

# Algorithms and Data Structures

Ulrike von Luxburg

Winter 2013/14

Department of Computer Science, University of Hamburg

(Version as of October 17, 2013)

Contents will be updated continuously, see the webpage for the latest version.

# Table of contents

## Design and analysis of Algorithms: Introduction and Methods

Introduction .....	10
What is it about? .....	11
Why is it important for you? .....	18
Analysis of algorithms: The formal setup .....	23
What is an algorithm? .....	24
Appetizer: computing Fibonacci numbers .....	28
Pseudo-code .....	40
O-Notation .....	44
Running time analysis .....	53
Space complexity analysis .....	75
Appetizer: Multiplying two integers .....	77

# Table of contents (2)

Tool: Master theorem for solving simple recurrence relations ..... 84

## Basic Data Structures

Arrays and lists .....	100
Trees .....	108
Stack and queue .....	117
Heaps and priority queues .....	120
Heaps .....	121
Priority queue .....	145
Hashing .....	148
Basics .....	149
Some simple hash functions .....	157

## Sorting

Elementary Sorting Algorithms .....	166
Selection sort .....	169

# Table of contents (3)

Insertion sort .....	184
Bubble sort .....	190
Mergesort .....	192
Heap sort .....	205
Lower bound .....	218
Appetizer / Excursion: finding the median .....	232
Quicksort .....	254
Sorting in linear time .....	274
Counting sort / Ksort .....	276
Radix sort .....	296
Bucket sort / uniform sort .....	301
Sorting: summary and state of the art .....	310
High-level summary: lessons about algorithm development .....	315

## Searching

# Table of contents (4)

Search Trees .....	319
AVL trees? .....	320
(a,b)-Trees .....	321
Red-Black-Trees .....	322

# Standard Literature

- ▶ The lecture closely follows the following book:  
K. Mehlhorn, P. Sanders: Algorithms and data structures: the basic toolbox. Springer, 2008.  
(25 Euros via Springer mycopy)  
German version of the book was supposed to be available this autumn, but got delayed until next year ☹:  
K. Mehlhorn, P. Sanders, M. Dietzfelbinger: Algorithmen und Datenstrukturen. Springer, 2014. 25 Euros.
- ▶ A book I like a lot because it conveys intuitions (not available in German):  
Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani.  
Algorithms. McGraw-Hill, 2008.  
pdf preprint available online.

## Standard Literature (2)

- ▶ The international standard text book on algorithms and data structures is the following:  
**Cormen, Leiserson, Rivest, Stein: Introduction to algorithms and data structures. 3rd edition. MIT Press, 2009.**  
The book also exists in German translation. Many copies available in the library, both in german and english.
- ▶ Another by-now-standard book on algorithms (english only):  
Jon Kleinberg, Eva Tardos: Algorithm Design. 2005

## Standard Literature (3)

For fun reading on algorithms (popular science style) I highly recommend:

- ▶ Vöcking et al., Taschenbuch der Algorithmen. Springer, 2008  
(English version: Algorithms unplugged, Springer 2011)
- ▶ John MacCormick: 9 Algorithms that changed the future.  
Princeton University Press, 2012

For preparing a job interview at one of the top companies:

- ▶ G. Laakmann: Cracking the coding interview. 2010.

# Design and analysis of Algorithms: Introduction and Methods

# Introduction

# What is it about?

# Algorithms occur in every days life, all the time!

- ▶ Railway:
  - ▶ Build a time table that satisfies the requirements.
  - ▶ Assign which physical train is supposed to serve which line of the schedule.
  - ▶ If one line is blocked because of an accident, quickly find an alternative schedule that minimizes the delays
- ▶ Stock market: auctions are performed by computers
  - ▶ Have to solve difficult combinatorial optimization problems.
  - ▶ Time is crucial (trading happens within milliseconds), and of course the quality of the solution is crucial.
- ▶ Biology:
  - ▶ Sequencing a gene!
  - ▶ Once you have the genetic code, analyze it! Compare it to others.

# Algorithms occur in every days life, all the time!

## (2)

- ▶ Internet:
  - ▶ Network protocols that are robust to failures.
  - ▶ Routing standards  $\leadsto$  questions about advantages/disadvantages of various shortest path algorithms
- ▶ Companies like google:
  - ▶ How can we index all internet pages?
  - ▶ How can we save storage capacity without giving up on redundancy?
  - ▶ How can we efficiently answer queries, say in less than a second?

# Computer science is not so much about programming ...

- ▶ Many people can program (or think that they can program). You don't need to be a computer scientist to be able to write Java Code.
- ▶ But to solve real-world problems, you need to know more than the syntax of Java. You cannot just sit down and implement solutions!
- ▶ What a computer **scientist** should be able to do:
  - ▶ Given a real world problem, "**abstract away**" all the messy, application-dependent details
  - ▶ Have a **toolbox** how to approach various problems
  - ▶ **Analyze**: How difficult is the problem? How good is my solution? Can I hope to get a better one?

# Computer science is not so much about programming ... (2)

- Once you come up with a **solution, describe it** precisely enough such that a programmer can sit down and implement it

# Algorithms are the heart of computer science!

Wikipedia: “*Computer science is the study of the theoretical foundations of information and computation and of practical techniques for their implementation and application in computer systems. Computer scientists invent algorithmic processes that create, describe, and transform information and formulate suitable abstractions to model complex systems.*”

- ▶ Want to solve particular problems with the help of a computer.
- ▶ Model the problem in a way that is abstract enough to be able to study it (that is, more abstract than actual Java code)
- ▶ Derive good solutions for it.
- ▶ Always keep in the back of our mind, that at the end of the day our solution is supposed to run on a computer.

# Algorithms are the heart of computer science! (2)

It is designing algorithms that makes computer science different from other disciplines like mathematics or electrical engineering.

Why is it important for *you*?

# Your later job!

- ▶ Most computer scientists are bound to run into algorithmic problems in their jobs
- ▶ You should at least be able to tell when this happens!
- ▶ You should have at least a basic understanding of how such problems might be handled (e.g., use an efficient data structure).
- ▶ This basic understanding is necessary to find reasonable solutions (even if you just search for literature)

# Example questions from a google job interview

- ▶ Given two binary trees:  $T_1$  (millions of nodes),  $T_2$  (hundreds of nodes). Design an algorithm to decide whether  $T_2$  is a subtree of  $T_1$ .
- ▶ You are given two sorted arrays,  $A$  and  $B$ .  $A$  has enough buffer at the end to hold  $B$ . Write a method to merge  $B$  directly into  $A$  in sorted order (without using any extra buffer).
- ▶ Design an algorithm to remove the duplicate characters in a string, without using any additional buffer (one or two additional variables are allowed, an extra copy of the array is not allowed).
- ▶ Given an image represented by an  $n \times n$  matrix, where each pixel in the image is 4 bytes, write a method to rotate the image by 90 degrees.

## Example questions from a google job interview (2)

- ▶ Assume you build a browser-game for tic-tac-toe that is going to be played online (on your server). How do you design a function to decide if someone has won? What if the board has size  $N \times N$  instead of  $3 \times 3$ ?
- ▶ Describe an algorithm to find the largest 1 million numbers in 1 billion numbers. Assume that the computer memory can hold all the 1 billion numbers.

# Formalities for this lecture

- ▶ Literatur
- ▶ Übungsgruppen
- ▶ Olat-Quizze
- ▶ Klausurzulassung
- ▶ Vorlesungszeiten

Was noch?

# Analysis of algorithms: The formal setup

# What is an algorithm?

# What is an algorithm?

Informal definition from wikipedia:

*An algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function. Starting from an initial state and initial input (perhaps empty), the instructions describe a computation that, when executed, proceeds through a finite number of well-defined successive states, eventually producing output and terminating at a final ending state.*

## What is an algorithm? (2)

It is pretty much impossible to formally define what an algorithm is.  
See the following references:

- ▶ Y. Moschovakis. What is an algorithm? In B. Engquist, W. Schmid (editors): Mathematics Unlimited, 2001 and beyond. Springer, 2001.
- ▶ A. Blass and N. Dershowitz and Y. Gurevich. When are two algorithms the same? Bulletin of Symbolic Logic, 2009.

We are going to come back to this issue at the end of the semester.

# Where does the name come from?

Book written by **Al Khwarizmi** in the 9th century: explains how to use the decimal number system.

This is the book that made the decimal system known to the western world.

Gave exact, precise, non-ambiguous descriptions about how to add, subtract, multiply, divide, take square roots, solve linear equations, etc.

Was translated into Latin in the 12th century under the title  
**“Algoritmi de numero Indorum”**

## Appetizer: computing Fibonacci numbers

Literature: Dasgupta Chapter 0

# Fibonacci numbers

Can you recall the definition of Fibonacci numbers?

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2 \end{cases}$$

Gives the sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Fibonacci numbers grow very fast. It is easy to prove that for  $n \geq 6$  we have  $2^{n/2} \leq F_n \leq 2^n$ .

## Fibonacci numbers (2)

We are now going to look at various algorithms to compute  $F_n$  and argue about how good they are.

We keep it informal in this section, take it as an appetizer.

# Fibonacci numbers (3)

EVERYBODY:

- ▶ CAN YOU TRY TO WRITE DOWN AN ALGORITHM TO COMPUTE THE FIBONACCI NUMBERS?
- ▶ CAN YOU ALSO COME UP WITH A DIFFERENT ALGORITHM FOR COMPUTING FIBONACCI NUMBERS?

# Algorithm 1: naive loop with lookup

FibLoop( $n$ )

- 1 Create Array  $A(0, \dots, n)$
- 2  $A(0) = 0; A(1) = 1$
- 3 **for**  $i=2, \dots, n$
- 4      $A(i) = A(i - 1) + A(i - 2)$
- 5 Return  $A(n)$

## Algorithm 2: recursive

FibRecursive( $n$ )

```
1 if  $n=0$ 
2   Return 0
3 if  $n=1$ 
4   Return 1
5 Return FibRecursive( $n-1$ ) + FibRecursive( $n-2$ )
```

# Running times of the Fibonacci algorithms

WHAT DO YOU THINK, WHICH ALGORITHM IS GOING TO TAKE LONGER? WHY?

HOW WOULD YOU ARGUE FORMALLY?

# Running times of the Fibonacci algorithms (2)

How many “elementary operations”  $T(n)$  do we have to perform in `FibLoop(n)`?

- ▶ Line 1: Allocate an array of length  $n$ : depending on how exactly this allocation procedure works, it might take either 1 elementary operation (just give a pointer to the beginning of the array) or up to  $n$  elementary operations
- ▶ Line 2: 2 elementary operations
- ▶ In the loop, we have to add two numbers.
  - ▶ These two numbers can be as large as  $F_{n-1}$ .
  - ▶ We have already seen that  $2^{n/2} \leq F_n \leq 2^n$ . So in binary representation,  $F_n$  is going to use at most  $n$  bits.
  - ▶ To add two numbers with  $n$  bits needs on the order of  $n$  elementary operations (just take this as a fact for now).
- ▶ The loop is going to be executed  $n - 1$  times.

## Running times of the Fibonacci algorithms (3)

So all in all we end up with  $T(n) := 1 + 2 + (n - 1) \cdot n \approx n^2$  elementary operations.

# Running times of the Fibonacci algorithms (4)

How many “elementary operations” do we have to perform in **FibRecursive**?

- ▶ Lines 1, 2, 3, 4 each need 1 elementary operation.
- ▶ Line 5 is one elementary operation (the addition), plus the time needed in the recursive calls. We ignore the elementary operation and just consider a lower bound:

We end up with the formula

$$T(n) \geq T(n-1) + T(n-2) + 4$$

Comparing with the definition of  $F_n$  we see immediately that  
 $T(n) \geq F_n$ .

As we have already seen that for  $n \geq 6$  we have  $F_n \geq 2^{n/2}$ , we finally obtain:

$$T(n) \geq 2^{n/2}$$

# Running times of the Fibonacci algorithms (5)

Comparing the running times:

- ▶ The running time of FibRecursive is exponential in  $n$ . This means that even for moderate  $n$  the running time is too large for any computer you might think of.
- ▶ The running time of FibLoop is quadratic in  $n$ . This is ok for reasonable large  $n$  (even though not great). At least it is “polynomial”.
- ▶ Do you think that there exists a faster algorithm than FibLoop? The answer is yes, and you will see this in your exercises. ☺

# Take home message

- ▶ Even for very simple problems, slightly different implementations can lead to huge performance differences!!!
- ▶ The obvious implementations are often quite slow.
- ▶ To obtain fast implementations is often not so obvious.

The main purpose of this lecture is

- ▶ to show you some of the fast, elegant algorithms for standard problems
- ▶ to give you a toolbox to be able to analyze algorithms and figure out whether they are “good”
- ▶ to teach you “algorithmic thinking”: this is what you need to develop new algorithms yourself

# Pseudo-code

# Pseudo-Code

In order to argue about different algorithms, we want to have a language to “write down” an algorithm:

- ▶ compact and informal high-level description of the operating principle of an algorithm
- ▶ It uses the structural conventions of a programming language, but is **intended for human reading** rather than machine reading.

We use what is called “pseudo-code”:

- ▶ use the standard constructions of programming languages (if, for, while, ... )
- ▶ But we do not care about particular syntax details such as semicolons, differences between = and ==, and so on.
- ▶ Often, we are more high-level than a programming language. For example, we might state:

# Pseudo-Code (2)

- ▶ “allocate an array of length  $n$ ”
- ▶ “Sort the entries of the array in increasing order”
- ▶ Many authors use different conventions and syntax for pseudo-code, it is not really important which one we use.

Conventions for the assignments:

- ▶ We are not going to insist on a particular syntax. Use java-like or C-like syntax or whatever you prefer.
- ▶ If you write pseudo-code, use common sense — your code has to be understandable by any “educated programmer” (and, as a matter of fact, by your Übungsleiter ... )
- ▶ In particular, do not use syntax-specific tricks or shortcuts such as:  $a=(b==4)$
- ▶ Always use indentations to show the block structure

# Pseudo-Code (3)

Example: search through an array  $A$  to check whether it contains a particular element  $a_0$

**NaiveSearch( $A, a_0$ )**

```
1  it = 0
2  foundit = false
3  while ( (NOT foundit) AND (it < length( $A$ ))
4      it = it + 1
5      if  $A(it) = a_0$ 
6          foundit = true
7          position = it
8      if foundit
9          Return position
10     else
11         Return NaN
```

# O-Notation

## Literature:

- ▶ Mehlhorn/Sanders Section 2.1
- ▶ Cormen Section I.3

# The growth of functions

Running times of algorithms will depend on the “size of the problem”:

- ▶ It makes a difference whether we have to sort 10 items or  $10^3$  items:

Ultimately, we are not so much interested in the running time on small instances:

- ▶ Even if we are very naive, sorting 10 numbers will not take long.

We want to know how the running times behave if the “size”  $n$  of the input “gets large”.

To this end, we want to have an easy but precise tool to discuss the growth of functions in general.

# The growth of functions (2)

Example:

Assume we have three algorithms to solve a particular problem.  
They perform the following number of elementary operations:

- ▶ Algorithm 1 needs  $17n^3 + 3n^2 - 7$  operations
- ▶ Algorithm 2 needs  $4n^3 + 100n^2$  operations
- ▶ Algorithm 3 needs  $n + 10000$  operations
- ▶ Algorithm 4 needs  $2^n$  operations

We want to solve an instance of the problem of size  $n = 10^5$ .

EVERYBODY: WHICH ONE IS BEST? HOW LARGE ARE THE DIFFERENCES IN RUNNING TIME?

# The growth of functions (3)

Punchline:

We only care for asymptotic growth rates, the “order of magnitude”. In particular,

- ▶ we only want to look at “the largest term” in the expressions above.
- ▶ we want to ignore all constants

Now we want a mathematically precise formalism for doing this:  
~ O-notation.

# O-Notation: Definitions

To analyze the running times, we are going to use the  $O$ -Notation (sometimes called Landau notation). This is a tool to describe the asymptotic growth of functions.

In this section we always assume that all functions are non-negative.

$f \in O(g)$   $f$  is of order at most  $g$ :

$$\exists c > 0 \exists n_0 > 0 \forall n > n_0 : 0 < f(n) \leq cg(n)$$
$$\iff 0 \leq \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

$f \in o(g)$   $f$  is of order strictly smaller than  $g$ :

$$\forall c > 0 \exists n_0 > 0 \forall n > n_0 : 0 < f(n) < cg(n)$$
$$\iff 0 \leq \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

# O-Notation: Definitions (2)

$f \in \Omega(g)$   $f$  is of order at least  $g$ :

$$\exists c > 0 \exists n_0 > 0 \forall n > n_0 : f(n) \geq cg(n) > 0$$

$$\iff 0 < \liminf_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq \infty$$

$$\iff g \in O(f)$$

$f \in \omega(g)$   $f$  is of order strictly larger than  $g$ :

$$\forall c > 0 \exists n_0 > 0 \forall n > n_0 : f(n) > cg(n) > 0$$

$$\iff \liminf_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

$$\iff g \in o(f)$$

$f \in \Theta(g)$   $f$  has exactly the same order as  $g$ :

$$\exists c_1, c_2 > 0 \exists n_0 > 0 \forall n > n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n)$$

$$\iff 0 < \liminf_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

$$\iff f \in O(g) \text{ and } f \in \Omega(g)$$

# O-Notation: Examples

- ▶ Consider  $f(n) = n^2 + 7$ . Then:  $f \in O(n^{10})$ ;  $f \in o(n^{10})$ ;  $f \in \omega(\log(n))$ ;  $f \in \Theta(n^2)$ ;
- ▶ Consider  $f(n) = n^2 + 5n$ . Then  $f \in O(n^2)$ , but  $f \notin o(n^2)$ . Instead  $f \in \Omega(n^2)$ , but  $f \notin \omega(n^2)$ . In particular,  $f \in \Theta(n^2)$ .

# O-Notation: Examples (2)

Important special cases:

- ▶ If  $f$  is any polynomial of degree  $d$ , then  $f \in \Theta(n^d)$ .
- ▶ Any polynomial function  $f$  is in  $o(\exp(n))$  and  $\omega(\log(n))$ .
- ▶ Observe: if we use logarithms, it does not matter what basis of the logarithm we choose:  $\log_{b_1}(n) \in \Theta(\log_{b_2}(n))$ .

Reason:

$$\log_b x = \frac{\log_a x}{\log_a b}$$

- ▶ Any function that is bounded between constants  $a > 0$  and  $b > 0$ , where  $a$  and  $b$  are independent of  $n$ , is said to be  $O(1)$ .

# O-Notation: Examples (3)

Some rules:

- ▶  $f \in O(g_1 + g_2)$  and  $g_1 \in O(g_2) \implies f \in O(g_2)$ .
- ▶  $f_1 \in O(g_1)$  and  $f_2 \in O(g_2) \implies f_1 + f_2 \in O(g_1 + g_2)$
- ▶  $f \in g_1 \cdot O(g_2) \implies f \in O(g_1 g_2)$
- ▶ Note: for any constant  $c > 0$ ,  $O(cg)$  is identical to  $O(g)$ .
- ▶  $f \in O(g_1)$ ,  $g_1 \in O(g_2) \implies f \in O(g_2)$
- ▶ Abuse of notation: we often write  $f = O(g)$ , but we mean  $f \in O(g)$ .

# Running time analysis

Literature:

- ▶ Mehlhorn/Sanders Sec. 2

# Physical running time

We could simply go ahead, run an algorithm on our computer, and measure its running time.

IS THIS A GOOD IDEA?

# Physical running time

We could simply go ahead, run an algorithm on our computer, and measure its running time.

IS THIS A GOOD IDEA?

NO, physical running time is not a good model:

- ▶ depends on the computer we are using
- ▶ depends on the particular input we are using

Instead we are going to do the following:

- ▶ Introduce a simplified “model” of a computer
- ▶ Count the number of “elementary operations” of this computer

# Simplified machine model: RAM machine

Random access machine (RAM) model:

- ▶ Sequential processing
- ▶ Uniform memory (all access to memory takes the same amount of time)

The machine can perform the following actions as “elementary operations” in one unit of time:

- ▶ Load one bit from the memory
- ▶ Write one bit of memory
- ▶ Elementary arithmetics: for numbers of bounded length, we assume that we can add, subtract, multiply, divide them in constant time
- ▶ Similarly, each logical operation (and, or, not) is an elementary operation.

## Simplified machine model: RAM machine (2)

Some care is needed to deal with arithmetic operations of “long numbers”:

- ▶ As long as numbers are long enough to fit into, say, 64 bits, we treat arithmetic operations as constant time operations.
- ▶ However, in some algorithms the numbers grow as the instances get larger (this was the case in FibLoop). Then we can no longer treat the arithmetic operations as elementary operation.
- ▶ In this case, we need to consider the actual costs of doing such an operation.

WHAT DO YOU THINK IS THE COST OF ADDING TWO NUMBERS OF  $n$  BITS?

WHAT ABOUT MULTIPLYING THEM?

# Simplified machine model: RAM machine (3)

Common sense tells us:

- ▶ The cost to add two numbers depends on the numbers of bits we have to touch. To add two numbers of  $u$  and  $v$  bits is going to be in  $O(\max(u, v))$ , that is it is linear in the number of bits.
- ▶ For (naive) multiplication, the cost is quadratic in the number of bits:

# Simplified machine model: RAM machine (4)

Basic Integer Multiplication:

$$\begin{array}{r} x_1 \ x_2 \ x_3 \\ 3 \ 7 \ 5 \\ \cdot \ 2 \ 4 \ 6 \\ \hline \end{array}$$

$$\begin{array}{rl} y_1 \cdot x_3 & 10 \\ y_1 \cdot x_2 & 14 \\ y_1 \cdot x_1 & 6 \\ y_2 \cdot x_3 & 20 \\ y_2 \cdot x_2 & 28 \\ y_2 \cdot x_1 & 12 \\ y_3 \cdot x_3 & 30 \\ y_3 \cdot x_2 & 42 \\ y_3 \cdot x_1 & 18 \\ \hline & 92250 \end{array}$$

See later for a formal derivation of these results.

# Counting RAM elementary operations, examples

How many elementary operations does it take to do the following operations:

- ▶ add two standard integers:

# Counting RAM elementary operations, examples

How many elementary operations does it take to do the following operations:

- ▶ add two standard integers:  $O(1)$
- ▶ Write an  $m \times n$  matrix to memory:

# Counting RAM elementary operations, examples

How many elementary operations does it take to do the following operations:

- ▶ add two standard integers:  $O(1)$
- ▶ Write an  $m \times n$  matrix to memory:  $O(n \times m)$
- ▶ Read  $n$  strings of length 10:

# Counting RAM elementary operations, examples

How many elementary operations does it take to do the following operations:

- ▶ add two standard integers:  $O(1)$
- ▶ Write an  $m \times n$  matrix to memory:  $O(n \times m)$
- ▶ Read  $n$  strings of length 10:  $O(n)$
- ▶ Perform the operation  $a_1 + a_2 \cdot a_3$  if we know that  $a_1$  has  $n^2$  bits,  $a_2$  has  $\log n$  bits,  $a_3$  has  $n$  bits:

# Counting RAM elementary operations, examples

How many elementary operations does it take to do the following operations:

- ▶ add two standard integers:  $O(1)$
- ▶ Write an  $m \times n$  matrix to memory:  $O(n \times m)$
- ▶ Read  $n$  strings of length 10:  $O(n)$
- ▶ Perform the operation  $a_1 + a_2 \cdot a_3$  if we know that  $a_1$  has  $n^2$  bits,  $a_2$  has  $\log n$  bits,  $a_3$  has  $n$  bits:
  - ▶ Computing  $a_2 \cdot a_3$  takes  $O(n \cdot \log n)$  time.
  - ▶ The resulting number is going to have of the order  $O(n \cdot \log n)$  bits.
  - ▶ To add a number with  $n^2$  bits to a number of  $n \log n$  bits is going to cost  $O(n^2)$  time.

So overall, this operation takes  $O(n^2)$ .

# How to measure the “length of an input”

We want to compute the **running time as a function of the size  $n$  of the input**.

- ▶ Sorting  $n$  standard integers  $\leadsto$  input size is  $n$
- ▶ Comparing two strings of variable lengths  $n_1$  and  $n_2$   $\leadsto$  input size is the sum of the lengths of the two strings. Because we don't care about constants, this is of the same order as the length of the longer of the two strings.
- ▶ Compare  $m$  strings with length at most  $n$   $\leadsto$  input size  $m \cdot n$
- ▶ Finding a shortest path in a graph  $\leadsto$  input size is the number of vertices and edges in the graph

If this sounds a bit confusing, don't worry, this is going to become clear during the lecture.

# What instance to look at?

Depending on the instance we look at, an algorithm might take long or not so long:

- ▶ If we want to test whether two sequences are identical, and they already disagree at the first element we are looking at, then we are done. But if they disagree only in the last element, this will take us much longer to figure out.
- ▶ If our task is to sort a sequence, and the sequence is already sorted, we are done already.
- ▶ To compute the shortest path in a tree is much simpler than computing the shortest path in a general graph.

But we want a statement about the running time that is reasonably general.

ANY IDEA WHAT WE COULD DO?

# Worst case running time: definition

Among all possible instances of size  $n$ , what is the longest time it might take the algorithm to finish?

Formally:

- ▶ Let  $\mathcal{I}_n$  denote the space of all instances of length  $n$
- ▶ Denote by  $T(I_n)$  the running time of the algorithm on instance  $I_n$
- ▶ Then the worst case running time for input of size  $n$  is defined as

$$T_{wc}(n) := \max\{ T(I_n) \mid I_n \in \mathcal{I}_n \}$$

# Worst case running time: definition (2)

Example  $\text{NaiveSearch}(A, a_0)$ :

- ▶  $n :=$  number of elements in the array
- ▶ Initialization = 2 units of time
- ▶ In the worst case we have to execute  $n$  while loops
- ▶ Inside the while loop, we have a constant number of operations [if we count exactly, line 3 is 3 operations, line 4 is 2 operations (one addition, one access to storage), lines 6 and 7 one operation each].
- ▶ after we finish the while loop, we have 2 more operations
- ▶ So the overall worst case running time is:  
$$2 + n \cdot 7 + 2 \in O(n).$$

## Worst case running time: definition (3)

ASSUME YOU WANT TO GET A LOWER / UPPER BOUND ON THE WORST CASE RUNNING TIME. WHAT DO YOU HAVE TO DO?

## Worst case running time: definition (4)

- ▶ Lower bounds are easy: If you find **one particular instance**  $I_n$  on which the running time is at least  $T_{lower}$ , then you can conclude that  $T_{wc}(n) \geq T_{lower}$ .
- ▶ Upper bounds: If you know that for **all instances**  $I_n \in \mathcal{I}_n$ , the running time is at most  $T_{upper}$ , then you can conclude that  $T_{wc}(n) \leq T_{upper}$ .

# Average case running time

Observation:

- ▶ While worst case running times always give an upper bound, they are often much too conservative:
- ▶ Some algorithms behave very well in practice even though they have a bad worst case running time.
- ▶ The reason is that the “normal” instances we have to solve in every day life are often “easy”, whereas the worst case running time only occurs for very un-natural, hand-constructed instances.

To circumvent the problem we introduce the average case running time:

## Average case running time (2)

On average over all instances of size  $n$ , what is the time it takes algorithm  $A$  to finish?

Formally:

- ▶ Let  $\mathcal{I}_n$  denote the space of all instances of length  $n$
- ▶ Denote by  $T(I_n)$  the running time of the algorithm on instance  $I_n$
- ▶ Then the average case running time for input of size  $n$  is defined as

$$T_{av}(n) := \frac{1}{|\mathcal{I}_n|} \sum_{I_n \in \mathcal{I}_n} T(I_n)$$

## Average case running time (3)

There are some subtleties about the fact that the average time analysis heavily depends on “how often” each instance is supposed to occur! We skip details, because this lecture is mainly concerned with worst case analysis.

# Special cases

We say that the running time of an algorithm is ...

- ▶ linear  $\sim \Theta(n)$
- ▶ sublinear  $\sim o(n)$
- ▶ superlinear  $\sim \omega(n)$
- ▶ polynomial  $\sim \Theta(n^a)$  for some constant  $a$  independent of  $n$
- ▶ exponential  $\Theta(2^n)$

# Space complexity analysis

# Space complexity

- ▶ We also need to know how much storage capacity an algorithm uses.
- ▶ One model “unit” of storage for:
  - ▶ A letter
  - ▶ A logical bit
  - ▶ A number of fixed, bounded length
- ▶ We usually look at worst case behavior.

For many problems, there is a space-time tradeoff:

- ▶ Some algorithms are very fast, but need a lot of space.
- ▶ Other algorithms are very slow, but need nearly no storage.

Depending on the application, one or the other case might be better.

DOES ANYBODY KNOW AN EXAMPLE OF SUCH A BEHAVIOR?

# Appetizer: Multiplying two integers

Literature:

- ▶ Dasgupta 2.1
- ▶ Mehlhorn 1.3-1.5

# Problem statement

We consider the following, very simple problem:

Given two integers  $x$  and  $y$ , compute their product  $x \cdot y$ .

For simplicity, let us assume that both  $x$  and  $y$  can be represented as binary numbers of length  $n$ , and that  $n$  is a power of 2.

# Problem statement (2)

Naive method (as in school) takes  $O(n^2)$ :

Basic integer Multiplication:

$$\begin{array}{r} x_1 \ x_2 \ x_3 \\ 3 \ 7 \ 5 \\ \cdot \ 2 \ 4 \ 6 \\ \hline \end{array}$$

$$\begin{array}{rl} y_1 \cdot x_3 & 1 \ 0 \\ y_1 \cdot x_2 & 1 \ 4 \\ y_1 \cdot x_1 & 6 \\ y_2 \cdot x_3 & 2 \ 0 \\ y_2 \cdot x_2 & 2 \ 8 \\ y_2 \cdot x_1 & 1 \ 2 \\ y_3 \cdot x_3 & 3 \ 0 \\ y_3 \cdot x_2 & 4 \ 2 \\ y_3 \cdot x_1 & 1 \ 8 \\ \hline & 9 \ 2 \ 2 \ 5 \ 0 \end{array}$$

# First divide and conquer solution

Idea: try a recursive algorithm ( $\leadsto$  divide and conquer):

- ▶ Split  $x$  and  $y$  in their left and right parts. We have:

$$x = 2^{n/2}x_l + x_r$$

$$y = 2^{n/2}y_l + y_r$$

$$\begin{aligned} x &= \boxed{\textcolor{red}{10110011}} \textcolor{green}{\boxed{00101111}} \\ y &= \boxed{\textcolor{red}{11101011}} \textcolor{green}{\boxed{00011110}} \\ &\quad \textcolor{red}{x_l} \qquad \qquad \textcolor{green}{x_r} \\ &\quad \textcolor{red}{y_l} \qquad \qquad \textcolor{green}{y_r} \end{aligned}$$

- ▶ So we can write:

$$x \cdot y = 2^n x_l y_l + 2^{n/2}(x_l y_r + x_r y_l) + x_r y_r$$

We now have **4 subproblems**: multiply integers of length  $n/2$

# First divide and conquer solution (2)

## Running time:

- ▶ Write  $T(n)$  for the running time on integers of length  $n$
- ▶ The recursive call leads to the following recurrence relation:

$$T(n) = 4T(n/2) + O(n)$$

- ▶ By the master theorem (see later), the solution to this equation is  $T(n) = O(n^2)$ .

So the recursive approach did not improve the running time ☺

# Better divide and conquer solution

Simple, but powerful observation:

- We have:

$$(x_l y_r + x_r y_l) = (x_l + x_r)(y_l + y_r) - x_l y_l - x_r y_r$$

- This gives:

$$\begin{aligned}x \cdot y &= 2^n x_l y_l + 2^{n/2} (x_l y_r + x_r y_l) + x_r y_r \\&= 2^n x_l y_l + 2^{n/2} ((x_l + x_r)(y_l + y_r) - x_l y_l - x_r y_r) + x_r y_r \\&= 2^{n/2} \textcolor{red}{x_l y_l} + (1 - 2^{n/2}) \textcolor{green}{x_r y_r} + 2^{n/2} (\textcolor{blue}{x_l + x_r})(\textcolor{blue}{y_l + y_r})\end{aligned}$$

Thus, instead of computing 4 multiplications of  $n/2$  bits each, we only need 3 of them!!!

## Better divide and conquer solution (2)

The resulting algorithm satisfies the recurrence relation

$$T(n) = 3T(n/2) + O(n)$$

which has the solution  $T(n) = O(n^{\log_2 3}) \approx O(n^{1.59})$  due to the master theorem.

Thus we found an algorithm that is substantially faster than the naive approach ☺

## Tool: Master theorem for solving simple recurrence relations

Literature:

Dasgupta 2.2; Cormen 4.5

# Divide and conquer and recurrence relations

We are going to use the **divide and conquer principle**:

- ▶ Given a problem of size  $n$
- ▶ Decompose it into several problems of smaller size and solve these problems
- ▶ From the solutions, reconstruct the overall solution

To **analyze the running time**, we will always follow the same principle.

- ▶ Denote by  $T(n)$  the running time of the algorithm on an instance of size  $n$ .
- ▶ Let  $a$  be the number of subproblems,  $m = n/b$  the size of the subproblems, and assume we need  $O(n^d)$  time to combine the smaller solutions to the final one.

# Divide and conquer and recurrence relations (2)

- ▶ Then the running time satisfies the recurrence relation

$$T(n) = aT(\lceil n/b \rceil) + O(n^d)$$

# The master theorem for solving recurrences

## Theorem 1 (Master theorem)

If  $T(n) = aT(\lceil n/b \rceil) + O(n^d)$  for some constants  $a > 0$ ,  $b > 1$ , and  $d \geq 0$ , then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

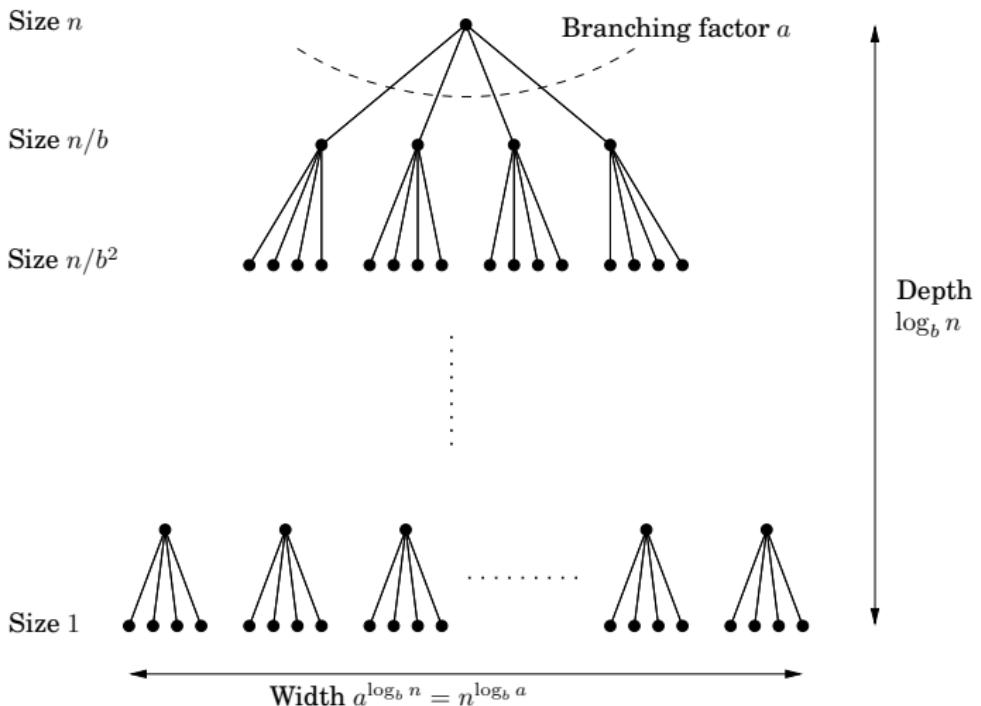
In this context, recall how to change the base of a logarithm:

$$\begin{aligned} \log_b x &= \log_b (a^{\log_a x}) = \log_a x \log_b(a) \\ \implies \log_a x &= \frac{\log_b x}{\log_b a} \end{aligned}$$

# The master theorem for solving recurrences (2)

**Proof.** The proof of the Master theorem is surprisingly simple. It boils down to analyze the following recursion tree:

# The master theorem for solving recurrences (3)



# The master theorem for solving recurrences (4)

- ▶ **Depth of the tree:** at each level, the problem gets smaller by a factor of  $1/b$ . Thus we reach the bottom of the tree when  $n/b^{\text{depth}} = 1$ , that is the depth =  $\log_b n$ .
- ▶ **Level  $k$  of the tree, for all intermediate values of  $k$ :**
  - ▶ Level  $k$  of the tree is made up of  $a^k$  subproblems
  - ▶ Each of these subproblems has size  $n_k := n/b^k$
  - ▶ Each of these problems is solved by calling further subproblems (no work required in level  $k$ ) and by then recombining these solutions (this is the work done in level  $k$ ). For each of these problems, this recombination takes time  $O((n_k)^d) = O((n/b^k)^d)$ .
  - ▶ So the total work to be done at level  $k$  is

$$a^k \cdot O\left(\left(\frac{n}{b^k}\right)^d\right) = O(n^d) \cdot \left(\frac{a}{b^d}\right)^k$$

# The master theorem for solving recurrences (5)

- **Bottom level of the tree:** The bottom level consists of  $a^{\text{depth}} = a^{\log_b n} = n^{\log_b a}$  problems of constant size (say, size 1). Solving each of these problems requires  $O(1)$  time, so the time required by the algorithm in the bottom level of the tree is  $O(n^{\log_b a})$ .

Altogether, the time to go work through the whole tree is

$$\begin{aligned}& \left( \sum_{k=0}^{\text{depth}-1} O(n^d) \cdot \left(\frac{a}{b^d}\right)^k \right) + O(n^{\log_b a}) \\&= O(n^d) \sum_{k=0}^{\log_b n - 1} \left(\frac{a}{b^d}\right)^k + O(n^{\log_b a})\end{aligned}$$

The rest of the proof just evaluates this expression:

# The master theorem for solving recurrences (6)

First consider the red sum.

- ▶ It is a geometric series of the form  $\sum_{k=0}^m c^k$ .
- ▶ It is straight forward to see that

$$O\left(\sum_{k=0}^m c^k\right) = \begin{cases} O(1) & \text{if } c < 1 \text{ (because sum} < 1/(1 - c)\text{)} \\ O(m) & \text{if } c = 1 \text{ (because then sum} = m\text{)} \\ O(c^m) & \text{if } c > 1 \text{ (proof by simple induction)} \end{cases}$$

In our setting, this now translates into the following three cases:

# The master theorem for solving recurrences (7)

- ▶ **Case 1.**  $c < 1 \iff a/b^d < 1 \iff d > \log_b a$ .

Then we are left with

$$O(n^d) \cdot O(1) + O(\underbrace{n^{\log_b a}}_{< n^d}) = O(n^d)$$

- ▶ **Case 2.**  $c = 1 \iff a/b^d = 1 \iff d = \log_b a$ .

We are left with

$$O(n^d) \cdot O(\log_b n) + O(\underbrace{n^{\log_b a}}_{= n^d}) = O(n^d \log n)$$

# The master theorem for solving recurrences (8)

- **Case 3.**  $c > 1 \iff a/b^d > 1 \iff d < \log_b a$ .

We are left with

$$O(n^d) \cdot O\left(\underbrace{\left(\frac{a}{b^d}\right)^{\log_b n}}_{(*)}\right) + O(n^{\log_b a})$$

$$(*) = \frac{a^{\log_b n}}{(b^{\log_b n})^d} = \frac{a^{\log_b n}}{n^d} = \frac{a^{(\log_a n)(\log_b a)}}{n^d} = \frac{n^{\log_b a}}{n^d}$$

Combining the terms leads to

$$n^d \cdot \frac{n^{\log_b a}}{n^d} + n^{\log_b a} = O(n^{\log_b a})$$

# Examples

EXAMPLE ( $\leadsto$  used for the analysis of integer multiplication):

$$T(n) = 4T(n/2) + O(n)$$

## Examples (2)

SOLUTION:

- ▶ Have  $a = 4$ ,  $b = 2$ ,  $d = 1$ ,
- ▶ In particular,  $\log_b a = \log_2 4 = 2 > 1 = d$ .
- ▶ Thus, by the third case of the master theorem we get

$$T_n = O(n^{\log_b a}) = O(n^2)$$

# Examples (3)

## EXAMPLE:

Binary search. Given a sorted array and a value  $v$ , we want to identify the element of the array that has value closest to  $v$ . What is the recurrence equation for the running time of this algorithm, and what is the solution according to the master theorem?

## Examples (4)

SOLUTION:

- ▶ Recurrence equation is  $T(n) = T(n/2) + O(1)$ .
- ▶ In the notation of the Master theorem we have  $a = 1$ ,  $b = 2$ ,  $d = 0$ . In particular  $\log_b a = \log_2 1 = 0 = d$ .
- ▶ So we are in the second case of the theorem. The running time is  $O(n^d \log n) = O(n^0 \log n) = O(\log n)$ .

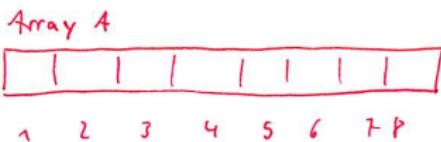
# Basic Data Structures

# Arrays and lists

# Array

Everybody knows arrays already:

- ▶ An array of length  $n$  is an arrangement of  $n$  objects of the same type in equally spaced addresses in memory.
- ▶ This is the most basic data structure one can think of.



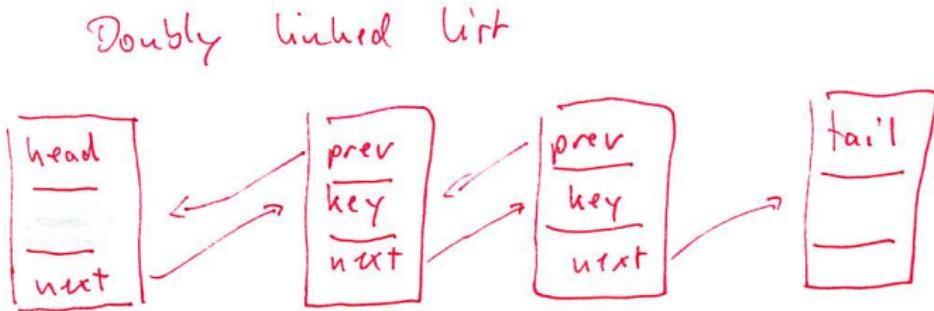
Standard operations on an array:

- ▶ Read / write an element of the array. Happens in constant time, no matter which element we want to access (reason: we know their address in memory).

Disadvantage of arrays: we need to allocate them in advance.

# Doubly linked list

- ▶ Each list element consists of a key value, a “previous” pointer and a “next” pointer
- ▶ It contains one special element, the so called “sentinel” (Wächter) to mark the beginning / end of the list



# Doubly linked list (2)

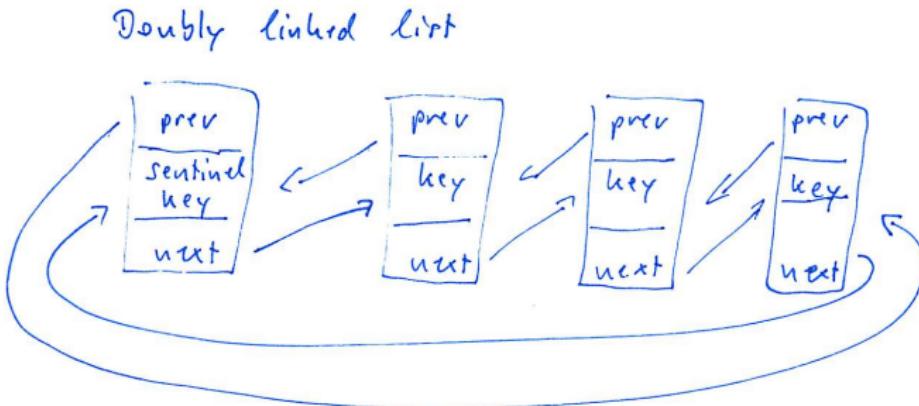
Basic operations:

- ▶ Insert element  $x$  at after element  $e$  in the list, possible in  $O(1)$
- ▶ Delete an element from the list, possible in  $O(1)$
- ▶ Search element with a particular key value  $k$ , needs  $O(n)$
- ▶ Delete a whole sublist,  $O(1)$  as well
- ▶ Insert an existing second list after a particular element in the first list, also possible in  $O(1)$  (independent of the lengths of the lists!)

# Doubly linked list (3)

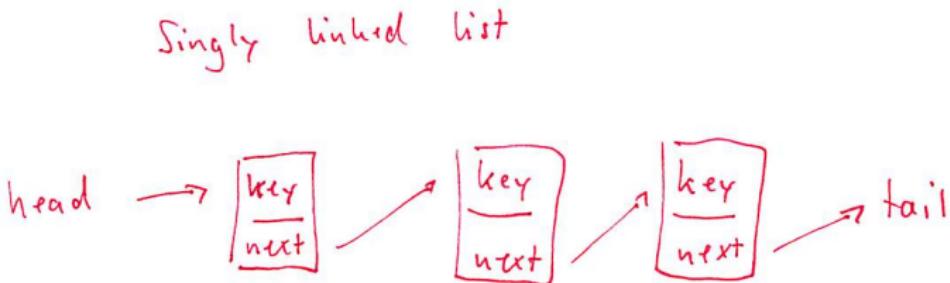
Two different ways to implement them:

- ▶ Using a “head” and “tail” element, as in the figure above
- ▶ Link tail to head, but insert a “sentinel” element (German: Wächter)



# Singly linked list

As before, but each element only contains a pointer to the next element, not to the previous one.



WHAT ARE THE MAIN DIFFERENCES BETWEEN DOUBLY AND SINGLY LINKED LISTS?

## Singly linked list (2)

- ▶ Singly linked lists need less storage (but just by a factor of 2)
- ▶ But we cannot delete arbitrary elements in constant time because we first have to find the previous element in the list

# Linked lists vs. array

Linked lists:

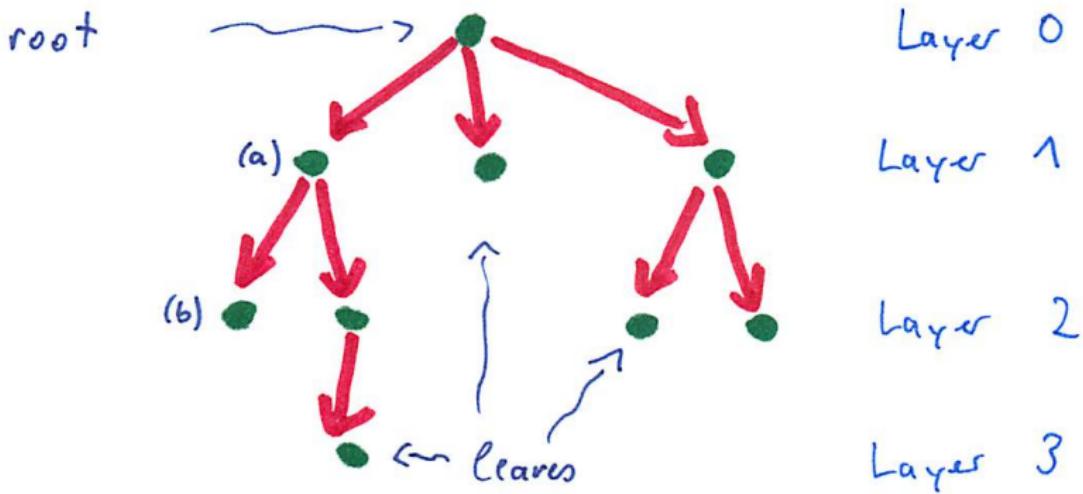
- ☺ we don't need to worry about allocation (as it would be the case in an array)
- ☹ But to access a particular element we need to walk through the list, this is costly if it has to happen very often.

Array:

- ☺ Fast access to all its elements
- ☹ Allocation is problematic if we don't know in advance how large the array is going to be.

# Trees

# General definitions



- (a) is the parent/predecessor of (b)
- (b) is a child of (a)

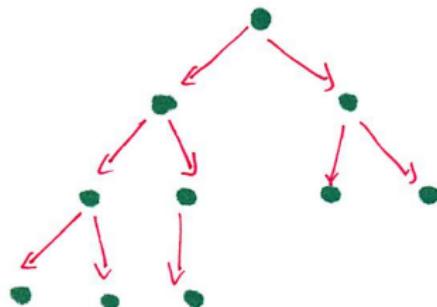
## General definitions (2)

Height of a vertex: length of the shortest path from the vertex to the root

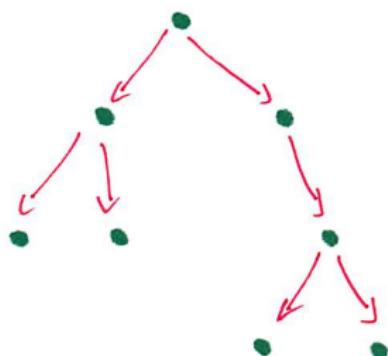
Height of the tree: maximum vertex height in the tree

# Binary tree

- ▶ In a **binary tree**, each vertex has at most 2 children.
- ▶ In a **complete binary tree**, all layers except the last one are filled. In the last layer, vertices are “as left as possible”.
- ▶ A **full binary tree** is a complete binary tree in which the last level is filled completely.



Complete binary tree



A binary tree

## Binary tree (2)

What is the height of a complete binary tree of  $n$  elements?

### Proposition 2 (Height of binary tree)

A full binary tree with  $n$  vertices has height

$\log_2(n + 1) - 1 \in \Theta(\log n)$ . A complete binary tree height  
 $\lceil \log_2(n + 1) - 1 \rceil \in \Theta(\log n)$ .

Proof:

## Binary tree (3)

- ▶ Number  $N$  of vertices in a full binary tree of height  $h$ :

$$N = 1 + 2 + 4 + 8 + 16 + \dots + 2^h$$

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

Here we used the formula for the geometric series: For  $a \neq 1$ ,

$$\sum_{i=0}^h a^i = \frac{1}{a-1} \sum_{i=0}^h (a-1)a^i = \frac{1}{a-1}(a^{h+1} - 1)$$

- ▶ Now solve the formula for  $N$ .

$$n = 2^{h+1} - 1 \iff h = \log_2(n + 1) - 1$$

This gives the statement for a full binary tree.

## Binary tree (4)

- ▶ In case of a complete binary tree, the last layer might not be filled. Then we get:

$$h = \lceil \log_2(n + 1) - 1 \rceil$$

# Representation of trees

To implement a binary tree:

- ▶ Each vertex contains the key value and pointers to left and right child vertices, and a pointer to its parent vertex

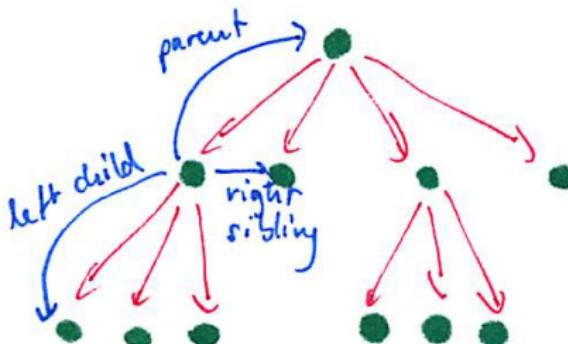
A similar scheme works for trees for which we know an upper bound for the number of children of each vertex.

## Representation of trees (2)

If the number of children per vertex is not known in advance, we use linked lists instead of arrays to store the siblings of a vertex.

A particular clever way to do this is the following:

- ▶ Each vertex has three pointers:
  - ▶ A pointer to its parent
  - ▶ A pointer to its leftmost child
  - ▶ A pointer to its right sibling



# Stack and queue

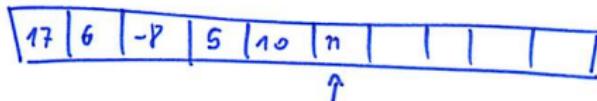
# Stack (German: Stapel)

- ▶ Dynamic structure to store a set of elements
- ▶ Analogy: Stack of books to read
- ▶ **Push(x)** (also called **Insert**) inserts the new element  $x$  to the stack
- ▶ **Pop** (also called **Delete**) removes the next element from the stack (LIFO: Last in first out)

Implementation: many possible ways.

- ▶ For example with an array and a pointer that points to the current element.
- ▶ Then Push and Pop take time  $O(1)$ .

Array:



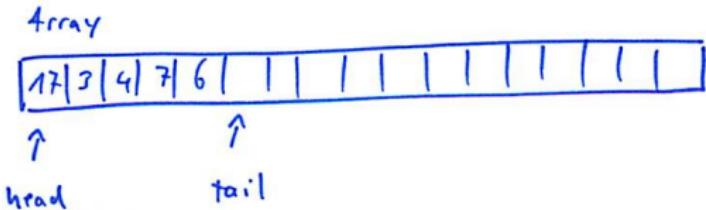
pointer to the current position

# Queue (German: Warteschlange)

- ▶ Dynamic structure to store a set of elements
- ▶ Analogy: line of customers
- ▶ **Insert(x)** (also called **Enqueue**) inserts the new element  $x$  to the end of the queue
- ▶ **Delete** (also called **Dequeue**, **Pop**) removes the next element from the queue (FIFO: First in first out)

Implementation: many possible ways.

- ▶ For example, array with two pointers, one to the head and one to the tail.
- ▶ Then both Insert and Delete take time  $O(1)$ .



# Heaps and priority queues

Literature:

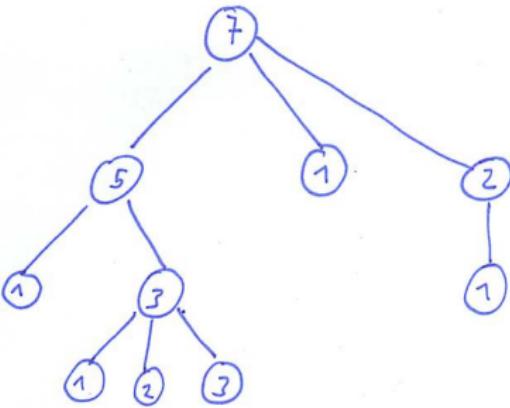
- ▶ Cormen Sec. 6
- ▶ Mehlhorn Sec. 6

# Heaps

# Heaps

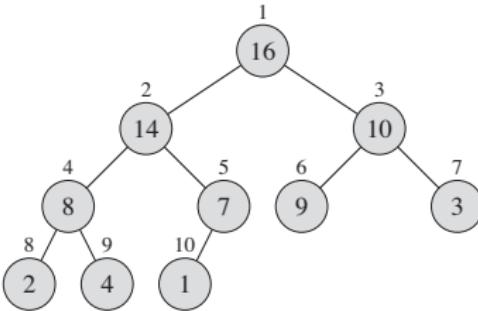
Heap in general:

- ▶ Data structure that stores elements as vertices in a tree
- ▶ Each element has a key value assigned to it
- ▶ Max-heap property: all vertices in the tree satisfy  
 $\text{key}(\text{parent}(v)) \geq \text{key}(v)$



# Heaps (2)

- ▶ Binary heap:
  - ▶ each vertex has at most two children.
  - ▶ Vertices are filled up such that we first complete a layer before we start a new one; when we do so, we fill vertices “from left to right”



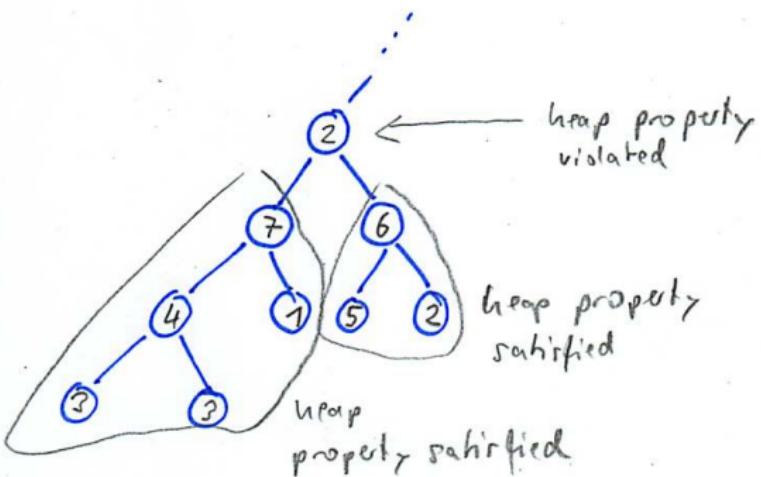
# Heaps (3)

Advantage of “binary”:

- ▶ control over height and width of the tree
- ▶ we can easily store the heap in an array without any additional pointers

# The most important operation: heapify

- ▶ Consider vertex  $i$  in the binary tree
- ▶ Assume that both of its subtrees satisfy the heap property, but the heap property is violated at  $i$  itself:  $\text{key}(i) < \text{key}(\text{child}(i))$  for at least one child of  $i$
- ▶ We now want to “repair” the heap.

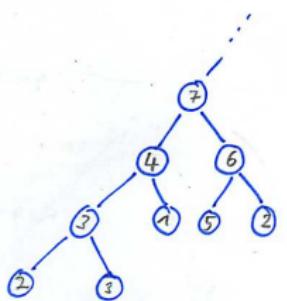
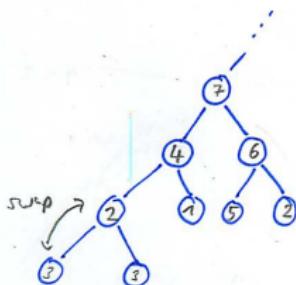
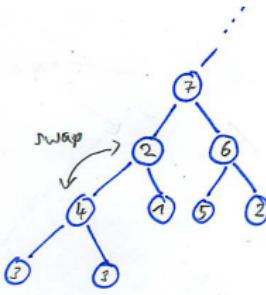
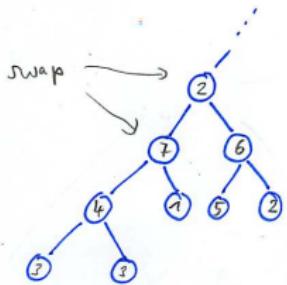


# The most important operation: heapify (2)

Heapify procedure: “Let  $\text{key}(i)$  float down”

- ▶ Swap  $i$  with the larger of its children.
- ▶ Then recursively call heapify on this child.
- ▶ Stop when the heap condition is no longer violated.

# The most important operation: heapify (3)



# The most important operation: heapify (4)

Formally:

```
MAX-HEAPIFY( $A, i$ )
1    $l = \text{LEFT}(i)$ 
2    $r = \text{RIGHT}(i)$ 
3   if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4        $largest = l$ 
5   else  $largest = i$ 
6   if  $r \leq A.\text{heap-size}$  and  $A[r] > A[largest]$ 
7        $largest = r$ 
8   if  $largest \neq i$ 
9       exchange  $A[i]$  with  $A[largest]$ 
10      MAX-HEAPIFY( $A, largest$ )
```

# The most important operation: heapify (5)

Worst case running time of heapify:

- ▶ Assume the binary heap contains  $n$  vertices.
- ▶ The number of swapping operations is at most the height of the tree.
- ▶ Each swapping operation can be done in constant time.
- ▶ Then we know that its height is at most  $h = \lceil \log(n) \rceil = O(\log n)$ .
- ▶ So overall, the worst case running time is  $O(\log n)$ .

# Operation BuildMaxHeap

**Given** an unsorted array  $A$  of  $n$  elements, make a heap out of them.

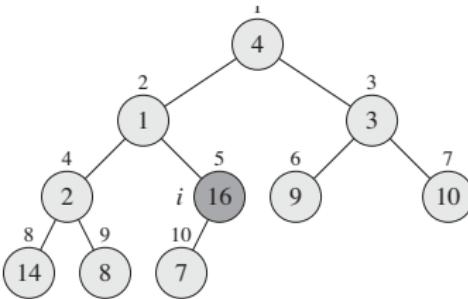
**Solution:** Starting at the bottom level, go through the vertices in each level and call heapify on each vertex. Do this “all the way up” to the root.

- 1 `# A is an unsorted array of  $n$  elements, filled in a binary tree`
- 2 `for  $i = n$  to 1 (in inverse order!)`
- 3   `MaxHeapify(A,i)`

(Note: we could be a bit cleverer by observing that elements  $\lfloor n/2 \rfloor + 1$  to  $n$  are definitely leafs, so we could start the for-loop at  $\lfloor n/2 \rfloor$  instead of  $n$ )

# Operation BuildMaxHeap (2)

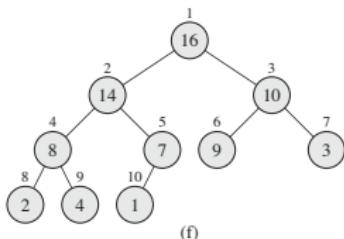
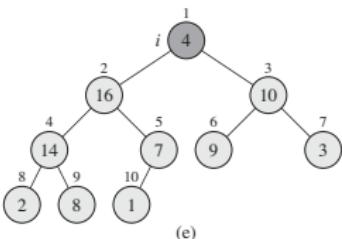
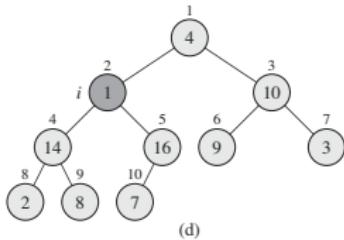
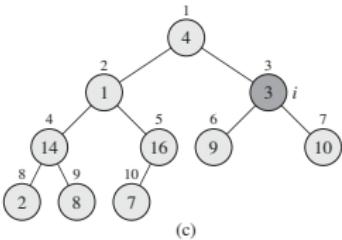
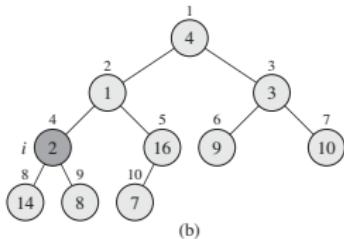
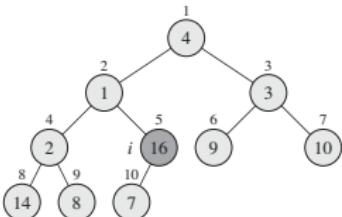
EVERYBODY: TRANSFORM THE FOLLOWING TREE IN A HEAP USING BUILDHEAP:



# Operation BuildMaxHeap (3)

Solution:

# Operation BuildMaxHeap (4)



# Operation BuildMaxHeap (5)

## Running time:

- ▶ Simple upper bound: we call heapify on all  $n$  vertices. Each heapify operation takes  $O(\log n)$ , so overall we get  $O(n \log n)$ .
  - ▶ More clever analysis: Observe that a heapify operation on a vertex of height  $h(v)$  has running time  $O(h(v))$ , and the height is much smaller than  $\log n$  for many vertices.
    - ▶ Denote by  $H$  the height of the tree.
    - ▶ To heapify a single vertex in level  $h(v)$  we have to make at most  $H - h(v)$  swap operations (bubble all the way down)
    - ▶ There are  $2^h$  vertices in level  $h$  of the tree
    - ▶ So the running time of BuildMaxHeap on  $n$  vertices is given as  $T(n) = \sum_{h=0}^H 2^h (H - h)$ .
- We can compute this sum as follows:

# Operation BuildMaxHeap (6)

$$\textcircled{1} \quad T(n) = \sum_{h=0}^H 2^h (H-h) \quad | \cdot 2$$

$$\textcircled{2} \quad 2 \cdot T(n) = \sum_{h=0}^H 2^{h+1} (H-h)$$

$$\begin{aligned} \textcircled{2} - \textcircled{1} : \quad T(n) &= 2 \cdot T(n) - T(n) = \\ &= (2^1(H-0) + 2^2(H-1) + \dots + 2^H(H-(H-1))) \\ &\quad - (2^0(H-0) + 2^1(H-1) + 2^2(H-2) + \dots + 2^{H-1}(H-1) + 2^H(H-H)) \\ &\quad = 2^1 = 2^2 \\ &\quad = 2^H \end{aligned}$$

$$\begin{aligned} &= \underbrace{2^1 + 2^2 + \dots + 2^H}_{2^{H+1}-1} - H \\ &= 2^{H+1}-1 - H \end{aligned}$$

Substituting  $H = \log(n)$  gives  $O(2^{\log n+1} - 1 - \log n) = O(n)$ .

# Operation ExtractMax

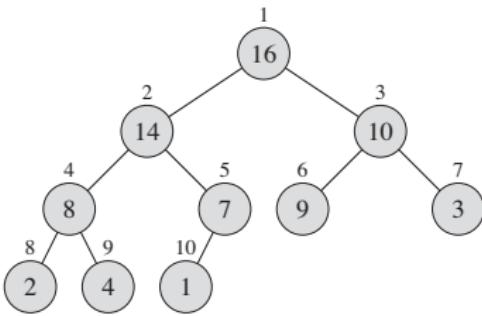
Goal: remove the largest element from the heap.

Solution:

- ▶ By construction, the largest element sits in the root.
- ▶ So we first extract the root element.
- ▶ Next, we replace the root element by the last leaf in the tree and remove that leaf
- ▶ Call  $\text{heapify}(\text{root})$

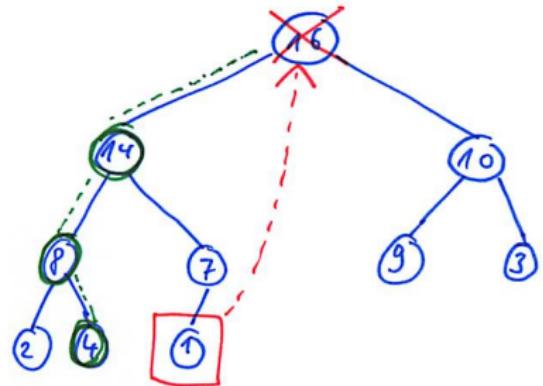
# Operation ExtractMax (2)

EVERYBODY: TRY TO REMOVE THE ROOT OF THE FOLLOWING HEAP:

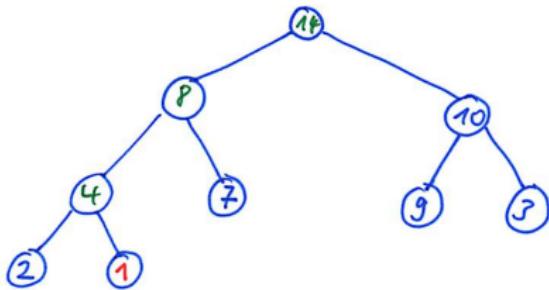


# Operation ExtractMax (3)

Solution: “let it bubble down”



Replace the root by the last element  
Heapify the root again



# Operation ExtractMax (4)

Running time::  $O(\log n)$

# Operation: DecreaseKey

Goal: *decrease* the key value of a particular element.

Solution:

- ▶ Decrease the value of the key
- ▶ Call heapify at this vertex to let it bubble down.

Running time:  $O(\log n)$

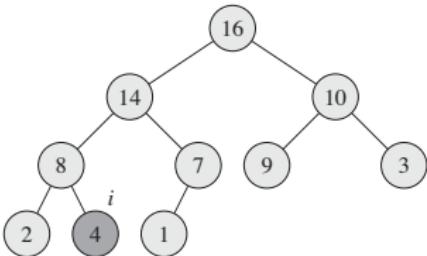
# Operation: IncreaseKey

Goal: *increase* the key value of a particular element.

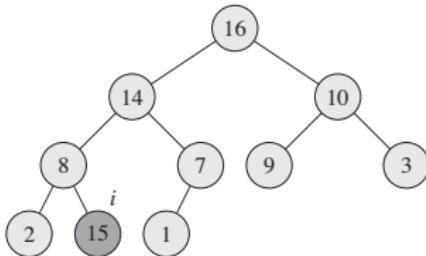
Solution:

- ▶ Increase the value of the key
- ▶ Do “heapify-up”: Walk upwards to the root, in each step exchanging the key values of a vertex and its parent if the heap property is violated.

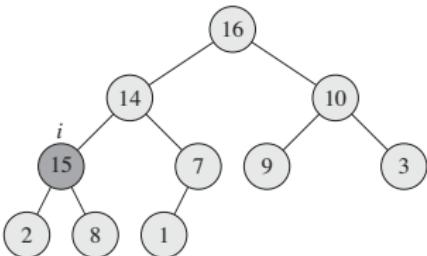
# Operation: IncreaseKey (2)



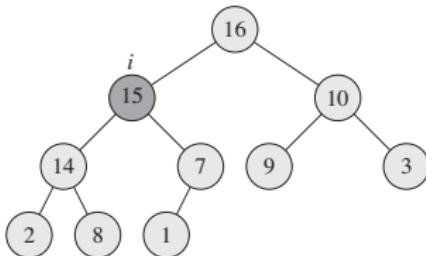
(a)



(b)



(c)



(d)

Running time:  $O(\log n)$

# Operation: InsertElement

Goal: insert a new element to the heap

Solution:

- ▶ insert it at the next free position, first assign it the key  $-\infty$ .
- ▶ Then call IncreaseKey to set the key to the given value.

Running time:  $O(\log n)$

# Outlook

- ▶ Heaps are used all over the place, we are going to see them often in this course.
- ▶ There exist much more elaborate heap implementations that try to decrease the running times of some of the operations.

If you are interested, visit the “Algorithmik”-lectures in the master program ...

# Priority queue

# Priority queue

- ▶ Data structure to maintain a set  $S$  of elements
- ▶ Each element has a key value
- ▶ When we dequeue the next element, we want to get the one with the largest key value.

Standard operations are:

- ▶ Enqueue, Dequeue, IncreaseKey of a particular element

# Priority queue (2)

Standard implementation uses heaps:

- ▶ Store the elements in a heap
- ▶ Enqueue: heap InsertElement,  $O(\log n)$
- ▶ Dequeue: heap ExtractMax  $O(\log n)$
- ▶ IncreaseKey: heap Increase key  $O(\log n)$

Note: there exist more elaborate implementations with better amortized running times, see the “Algorithmik” master class.

# Hashing

Literature: Cormen 11, Dasgupta 1.5

# Basics

# General problem

- ▶ Assume we are an online shop and want to maintain a list of data related to the customers that are currently online (say, the pages they clicked on during the last 20 min).
- ▶ We can identify customers by IP address
- ▶ But who is online changes constantly
- ▶ We want to have fast access time to the data.

First attempt:

- ▶ create an array of the size of all possible IP address:  $2^{32}$  (IP addresses have 32 bits)
- ▶ Fill the entries corresponding to the customers that are currently active.
- ▶ Fast access time

## General problem (2)

- ▶ But obviously not a good idea as the number of active customers is much smaller than the number of possible IP addresses.

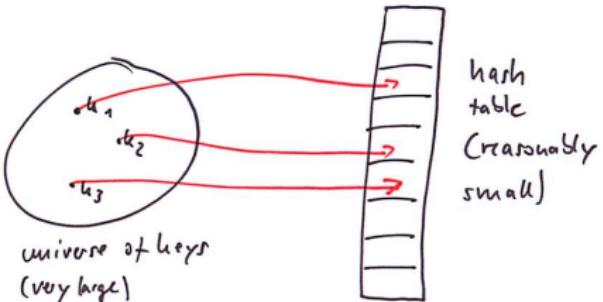
# General problem (3)

Second attempt:

- ▶ Simply generate a linked list of some kind.
- ▶ Efficient for space, but might have long access times for a particular customer.

The solution is hashing.

# Hashing, general principle



- ▶ Want to store data that is assigned to particular key values (the IP addresses)
- ▶ Give a “nickname” to each of the key values (for all  $2^{32}$  IP addresses).
- ▶ Choose the space of nicknames reasonably small (a bit larger than the number of customers that are usually online)
- ▶ Have a way to compute “nicknames” from the keys themselves

# Hashing, general principle (2)

- ▶ Then store the information in an array (size = number of nicknames)

# Hashing, formally

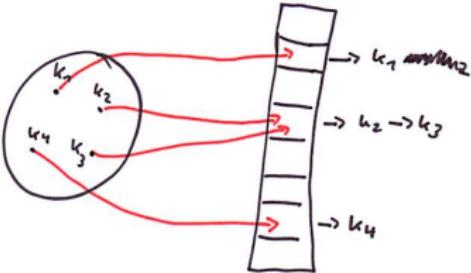
- ▶ The set of all possible keys we want to access is called the **universe  $U$  of keys**.
- ▶ Assume that the set of actual key values is much smaller, say it has size  $m \ll |U|$ .
- ▶ A **hash function** is a function  $h : U \rightarrow \{1, \dots, m\}$ .
- ▶ We say: **the element with key  $k$  hashes to slot  $h(k)$** .
- ▶ The values  $h(k)$  are called **hash values**.

# Hashing, formally (2)

## Collisions:

- ▶ From the definition of  $h$  it is obvious that many keys get the same hash values:  $h(k_1) = h(k_2)$  even though  $k_1 \neq k_2$ .
- ▶ This is called a **collision**.
- ▶ To cope with them:

Let each entry of the hash table point to a linked list that contains all elements with this particular hash key.



# Some simple hash functions

# What is a good hash function?

- ▶ We want that all our elements are distributed to the hash keys as uniformly as possible.
- ▶ We want to avoid collisions (they decrease our access speed as we have to travel through the list of collided keys to find the one we are interested in)
- ▶ We don't want the set of hash values to be too large (otherwise we waste space)

# Examples for simple hash functions

Division method for hashing integers:

- ▶ Assume that the universe of keys is  $\mathbb{N}$ .
- ▶ Assume you want to hash to  $m$  elements.
- ▶ Define the hash function

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

# Examples for simple hash functions (2)

Choice of  $m$ :

- ▶ If we don't know anything about our data set, then any  $m$  is as good as any other.
- ▶ But in practice, some values of  $m$  should be avoided:
  - ▶ For example: If we choose  $m = 2^a$ , then  $h(k)$  is just the  $a$  lowest-order bits of  $k$ . If the lower-order bits are not distributed uniformly, then this is a bad choice.
  - ▶ A general rule of thumb: if you want to hash to approximately  $m$  slots, then choose a prime number  $m_p$  that is a bit larger than  $m$ .

# Examples for simple hash functions (3)

Multiplication method for hashing integers:

- ▶ Multiply the key value by a constant  $a$  with  $0 < a < 1$
- ▶ Then take the fractional part  $p$  of the result (the part behind the comma)
- ▶ Then multiply  $p$  by  $m$  and take the ceil

In formulas:

$$h(k) = \lceil m (k \cdot a \bmod 1) \rceil$$

Example: Say,  $m = 50$ . Choose  $a = 0.17$ . Then:

$$k = 1 \implies k \cdot a = 0.17 \implies 0.17 \cdot m = 8.5 \implies \lceil 0.17 \cdot m \rceil = 9$$

$$k = 2 \implies k \cdot a = 0.34 \implies 0.34 \cdot m = 17$$

# Examples for simple hash functions (4)

Rationale:

- ▶ if  $p$  were a number uniformly drawn from  $[0, 1]$ , then this would be a good choice.
- ▶ We hope that the actual values of  $p$  behave more or less similarly

# What is the best hash function?

Depending on the actual key values:

- ▶ If we know the distribution of the key values, we can try to exploit this when selecting a hash function.
- ▶ If we don't know it, we might make a bad choice.

Can we design a hash function that works good for all sets of keys?

# What is the best hash function? (2)

No!!!

- ▶ If you give me a hash function ...
- ▶ and I am your adversary, then I can design a data set that performs worst for your hash function (HOW?)

A “best hash function” that works for all kinds of data sets does not exist!!!

# Sorting

# Elementary Sorting Algorithms

# Problem of Sorting

Given: an array of  $n$  numbers

76, 354, 2, 653, 24, 3, 4, 6

Goal: Sort the numbers in increasing order.

2, 3, 4, 6, 24, 76, 354, 653

- ▶ This is one of the most basic operations on an array, and it is needed all over the place.
- ▶ We are now going to spend quite some time to investigate various sorting algorithms.

# Warmup: Elementary Sorting algorithms

EVERYBODY:

- ▶ Take a couple of minutes to come up with your own sorting algorithm.
- ▶ Try to write it up in pseudo-code.
- ▶ Try to estimate its running time.

# Selection sort

# Selection sort

Idea:

- ▶ Find the smallest element in the input array  $A$
- ▶ Remove it from the input array and write it in the output array  $B$
- ▶ Repeat this process.

## SelectionSort( $A$ )

- 1  $n = \text{length}(A)$
- 2  $B = \text{new array of length } n$
- 3 **for**  $it=1, \dots, n$ 
  - 4 Find the smallest element  $a$  and its index  $p$  in  $A$
  - 5  $B(it) = a$  # Insert the element in  $B$
  - 6  $A(p) = \text{NaN}$  # Replace the element in  $A$  by NaN
  - 7 Return  $B$

## Selection sort: example

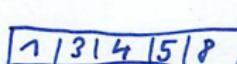
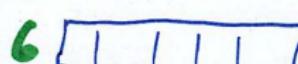
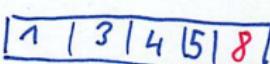
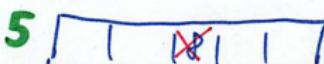
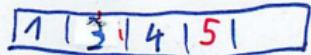
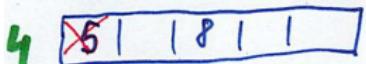
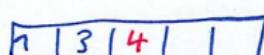
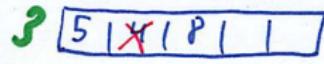
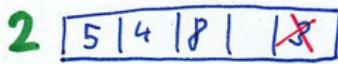
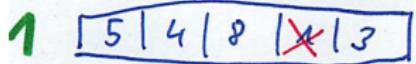
EVERYBODY: RUN SELECTION SORT ON THE EXAMPLE SEQUENCE

5,4,8,1,3

KEEP TRACK OF THE INPUT AND OUTPUT ARRAY IN EACH STEP.

# Selection sort: example (2)

Selection sort:



# Analysis of selection sort: termination

**Does the algorithm always terminate?**

# Analysis of selection sort: termination

**Does the algorithm always terminate?**

Yes. The for loop has a fixed number of iterations, so nothing can go wrong.

# Analysis of selection sort: correctness

**Is the array returned by the algorithm always sorted in increasing order?**

# Analysis of selection sort: correctness (2)

**Is the array returned by the algorithm always sorted in increasing order?**

Yes. To prove it formally, we use the concept of an **invariant**:

- ▶ An invariant is a logical assertion for which we prove that it always holds true during the execution of an algorithm.

# Analysis of selection sort: correctness (3)

## Proposition 3 (Invariant of selection sort)

- (1) After the  $i$ -th execution of the for-loop of selection sort, the first  $i$  elements in Array  $B$  are sorted in increasing order.
- (2) No element in  $B$  is strictly larger than an element in  $A$ .

Proof by induction.

- ▶ Base case (“Induktionsanfang”)  $i = 1$ : clear:
  - (1)  $B$  just contains one element so far, so it is sorted.
  - (2) The element in  $B$  was the smallest in  $A$ , so (2) correct.
- ▶ Induction hypothesis (“Induktionsannahme”): Assume the statement holds for  $i = 1, \dots, k$ .

## Analysis of selection sort: correctness (4)

- ▶ Induction step (“Induktionsschritt”): We now have to prove that it also holds for  $k + 1$ .

In the for-loop, we pick the smallest remaining element of  $A$ . By the induction hypothesis (2), this is never smaller than any element in  $B$ . So when we add it to the end of  $B$ , the sequence in  $B$  is sorted increasingly, hence (1) is true for  $k + 1$ .

(2) remains true as well, obviously by the way we choose the new element.



The correctness of the algorithm now follows from the invariant.

# Analysis of selection sort: time complexity

**What is the worst case running time of the algorithm?**

# Analysis of selection sort: time complexity (2)

## What is the worst case running time of the algorithm?

The running time is  $O(n^2)$ :

- ▶ line 1:  $O(1)$
- ▶ line 2:  $O(n)$
- ▶  $n$  runs of the for-loop. Inside the for-loop:
  - ▶ line 4:  $O(n)$
  - ▶ lines 5 and 6:  $O(1)$
- ▶ line 7:  $O(1)$

So overall, the dominating part is that we run the for-loop  $n$  times and each run needs time  $O(n)$ , so we end with worst case running time  $O(n^2)$ .

# Analysis of selection sort: time complexity (3)

**What is the best case running time?**

... is  $O(n^2)$  as well.

# Analysis of selection sort: space complexity

**What is the space complexity of the algorithm?**

# Analysis of selection sort: space complexity (2)

**What is the space complexity of the algorithm?**

It is  $O(n)$ :

- ▶ Arrays  $A$  and  $B$  both need space  $O(n)$ , everything else is constant.

Exercise: The algorithm can also be implemented “in-place”, that is without an additional array  $B$ .

# Insertion sort

# Insertion sort

This is the principle that most people use when playing card games and they want to sort the cards in their hand:

- ▶ One after the other, pick up the next element from the unsorted array.
- ▶ Maintain a sorted output array  $B$  and always insert the current element to its correct position.

# Insertion sort (2)

INSERTION-SORT( $A$ )

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3      // Insert  $A[j]$  into the sorted sequence  $A[1..j - 1]$ .
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i + 1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i + 1] = key$ 
```

## Insertion sort: example

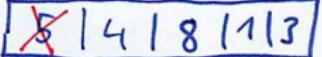
EVERYBODY: RUN SELECTION SORT ON THE EXAMPLE SEQUENCE

5,4,8,1,3

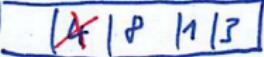
KEEP TRACK OF THE INPUT AND OUTPUT ARRAY IN EACH STEP.

# Insertion sort: example (2)

Insertion sort

1 



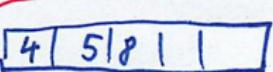
2 

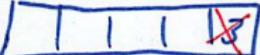


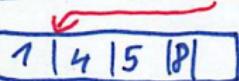
3 

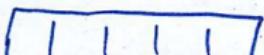


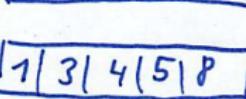
4 



5 



6 



# Selection sort: properties

Properties:

- ▶ Termination: yes
- ▶ Correctness: yes
- ▶ Worst case running time:  $O(n^2)$
- ▶ Best case running time:  $O(n)$
- ▶ Space complexity:  $O(n)$

Proof: exercise!

# Bubble sort

# Bubble sort

You have seen this algorithm in the exercises already:

BUBBLESORT( $A$ )

```
1  for  $i = 1$  to  $A.length - 1$ 
2      for  $j = A.length$  downto  $i + 1$ 
3          if  $A[j] < A[j - 1]$ 
4              exchange  $A[j]$  with  $A[j - 1]$ 
```

It always terminates, is correct, has running time  $O(n^2)$  (both in worst and average case).

# Mergesort

Literature: all text books

History: apparently, invented by John von Neumann in the 1940ies

# Motivation

So far we have seen a number of naive sorting algorithms, but all of them had worst case running time  $O(n^2)$ .

We now want to show an algorithm that is provably faster.

We want to use the **divide-and-conquer principle**:

- ▶ We iteratively divide the given problems into smaller subproblems
- ▶ Then we construct the solution of the whole problem by combining the solutions of the smaller problems.

# Idea of the algorithm

Idea of merge sort:

- ▶ Assume we are given a list of numbers
- ▶ Split the list in two halves
- ▶ Recursively sort each of the lists
- ▶ Then merge the two sorted lists again.

# Mergesort: Pseudo code

```
function mergesort( $a[1\dots n]$ )
```

Input: An array of numbers  $a[1\dots n]$

Output: A sorted version of this array

```
if  $n > 1$ :
```

```
    return merge(mergesort( $a[1\dots \lfloor n/2 \rfloor]$ ), mergesort( $a[\lfloor n/2 \rfloor + 1\dots n]$ ))
```

```
else:
```

```
    return  $a$ 
```

```
function merge( $x[1\dots k], y[1\dots l]$ )
```

```
if  $k = 0$ : return  $y[1\dots l]$ 
```

```
if  $l = 0$ : return  $x[1\dots k]$ 
```

```
if  $x[1] \leq y[1]$ :
```

```
    return  $x[1] \circ \text{merge}(x[2\dots k], y[1\dots l])$ 
```

```
else:
```

```
    return  $y[1] \circ \text{merge}(x[1\dots k], y[2\dots l])$ 
```

Here  $\circ$  denotes the concatenation of the lists.

# Merge sort: Example

EVERYBODY: APPLY THE MERGE OPERATIONS TO THE TWO SEQUENCES

3 4 5 9

and

1 2 3 4

## Merge sort: Example (2)

Merge (3459, 1235)

1 o merge (3459, 235)

2 1 o merge (3459, 35)

3 2 1 o merge (459, 35)

3 3 2 1 o merge (459, 5)

4 3 3 2 1 o merge (59, 5)

5 4 3 3 2 1 o merge (9, 5)

5 5 4 3 3 2 1 o merge (9, [])

9 5 5 4 3 3 2 1

## Merge sort: Example (3)

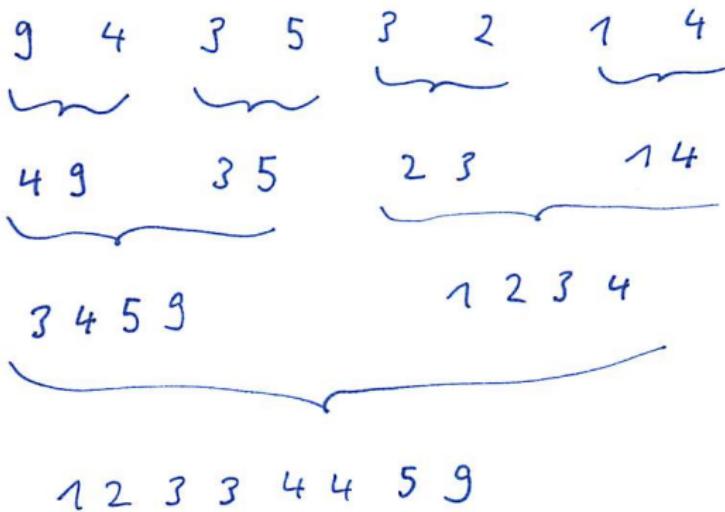
EVERYBODY: APPLY THE MERGE SORT ALGORITHM TO THE SEQUENCE

9, 4, 3, 5, 3, 2, 1, 4

## Merge sort: Example (4)

Input 9 4 3 5 3 2 14

Recursively split in singletons, then merge:



# Merge sort: Running time

Proposition 4 (Worst case running time of merge sort)

The worst case running time of merge sort is  $O(n \log n)$ .

Proof:

- ▶ `merge(x[1..k], y[1..l])` has running time  $O(k + l)$ :
  - ▶ Constant amount of work per recursive call
  - ▶  $k + l$  recursive calls
- ▶ Because we only ever apply merge to arrays of length at most  $n$ , we have  $k, l \leq n$ , hence  $O(k + l) \subset O(n)$ .
- ▶ So the running time of merge sort satisfies

$$T(n) = 2T(n/2) + O(n)$$

# Merge sort: Running time (2)

- ▶ By the master theorem, this leads to  $O(n \log n)$ .



# Merge sort: Running time (3)

**What is the best case running time for merge sort?**

We can't really save any time. Even if we start with a sorted sequence, we still go through all the steps.

Hence the best case running time is still  $O(n \log n)$ !

Consequently, also the average case running time of merge sort is  $O(n \log n)$  as well.

## Merge sort: space complexity

- ▶ A naive implementation needs extra space of the size  $n$  of the original array in order to store the intermediate results.
- ▶ It is reasonably easy to bring this down to extra space of size  $n/2$ .
- ▶ It is not straight forward but possible to achieve in-place sorting with mergesort (if this is the goal, other algorithms are better, see later for discussion). Then the running time increases slightly, to  $n \log^2 n$ .

# Termination and correctness

Is pretty straight forward to termination and correctness. Exercise ...

# Heap sort

Literature: Cormen Sec. 6

Original paper for heap sort:

Williams, J. W. J. (1964). Algorithm 232 - Heapsort.  
Communications of the ACM 7 (6): 347-348

# Heapsort: idea

Idea is to exploit the heap data structure.

- ▶ We simply build a heap out of our input.
- ▶ Then we remove the largest element.
- ▶ Then we heapify again.
- ▶ And so on ...

# Heap sort: pseudo-code

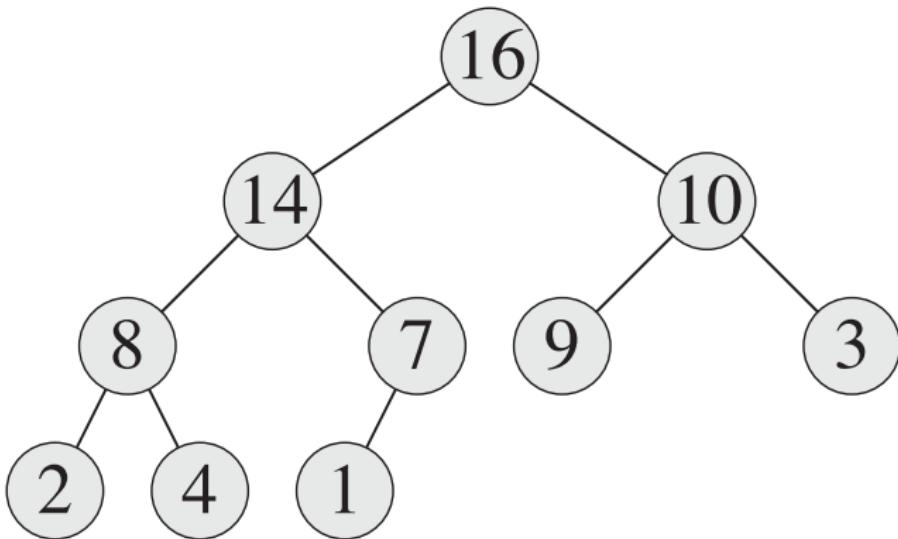
HEAPSORT( $A$ )

- 1 BUILD-MAX-HEAP( $A$ )
- 2 **for**  $i = A.length$  **downto** 2
- 3     exchange  $A[1]$  with  $A[i]$
- 4      $A.heap-size = A.heap-size - 1$
- 5     MAX-HEAPIFY( $A, 1$ )

## Heap sort: example

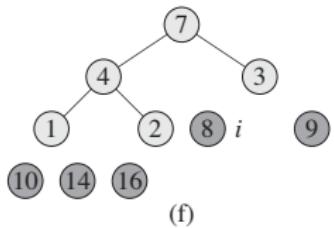
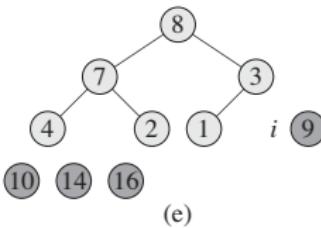
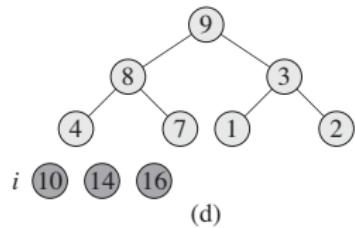
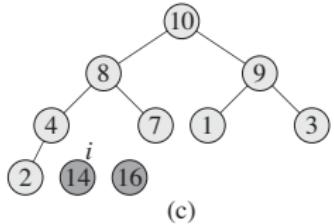
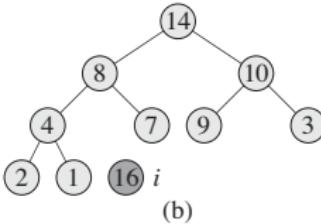
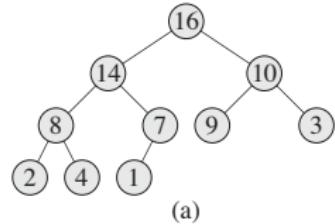
EVERYBODY, TRY HEAP SORT ON THE FOLLOWING INSTANCE:

## Heap sort: example (2)

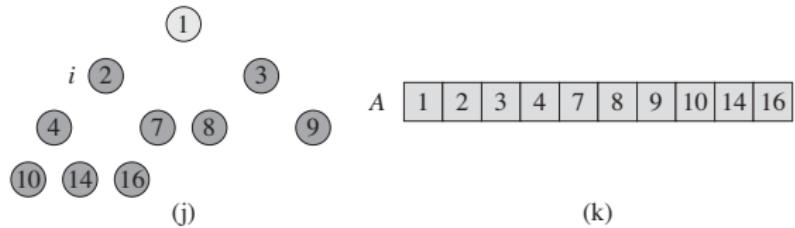
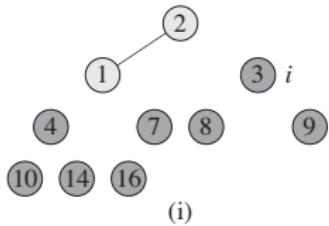
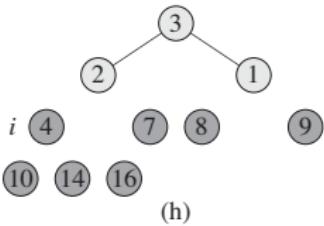
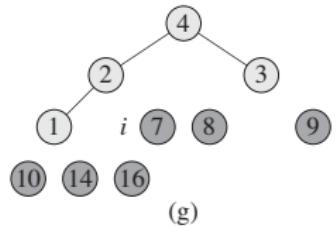


(a)

# Heap sort: example (3)



# Heap sort: example (4)



A	[ 1   2   3   4   7   8   9   10   14   16 ]
---	--

(k)

# Heap sort: termination and correctness

Both clear.

Exercise: prove it formally.

# Heap sort: running time

**What is the running time of heap sort?**

# Heap sort: running time

**What is the running time of heap sort?**

- ▶ Build-max-heap takes  $O(n)$
- ▶ Each of the heapify calls takes  $O(\log n)$ , and we have  $n$  such calls.

So altogether:  $O(n \log n)$ .

# Heap sort: running time

**What is the running time of heap sort?**

- ▶ Build-max-heap takes  $O(n)$
- ▶ Each of the heapify calls takes  $O(\log n)$ , and we have  $n$  such calls.

So altogether:  $O(n \log n)$ .

**What about the best case running time?**

# Heap sort: running time

**What is the running time of heap sort?**

- ▶ Build-max-heap takes  $O(n)$
- ▶ Each of the heapify calls takes  $O(\log n)$ , and we have  $n$  such calls.

So altogether:  $O(n \log n)$ .

**What about the best case running time?**

Is  $O(n \log n)$  as well (WHY?)

# Heap sort: space complexity

Heap sort works in-place! So we don't need any extra space.

... make sure you really understand why ...

# Lower bound

Literature:

Dasgupta Sec 2.3

Cormen Sec. 8.1

Mehlhorn Sec. 5.3

# Motivation

We have seen several algorithms that solve the sorting problem in  $\Theta(n \log n)$ . Is this the best we can do?

To answer this question:

- ▶ We need a **lower bound** (German: untere Schranke) on the worst case running time
- ▶ The statement is going to look as follows:

Under [some assumptions], no algorithm that solves the sorting problem can have a worst case algorithm at least [lower bound].

- ▶ Lower bounds are notoriously difficult (WHY?), and only exist for a few problems (sorting is one of them).

# Comparison-based sorting algorithms

In this section we only consider sorting algorithms that are based on pairwise comparison of elements:

- ▶ Input to the algorithm is the sequence  $A = [a_1, \dots, a_n]$ . Assume that all elements in  $A$  are distinct, that is  $a_i \neq a_j$  for all  $i \neq j$ .
- ▶ The algorithm is allowed to make arbitrarily many comparisons of the form: Is  $a_i \leq a_j$  or is  $a_i \geq a_j$ .
- ▶ Based on the results of these comparisons, it finally produces the correctly sorted output.

Remark:

- ▶ All sorting algorithms we have seen so far follow this principle.
- ▶ Later we will see that other algorithms exist as well.

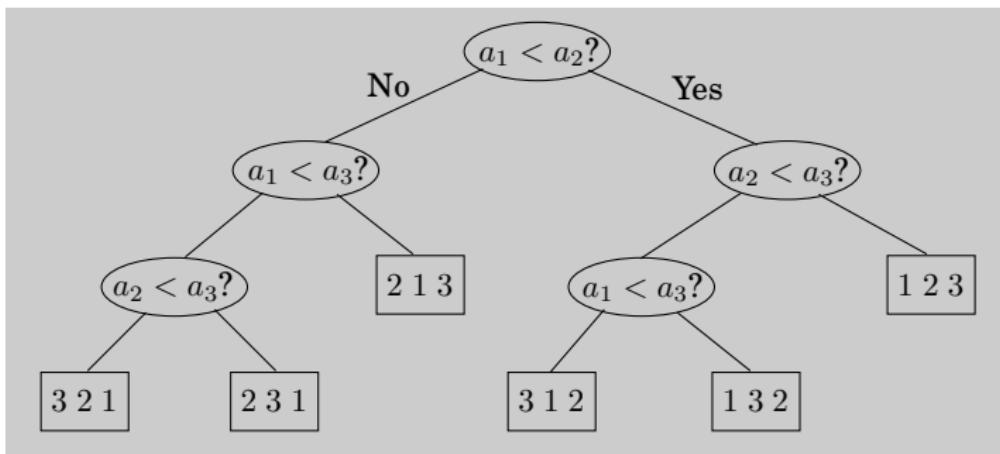
# The lower bound

## Theorem 5 (Lower bound for sorting)

Any comparison-based sorting algorithm requires  $\Omega(n \log n)$  comparisons in the worst case.

# Proof of the lower bound

Consider the computation tree of a comparison-based algorithm:



- ▶ Any comparison-based algorithm can be described by a computation tree (WHY?).
- ▶ The tree is always binary (WHY?), but not necessarily complete (WHY?).

## Proof of the lower bound (2)

Given the tree, how can you see the worst case running time of the algorithm?

## Proof of the lower bound (2)

Given the tree, how can you see the worst case running time of the algorithm?

The height  $h$  of the tree.

## Proof of the lower bound (2)

Given the tree, how can you see the worst case running time of the algorithm?

The height  $h$  of the tree.

To get a lower bound we need to find out what is the minimal height of the tree. Ideas?

## Proof of the lower bound (2)

Given the tree, how can you see the worst case running time of the algorithm?

The height  $h$  of the tree.

To get a lower bound we need to find out what is the minimal height of the tree. Ideas?

What do we know about the number of leaves of the tree?

## Proof of the lower bound (2)

Given the tree, how can you see the worst case running time of the algorithm?

The height  $h$  of the tree.

To get a lower bound we need to find out what is the minimal height of the tree. Ideas?

What do we know about the number of leaves of the tree?

At least  $n!$  leaves, one for each possible permutation of the input sequence.

## Proof of the lower bound (2)

Given the tree, how can you see the worst case running time of the algorithm?

The height  $h$  of the tree.

To get a lower bound we need to find out what is the minimal height of the tree. Ideas?

What do we know about the number of leaves of the tree?

At least  $n!$  leaves, one for each possible permutation of the input sequence.

What is the minimal height of a binary tree with at least  $n!$  leaves?

Height is minimal if the binary tree is complete. By Stirling's approximation,

$$h = \Omega(\log(n!)) = \Omega\left(\log\left(\sqrt{2\pi n}(n/e)^n(1 + \Theta(1/n))\right)\right) = \Theta(n \log n)$$



## Remarks

- ▶ The lower bound shows that there cannot exist any comparison-based sorting algorithm that has worst case running time faster than  $n \log n$ .
- ▶ In this sense, heap sort and merge sort are asymptotically worst-case optimal.
- ▶ Note again that all constants in the running time got swallowed by the Landau notation. It might very well be that one sorting algorithm takes  $2n \log n$  and the other one takes  $10^5 n \log n$ , which would make a considerable difference in practice.

Make sure you really understand what it says:

For any comparison-based sorting algorithm there always exists an instance such that the algorithm on that instance needs  $\Omega(n \log n)$ .

# Outlook

- ▶ One can prove that the  $n \log n$  lower bound also holds for the average running time.
- ▶ One can also show that the  $n \log n$  lower bound also holds for the seemingly simpler “element uniqueness problem”: the problem of deciding whether a sequence contains two elements that are equal (using comparisons only).

## Outlook (2)

- ▶ History: I did not manage to find out where and when the lower bound was proved first. A generalization is the work on algebraic decision trees, with the following two seminal papers:
  - ▶ David P. Dobkin and Richard J. Lipton. On the complexity of computations under varying sets of primitives. *Journal of Computer and System Sciences*, 18(1):86-91, Feb. 1979.
  - ▶ Michael Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (STOC 1983)*, pp. 80-86, April 1983.

# Appetizer / Excursion: finding the median

Literature:

- ▶ Dasgupta Sec. 2.4
- ▶ Kleinberg Sec. 13.5
- ▶ Mehlhorn Sec. 5.5

# Median

- ▶ The median element of a sorted sequence of integers  $a_1, \dots, a_n$  is the element with index  $\lceil n/2 \rceil$ : the element in the middle.
- ▶ The median is important in many applications, the median element is interpreted as the “typical” element of the list.
- ▶ Note the difference between median and mean:

Example:

$$\underbrace{1, 1, 1, 1, 1, \dots, 1}_{\text{100 times}}, 1000$$

- ▶ Median is 1
- ▶ Mean is  $\sum_{i=1}^n a_i / n = 11$

# Median and selection problem

Median finding problem:

Given an unordered sequence of  $n$  elements, we want to find the median element.

Generalization: Selection problem:

Given an unordered sequence of  $n$  elements, we want to find the  $k$ -th smallest element.

# Naive solution

Any ideas how to solve the median problem?

## Naive solution

Any ideas how to solve the median problem?

Sort the sequence, and then pick the middle element. Takes  $O(n \log n)$ .

Intuitively, this seems like an overkill! (WHY?) Can we do better?

# Generic algorithm, intuition

Idea: divide and conquer

- ▶ At each step in time, select a “splitter element”  $s$  and split the given sequence in two pieces  $S_-$  and  $S_+:$ 
  - ▶  $S_-$  contains all elements that are smaller than  $s$
  - ▶  $S_+$  contains all elements that are larger than  $s$
- ▶ Then, depending on the sizes of  $S_-$  and  $S_+$ , we know in which of the two parts  $S_-$  and  $S_+$  we have to continue our search:

## Generic algorithm, intuition (2)

Sequence 10, 25, 1, 30, 17, 83, 70, 40, 21

Split at 30 :

10, 25, 1, 17, 21 || 30 || 83, 70, 40

Then we know:

- Smallest element must be in  $S_-$
- Second-smallest element must be in  $S_-$
- Median must be in  $S_-$

---

# Generic algorithm, pseudo-code

Select ( $S, k$ )

Choose splitter element  $a_i \in S$

For each  $a \in S$

If  $a < a_i$ , put  $a$  in  $S_-$

If  $a = a_i$ , put  $a$  in  $S_=$

If  $a > a_i$ , put  $a$  in  $S_+$

If  $|S_-| \leq k \leq |S_-| + |S_=|$

Return  $a_i$       The splitter was the desired answer.

If  $|S_-| \geq k$       Solution must be in  $S_-$

Return Select ( $S_-, k$ )

If  $|S_-| + |S_=| < k$       Solution must be in  $S_+$

Return Select ( $S_+, k - |S_-| - |S_=|$ )

# How to choose a splitter?

WHAT WOULD BE THE BEST SPLITTER ELEMENT?

## How to choose a splitter? (2)

The best situation that could possibly happen is that we are very lucky, and in the first round choose the splitter as the element we are looking for.

Then we are done after this round, and just needed  $n - 1$  comparisons.

But it is unrealistic to hope that this always happens ...

## How to choose a splitter? (3)

WHAT CAN HAPPEN IF WE ALWAYS CHOOSE A BAD SPLITTER ELEMENT?

# How to choose a splitter? (4)

In the worst case:

- ▶ Assume we want to find the median of the sequence.
- ▶ But we have a stupid rule for selecting the splitter, it always chooses the maximum element in the sequence.
- ▶ Then, in each step the size of  $S_-$  decreases by 1. In each round we need to compare against all elements in  $S_i$ . We need to do so until  $S_i$  has size  $n/2$ , then we have found the median.

$$(n - 1) + (n - 2) + \dots + n/2 = \Theta(n^2)$$

number of comparisons.

## How to choose a splitter? (5)

SO WHAT WOULD BE A GOOD STRATEGY TO AVOID THE WORST CASE?

# How to choose a splitter? (6)

- ▶ Try to split the sequence in two equal-sized parts.
- ▶ The corresponding splitter element would be the median ...  
hmmm

# Randomized selection algorithm

We are now going to use the following rule for selecting the splitter:

We pick the splitter element  $s$  randomly from the elements in our list!

We call the resulting algorithm the Randomized selection algorithm.

Intuition why this makes sense:

- ▶ As long as most of the splitter elements are “somewhat from the middle”, we significantly reduce the length of the two sublists.
- ▶ And it is quite likely to choose an element “somewhat from the middle”

# Randomized algorithms

This is our first encounter with a powerful tool, a **randomized algorithm**:

- ▶ A randomized algorithm has access to a random bit generator (a black box mechanism that can produce either 0 and 1 with probability 1/2 each).
- ▶ The model of computation is that generating one random bit costs one unit of time.
- ▶ In this case, the algorithm is a so-called Las-Vegas-algorithm: It is always going to produce the correct result, but the running time is going to depend on “how lucky” we are when drawing the random elements.

# Randomized algorithms (2)

For randomized algorithms, we are interested in the **expected running times**.

Be careful, this expectation is with respect to the randomness in the algorithm (not with respect to the randomness in the instance).

- ▶ Expected worst case behavior:

$$\max_{I \in \mathcal{I}_n} E(T(I))$$

- ▶ Expected average case behavior:

$$\frac{1}{n} \sum_{I \in \mathcal{I}_n} E(T(I))$$

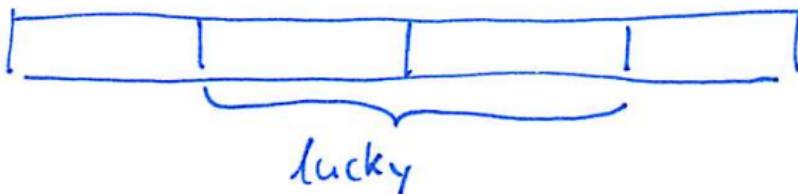
# Expected running time

## Theorem 6 (Expected running time)

On any input sequence of length  $n$ , the expected running time of the randomized median algorithm is  $O(n)$ .

Proof:

We call the splitter element  $s$  “lucky” if it is in the middle 50 % of the (sorted) elements in  $S$ :



## Expected running time (2)

- ▶ If  $s$  is lucky, then both  $S_-$  and  $S_+$  have size at most  $3/4|S|$ . (WHY?)
- ▶ At least half of the elements in  $S$  are lucky ones. If we randomly pick an element of  $S$ , then with probability at least  $1/2$  it is going to be lucky.
- ▶ Assume we repeatedly draw “splitter candidates” from  $S$ . On average, we have to draw twice before we get a lucky one.
  - ▶ If we draw a lucky one in the first place, we are done.
  - ▶ With probability  $1/2$ , however, we have to repeat.
  - ▶ Hence, the expected number of trials before we get a lucky element is  $E = 1 + E/2$ , which gives  $E = 2$ .
- ▶ So on average, after at two split operations the arrays will shrink by a factor of  $3/4$ .

## Expected running time (3)

- ▶ Observe that the “time taken on array of size  $n$ ” is smaller than the sum “time on array of size  $3n/4$ ” plus the “time to reduce array size to  $3n/4$ ”. Thus:

$$ET(n) \leq ET(3/4n) + O(n)$$

- ▶ The master theorem now gives us that  $ET(n) = O(n)$ .



## Expected running time (4)

Digest again the statement in the theorem: The **expected running time** is  $O(n)$ .

- ▶ “Expected” means “on average”. There can still be instances where the running time is much longer, and such an average case statement does not give any guaranteed upper bound on the running time.
- ▶ The theorem does not say anything about the variance of the running time (variance is a measure for how much the running time varies). Small variance would be good, because then we would know that the actual running time tends to be very close to its expectation.

# Comments

- ▶ Obviously, the algorithm cannot be better than  $O(n)$  (WHY?).
- ▶ It is not surprising that one can solve the problem  $\text{Select}(S,1)$  or  $\text{Select}(S,n)$  in time  $O(n)$  (WHY?)
- ▶ But it is somewhat surprising that we can solve the median problem in  $O(n)$  (IS IT?)

Take this algorithm as an appetizer for the power and beauty of randomized algorithms ☺

# Quicksort

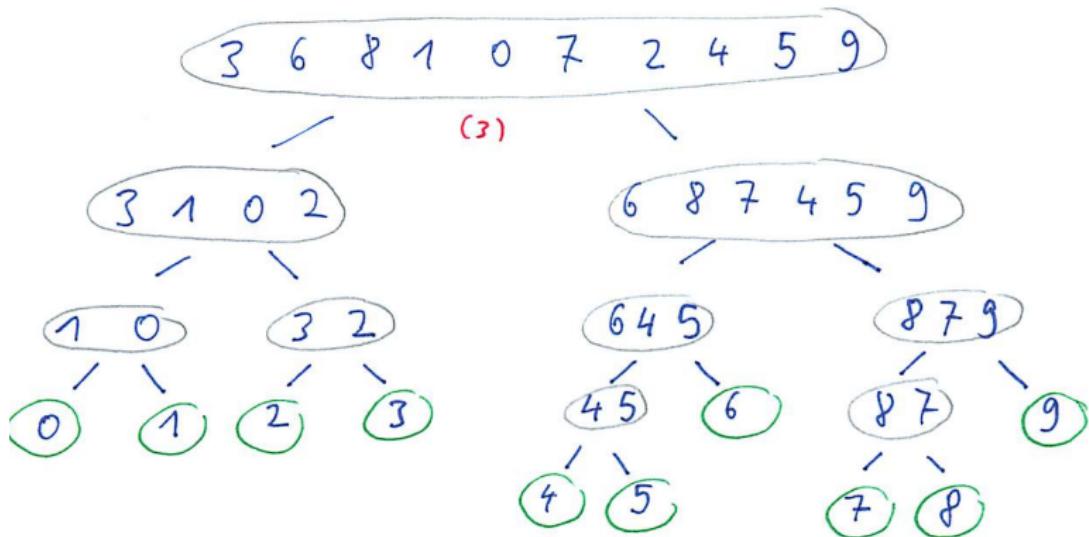
Literature: Mehlhorn Sec. 5.4, Cormen Sec. 7

# Idea

Yet again a divide-and-conquer algorithm, very similar to the median-finding algorithm.

- ▶ Given a sequence, we split it two pieces. We do this in such a way that the elements in the first piece are always smaller than the elements in the second piece. This is the “divide step”.
- ▶ Conquer step: Then we recurse. In the end, we just have to concatenate the resulting sequences. This is the “conquer” step.

## Idea (2)



## Idea (3)

Selecting the pivot element:

- ▶ As in the median finding problem, it is crucial to achieve “lucky splits”: the two resulting sequences should have (approximately) the same length.
- ▶ As in the median finding problem, a very good strategy is to select the pivot element uniformly at random from the elements of the input sequence.

The resulting algorithm is called randomized quicksort.

# Randomized quicksort pseudo-code

**Function**  $\text{quickSort}(s : \text{Sequence of Element}) : \text{Sequence of Element}$

**if**  $|s| \leq 1$  **then return**  $s$

pick  $p \in s$  uniformly at random

$a := \langle e \in s : e < p \rangle$

$b := \langle e \in s : e = p \rangle$

$c := \langle e \in s : e > p \rangle$

**return** concatenation of  $\text{quickSort}(a)$ ,  $b$ , and  $\text{quickSort}(c)$

# Correctness and termination

Clear (REALLY? WHY?)

# Worst case running time

## Theorem 7 (Randomized quicksort worst case)

The worst case running time of randomized quicksort is  $\Theta(n^2)$ .

**Proof:** We need to show that there exists a sequence of pivot elements that can be chosen with positive probability and an input instance such that the running time of the resulting algorithm is  $\Theta(n^2)$ .

- ▶ To move all elements to the sequences  $a, b, c$  we need  $n - 1$  comparisons.

## Worst case running time (2)

- Then we recurse on  $a$  and  $c$ . Assume that  $a$  contains  $k$  elements and  $c$  contains  $k'$  elements (which can happen with positive probability). Then:

$$T(n) = T(k) + T(k') + n - 1$$

For the worst case, this means that

$$T(n) = \left( \max_{k, k': 0 \leq k \leq n-1, 0 \leq k' \leq n-k} T(k) + T(k') \right) + n - 1$$

- It is easy to prove by induction that this implies

$$T(n) \leq \frac{n(n-1)}{2} = \Theta(n^2).$$



So the worst-case running time is much slower than for heapsort or merge sort.

# Expected running time

## Theorem 8 (Quicksort expected average time)

For each instance, the expected running time of randomized quicksort is  $O(n \log n)$ .

# Expected running time (2)

Intuition why this is true:

- ▶ Assume that each split is “somewhat balanced”: none of the arrays contains more than, say,  $3/4$  of the elements.
- ▶ Then the computation tree of the algorithm would have height  $\Theta(\log n)$ , and in each level the amount of computation is  $O(n)$  (we need to make at most  $n$  comparisons to the pivots).
- ▶ This would lead to a running time of  $O(n \log n)$ .
- ▶ The trick of the prove is to argue that most of the splits will be balanced indeed, and that the few unbalanced splits don't seriously hurt our running time.

## Expected running time (3)

Formal proof is very much along the line of the median proof:

It is easy to see that the running time of the algorithm just depends on the number of comparisons we make. So we are going to count how many comparisons we make during the algorithm.

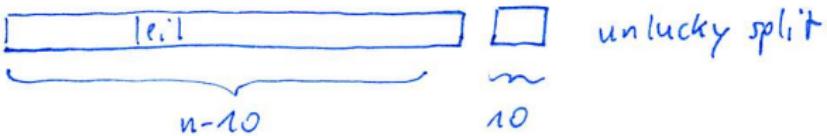
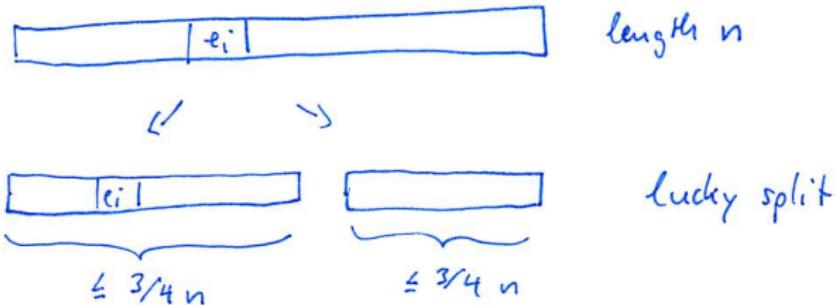
Setup:

- ▶ Fix one element  $e_i$  in the sequence.
- ▶ Denote by  $X_i$  the number of times this element is compared against a pivot element, throughout the whole algorithm.
- ▶ Then  $\sum_{i=1}^n X_i$  is the total number of comparisons.
- ▶ Obviously,  $X_i \leq n - 1$  (after each comparison, the element  $e_i$  ends in a strictly smaller subproblem, so there can only be  $n - 1$  subproblems involving  $e_i$ ).

# Expected running time (4)

Lucky comparisons:

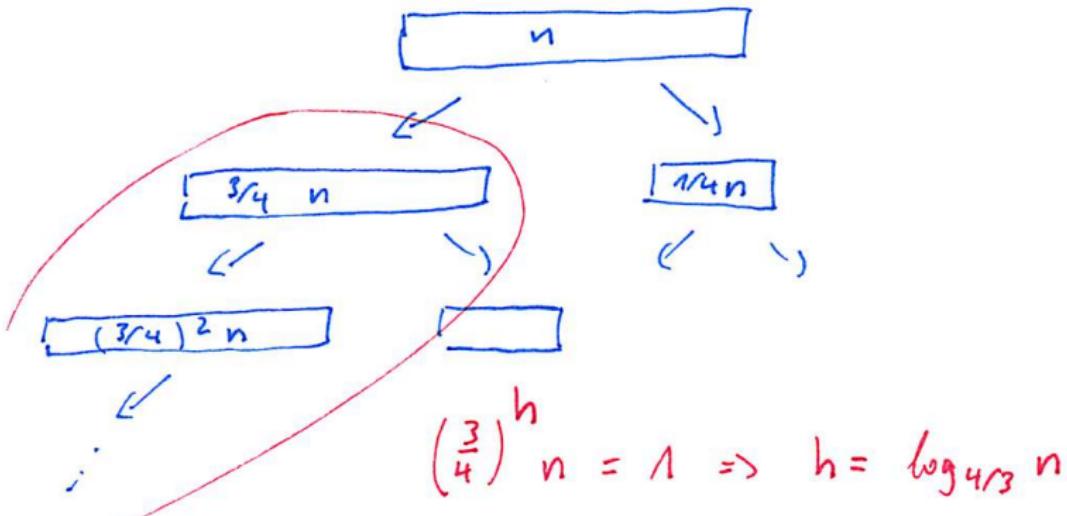
- Call a comparison between  $e_i$  and a pivot element “lucky” if  $e_i$  moves to a subproblem of size at most  $3/4$  the size of the current problem.



# Expected running time (5)

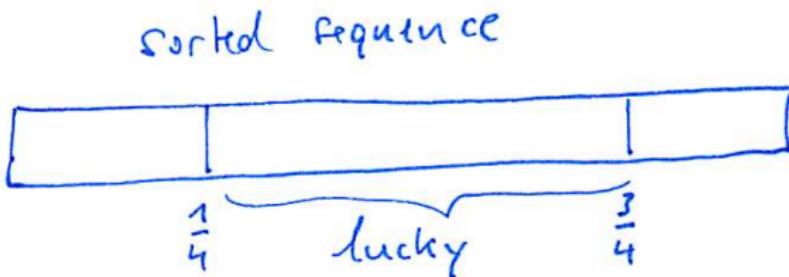
- If  $e_i$  is only involved in lucky comparisons, it needs to go through at most  $\log_{4/3} n$  comparisons.

Worst case if all splits are lucky:



## Expected running time (6)

- When we randomly pick a pivot element from the current elements, the likelihood to be “lucky” is  $1/2$  (just don’t pick one from the first and last quarter).



Half of its elements are lucky.

## Expected running time (7)

- ▶ By the same argument as in the median proof, it takes 2 splits on expectation to reduce the size of an array to  $3/4$  of its size.
- ▶ In each level, we need to perform at most  $n$  comparisons (each element against some pivot).
- ▶ So overall we end up with running time upper bound  $2n \log_{4/3} n = O(n \log n)$ .



# Comments about running time

- ▶ The theorem says “for each instance”. This implies that the expected best, average, and worst case running times are all  $O(n \log n)$ , and by the lower bound for comparison based sorting in fact  $\Theta(n \log n)$ .
- ▶ A slightly more subtle argument can be used to improve the constants in the running time to achieve expected time  $O(2n \ln n)$  where  $\ln$  is the natural logarithm. See the text books if you are interested in details.

# Space complexity

- ▶ Quicksort must store a constant amount of information for each nested recursive call (notably, it needs to keep track of the return addresses of the recursive calls on the so-called call stack).
- ▶ In the best case, we have to make  $O(\log n)$  recursive calls. In this case, we need  $O(\log n)$  additional space.
- ▶ In the worst case, we make  $O(\log n)$  recursive calls. Then we need additional  $O(n)$  space.

# Space complexity (2)

- ▶ There is a trick that can be used to limit the recursion stack of the algorithm to  $O(\log n)$ :
  - ▶ When we process the two recursive calls of quicksort, we always start with the recursively sort call of the smaller partition. The recursion depth here is  $O(\log n)$  (WHY?)
  - ▶ The other call is resolved using tail recursion:  
Procedure calls in the tail position of a procedure are treated as a direct transfer of control to the called procedure, hereby avoiding the need to add a new element to the call stack (intuitively, we replace the recursive call by a “GOTO”).

# Refinements

- ▶ Many more refinements of the quicksort algorithm are used in practice. In particular, it can be done in-place, and we can control the size of the recursion stack.
- ▶ Today, quicksort is the most widely used sorting algorithm in practice.

# History

According to wikipedia, the quicksort algorithm was developed in 1960 in the Soviet Union by Tony Hoare (who was a visitor at that time, and later became a very famous computer scientist who won a large number of prizes, among them the ACM Turing Award).

# Sorting in linear time

Literature:

Mehlhorn Sec. 5.6, Cormen Sec. 8.2-8.4

# Motivation

Have seen:

- ▶ Comparison-based sorting has lower bound  $\Omega(n \log n)$ .
- ▶ If we don't have any further information about our input, it seems hardly possible to come up with a sorting algorithm that is not comparison-based.
- ▶ However, in the following sections we will see how to break the lower bound if we make further assumptions on the input data.

# Counting sort / Ksort

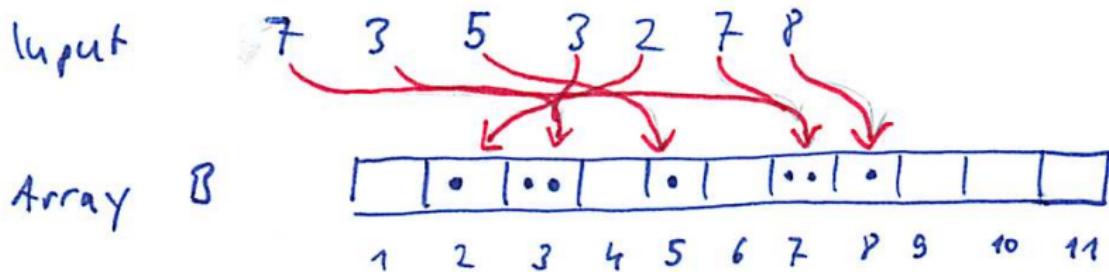
# Idea

Assume that the numbers to be sorted are integers in a fixed range, say between 0 and  $K$ .

- ▶ Provide an array  $B$  of  $K$  “buckets”
- ▶ Walk along the input sequence. Upon reading element  $A(i) =: k$ , increase a counter for bucket  $B(k)$ .
- ▶ In the end, just walk along  $B$  and collect the elements.

## Idea (2)

Counting sort:



Output      2 3 3 5 7 7 8

## Idea (3)

- ▶ More generally, assume you want to sort larger structs according to some key value.
- ▶ Assume that the key value is an integer between 0 and  $K$ .
- ▶ Provide an array of buckets again. The contents in each bucket are a linked list of elements.
- ▶ Whenever you encounter a struct with key value  $k$ , concatenate it to the list in bucket  $B(k)$ .
- ▶ In the end, concatenate the non-empty buckets.

# Pseudo-code

**Procedure**  $K\text{Sort}(s : \text{Sequence of Element})$

$b = \langle \langle \rangle, \dots, \langle \rangle \rangle : \text{Array } [0..K - 1] \text{ of Sequence of Element}$

**foreach**  $e \in s$  **do**  $b[\text{key}(e)].\text{pushBack}(e)$

$s := \text{concatenation of } b[0], \dots, b[K - 1]$

# Running time

Running time (worst, best, average) is  $O(n + K)$

- ▶ Allocate array  $C$  with  $K$  slots:  $O(K)$
- ▶ Read the sequence and each time update one of the counters / lists:  $O(n)$

It can beat the  $n \log n$  bound because the algorithm is not comparison-based. Such an algorithm is possible because we make additional assumptions on the input.

# Stability

We say that a sorting algorithm is stable if numbers with the same value occur in the output value in the same order as in the input value.

This property is important if we order lexicographically with several keys. Example:

- ▶ We have a data base of customers.
- ▶ We first sort them by their last name.
- ▶ Then we additionally sort by their first name.
- ▶ If the sorting algorithm is stable, the second sorting procedure does not destroy the first sorting.

# Stability (2)

Example:

Input		Sorted by last		Sorted by first	
Last name	First name	Last	First	Last	First
Newman	Alice	Blue	John	Blue	John
Smith	John	Newman	Alice	Newman	Alice
Blue	John	Smith	John	Smith	Bob
Smith	Bob	Smith	Bob	Smith	John

- ▶ In the second column, John Smith is before Bob Smith because this also was the case in the first column.
- ▶ In the third column, the sorting did not destroy the ordering by last name, it just additionally sorted by the first name. This only works if the sorting algorithm is stable (MAKE SURE YOU UNDERSTAND WHY).

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Counting sort:

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Counting sort: stable

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Counting sort: stable
- ▶ Insertion sort:

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Counting sort: stable
- ▶ Insertion sort: not stable

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Counting sort: stable
- ▶ Insertion sort: not stable
- ▶ Selection sort:

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Counting sort: stable
- ▶ Insertion sort: not stable
- ▶ Selection sort: are not

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Counting sort: stable
- ▶ Insertion sort: not stable
- ▶ Selection sort: are not
- ▶ Heap sort:

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Counting sort: stable
- ▶ Insertion sort: not stable
- ▶ Selection sort: are not
- ▶ Heap sort: not stable
- ▶ Merge sort:

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Counting sort: stable
- ▶ Insertion sort: not stable
- ▶ Selection sort: are not
- ▶ Heap sort: not stable
- ▶ Merge sort: stable

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Counting sort: stable
- ▶ Insertion sort: not stable
- ▶ Selection sort: are not
- ▶ Heap sort: not stable
- ▶ Merge sort: stable
- ▶ Quicksort:

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Counting sort: stable
- ▶ Insertion sort: not stable
- ▶ Selection sort: are not
- ▶ Heap sort: not stable
- ▶ Merge sort: stable
- ▶ Quicksort: not stable

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Counting sort: stable
- ▶ Insertion sort: not stable
- ▶ Selection sort: are not
- ▶ Heap sort: not stable
- ▶ Merge sort: stable
- ▶ Quicksort: not stable

# Radix sort

# Motivation

Assume that we still have to sort integers, but the upper bound  $K$  can be huge (in particular,  $K = \omega(n)$ ).

Idea is now:

- ▶ Treat each integer as a string
- ▶ First sort the strings according to the last digit (least significant digit) using ksort, then sort them according to the second last digit using counting sort, and so on.
- ▶ Because ksort is stable, this is going to lead to a sorted sequence of integers.

# Radix sort, pseudo-code

Let  $d$  be the maximal length of a digit in the input sequence.

RADIX-SORT( $A, d$ )

- 1 **for**  $i = 1$  **to**  $d$
- 2     use a stable sort to sort array  $A$  on digit  $i$

The name:

- ▶ Radix notation = “Stellwert-Schreibweise”
- ▶ Intuitively, radix sort means “Fächer-Sortieren”

# Running time

When sorting integers:

- ▶ Digits can be in the range  $0, 1, \dots, 9$ .
- ▶ On each digit, we spend time  $\Theta(n + 10) = \Theta(n)$
- ▶ So overall, we spend time  $\Theta(dn)$ .

More generally:

- ▶ each key can have up to  $K$  different values
- ▶ We have  $d$  different keys to use
- ▶ Then we obtain a running time of  $O(d(n + K))$ .
  - ▶ Example:  $d = O(1)$ ,  $K = O(n) \implies$  running time  $O(n)$
  - ▶ Example:  $d = O(\log n)$ ,  $K = O(n)$ , then running time  $O(n \log n)$ .

This case applies if we sort entries in a data base and the data base has no duplicates. Reason: to express  $n$  distinct elements with keys in the range  $0, \dots, K$ , we need to use  $d \geq \log_K n$  keys.

## A variant

We could also start radix sort with the largest (most significant) digit.

Even though it sounds like a minor change in the algorithm, it can lead to a disaster in performance (unless one is particularly careful). See exercises ...

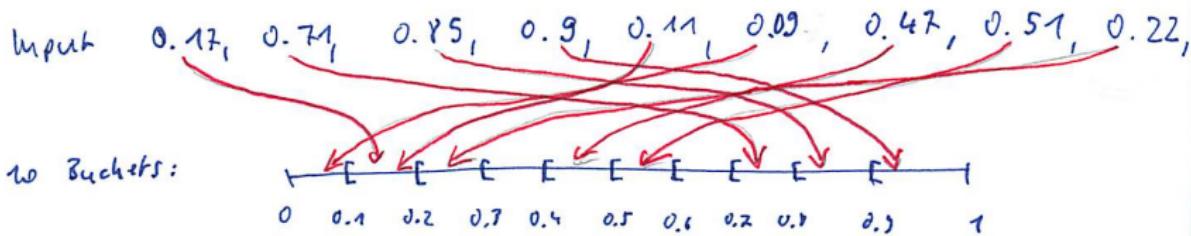
# Bucket sort / uniform sort

# Idea

- ▶ Assume that we want to sort reals that are uniformly distributed in  $[0,1]$ .
- ▶ Assume that we are given an input array of  $n$  elements.
- ▶ We split the interval  $[0,1]$  in  $n$  buckets, where bucket  $i$  is going to contain all keys from  $[i/n, (i+1)/n[$ .
- ▶ By assumption, we expect that most of the buckets only contain few elements. We sort the elements in each bucket by a naive sorting procedure, and then concatenate the buckets.

# Idea (2)

Bucket sort:



sort by  
in which  
sort:

# Pseudo-code

BUCKET-SORT( $A$ )

- 1 let  $B[0..n - 1]$  be a new array
- 2  $n = A.length$
- 3 **for**  $i = 0$  **to**  $n - 1$ 
  - 4 make  $B[i]$  an empty list
- 5 **for**  $i = 1$  **to**  $n$ 
  - 6 insert  $A[i]$  into list  $B[\lfloor nA[i] \rfloor]$
- 7 **for**  $i = 0$  **to**  $n - 1$ 
  - 8 sort list  $B[i]$  with insertion sort
- 9 concatenate the lists  $B[0], B[1], \dots, B[n - 1]$  together in order

# Running time

## Theorem 9 (Bucket sort running time)

If the keys are uniformly distributed in  $[0, 1[$ , then bucket sort has average running time  $O(n)$  and worst case running time  $O(n \log n)$ .

**Proof worst case:** In the worst case, all elements end up in the same bucket. Then we need to sort that bucket by a standard sorting algorithm (without particular assumptions), which needs  $O(n \log n)$ .

# Running time (2)

## Proof average case.

- ▶ Time for setting up the buckets and concatenating the sorted buckets is always  $O(n)$ .
- ▶ Additionally, we have to spend time  $T_i$  for sorting the elements in bucket  $i$ .
- ▶ So the expected running time on instances with  $n$  elements satisfy:

$$ET = O(n) + E \sum_{i=0}^{n-1} T_i$$

(the expectation is over the random drawing of an instance).

## Running time (3)

- ▶ Because all buckets have the same size and the input sequence is uniformly distributed, we have  $ET_i = ET_0$  for all  $i$ :

$$ET = O(n) + nET_0$$

- ▶ Now we show that even if we use insertion sort to sort the elements in bucket  $i$  we still get time  $O(n)$  in the end. With insertion sort,

$$ET = O(n) + n(EB_0)^2$$

where  $B_0$  is the number of elements in bucket 1.

# Running time (4)

- ▶ Note that  $B_0$  is binomially distributed with  $p = 1/n$ . Standard probability theory arguments then give:

$$\text{prob}(B_0 = i) = \binom{n}{i} \frac{1}{n^i} \left(1 - \frac{1}{n}\right)^{n-i} \leq \frac{n^i}{i!} \frac{1}{n^i} \leq \frac{1}{i!} \leq \left(\frac{e}{i}\right)^i$$

$$\begin{aligned} \mathbb{E}[B_0^2] &= \sum_{i \leq n} i^2 \text{prob}(B_0 = i) \leq \sum_{i \leq n} i^2 \left(\frac{e}{i}\right)^i \\ &\leq \sum_{i \leq 5} i^2 \left(\frac{e}{i}\right)^i + e^2 \sum_{i \geq 6} \left(\frac{e}{i}\right)^{i-2} \\ &\leq \mathcal{O}(1) + e^2 \sum_{i \geq 6} \left(\frac{1}{2}\right)^{i-2} = \mathcal{O}(1) \end{aligned}$$

- ▶ So we end with  $ET = O(n)$ .



# Generalization of bucket sort

Note that we might be able to use bucket sort more generally:

- ▶ Assume we know the distribution of keys (but it is not necessarily uniform).
- ▶ All we need to do is to come up with a rule to determine the boundaries of the  $n$  buckets so that the probability mass of each bucket is  $1/n$ .

# Sorting: summary and state of the art

# Overview table

	Time			Space	
	Best	Avg	Worst	Avg	Worst
Quicksort	$n \log n$	$n \log n$	$n^2$	$\log n$	$n$
Merge sort	$n \log n$	$n \log n$	$n \log n$		$(n)$
lheap sort	$n \log n$	$n \log n$	$n \log n$		1
Inversion sort	$n$	$n^2$	$n^2$		1
Selection sort	$n^2$	$n^2$	$n^2$		1
Bubble sort	$n$	$n^2$	$n^2$		1

## Overview table (2)

- ▶ Looking at the asymptotic running times does not reveal many differences.
- ▶ In practice, the size of the constants are very important.

## Overview table (3)

In practice:

- ▶ For small instances, insertion sort is used.
- ▶ For larger instances, quicksort (with many refinements) is popular.
- ▶ Often, hybrid algorithms are used that try to combine the advantages of different approaches.

Example: Tim sort, published 2002:

- ▶ Hybrid algorithm with elements from merge sort and insertion sort
- ▶ Exploits carefully the situation that pieces of the input are already sorted
- ▶ Now the standard algorithm in Python, Java SE 7, on the Android platform, and in GNU Octave.

# Other properties to look at

We have seen the following properties:

- ▶ Running times (worst, best, average)
- ▶ Space complexity

There might also be other properties that are important (but we won't discuss them):

- ▶ Online application (data arrives in a stream, and we need to sort each new data point immediately to its correct position)
- ▶ Extremely limited capacity (e.g., on limited mobile devices, on particular sensors, ... )

# High-level summary: lessons about algorithm development

## Very high level lesson:

- ▶ The same problem can have a wide variety of algorithms, which all differ in running times, space complexity, ...

## Divide and conquer:

- ▶ Often it is a good idea to use divide and conquer strategies and divide your problem in smaller subproblems.  
Example: merge sort, quick sort.
- ▶ Ideally, the subproblems should become considerably smaller in each step. If you manage to reduce the problem size by at least a constant factor, the height of the recursion tree is only logarithmic.
- ▶ Divide and conquer does not always automatically reduce the running time (Example: naive recursive digit multiplication). Sometimes you have to be clever...
- ▶ To analyze the recursion formulas for the running times, the master theorem is often a good tool.

## Use of randomized algorithms:

- ▶ You need to select a “typical” element, but you don't know how to achieve it in a deterministic way. In this situation consider drawing it randomly.
- ▶ If the “good event” that you pick a typical element happens with high enough probability, then your randomized algorithm is going to work fine.
- ▶ Randomized algorithms are often very simple, but their theoretical analysis can be very difficult.

# Searching

# Search Trees

# AVL trees?

# (a,b)-Trees

# Red-Black-Trees