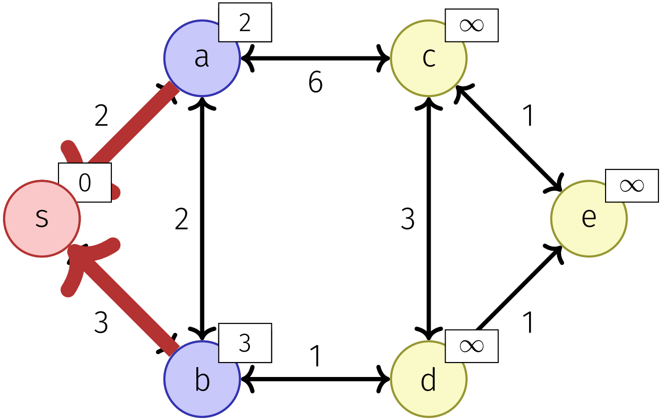


$$M = \{s\}$$

$$R = \{\}$$

$$U = \{a, b, c, d, e\}$$

Example

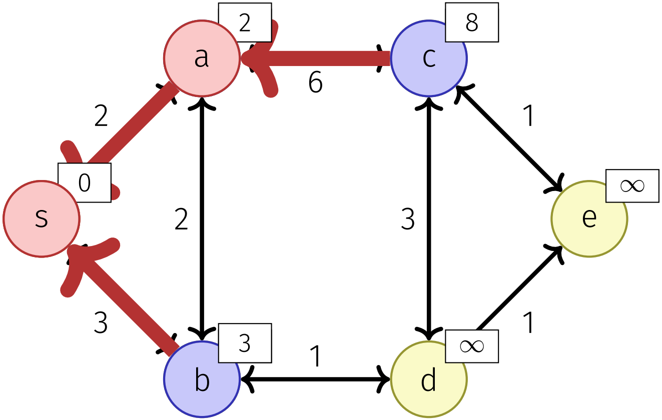


$$M = \{s\}$$

$$R = \{a, b\}$$

$$U = \{c, d, e\}$$

Example

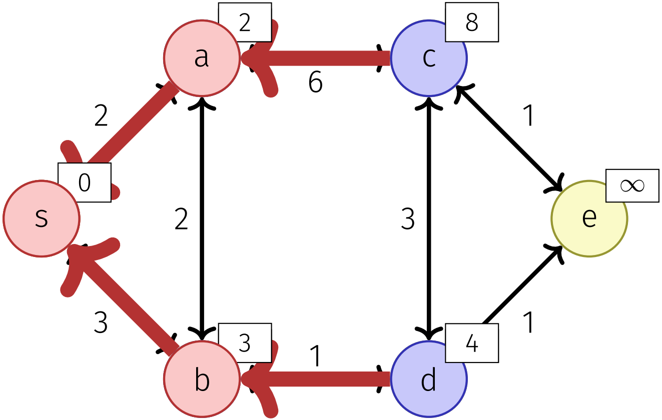


$M = \{s, a\}$

$R = \{b, c\}$

$U = \{d, e\}$

Example

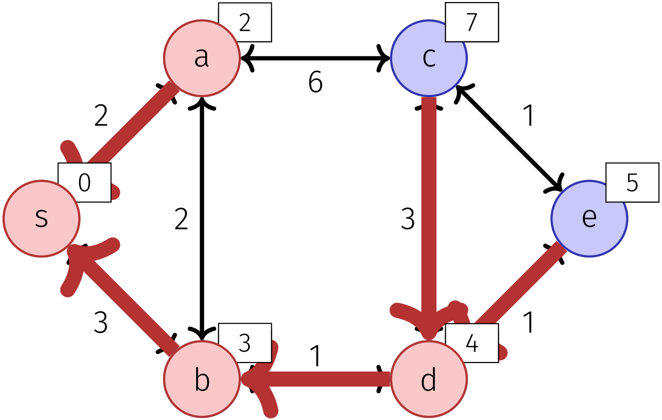


$$M = \{s, a, b\}$$

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

$$U = \{e\}$$

Example

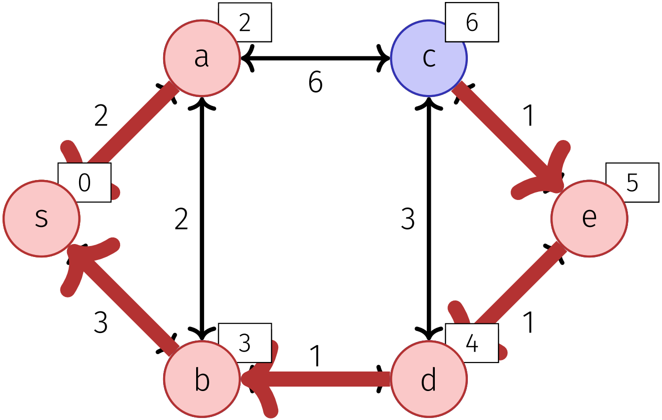


$$M = \{s, a, b, d\}$$

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

$$U = \{\}$$

Example

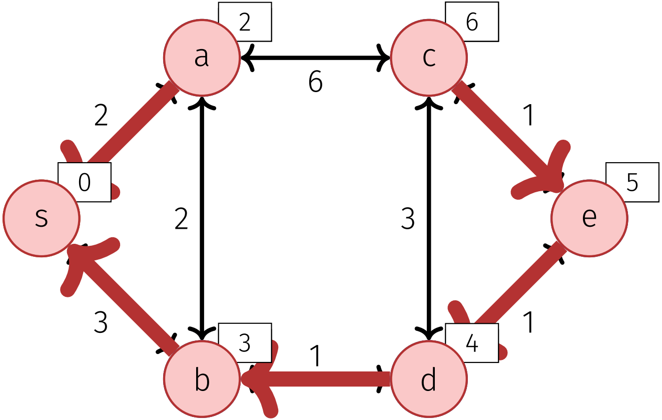


$$M = \{s, a, b, d, e\}$$

$$R = \{c\}$$

$$U = \{\}$$

Example



$$M = \{s, a, b, d, e, c\}$$

$$R = \{\}$$

$$U = \{\}$$

Implementation: Data Structure for R?

Required operations:

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

```
foreach v ∈ N⁺(u) do
    if d_s[u] + c(u,v) < d_s[v] then
        d_s[v] ← d_s[u] + c(u,v)
        π_s[v] ← u
        if v ∈ R then
            DecreaseKey(R, v)          // Update of a d(v) in the heap of R
        else
            R ← R ∪ {v}                // Update of d(v) in the heap of R
```

Implementation: Data Structure for R?

Required operations:

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

```
foreach v ∈ N⁺(u) do
    if d_s[u] + c(u,v) < d_s[v] then
        d_s[v] ← d_s[u] + c(u,v)
        π_s[v] ← u
        if v ∈ R then
            DecreaseKey(R, v)          // Update of a d(v) in the heap of R
        else
            R ← R ∪ {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?

[17] For lazy deletion a pair of egde (or target node) and distance is required.

DecreaseKey

- DecreaseKey: climbing in MinHeap in $\mathcal{O}(\log |V|)$
- Position in the heap?
  - alternative (a): Store position at the nodes

[17] For lazy deletion a pair of egde (or target node) and distance is required.

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

[17] For lazy deletion a pair of egde (or target node) and distance is required.

DecreaseKey

- DecreaseKey: climbing in MinHeap in $\mathcal{O}(\log |V|)$
- Position in the heap?
  - alternative (a): Store position at the nodes
  - alternative (b): Hashtable of the nodes
  - alterantive (c): re-insert node after successful relax operation and mark it "deleted" once extracted (Lazy Deletion).[17]

[17] For lazy deletion a pair of egde (or target node) and distance is required.

Runtime

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

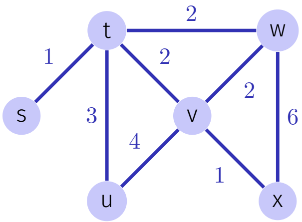
14. Minimum Spanning Trees

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. [18]

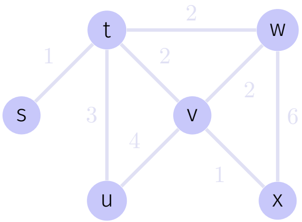
[18] The best known algorithm to solve the TS problem exactly has exponential running time.

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)

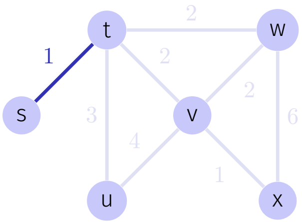
Construct T by adding the cheapest edge that does not generate a cycle.



(Solution is not unique.)

Greedy Idea (Kruskal, 1956)

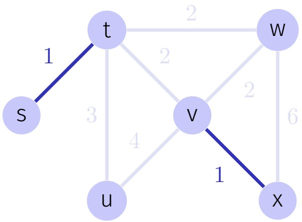
Construct T by adding the cheapest edge that does not generate a cycle.



(Solution is not unique.)

Greedy Idea (Kruskal, 1956)

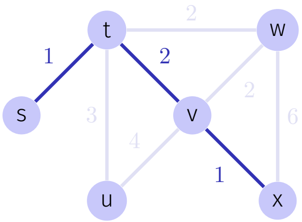
Construct T by adding the cheapest edge that does not generate a cycle.



(Solution is not unique.)

Greedy Idea (Kruskal, 1956)

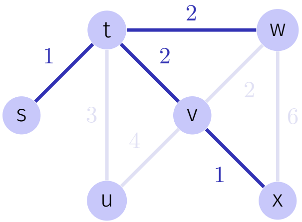
Construct T by adding the cheapest edge that does not generate a cycle.



(Solution is not unique.)

Greedy Idea (Kruskal, 1956)

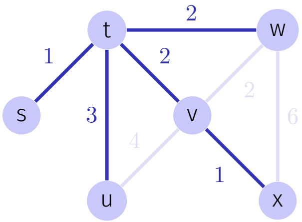
Construct T by adding the cheapest edge that does not generate a cycle.



(Solution is not unique.)

Greedy Idea (Kruskal, 1956)

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



(Solution is not unique.)

Algorithm MST-Kruskal(G)

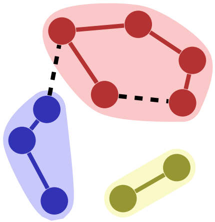
Input: Weighted Graph $G = (V, E, c)$
Output: Minimum spanning tree with edges A.

Sort edges by weight $c(e_1) \leq ... \leq c(e_m)$
$A \leftarrow \emptyset$
for $k = 1$ **to** $|E|$ **do**
    if $(V, A \cup \{e_k\})$ acyclic **then**
        $A \leftarrow A \cup \{e_k\}$

return (V, A, c)

Implementation Issues

Consider a set of sets $i \equiv A_i \subset V$. To identify cuts and cycles: membership of the both ends of an edge to sets?



Implementation Issues

General problem: partition (set of subsets) .e.g.
$\{\{1, 2, 3, 9\}, \{7, 6, 4\}, \{5, 8\}, \{10\}\}$
Required: Abstract data type “Union-Find” with the following operations

- Make-Set(i): create a new set represented by i.
- Find(e): name of the set i that contains e .
- Union(i, j): union of the sets with names i and j.

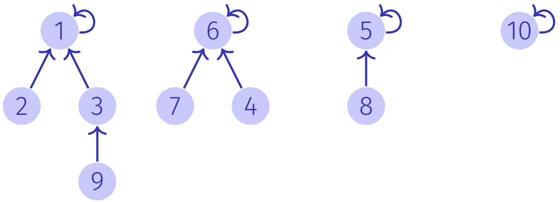
Union-Find Algorithm MST-Kruskal(G)

Input: Weighted Graph $G = (V, E, c)$
Output: Minimum spanning tree with edges A.

```
Sort edges by weight c(e_1) ≤ ... ≤ c(e_m)
A ← ∅
for k = 1 to |V| do
    MakeSet(k)
for k = 1 to m do
    (u, v) ← e_k
    if Find(u) ≠ Find(v) then
        Union(Find(u), Find(v))
        A ← A ∪ e_k
    else                                  // conceptual: R ← R ∪ e_k
return (V, A, c)
```

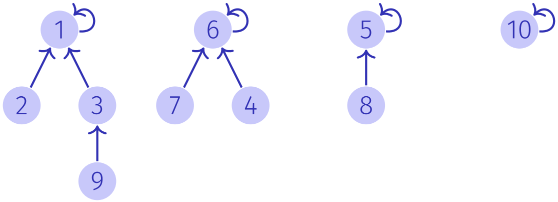
Implementation Union-Find

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) [19]	$p[j] \leftarrow i$;

[19] i and j need to be names (roots) of the sets. Otherwise use Union(Find(i),Find(j))

Optimisation of the runtime for Find

Tree may degenerate. Example: Union$(8,7)$, Union$(7,6)$, Union$(6,5)$, ...

Index	1	2	3	4	5	6	7	8	..
Parent	1	1	2	3	4	5	6	7	..

Worst-case running time of Find in $\Theta(n)$.

Optimisation of the runtime for Find

Idea: always append smaller tree to larger tree. Requires additional size information (array) g

Make-Set(i)	$p[i] \leftarrow i$; $g[i] \leftarrow 1$; **return** i
Union(i, j)	**if** $g[j] > g[i]$ **then** swap(i, j) $p[j] \leftarrow i$ **if** $g[i] = g[j]$ **then** $g[i] \leftarrow g[i] + 1$

$\Rightarrow$ Tree depth (and worst-case running time for Find) in $\Theta(\log n)$

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).[20]

[20] We do not go into details here.

Running time of Kruskal's Algorithm

- Sorting of the edges: $\Theta(|E|\log|E|) = \Theta(|E|\log|V|)$. [21]
- Initialisation of the Union-Find data structure $\Theta(|V|)$
- $|E|\times$ Union(Find(x),Find(y)): $\mathcal{O}(|E|\log|E|) = \mathcal{O}(|E|\log|V|)$.

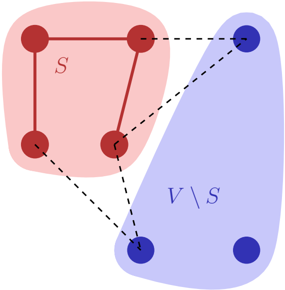
Overal $\Theta(|E|\log|V|)$.

[21]because G is connected: $|V| \le |E| \le |V|^2$

Algorithm of Jarnik (1930), Prim, Dijkstra (1959)

Idea: start with some $v \in V$ and grow the spanning tree from here by the acceptance rule.

$A \leftarrow \emptyset$
$S \leftarrow \{v_0\}$
for $i \leftarrow 1$ **to** $|V|$ **do**
  Choose cheapest (u, v) mit $u \in S$, $v \notin S$
  $A \leftarrow A \cup \{(u, v)\}$
  $S \leftarrow S \cup \{v\}$ // (Coloring)



Remark: a union-Find data structure is not required. It suffices to color nodes when they are added to S.

Running time

Trivially $\mathcal{O}(|V| \cdot |E|)$.

Improvement (like with Dijkstra's ShortestPath)

- With Min-Heap: costs
  - Initialization (node coloring) $\mathcal{O}(|V|)$
  - $|V|\times$ ExtractMin = $\mathcal{O}(|V| \log |V|)$,
  - $|E|\times$ Insert or DecreaseKey: $\mathcal{O}(|E| \log |V|)$,

  $\mathcal{O}(|E| \cdot \log |V|)$

15. Flow in Networks

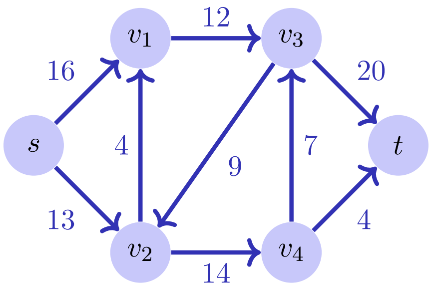
Flow Network, Maximal Flow, Cut, Rest Network, Max-flow Min-cut Theorem, Ford-Fulkerson Method, Edmonds-Karp Algorithm, Maximal Bipartite Matching [Ottman/Widmayer, Kap. 9.7, 9.8.1], [Cormen et al, Kap. 26.1-26.3]

Motivation

- Modelling flow of fluents, components on conveyors, current in electrical networks or information flow in communication networks.
- Connectivity of Communication Networks, Bipartite Matching, Circulation, Scheduling, Image Segmentation, Baseball Eliminination...

Flow Network

- **Flow network** $G = (V, E, c)$: directed graph with **capacities**
- Antiparallel edges forbidden: $(u, v) \in E \;\Rightarrow\; (v, u) \notin E$.
- Model a missing edge (u, v) by $c(u, v) = 0$.
- **Source** s and **sink** t: special nodes. Every node v is on a path between s and t : $s \rightsquigarrow v \rightsquigarrow t$

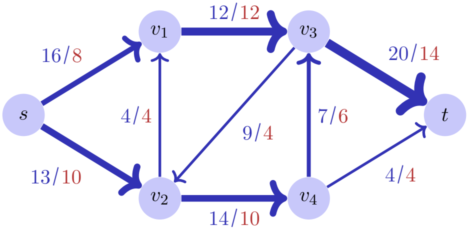


Flow

A **Flow** $f : V \times V \to \mathbb{R}$ fulfills the following conditions:

- **Bounded Capacity**:
  For all $u, v \in V$: $f(u,v) \le c(u,v)$.
- **Skew Symmetry**:
  For all $u, v \in V$: $f(u,v) = -f(v,u)$.
- **Conservation of flow**:
  For all $u \in V \setminus \{s,t\}$:

$$\sum_{v \in V} f(u,v) = 0.$$



Value of the flow:
$|f| = \sum_{v \in V} f(s,v)$.
Here $|f| = 18$.

How large can a flow possibly be?

Limiting factors: cuts

- **cut separating s from t**: Partition of V into S and T with $s \in S$, $t \in T$.
- **Capacity** of a cut: $c(S, T) = \sum_{v \in S, v' \in T} c(v, v')$
- **Minimal cut**: cut with minimal capacity.
- **Flow over the cut**: $f(S, T) = \sum_{v \in S, v' \in T} f(v, v')$

Implicit Summation

Notation: Let $U, U' \subseteq V$

$$f(U,U') := \sum_{\substack{u \in U \\ u' \in U'}} f(u,u'), \qquad f(u,U') := f(\{u\},U')$$

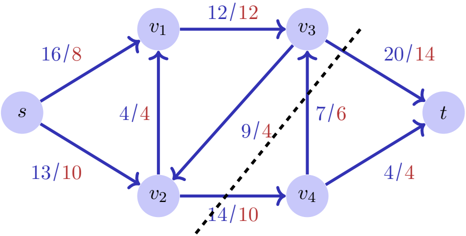
Thus

- $|f| = f(s,V)$
- $f(U,U) = 0$
- $f(U,U') = -f(U',U)$
- $f(X \cup Y, Z) = f(X,Z) + f(Y,Z)$, if $X \cap Y = \emptyset$.
- $f(R,V) = 0$ if $R \cap \{s,t\} = \emptyset$. [flow conversation!]

How large can a flow possibly be?

For each flow and each cut it holds that $f(S,T) = |f|$:

$$f(S,T) = f(S,V) - \underbrace{f(S,S)}_{0} = f(S,V)$$
$$= f(s,V) + f(\underbrace{S - \{s\}}_{\not\ni t, \not\ni s}, V) = |f|.$$

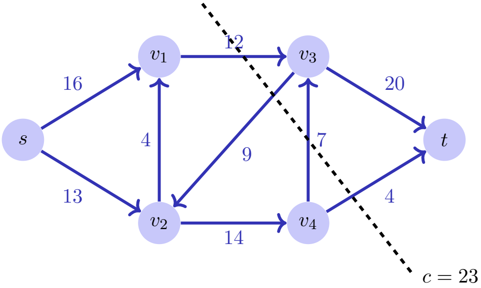


Maximal Flow ?

In particular, for each cut (S, T) of V.

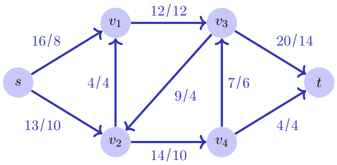
$$|f| \leq \sum_{v \in S, v' \in T} c(v, v') = c(S, T)$$

Will discover that equality holds for $\min_{S,T} c(S, T)$.



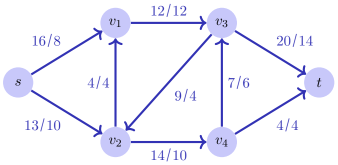
Maximal Flow ?

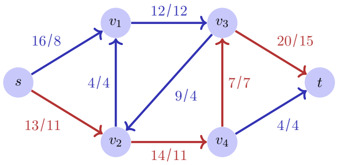
Naive Procedure



Maximal Flow ?

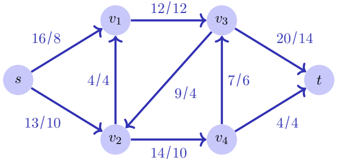
Naive Procedure

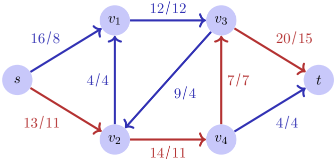


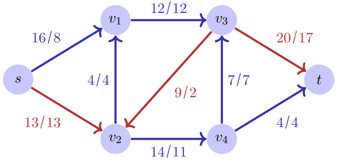


Maximal Flow ?

Naive Procedure

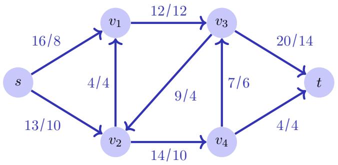


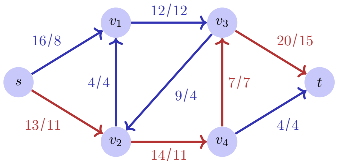


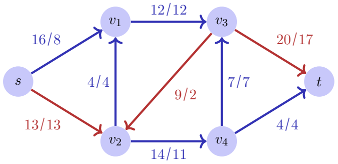


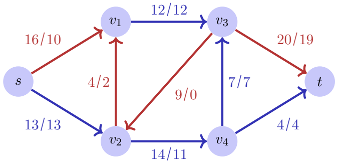
Maximal Flow ?

Naive Procedure



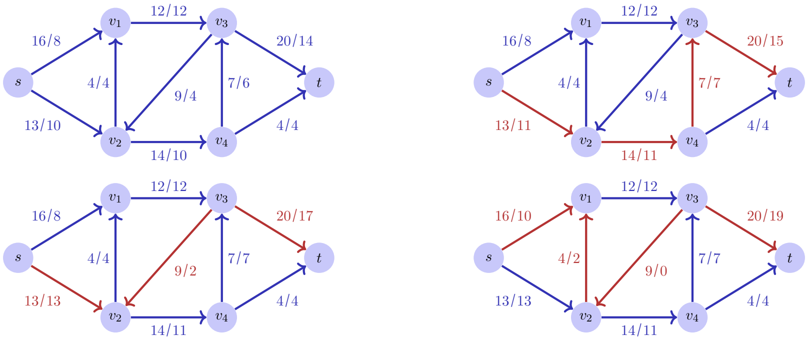






Maximal Flow ?

Naive Procedure



Conclusion: greedy increase of flow does not solve the problem.

The Method of Ford-Fulkerson

- Start with $f(u,v) = 0$ for all $u, v \in V$
- Determine rest network* G_f and expansion path in G_f
- Increase flow via expansion path*
- Repeat until no expansion path available.

$$
\begin{aligned}
G_f &:= (V, E_f, c_f) \\
c_f(u,v) &:= c(u,v) - f(u,v) \quad \forall u, v \in V \\
E_f &:= \{(u,v) \in V \times V | c_f(u,v) > 0\}
\end{aligned}
$$

*Will now be explained

Increase of flow, negative!

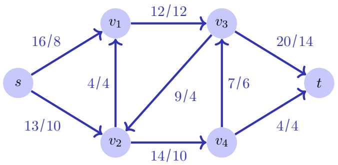
Let some flow f in the network be given.

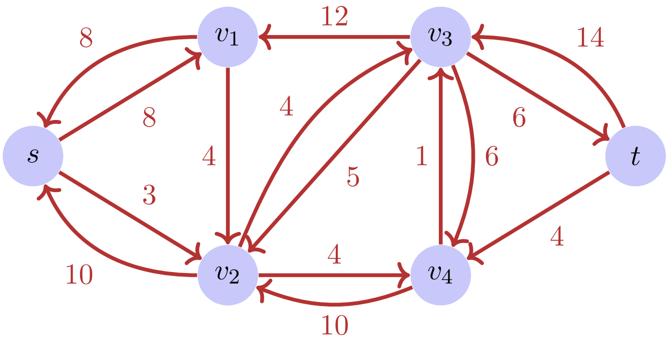
Finding:

- Increase of the flow along some edge possible, when flow can be increased along the edge,i.e. if $f(u, v) < c(u, v)$.
  Rest capacity $c_f(u, v) = c(u, v) - f(u, v) > 0$.
- Increase of flow **against the direction** of the edge possible, if flow can be reduced along the edge, i.e. if $f(u, v) > 0$.
  Rest capacity $c_f(v, u) = f(u, v) > 0$.

Rest Network

Rest network G_f provided by the edges with positive rest capacity:





Rest networks provide the same kind of properties as flow networks with the exception of permitting antiparallel capacity-edges

Observation

Theorem 10

Let $G = (V, E, c)$ be a flow network with source s and sink t and f a flow in G. Let G_f be the corresponding rest networks and let f' be a flow in G_f. Then $f \oplus f'$ with

$$(f \oplus f')(u, v) = f(u, v) + f'(u, v)$$

defines a flow in G with value $|f| + |f'|$.

Proof

$f \oplus f'$ defines a flow in G:

- capacity limit

$$(f \oplus f')(u,v) = f(u,v) + \underbrace{f'(u,v)}_{\leq c(u,v) - f(u,v)} \leq c(u,v)$$

- skew symmetry

$$(f \oplus f')(u,v) = -f(v,u) + -f'(v,u) = -(f \oplus f')(v,u)$$

- flow conservation $u \in V - \{s,t\}$:

$$\sum_{v \in V} (f \oplus f')(u,v) = \sum_{v \in V} f(u,v) + \sum_{v \in V} f'(u,v) = 0$$

Proof

Value of $f \oplus f'$

$$
\begin{aligned}
|f \oplus f'| &= (f \oplus f')(s, V) \\
&= \sum_{u \in V} f(s, u) + f'(s, u) \\
&= f(s, V) + f'(s, V) \\
&= |f| + |f'|
\end{aligned}
$$

■

Augmenting Paths

expansion path p: simple path from s to t in the rest network G_f.
Rest capacity $c_f(p) = \min\{c_f(u, v) : (u, v)$ edge in $p\}$

Flow in G_f

Theorem 11

The mapping $f_p : V \times V \to \mathbb{R}$,

$$f_p(u,v) = \begin{cases} c_f(p) & \text{if } (u,v) \text{ edge in } p \\ -c_f(p) & \text{if } (v,u) \text{ edge in } p \\ 0 & \text{otherwise} \end{cases}$$

provides a flow in G_f *with value* $|f_p| = c_f(p) > 0$.

f_p is a flow (easy to show). there is one and only one $u \in V$ with $(s,u) \in p$. Thus $|f_p| = \sum_{v \in V} f_p(s,v) = f_p(s,u) = c_f(p)$.

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 12

Let f be a flow in a flow network $G = (V, E, c)$ with source s and sink t. The following statementsa are equivalent:

1. *f is a maximal flow in G*
2. *The rest network G_f does not provide any expansion paths*
3. *It holds that $|f| = c(S, T)$ for a cut (S, T) of G.*

Algorithm Ford-Fulkerson(G, s, t)

Input: Flow network $G = (V, E, c)$
Output: Maximal flow f.

for $(u, v) \in E$ **do**
  $f(u, v) \leftarrow 0$

while Exists path $p : s \rightsquigarrow t$ in rest network G_f **do**
  $c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \in p\}$
  foreach $(u, v) \in p$ **do**
    $f(u, v) \leftarrow f(u, v) + c_f(p)$
    $f(v, u) \leftarrow f(v, u) - c_f(p)$

Practical Consideration

In an implementation of the Ford-Fulkerson algorithm the negative flow egdes are usually not stored because their value always equals the negated value of the antiparallel edge.

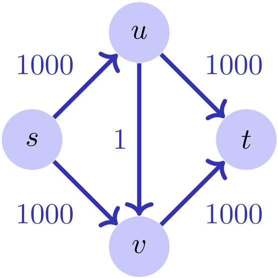
$$f(u,v) \leftarrow f(u,v) + c_f(p)$$
$$f(v,u) \leftarrow f(v,u) - c_f(p)$$

is then transformed to

if $(u,v) \in E$ **then**
  $f(u,v) \leftarrow f(u,v) + c_f(p)$
else
  $f(v,u) \leftarrow f(v,u) - c_f(p)$

Analysis

- The Ford-Fulkerson algorithm does not necessarily have to converge for irrational capacities. For integers or rational numbers it terminates.
- For an integer flow, the algorithms requires maximally $|f_{\max}|$ iterations of the while loop (because the flow increases minimally by 1). Search a single increasing path (e.g. with DFS or BFS) $\mathcal{O}(|E|)$ Therefore $\mathcal{O}(f_{\max}|E|)$.



With an unlucky choice the algorithm may require up to 2000 iterations here.

Edmonds-Karp Algorithm

Choose in the Ford-Fulkerson-Method for finding a path in G_f the expansion path of shortest possible length (e.g. with BFS)

Edmonds-Karp Algorithm

Theorem 13

When the Edmonds-Karp algorithm is applied to some integer valued flow network $G = (V, E)$ with source s and sink t then the number of flow increases applied by the algorithm is in $\mathcal{O}(|V| \cdot |E|)$.

$\Rightarrow$ *Overal asymptotic runtime:* $\mathcal{O}(|V| \cdot |E|^2)$

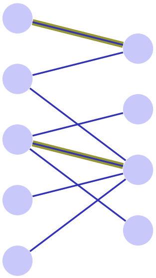
[Without proof]

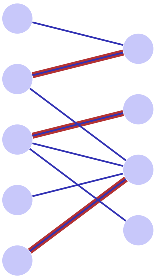
Application: maximal bipartite matching

Given: bipartite undirected graph $G = (V, E)$.

Matching M: $M \subseteq E$ such that $|\{m \in M : v \in m\}| \leq 1$ for all $v \in V$.

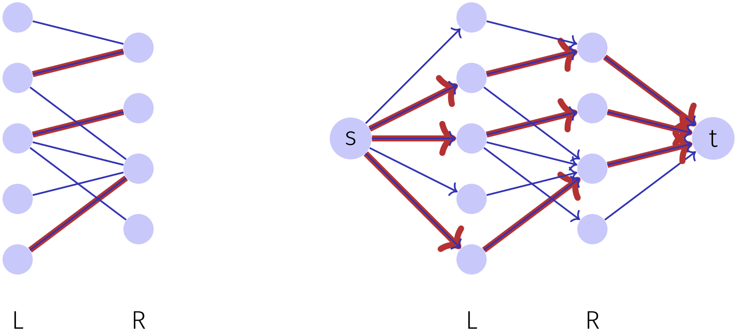
Maximal Matching M: Matching M, such that $|M| \geq |M'|$ for each matching M'.





Corresponding flow network

Construct a flow network that corresponds to the partition L, R of a bipartite graph with source s and sink t, with directed edges from s to L, from L to R and from R to t. Each edge has capacity 1.



16. Dynamic Programming

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

[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 ist the recursive algorithm so slow?

Algorithm FibonacciRecursive(n)

Input: $n \geq 0$
Output: n-th Fibonacci number

if $n < 2$ **then**
  $f \leftarrow n$
else
  $f \leftarrow$ FibonacciRecursive$(n-1)$ + FibonacciRecursive$(n-2)$
return f

Analysis

$T(n)$: Number executed operations.

- $n = 0, 1$: $T(n) = \Theta(1)$

Analysis

$T(n)$: Number executed operations.

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

Analysis

$T(n)$: Number executed operations.

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

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

Analysis

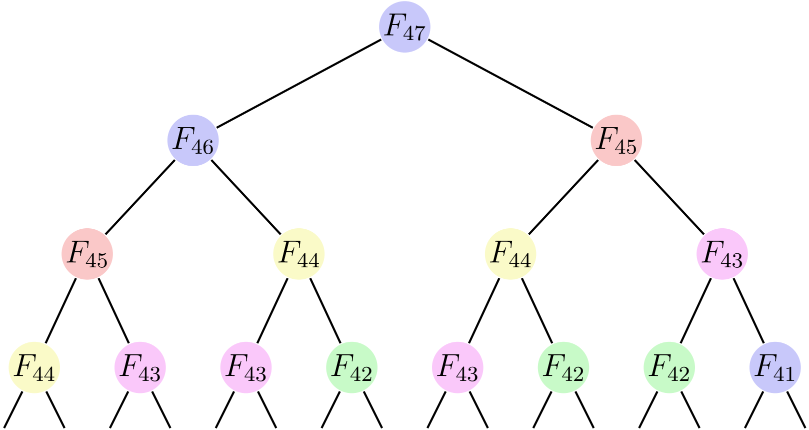
$T(n)$: Number executed operations.

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

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

Algorithm is **exponential** in n.

Reason (visual)



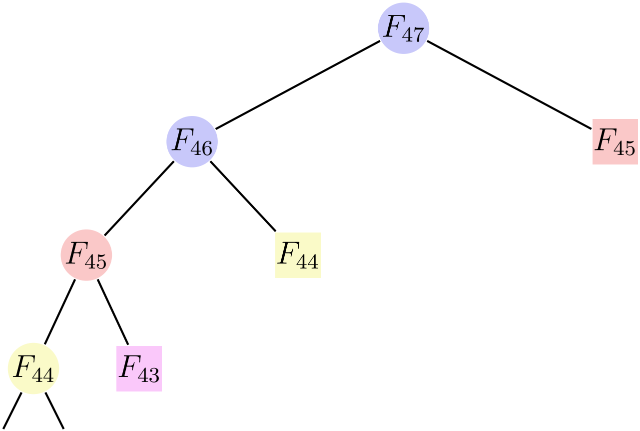
Nodes with same values are evaluated (too) often.

Memoization

Memoization (sic) saving intermediate results.

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

Memoization with Fibonacci



Rechteckige Knoten wurden bereits ausgewertet.

Algorithm FibonacciMemoization(n)

Input: $n \geq 0$
Output: n-th Fibonacci number

if $n \leq 2$ **then**
    $f \leftarrow 1$
else if $\exists$memo$[n]$ **then**
    $f \leftarrow$ memo$[n]$
else
    $f \leftarrow$ FibonacciMemoization$(n-1)$ + FibonacciMemoization$(n-2)$
    memo$[n] \leftarrow f$

return f

Analysis

Computational complexity:

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

because after the call to $f(n-1)$, $f(n-2)$ has already been computed.

A different argument: $f(n)$ is computed exactly once recursively for each n. Runtime costs: n calls with $\Theta(1)$ costs per call $n \cdot c \in \Theta(n)$. The recursion vanishes from the running time computation.

Algorithm requires $\Theta(n)$ memory.[22]

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

Looking closer ...

... the algorithm computes the values of F_1, F_2, F_3,... in the **top-down** approach of the recursion.

Can write the algorithm **bottom-up**. This is characteristic for **dynamic programming**.

Algorithm FibonacciBottomUp(n)

Input: $n \geq 0$
Output: n-th Fibonacci number

$F[1] \leftarrow 1$
$F[2] \leftarrow 1$
for $i \leftarrow 3, \ldots, n$ **do**
  $F[i] \leftarrow F[i-1] + F[i-2]$

return $F[n]$

Dynamic Programming: Idea

- Divide a complex problem into a reasonable number of sub-problems
- The solution of the sub-problems will be used to solve the more complex problem
- Identical problems will be computed only once

Dynamic Programming Consequence

Identical problems will be computed only once

$\Rightarrow$ Results are saved

We trade spee against memory consumption

Dynamic Programming: Description

1. Use a **DP-table** with information to the subproblems.
   Dimension of the entries? Semantics of the entries?

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?

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?

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?

Dynamic Programming: Description

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

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

Dynamic Programing: Description with the example

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

Dynamic Programing: Description with the example

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

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

Dynamic Programing: Description with the example

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

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

2. **Which entries do not depend on other entries?**

Dynamic Programing: Description with the example

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

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

2. **Which entries do not depend on other entries?**

   Values F_1 and F_2 can be computed easily and independently.

Dynamic Programing: Description with the example

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

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

2. **Which entries do not depend on other entries?**

   Values F_1 and F_2 can be computed easily and independently.

3. **Computation order?**

Dynamic Programing: Description with the example

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

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

2. **Which entries do not depend on other entries?**

   Values F_1 and F_2 can be computed easily and independently.

3. **Computation order?**

   F_i with increasing i.

Dynamic Programing: Description with the example

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

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

2. **Which entries do not depend on other entries?**

   Values F_1 and F_2 can be computed easily and independently.

3. **Computation order?**

   F_i with increasing i.

4. **Reconstruction of a solution?**

Dynamic Programing: Description with the example

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

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

2. **Which entries do not depend on other entries?**

   Values F_1 and F_2 can be computed easily and independently.

3. **Computation order?**

   F_i with increasing i.

4. **Reconstruction of a solution?**

   F_n ist die n-te Fibonacci-Zahl.

Dynamic Programming = Divide-And-Conquer ?

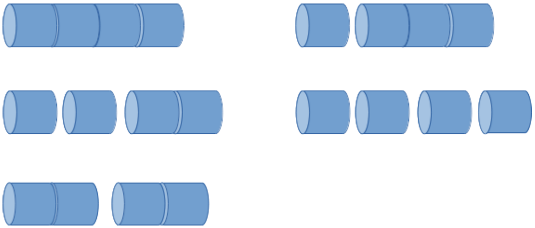
- In both cases the original problem can be solved (more easily) by utilizing the solutions of sub-problems. The problem provides **optimal substructure**.
- Divide-And-Conquer algorithms (such as Mergesort): sub-problems are independent; their solutions are required only once in the algorithm.
- DP: 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**

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

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)

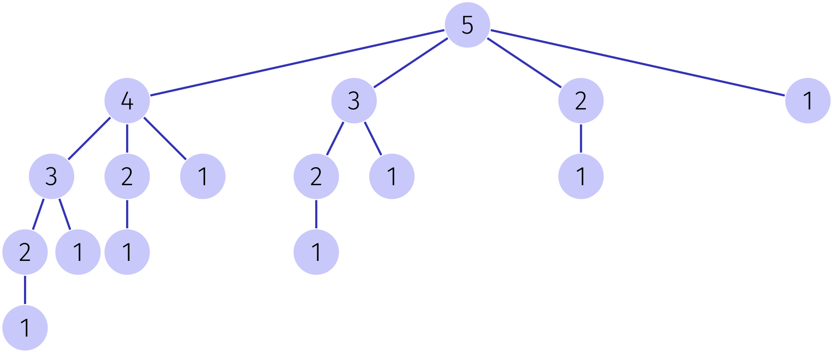
Input: $n \geq 0$, Prices v
Output: best value

$q \leftarrow 0$
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 $T(n) = \sum_{i=0}^{n-1} T(i) + c \quad \Rightarrow$[23] $\quad T(n) \in \Theta(2^n)$

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

Recursion Tree



Algorithm RodCutMemoized(m, v, n)

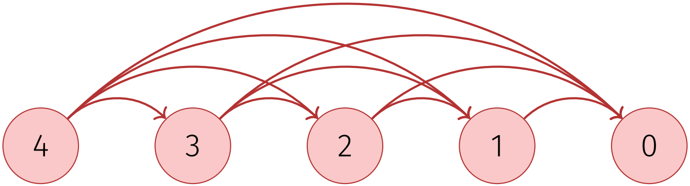
Input: $n \geq 0$, Prices v, Memoization Table m
Output: best value

$q \leftarrow 0$
if $n > 0$ **then**
  if $\exists\ m[n]$ **then**
    $q \leftarrow m[n]$
  else
    for $i \leftarrow 1, \ldots, n$ **do**
      $q \leftarrow \max\{q, v_i + \text{RodCutMemoized}(m, v, n-i)\}$;
    $m[n] \leftarrow q$
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 lenght of the first rod in a separate table of length n

Bottom-up Description with the example

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

Bottom-up Description with the example

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

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

Bottom-up Description with the example

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

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

2. **Which entries do not depend on other entries?**

Bottom-up Description with the example

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

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

2. **Which entries do not depend on other entries?**

   Value r_0 is 0

Bottom-up Description with the example

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

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

2. **Which entries do not depend on other entries?**

   Value r_0 is 0

3. **Computation order?**

   .

Bottom-up Description with the example

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

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

2. **Which entries do not depend on other entries?**

   Value r_0 is 0

3. **Computation order?**

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

Bottom-up Description with the example

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

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

2. **Which entries do not depend on other entries?**

   Value r_0 is 0

3. **Computation order?**

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

4. **Reconstruction of a solution?**

Bottom-up Description with the example

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

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

2. **Which entries do not depend on other entries?**

   Value r_0 is 0

3. **Computation order?**

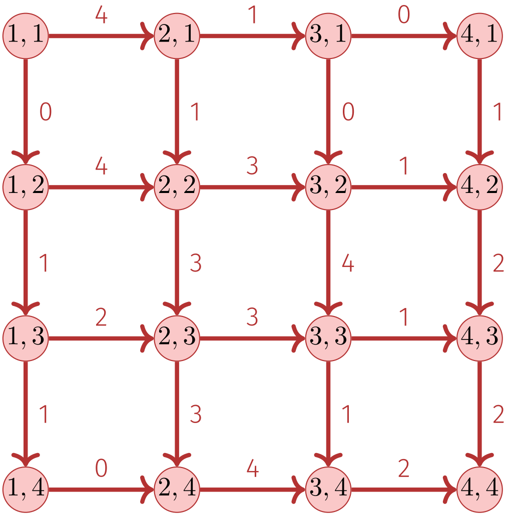
   r_i, $i = 1, \ldots, n$.

4. **Reconstruction of a solution?**

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

Rabbit!

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

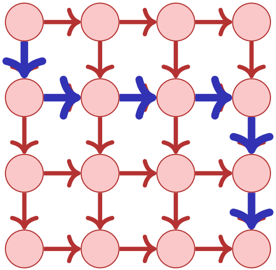


Rabbit!

Number of possible paths?

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

$\Rightarrow$ No chance for a naive algorithm



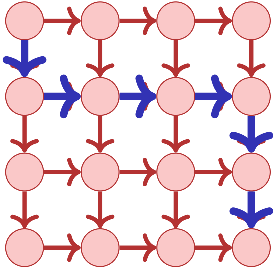
The path 100011
(1:to south, 0: to east)

Rabbit!

Number of possible paths?

- Choice of $n-1$ ways to south out of $2n-2$ ways overal.
- $$\binom{2n-2}{n-1} \in \Omega(2^n)$$

$\Rightarrow$ No chance for a naive algorithm



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

Recursion

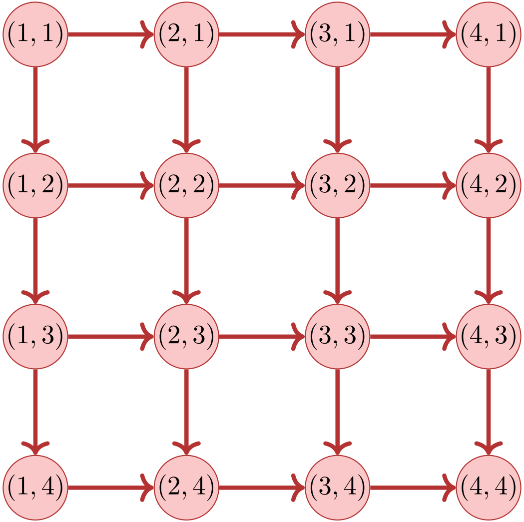
Wanted: $T_{0,0}$ **= maximal number carrots from** $(0,0)$ **to** (n,n)**.**

Let $w_{(i,j)-(i',j')}$ number of carrots on egde from (i,j) to (i',j').

Recursion (maximal number of carrots from (i,j) to (n,n)

$$T_{ij} = \begin{cases} \max\{w_{(i,j)-(i,j+1)} + T_{i,j+1}, w_{(i,j)-(i+1,j)} + T_{i+1,j}\}, & i < n, j < n \\ w_{(i,j)-(i,j+1)} + T_{i,j+1}, & i = n, j < n \\ w_{(i,j)-(i+1,j)} + T_{i+1,j}, & i < n, j = n \\ 0 & i = j = n \end{cases}$$

Graph of Subproblem Dependencies



Bottom-up Description with the example

Dimension of the table? Semantics of the entries?

1.

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).

Bottom-up Description with the example

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

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

2. **Which entries do not depend on other entries?**

Bottom-up Description with the example

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

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

2. **Which entries do not depend on other entries?**

   Value $T_{n,n}$ is 0

Bottom-up Description with the example

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

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

2. **Which entries do not depend on other entries?**

   Value $T_{n,n}$ is 0

3. **Computation order?**

   .

Bottom-up Description with the example

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

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

2. **Which entries do not depend on other entries?**

   Value $T_{n,n}$ is 0

3. **Computation order?**

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

Bottom-up Description with the example

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

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

2. **Which entries do not depend on other entries?**

   Value $T_{n,n}$ is 0

3. **Computation order?**

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

4. **Reconstruction of a solution?**

Bottom-up Description with the example

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

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

2. **Which entries do not depend on other entries?**

   Value $T_{n,n}$ is 0

3. **Computation order?**

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

4. **Reconstruction of a solution?**

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

17. Dynamic Programming II

Editing Distance, Bellman-Ford Algorithm

[Cormen et al, Kap. 24.1]]

Minimal Editing Distance

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

Editing operations:

- Insertion of a character
- Deletion of a character
- Replacement of a character

Question: how many editing operations at least required in order to transform string A into string B.

TIGER ZIGER ZIEGER ZIEGE

Minimal Editing Distance

Wanted: cheapest character-wise transformation $A_n \to B_m$ with costs

operation	Levenshtein	LCS[24]	general
Insert c	1	1	ins(c)
Delete c	1	1	del(c)
Replace $c \to c'$	$\mathbb{1}(c \neq c')$	$\infty \cdot \mathbb{1}(c \neq c')$	repl(c, c')

Beispiel

T	I	G	E	R	T	I	_	G	E	R	T→Z	+E	-R
Z	I	E	G	E	Z	I	E	G	E	_	Z→T	-E	+R

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

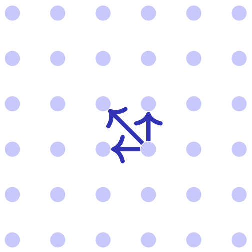
DP

0. $E(n, m)$ = mimimum number edit operations (ED cost) $a_{1...n} \to b_{1...m}$
1. Subproblems $E(i, j)$ = ED von $a_{1...i}$. $b_{1...j}$. #SP $= n \cdot m$
2. Guess Costs$\Theta(1)$
   - $a_{1..i} \to a_{1...i-1}$ (delete)
   - $a_{1..i} \to a_{1...i}b_j$ (insert)
   - $a_{1..i} \to a_{1...i-1}b_j$ (replace)
3. Rekursion

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

DP

4. Dependencies



$\Rightarrow$ Computation from left top to bottom right. Row- or column-wise.

5. Solution in $E(n, m)$

Example (Levenshtein Distance)

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

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

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

Bottom-Up DP algorithm ED

Dimension of the table? Semantics?

1.

Bottom-Up DP algorithm ED

Dimension of the table? Semantics?

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

Bottom-Up DP algorithm ED

Dimension of the table? Semantics?

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

Computation of an entry

2.

Bottom-Up DP algorithm ED

Dimension of the table? Semantics?

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

Computation of an entry

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

Bottom-Up DP algorithm ED

3. **Computation order**

Bottom-Up DP algorithm ED

3. **Computation order**

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

Bottom-Up DP algorithm ED

3. **Computation order**

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

4. **Reconstruction of a solution?**

Bottom-Up DP algorithm ED

3. **Computation order**

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

4. **Reconstruction of a solution?**

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

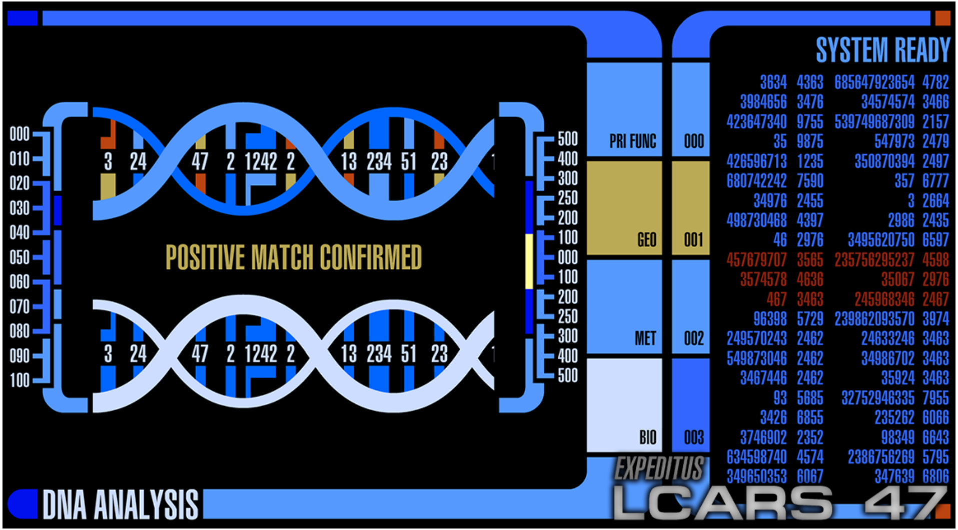
Analysis ED

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

Runtime overal:

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

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.

Reminder: Shortest Path Algorithm

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

   Relaxiere (u, v):

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

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

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

Dynamic Programming Approach (Bellman)

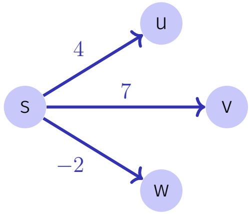
Induction over number of edges $d_s[i,v]$: Shortest path from s to v via maximally i edges.

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

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

Dynamic Programming Approach (Bellman)

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



Algorithm: Iterate over last row until the relaxation steps do not provide any further changes, maximally $n-1$ iterations. If still changes, then there is no shortest path.

Algorithm Bellman-Ford(G, s)

Input: Graph $G = (V, E, c)$, starting point $s \in V$
Output: If return value true, minimal weights d for all shortest paths from s, otherwise no shortest path.

foreach $u \in V$ **do**
    $d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow$ **null**
$d_s[s] \leftarrow 0$;
for $i \leftarrow 1$ **to** $|V|$ **do**
    $f \leftarrow$ false
    foreach $(u, v) \in E$ **do**
        $f \leftarrow f \vee \text{Relax}(u, v)$
    if $f =$ false **then return** true
return false;